THE

WAY TO DATA QUALITY AND DATA OBSERVABILITY

First Version, 2026

The DataOps Way to Data Quality and Data Observability

A Practitioner's Guide to Building Data Teams That Deliver

Christopher Bergh, Charles Bloche, and Gil Benghiat

Copyright DataKitchen, Inc. 2026

Table of Contents

Why Data Quality Matters **5**

Data Quality Testing is at the Core of Four Critical Data Team Processes 6
The Mystery Box Full of Data Errors 24
Data Quality vs Data Observability: The Pets and Cattle of Your Data Estate 39
Stop Paying the Data Quality Tax 48
How I Broke Our SLA and Delighted Our Customer 55
The Economics of Data Quality: When to Test, How Much to Spend 57
When Timing Goes Wrong: How Latency Issues Cascade 63
We Got Roasted on Reddit for Asking Why Data Engineers Don't Test 67
We Just Eyeball Row Counts and Pray 77
A Data Usability Crisis is Costing Your Company 95

Culture and Organization **104**

The Seven Deadly Sins of Data Quality 105
The Data Errors Shame Game 114
Is Your Team in Denial about Data Quality? 121
The Ostrich Problem: When Your Data Team Stops Caring After Deployment 128
The Data Quality Revolution Starts with You 135
The Five Organizational Sins of Data Teams 141
You're Thinking About Data Products All Wrong 148
Process Guardianship: The Most Valuable Work You Are Probably Not Doing 156
How Data Quality Leaders Can Gain Influence and Avoid the Tragedy of the Commons 165
How Do I Get My Data Team More Efficient? 173

Building Your Testing Strategy **181**

Shift-Left, Shift-Down: How to Take Control of Data Quality 182
Scaling Data Reliability: The Definitive Guide to Test Coverage 189
Data Quality Test Coverage in a Medallion Data Architecture 199
The Race for Data Quality in a Medallion Architecture 209
Building Data Quality Into dbt: A Complete Guide 220
Data Contracts: Shifting the Burden of Quality to the Source 234
Building and Scaling a Data Quality Practice: The First Year 248
Impact Dimensions Cure A Data Quality Blind Spot 255
Your Best Data Quality Rules Live in Someone Else's Head 265
The Blueprint or the Fire Alarm Panel: Why Data Test Coverage Beats Data Lineage 274
The Data Contract Is Your Receipt 288
The Four Points in Your Medallion Architecture Where Data Testing Really Matters 298
Testing Every Layer of the Medallion Architecture, Now with OneLake at the Center 316

Monitoring and Observability 326
Data Observability: What to Monitor and Why It Matters 327
The Six Types of Data Quality Dashboards 335
Data Quality Alerting: Avoiding Alert Fatigue 350
Why Your Data Quality Dashboard Isn't Working And What to Do About It 356
Data Production Tripwires in Databricks: Stop Bad Data Before It Reaches Production 366

Implementation and Case Studies 376
Data Testing Anti-Patterns: Why Your Tests Are Failing You 377
Three Data Quality Disasters and How They Were Averted 385
When Not to Test: Setting Reasonable Bounds on Data Quality Effort 392
I Wrote Data Quality Tests for 3 Million NYC Taxi Rides in Five Minutes 397
Data Quality's Most Wanted: The 2026 Case Files 407
Why Testing Commercial Pharma Data Is Harder Than Anyone Tells You 414
IQVIA, specialty pharmacy, and your own files: the commercial data most likely to be wrong 421
The Illusion of Clean Data: The Pharma Identifiers Generic Tools Never Check 428

AI and Data Quality 437
Sure, Go Ahead And Feed That Data To The LLM ... What Could Possibly Go Wrong? 438
The Equation For AI Success: DT + DX + CTX = 10x 450
DataOps + FITT + Data Testing = 10x Data Engineering Productivity with AI 465
Context As Infrastructure: The Six-Category Framework for AI-Ready Data Analysis 485
Stop Burning Tokens: Data Observability and Quality as Your AI's System of Record 505
Put Your AI At The Center Of Your Data Quality And Observability Decision-Making Process 508
Agents to Fix Data Quality Automatically With Control: A Design Pattern 516
The Seven Deadly Sins of AI-Generated Data Pipelines 523

Reference and Tooling 534
A Practical Guide to Selecting Data Quality Tools 535
The Data Quality and Observability Tools Landscape 542
The 2026 Data Quality and Data Observability Commercial Software Landscape 551
The 2026 Open Source Data Profiling Software Landscape 556
The Consultant's Growth Playbook: Accelerating Client Acquisition with DataKitchen Open Source TestGen 562
$1 Billion in Data Observability VC Investment: This Is Not Going to End Well 574
There are a lot of freaking data quality and data observability vendors 587
There Is No Microsoft Data Quality Tool in 2026. There Are Nineteen. 592
Where to Actually Start with DataOps in 2026 613

Introduction

Data quality is not a technology problem. It never has been. The data industry has spent years chasing the next tool, platform, or vendor promise, yet data errors still dominate the daily work of data engineers. Teams keep blaming each other when pipelines break, and business stakeholders quietly distrust the numbers they see. After working with data teams across many industries, we believe the real barriers to data quality are cultural, organizational, and process-related. Teams get stuck in reactive habits, scrambling to fix errors after they happen, pointing fingers across silos, and treating data quality as someone else's issue. The articles in this book aim to honestly identify these patterns and offer a better way.

That better way is DataOps, which applies Agile, DevOps, and lean manufacturing ideas to data and analytics work. DataOps isn't a product you can buy; it's a set of habits and disciplines that, when followed consistently, turn data quality from a constant crisis into a managed, measurable, and ever-improving part of your data environment. The chapters here cover everything from the organizational issues that hurt data quality efforts, like warring tribes, the data shame game, and the ostrich problem, to technical practices that give teams real control: shift-left testing, systematic test coverage, continuous monitoring, and using data observability to spot and fix failures before they affect customers. Together, these chapters offer data teams a clear, honest, and practical guide to doing this work well.

Now, why did we open-source our tools? DataKitchen builds software like DataOps TestGen for data quality testing and profiling, and DataOps Observability for pipeline monitoring, but we've never believed that software alone solves the problem. The main reason data teams struggle with data quality isn't a lack of technology; it's a lack of knowledge, vocabulary, and process frameworks to use that technology well. By releasing TestGen and DataOps Observability as free, open-source products, we're removing the financial barrier that has kept smaller teams, independent consultants, and budget-conscious organizations from accessing enterprise-grade data quality tools. We want every data team, regardless of budget, to have tools that automatically generate tests, continuously monitor tables, and spot anomalies before they cause problems.

We also open-sourced our tools because we believe trust comes from transparency. The data quality and observability market is full of vendors who hide their pricing, lock customers into proprietary systems, and charge per-table fees that make full coverage too expensive. We believe that's wrong. Data quality testing should be available to everyone, not limited. Observability should cover every pipeline and dataset, not just the ones a team can afford. This book and these tools share the same belief: better data quality is possible for every organization and worth working for. We hope you find not just ideas that resonate, but practical steps you can start using today.

This book was updated on September 23, 2026.

SECTION 1

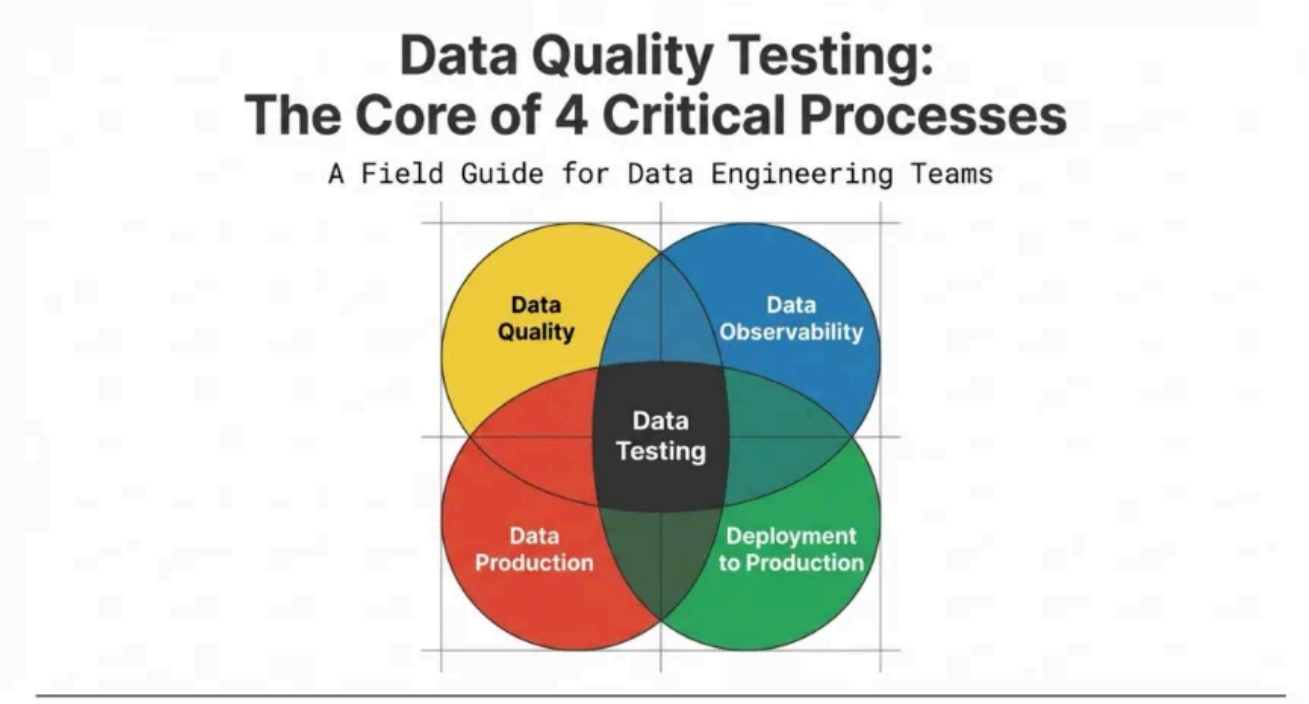
Why Data Quality Matters

1. Data Quality Testing is at the Core of Four Critical Data Team Processes
2. The Mystery Box Full of Data Errors
3. Data Quality vs Data Observability: The Pets and Cattle of Your Data Estate
4. Stop Paying the Data Quality Tax
5. How I Broke Our SLA and Delighted Our Customer
6. The Economics of Data Quality: When to Test, How Much to Spend
7. When Timing Goes Wrong: How Latency Issues Cascade
8. We Got Roasted on Reddit for Asking Why Data Engineers Don't Test
9. We Just Eyeball Row Counts and Pray
10. A Data Usability Crisis is Costing Your Company

WHY DATA QUALITY MATTERS

Data Quality Testing is at the Core of Four Critical Data Team Processes

Read the original article at: https://datakitchen.io/blog/data-quality-testing-is-at-the-core-of-four-critical-data-team-processes/

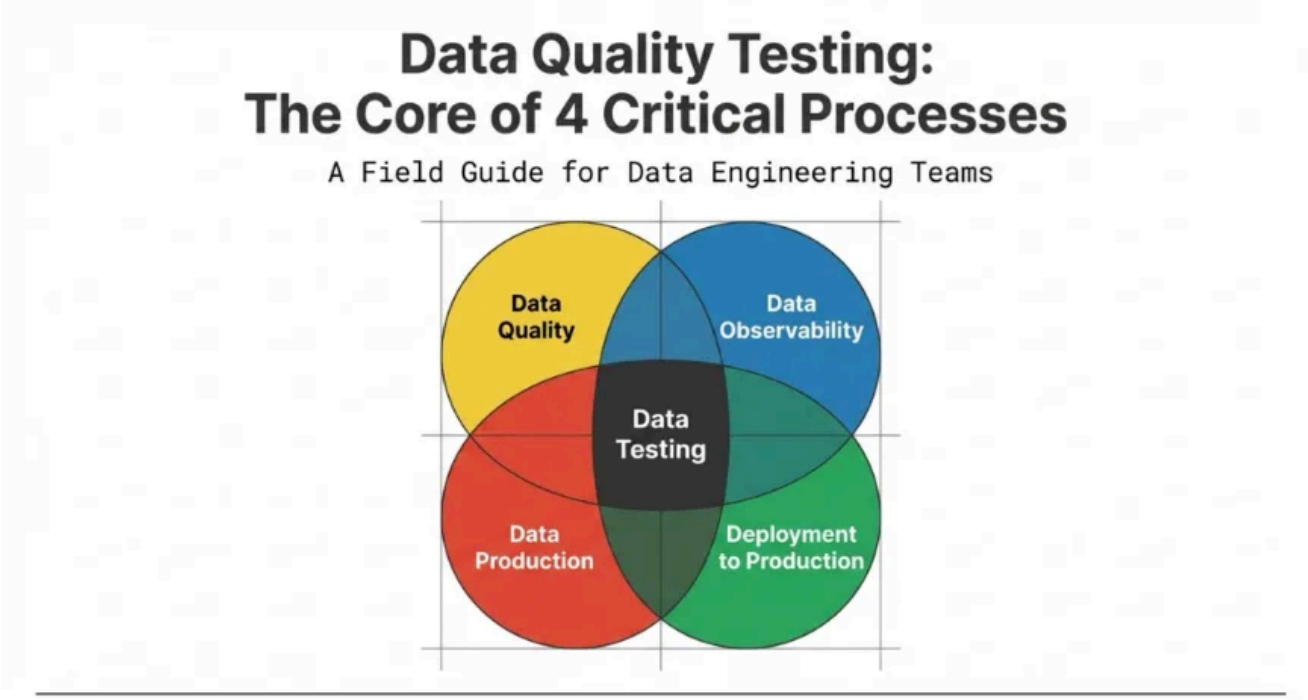


Every data and analytics team juggles multiple responsibilities. You are expected to ensure your data is accurate and fit for purpose. You need to prevent problematic data from entering your data environment. You must stop your data pipelines from delivering embarrassing errors to your stakeholders. And you have to deploy new code and data sets into production without breaking what already works. These four activities might seem distinct at first glance, but they share a fundamental characteristic: they all depend on robust data quality testing to succeed.

The challenge is that most organizations treat these activities as separate concerns, with different tools, different teams, and different approaches. This siloed thinking obscures a crucial insight. Data quality testing is not just one activity among many. It is the foundational capability that enables all four of these processes. When you understand this connection, you can build a more coherent and effective approach to managing your data estate.

The Four Processes That Define Data Team Operations

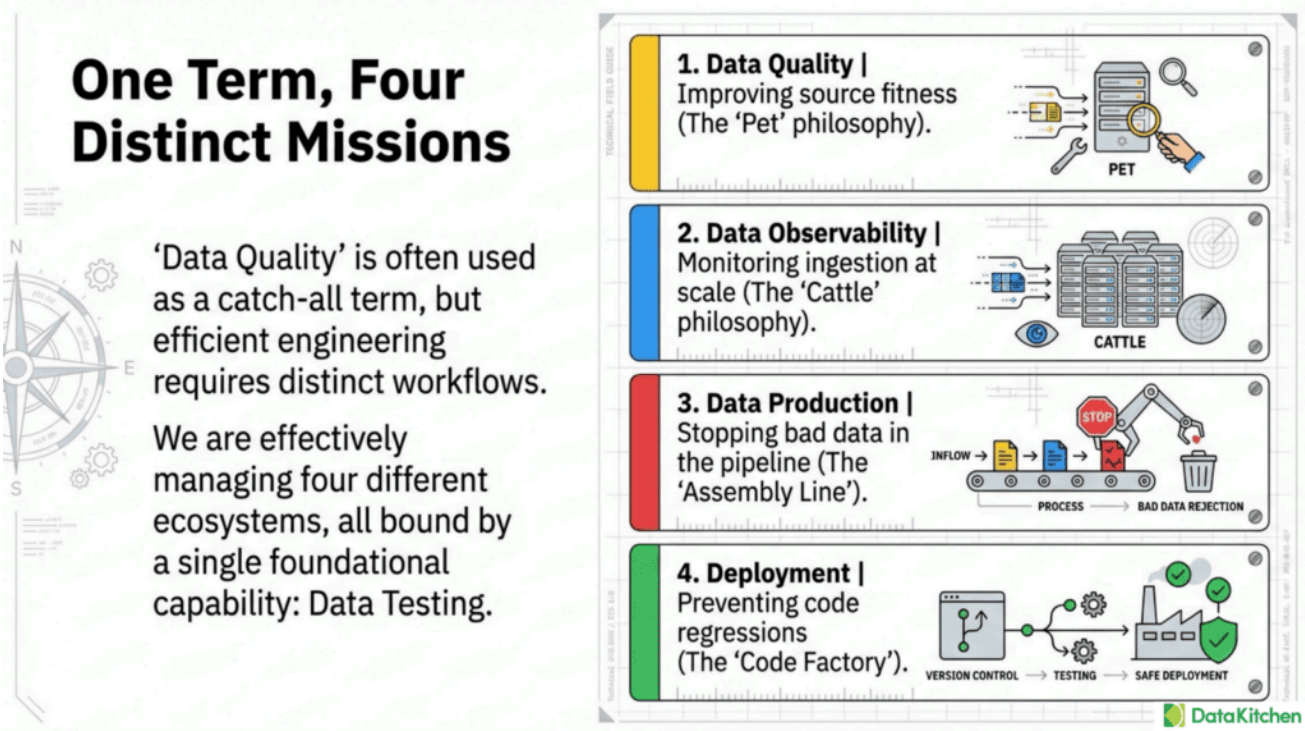
Before diving into how testing connects these activities, let us establish what each process entails and why it matters to your organization.



The first process is Data Quality, which focuses on improving the quality of your data sources and ensuring they are fit for purpose. This is the most traditional understanding of data quality work. Teams engaged in this process spend their time examining specific tables and datasets, running detailed checks against data quality dimensions, building dashboards to track quality scores, and working to influence upstream data sources to fix problems at their origin.

The second process is Data Observability, which has emerged as a critical discipline for teams managing large numbers of tables and datasets. The core problem here is avoiding blame for obvious data problems that slip through unnoticed. When you have hundreds or thousands of tables, you cannot manually inspect each one. Data observability solutions poll your databases looking for anomalies in freshness, volume, schema changes, and statistical patterns. The goal is to identify data ingestion errors and reduce data downtime.

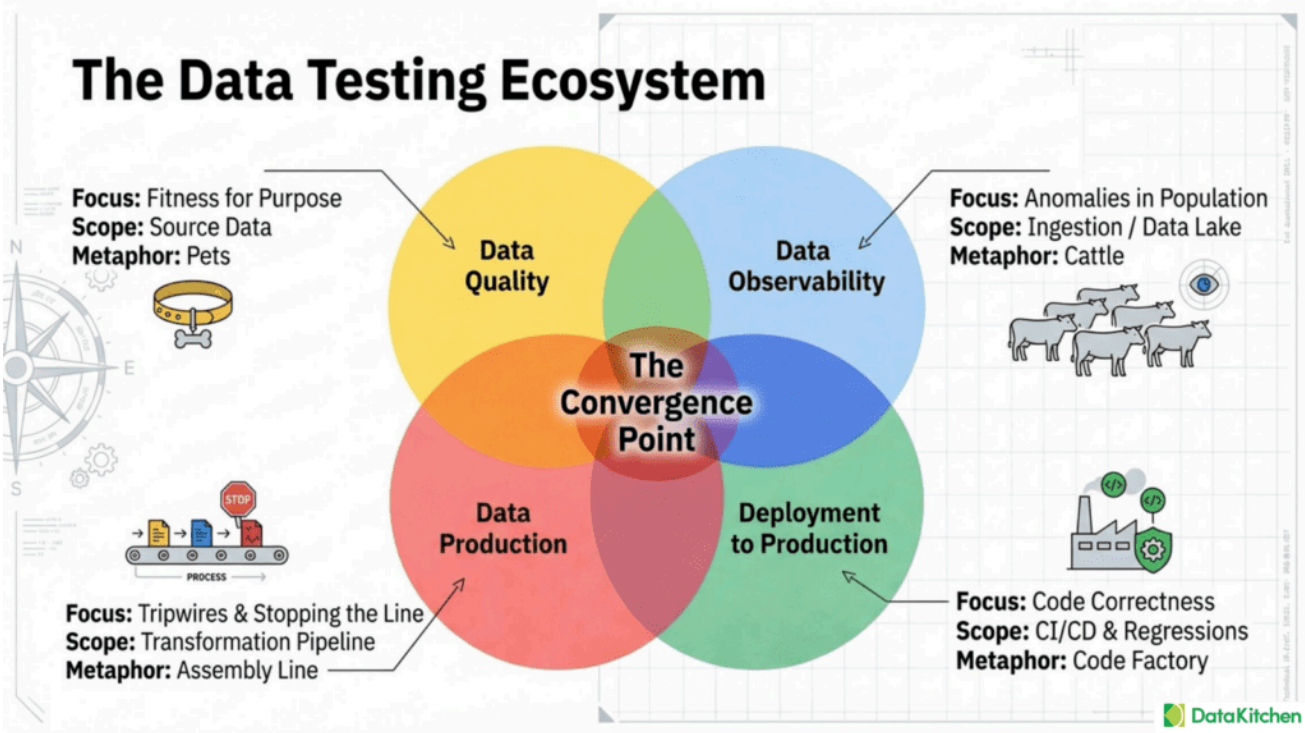
The third process is Data Production, which we might also call Data Journey Quality. This goes beyond simply observing data at rest to monitoring the entire journey that data takes through your production systems. The fundamental question here is how to stop errors in data and deliverables before they reach production. This requires placing tests and monitors at every step of the data production process, with tripwires that can halt processing when problems are detected.



The fourth process is Deployment to Production, specifically the regression and impact testing of new code. Your team of data engineers, scientists, and analysts is constantly writing and modifying code. Before that code reaches production, you need to ensure it has been adequately tested. This means going beyond simple unit tests to include full regression testing and functional testing in development environments during CI/CD.

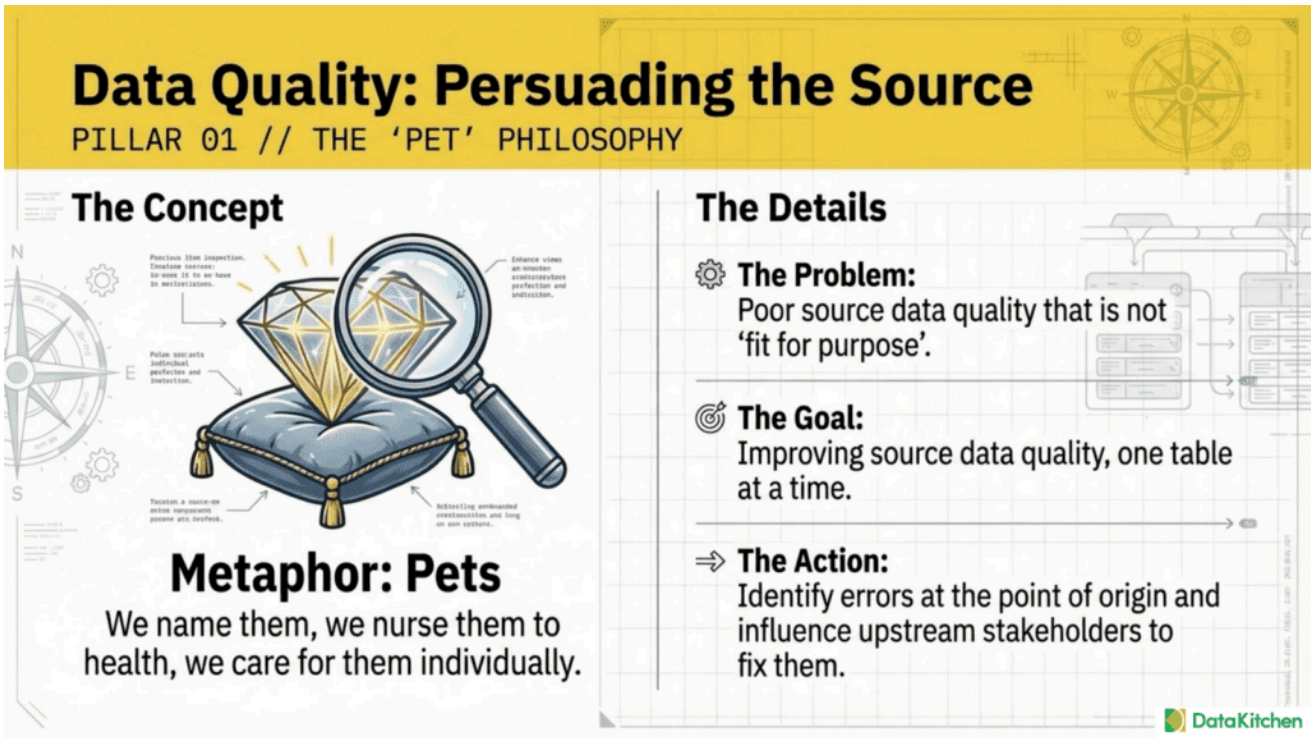
The Data Quality Process: Treating Your Critical Data Like Pets

The metaphor of "pets versus cattle" has become common in infrastructure discussions, but it applies equally well to data quality work. When you treat data like pets, you give individual datasets names, you care for them individually, and you invest significant effort in keeping each one healthy. This approach makes sense for your most critical data assets.



Data quality work in this mode involves persuading the source to improve its data quality. You are not just detecting problems; you are building the case for why upstream systems and processes need to change. This requires finding errors at the point of origin, as close to where data enters your systems as possible. The work involves creating detailed data quality dashboards that track specific tests across data quality dimensions like completeness, accuracy, consistency, validity, timeliness, and uniqueness.

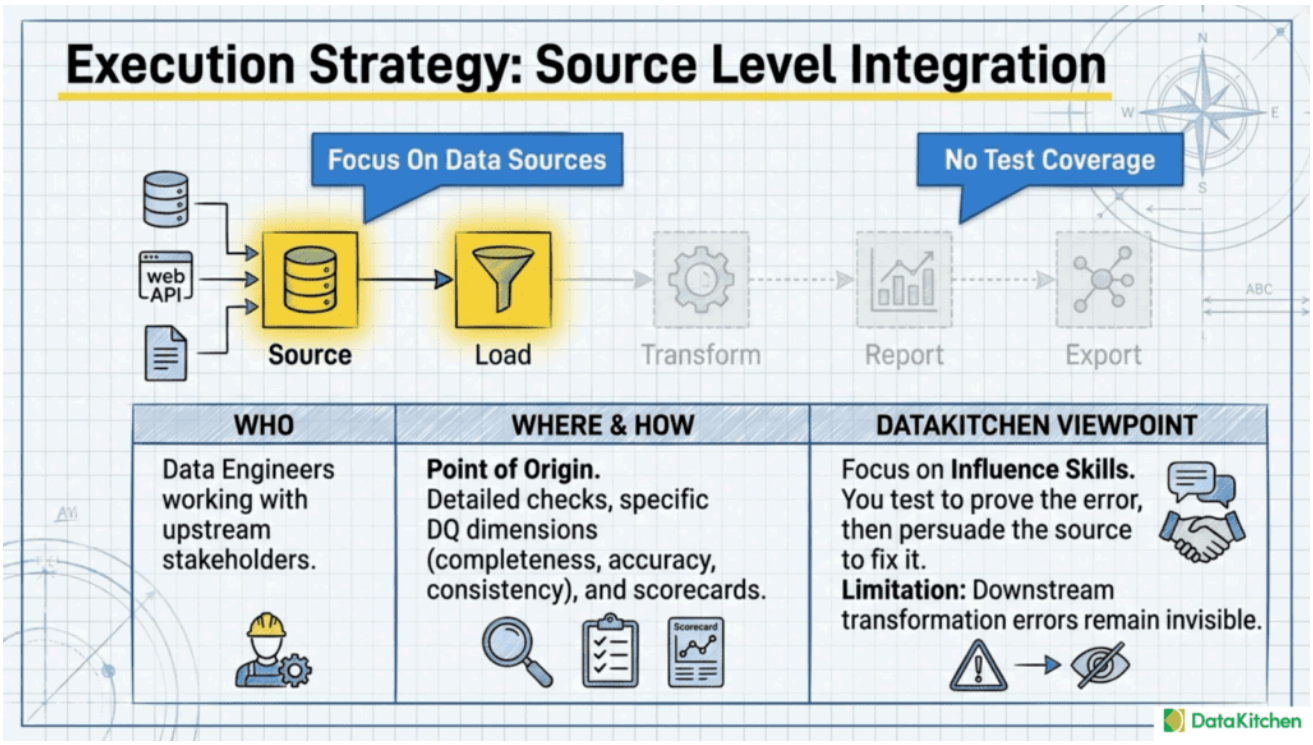
The testing approach for traditional data quality work tends to be intensive and specific. You write lots of tests for each table, covering every aspect of the data that matters to your business. You score and track results over time. You use those results to influence your data sources, whether those are internal systems, external vendors, or manual data entry processes.



This focused approach has real benefits. It keeps things relatively simple at the data level and allows you to build deep expertise in specific datasets. However, it also has significant limitations. When you focus exclusively on source data quality, errors that occur in downstream tools and deliverables remain invisible to you. Your customers might receive incorrect analytic results even though your source data quality scores look excellent.

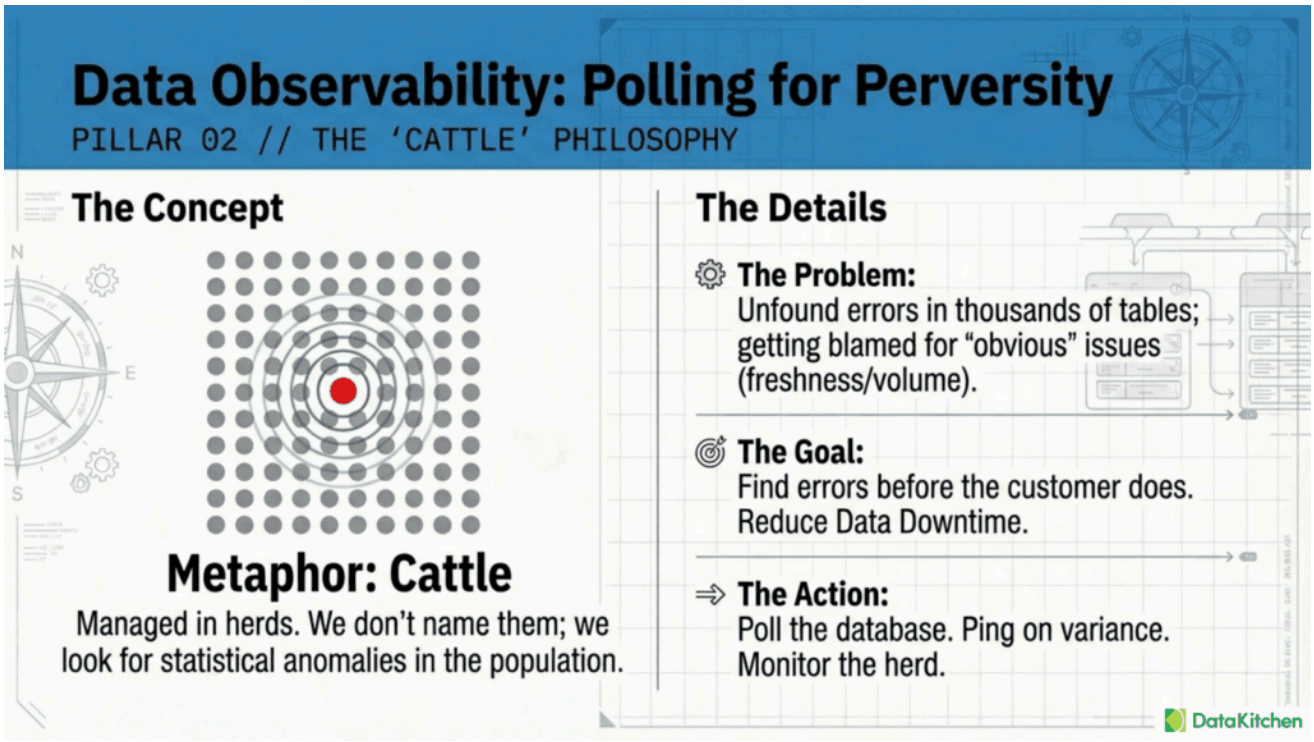
The Data Observability Process: Managing Data Scaled Like Cattle

When you have thousands of tables spread across multiple databases and data platforms, the pet approach simply does not scale. You cannot write hundreds of specific tests for every table. You cannot manually review quality dashboards for each dataset. You need a fundamentally different approach.



Data observability treats data more like cattle. Instead of naming and caring for individual animals, ranchers manage herds at scale using statistical sampling and pattern recognition. Similarly, data observability solutions monitor your databases and alert you to errors. They look for variance across large populations of tables, checking for things like unexpected changes in row counts, unusual freshness delays, schema modifications, and statistical drift that might indicate data problems.

The target audience for data observability is typically the data ingestion engineer responsible for reliably getting data into the platform. These engineers need to find errors in large databases before customers do. The primary goal is to reduce data downtime, which is the period when data is unavailable, inaccurate, or otherwise unusable.

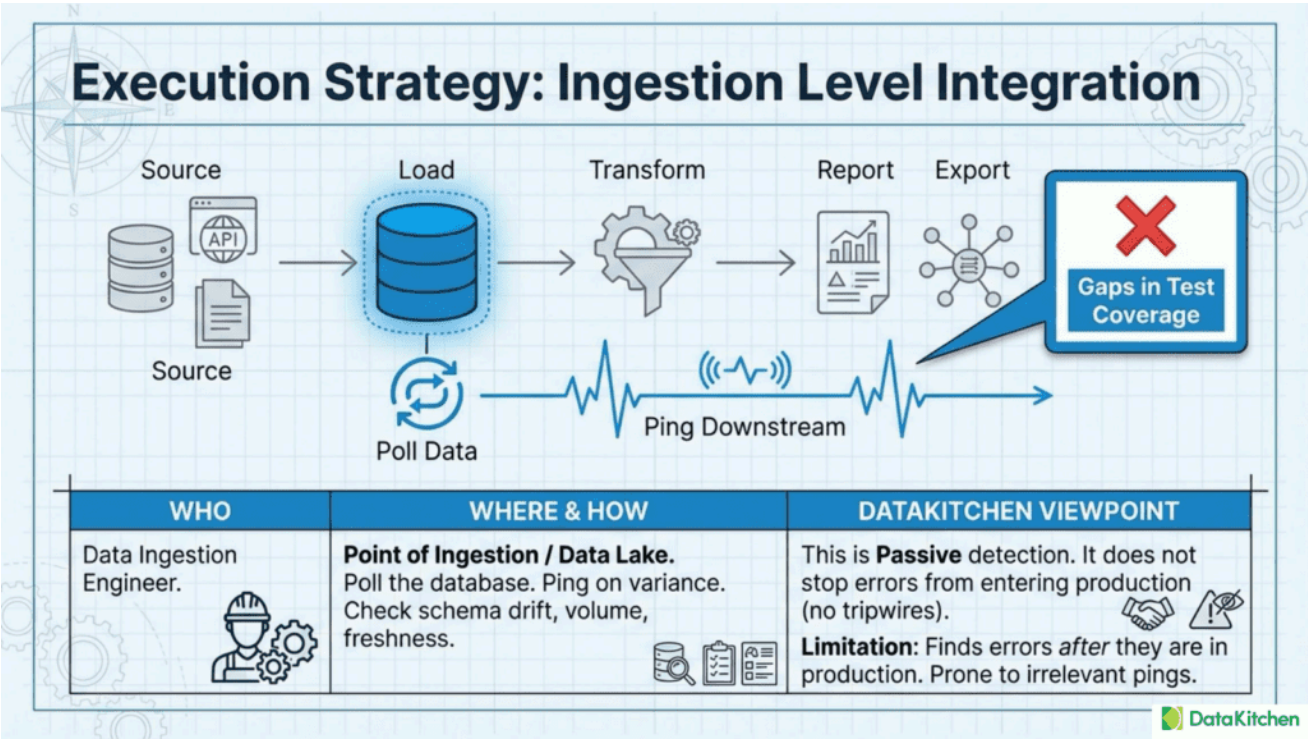


This polling approach enables coverage across far more tables than would be possible with hand-crafted tests. It also provides visibility into data lineage, helping you understand how problems propagate through your systems. However, the observability approach has its own limitations. It finds data errors after they have already gotten into production systems. The automated anomaly detection can generate irrelevant pings that waste time and erode trust. And the anomaly testing, while broad, is often shallow compared to what you would achieve with specific hand-crafted tests for critical tables.

Perhaps most importantly, data observability alone does not prevent errors from reaching production. It detects problems after the fact rather than preventing them from reaching your stakeholders.

The Data Production Process: Protecting Production with Data Journey Quality

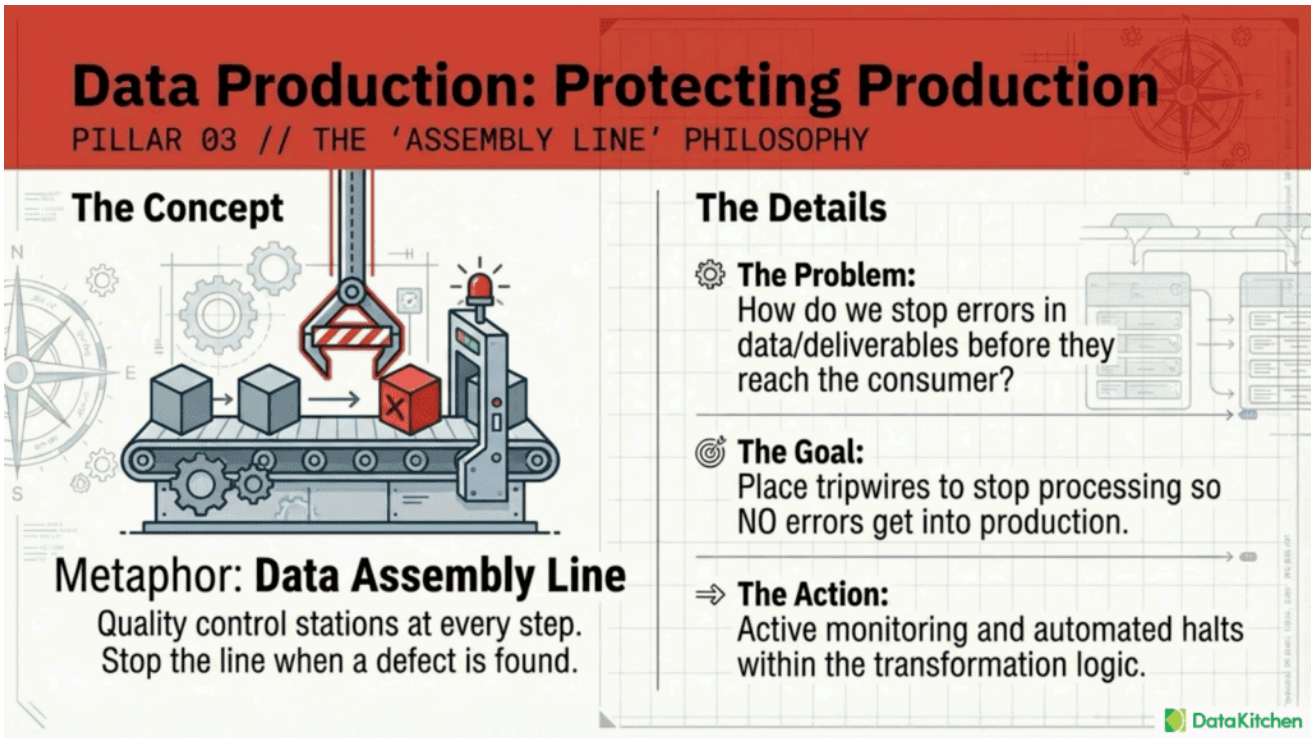
The third process addresses a fundamental gap in the first two approaches. Traditional data quality focuses on sources, and data observability polls data at rest. Neither adequately addresses what happens during the data production process itself. This is the data journey from raw inputs through transformation, enrichment, and aggregation to final deliverables.



Data Journey Quality operates on a different metaphor: the assembly line. In manufacturing, quality control does not just inspect raw materials at the loading dock or finished goods in the warehouse. Inspections happen at every step of the production process, with the ability to 'stop the line' when problems are detected. This prevents defective work from moving forward and wasting resources on products that will ultimately fail.

The core question for data production is straightforward but challenging: how do you stop errors in data and deliverables before they reach production? This requires placing tests and monitors within the data production processing steps themselves, not as an afterthought. You need either a test-aware orchestrator that can respond to test failures or a dedicated data journey observability tool that integrates with your existing infrastructure.

The goal is full test coverage across your production process, with the ability to find all errors during data production and place tripwires that stop processing. When an error is detected, the system should halt before the problem propagates further downstream or reaches customer-facing systems.

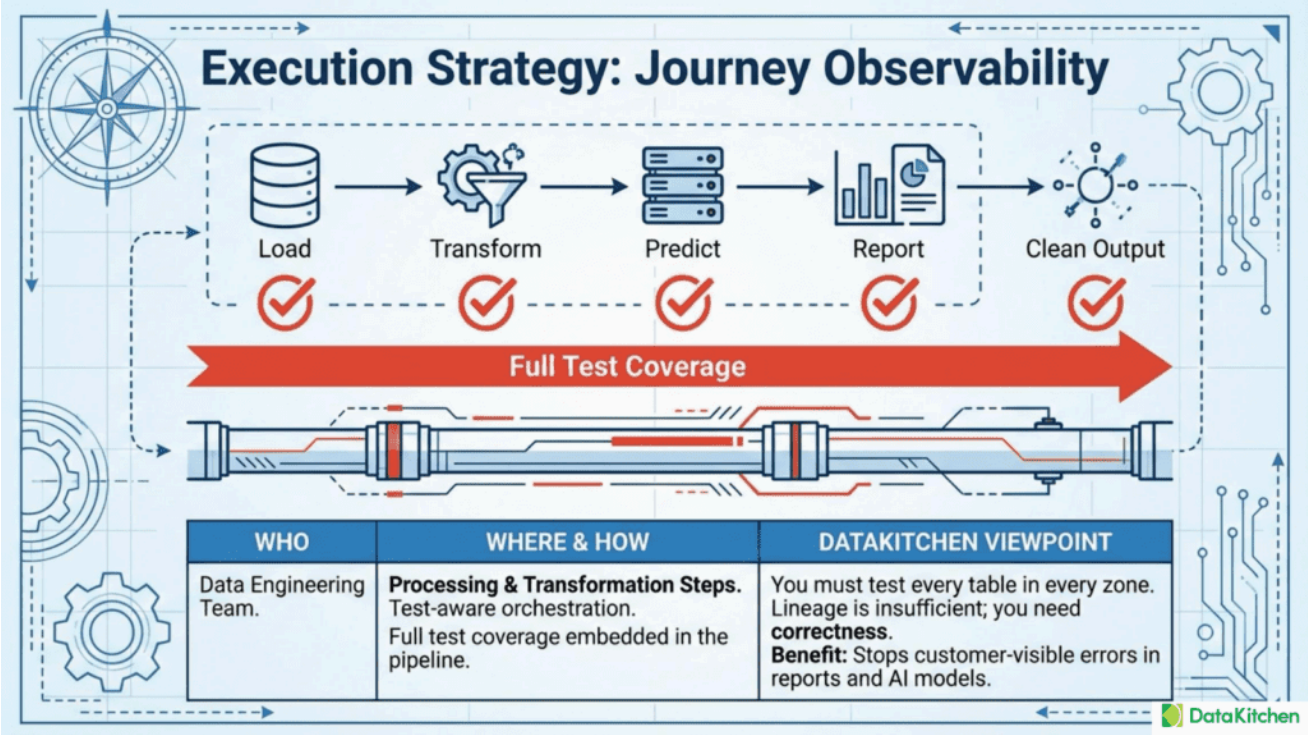


This approach requires testing every table in every zone of your data platform. It means monitoring every tool involved in your data production process and integrating them into a single unified view. Data lineage alone is insufficient because it tells you how data flows but not whether that data is correct at each step.

The testing requirements for data production are substantial. You need both data integration and tool integration, which makes implementation more complex than either traditional data quality or data observability alone. But the benefit is significant: you can actually stop customer-visible errors in data, reports, and models before they cause damage.

The Deployment to Production Process: Regression and Impact Testing

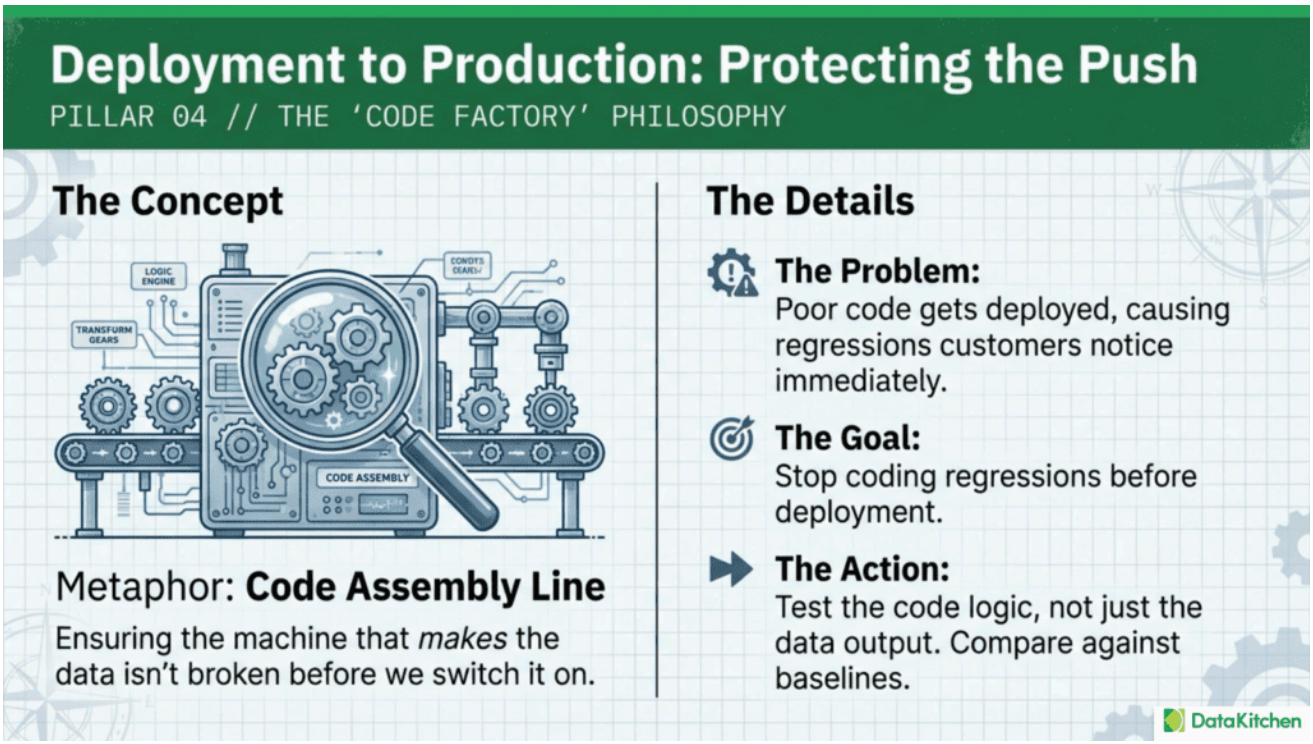
The fourth process shifts focus from data to code. Your data engineering team is constantly writing new transformations, building new pipelines, updating existing logic, and deploying changes to production. Each of these changes carries risk. A modification that works perfectly in development might break something in production. A new feature might introduce a regression in existing functionality.



The metaphor here is another assembly line, but this one produces code rather than data. Just as the data production assembly line needs quality gates, the code production process needs its own checkpoints to ensure that poor code does not reach production, where customers will notice the problems.

The traditional approach to code testing relies heavily on unit tests and manual business reviews. Unit tests verify that individual functions work correctly in isolation, and business reviews ensure that outputs look reasonable to human experts. These approaches are necessary but not sufficient. Unit tests cannot catch problems that only emerge when components interact in production environments. Manual reviews cannot scale to cover every possible scenario and are prone to human error and fatigue.

What data teams need is a fully automated system with adequate test coverage. This includes regression testing that compares the outputs of new code against baseline results from the existing production system. It also includes impact testing that identifies which downstream assets might be affected by a change. And it means integrating these tests into your CI/CD pipeline so they run automatically before any code can be deployed. Great test data is key. Some teams say: today's code, yesterday's data' for deployment testing.



The testing limitations here involve practical concerns: you need appropriate test data that reflects production scenarios, tools to automate comparison, and CI/CD integration that enforces testing requirements without creating bottlenecks for your development team.

The benefit of getting this right is substantial. Every regression error that slips into production costs far more to fix than one caught during development. The commonly cited figure is that production bugs cost one hundred times more than bugs caught earlier in the development process. This multiplier reflects not only the technical cost of fixing the bug but also the customer impact, the loss of trust, and the emergency response required by production incidents.

The Common Thread: Testing at the Core

When you examine these four processes carefully, a pattern emerges. Each one depends fundamentally on data quality testing, but each applies testing in a different context and for a different purpose.

	Data Quality	Data Observability	Data Production	Deployment to Production
Metaphor	Pets	Cattle	Assembly Line	Code Delivery Line
Primary Goal	Source improvement	Reduce data downtime	Prevent production errors	Prevent code regressions
Action	Persuading the Source	Polling/Pinging	Protecting Production	Protecting the Push
Where	Point of Origin	Data Ingestion	Every step (Tripwires)	Development / CI-CD
Benefit	Improves source data quality	Finds obvious data load issues	Stops customer-visible errors	Reduces fix costs by 100x

Testing techniques overlap significantly across these domains. A test that checks for null values in a critical column might serve data quality in one context, data observability in another, and deployment validation in a third. The difference lies not in the test itself but in when it runs, what triggers it, and what happens when it fails.

This insight has practical implications for how you structure your testing capabilities. Rather than building separate testing infrastructure for each of these four processes, you can create a unified testing foundation that serves them all. A comprehensive test suite can feed data quality dashboards for source improvement work while simultaneously providing the tripwires needed for data production protection and the regression baselines needed for deployment validation.

Bridging the Gaps Between Processes

Understanding how these four processes relate reveals several gaps that many organizations struggle with.

The gap between data quality and data observability is a matter of depth versus breadth. Data quality provides deep testing for critical tables, but cannot scale to cover everything. Data observability provides broad coverage but often misses problems that only detailed tests would catch. Organizations need both approaches, with clear criteria for which tables get the pet treatment and which are managed as cattle.

	Data Quality	Data Observability	Data Production	Deployment to Production
Metaphor	Pets	Cattle	Assembly Line	Code Delivery Line
Primary Goal	Source improvement	Reduce data downtime	Prevent production errors	Prevent code regressions
Action	Persuading the Source	Polling/Pinging	Protecting Production	Protecting the Push
Where	Point of Origin	Data Ingestion	Every step (Tripwires)	Development / CI-CD
Benefit	Improves source data quality	Finds obvious data load issues	Stops customer-visible errors	Reduces fix costs by 100x

The gap between data observability and data production lies between prevention and detection. Observability tells you when something has gone wrong. Data journey observability with tripwires can prevent errors from propagating. Organizations that rely solely on after-the-fact detection will always be playing catch-up, fixing problems after customers have already been affected.

The gap between data production and deployment testing is about runtime versus development time. Data production monitoring catches problems during actual production runs. Deployment testing catches problems before code even reaches production. Both are necessary because some problems manifest only at scale in production environments, while others can and should be caught earlier in development.

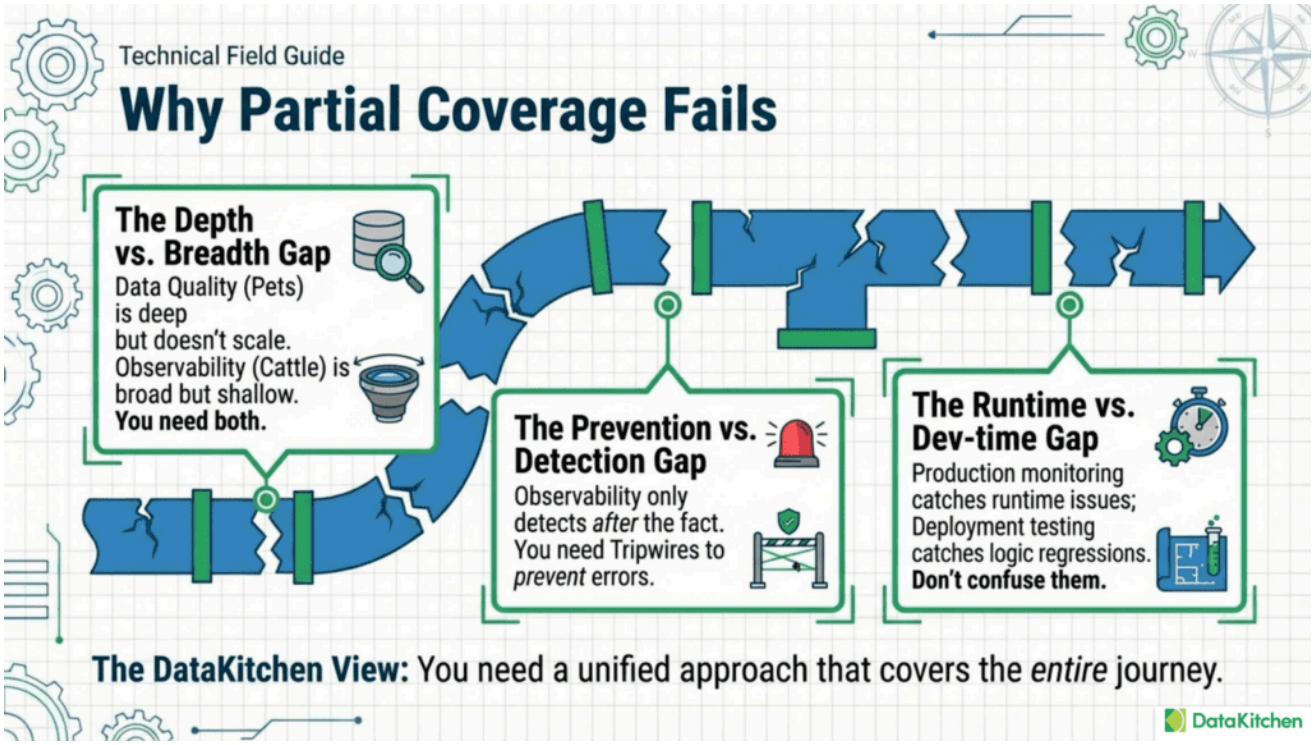
Building a Comprehensive Testing Strategy

Given that testing underlies all four of these critical processes, how should data teams approach building their testing capabilities?

The first step is recognizing that testing is not a one-time project but an ongoing capability that requires investment and maintenance. Your test suite will need to evolve as your data assets, pipelines, and business requirements change. Building tests is not enough; you also need processes for reviewing, updating, and retiring tests as circumstances warrant.

The second step is prioritizing test coverage based on business criticality. Not every table needs the same level of testing attention. Your most critical datasets, the ones that drive key business decisions or feed customer-facing applications, deserve the intensive pet treatment. Less critical tables can be

managed with a lighter-touch cattle-style observability approach. The key is making this prioritization explicit rather than letting it happen by accident.



The third step is to integrate testing into your production processes rather than treat it as a separate activity. Tests should run automatically as part of your data pipelines, with clear tripwires that halt processing when critical thresholds are breached. This requires both technical integration with your orchestration tools and organizational agreement on which severity of test failure should stop production.

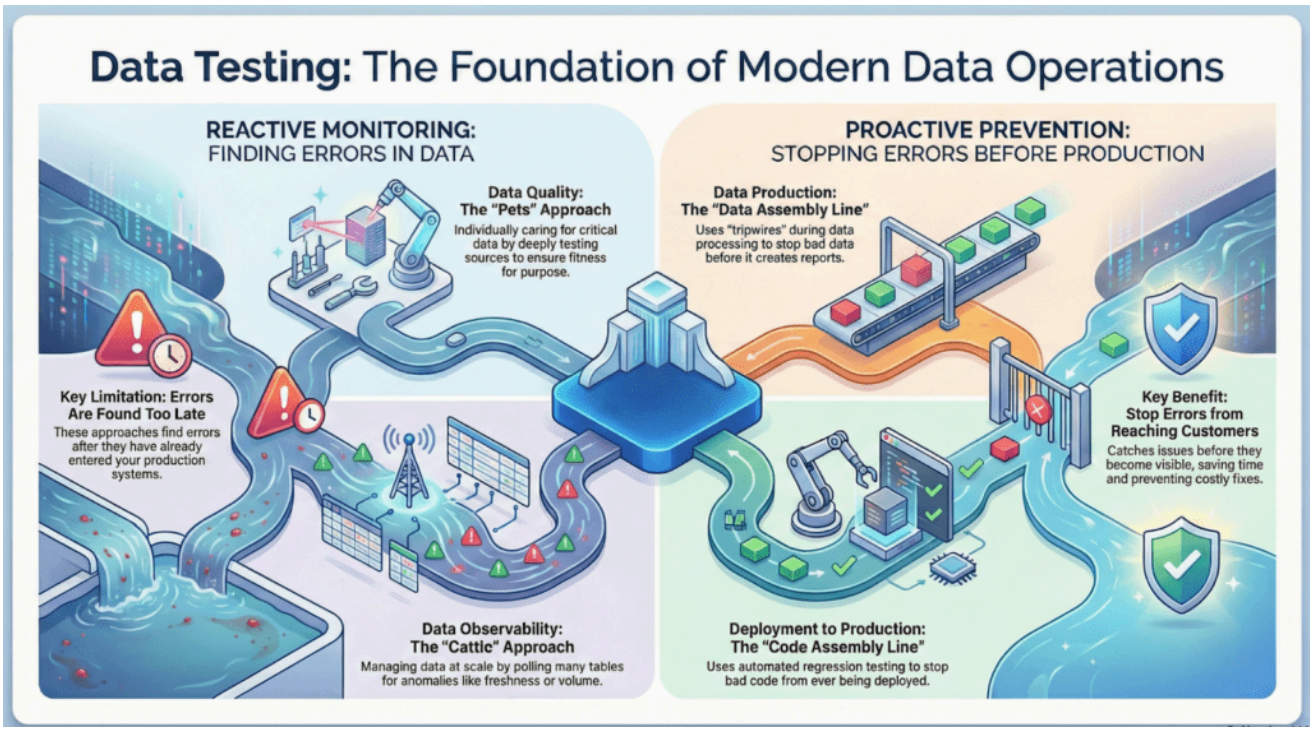
The fourth step is connecting testing to your deployment workflow. Every code change should trigger appropriate regression and impact tests before it can reach production. This requires investment in test automation and CI/CD integration, but the payoff in prevented production incidents is substantial.

The fifth step is to build feedback loops that use test results to drive improvement. Test failures should not just trigger alerts; they should inform root-cause analysis and drive permanent fixes. Data quality scores should drive conversations with upstream data sources. Deployment test failures should prompt an investigation into why the problem was not caught earlier in development.

The Cost of Inadequate Testing

Organizations that underinvest in data quality testing pay the price across all four of these processes. The 1:10:100 rule applies: find problems early to avoid paying the high price of failure.

Without adequate source data quality testing, problems at the origin propagate through your entire data estate, multiplying their impact at every step. You might clean up the symptoms downstream, but the root cause continues generating new problems.



Without adequate data observability testing, embarrassing errors slip through to your customers. You get blamed for obvious problems that automated checks should have caught. Your credibility with business stakeholders erodes over time.

Without adequate data production testing, errors in your transformation logic or pipeline configurations reach production systems. You might not even know about the problems until a customer reports incorrect numbers in a dashboard or a model starts making bad predictions.

Without adequate deployment testing, code changes introduce regressions that break things that used to work. Your team spends increasing amounts of time on emergency fixes and production support, leaving less time for the new development work that the business is waiting for.

The cumulative effect of these testing gaps is a data platform that becomes increasingly unreliable and a data team that becomes increasingly reactive. Instead of building new capabilities and delivering business value, you spend your time fighting fires and apologizing for problems.

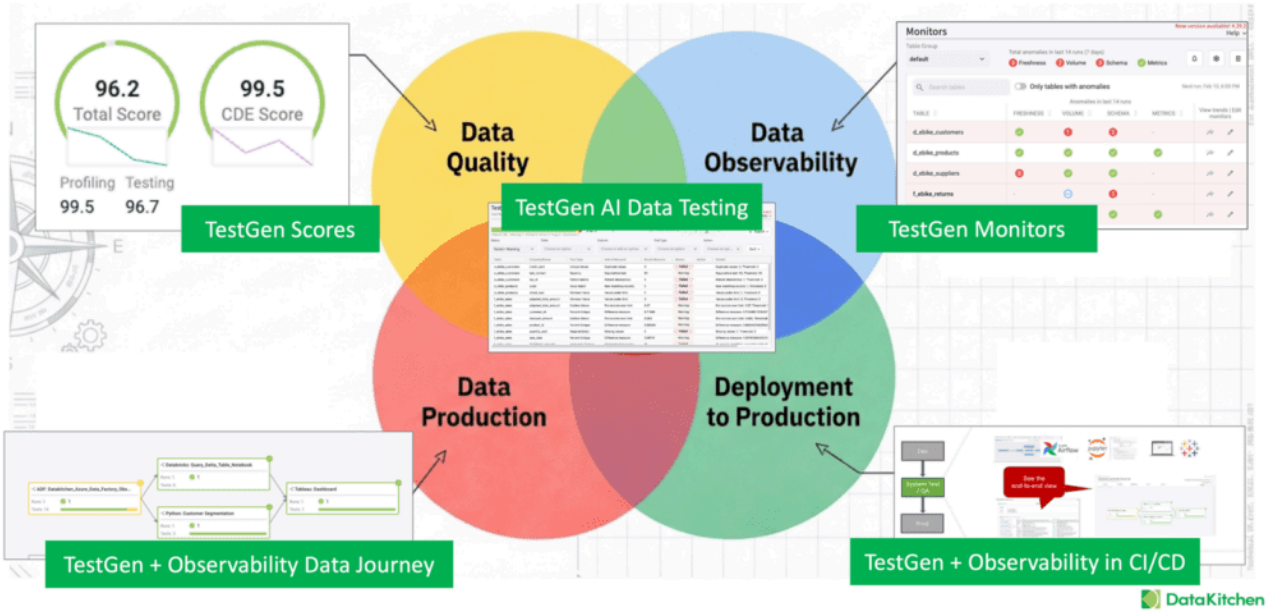
The Testing Challenge and a Practical Solution

By now, the central role of data quality testing should be clear. Whether you are improving source data quality, implementing data observability across hundreds of tables, building tripwires into your data production pipelines, or validating code deployments, you need tests. Lots of tests. Tests that cover your data comprehensively and catch problems before they reach your customers.

Here is the problem: achieving adequate test coverage across all four of these use cases is extraordinarily time-consuming. Writing tests manually does not scale. A single table might need dozens of tests to cover all relevant data quality dimensions, detect anomalies, serve as a production tripwire, and provide a regression baseline. Multiply that by hundreds or thousands of tables, and you are looking at months of work just to configure or code the tests you need. Most teams never get there. They write tests for a handful of critical tables and hope for the best with everything else.

The remaining twenty percent, the tests that encode specific business rules and domain knowledge unique to your organization, still require human input. But starting with a solid foundation of automatically generated tests means your team can focus their limited time on those high-value custom tests rather than writing boilerplate checks that a tool can handle. This is how you achieve the comprehensive test coverage that makes all four of these critical processes work effectively. This is how you move from reactive firefighting to proactive data quality management.

DataKitchen Open-Source Tools



The Mystery Box Full of Data Errors

Read the original article at: https://datakitchen.io/white-papers/data-quality-the-dataops-way/

Why a DataOps Approach to Data Quality: Data Quality Challenges

Data quality is more critical than ever in the age of artificial intelligence (AI) and when organizations are 'competing with data.' As teams handle increasingly vast amounts of data, the demand for accuracy, consistency, and reliability in data sources has risen. Despite advancements in data management technologies, a significant gap persists in ensuring data quality. According to dbt Labs' 2024 State of Analytics Engineering survey, 57% of respondents rated data quality as one of the top three most challenging aspects of the data preparation process, a sharp increase from 41% in 2022. This statistic highlights an important reality: data quality is not merely an operational problem but a strategic imperative.

Moreover, IDC reports that 73% of data practitioners do not fully trust their data, signaling deep-seated issues with the reliability of organizational data. This mistrust directly impacts decision-making processes across departments. Forrester's research adds another layer of urgency, estimating that millions were lost in 2023 due to poor data quality, with the potential for these losses to escalate into billions as AI becomes more integral to business operations. Without a concerted intervention, the symbiotic relationship between AI and data quality could become a global liability for organizations.

The severity and scale of these problems indicate that the traditional methods of addressing data quality must be revised. Many organizations have approached data quality issues through failed initiatives , avoiding fixing problems or through long, slow, get-little-done projects. However, data quality challenges are not static; they are dynamic and continuous, simultaneously impacting multiple aspects of an organization. Addressing these data quality challenges effectively requires more than avoidance or slow bureaucracy. It calls for an ongoing, adaptive methodology that can evolve with the data itself , a methodology found in DataOps.

Why a DataOps Approach to Data Quality: Leadership Challenges

While identifying and understanding data quality issues is crucial, data quality leaders face a different, more nuanced challenge: driving change at scale across their organizations. In many instances, the people responsible for the data quality often differ from those responsible for fixing it. Data quality problems often span multiple departments, crossing into different business units and functional areas. This creates an environment where achieving widespread improvement becomes incredibly complex, as it requires cooperation from a diverse range of stakeholders, many of whom might not see the immediate benefit of investing in data quality improvements.

One of the core issues faced by data quality leaders is that source system data is often deemed “good enough” for the operational systems where it originates. However, this data frequently needs to be revised when repurposed for analytical use or downstream processes. Satisfied with the performance of their data in its original context, source system owners often do not prioritize or even recognize the need for data quality improvements beyond their immediate domain. This scenario creates friction between those who rely on data for analysis and decision-making and those who maintain the systems generating the data.

Compounding this problem is that data quality leadership is often a role of *influence* rather than *direct control*. Data quality leaders frequently identify problems but need more authority, resources, or technical capacity to implement solutions directly. They must rely on others , teams and individuals who control the data sources , to execute changes. This dependence on others makes driving action a formidable challenge, as it involves convincing people across the organization to prioritize data quality improvements. The questions that data quality leaders must grapple with are not just technical but are social:

- Where should the changes be made?
- Who should be responsible for making those changes?
- Why should the broader organization care?
- What information is needed to justify and guide the changes?

- How do I motivate someone to make the change to improve the data?

We went deeper on this dynamic in How Data Quality Leaders Can Gain Influence and Avoid the Tragedy of the Commons.

These questions underscore the leadership challenge at the heart of data quality improvement. More is needed to identify and understand the problem; data quality leaders must act as influencers, facilitators, and catalysts for change. The solution to this challenge lies in creating a structured, repeatable, and agile process that encourages continuous improvement and aligns the interests of all stakeholders , a solution that DataOps provides.

Critical Ideas in Data Quality The DataOps Way

The DataOps approach to data quality offers a compelling framework for addressing the leadership challenges and systemic issues outlined above. Drawing inspiration from Agile, Lean, and DevOps practices, DataOps applies these principles to data and analytics. It shifts the focus from a reactive, one-off project mindset to a proactive, iterative, continuous improvement process. At its core, DataOps enables individuals and teams to rapidly, collaboratively, and at scale manage and improve data quality. It encourages individuals to start small, iterate quickly, measure success, and 'ship' working results early and often.

Individual Empowerment

DataOps emphasizes individual empowerment, recognizing that the people closest to the data are often best positioned to identify and address quality issues. Rather than waiting for top-down mandates, DataOps encourages organizations to equip individuals with the tools, autonomy, and authority to drive change within their respective domains. This empowerment enables faster problem identification and more agile responses.

Experimentation and Iteration

In the DataOps approach, experimentation and iteration are central. A perfect data quality score or test is optional before taking action. Instead, teams are encouraged to start small, experiment with solutions, and iterate based on feedback and results. This iterative approach allows organizations to make incremental improvements quickly, learning from each cycle and refining their processes over time.

Leverage

One of the critical principles of DataOps is leverage , the ability to maximize the impact of limited resources. In many cases, data quality improvements can be achieved not by scaling resources but by identifying the right leverage points within the organization. This could involve automating repetitive tasks, applying AI-driven insights, or tapping into the knowledge of subject matter experts at critical moments. Data quality leaders should use scoring models tailored to distinct business needs to make data scoring more focused and impactful. This ensures scores are relevant to different departments or use cases, allowing data quality efforts to drive targeted improvements.

Agile/DataOps Approach to Data Quality

DataOps draws heavily from Agile methodologies, prioritizing adaptability, collaboration, and rapid iteration. A core tenet of DataOps is to “get a data quality process working right away” and then iterate and improve on it continuously. In practice, this means starting with a minimum viable data quality evaluation and scoring method that addresses the most immediate issues and then building on that foundation as more information becomes available. Documentation and analysis are seen not as obstacles to progress but as byproducts of the improvement process.

Measuring and Evaluating Data Quality Before Standards Are Established

DataOps encourages organizations to measure and evaluate data quality *before* establishing formal standards. Early measurements provide a baseline against which future improvements can be gauged. Over time, these measurements help inform the creation of data quality standards, which can then be refined and improved through further iterations. By starting with measurable outcomes, data quality leaders can make data-driven decisions about where to focus their efforts and how to prioritize improvements.

> *[G]ood quality and the proper uniformity have no meaning except with reference to the consumer’s demands.*

> *Deming, W. Edwards. (1953). Statistical Techniques and International Trade. Journal of Marketing, 17(4), 428-433. Sage Publications, Inc., page 428*

Continuous Learning

The DataOps approach is predicated on continuous learning. As teams cycle quickly through improvements, they learn more about their data, processes, and organizational needs. This rapid feedback loop enables teams to adapt quickly to changing requirements and evolving data landscapes, ensuring that the organization remains nimble in the face of new challenges.

Where to Start with Data Quality The DataOps Way

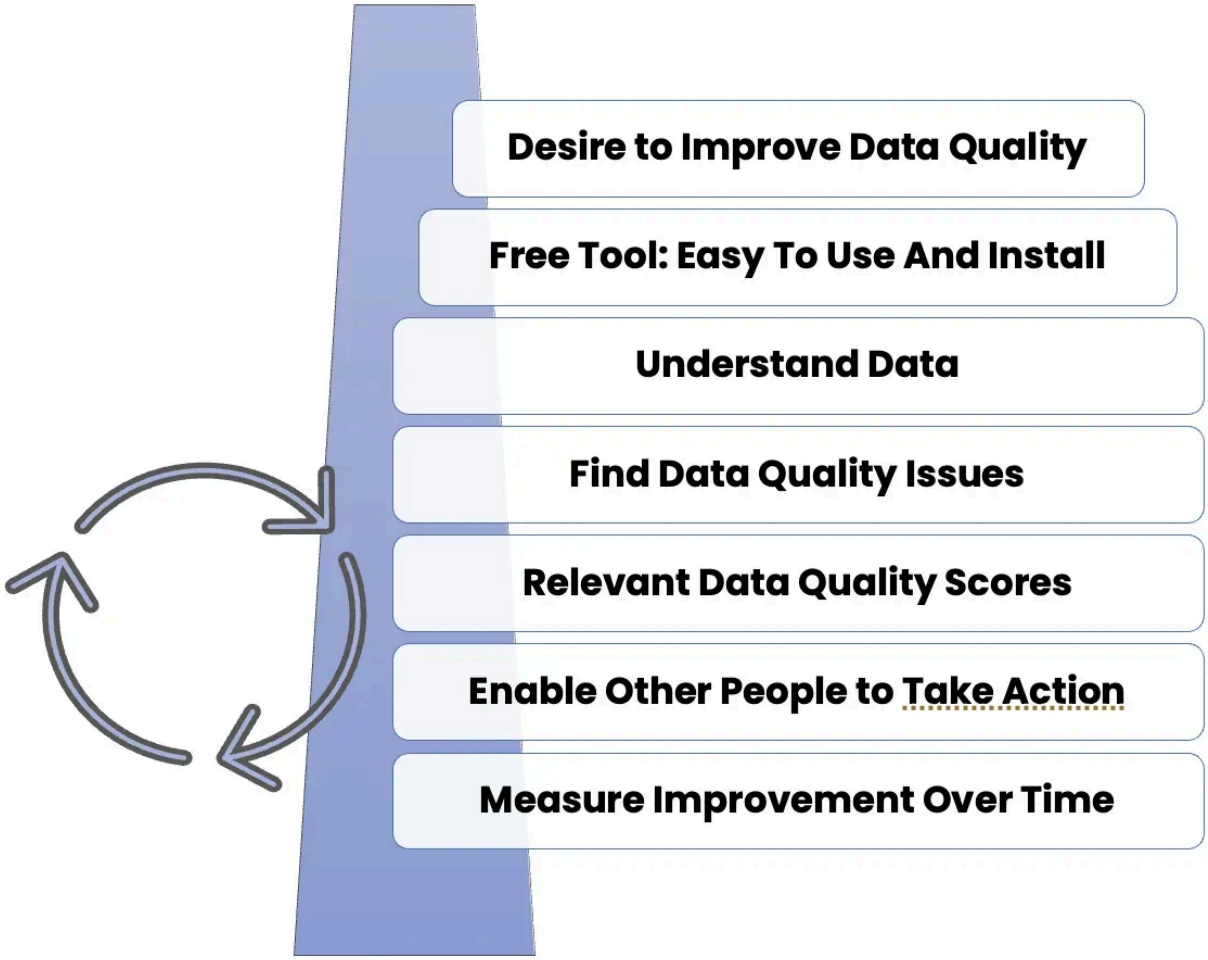
The first step in implementing DataOps for data quality is to start small, focusing on individual empowerment and personal leverage. Instead of attempting a large-scale, organization-wide rollout, DataOps data quality encourages organizations to begin with a single person who can gain influence.

Still, it may need more authority over the data. This individual should be given a tool that enables them to identify meaningful data quality improvements with minimal ramp-up and at no significant cost.

Organizations can quickly see the benefits of data quality by starting small. This approach allows for learning and improvement over time, as each iteration provides valuable feedback that informs future efforts. Automation and data quality AI can play a significant role in this process, helping to streamline testing and analysis while reducing the manual burden on individuals. The goal is to make it easy to achieve impact, fit the solution to the organization's specific needs, and measure success along the way.

How to Do Data Quality The DataOps Way: Seven Steps

The following outlines seven steps to do data quality the DataOps way. It focuses on empowering one person with no power but has influence and enables them to start the virtuous data quality influence, action, and measure cycle immediately.



Step 1: Start With A Desire to Improve Data Quality

Finding individuals within the organization who are motivated to improve data quality is essential. These individuals, often tired of being the “data quality nags,” can play a pivotal role in driving change. Organizations can harness their passion and expertise to make meaningful improvements by giving them the necessary tools to influence.

Step 2: Empower an Individual with a Free Tool That Gives Them Data Quality Superpowers

The first step in the DataOps approach is to empower an individual who may not have direct authority but influence within the organization. This person should understand the importance of data quality and be motivated to drive change, even if they do not control the systems generating the data. Organizations can start improving data quality from the ground up by giving these individuals the tools and resources they need to effect change. To enable these individuals to start making improvements quickly, providing a tool that is easy to install and use is crucial. This tool should not require extensive IT support or additional infrastructure, allowing individuals to begin improving data quality within hours. By making the process as simple as possible, organizations can remove barriers to entry and accelerate the path to improvement.

Step 3: Help Them Understand Their Data Through Profiling

Data profiling is a critical step in understanding the quality of data sources. It visually represents the data and checks for conformance with business rules. Profiling helps visualize data problems, such as missing values, inconsistencies, or patterns, and serves as an excellent starting point for data quality workshops. By enabling fact-based discussions about the causes and impacts of data problems, profiling helps foster collaboration and drive solutions.

> *Without data, you’re just another person with an opinion.*
>
> *W. Edwards Deming*

Step 4: Identify Data Quality Issues

Once data has been profiled, the next step is identifying specific data quality issues. Identifying data quality issues is critical to ensure that datasets are reliable, consistent, and ready for analysis. Data is complicated; coding hundreds of data quality tests for every table is time-consuming and never gets done. Data quality leaders need an accelerator that automatically creates data quality tests. They need to manage and build company-specific tests with ease.

One major category of issues involves data hygiene problems. These hygiene issues can often be detected and mitigated through qualification and hygiene screening. Regularly conducting a periodic review of source data is essential to keep track of any degradation in data quality over time. Additionally, it’s essential to surface data hygiene requirements early on to establish data cleanliness and reliability expectations. These requirements should be well-documented for collaborative review, ensuring that teams are aligned on the standards and any updates or concerns are shared.

A few example tests can help with hygiene management. These include confirming data typing, such as checking for numbers appearing in columns intended for text (alpha columns) and checking blank value representation to ensure no critical data is missing. It's also necessary to verify that distinct values are present where applicable and assess any string pattern inconsistencies that might suggest data formatting issues.

In addition to detecting hygiene issues, it is possible to automatically generate data quality scores based on data variations. One method for this is consistency testing, where anomalies from the baseline are identified. Baseline characteristics can be derived from data profiling, enabling teams to infer rules and documentation that establish acceptable data patterns automatically. These rules can then lead to the automatic generation of parameterized tests, providing a consistent mechanism for data validation over time.

Anomalies can arise in several forms. For instance, freshness anomaly checking ensures that the dataset is up-to-date, verifying that new data has been received. Similarly, volume anomaly checking confirms that the correct amount of data is being collected, while schema anomaly detection checks if the incoming data conforms to the expected schema. Data drift checks assess whether the received data makes sense compared to previous datasets. Deploying a wide net of simple tests , such as validating required values, calculating missing value percentages, and checking uniqueness , can help ensure data quality. These tests can also include list-of-values checks against baseline values, alpha and decimal truncation evaluations, and verifying that data recency is within acceptable ranges. Additional tests might involve comparing record counts within specific time windows, confirming min/max values against the baseline, and assessing distribution shifts over time.

Configurable business rules can also help address data quality concerns. These rules allow for tests that fit single-table and multi-table structures. Business rule testing leverages the knowledge of domain experts to establish accurate data validation rules. Once set, these rules double as documentation for the dataset. The system can then run parameterized tests to verify compliance with these rules, ensuring more signal and less noise in the results. Importantly, these evaluations should focus on the data in a stable state rather than in the middle of a batch process, as this ensures more accurate results.

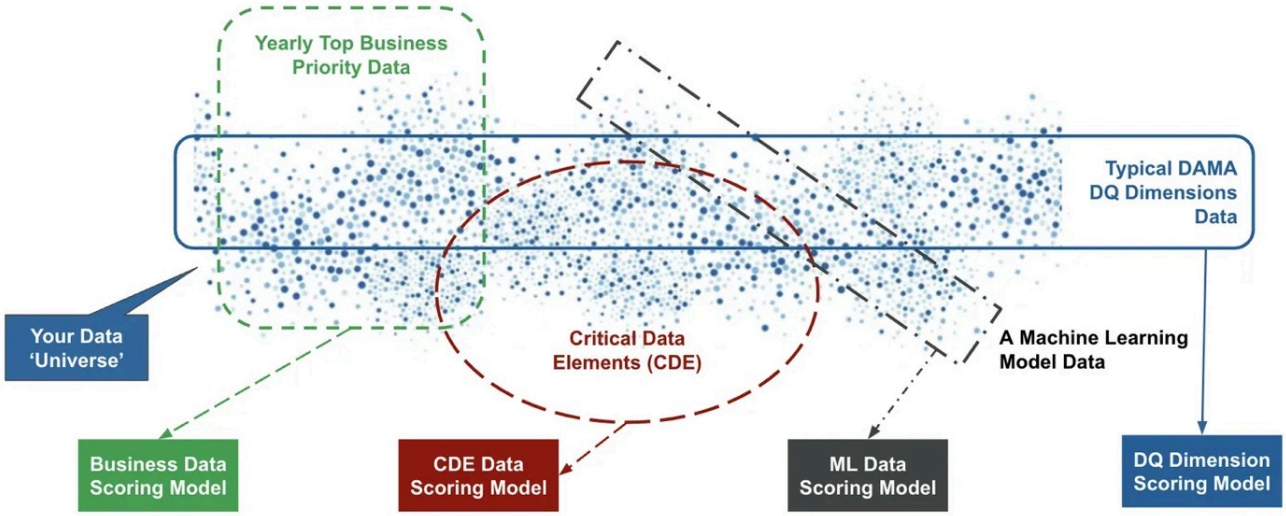
Several example tests can be applied using business rules. For instance, a test might validate that list-of-values data aligns with expected entries, such as checking if a delivery status field contains only acceptable values. Another example could be ensuring that entity relationships are enforced, such as verifying that every SKU exists in the distributor list. Additional tests could confirm the absence of duplicate values in unique fields or perform referential integrity checks across related tables.

For more specific scenarios, custom data quality tests can be created. These tests can be tailored to specific use cases by allowing users to supply the SQL queries needed to validate the data. Vertical domains and industries often have unique requirements, making it necessary to design industry-specific tests. For example, in the healthcare industry, a rule might state that the number of medical practices should not exceed the number of doctors, or in logistics, it could be required that no shipments are made on Fridays. By reserving programming expertise for these complex, custom challenges, organizations can maximize the value of their technical resources without overburdening them with routine checks.

Finally, managing the people involved in ensuring data quality is critical to a successful data governance strategy. This can be done through a user interface (UI) or a command-line interface (CLI), allowing users to interact with the data quality process, run tests, review results, and make real-time decisions.

Step 5: Create Relevant Data Quality Scores

The final step is to implement a scoring system that measures data quality. Scoring allows data quality leaders to quantify the state of their data and communicate it effectively to stakeholders. These scores permit focus and be configurable, allowing organizations to tailor them to different use cases, whether assessing critical data elements (CDEs), DAMA quality dimensions, or machine learning model data. Organizations can track improvements over time by continuously monitoring data quality scores and adjusting their efforts.

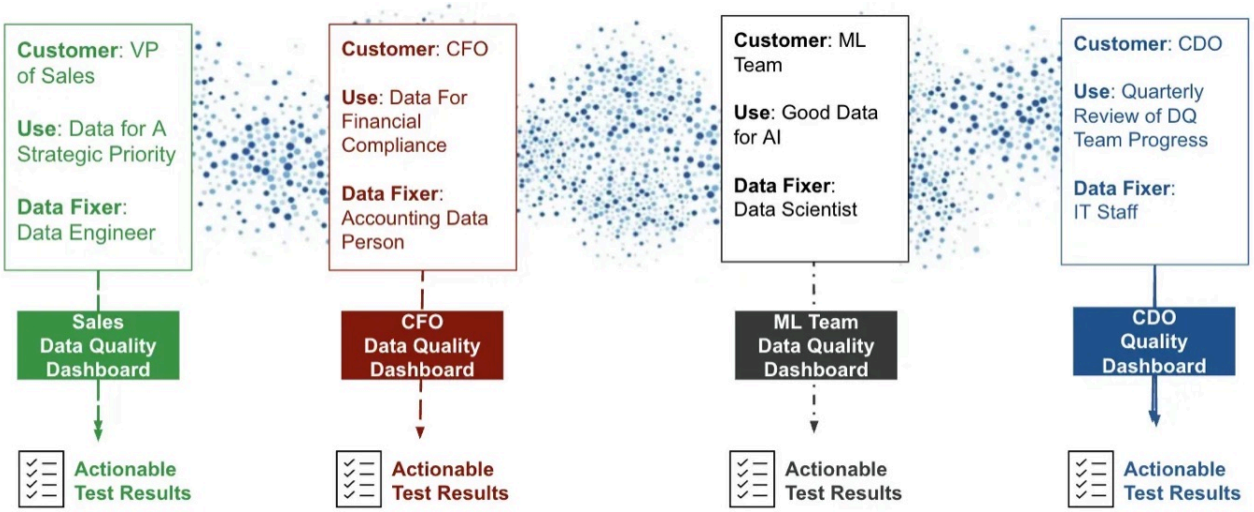


By implementing targeted and relevant data-scoring models, data quality (DQ) leaders can significantly enhance their organizational influence. It is crucial to prioritize scoring only the data elements that matter most to the business, avoiding extraneous metrics like fax numbers that add little value. Instead of adopting a broad approach to data quality scoring, which often results in marginal insights, leaders should focus on scoring data that aligns with the organization's needs. This means designing scoring models that address critical business requirements and customer priorities, ensuring meaningful and actionable results.

Data quality leaders should develop multiple scoring models tailored to distinct business needs to make data scoring more impactful, rather than relying on a one-size-fits-all approach. This ensures scores are relevant to different departments or use cases, allowing data quality efforts to drive targeted improvements. Each scoring model should prioritize business-critical data elements, starting with a few metrics that genuinely resonate with the customer. As confidence and stakeholder buy-in grow, these models can be expanded to include additional elements supporting the organization's goals. This approach keeps data scoring manageable and maximizes the influence of DQ initiatives by continually

aligning data quality metrics with evolving business priorities. Focused data scoring creates a feedback loop where data quality directly contributes to operational success, cementing the DQ leader's role as a strategic partner.

On choosing what a score should actually measure, see The Six Types of Data Quality Dashboards and Why Your Data Quality Dashboard Isn't Working , and What to Do About It.



Step 6: Make It Easy For Data Quality Leaders to Cause Other People to Take Action

It's crucial to make sharing data quality scores and test results simple and impactful. Leaders must communicate the significance of identified issues effectively, convincing others to take corrective measures. Providing a shareable summary package that clearly outlines the problem, its impact, and steps for resolution makes it easier for teams to understand and fix the issue. Additionally, simplifying the process of recreating the test conditions ensures transparency and helps others effectively verify and address the problem. To streamline this process, integrating data quality management with existing workflow tools ensures that issues can be tracked, assigned, and resolved within the organization's standard processes, making it easier for teams to take action and improve data quality.

Step 7: Measure Data Quality Improvement Over Time

It is crucial to track data quality (DQ) scores over time and consistently measure improvements to demonstrate a meaningful impact on data quality. By keeping a record of these scores, data quality leaders can highlight tangible progress in data quality management and showcase their role in driving that improvement. Additionally, identifying related operational KPIs (key performance indicators) that are influenced by data quality improvements can strengthen the case for ongoing quality initiatives. For example, enhanced data quality could lead to better decision-making, fewer errors, and improved operational efficiency, all visible through KPIs. Measuring progress consistently before setting rigid standards is essential, allowing for flexibility and continuous refinement. Adopting a pragmatic approach, where an initial 70% accuracy score is acceptable, enables teams to make improvements incrementally rather than waiting for perfection, which can cause delays.

As new data comes in, data quality scores are expected to evolve. This makes it crucial to continuously revisit and refine the data quality tests that generate those scores. Regularly refining DQ standards ensures that tests become more accurate and yield fewer false positives, providing a clearer picture of data quality issues. Additionally, real-world data scenarios offer valuable learning opportunities, allowing teams to adapt their testing processes and standards based on practical experience. Over time, this continuous learning process means that even less-skilled team members can be involved in maintaining and improving data quality as they build knowledge of the data and processes. Monitoring DQ score trends on a chart provides visibility into data quality improvements and the effectiveness of evaluation criteria. As standards and data improve, the chart will reflect the progress, reinforcing the impact of refined data quality efforts.

Keep track of DQ scores over time.

Identify related operational KPIs that improvements in data quality would impact.

Be able to point to where you have shown improvement as a DQ leader.

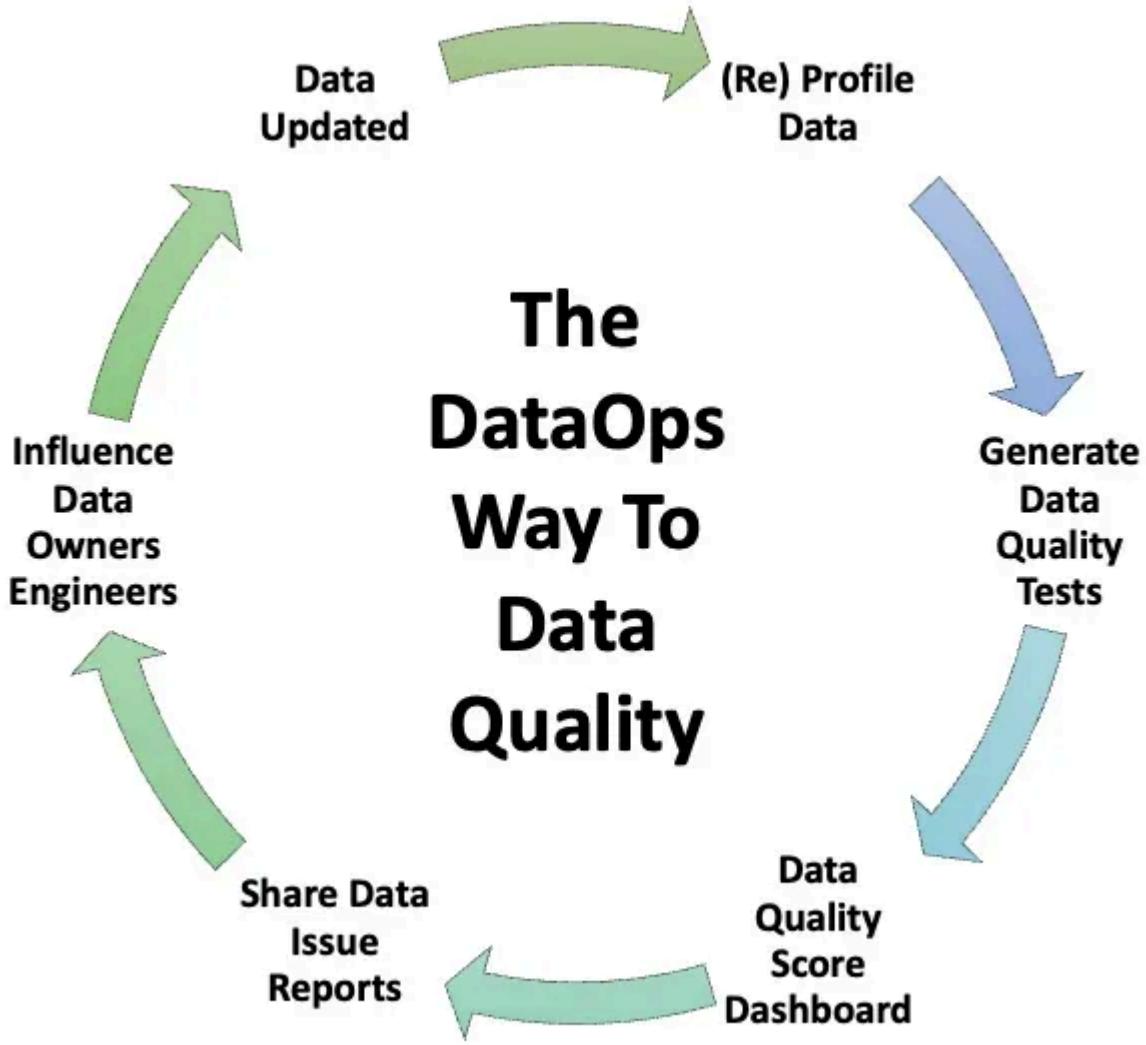
Measure first, then establish standards later.

Traditional, Waterfall, Data Quality Projects Versus The DataOps Data Quality ‘Influence and Action’ Cycle

Traditional approaches are about requirements gathering, analysis, meetings, documentation, and planning before any action is taken to improve data quality. Waterfall project management approaches often fail because they rely on rigid, sequential processes that need more flexibility in accommodating changes. In traditional waterfall models, project phases such as requirements gathering, design, development, testing, and deployment occur one after the other, with little room for feedback or iteration until the end of the process. However, many projects, particularly in dynamic environments like data analytics, require continuous adaptation based on new information or changes in requirements. As a result, waterfall methodologies often result in delayed responses to evolving needs, causing misalignment with stakeholders’ expectations and leading to a mismatch between the final deliverable and the users’ needs.

Another major issue with the waterfall approach is that it assumes that all data quality requirements can be fully understood upfront. In reality, it is often difficult to define all aspects of a project in the initial stages. This assumption leads to a need for more iterative feedback loops, making it harder to correct mistakes or refine deliverables throughout the project. In cases where problems arise, they may only be

discovered late in the process, resulting in costly delays or even project failure. Agile methodologies, in contrast, allow for more incremental development and testing, enabling teams to identify issues earlier and make adjustments on the fly, which is essential for modern data quality environments where adaptability is critical.



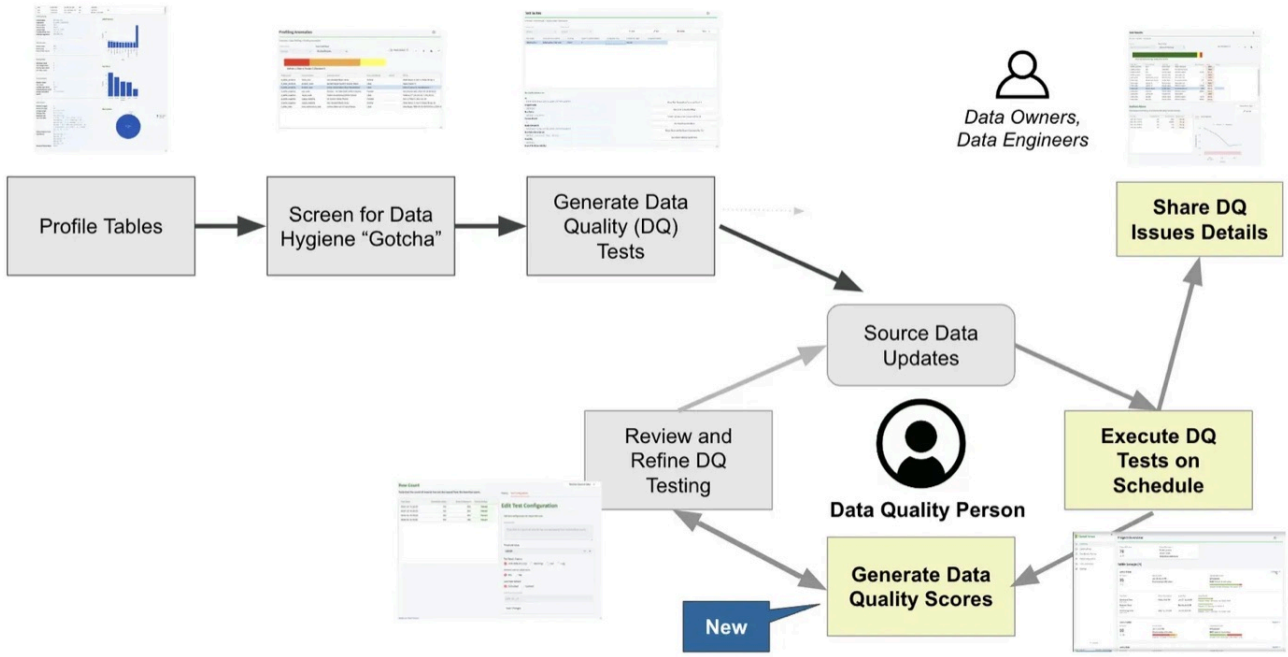
The DataOps approach to improving data quality is a cycle that empowers an individual who may not have direct authority but has influence within the organization. Organizations can start improving data quality from the ground up by giving these individuals the tools and resources they need to effect change. This person is working in a continuous cycle of influence and action. This cycle involves the following key activities:

- New data or updated data
- (Re) profile data: Establish a baseline by measuring the current state of the data with data profiling Get a foundation for tracking progress and identifying areas for improvement repeatedly

- Generate data quality tests: Generative data quality tests can streamline this process, making it easier to catch problems before they become more significant without coding
- Data quality score dashboard: Scoring provides a quantifiable way to assess the health of the data and communicate the results Scores should be focused on the data elements that matter and reflect top organizational needs
- Share data quality issues report: Give actionable, specific changes to those who can do the technical work
- Influence data owners and engineers: Once data quality issues have been identified, the next step is to drive the necessary organizational changes Influence data owners, system administrators, or other stakeholders to address the identified issues
- Establish a baseline by measuring the current state of the data with data profiling
- Get a foundation for tracking progress and identifying areas for improvement repeatedly
- Generative data quality tests can streamline this process, making it easier to catch problems before they become more significant without coding
- Scoring provides a quantifiable way to assess the health of the data and communicate the results
- Scores should be focused on the data elements that matter and reflect top organizational needs
- Give actionable, specific changes to those who can do the technical work
- Once data quality issues have been identified, the next step is to drive the necessary organizational changes
- Influence data owners, system administrators, or other stakeholders to address the identified issues

Advantages of Data Quality The DataOps Way

DataOps provides several distinct advantages when applied to data quality management. These advantages are particularly pronounced in organizations dealing with large volumes of complex data, where traditional approaches often fall short.



Handling data at scale. DataOps helps teams handle massive amounts of changes at scale. As organizations grow and their data requirements expand, the ability to manage and ensure the quality of that data becomes increasingly important. DataOps automatically and at scale applies universal quality principles , such as consistency, accuracy, and completeness , ensuring that large datasets can be managed efficiently without sacrificing quality.

Knowledge-based trade-offs. One key benefit of the DataOps approach is its ability to facilitate knowledge-based trade-offs. Rather than making assumptions about data quality or relying on hope, DataOps encourages organizations to use real-time data and insights to inform their decisions. This ensures that trade-offs between speed, cost, and quality are made based on actual knowledge, leading to better outcomes. Don't rely on your intuition; look at the facts on the ground with actual data.

Leveraging expertise. Subject matter experts and skilled data engineers often need to improve data quality. By focusing their efforts where they are most needed, organizations can maximize the impact of their resources and ensure that expertise is applied where it counts.

Faster, easier implementation. The iterative, Agile nature of DataOps makes it faster and easier to implement real data quality improvements. Rather than waiting for a perfect solution, teams can begin implementing changes immediately, learning from each iteration and refining their processes over time. This iterative approach accelerates the implementation of data quality initiatives and ensures that improvements are continuously being made. Getting something 70% right working today is much better than a perfect solution sometime in the future.

More iterations, better data quality standards. The more iterations a team goes through in its data quality processes, the more refined its standards become. Each cycle provides an opportunity to learn, improve, and adjust the standards to reflect the organization's needs and goals better. Over time, this

leads to establishing more robust, actionable standards that drive fundamental improvements in data quality. Maximize your team's learning from experience, not second-hand anecdotes.

Conclusion: Data Quality The DataOps Way

Ultimately, the DataOps approach allows organizations to get data quality processes working quickly and then iterate and improve over time. Documentation and analysis become natural outcomes, not barriers to progress. By starting with testing and measurements, even before standards are fully established, organizations can build a foundation for continuous improvement. The faster the iteration, the more organizations learn, refine their processes, and elevate their data quality standards.

Data quality, the DataOps way, is about action, agility, and learning , making it the best way to drive meaningful, scalable, and lasting improvements in data quality. With data quality the DataOps way, organizations can:

- Start with minimal cost, using tools that run on individual laptops.
- Learn and improve over time with a low ramp-up.
- Leverage automation and AI to streamline testing and improvements.
- Ensure that the process fits the specific needs of the organization.
- Continuously measure success and drive further improvements.

The DataOps approach to data quality provides a scalable, iterative, and flexible framework for improving data quality across an organization. By empowering individuals, leveraging expertise, and focusing on continuous improvement, DataOps enables organizations to address their data quality challenges proactively and effectively. Rather than waiting for perfect solutions, DataOps encourages organizations to start small, iterate quickly, and learn from each improvement cycle.

WHY DATA QUALITY MATTERS

Data Quality vs Data Observability: The Pets and Cattle of Your Data Estate

Read the original article at: https://datakitchen.io/blog/data-quality-vs-data-observability-the-pets-and-cattle-of-your-data-estate/

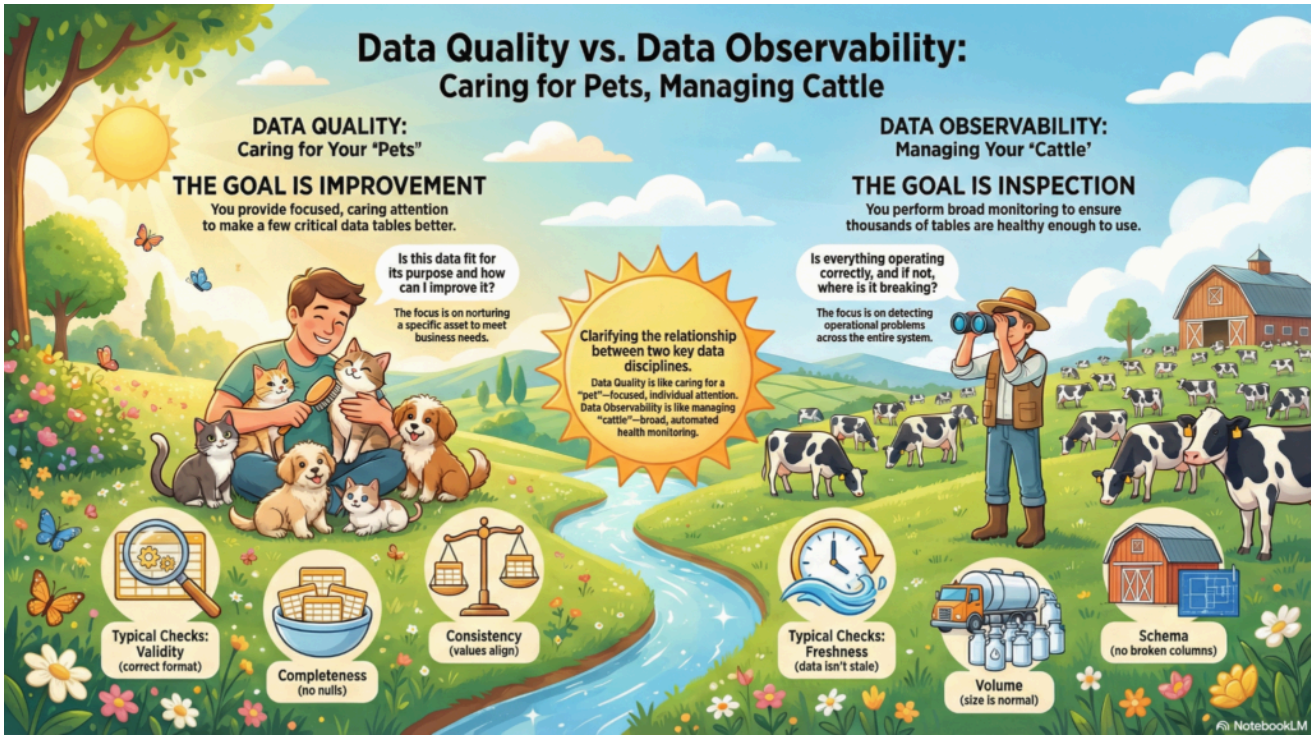
If you've spent any time in DevOps circles, you've heard the phrase "cattle, not pets." Randy Bias popularized this analogy around 2012 to explain a fundamental shift in how we think about infrastructure. Pets are precious, unique systems you know intimately and nurture individually. Cattle are herds you manage collectively, you don't name them or know their personalities, you just need to know they're healthy enough to go to market.

As Bias put it: "In the old way of doing things, we treat our servers like pets, for example, Bob the mail server. If Bob goes down, it's all hands on deck. In the new way, servers are numbered, like cattle in a herd. When one server goes down, it's taken out back, shot, and replaced on the line."

This framework provides a useful lens for understanding two distinct but complementary disciplines that data engineering leaders must master: data quality and data observability. Both aim to increase trust in your data. Both require data quality checks. But they operate at different scales, serve different purposes, and demand different tooling.

Data Quality Is Caring for Your Precious Pets

Your pet tables are the ones you know and love, those lovingly hand-modeled star schemas, business-critical fact tables, and gold-layer marts. You know their names, their purpose, their refresh schedules, and their quirks. Like a cherished family pet, each one gets individual attention. You care about their well-being. You want them to thrive.



Data quality is the pet approach: focused, caring attention on a limited set of tables at a time, with the explicit goal of making them better.

You Care for Pets Because You Want Them to Improve

When you have a precious pet, you don't just check that it's alive, you invest in its health and development. You track how it's doing over time. You notice when something's off and take action. You want it to flourish.

Data quality work is the same. You're not just monitoring, you're actively improving the data to ensure it meets the needs of downstream consumers. This requires dashboards that drive action, not vanity metrics that look impressive but change nothing.

As we explore in our Masterclass on the Six Types of Data Quality Dashboards, effective data quality programs need multiple dashboard types serving different audiences and purposes, from executive scorecards that drive accountability to operational views that help engineers prioritize remediation work. These dashboards track progress over time, connect quality issues to business outcomes, and motivate teams to improve data quality.

- Validity checks: Does every value conform to expected formats, ranges, or patterns? Are email addresses properly formatted? Are dates within reasonable bounds? Are status codes drawn from a valid set?
- Completeness checks: Are required fields populated? Are there unexpected nulls in columns that should always have values?
- Consistency checks: Do related values align? Does the sum of line items equal the invoice total? Does every foreign key reference a valid primary key?
- Business rule validation: Does the data satisfy domain-specific requirements? Are there exactly 25 sales territories? Does every transaction have an assigned region?
- Uniqueness and referential integrity: Are identifiers actually unique? Do relationships between tables hold?
- **… And Many More**

The goal isn't just detection, it's improvement. You define what "correct" means, track quality scores over time on dashboards, identify gaps, and drive remediation. You're making your precious pets healthier and stronger.

Data Observability Is Managing Your Cattle

Your cattle is a database, or data warehouse, or lakehouse, with thousands of tables. You don't know them all individually. You can't. No one has time to understand the purpose and semantics of every table in a modern data estate. Many are automatically generated, machine-produced, or simply not owned by any specific person.

You're not trying to improve each one. You just need to know: are they healthy enough to go to market?

Data observability is the cattle approach: broad monitoring across your entire herd, checking that each animal is big enough, has all its legs, and is ready to be useful.

Data observability asks: Is everything operating correctly across my data estate, and if not, where is it breaking?

You Check Cattle to Make Sure They're Market-Ready

When you're running cattle, you don't write a unique care plan for each animal or track its personal development over time. You run them through a chute and check the basics: Are they the right size? Do they have all their legs? Any obvious problems? If they pass inspection, they go to market. If something's wrong, you deal with it.

Data observability works the same way. You're not trying to make each table excellent, you're checking that it meets basic health standards

- **Freshness**: Is this table fresh enough? If a table that normally refreshes every hour hasn't been touched in six hours, something may be wrong upstream. It's like a cow that hasn't been eating. You need to investigate.
- **Volume**: Is the table the right size? A table that normally receives 100,000 rows per day suddenly receiving 10,000, or 10 million, signals a potential issue. Too small or too big, something's off.
- **Schema**: Does it have all its parts? Missing columns, type changes, and renamed fields, these are like missing legs or tails. The table might still exist, but it won't work properly downstream.
- **Drift** from baseline: Is it within normal bounds? You profile data characteristics, null rates, cardinality, value distributions, and check that current patterns haven't drifted too far from what's expected. If a cow suddenly looks completely different from the rest of the herd, you notice.

This is cattle-scale inspection: time series anomaly detection and alerts that tell you when something in the herd isn't right. You're not trying to make each table perfect, you're making sure none of them are broken enough to cause problems when they reach their destination.

Both Disciplines Require Data Quality Tests

Here's the important nuance: both data observability and data quality rely on data quality checks. The difference is how those checks are generated, applied, and acted upon.

In data quality, checks encode business knowledge and specific requirements. You write checks because you understand what "correct" means for this particular table and use case. You care about this pet and want it to improve. When a check fails, it's tracked on a dashboard, scored, and prioritized for remediation.

In data observability, checks are largely automated and pattern-based. You profile data to establish baselines, then generate checks that detect deviation from those baselines. You don't need to understand what each table is for, you just need to know if it's healthy enough to pass through. When a check fails, you get an alert to investigate.

The checks may look similar, both might verify that null rates stay below a threshold, but the context differs. Quality checks ask, "Does this meet our standard for excellence?" Observability checks ask, "Is this healthy enough to go to market?"

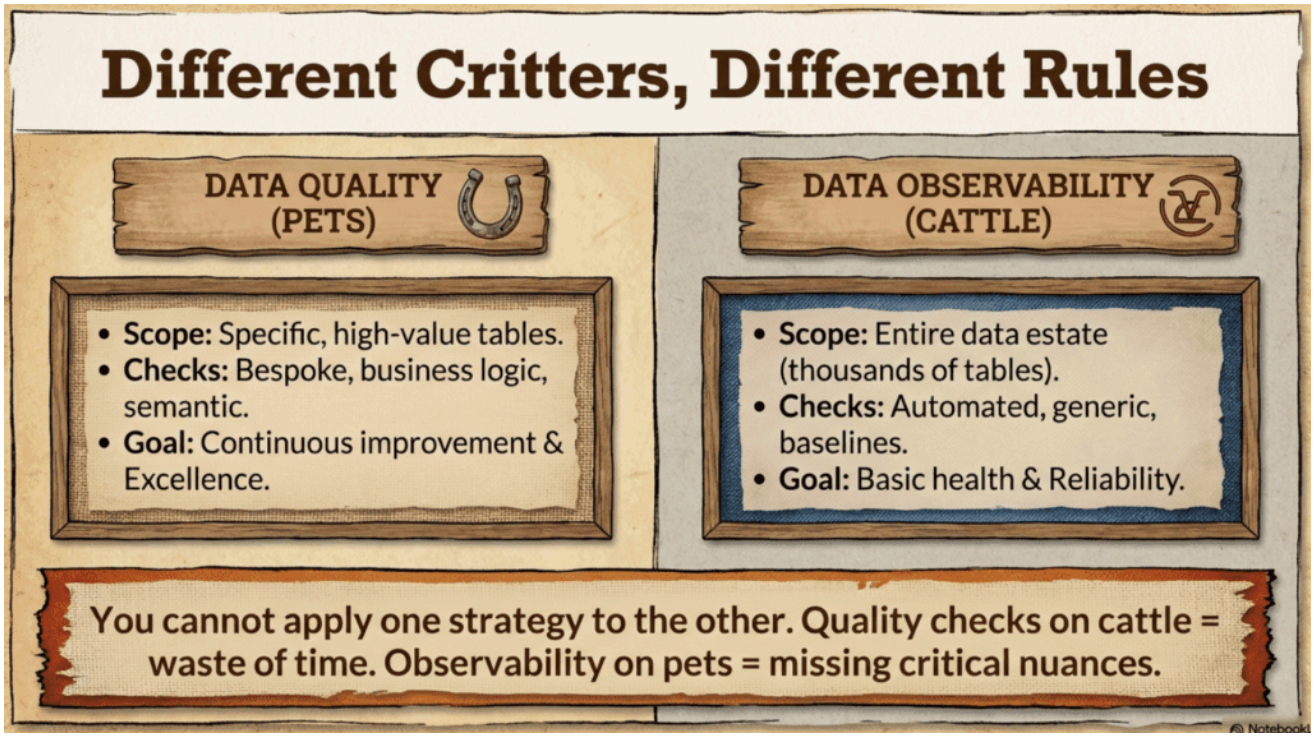
The Operational Models Differ

Quality failures require care and improvement. Someone understands that table's purpose. Someone knows the business impact. The failure isn't just "something changed", it's "our precious data asset isn't meeting its potential." The response is root cause analysis, remediation, and tracking improvement over time on your dashboards. You're nursing your pet back to health.

Observability alerts require triage. You may not know why the table exists or who depends on it. Your job is to confirm something actually looks wrong, assess whether it's serious, and either fix it quickly or route it to whoever owns that part of the pipeline. You're not trying to make this table excellent, you're just making sure it's not broken. Pull it out of the herd, deal with the problem, move on.

Building a Unified Strategy

Invest in data quality for your precious pets. Identify the tables that matter most, the ones driving decisions, the ones with clear consumers, the ones you want to make excellent. For these, define what "correct" means, implement comprehensive hygiene and quality checks, and track improvement on dashboards that drive action.



Deploy observability across your cattle. Use automated profiling to monitor all your tables, freshness, volume, schema, and statistical drift. This is your herd inspection. When something fails the basic health check, alerts fire, and you investigate.

Let tables graduate from cattle to pets. Many tables start as cattle, monitored generically through observability because you don't know their purpose. As you learn more about a table's importance, you can promote it by adding specific quality checks, tracking scores on dashboards, assigning ownership, and taking steps to improve it.

Recognize that most tables stay cattle. And that's fine. Not every table deserves the attention you give your pets. Generic observability coverage, checking that they're fresh enough, the right size, and structurally sound, is sufficient for tables that just need to be healthy enough to serve their purpose.

What This Means for Data Engineering Leaders

Understanding the distinction between data quality and data observability helps you allocate resources appropriately:

For your pets: Invest in dashboards that track improvement over time. Define quality standards and measure against them. Build remediation workflows. Make sure quality failures connect to business outcomes so stakeholders care about improving the data. These are your precious assets, treat them accordingly.

For your cattle: Invest in time series anomaly detection and alerting infrastructure. Automate baseline profiling. Build efficient triage workflows. Don’t try to write bespoke checks for thousands of tables, just make sure they’re healthy enough to pass inspection.

For your organization: Recognize that both disciplines are necessary. Quality without observability means your pet tables are excellent while unknown problems fester across your broader estate. Observability without quality means you catch broken tables but never truly make your critical data assets great.

The Cattle Drive Forward

The distinction between pets and cattle has helped a generation of infrastructure engineers understand modern operations. The same mental model clarifies the relationship between data quality and data observability.

Data quality is caring for your precious pets, focused improvement on specific tables you know and love, tracked through dashboards that drive action and accountability. You want these tables to be excellent.

Data observability is managing your cattle, checking thousands of tables you don't know individually to make sure they're fresh enough, the right size, and have all their parts. You just need them healthy enough to go to market.

Both require data quality checks with complete test coverage. Both contribute to trust. The difference is between nurturing something precious and inspecting a herd.

Master both disciplines, and your data estate, the pets you cherish and the cattle you manage, will deliver the trust your organization needs.

Are these the ontly to uses of data quality tests?

Having adequate data quality test coverage is very important to **four** key activities that every data and analytics team does:

1. **Data Quality:** trying to improve the source quality of your data, making sure it's fit for purpose.
2. **Data Observability:** This is where you try to ensure you don't have data errors when you have lots of tables and get blamed for obvious data problems like freshness.
3. **Data Production:** Monitoring the data journey and stopping it with tripwires before errors reach production
4. **Deployment to Production:** You have a team of data engineers, scientists, and others who are writing code. You want to ensure the code is adequately tested. Before it gets into production, this means going beyond unit tests into full regression and functional tests in development.

Stop Paying the Data Quality Tax

Read the original article at: https://datakitchen.io/blog/stop-paying-the-data-quality-tax/

Why Data Quality And Data Observability Tools Cost So Much (And Why That's Ridiculous). You're paying enterprise prices for commodity algorithms. Here's what's really going on.

We understand the struggle. Data pipelines fail, dashboards go stale, and someone in finance spots the problem before you do. Suddenly, you're firefighting instead of building. Data observability tools promise to fix this. Sounds great, right?

For what, exactly? Let's talk about what you're actually paying for, and why the pricing in this market has gone completely off the rails.

Part 1: The Dirty Secret of Data Quality and Data Observability Pricing

Enterprise data quality and data observability vendors have figured out the perfect scam: charge you more every time you try to do the right thing.

Got 10,000 tables? Congratulations. You now need a six-figure budget just to find out whether your data is broken. So you don't test everything. You pick favorites. You cross your fingers. And when bad data sneaks through the tables you couldn't afford to monitor? That's your problem, not theirs.

Data quality isn't a premium feature. It's not a luxury. It's the bare minimum for trusting your analytics, your ML models, your dashboards, your decisions. The idea that teams should ration their data quality coverage like wartime bread is absurd.

It's All Commodity Code Under the Hood

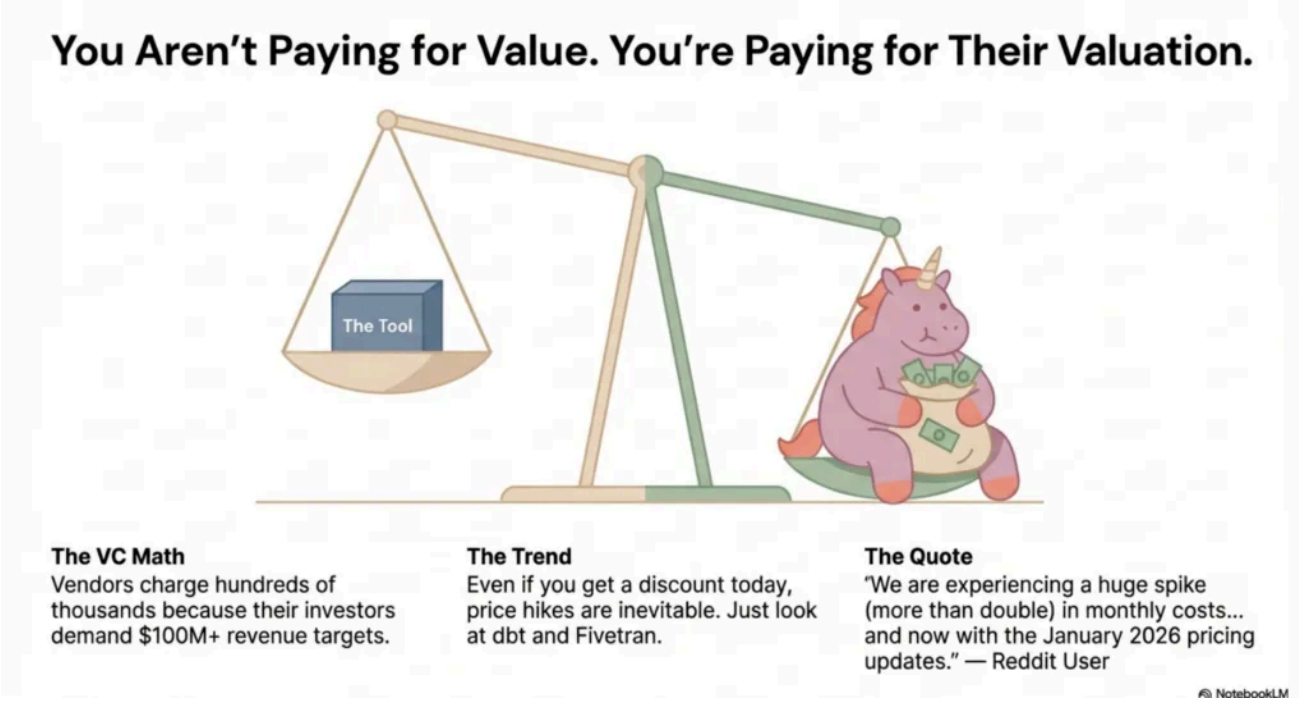
Here's something vendors don't like to discuss: every data observability tool runs on the same basic time-series anomaly detection algorithms, e.g., Facebook's Prophet orARIMA. Some combination of similar methods. There is no secret sauce. Time-series anomaly detection is still an active area of research in universities, and no vendor has cracked something the academic community hasn't already published.

Most of these tools take open-source algorithms, wrap them in a nice UI, and call it a platform. That's the product. That's data observability.

Does it work? Sure, it can catch issues. But when a sales rep tells you their "AI-powered" platform justifies enterprise pricing because of proprietary magic, they're selling you marketing copy, not technical differentiation.

The Real Reason for the Price Tags: Venture Capital Math

If the technology is a commodity, why are prices so high? Because vendors need to charge hundreds of thousands of dollars a year. Their investors expect them to hit $100 million or more in revenue. The pricing isn't based on cost to serve or value delivered. It's based on what VCs need to see on a spreadsheet.



Even if you're getting a discount today, a price hike is coming. Look at what happened with dbt and Fivetran. One Reddit user put it plainly: "We are experiencing a huge spike (more than double) in monthly costs following the March 2025 changes, and now with the January 2026 pricing updates." Every year brings a new price increase, and pay-per-usage models make it worse. Your costs rise automatically as your data grows.

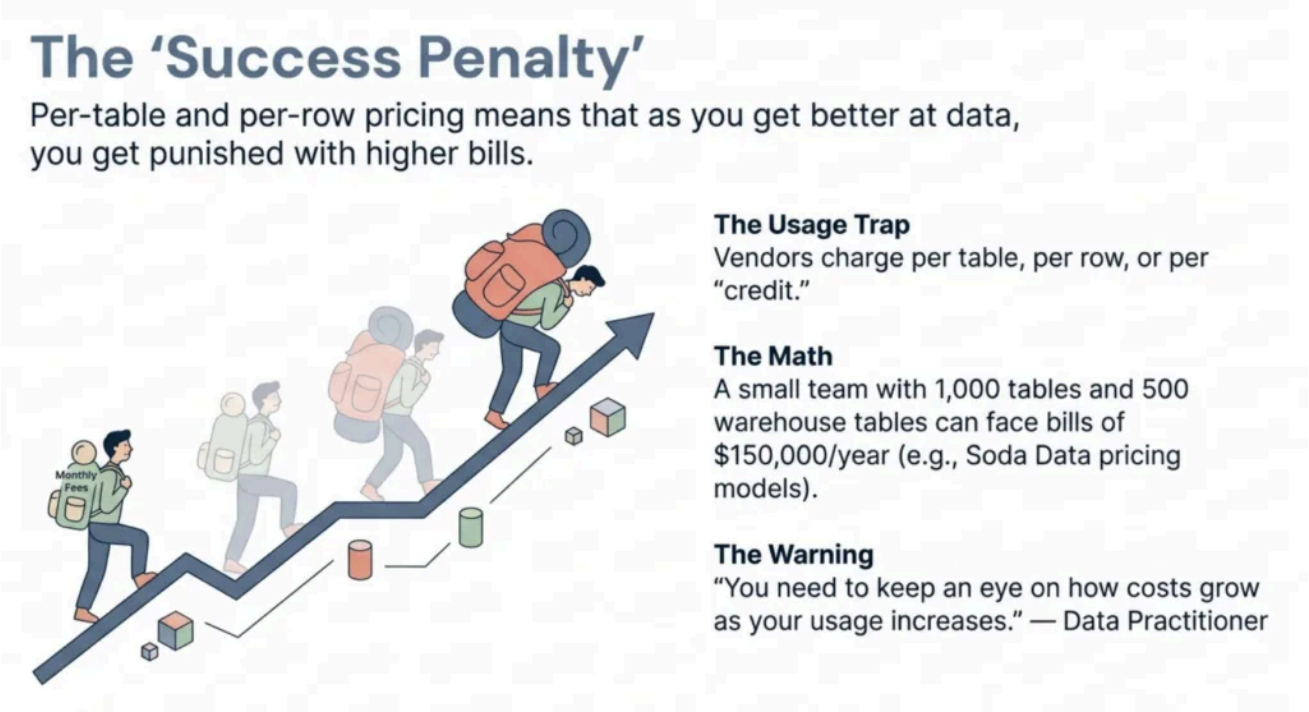
The data engineering community has caught on. As one practitioner warned: "You need to keep an eye on how costs grow as your usage increases." This isn't paranoia. It's pattern recognition from people who've watched their bills double and triple.

Most vendors lure you in with friendly pricing, then jack up rates once your data pipelines depend on them. Just ask a Fivetran or dbt customer what their bill looked like in year one versus year three.

What It's Actually Like to Use These Tools

The gap between the sales pitch and operational reality comes up constantly when practitioners speak honestly. One data engineer shared their experience on Reddit: "My personal experience working at a fintech startup leveraging Monte Carlo is that it has a lot of beneficial features but it's overly complex for use across the organization (the DE team found it useful), prohibitively expensive if you want to cover your entire infrastructure, and takes a dedicated effort (at least a month for us) to get the alerting to a manageable state to avoid alert fatigue."

Think about that. Even at a well-funded fintech, the tool was too expensive to cover the full cost of their infrastructure. The team spent a month tuning alerts to avoid being buried in noise. Only the data engineering team found it useful. Not the analysts, not the business users, not the broader organization that was supposed to benefit from better data quality.



Another engineer was more direct: "It's expensive as golden shit. If your company has money to throw around, go with it. If not, there are better ways to spend cash."

These prices create a strategic problem. As one frustrated practitioner explained: "Boiling the ocean: we understand not to implement it slowly, but the DQ tool is priced too high to go slow on value realization. They want a proven main player tool that can be implemented below $10K and ramps up when we need it. Our VPs would be all ears."

Best practice says start small, prove value, then expand. But these tools are priced for big, enterprise-wide commitments from day one. You can't start small and grow. You either overpay before proving ROI, or you walk away entirely.

Who Actually Needs Six-Figure Observability?

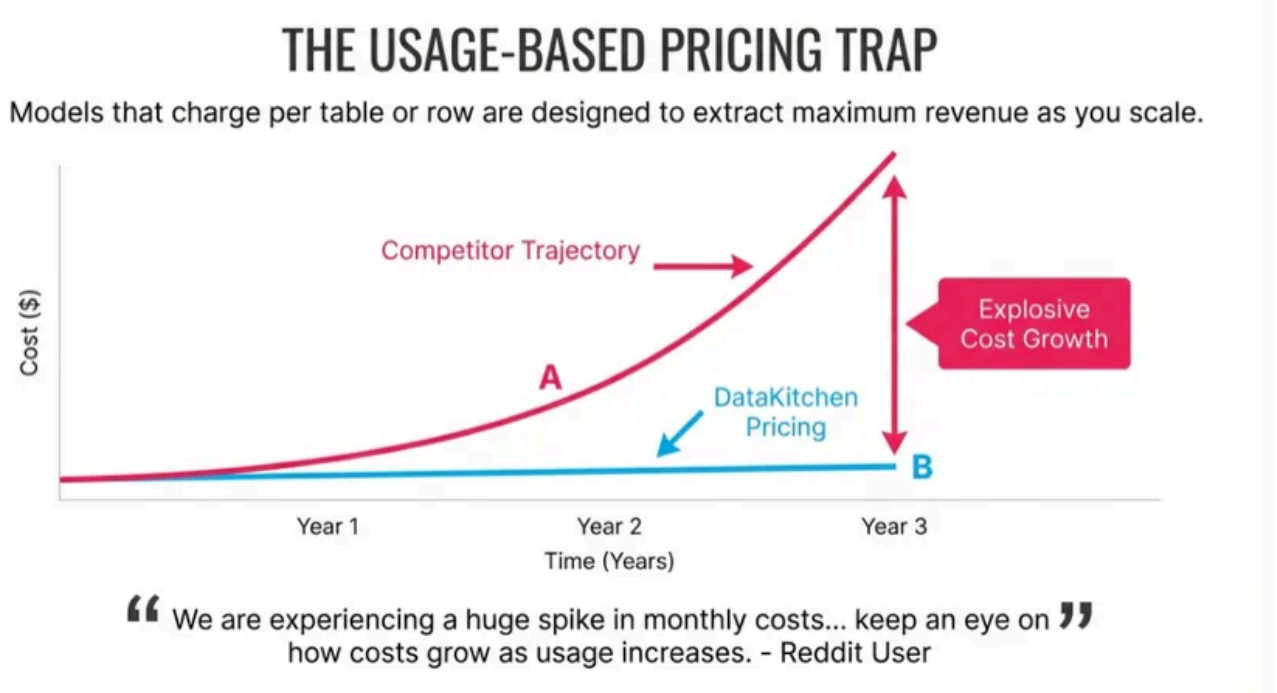
To be fair, there are scenarios where expensive tools might make sense. As one Redditor noted: "If your data team has more than 10ish people and you spend more than 30% of your time on quality issues, it's likely worth investigating."

Notice those qualifications: more than ten people, spending over 30% of your time on quality fires. That describes a large team drowning in data problems.

What about the majority of teams? Four to eight people, real quality issues, but no way to justify $80,000? The current market has essentially abandoned them.

Usage-Based Pricing Is a Disaster Waiting to Happen

One of our customers is a small company with 400 employees and a data team of six people. She has 1,000 tables in her main application and another 500 in a data warehouse. Soda Data would cost her $750 per month base plus per-table fees that add up to nearly $12,000 per month. That's over $150,000 per year. DataKitchen costs her $9,000.



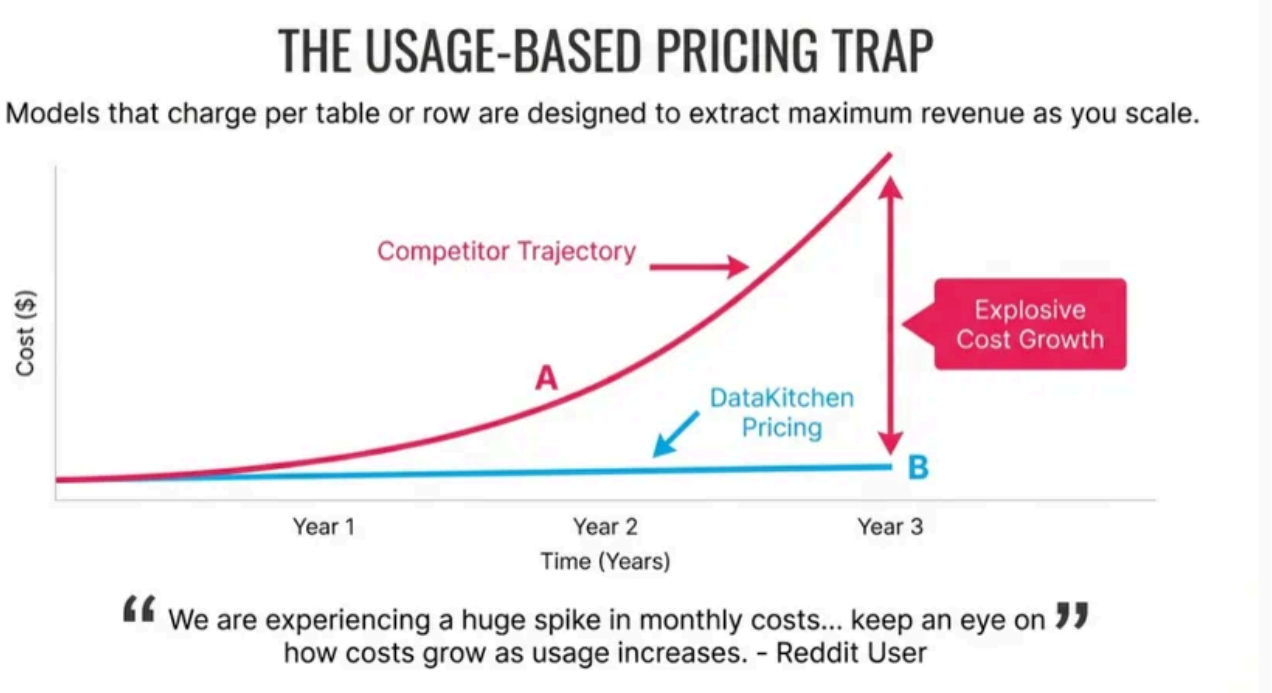
Why the difference? Soda just raised a new round of venture capital in April 2024. They have investors to satisfy. Prices will rise.

There are more than 50 companies offering data quality data observability tools, backed by over half a billion dollars in investment. They’re all waiting for you to buy, then raise prices to hit their growth targets.

With per-table pricing, your bill grows linearly (or worse) with your data. With flat pricing, you can actually afford to monitor everything. You stop guessing which tables matter. You stop crossing your fingers. You sleep at night.

What Data Observability Should Actually Cost

So what’s a reasonable price? Not $100,000 a year. Not a million dollars. For a small team of 4 people with a few databases, it should cost $0 or at most $9,000- $10,000 per year. That price should include support, updates, and a real partnership with your vendor. No extra charges for adding tables. No penalties for running more tests. Everything is included in a flat price.



This isn’t wishful thinking. The economics support it. The algorithms are open source. Cloud computing is cheap. The high prices are due to the business model, not the technology.

You should be able to detect broken pipelines for $100 per month, not $100,000 per year. The technology exists. The market just hasn’t caught up.

WHY DATA QUALITY MATTERS

How I Broke Our SLA and Delighted Our Customer

Read the original article at: https://datakitchen.io/blog/how-i-broke-our-sla-and-delighted-our-customer/

I broke one of our most critical SLAs just last week, and it was the best thing that could have happened.

It was shaping up to be a major embarrassment. One of our key data warehouse refreshes had failed. No new data. No dashboard updates. The refresh was long past its deadline, the project's key data engineer was on vacation, and I was playing backup. Where was I? At the moment, I was flying home from a data quality conference. This was not good.

Naturally, the airline's Wi-Fi was terrible, but I managed to get through to a colleague, who knew little about the project. And then we started troubleshooting, one Slack message at a time. I slacked him SQL. He slacked back data. Back and forth we went. An embedded test had failed. Where had our process gone wrong? Everything I could see looked fine. And I was tempted, so tempted, as the clock kept ticking, to disable the test and let it go.

Then it dawned on me that this test wasn't even ours. It was one of a series of tests that had been added by one of our principal stakeholders in collaboration with his analytics team - esoteric referential integrity checks that had been placed at key points in the build, validating consistency between multiple data sources that fed hourly into our reporting layer.

These tests weren't easy to define or implement. They came from an understanding of the business rules that we, as data engineers, lacked. And they caught a major problem: the new records we received from one source were completely out of sync with our other data.

Something had gone wrong upstream, very wrong. And because our process was designed to fail fast when integrity was violated, it refused to publish anything at all.

We trusted stakeholders to define critical business rules that would test for major problems. They trusted us to implement their logic with every refresh. We trusted each other not to blame and shame, but to steadily build out better and better testing to benefit everyone.

Now, together, we had protected leadership from a serious integrity issue. We had prevented a flood of inaccurate reports from reaching decision makers. And when we explained what happened, why the data was late and what we had caught, they weren't frustrated. They were grateful.

The value of data quality is often invisible. When things go right, no one notices. But this was a moment where things went wrong *exactly* the way we hoped they would: **the system raised a red flag, we caught the issue before it caused damage, and we preserved the integrity of our analytics.**

The takeaway? If you're not building comprehensive, automated data quality checks into your production pipelines, you're one bad refresh away from losing stakeholder confidence and risking devastating business outcomes.

Your data processes shouldn't just be fast, they should be right. And sometimes, the best outcome is one where the system *refuses* to run.

The Economics of Data Quality: When to Test, How Much to Spend

Data quality is not free, and neither is the absence of it. Every data leader eventually faces the same question: how much should we spend, and how do we justify it? This chapter gives you a practical framework for making that case, setting a realistic budget, and knowing when heavy investment is not warranted.

The 1:10:100 Rule and Its Limits

The 1:10:100 rule has been circulating in data circles for decades. The premise: fixing a data error at the source costs 1 unit of effort. Fixing it downstream after it has propagated costs 10. Fixing it after it has reached customers or driven a bad business decision costs 100.

The rule is useful as a conversation starter with executives. It creates an intuitive argument for investing in upstream data quality rather than reactive cleanup. But on its own, it is not enough.

The problem is that 1:10:100 is dimensionless. It tells you that prevention is cheaper than remediation but gives no guidance on how much to spend on prevention. Applied literally, it implies you should invest in quality up front as long as the cost is less than one-tenth of the expected remediation cost. But it says nothing about the probability of failure, the frequency of incidents, or the business impact of any specific table.

The rule also varies significantly by organization type. Consider three archetypes:

- A regulated financial services firm may face regulatory fines for data errors, making the downstream cost of a single incident easily $1 million or more. For this organization, the 100 in 1:10:100 is real and measurable.
- A startup analytics team with 15 people and a small pipeline primarily suffers the cost of engineer time and delayed decisions. Their "100" might be $5,000. Heavy tooling investment would be disproportionate.
- A retailer running personalization and pricing algorithms on live customer data sits somewhere in between: bad data affects revenue directly, and the cost accumulates quietly across millions of transactions.

Use the 1:10:100 rule to open the conversation, then replace it with numbers specific to your organization.

Quantifying the Cost of Data Downtime

The best way to justify a data quality program is to put a dollar figure on what happens when data is wrong or unavailable. Most organizations have never done this calculation explicitly, which is why data quality budgets remain vague.

Here is a three-step framework.

Step 1: Identify your most-used data assets.

Start by listing the 10 to 15 data assets (tables, pipelines, reports, or feeds) that the business depends on most. These are the assets where downtime or data errors would cause an immediate, visible problem. Good candidates include: the revenue dashboard used by the CFO each morning, the customer data feed that drives the CRM, the attribution model that informs marketing spend, and the operational tables that power customer-facing features.

For each asset, identify who uses it, what decisions it drives, and what happens if it is wrong or unavailable for one hour.

Step 2: Estimate the cost of one hour of downtime for each asset.

"Cost of downtime" has several components:

- Lost engineer time: who is pulled in to investigate and fix the problem, and for how long? A senior data engineer at $150k per year costs roughly $75 per hour fully loaded. If a P1 incident pulls in three engineers for four hours each, that is $900 in engineering cost before anything else.
- Lost analyst or business user time: if a revenue report is unavailable for two hours during a Monday morning business review, what is the cost of delayed decisions or the time spent in manual workarounds?
- Direct revenue impact: for data assets that drive automated decisions (pricing, recommendations, fraud detection), errors can have immediate revenue consequences.

- Trust cost: harder to quantify but real. Every time the data is wrong, business users trust it less. At some point they build their own spreadsheets, and the data team loses influence.

A reasonable shorthand: estimate the hourly value of the decisions driven by each asset, then discount by the severity of a typical failure. If a dashboard drives $2M per quarter in budget allocation decisions and it fails completely once, the cost of a bad quarter is roughly $500k in misdirected spend. If failures are partial (data is a day stale rather than completely absent), discount that figure significantly.

Step 3: Multiply by estimated annual downtime hours.

If you have incident records, use them. If not, use industry estimates: mature data teams report roughly 8 to 15 hours of material data quality incidents per year per critical pipeline. Immature teams report 40 to 80 hours or more.

Here is a worked example for a 50-person analytics organization.

Asset	Users	Cost per Hour Down	Estimated Hours Down/Year	Annual Cost of Downtime
Revenue dashboard	CFO, 5 VPs	$2,000	12	$24,000
Customer 360 table	Sales, CRM	$1,500	20	$30,000
Marketing attribution feed	Marketing team	$1,000	15	$15,000
Fulfillment ops pipeline	Operations	$3,000	10	$30,000
Finance close tables	Finance	$2,500	8	$20,000
Total				**$119,000**

This table is the foundation of your ROI argument. The $119,000 figure represents the estimated annual cost of data quality failures before any investment in prevention. Now you have something to work with.

Building a Simple ROI Model

With a downtime cost estimate in hand, you can build a basic ROI model. The structure is straightforward:

ROI = (Prevented losses) - (Cost of the program)

The cost of a data quality program for a mid-sized analytics team typically falls in the range of $50,000 to $200,000 per year. The wide range reflects tooling choices and team structure:

- A lightweight program using open-source tools (dbt tests, custom SQL checks) and a part-time owner might cost $50,000 to $75,000, mostly staff time.
- A mid-range program with a commercial observability tool, a dedicated data quality engineer, and regular process overhead lands around $100,000 to $150,000.
- A full program with enterprise tooling, multiple engineers, formal data contracts, and executive reporting runs $150,000 to $200,000 or more.

Returning to the worked example: if the organization estimates $119,000 per year in data downtime costs, and a data quality program at the $100,000 midpoint prevents 60% of those incidents, the math looks like this:

- Prevented losses: $119,000 x 0.60 = $71,400
- Program cost: $100,000
- Net in year one: -$28,600 (negative, because you are still building)
- Year two, assuming the same prevention rate and lower setup costs: $71,400 - $80,000 = -$8,600
- Year three, with better coverage and the team running efficiently: $71,400 - $75,000 = roughly breakeven

This model will not satisfy a CFO on its own, but it changes the conversation. You are no longer asking for money with no return. You are presenting a cost-reduction argument with a projected payback period of two to three years. For organizations with higher downtime costs or more aggressive incident rates, the payback period can be under 12 months.

To strengthen the model, add the cost of trust erosion and analyst time spent on manual workarounds. These are often larger than the direct incident costs.

Realistic Budget Benchmarks

There is limited published research on data quality spending as a percentage of overall data org budgets, but practitioners report a consistent range. Most data teams that have a functioning data quality practice spend somewhere between 10% and 20% of their total data organization budget on quality-related activities, including tooling, staff time for testing and monitoring, incident response, and process overhead.

For a 50-person analytics organization with a total staff and infrastructure budget of $5M, that suggests a data quality investment of $500,000 to $1M. That sounds high until you factor in that every senior engineer spends time on data incidents, every analyst builds workarounds, and every business review includes some version of "can we trust this number?"

A more practical starting point for organizations without an established practice: target 10% of data org spend in year one, focused on the highest-risk assets. Expand from there based on demonstrated value.

What a First-Year Program Costs

Breaking down the first-year investment helps make the budget concrete and defensible.

Cost Category	Low Estimate	High Estimate	Notes
Staff: 0.5 FTE data engineer focused on DQ	$60,000	$90,000	Salary plus benefits, partial allocation
Tooling: open source (dbt, SQL checks)	$0	$5,000	Hosting and infrastructure only

Cost Category	Low Estimate	High Estimate	Notes
Tooling: commercial observability platform	$20,000	$60,000	Monte Carlo, Bigeye, Soda, or similar
Process overhead: stakeholder meetings, documentation	$5,000	$15,000	Time cost of coordination
Training and onboarding	$2,000	$8,000	Conferences, courses, vendor training
Total	**$87,000**	**$178,000**	

The biggest lever is tooling choice. Starting with open-source tools and a single part-time owner is reasonable for most mid-sized teams. Expand to commercial tooling after you have proven the value and understand what you need.

When Not to Invest Heavily

Not every team needs a formal data quality program. Over-investing in quality infrastructure before the organization is ready wastes money and creates process burden that nobody will follow.

Consider staying lightweight if:

- Your team has fewer than five data engineers or analysts. At this scale, good code review and simple row count checks are often sufficient.
- Your data is primarily used for exploration or ad hoc analysis, not automated decisions or executive reporting. Exploratory work tolerates lower data quality.
- Your pipelines are early-stage and changing rapidly. Heavy test suites slow down iteration and become stale within weeks when the underlying data model shifts.
- Your business decisions are not meaningfully driven by data yet. If leadership rarely uses the data and is not yet relying on it for significant choices, the cost of bad data is low.
- You are pre-product market fit. Engineering resources spent on data quality infrastructure before you have stable data models are resources not spent on the product.

The signal to start investing more seriously is when any of these conditions change: the data team grows beyond five, automated decisions start relying on the data, the CFO starts asking about dashboard reliability, or the first major data incident causes significant business pain.

The Practical Takeaway

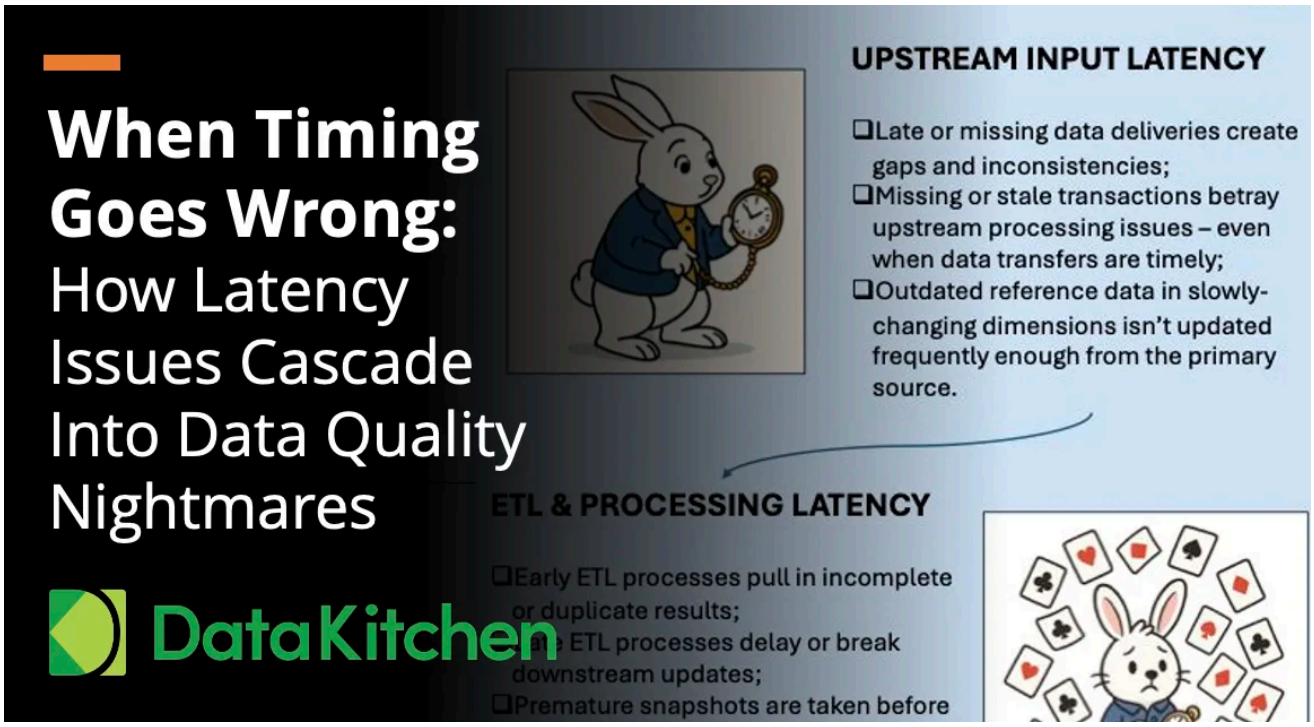
The case for data quality investment is not built on the 1:10:100 rule alone. It is built on a specific estimate of what bad data costs your organization, a realistic projection of what a quality program can prevent, and a clear-eyed view of what the program will cost in its first year.

Start with the three-step downtime cost framework. Use those numbers to build a simple ROI model. Set a first-year budget in the $75,000 to $150,000 range for most mid-sized teams. And recognize that the goal in year one is not perfection. The goal is to stop the most expensive incidents, create a feedback loop for improvement, and build the organizational muscle to operate more reliably over time.

WHY DATA QUALITY MATTERS

When Timing Goes Wrong: How Latency Issues Cascade

Read the original article at: https://datakitchen.io/blog/when-timing-goes-wrong-how-latency-issues-cascade/



As data engineers, we've all been there. A dashboard shows anomalous metrics, a machine learning model starts producing bizarre predictions, or stakeholders complain about inconsistent reports. We dive deep into data validation, check our transformations, and examine our schemas, only to discover the real culprit was something far more subtle: timing.

Process latency - the delays and misalignments when data moves through our systems - is one of the most underestimated threats to data quality in modern architectures. While we've gotten excellent at building robust pipelines and implementing data quality checks, we often treat timing as someone else's problem. This is a dangerous oversight.

The Hidden Nature of Latency Problems

Unlike schema violations or null value issues that surface immediately in our monitoring dashboards, latency problems are insidious. They don't break your pipelines - they corrupt your insights. A report might run successfully every morning, but if it's pulling yesterday's transactions against today's customer dimensions, you're serving up beautifully formatted garbage.

The challenge intensifies because latency issues rarely manifest as obvious timing problems. Instead, they masquerade as:

- **Data inconsistencies** when upstream sources haven't synchronized properly
- **Missing records** that appear to be data loss, but are delayed deliveries
- **Stale dimensions** that make fact tables look wrong
- **Integration conflicts** between systems operating on different schedules

The Four Faces of Latency-Induced Data Quality Issues

Upstream Input Latency

This is where data quality problems often begin. When source systems deliver data late or inconsistently, it creates gaps that propagate downstream. Missing transactions, stale reference data, and delayed dimension updates all stem from this root cause. Your perfectly architected data warehouse becomes unreliable not because of design flaws, but because the data it depends on arrives unpredictably.

ETL & Processing Latency

Even when source data arrives on time, processing delays create their own quality issues. Early ETL jobs might pull incomplete datasets, while late-running processes create cascading delays throughout your pipeline. Premature snapshots capture partial states, and overlapping batch windows intermingle different versions of the same data, creating temporal chaos.

Integration Drift

The most complex challenge emerges when different systems maintain separate versions of shared entities. Customer records updated in your CRM don't immediately reflect in your analytics warehouse. Product catalogs in your e-commerce platform are updated, while your recommendation engine still references the old version. These synchronization gaps create a fragmented view of reality across your data ecosystem.

Delivery & Availability Gaps

The final manifestation occurs when downstream consumers expect fresh data that isn't available. Reports are run on schedule, but they reflect outdated information. Machine learning models retrain on outdated features. Business users make decisions based on metrics that are hours or days behind reality, and they often remain unaware of this discrepancy.

Why We Ignore the Timing Problem

There's a reason latency issues persist despite their impact: they're fundamentally collaboration problems disguised as technical ones. It's much easier to focus on what's within our direct control - our schemas, our transformations, our infrastructure - than to coordinate schedules across teams, systems, and vendors.

We build our monitoring around what we can measure easily: pipeline success rates, data volumes, and schema compliance. But measuring cross-system timing dependencies? That requires visibility into other teams' processes, understanding of business logic we didn't design, and ongoing coordination that extends far beyond our immediate technical domain.

The Data Mesh Dilemma

As organizations embrace data mesh architectures, latency challenges multiply exponentially. When data ownership becomes distributed across multiple product teams, timing coordination becomes everyone's responsibility, which often means it becomes no one's responsibility.

Each domain team optimizes its data products independently. Sales refreshes customer segments nightly, marketing updates campaign performance hourly, and finance reconciles transactions on a weekly basis. These decisions make perfect sense within each domain, but the interactions between these schedules create systematic quality issues that no single team owns or can easily detect.

Taking Ownership of Time

The solution isn't to abandon modern data architectures, but to explicitly own the timing aspects of data quality. This means:

Treating schedules as first-class design artifacts. Document not just what data moves where, but when it moves and what depends on that timing. Make schedule dependencies as visible as data lineage.

Building latency into your data quality framework. Monitor not just whether data arrived, but whether it arrived when expected relative to its dependencies. Alert on timing anomalies before they become data quality incidents.

Establishing cross-domain timing contracts. In data mesh implementations, service level agreements should cover temporal aspects: not just "we'll provide customer data," but "we'll provide customer data by 2 AM EST, reflecting all transactions processed by midnight." Make these explicit as part of your Data Journey.

Designing for timing failures. Build graceful degradation into your systems. When fresh data isn't available, how does your system behave? Can it serve stale data with appropriate caveats rather than serving incorrect data silently?

Beyond Technical Solutions

Ultimately, addressing latency-induced data quality issues requires us to expand our definition of data engineering beyond purely technical concerns. We need to become coordinators of temporal complexity, building systems that not only move data efficiently but also do so predictably and transparently.

This means having uncomfortable conversations about cross-team dependencies. It means pushing back when unrealistic timing expectations create quality trade-offs. And it means designing systems that make timing problems visible rather than hiding them behind successful pipeline metrics.

The next time you're debugging a mysterious data quality issue, don't just look at the data - look at the clock. The problem might not be in your transformations or your infrastructure. It might be in time itself.

Timing issues in data systems are often the invisible root cause of visible data quality problems. By treating process latency as a first-class concern in our data architecture, we can build more reliable systems and prevent quality issues before they impact business decisions.

WHY DATA QUALITY MATTERS

We Got Roasted on Reddit for Asking Why Data Engineers Don't Test

Read the original article at: https://datakitchen.io/blog/we-got-roasted-on-reddit-for-asking-why-data-engineers-dont-test/

How This Research Started (And Why We're Telling You)

DataKitchen co-founder Gil Benghiat recently posted a question to r/dataengineering , one of the largest and most active data engineering communities on the internet. The question was genuine: "**What is actually stopping teams from writing more data tests?"**

The problem? He didn't disclose who he was. Within hours, a lead data engineer named SearchAtlantis spotted that just one week earlier, Gil had posted in a separate thread and explicitly said: "full disclosure, I co-founded DataKitchen and we built TestGen for exactly this problem." The callout got 102 upvotes. The replies were swift and merciless. "His picture tells a thousand words," wrote another commenter. The community, as it tends to do, did not let it slide.

To Gil's credit, he acknowledged it directly: "That is fair. I co-founded DataKitchen and should have said so here. The question is still genuine, though."

And it was genuine. The thread went on to generate over 60 thoughtful comments from working data engineers, and the insights that emerged are some of the most candid and practically useful we've seen on the topic. We're sharing them here , with full transparency about their source.

What's Actually Stopping Teams From Writing Data Tests?

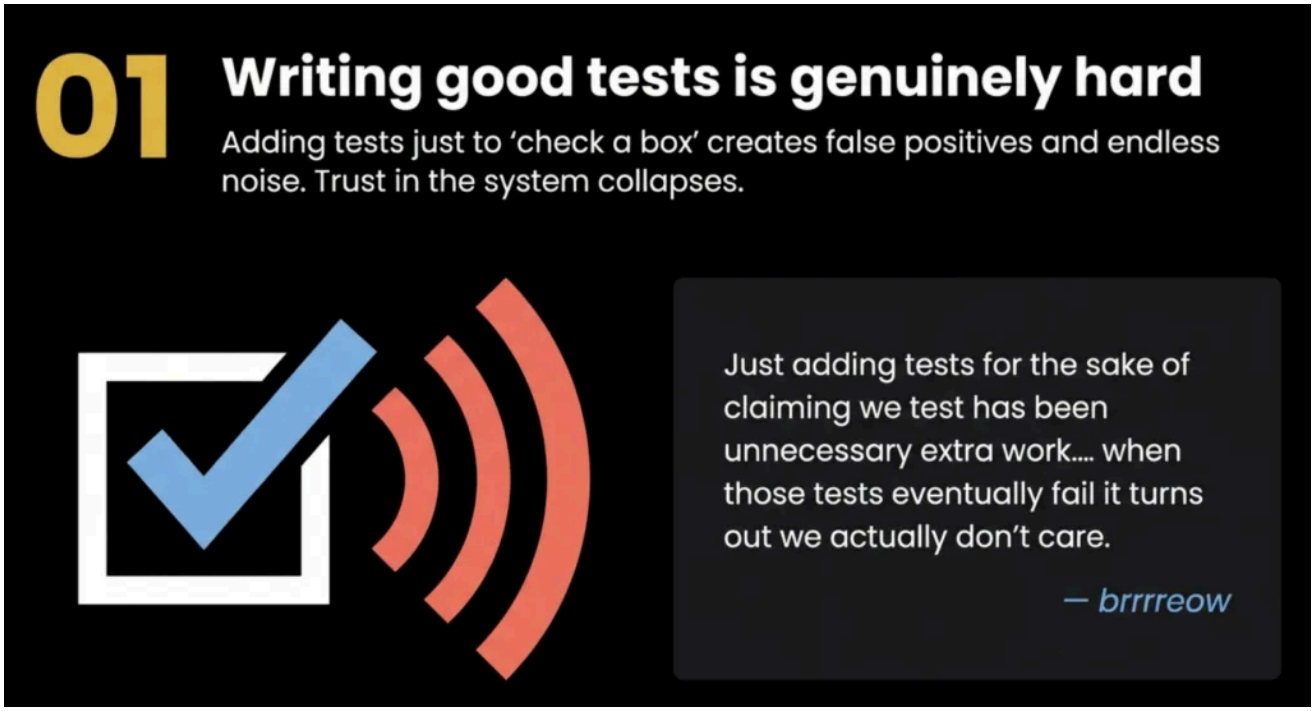
1. WritingGoodTests Is Genuinely Hard

The most upvoted response in the entire thread cut straight to the heart of it: writing tests that catch real bugs without being noisy is just difficult. This isn't false modesty , it's a real skill gap that takes time and painful experience to develop.

The problem isn't the mechanics of writing a test. It's knowing which tests are worth writing. Teams that skip this step and add tests just to "check a box" often end up worse off: tests that fire on irrelevant conditions, break jobs for illegitimate reasons, and eventually get deleted. Bad tests erode trust in the whole testing practice. Once engineers stop believing their alerts mean something, the entire system collapses.

One engineer described the downstream effect plainly: "I tried implementing a basic testing practice at my workplace… my manager insisted we need some version of unit testing on every table too as part of the policy. Just adding tests for the sake of claiming we test has been unnecessary extra work… then when those tests eventually fail and break a job it turns out we actually don't care or the dev never verified the rule was legitimate in the first place."

The compounding problem: DAG timing makes this worse. When pipelines have multiple arms refreshing on different cadences, false positives multiply. Engineers start ignoring alerts. And once alerts get ignored, you've lost the whole point.



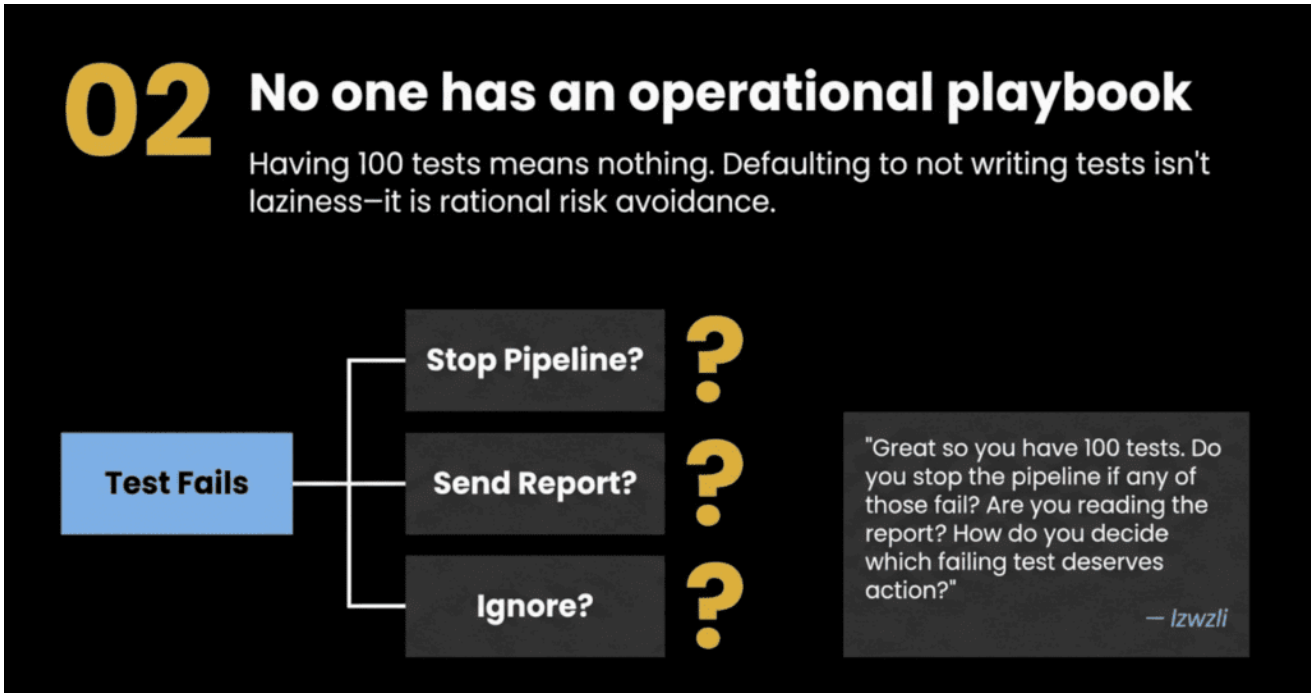
2. No One Has Thought Through What Happens When a Test Fails

One of the most insightful comments in the thread reframed the entire problem: “Writing tests isn’t the hard part. It’s deciding the operational playbook for each of the tests that gets tricky.”

- You have 100 tests. Which ones stop the pipeline when they fail?
- For those who don’t stop the pipeline, what do you do with the results?
- Is anyone actually reading the report sent for every pipeline run?
- How do you decide which non-critical failing test deserves action?

Without answers to these questions, tests become noise. And in the absence of a clear playbook, teams default to not writing them at all. It’s not laziness , it’s rational risk avoidance.

Gil’s own response in the thread is worth noting here: “Not all tests are equal. Some should stop the pipeline, some should be warnings that are investigated later or documented as release notes for data consumers, and others are more metric tracking than tests , a way for the data engineer to stay calibrated on how the data behaves. If a test is constantly failing and nobody acts on it, then remove it.”



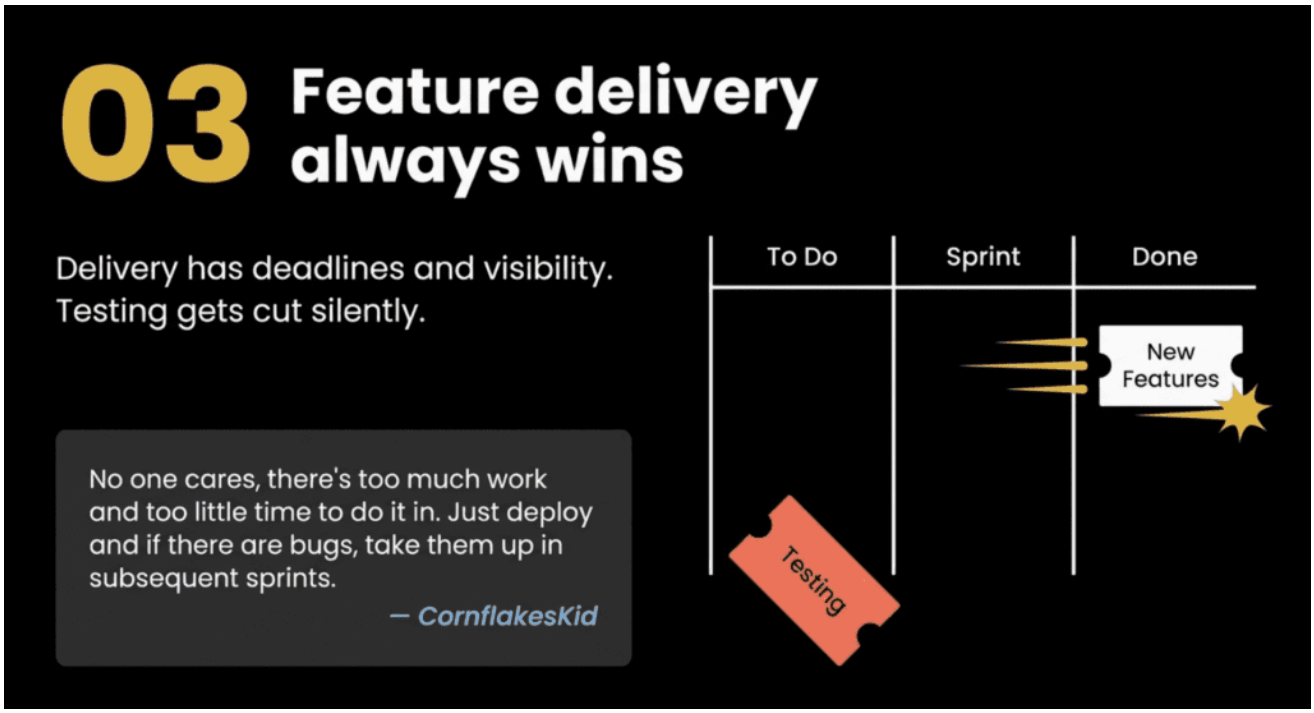
3. Feature Delivery Always Wins

At least four separate commenters called this out directly, and it’s probably the most universal barrier on the list. During a sprint, tests lose. Every time.

The dynamic is simple: delivery has a deadline, stakeholders, and visibility. Testing has none of those things , until something breaks in production and suddenly it has all three.

One senior data engineer summed it up bluntly: “From what I’ve seen, it is more of that last one. No one cares, there’s too much work and too little time to do it in. Just deploy and if there are bugs, take them up in subsequent sprints.”

This isn't just an individual failure. It's an organizational one. When teams are too small for the workload, testing is the first thing to get cut. And data testing gets cut silently, without anyone explicitly making that decision.



4. Tests Rot , And Nobody Budgets Time to Maintain Them

Even teams that invest in testing often find that investment decays. Upstream schemas change. Business logic shifts. A vendor updates their API without warning. Tests that were accurate six months ago now fire constantly , or, worse, silently pass bad data because the world has changed around them. One commenter described it well: "a lot of teams don't have stable expectations, schemas and upstream logic shift, so tests rot fast unless you budget time to maintain them."

Maintenance rarely gets prioritized. It doesn't appear on the roadmap, doesn't get sprint points, and doesn't have a deadline. So tests slowly rot, confidence erodes, and eventually someone deletes the whole suite and starts over , or doesn't bother starting over at all.

One commenter captured the fatigue well: teams that have been burned by upstream changes enough times learn to write schema validation and bail early rather than invest in elaborate fixtures, because fixtures require their own maintenance, which requires its own tests , a matryoshka doll of testing infrastructure that most teams don't have the bandwidth to sustain.

5. You Don't Know What to Test Until It's Already Broken

Data testing knowledge is largely reactive. Most engineers learn what to test by getting burned , a pipeline that produced zero rows, a join that silently dropped 40% of records, a schema change from upstream that nobody announced.

Until you've lived through a few of those, it's hard to know where the landmines are. And without that experiential knowledge, tests tend to be shallow , row counts, maybe nulls , and miss the things that actually matter.

One commenter put it plainly: "Lack of experience. Those of us who have been burned like you were with that four-hour pipeline that did nothing write data tests." Another echoed it: "You don't realize you need them until you've scrambled to get shit fixed cuz things are broken and the heat is on and it's embarrassing. That's when you realize oh wow it doesn't have to be like this."

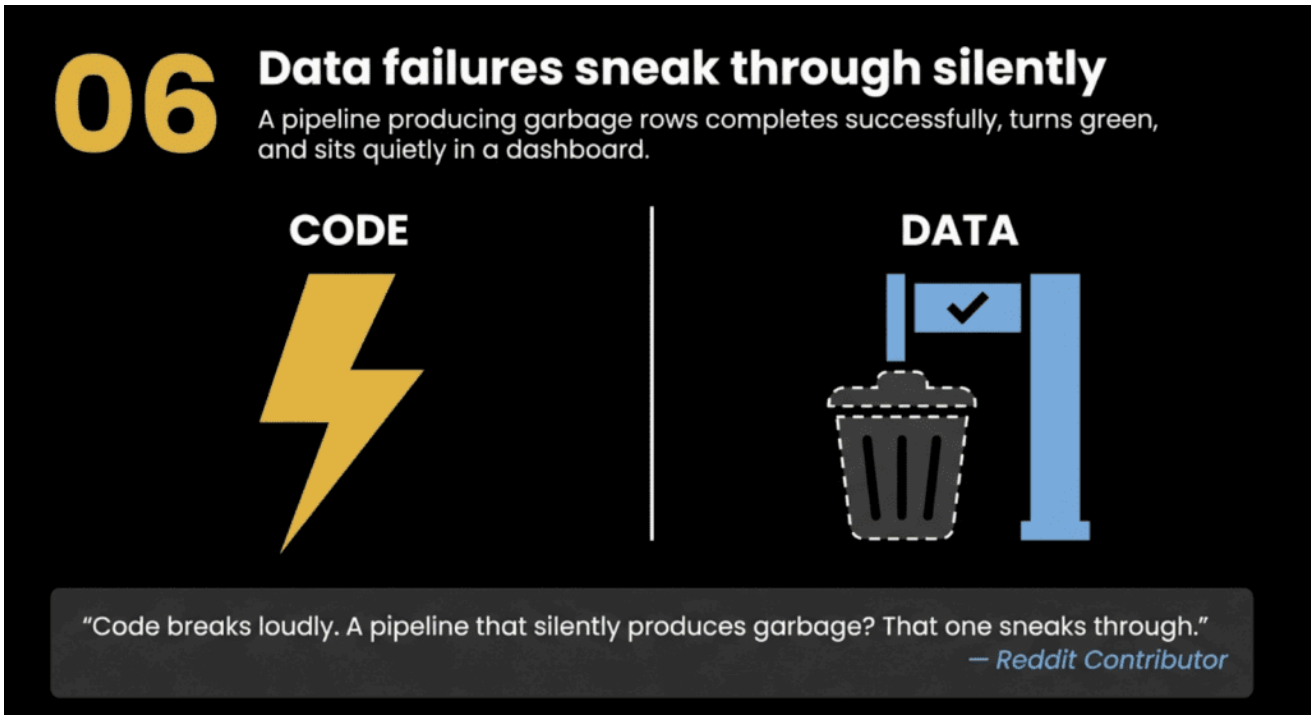
The deeper version of this problem is domain knowledge. One commenter observed that they'd seen teams with hundreds of tests still miss "revenue went to 0″, because nobody on the data team understood the business well enough to recognize that it was a possible failure mode worth testing for. As Gil put it in response: "Many data teams just move data and don't understand the business." Technical competence and business context are both required, and the latter is often missing.

6. Nobody Senior Enough Cares , Until It's Too Late

Code breaks loudly. A build fails, a deploy errors out, and an exception gets thrown. Data failures are different. A pipeline that produces garbage rows completes successfully. The job turns green. No alert fires. The wrong numbers sit quietly in a dashboard until someone notices them in a meeting , usually the wrong meeting, at the wrong time, in front of the wrong people.

One commenter nailed the asymmetry: "Code breaks loudly. A pipeline that silently produces garbage? That one sneaks through."

Because data failures are invisible until they're embarrassing, there's rarely organizational pressure to prevent them in advance. Leadership doesn't feel the cost of missing tests until a crisis makes it undeniable. And by then, the damage is done.

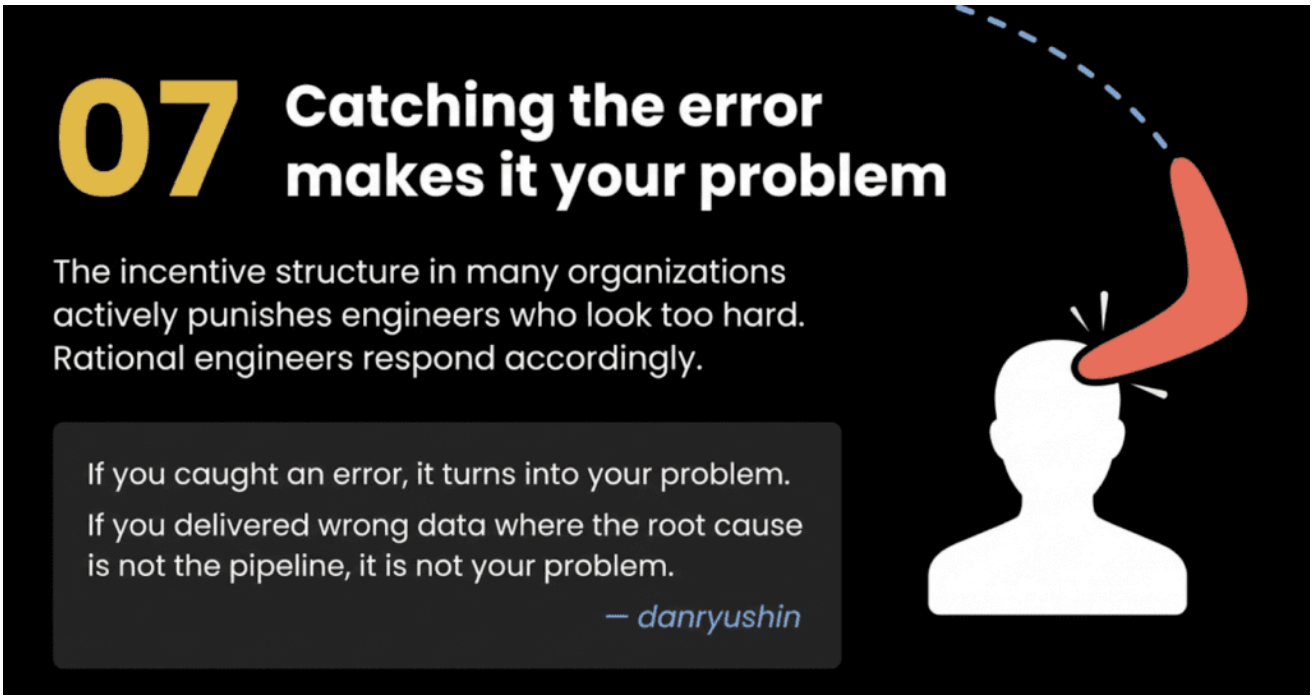


7. Catching the Error Makes It Your Problem

This one is uncomfortable but widely recognized. One commenter articulated it clearly: "If you caught an error, it turns into your problem. If you delivered wrong data where the root cause is not the pipeline, it is not your problem."

The incentive structure in many organizations actively punishes engineers who look too hard. Finding a problem means owning the investigation, the fix, the communication to stakeholders, and the post-mortem. Not finding it means the problem is the upstream's fault.

Gil's take: the only fix is leadership that actively rewards people who surface problems rather than penalizing them for finding them. This is ultimately a culture-and-values issue, not a tooling one.



8. At Scale, Comprehensive Testing Must Fit

One commenter described processing close to an exabyte of data per day. At that volume, full validation at every stage is too slow, too expensive, and practically impossible. Their solution , sampling, anomaly detection, and back-testing on pipeline deployments , is the pragmatic answer to a genuine constraint. As they put it: “At our volume something that is 1 in a billion will happen several times a day, so if it can happen, it will happen , plan for the possibility.”

This is worth naming clearly: for some teams, the barrier to comprehensive testing isn’t skill or culture. It’s physics and economics. The right answer at hyperscale looks very different from the right answer at a mid-market company running 50 pipelines.

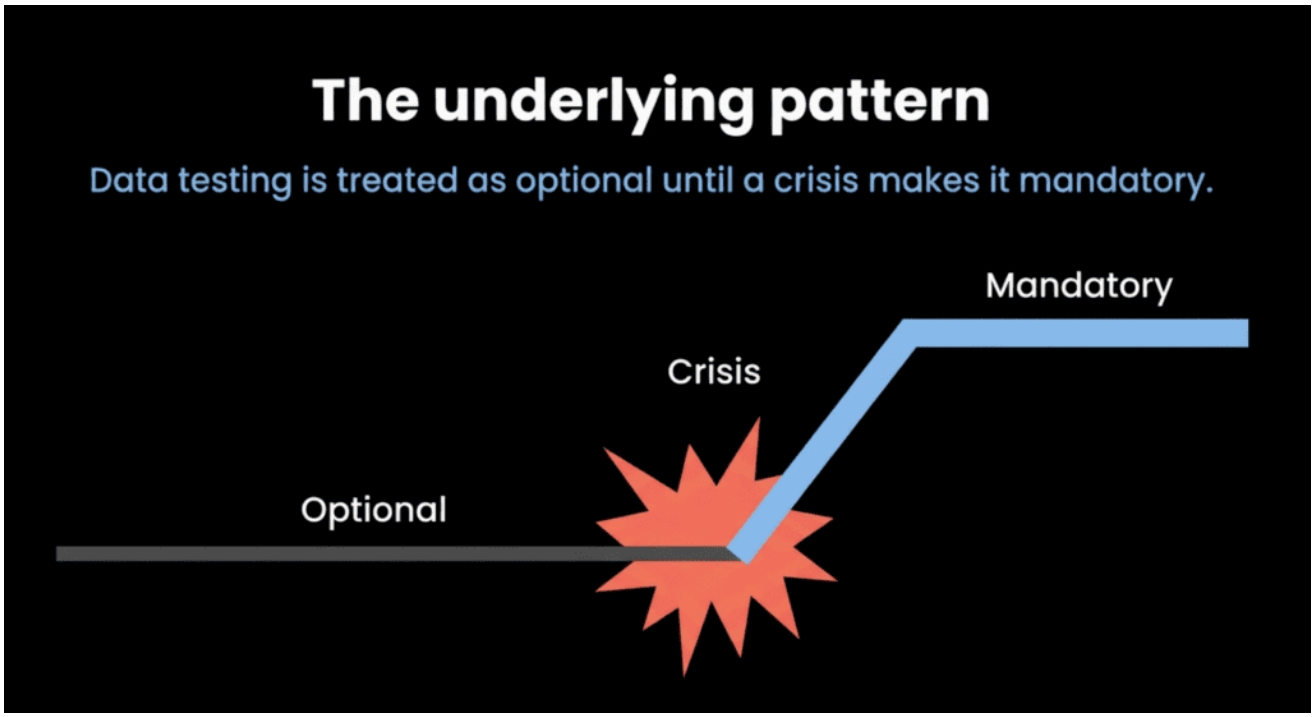
The Pattern Underneath All of It

Across every comment in the thread, one pattern emerges consistently: data testing is treated as optional until a crisis makes it mandatory.

Most teams build test coverage reactively , one incident at a time , rather than proactively as a standard engineering discipline. The barriers are a mix of technical difficulty, organizational culture, misaligned incentives, and resource constraints. Tooling is rarely the limiting factor.

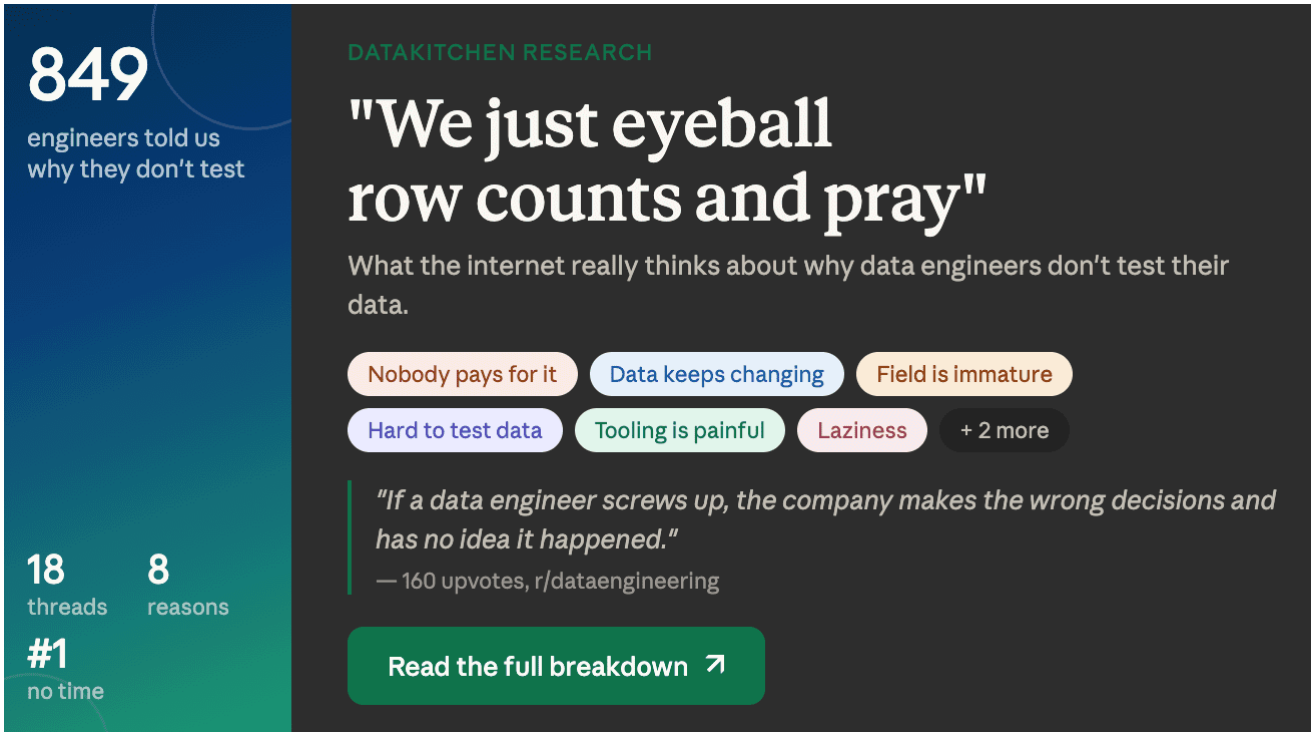
- Test noise , false positives erode trust in the entire suite
- Time pressure , feature delivery consistently wins over quality investment. No one has time to write tests.
- Scale , at high volume, writing and maintaining comprehensive testing becomes impractical
- Test rot , upstream changes break tests, and maintenance is never prioritized
- Perverse incentive , catching errors makes them your problem
- Domain knowledge gaps , without business context, it’s hard to know which tests actually matter

These aren’t problems that disappear when you hand a team a testing tool. They require investment in process, culture, and clear operational norms around what testing means and what happens when it fails.


The underlying pattern
Data testing is treated as optional until a crisis makes it mandatory.
Mandatory
Crisis
Optional

We Just Eyeball Row Counts and Pray

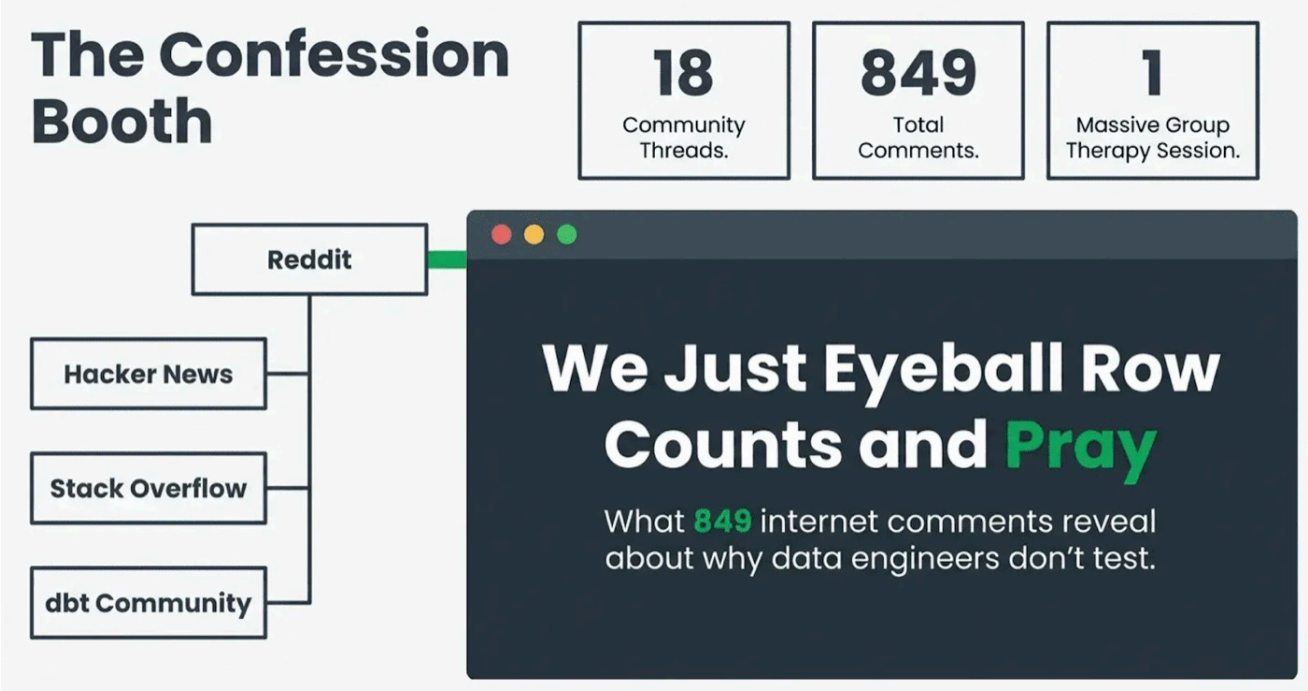
Read the original article at: https://datakitchen.io/blog/we-just-eyeball-row-counts-and-pray/



What Internet Communities Really Think About Why Data Engineers Don't Test

The fastest way to get a data engineer to type a thousand words is to ask them why they don't write tests.

That is what happened across a series of threads on Reddit, Hacker News, Stack Overflow, and the dbt Community Forum over the past couple of years. The conversations drew hundreds of comments from working data engineers, and what emerged was not a polished conference talk on best practices. It was something far more honest and far more useful: a messy, funny, occasionally brutal group therapy session about the state of quality in data engineering.



A few weeks ago, Gil asked a question, 'Why Data Engineers Don't Test?' and got 50 answers (and was roasted for having ulterior motives). But what if we expand our search? So we asked Claude AI to read every comment across 18 of these threads, for a total of 849 comments. What follows is what the community actually said, in their own words, about why data engineers do not test their data, and what they think might fix it.

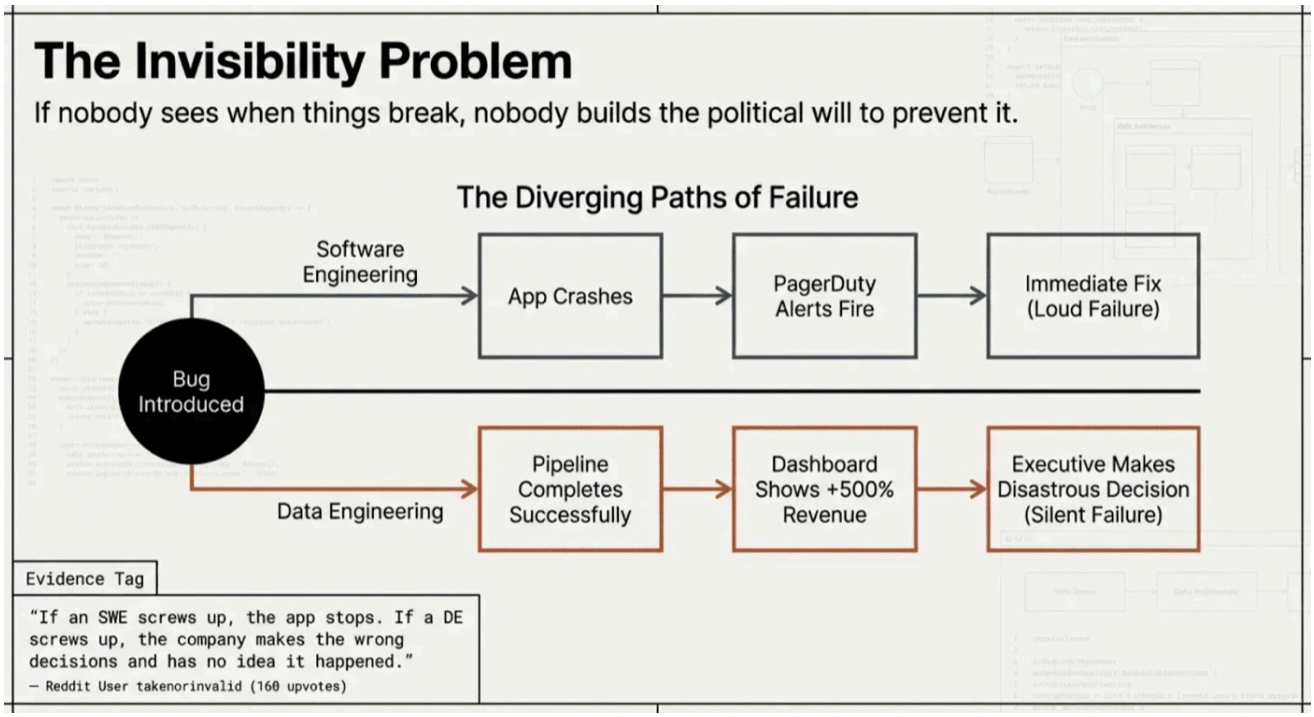
The Confession Booth

The most upvoted comment in the first Reddit thread set the tone immediately. User takenorinvalid, scoring 160 points, wrote: *"What we do is have no QA framework in place, not realize the data is wrong for months or even years, and then blame each other when it comes out."*

That is not someone being flippant. That is someone describing Tuesday.

The same commenter articulated what might be the single most important insight in all eighteen threads: *"Data Engineering is invisible. If a software engineer screws up, the app stops working, and everybody knows it. If a data engineer screws up, the company makes the wrong decisions and has no idea it happened."*

This observation echoed across every platform. On Hacker News, a commenter noted that *"the successful operation of a pipeline doesn't conclusively determine whether the data is actually correct for the end user."* On Reddit, doryllis put it more bluntly: *"The biggest danger in data engineering is that no matter how wrong your query or data, if you write it so it works, it returns results. Results, but not necessarily correct ones."*



The invisibility problem is not just philosophical. It directly explains why testing gets deprioritized. If nobody can see when things are broken, nobody builds the political will to prevent breakage. On the "There is no data engineering roadmap" Hacker News thread, Simon_O_Rourke nailed this: *"if you do an absolutely superb job you end up being largely invisible to the business. Like the very necessary guys who fix the water mains or the sewers, the expectation is that the service/data pipeline/database will be always available, and nobody really cares too much either way once everything's working as expected."* And steveBK123 added the economic punchline: *"DQ is a big part of it as well. Everyone wants it, no one wants to pay the hours/dollars to do it. No one is sure what they want, but probably what you propose is too simple. Can't AI do it? Etc."*

As user trentsiggy observed on Reddit: *"Testing should be non-negotiable. It only becomes negotiable in immature environments."* And then, with the weary realism of someone who has seen a few things: *"Sometimes, you have single DEs or a tiny team of them at a company that has drastically underfunded their data needs who are trying to string 40 exec level demands together with bailing twine and duct tape, with an exec team that fires anyone who pushes back against it."*

Data Engineering Is Not Software Engineering, and That Is the Setup

Before we get into the ranked reasons, it helps to understand a point the community made forcefully across every thread: data engineering is simply not software engineering, and trying to port SWE testing practices directly into the data world is a category error.

Data Engineering is NOT Software Engineering.

The Category Error Matrix

	Software Engineering	Data Engineering
Environment	Controls the stack & interfaces.	Relies completely on volatile 3rd-party inputs.
Testing Scale	Uses small, local mocks.	Requires standing up massive, distributed infrastructure.
Cadence	Code syncs via version control.	Data changes independently of CI/CD pipelines.
Correctness	Deductive and axiomatic.	Fuzzy, defined simply by "no complaints".

Evidence Tag

"Fundamentally data engineering is not software engineering. They are at best cousins, not brothers, and definitely not twins."

User naijaboiler, with 45 upvotes, put it plainly: *"fundamentally data engineering is not software engineering. They are at best cousins, not brothers, and definitely not twins."* MonochromeDinosaur captured the asymmetry: *"Testing in SWE is easier because you usually control most of the stack and interfaces. Third party integrations/APIs usually respect their contracts more when it's webdev related. When you need fixtures and mocks they're relatively small."* And on Hacker News, azirale painted the full picture: *"Imagine if you wanted to test the inputs and outputs of a function you had to stand up an entire infra stack, run the full application, push significant data into it, then run your test."*

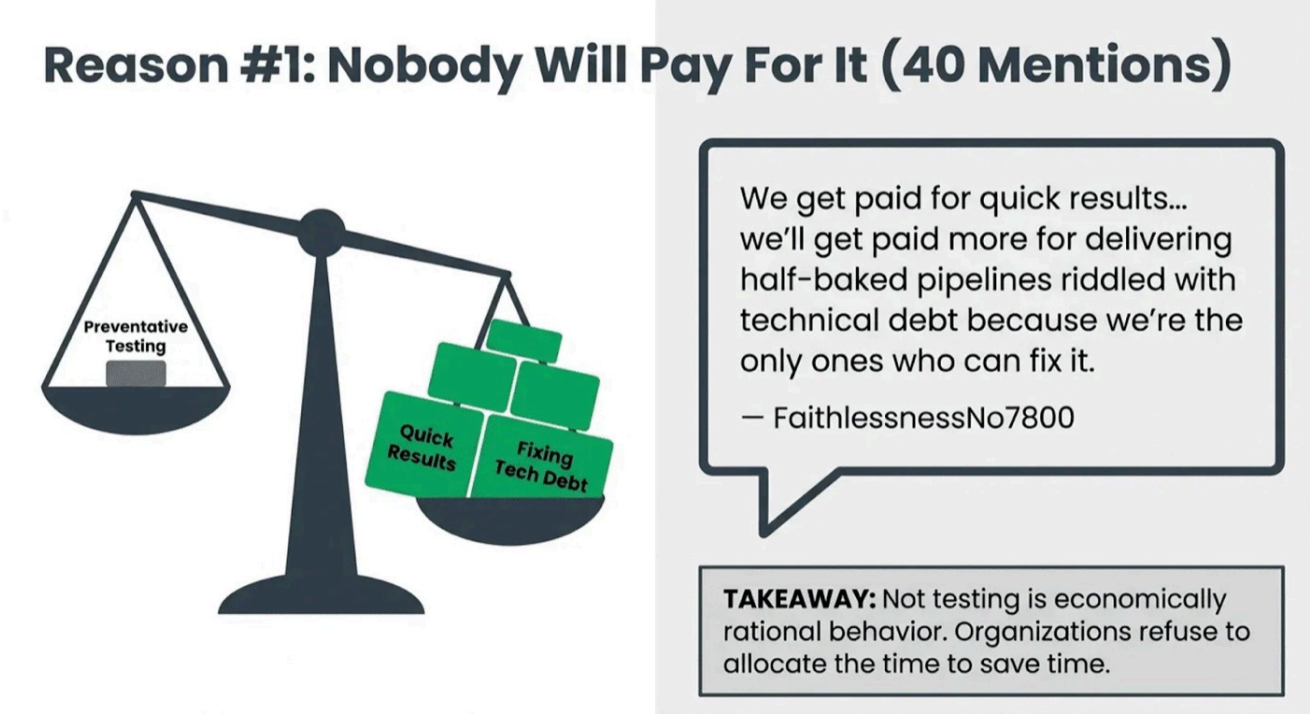
That is the context. Now, here is what data engineers say when you ask them specifically why they do not test.

The Eight Reasons, Ranked

Reading across all eighteen sources, the reasons data engineers do not test cluster into a remarkably consistent set of themes. We tagged every comment that cited a reason, counted each comment at most once per reason, and ranked them. The ordering was not subtle.

#1. Nobody will pay for it (40 mentions). This was the runaway winner, and it was not even close. When data engineers explain why they do not test, the answer most often is not technical. It is organizational.

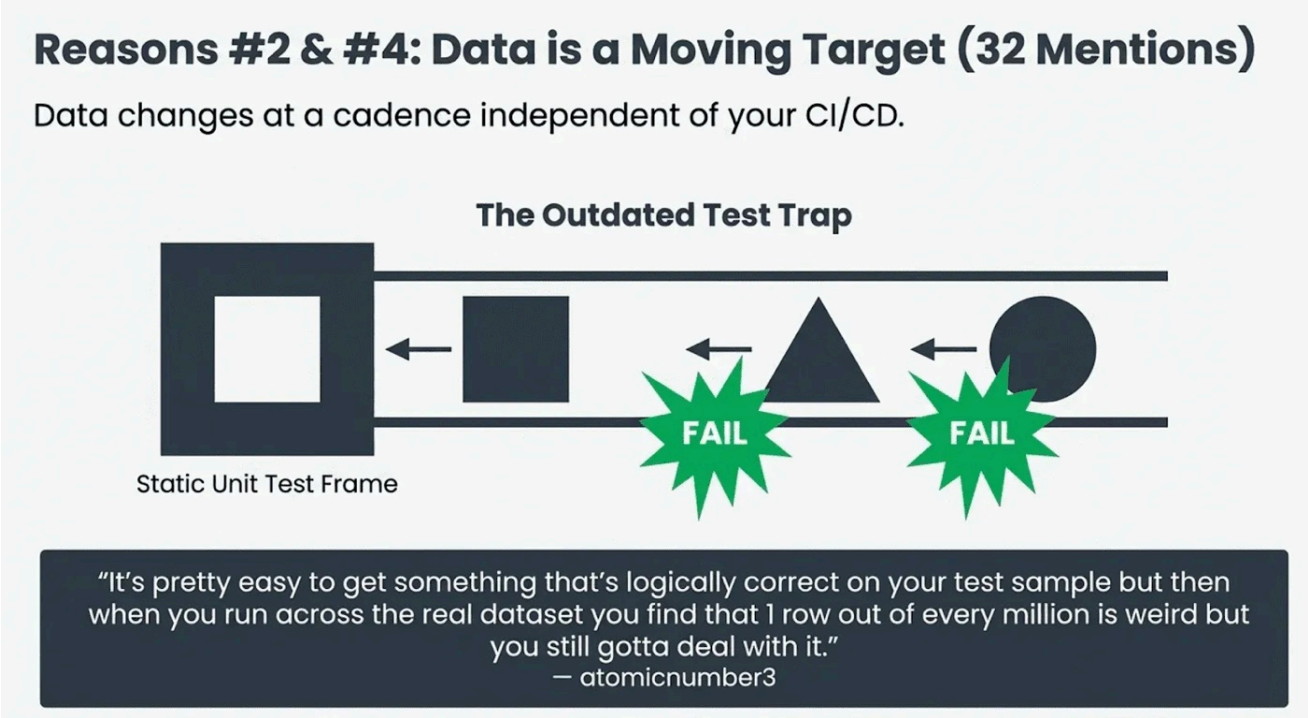
FaithlessnessNo7800 was brutally honest: *"we get paid for quick results, not well-developed results. In fact, we'll get paid more for delivering half-baked pipelines riddled with technical debt because we're the only ones who can fix it. So, there's no real incentive to implement solid testing. Plus, stakeholders are rarely willing to pay for it."*



PotokDes pushed back on that logic: *"I've had to do those investigations under tight SLAs, and I wouldn't wish that experience on anyone. For me, that's the strongest reason to invest in good testing: I hate debugging SQL across dozens of models, each with multiple layers of CTEs. It's a nightmare."* But then FaithlessnessNo7800 came back with the structural problem: *"if the decision makers don't care about it, it will not become an organizational standard. And if there are no obvious incentives to it, only few developers will actually care enough to implement it."*

That exchange captures the whole dynamic. Individual engineers know testing saves time. Organizations refuse to allocate the time to save.

#2. Data and schema change constantly (20 mentions). The "we don't control the inputs" problem. Several commenters pointed out that the source data itself is a moving target. NaturalBornLucker described the cycle: *"there are some unit tests that previous team did but they are already outdated and obsolete and I disabled some of them. It's like there's some fine logic and thought in them but all that work is entirely unnecessary. And if one of table schemas is changed they fail and you need new test DataFrames."*

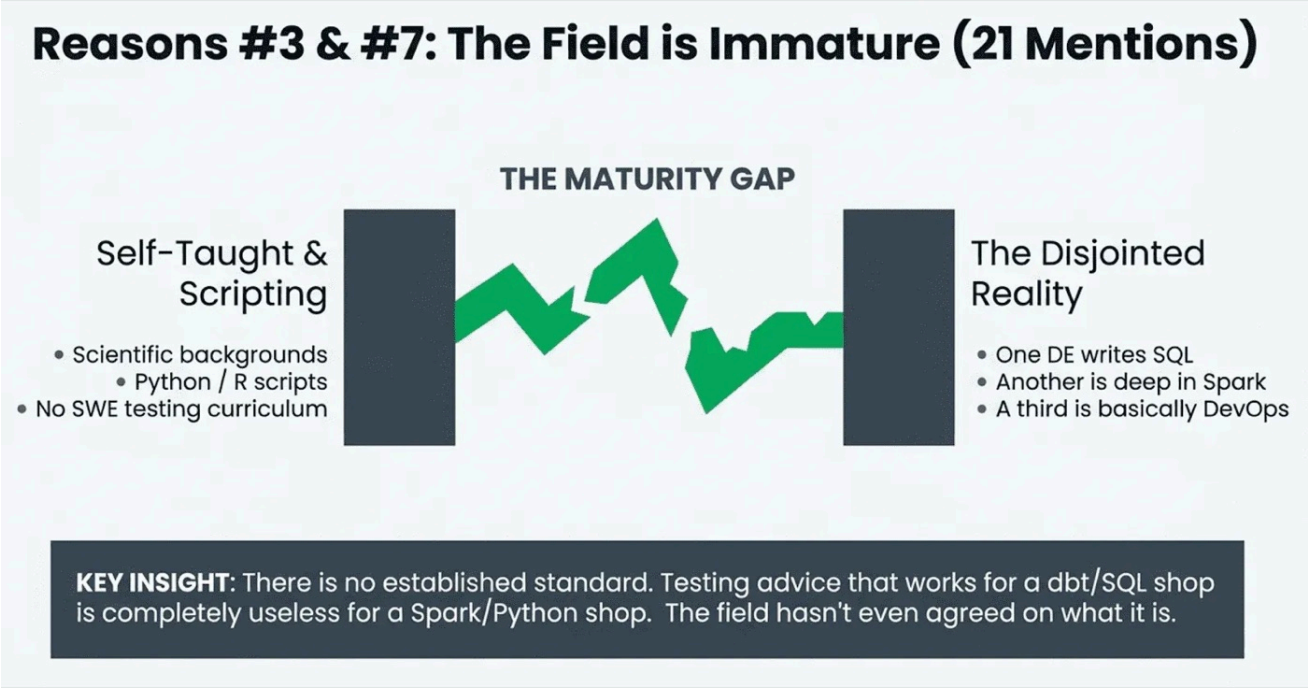


Oh_Another_Thing put it more concisely: *"It's something that changes weekly, the overhead of testing on something that changes every few weeks is too much."*

On Hacker News, a user framed this as the core difference between DE and SWE: *"the real problem with testing data engineering is that the data changes at a cadence that is independent to your version control/CI/CD cadence. You can have airtight unit and integration tests that get wrecked when the data schema changes without warning, the distribution of the data shifts, etc."* And atomicnumber3 described the scale problem: *"it's pretty easy to get something that's logically correct on your test sample but then when you run across the real dataset you find that 1 row out of every million is weird but you still gotta deal with it."*

#3. The field is immature (15 mentions). User RobDoesData, with 160 upvotes, delivered the uncomfortable truth: *"Reddit is full of bad engineers is the obvious answer. The more nuanced answer is that data engineering has a track record of being immature and not wanting to learn from SWE."*

User kaalaakhatta, who moved from software testing to data engineering, confirmed: *"98% of the people in DE field, don't care about testing. Basically it's a habit/mindset built who has worked closely with Testing."*



And it is worth noting that "data engineering" is not one job. As PotokDes observed: *"you can put a few DE in the same room, and there's a good chance none of them will fully understand each other. One's mostly doing SQL transformations, another is deep into Spark, and a third is basically DevOps."* Testing advice that works for dbt/SQL shops is useless for Spark/Python shops. The field has not even agreed on what it is, let alone how to test it.

#4. Data is inherently hard to test (12 mentions). User Outside-Storage-1523: *"There are so many business logic inside that if you want to cover all test cases you are going to essentially write a second set of pipelines and see if both have the same outcomes, which is pointless."*

User robberviet summarized it in six words: *"You can test code. Data is another story."*

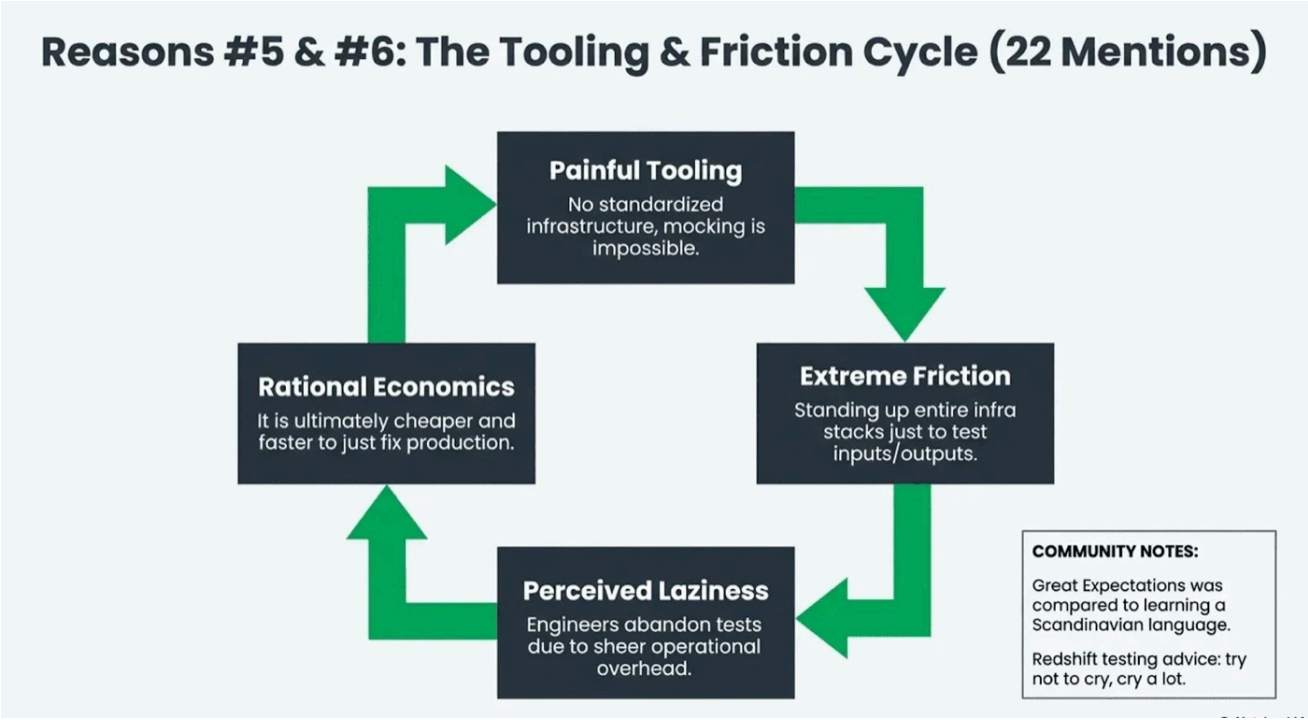
On Hacker News, user snidane argued that data engineering operates in a fundamentally different correctness model: *"Data engineering is typically closely aligned with business and its processes are inherently fuzzy. Things are 'correct' as long as no people/quality checks are complaining. There is no deductive reasoning. No true axioms. No 'correctness'. You can only measure non-quality by how many complaints you have received."*

#5. The tooling makes it painful (11 mentions). On Hacker News, the "How do you test SQL?" thread drew 315 comments and 663 upvotes, and the sheer variety of answers revealed a field that still has not settled on basic infrastructure. Some people use DuckDB as a local proxy. Others spin up Docker containers with Postgres template databases. Still others use pgTAP, or pytest harnesses, or Spark sessions with custom dependency injection. One commenter tried to test against Redshift and the top reply was: *"try not to cry, cry a lot, and then resolve not to vendor lock yourself to a black box data store next time."*

Even the tools that exist frustrate people. User eccentric2488 wrote of Great Expectations: *"Great expectations feels like learning mandarin or a scandinavian language. I want to meet the guy who designed it."* MonochromeDinosaur was harsher: *"Great Expectations IMO is garbage it promises a lot but delivers on nothing and you end up with a mess to maintain."*

User SearchAtlantis on Reddit captured the tooling frustration perfectly: *"Data Engineers don't test because the majority of tooling doesn't functionally allow it."* And on the "There is no data engineering roadmap" HN thread, rectang articulated why SQL itself is part of the problem: *"the cultural traditions of testing and validation are underdeveloped because SQL has no core testing story. People count on joins being logically correct, but don't go to the trouble to prove that true."*

#6. Laziness (11 mentions, tied with tooling). RexehBRS did not mince words: *"Another answer… Laziness. 'I don't want to write more code' I've heard, from the same people who write hundred line functions doing 13 different things."* NotAToothPaste went further: *"No matter the answer, I always understand 'Testing is not implemented because I don't know how to make it happen and I am too lazy to learn about it.'"*



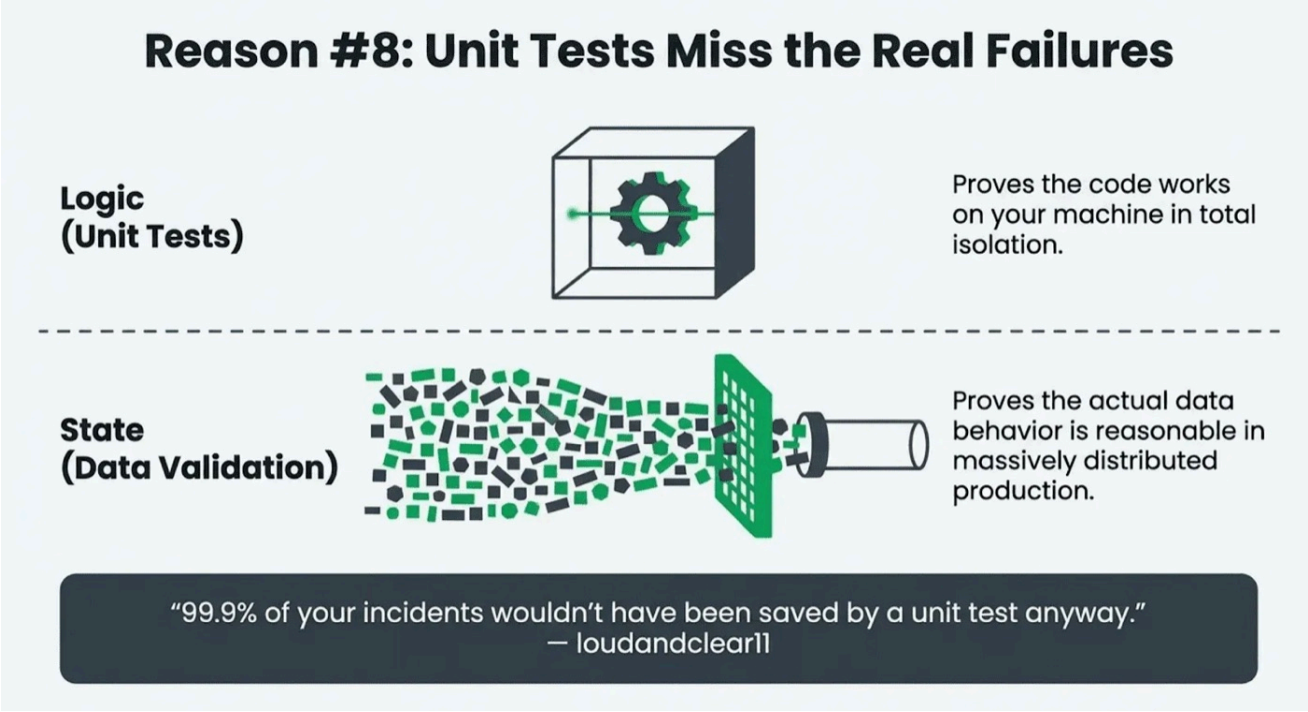
To be fair, what looks like laziness from the outside is often a rational response to the incentive structure described in #1. If you get paid more to fix technical debt than to prevent it, not testing is economically rational behavior. The system rewards it.

#7. They were never trained to (6 mentions). User peter-peta explained: *"in scientific, most people not only aren't software engineers, but most often entirely self-taught programmers, because it's often not really part of curriculum at university, it's rather 'just expected' that you can manage yourself with Python or R for data related tasks. Thus, many of them just don't know that something like unit tests etc. exist and are a thing in the first place."*

MonochromeDinosaur expanded on this: *“many data engineers are heavily SQL focused and programming is a secondary tool so they’ve simply either never had the means to test things in the way software engineers do or never had to write something larger than an ETL script.”* And then the kicker: *“it took experience writing actual application code (not scripts) to learn how to deliberately write testable code and not some untestable script kiddie slop.”*

#8. Unit tests do not catch the failures that actually hurt (4 mentions). This was the least common reason but arguably the most sophisticated. Jumpy-Possibility754 wrote: *“unit tests help but they don’t catch the kinds of failures that actually break things in production. What tends to matter more is data validation layers, pipeline observability, and the ability to replay parts of the flow when something fails mid-chain.”*

On the “Evolution of the data engineer role” HN thread, user prions, speaking from experience at Spotify, gave the most detailed version of this argument: *“the biggest challenge is the slow and distant feedback loop. The vast majority of data pipelines don’t run on your machine and don’t behave like they do on a local machine. They run as massively distributed processes and their state is opaque to the developer. Validating the correctness of a large scale data pipeline can be incredibly difficult as the successful operation of a pipeline doesn’t conclusively determine whether the data is actually correct for the end user. People working seriously in this space understand that traditional practices here like unit testing only go so far.”*



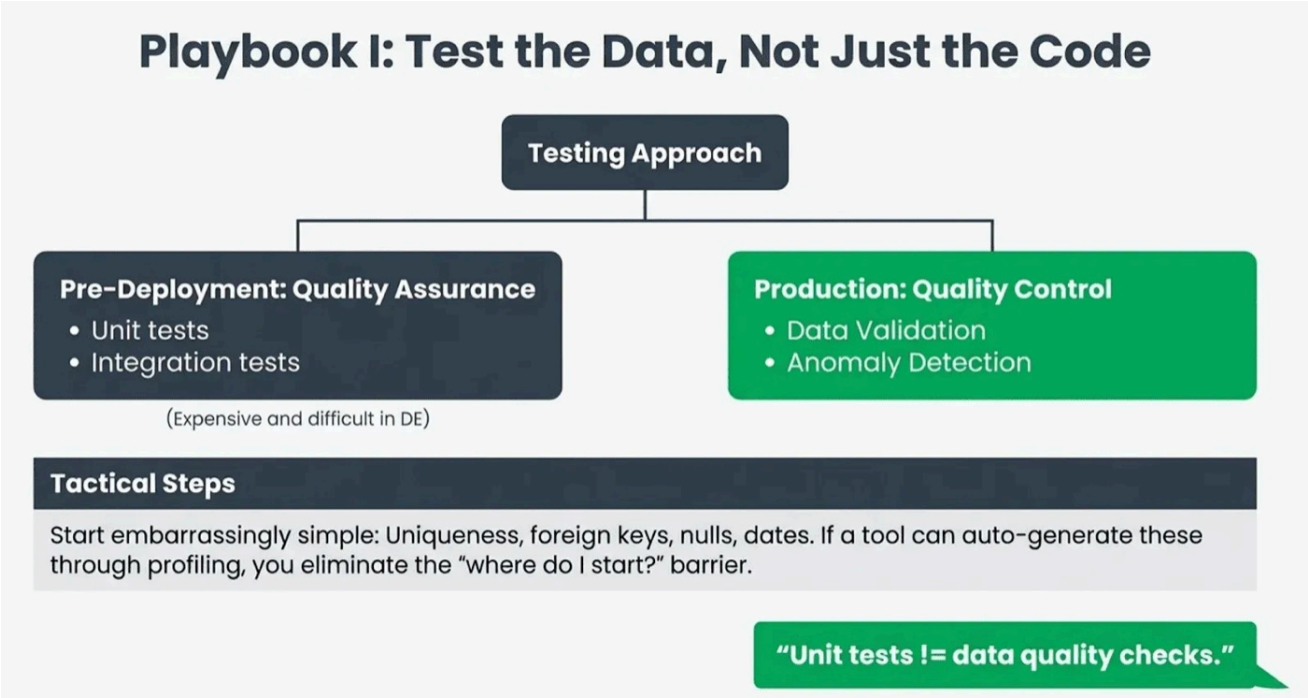
User loudandclear11 was even more direct: *“99.9% of your incidents wouldn’t have been saved by a unit test anyway. Take a look at your actual incidents and see what they have in common. That should give an indication on where you should focus your energy.”*

On Hacker News, the comment that started the pipeline validation thread said: *"Reliably getting data into a database is rarely where I've experienced issues. That's the easy part."* The hard part is knowing whether the data is correct.

What the Community Thinks We Should Do About It

The threads were not just a litany of complaints. Plenty of commenters offered concrete, battle-tested approaches. User anxiouscrimp offered the most concise testing strategy in all 849 comments: *"Smash straight into prod. You guys need to live a little."* For the rest of us who prefer to keep our jobs, the community converged on three big ideas.

Test the data, not just the code, and start embarrassingly simple. This was the closest thing to consensus across all eighteen threads. User EarthGoddessDude stated it crisply: *"Unit tests != data quality checks."* Unit tests verify that your code does what you intended. Data quality checks verify that the data you received is what you expected. In data engineering, the second category catches far more real-world failures than the first.



User kenfar laid out the full taxonomy: quality assurance through integration and unit testing (things you do before deployment), and quality control through data validation and anomaly detection (things you do in production). *"All of these are important and worth doing,"* he wrote, adding the practical caveat: *"unit testing SQL transformations is extremely expensive. It takes too much setup, whereas unit testing of python, ruby, java, go, etc is extremely straightforward."*

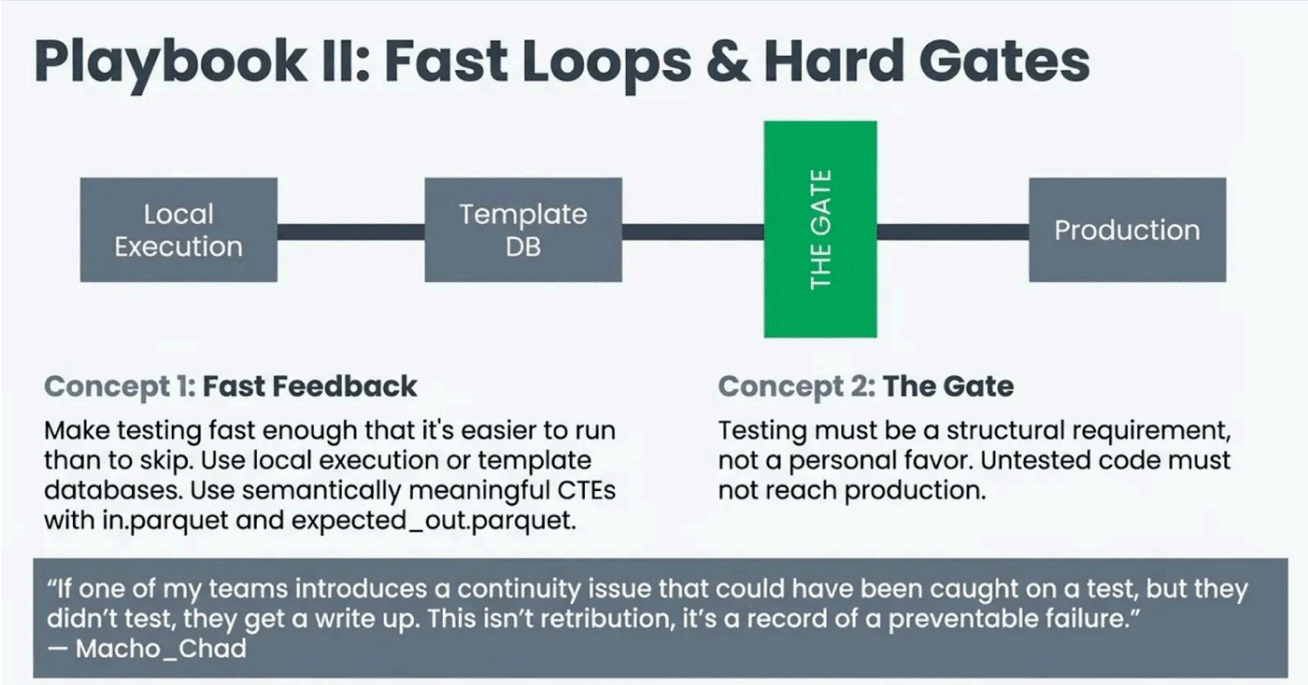
Where to begin? User kenfar recommended: *"I'd start with very obvious and simple items: checks for uniqueness, foreign keys, nulls, appropriate dates. Once that's in place I'd add reconciliation and anomaly detection checks. Only after that would I consider unit tests for SQL."* User bengen343

described a pragmatic approach that goes one step further: every model gets at least the out-of-the-box dbt tests for uniqueness and not-null, every model exposed externally gets unit tests with CSV seed files for input and expected output, and all of it runs automatically on merge. *"The real beauty,"* bengen343 wrote, *"is that the csv files are part of the repo so if someone makes changes to output models, the person reviewing the merge will see the expected output as the expectations csv needs to be changed as well. So it provides a gut-check."*

Several commenters noted that the biggest barrier is getting started, not maintaining tests once they exist. If a tool can profile your data and auto-generate sensible default tests for nulls, uniqueness, freshness, volume, and schema conformance, it eliminates the "where do I even start?" problem. The community mentioned dbt tests, dbt-expectations, and Elementary.

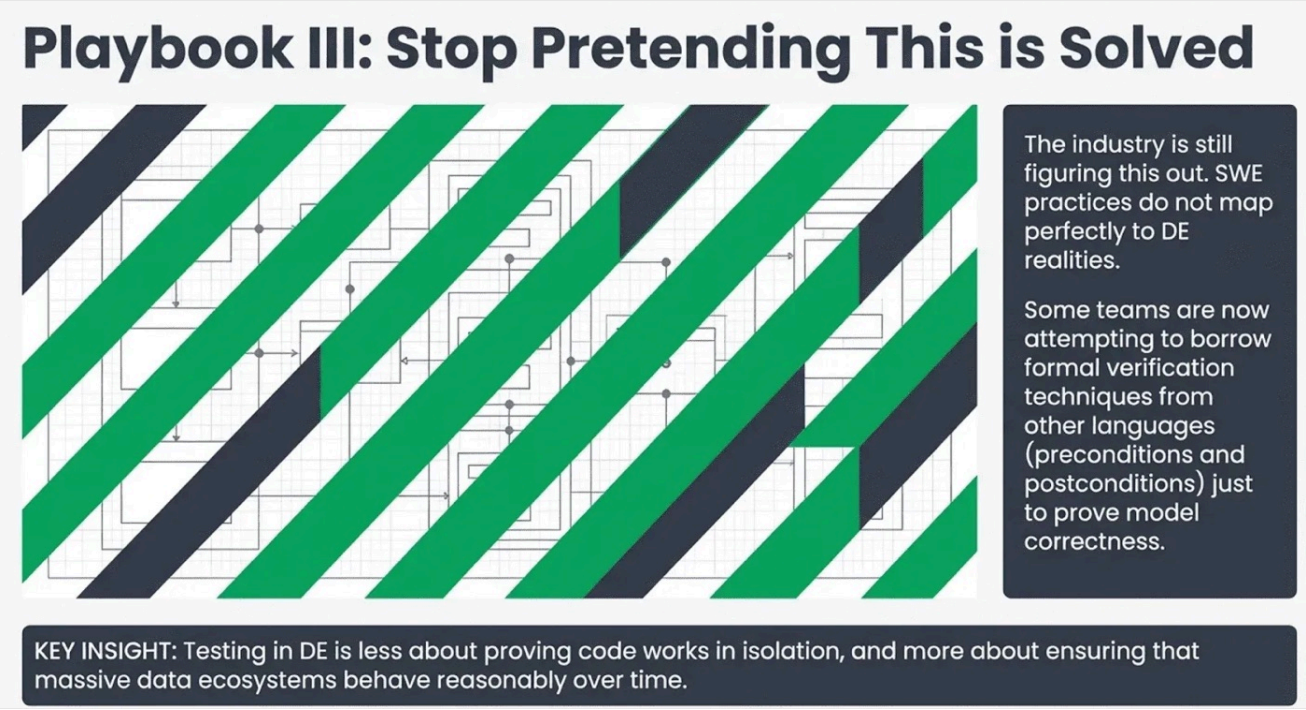
Full disclosure: DataKitchen built an open-source tool called DataOps TestGen to solve exactly this problem. It generates data quality tests through profiling and runs ongoing anomaly monitoring so engineers do not have to hand-write every test from scratch. We are biased, but we built it because this same conversation keeps repeating, and the data keeps rotting.

Make the feedback loop faster and make testing a gate. On Hacker News, the top-voted advice from RobinL was: *"Try and write any complex SQL as a series of semantically meaningful CTEs. Test each part of the CTE pipeline with an in.parquet and an expected_out.parquet."* The approach uses DuckDB for fast local execution. MonochromeDinosaur prescribed a complementary pattern: *"write pure functions, separate I/O from transformations and do unit tests with in-line fixtures using data structures native to the tool you're using. This is easy because you control the intermediate schemas and state."* User feike on HN described getting 82 integration tests to run in 6.9 seconds using Postgres template databases, creating a fresh database for each test from a template snapshot. The common thread: make testing fast enough that it is easier to run than to skip.



But faster tests only matter if someone actually requires them. User Macho_Chad described an environment where untested code results in a written record of preventable failure: *"If one of my teams introduces a continuity issue that could have been caught on a test, but they didn't test, they get a write up. This isn't retribution, it's a record of a preventable failure."* User killerfridge took the same approach with code review: *"If they want it to get past code review, it better have greater than 80% coverage, and the tests must pass. It's amazing how much it's exposed the poor practices, but also how much it's shone a light on how much doesn't actually work properly."*

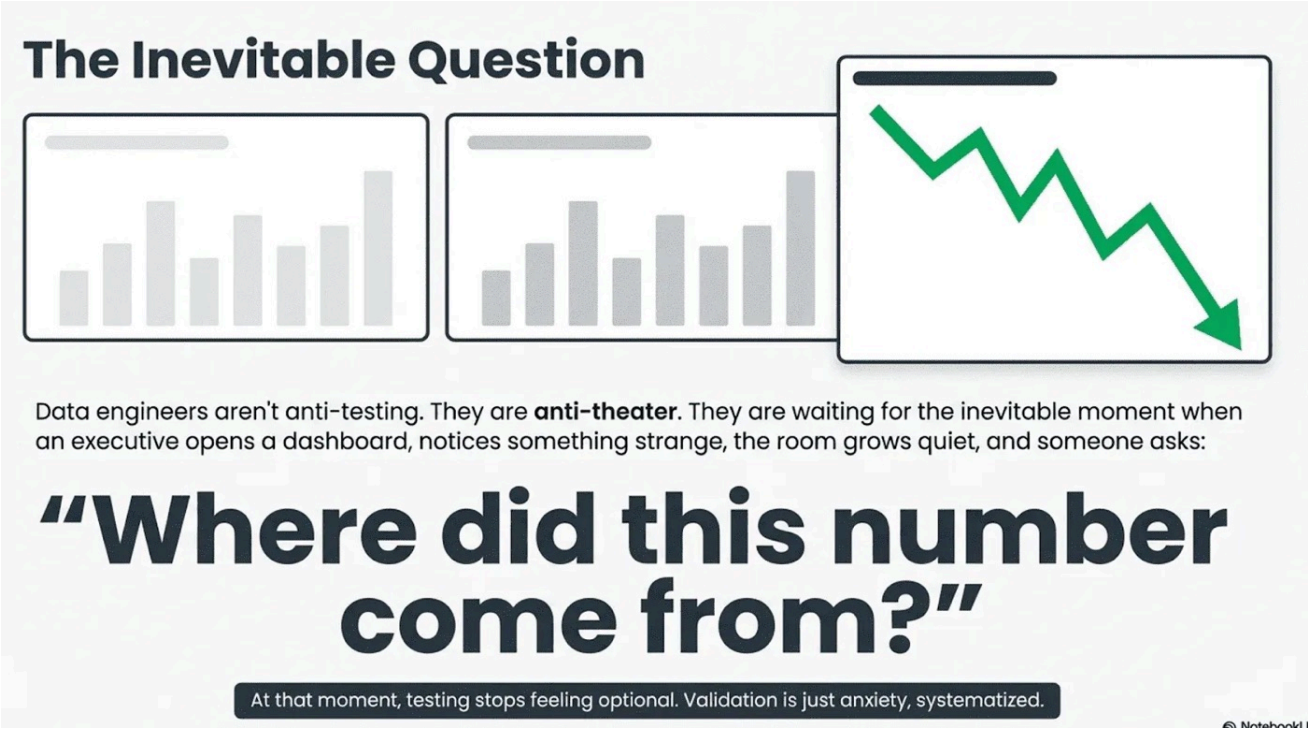
Stop pretending this is a solved problem. On the dbt Community Forum, the "State of Testing in dbt" thread acknowledged openly that the field is still figuring out what testing should look like for data transformations. One contributor proposed borrowing a formal verification technique from another programming language: define preconditions and postconditions for each model, and prove correctness by showing that any run that starts in a state where the preconditions hold will end in a state where the postconditions hold. The fact that someone felt the need to reach that far outside the data ecosystem for a testing idea tells you how unsettled the question still is.



When You Get Asked, "Where Did This Number Come From?"

Read enough of these threads and a pattern emerges. Data engineers are not anti-testing. They are anti-theater. They have watched SWE teams mandate 80% code coverage on trivial SQL and declare victory while the actual data rots unmonitored in production. They have been told to "just write unit tests" by people who have never tried to mock a Snowflake connection. They have been asked to test against source data that changes schema every other Tuesday with no notice and no contract.

The number one reason data engineers do not test, by a factor of two over all other reasons, is that nobody gives them the time or budget to do it. That is not an engineering problem. That is a management problem. And the number eight reason, the one with the fewest mentions, might be the most important one to sit with: that traditional unit tests simply do not catch the class of failures that actually hurt in data engineering.



But here is the surprising conclusion from the internet's collective investigation: data engineers do test. They simply do it differently than software engineers expect. They validate assumptions about data. They monitor systems for anomalies. They inspect outputs when logic changes. They write targeted tests for complex transformations. Testing in data engineering is less about proving that code works in isolation and more about ensuring that entire data ecosystems behave reasonably over time. The real question is not whether every possible scenario has been tested. The real question is how much uncertainty an organization is willing to tolerate before it starts verifying the numbers it depends on.

User roastmecerebrally captured the human side of this perfectly. After describing how they implemented validation testing for their warehouse and were *"treated like a god,"* they added the self-aware punchline: *"Sorry, no, I just have anxiety."*

Maybe that is the real answer. The data engineers who test are not smarter or more disciplined. They are just more nervous. And sooner or later, every organization confronts the same moment that turns the nervous ones into the essential ones. An executive opens a dashboard and notices something strange. The numbers look wrong. The room grows quiet.

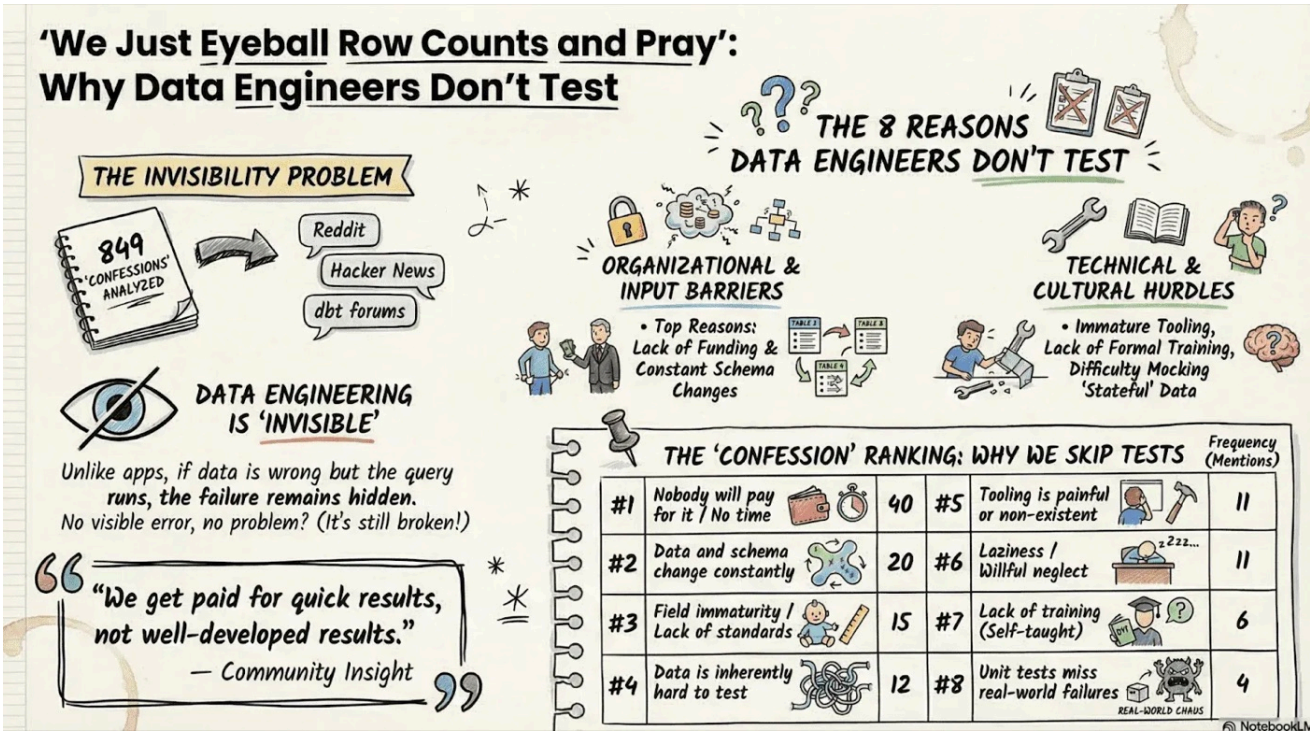
Someone eventually asks the question that echoes through every data team.

Where did this number come from?

At that moment, testing stops feeling optional.

Appendix: How We Counted

We read every comment across all eighteen source threads and tagged each individual comment that cited a reason for not testing. Each comment was counted at most once per reason, even if it mentioned the reason multiple times. Here is the ranked order by total mentions:



1. **Nobody pays for it / no time** (40 mentions). The runaway winner, showing up in nearly every high-engagement thread. It was not even close. When data engineers explain why they do not test, the answer most often given is not technical. It is organizational. They are not given the time, the budget, or the incentive.
2. **Data and schema change constantly** (20 mentions). The "we don't control the inputs" problem. Source systems change without notice, schemas drift, and third-party APIs break contracts.
3. **The field is immature** (15 mentions). Culture, legacy systems, silos, and the absence of established practices that SWE has had for decades.
4. **Data is inherently hard to test** (12 mentions). The "it's not code, it's state" argument. SQL is declarative; you cannot set breakpoints, and mocking is nearly impossible.
5. **Tooling is painful** (11 mentions). Mocking tables, vendor lock-in, slow feedback loops, and no standardized test infrastructure.
6. **Laziness and willful neglect** (11 mentions). Tied with tooling. Some commenters were blunt about this.

7. **No training or CS background** (6 mentions). Self-taught programmers who never learned testing as a practice.
8. **Unit tests miss real failures** (4 mentions). The sophisticated argument that traditional tests catch the wrong class of errors in data work.

The Stack Overflow threads (sources 9, 14 through 18) were included for completeness but contributed very few reason mentions. They were mostly "how do I do this?" questions with short tactical answers. One thread had zero replies. Another's top answer listed "stare and compare" as a legitimate testing method. The substantive debate occurred across the four Reddit and four Hacker News threads, which together account for roughly 90% of all reason mentions. The dbt Community Forum threads leaned more toward solutions than complaints.

Appendix: Compared to We Got Roasted On Reddit … Article

There are more reasons in 'We Got Roasted On Reddit For Asking 'Why Data Engineers Don't Test?' … how does it compare to the eight reasons from our blog post?

The 'We Got Roasted' (WGR) article lists these eight reasons:

1. Writing good tests is genuinely hard
2. No one has thought through what happens when a test fails
3. Feature delivery always wins
4. Tests rot, and nobody budgets time to maintain them
5. You don't know what to test until it's already broken
6. Nobody senior enough cares until it's too late
7. Catching the error makes it your problem
8. At scale, comprehensive testing must fit

Our blog post lists these eight reasons (ranked by mention count):

1. Nobody will pay for it / no time (40)
2. Data and schema change constantly (20)
3. The field is immature (15)
4. Data is inherently hard to test (12)
5. Tooling is painful (11)
6. Laziness / willful neglect (11)
7. No training / CS background (6)
8. Unit tests miss real failures (4)

Here's how they map:

WGR article reason	This blog's reason	Match type
1. Writing good tests is genuinely hard	4. Data is inherently hard to test + 7. No training	Partial overlap. WGR focuses on the skill of writing non-noisy tests. Our #4 is about data being fundamentally hard to test, and #7 is about not having the training. Both contribute.
2. No playbook for when tests fail	No direct match	New reason. Our threads didn't surface this as a distinct theme. It's a genuinely novel insight from the DK thread.
3. Feature delivery always wins	1. Nobody will pay for it / no time	Direct match. Same reason, same framing.
4. Tests rot / no maintenance budget	2. Data and schema change constantly	Direct match. Same root cause (upstream changes), slightly different framing (DK emphasizes maintenance cost, ours emphasizes the change itself).
5. Don't know what to test until it breaks	7. No training / CS background	Partial overlap. WGR frames it as experiential knowledge. Our #7 frames it as formal training. Both are about not knowing what to test, but for different reasons.
6. Nobody senior enough cares	1. Nobody will pay for it / no time + 3. The field is immature	Strong overlap. WGR focuses on leadership, not caring until a crisis. Our #1 is the budgetary consequences; our #3 is cultural immaturity.
7. Catching the error makes it your problem	6. Laziness / willful neglect	Partial overlap, but WGR framing is sharper. What our threads call "laziness" is what WGR's thread reframes as a rational response to perverse incentives. Our blog actually made this connection in the laziness section.
8. At scale, comprehensive testing must fit	No direct match	New reason. Our threads didn't have hyperscale commenters (exabyte-per-day). This is a distinct constraint that only applies to very large operations.

Summary:

- **Direct match , 3 (reasons 3, 4, 6)**
- **Partial overlap - 3 (reasons 1, 5, 7)**
- **New / not in our blog - 2 (reasons 2, 8)**

Two genuinely new reasons from the WGR article that our 18-thread analysis did not surface:

"No playbook for when tests fail." This is the operational gap: even if you write tests, nobody has decided which ones stop the pipeline, which generate warnings, and who will act on the results. Our blog talked about why people don't write tests. WGR thread surfaced why people who do write tests still fail.

One reframing that improves on our analysis: WGR's reason #7, "catching the error makes it your problem," is a much sharper version of what our threads loosely called "laziness." The WGR thread names the actual incentive structure rather than the behavior it produces. This blog already connected these two (the sentence "what looks like laziness from the outside is often a rational response to the incentive structure"), but the WGR article gives it its own named reason, which is probably more accurate.

THAT'S WHY WE BUILT THIS

TestGen is open source. No excuses left.

These barriers are exactly why we open sourced a full-featured version of TestGen — and why our enterprise pricing is flat-rate, not per-user or per-connection.

↗ Get TestGen on GitHub Learn about Enterprise →

Free to download · Free to use · Flat-rate enterprise for teams

Appendix: Sources

#	Community	Topic	URL
1	Reddit r/dataengineering	Testing in DE feels decades behind traditional SWE. What does your team actually do?	Link
2	Reddit r/dataengineering	Why don't data engineers test like software engineers do?	Link
3	Reddit r/dataengineering	Why data engineers don't test: according to Reddit	Link
4	Reddit r/dataengineering	Your opinion on testing in data engineering	Link
5	Hacker News	Ask HN: How do you test SQL?	Link
6	Hacker News	Validating the correctness of a large scale data pipeline	Link
7	Hacker News	The evolution of the data engineer role	Link
8	Hacker News	There is no data engineering roadmap	Link
9	Stack Overflow	How to test (unit test) on the ETL process?	Link
10	dbt Community Forum	How to get started with data testing	Link
11	dbt Community Forum	dbt model (think unit) tests POC	Link
12	dbt Community Forum	Running dbt unit tests in CI/CD	Link
13	dbt Community Forum	State of testing in dbt	Link

#	Community	Topic	URL
14	Stack Overflow	How should I perform automated testing of SQL ETL scripts and stored procedures?	Link
15	Stack Overflow	How To Test Talend ETL	Link
16	Stack Overflow	ETL and QA testing	Link
17	Stack Overflow	How do I automate ETL tests on a Data Warehouse project using .NET?	Link
18	Stack Overflow	What is the best way to test ETLs: Source vs Target data comparison	Link

Total comments read across all 18 sources: 849.

A Data Usability Crisis is Costing Your Company

Read the original article at: https://datakitchen.io/blog/a-data-usability-crisis-is-costing-your-company/

I recently met with a data lead who had this reaction to a litany of data inconsistencies and usability problems: “Oh, we don’t have to worry about those. Users know about them. They have workarounds.” It’s a surprisingly common approach on data teams where the sheer bulk of data overwhelms the best intentions. In this company, like many teams I’ve observed, it was dangerous in subtle as well as obvious ways.

Our company, DataKitchen, focuses on applying DataOps principles to support data pipelines. We recently released an open-source data quality screening and assessment tool, TestGen, that identifies and documents a wide range of errors, inconsistencies, and usability issues in SQL datasets. One fascinating benefit of the work has been the chance to review the data pipelines of many companies. We've developed a set of Impact Dimensions that pinpoint the level and locus of risk that test failures signify. Along the way, I've found some eye-opening trends in the way organizations can talk themselves out of even acknowledging these issues, much less coming to terms with their effects.

These rationalizations are flimsy, and they are everywhere. The mindset has become more damaging as companies focus on AI Readiness and pull more of their data into uses nobody planned for. It sabotages the innovators who should be producing insights and data science advances. Then the failures snowball, and the trust the company loses is hard to win back.

The obvious problem at this company was that the data team's "users" (data scientists and analysts focused on building data products and answering business questions) were not aware of all the data problems, so inaccuracies cascaded. And when analysts did know, they each addressed the problems differently in their own deliverables, which spawned different versions of the truth.

Of course, if data engineers were too swamped to deal with quality problems, analysts and data scientists surely weren't going to resolve them. Data quality wasn't their priority, and they were not going to manage it effectively. Self-service had evolved to mean, "I get to do someone else's job along with mine." Not good.

Some of these data problems involve business context. Many do not. These are not classic data quality issues, where data in a system doesn't match the business activity it tracks. Many usability issues we screen for (inconsistent representations of categories or missing values, casing problems, invalid characters, obvious dupes) are artifacts of insufficient validation, processing failures, or integration discrepancies. They don't require long investigations or business rule deep-dives to fix.

But they do create landmines for downstream users, who need to know how and where to find them and then need to spend extra time to clean them up. And when data is swept into AI systems at scale, the impact isn't evident until too late.

Since this data team hadn't taken responsibility for identifying these issues, they didn't have their fingerprints on the result. The problems originated upstream, and failures blew up much later. Not their fault, and not their concern. The result, inevitably, was Usability Debt that someone was going to pay downstream.

Through nearly 40 years in this business, I've found usability to be a critical but overlooked data quality dimension. Data can be accurate, valid, and utterly unusable.

What's really surprising is the side effects. For one thing, data teams can undercut themselves. It's harder to create effective logical DQ checks when representation is faulty or inconsistent, for exactly the same reason it's harder on data consumers later.

But there are unpredictable consequences too. You can't trace a complex data environment like code. Any enterprise has autonomous agents with their own incentives and goals, unpredictable external changes and unexpected feedback loops. Lapses can have surprising indirect impacts, and that's exactly what I found.

Consider these emergent effects, reported by stakeholders. They arose from the strategy and from analysts' natural responses to it. Any one issue looks trivial. The whole set is not.

- **Analysts felt frustrated and helpless**, despite the company's emphasis on empowerment through self-service. At best, they lost valuable time and productivity. At worst, they were blindsided by data idiosyncrasies. It didn't always occur to them that these problems could or should have been resolved upstream. They just knew that "data wrangling" was a slog and a minefield.
- **Analysts and end-users lost trust.** The distrust spread past the datasets that had quality issues, onto good datasets, well-managed processes, and successes the data team achieved effectively. Bitten enough times, users feared the unknown-unknowns, whether they were lurking or not.
- Analysts with **expertise in workarounds** outperformed those who excelled in their own specialties. So skilled data scientists **couldn't innovate** as quickly, while "old-hands" (using old approaches) remained irreplaceable.
- Facing this information imbalance, employees felt incentivized to **hoard knowledge**, not share it. Guess what: the problems grew worse.
- Insider tips were shared in **informal networks**, since official information was scarce. One young analyst complained to me that "who you know" had become more important than "what you know" on

the data team.

- **Onboarding was slow** and fraught with challenges. New employees took much longer to contribute and didn't get a fair chance to succeed.
- Outside consultants and vendors couldn't jump in and add value. The company **lost access to invaluable expertise**. High-visibility projects were hobbled by flurries of small failures.
- The "old-hands" got their job security, but at the cost of **work-life balance**, complaining they were continually called into crises during evening hours and vacations.
- Savvy data scientists and analysts **lost valuable time**, getting drawn inevitably into upstream operational troubleshooting to protect their deliverables. This frustrated them, and it **impacted turnover**.
- **Workaround logic was fragmented** within deliverables and documents: Tableau formulas, spreadsheets, Jupyter Notebooks. Business rule changes became impossible to implement without breaking something. Data corrections themselves became impossible to make, for fear of breaking downstream workarounds.

The funny thing is that this was in many ways a fantastic company. Every single person I spoke with was committed to excellence. Stakeholders felt safe enough to do candid self-assessments, a credit to them and their corporate culture. In fact, everyone was doing their job as the organization defined it. I've seen so many root-cause analyses become a blame game. In this case, there was nobody to blame. But all the problems had a feeling of inevitability about them, like a natural disaster.

You might say the data team should have risen to the occasion and taken responsibility. But actually, their choices were understandable and rational. It wasn't their job to master or manage the business meaning of the data. Their primary role was to make sure that the outputs they produced accurately reflected the inputs they received. And no one had built the time, effort or budgets into project plans or operations to come to terms with these gaps.

The feedback loop that might have held them to account didn't really exist. Downstream analysts often see data cleansing and standardization as their own burden. They literally don't know what they're missing, and nothing changes. From the company's perspective, the result was hidden bottlenecks and hidden costs. And if all these costs are misattributed or ignored, there's no way for management to track or address them.

The worse it got, the more painful it was to admit the problem at all. Bad data can be corrected. But the feedback does catch up with you, and when it does, it hits hard. The loss of trust in a company's data products becomes a catastrophic issue. There are always more people ready to cast blame than catalyze prevention. And as every chef knows: you may not have caused the rot, but it's on you if you didn't smell the fish.

US auto manufacturers didn't take quality so seriously until foreign competitors took full advantage of their complacency. Manufacturing has a cost model for this. Juran's cost-of-quality work, from the 1951 *Quality Control Handbook*, puts prevention at 1x, internal detection at 10x, and a defect that reaches the customer at 100x, with regulated industries running higher still. That is the 1:10:100 rule, and it is why prevention is the cheapest point on the curve by orders of magnitude. Funny how, even in manufacturing companies, data work is still artisanal. Standardizing how data is depicted sounds as trivial as standardizing the size of bolts and screws. It's time to realize how crucial it is.

But be warned. A data team that relies on downstream workarounds to handle data quality issues causes more harm than it realizes. Dirty data carries a high cost, and usability is no luxury. Leaders assume everything is ok because a system like this keeps sputtering along. In truth, they're missing the chance to improve their organization along with their results.

SECTION 2

Culture and Organization

1. The Seven Deadly Sins of Data Quality
2. The Data Errors Shame Game
3. Is Your Team in Denial about Data Quality?
4. The Ostrich Problem: When Your Data Team Stops Caring After Deployment
5. The Data Quality Revolution Starts with You
6. The Five Organizational Sins of Data Teams
7. You're Thinking About Data Products All Wrong
8. Process Guardianship: The Most Valuable Work You Are Probably Not Doing
9. How Data Quality Leaders Can Gain Influence and Avoid the Tragedy of the Commons
10. How Do I Get My Data Team More Efficient?

CULTURE AND ORGANIZATION

The Seven Deadly Sins of Data Quality

Read the original article at: https://datakitchen.io/blog/the-seven-deadly-sins-of-data-quality/

Data quality problems don't just appear out of nowhere. They fester in organizational cultures where dysfunction becomes normalized, where warning signs are ignored, and where the path of least resistance leads to ever-growing data debt. After years of working with data teams across industries, we've identified seven behavioral patterns that consistently undermine data quality initiatives. Like the classical deadly sins, these aren't simply technical failures, they're deeply human tendencies that prevent teams from achieving the data quality their organizations desperately need.

Understanding these sins is the first step toward overcoming them. Each represents a common trap that even the most well-intentioned data teams fall into. The good news? Recognition is the beginning of recovery, and there are practical paths forward for teams willing to confront these challenges head-on.

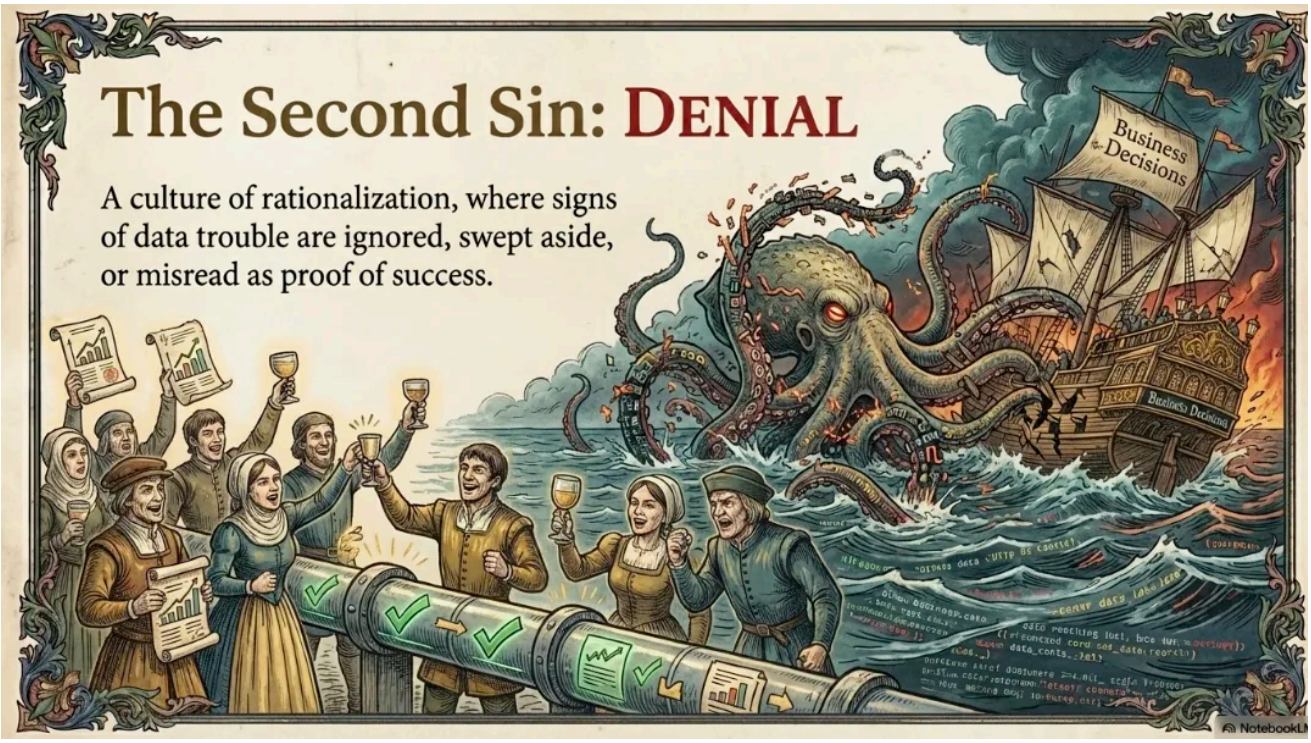
1. Shame: The Fear of Finding Problems

Here's something we don't talk about enough in data engineering: shame. The quiet, corrosive feeling that if something is wrong with the data, it must be your fault. And if you're the one who finds the problem, you're the one who'll have to answer for it. As explored in The Data Errors Shame Game, this shame creates a strange and painful paradox: the very people best positioned to catch data quality issues often develop an unspoken preference not to look too closely. Not because they don't care, but because they've learned that discovering a problem means owning the blame for it.

When data quality responsibility lives entirely within the engineering team, every error feels personal. Engineers are expected to anticipate problems before they happen, catch them instantly when they do, and fix them without anyone noticing there was ever an issue. Over time, this creates a defensive posture: if I don't find the error, I can't be blamed for it. The antidote to shame is shared visibility. When data engineers and business users look at quality problems together, the dynamic shifts from "what did you miss?" to "what should we tackle first?" Building quality circles where stakeholders regularly review data issues together dissolves the isolation that makes shame thrive.

2. Denial: The Culture of Rationalization

In many organizations, data quality problems fester in the shadows, ignored, rationalized, or swept aside with confident-sounding statements that mask deeper dysfunction. As described in Is Your Team in Denial about Data Quality?, teams often build a culture of rationalization where signs of trouble are misread as proof of success. A pipeline ran "all green"? That doesn't mean the data inside was correct. "I haven't heard a single complaint."? That usually means users have stopped expecting anything better or have worked around the data team entirely.



The woman on the phone saying "The users know about that, they have a workaround!" might be trying to help, but she's normalizing failure. Workarounds don't fix broken data; they institutionalize it. Confidence is not a substitute for verification, manual processes, however well-intentioned, are rarely repeatable, scalable, or trustworthy. Breaking the cycle of denial starts with playing "Data Quality Denial Bingo" with your team: printing out common rationalizations and checking off every statement team members have heard, or said themselves, in the past month. Use the laughter and mild embarrassment that follows as a gateway to having a serious conversation about your data quality practices.

3. Avoidance: The Ostrich Mentality

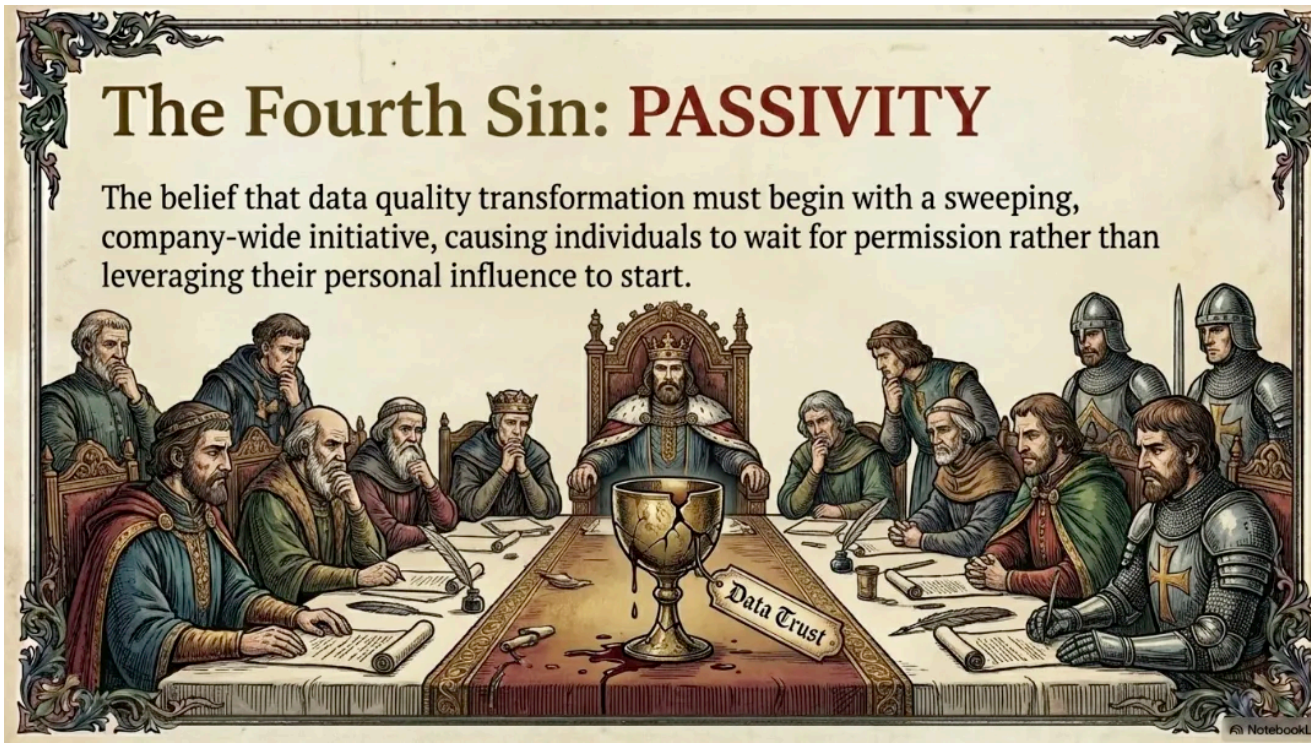
Your team has gone full ostrich. Heads buried deep in the sand, convinced that if they can't see the production data errors, those errors must not exist. The pipeline is deployed. The tests passed. Job done, right? As examined in The Ostrich Problem: Your Data Team Thinks Their Job Ends at Deployment, this "deploy and forget" mindset is one of the most common, and most corrosive, patterns in data engineering teams. The team sees their job as done once pipeline code is in production. They don't feel responsible for data quality problems that pop up after deployment

The frustrating part? Your team isn't lazy or incompetent, they're just overwhelmed with development work and genuinely don't realize how bad things have gotten downstream. The ostrich approach feels safe when you're already buried in sprint work. The fix starts with gathering the facts: survey end customers about their trust in the data, count recent data errors from tickets and Slack threads, and use tools to score your data quality. Share the results with your team. Most engineers don't want to ship bad data; they just don't have visibility into what's happening after deployment. Turn on the lights, and the ostriches will lift their heads.

4. Passivity: Waiting for Permission

Picture this: You're sitting in yet another meeting where someone asks, "Can we trust this data?" and the room falls silent. If you're nodding along, you've identified yourself as the perfect candidate to become your organization's data quality champion. As we discussed in The Data Quality Revolution Starts with You, data quality transformation doesn't begin with company-wide initiatives or sweeping organizational

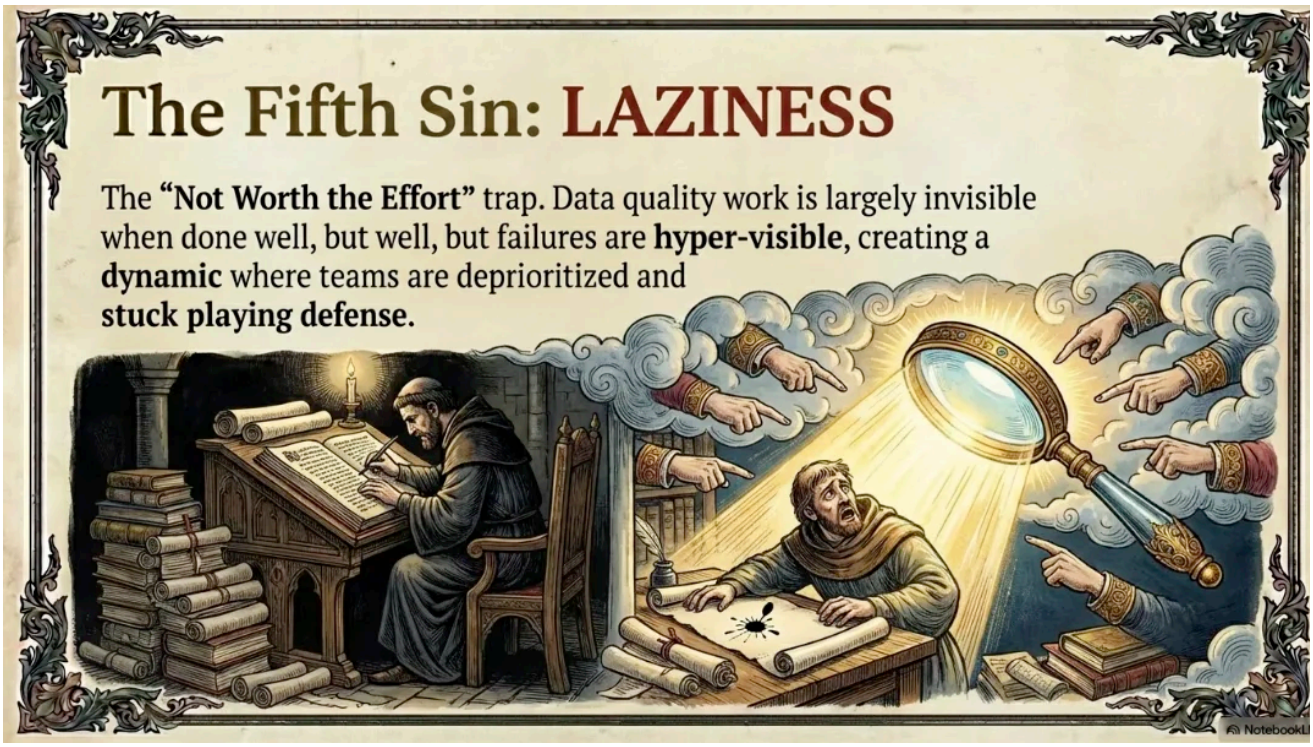
changes. It starts with a single empowered individual who decides that enough is enough, and that person could be you, whether you're a data engineer debugging pipeline issues at 2 AM or a data leader watching your team struggle.



The magic happens when you stop waiting for permission and start leveraging your personal influence. You already have more power than you think. You know which datasets cause the most headaches, which stakeholders ask the toughest questions, and which processes break down most frequently. This insider knowledge is your superpower. Instead of attempting to fix every data problem in your organization, focus on a specific end customer with a specific pain point. Choose one customer, understand their world, and make their life measurably better. When you transform someone's work life by fixing their data headaches, you're recruiting a champion who will advocate for your initiatives.

5. Laziness: The "Not Worth the Effort" Trap

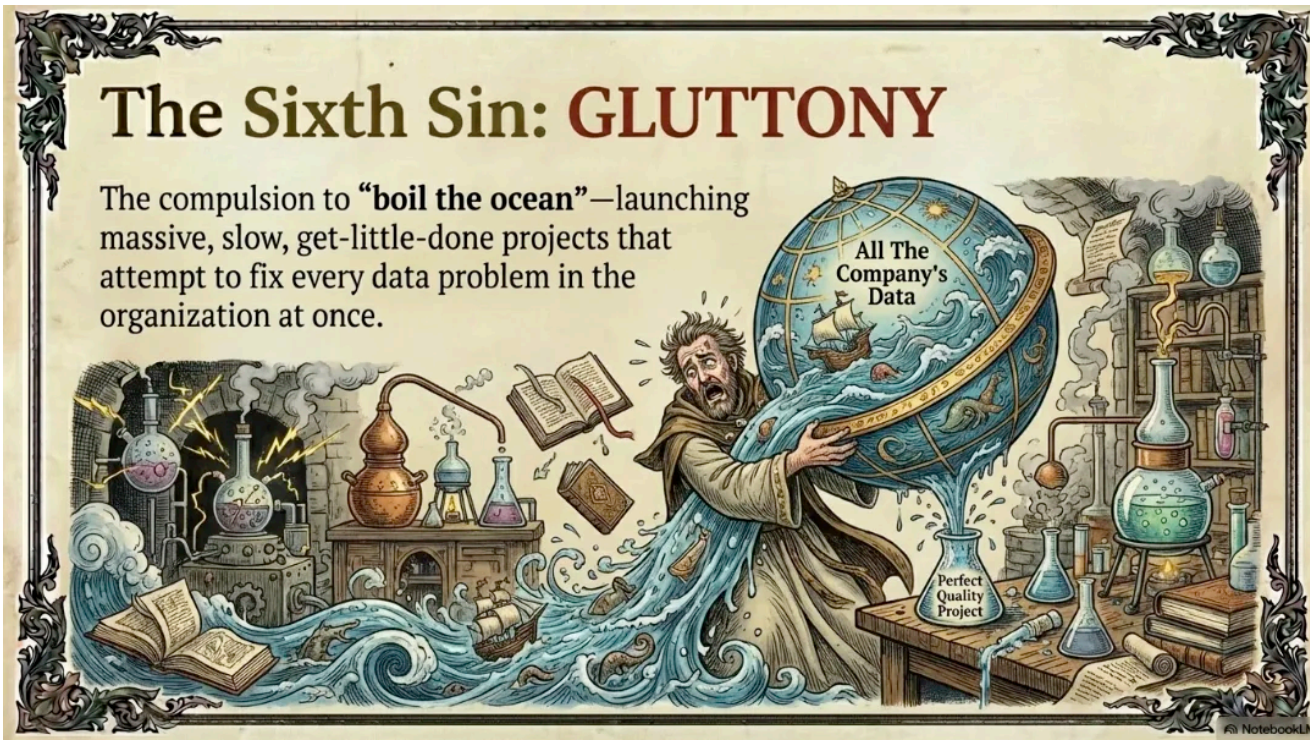
Data quality work is largely invisible. When everything is running smoothly, no one notices the effort behind it. But everyone notices the moment something breaks, when a report is wrong, or a dashboard fails. As discussed in Why Data Quality Isn't Worth The Effort, this creates a dynamic where data teams are expected to prevent problems without being seen, making it hard to advocate for the time and resources required to do the job well. Data quality efforts are often deprioritized because they don't directly generate revenue.



Data environments are constantly changing: new data sources are introduced, pipelines are updated, and business logic evolves. These changes often occur without proper cross-team communication, leaving data engineers and analysts to discover issues only after they've caused damage. The result is teams stuck playing defense, constantly reacting to problems rather than proactively preventing them. The key insight is that automation doesn't have to be overwhelming. Tools that automatically generate tests based on the structure and behavior of the data itself can dramatically simplify the process, shifting data quality from a reactive chore to a proactive, integrated part of the development process.

6. Gluttony: Trying to Boil the Ocean

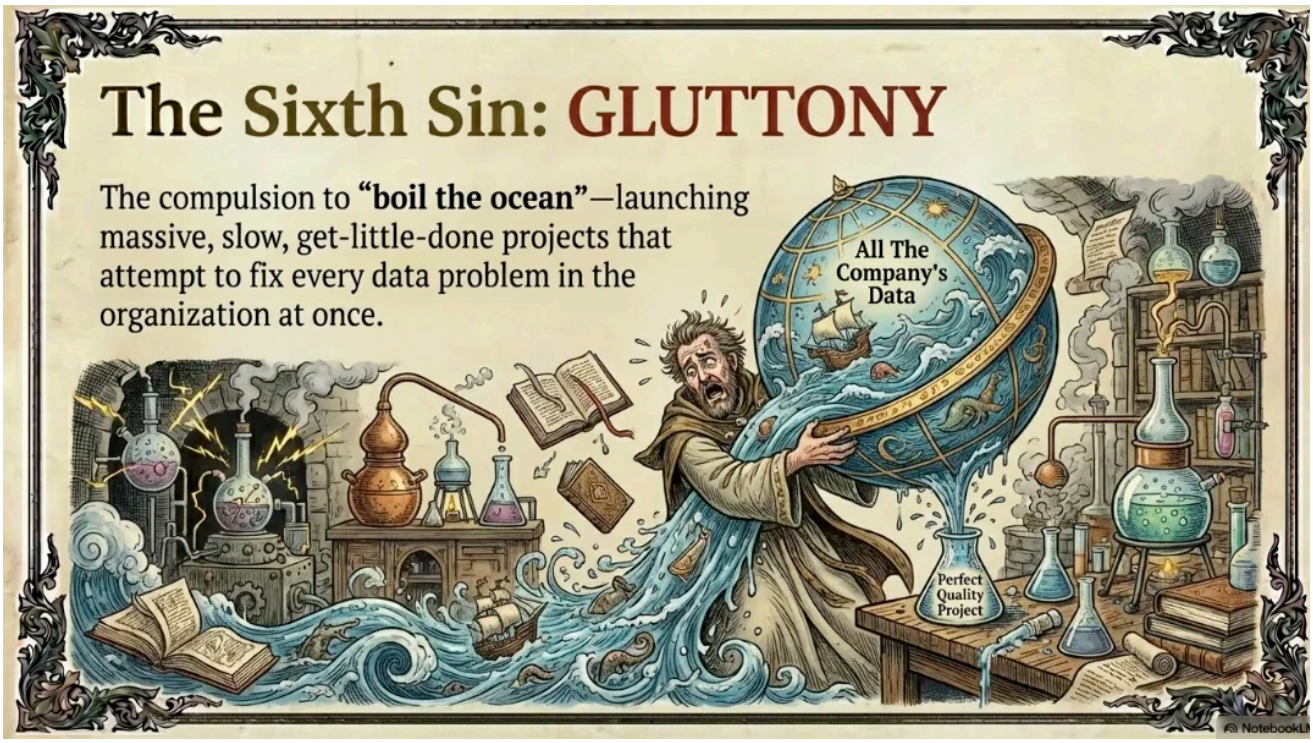
Here's where most data quality initiatives go wrong: they try to boil the ocean. Improving data quality can feel overwhelming, there are so many things to fix and so many processes to improve. Where do you even start? As outlined in A DataOps Way To Data Quality, the answer lies in starting small, iterating quickly, and treating every improvement as a test or experiment. Instead of attempting to fix every data problem in your organization at once, focus on a specific pain point with a specific customer.



This laser focus does two things: it makes success achievable, and it creates visible wins that build momentum for bigger changes. Think of it like renovating a house, you wouldn't tear down every wall at once. Instead, you'd start with one room, perfect the process, then apply those lessons to the next space. DataOps principles provide the roadmap: start small, iterate quickly, launch a simple data quality check, measure its impact, learn from what works, then iterate. This approach feels less intimidating than traditional "big bang" implementations and delivers results faster.

7. Ignorance: Not Understanding Your Data

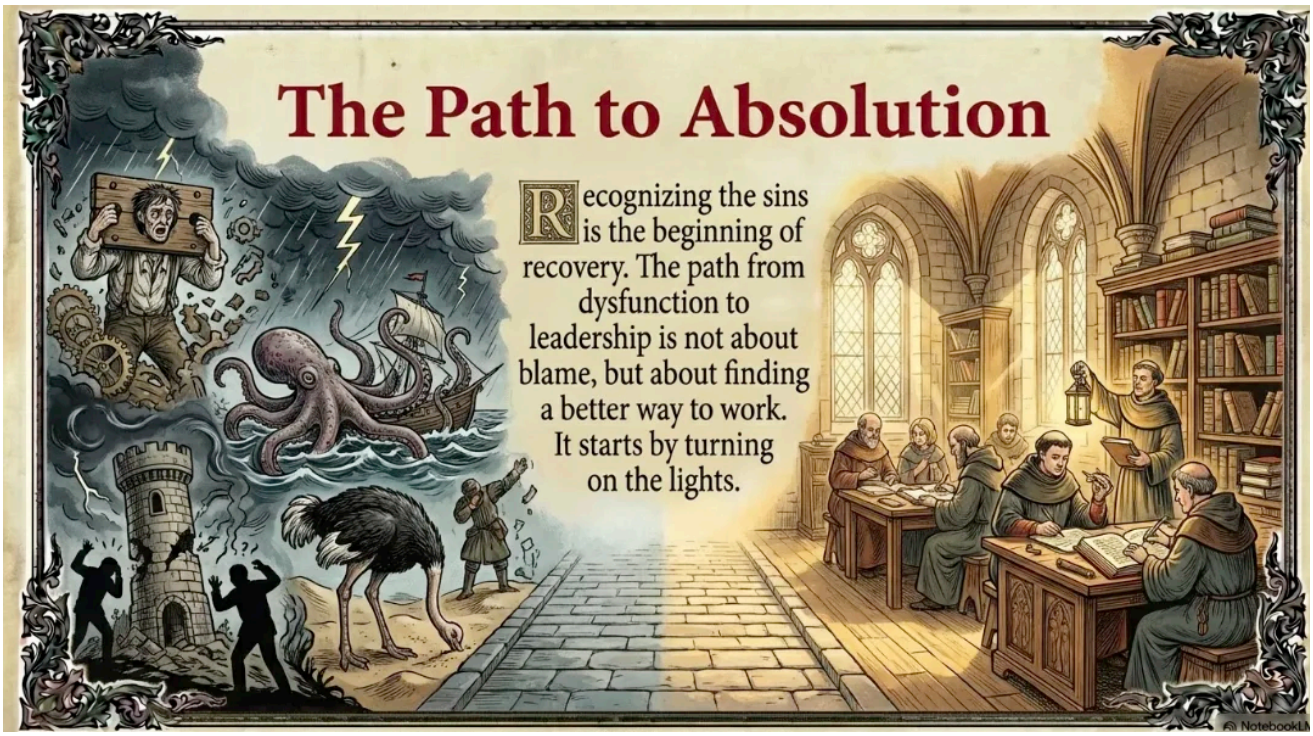
The most fundamental sin is simply not understanding the data you're responsible for. As explored in Data Quality When You Don't Understand the Data, knowledge gaps, a lack of comprehensive understanding of the data being handled and the business context it serves, can lead to misinterpretations and incorrect data processing. This knowledge deficit can stem from insufficient training, high turnover rates, or poor knowledge transfer practices. When knowledge is concentrated among a few individuals rather than distributed across the team, it creates risky dependencies.



Poor documentation exacerbates this issue, making it difficult for new team members to get up to speed or for others to fill in during absences. In fast-paced environments, teams often prioritize immediate deliverables over long-term quality improvements. This lack of time for collaboration and learning can lead to shortcuts that compromise data quality. The path forward involves building institutional knowledge through data profiling, creating documentation that travels with the data, and establishing quality circles where cross-functional teams regularly discuss data issues and share context.

Acceptance And Recovery.

Understanding these sins is critical because recognition is the first step toward recovery. These behaviors explain why, despite the availability of tools and methodologies, data quality remains a persistent struggle for many organizations.

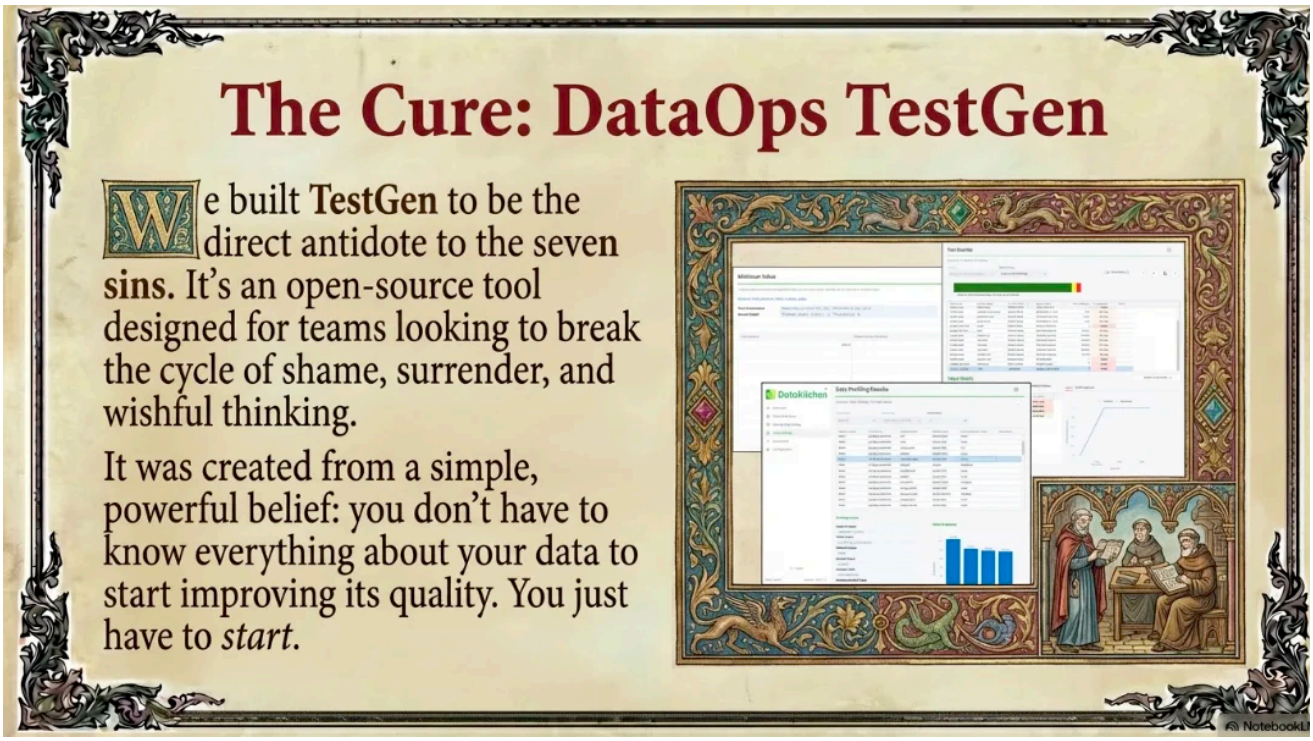


- They Prevent Proactivity: These sins trap teams in a reactive “firefighting” mode, where they spend significant time (up to 40%) responding to defects rather than building new value.
- They Erode Trust: When these sins lead to poor data quality, they undermine customer trust and confidence in decision-making, potentially costing organizations millions annually.
- They Mask Structural Issues: Sins like Denial and Ignorance hide the fact that technical teams and domain experts often operate in silos, speaking different languages and failing to share context.

If the mechanics (data engineers) are afraid to open the hood because they might be blamed for what they find (Shame), or if they ignore a rattling noise because “the radio still works” (Denial), the car will eventually break down on the highway. If they believe their job ends the moment the car leaves the garage (Avoidance) or refuse to perform maintenance because no one applauds an oil change (Laziness), the engine will fail. The Seven Sins are the bad habits of the mechanics that guarantee the car will crash, regardless of how expensive or well-designed the vehicle originally was.

Breaking Free from the Seven Sins

The path from data quality dysfunction to data quality leadership isn’t as long as you think. It starts with one person making one small improvement for one specific customer. From there, it’s about building momentum, measuring impact, and expanding your influence one success at a time. The tools are available, including open-source options that remove budget barriers, the methodologies are proven, and the need is obvious.



Getting your team to overcome these seven deadly sins isn't about shame or blame, it's about showing them there's a better way. When they can spot data problems before customers do, when the same issues stop breaking things week after week, and when those dreaded stakeholder calls turn into the occasional "hey, nice work", suddenly, caring about data quality doesn't feel like a burden anymore. It feels like the thing that makes their jobs less stressful. Your organization's data quality transformation is waiting for someone to take the first step.

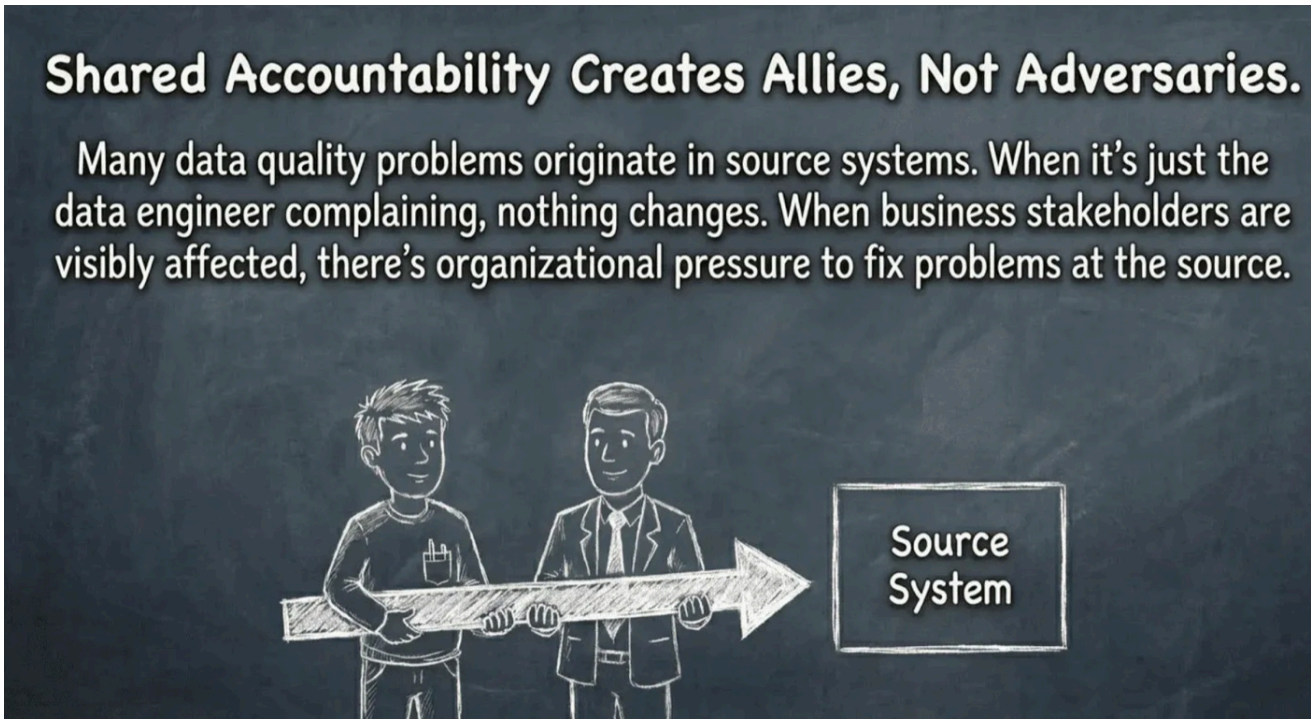
CULTURE AND ORGANIZATION

The Data Errors Shame Game

Read the original article at: https://datakitchen.io/blog/the-data-errors-shame-game-why-data-engineers-avoid-harsh-truths/

The Perverse Incentive Nobody Talks About

There is a rule that experienced data engineers learn without anyone ever teaching it directly: if you find the problem, you own the problem.

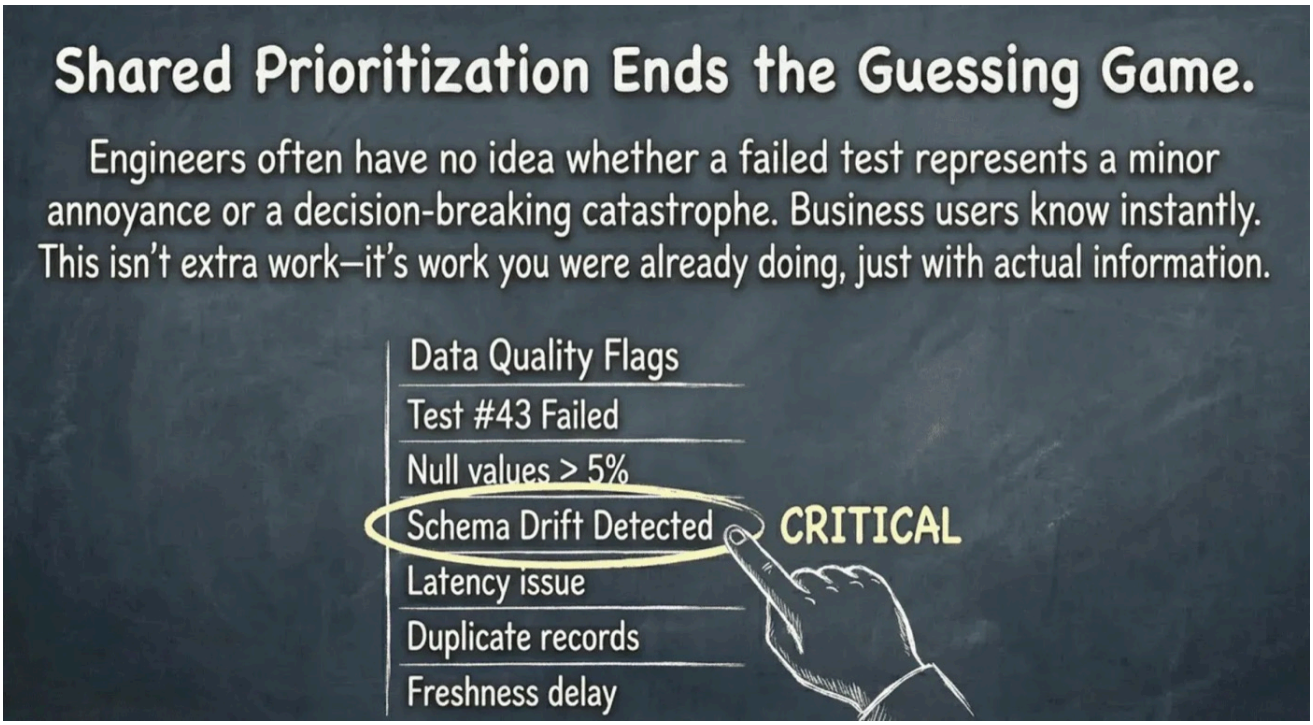


It does not matter that you did not create the bug. It does not matter that the data arrived broken from an upstream source. It does not matter that the schema changed without notice. The moment your name is attached to a finding, you become the responsible party in the eyes of whoever is paying attention. So the rational response, when you want to protect your reputation and your time, is simple. Do not look too hard.

This is how shame quietly destroys data quality practices. It is not a character flaw. It is a structural incentive that most data teams have built into their workflows without realizing it.

What Shame Looks Like in Practice

Consider a scenario that plays out, in some form, on data teams across every industry.



A Tuesday morning. The VP of Sales opens the regional performance dashboard for the weekly revenue review meeting. The numbers are off. Not slightly off. Off by a factor that makes last quarter look like a disaster compared to the prior year. She types a message into Slack: "Something is wrong with the numbers in the sales dashboard. Who is responsible for this?"

Three data engineers see the message. One of them, Marcus, built the pipeline that feeds the dashboard. He recognizes the symptom. He had noticed a few weeks earlier that the currency conversion table was not refreshing reliably on weekends, but the issue seemed intermittent and he had made a mental note to look into it after his current sprint. He says nothing in Slack for eleven minutes.

Another engineer, Priya, has never touched that pipeline but has general knowledge of the data infrastructure. She starts investigating to be helpful. She finds the stale currency table within four minutes. She posts in Slack: "Found it. The fx_rates table stopped refreshing last Friday. Fixing now."

Within twenty minutes, the question arrives: "How long has this been broken?" Then: "Has this affected reports before?" Then, from a director who was copied in: "How did this get past QA?"

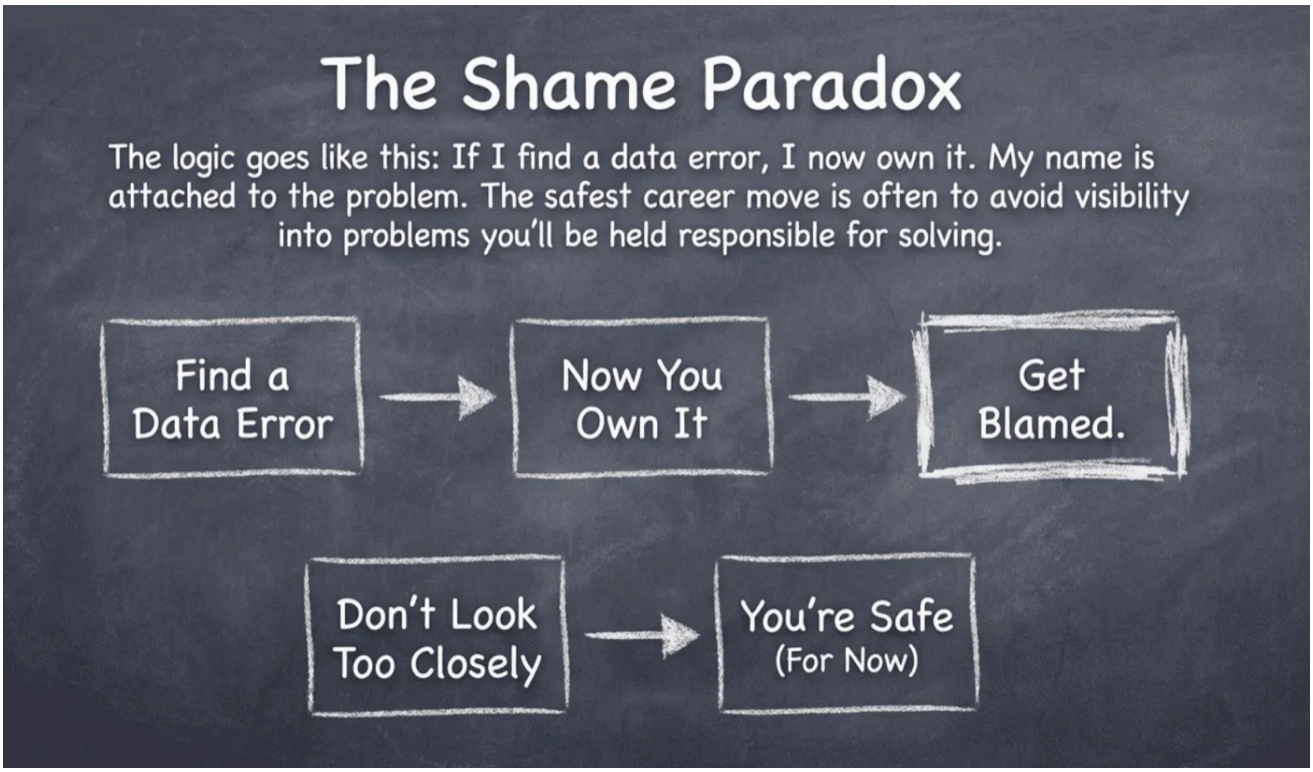
Priya, who fixed the problem, is now fielding the blame questions. Marcus, who knew there was a risk, has gone quiet. He eventually posts a brief acknowledgment that he had seen some instability in that table and was planning to address it. The word "planning" does not land well.

The VP of Sales does not get an apology. She gets a postmortem document two days later that lists technical root causes and uses passive voice throughout. The meeting is rescheduled. Trust in the dashboard drops. The team starts keeping a spreadsheet of "known issues" that they track separately from the official pipeline documentation, because putting things in official documentation means someone might ask why they have not been fixed.

This is the shame dynamic. It does not require a single villain. Everyone in this story is acting rationally within a broken system. Marcus was not malicious. Priya was not naive. The VP was not unreasonable. But the incentives guaranteed a bad outcome. The person who looked carefully got punished, the person who avoided looking got to stay quiet, and the organization learned nothing durable about why the problem happened or how to prevent the next one.

Why Shame Grows in Dark Places

The shame dynamic compounds over time in ways that are hard to reverse once they are established.



First, problems stop getting documented. Engineers learn that writing down a known issue creates a paper trail of accountability. The issue goes into someone's head instead of a ticket. When it eventually causes an incident, there is no record of when it was first observed.

Second, monitoring and alerting gets treated as a liability. If your monitoring system notifies the team about anomalies, those anomalies become visible. Visible problems create questions. So the monitoring system stays under-configured. Alert thresholds get set wide enough that only catastrophic failures trigger them. The team tells itself they are avoiding alert fatigue. They are actually avoiding scrutiny.

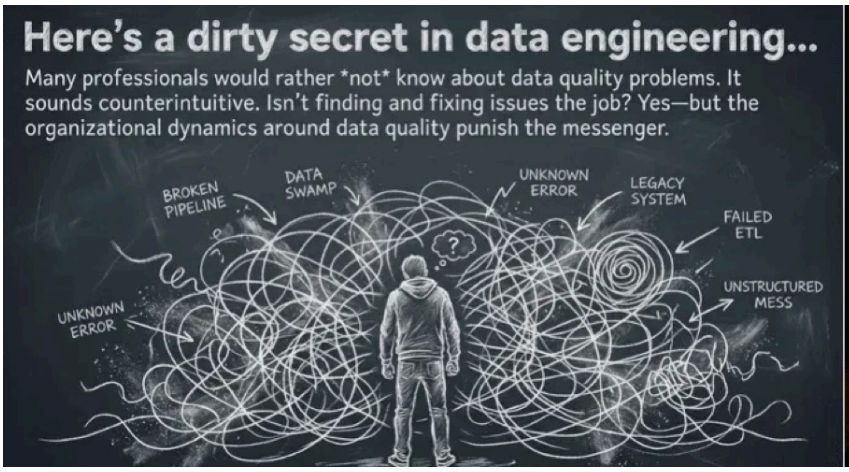
Third, data engineers stop probing unfamiliar datasets. They stick to what they know, which means they avoid discovering that something they are responsible for adjacent to, rather than directly responsible for, is quietly accumulating problems.

Fourth, the business learns not to trust the data, but to not say so directly. They build their own spreadsheets. They add "gut check" steps to every analysis. They stop raising quality issues because raising issues creates friction without resolution.

The result is an organization that operates on data it does not fully trust, staffed by engineers who are skilled at not finding problems, supported by tooling deliberately kept too quiet to reveal inconvenient truths.

Shame Thrives in Isolation and Dissolves in Partnership

The antidote is not better technology. It is structural redesign.

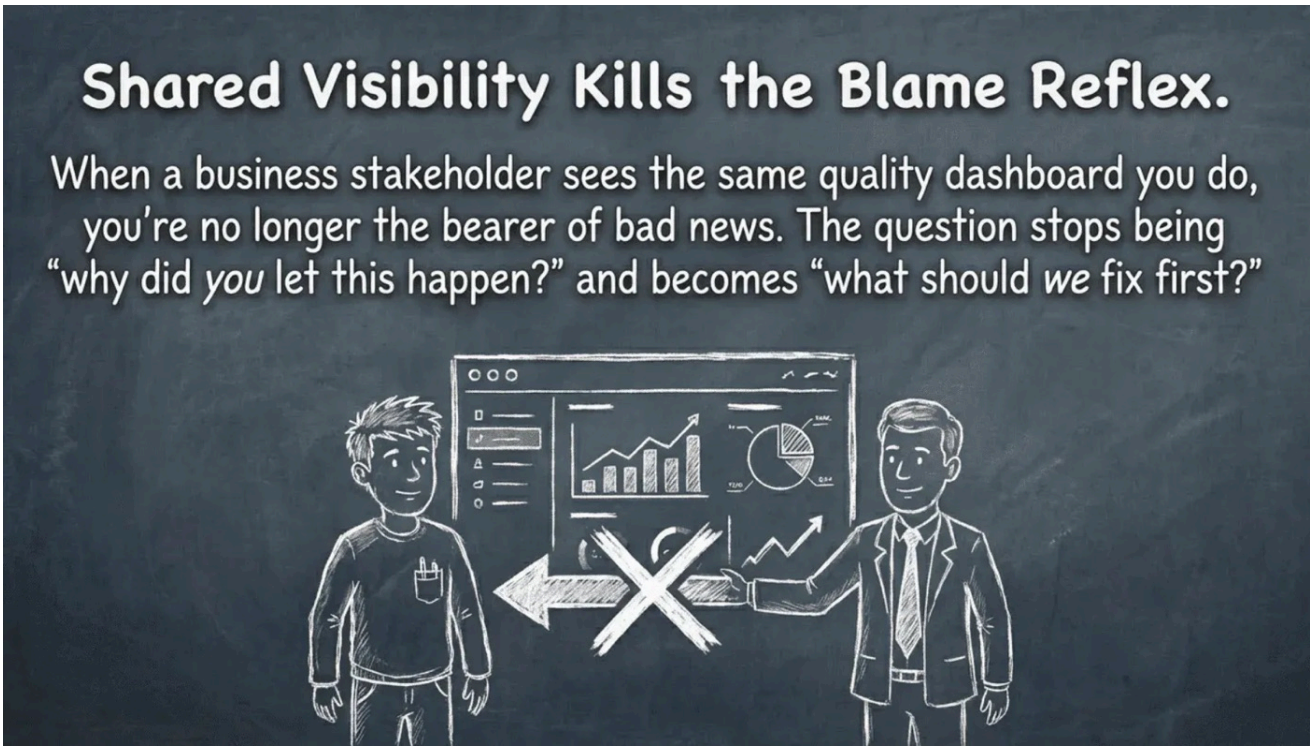


When a data engineer and a business analyst look at a quality problem together, from the start, the shame has nowhere to land. Nobody is in the hot seat. There is no interrogation. There is a shared problem that both parties are solving. The business user brings context about which data matters most and how errors affect decisions. The data engineer brings context about where errors originate and how the systems behave. Together they can solve problems that neither could solve alone, and neither is the singular target of blame.

This is not idealism. It is a structural observation. Shame requires a single person to be identified as responsible. Shared ownership distributes that responsibility in a way that removes the incentive to hide problems.

Structural Changes That Actually Work

The following changes are concrete and implementable without organizational approval from above. A team lead can introduce them at the team level. Many of them take a single sprint to put in place.



Define error rates as a team metric, not an individual metric. The question should never be "whose pipeline broke?" The question should be "what is the team's data quality error rate this week, and what are we doing about it?" When error rates are reported at the team level, finding a problem becomes a contribution, not a confession.

Hold blameless postmortems after every data quality incident. The blameless postmortem format, borrowed from software reliability engineering, asks: what failed in the system? What could be changed to make the system more resilient? It explicitly prohibits asking: who is at fault? This is not about letting people off the hook. It is about recognizing that individual blame produces defensiveness, and defensiveness produces silence, and silence guarantees the problem recurs. A good postmortem ends with specific changes to process, tooling, or documentation, not with a person being held accountable.

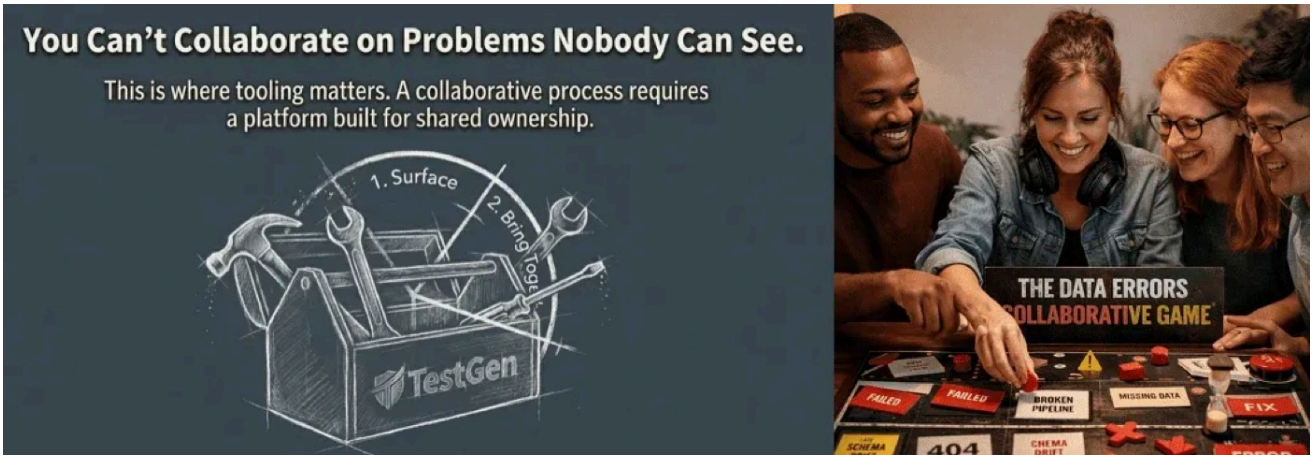
Create a shared known-issues register that is treated as a sign of team maturity, not team failure. Every team has known issues. The teams with no documented known issues are not better than other teams. They are hiding their known issues in their heads. A public register of known issues, reviewed in sprint planning, signals that the team is honest about its system's current state. It protects engineers from being blind-sided by problems they never knew existed. It gives business users realistic expectations.

Pair engineers with business users on quality review, not just on incident response. When Priya and the VP of Sales talk about the fx_rates table in a monthly data review, before it breaks, the conversation is collaborative rather than adversarial. The VP learns what conditions make that table unreliable. Priya learns how currency conversion affects downstream reporting. When the table eventually fails, and it will, both parties understand the system well enough to communicate clearly rather than retreating into blame and silence.

Give credit when problems are found proactively. This one costs nothing and matters more than most technical interventions. If an engineer finds a problem before it reaches a user and documents it in the team's quality register, that is a win. It should be recognized as a win in team standups, sprint reviews, and any other forum where work gets acknowledged. Organizations get the behavior they reward. If proactive problem-finding is rewarded, more proactive problem-finding happens. If it is punished with interrogations, it stops.

The Honest Assessment

Shame is not a data engineering problem. It is a human problem that data engineering has not figured out how to manage. The stakes are high because data quality failures are often visible to the whole business in ways that software bugs are not. A broken feature is caught in QA. A broken dashboard is caught in the Monday revenue review.



But the shame dynamic is not inevitable. Teams that have deliberately redesigned their ownership models, their postmortem practices, and their credit structures tend to have better data quality over time. Not because they hired better engineers, but because they built systems where finding problems is safe. Where the person who finds the bug is the hero, not the suspect.

That shift does not start with a platform or a policy memo. It starts with how a team lead responds the next time something breaks in production.

That response sets the precedent. Make it count.

CULTURE AND ORGANIZATION

Is Your Team in Denial about Data Quality?

Read the original article at: https://datakitchen.io/blog/is-your-team-in-denial-of-data-quality/

Play Denial Bingo First

Before reading any further, scan the list below. Check off any phrase you have heard on your team in the last six months. Check off any phrase you have said yourself.

- "I haven't heard any complaints."
- "We passed QA."
- "Users have a workaround."
- "I trust this data because I built it in Excel."
- "That table is legacy. Nobody uses it anymore."
- "The business knows to add ten percent on top."
- "It only breaks during the end-of-quarter crunch."
- "We've always done it this way."
- "If it were really wrong, someone would have flagged it."
- "The data looks right to me."
- "That's a reporting problem, not a data problem."
- "We'll clean that up in the next sprint."

If you checked three or more, your team is in denial. If you checked five or more, your team has been in denial long enough that it has developed coping rituals. If you recognized every single phrase, you are probably the person who hears these statements the most and has quietly stopped arguing about it.

Denial about data quality is not laziness. It is a rational adaptation to a situation where acknowledging the full extent of problems feels more threatening than managing around them. This chapter is about what denial actually sounds like in practice, why each rationalization feels true to the person saying it, and what to do in the first 30 days after your team admits there is a real problem.

The Bingo Squares, One by One

"I Haven't Heard Any Complaints"

This phrase is load-bearing in a lot of data teams. It is used to close conversations, end escalations, and justify not investigating anomalies. The logic seems reasonable: if users were finding errors, they would tell us.

They would not. Users stop reporting errors when reporting errors produces friction without resolution. If the first three times a business analyst flagged a data inconsistency, she was told the data was correct and she should check her filter settings, she stopped reporting on the fourth incident. She added a column to her spreadsheet called "adjust for known pipeline lag" and moved on. The absence of complaints is evidence of adaptation, not accuracy.

The engineering team's silence-as-signal fails the moment you ask the business analyst directly: "How much do you trust the numbers in this dashboard?" The answer is usually illuminating.

"We Passed QA"

QA, in most data contexts, means that a set of tests were written once, ran green at the time of deployment, and have not been revisited since. This is not a quality guarantee. It is a snapshot of a system that no longer looks the same.

The upstream schema changed. The volume increased by 40 percent. A new source was added without updating the tests. The definition of "active customer" shifted after a reorg, but the pipeline still uses the old definition. QA passing in March does not mean the data is correct in November.

"We passed QA" is one of the most confident-sounding statements a team can make. It is also one of the least meaningful.

"Users Have a Workaround"

The existence of a workaround should be alarming. Instead, it is often treated as a resolution. "Oh, the revenue reconciliation? Yeah, finance adds a correction factor every Monday morning. They've been doing it for two years."

Two years. Every Monday. A manual correction factor applied to production reporting data, because the pipeline has a known problem that nobody has fixed.

This is not a workaround. This is a second job that finance invented because the data team's job is not complete. The fact that users have adapted to broken data does not make the data less broken. It makes the problem harder to see, which means it will grow.

"I Trust This Because I Built It in Excel"

This one shows up less often in engineering conversations and more often in the hybrid environments where business teams own part of the data supply chain. An analyst built a summary model in Excel. It joined three source tables, applied some logic, and produced a metric that the executive team now uses in monthly reviews.

The analyst trusts it because she understands it. But nobody else can audit it. Nobody receives an alert when one of the source tables fails to refresh. Nobody has documented the business logic outside the spreadsheet. When she changes jobs, the model stays, and the institutional knowledge leaves with her.

This is not a criticism of spreadsheets as tools. It is a statement about systems. Any critical data asset needs to be testable, auditable, and observable by people who did not build it.

"That Table Is Legacy. Nobody Uses It Anymore."

Unless you have confirmed this by checking query logs, you do not know that nobody uses it. Somebody's script runs at 6 AM against that table. Somebody's reporting layer joins to it. It is referenced in a dashboard that the marketing team keeps open as a tab because they bookmarked it three years ago and still occasionally screenshot it for QA reviews.

"Legacy" is a word teams use to describe systems they are not actively managing. It is not the same as "decommissioned." Until a table is actually removed, it is a liability. If it contains bad data and someone is still querying it, it is producing bad outputs.

"The Business Knows to Add Ten Percent on Top"

This is perhaps the most dangerous rationalization of them all. It means the business has quantified the error and accepted it as a permanent cost of doing business with your data. It also means the adjustment is informal, inconsistently applied, and has no documentation. When a new analyst joins the team, nobody tells her about the ten percent. Her first quarterly report is wrong. Everyone is surprised that she did not know.

Informal correction factors are organizational memory stored in the worst possible place: the heads of the people who have been around longest.

"It Only Breaks During End-of-Quarter Crunch"

Translation: it breaks precisely when the data matters most. End-of-quarter is when executives review actuals, when sales teams need accurate pipeline numbers, when finance closes the books. The statement "it only breaks during the crunch" is an acknowledgment that the system fails under load at exactly the moments when failure is most consequential.

This is not a minor quirk. This is a architectural risk that has been accepted without being named.

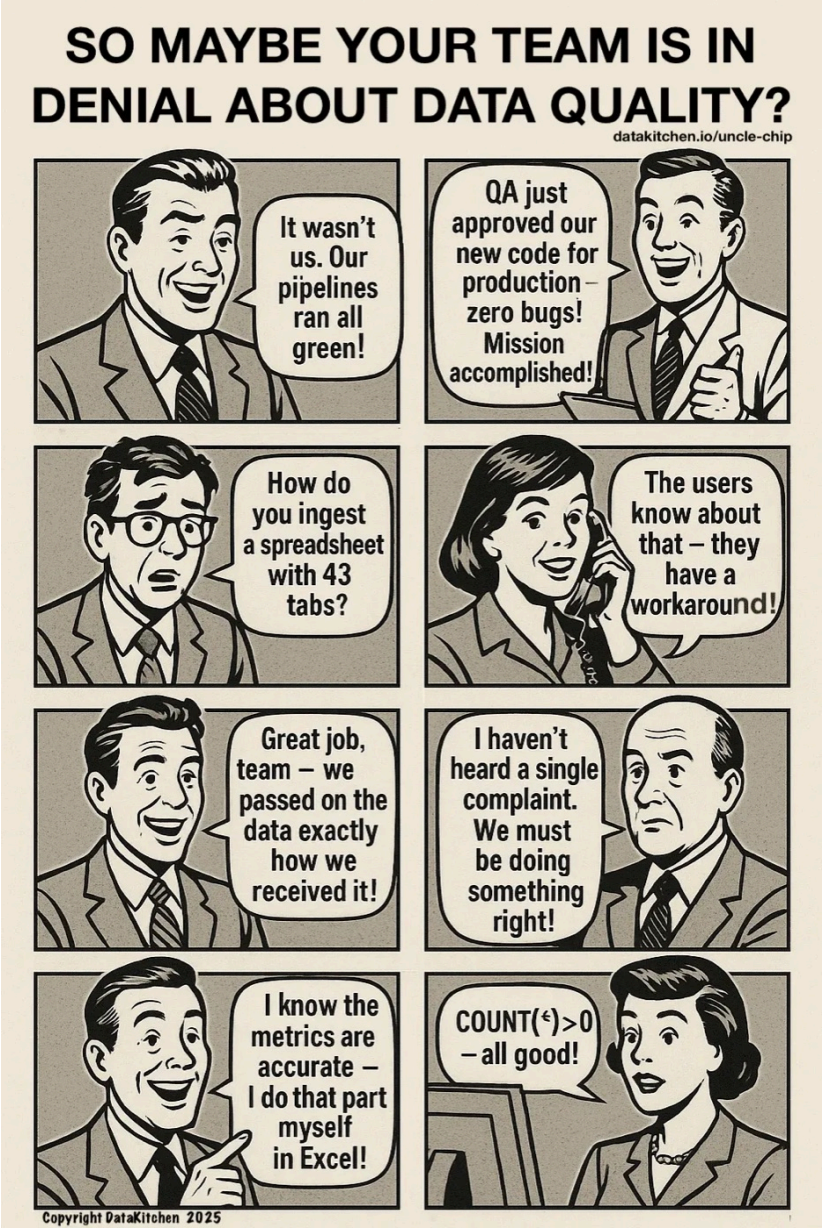
"We'll Clean That Up in the Next Sprint"

The next sprint never arrives. Not because teams are dishonest when they say this, but because sprint planning is driven by incoming work, not by accumulated technical debt. Each sprint brings new priorities. The cleanup task stays in the backlog, gets deprioritized, and eventually becomes so old that nobody is quite sure whether the problem it describes still exists in the same form.

Technical debt in data systems compounds because data is cumulative. A schema inconsistency introduced twelve months ago has been copied into downstream tables, referenced in reports, and baked into analyst workflows. Cleaning it up in the next sprint now requires coordinating with five teams instead of one.

Why Denial Is Hard to Break

Every statement on the bingo card feels true to the person saying it. The data engineer who says "users have a workaround" is not lying. The workaround exists. The business analyst who says she trusts her Excel model is not being naive. She built it carefully and it has worked reliably for her purposes.



Denial works because it allows work to continue. A team that stopped to address every data quality signal would get paralyzed. Some amount of rationalization is adaptive. The problem is when rationalization becomes structural, when the team's operating model depends on not looking at problems too carefully.

The signal that denial has become structural: when the team's answer to every quality concern is drawn from the bingo card, and those answers stop producing curiosity, investigation, or change.

The 30-Day Action Plan

Once a team recognizes it is in denial, the next question is what to do. Here is a concrete sequence.

Week 1: Call the meeting. One person needs to initiate it. This is usually the data team lead, but it can be any senior engineer or an analyst who has been quietly tracking problems for months. The meeting has one agenda item: "We are going to be honest about the state of our data quality." It is not a blame session. It is an assessment session. The output is a list of the top ten data assets that the team is least confident about.

Bring query logs, incident history, and a short user survey if possible. The user survey has one question: "On a scale of 1 to 5, how much do you trust the data you use most often in your work, and what is one thing you do to compensate for data you do not fully trust?" The answers to this survey will be educational.

Week 2: Measure what you have. For each of the ten flagged assets, document: when was the last time a quality test was run? What does that test check? When was it last updated? Who gets notified when it fails? How many open issues are associated with this asset? You are not fixing anything yet. You are establishing a baseline.

At the end of week two, you should know: how many of your ten highest-risk assets have no active testing. The answer is almost always most of them.

Week 3: Fix the worst visible problem. Pick the one issue from your list that is causing the most active pain to a business user right now. Fix it fully. Not with a workaround. Fix the root cause. Document the fix. Write a test that would have caught the original problem. Add that test to your CI pipeline.

Then tell the business user you fixed it. Be specific about what was wrong and what now prevents it from happening again. This builds trust and demonstrates that the team's assessment process produces actual results, not just documentation.

Week 4: Change one process. Choose one practice that allowed a denial pattern to persist and replace it with something that makes problems visible. Options include: adding a data quality section to your weekly team standup, setting up a shared dashboard that shows test pass rates for your ten flagged assets, or establishing a rule that any "known issue" must be in writing in a shared register within 48 hours of being identified.

The point of week four is not to solve everything. It is to establish that the team's operating model has changed in a visible, sustained way.

Day 31 and beyond: Report the metric. Every month, report your team's data quality error rate to stakeholders. Not to be punished for it. To demonstrate that you are measuring it, tracking it, and treating improvement as an ongoing goal. Teams that report quality metrics publicly build credibility over time. Teams that stay quiet about quality metrics are in denial about them.

One Final Observation

Denial does not mean a team is incompetent. It means a team has not yet decided that the cost of honesty is worth paying. Once a team makes that decision and sees that honesty leads to better outcomes, the denial patterns tend to dissolve on their own. The bingo phrases stop making sense when

the team has evidence that looking at problems carefully produces results instead of blame.

That evidence takes about 30 days to create. The hardest part is calling the first meeting.

CULTURE AND ORGANIZATION

The Ostrich Problem: When Your Data Team Stops Caring After Deployment

Read the original article at: https://datakitchen.io/blog/the-ostrich-problem-your-data-team-thinks-their-job-ends-at-deployment/

The Number That Should Alarm You

According to Wakefield Research, data engineering teams spend 40 percent of their time firefighting data issues in production. That is not 40 percent of a slow week. That is 40 percent of every week, on average, across the teams surveyed. Two days out of every five, doing reactive work that could have been preventive work.

Think about what your team built in the last quarter. Now imagine that every engineer on the team had 40 percent more of their time available. That is the hidden cost of reactive data quality management.

The cause is not incompetence. It is a pattern of behavior that has a name: the Ostrich Problem. Once a pipeline deploys, the team treats their job as complete. The code ships, the tests go green, and attention moves to the next project. What happens to that pipeline in production is someone else's problem, or more precisely, it is everyone's problem when it breaks and nobody's problem until it does.

This is how you end up spending your Mondays firefighting instead of building.

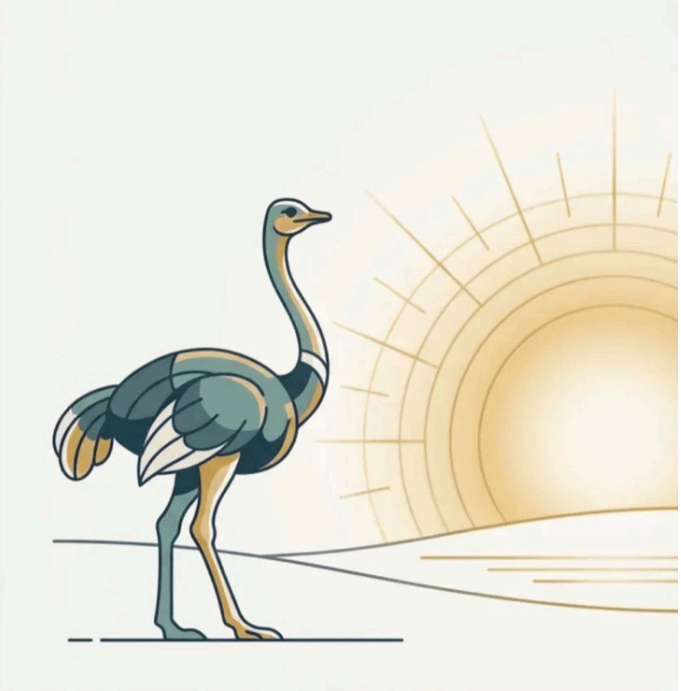
Why Teams Go Ostrich After Deployment

The Ostrich Problem is not irrational. It emerges from the way data engineering work is structured and rewarded.

They didn't get into engineering to disappoint people. They just need a reason to look up.

Getting your team to pull their heads out of the sand isn't about shame or blame—it's about showing them a better, less stressful way to work. When they can spot problems before customers do, the same issues stop breaking things, and stakeholders start saying "nice work," caring about quality stops feeling like a chore.

Start small, gather the evidence, and let the results do the convincing.



Deployment is a clear finishing line. There is a pull request, a review, a merge, a deploy. The pipeline works. The tests pass. The product is shipped. This feels like completion, and in software terms, it is. The job of building the pipeline is done.

But a data pipeline is not a finished product at the moment of deployment. It is a system that interacts with a changing world. Upstream schemas change without notice. Source system behavior changes as usage patterns shift. Business logic evolves and the old pipeline no longer captures what it was designed to capture. Data volumes grow past assumptions baked in at design time. What was correct at deployment becomes incorrect over time, not because of anything the engineering team did wrong, but because the world the pipeline models keeps moving.

Teams that treat deployment as completion are making a category error. They are treating a data pipeline like a static artifact rather than a living system that requires ongoing care.

The consequences follow predictably. Problems accumulate in production without anyone noticing. By the time a user flags an anomaly, the issue has been present for days or weeks. Debugging is harder because the blast radius is larger and the trail is cold. The fix takes longer. Other work gets disrupted. The 40 percent of time that should be going to new work goes to investigation and repair instead.

The Monitoring Gap

Most data teams have some monitoring. The problem is that monitoring is often configured to catch catastrophic failures rather than quality degradation.

Step 2: Demolish the 'We Don't Have Time" Excuse.

Resistance kicks in here. The team is slammed. Don't propose a massive initiative; propose a small experiment.

The Plan

1. **Pick a few high-pain tables.**
2. **Add data quality tests and let them run for a few few weeks.**
3. **Start a bi-weekly "quality circle"** to review what the tests and customers found.

The Secret Weapon

Tools like TestGen automatically generate several dozen data quality tests with the push of a button, providing 80% test coverage instantly. The other 20% can be built from simple UI templates.

A pipeline that crashes sends an alert. A pipeline that runs successfully but produces data with a 15 percent row count drop, a new source of nulls in a critical column, or a set of duplicate keys will often run silently. The monitors are watching the plumbing, not the water quality.

This gap exists because quality monitoring requires more context than infrastructure monitoring. To know that a 15 percent row count drop is a problem, you need to know what a normal row count looks like. To know that a null rate is rising, you need a baseline. To catch duplicate keys, you need to know which keys should be unique and under what conditions.

This context does not configure itself. It requires someone on the team to sit down with the dataset, understand its semantics, and define what good looks like. That work is often skipped after deployment because the team has already moved on.

The result is a monitoring system that provides false confidence. The alerts do not fire, so the team assumes things are fine, when in reality the monitors are not watching for the things that matter.

A Framework for Deciding Where to Start

The practical question is not whether to instrument your pipelines for quality. The practical question is where to start when you have limited time and dozens or hundreds of tables to consider.

The following framework narrows the selection to a manageable pilot set. It uses three criteria: downstream impact, change frequency, and current test coverage.

Downstream impact is a measure of how many decisions or users depend on a given dataset. A table that feeds one internal report used monthly has low downstream impact. A table that powers the main customer-facing analytics dashboard used daily by the sales team has high downstream impact. To assess this, look at query logs. Which tables are queried most frequently? Which tables are joined into the most downstream models? These are the datasets where a quality failure causes the most damage.

Change frequency is a measure of how often the data changes in ways that could introduce errors. Tables that receive updates from multiple upstream sources, tables fed by third-party APIs, and tables that depend on external data loads are higher risk than static reference tables or tables generated by internal systems under your team's direct control. High change frequency means more opportunity for things to go wrong.

Current test coverage is the simplest assessment: what tests, if any, exist for this table today? A table with high downstream impact, high change frequency, and no existing tests is your first priority. A table with high downstream impact and high change frequency but solid test coverage is already somewhat protected.

Score each of your candidate tables on these three criteria and sort them. The tables at the top of the list are where your pilot should start.

What a Good Pilot Looks Like

A quality monitoring pilot should cover five to eight tables, run for four to six weeks, and produce a measurable outcome.

Step 1: Make the Invisible Impossible to Ignore.

Start by gathering the facts that prove there's a problem.

Survey Your Users: Send a quick poll. "Do you trust the data?" "Have you built your own shadow QA processes?"

Count Past Errors: Do some archaeology. Dig through tickets, bug reports, and Slack threads from the last month.

Get a Baseline Score: Use an open-source tool like **TestGen** to profile your data and get a quality dashboard running in under an hour. No massive project required.

Present this evidence—survey results, error history, and quality scores—to your team. They don't *want* to ship bad data; they just lack visibility.

Start with the tables at the top of your prioritized list. For each table, define three to five checks that represent the minimum bar for quality. These should cover: row count within expected range, null rates below defined thresholds for critical columns, no duplicate values in columns that should be unique, and referential integrity for any key joins. These four check types catch the majority of quality failures in production pipelines.

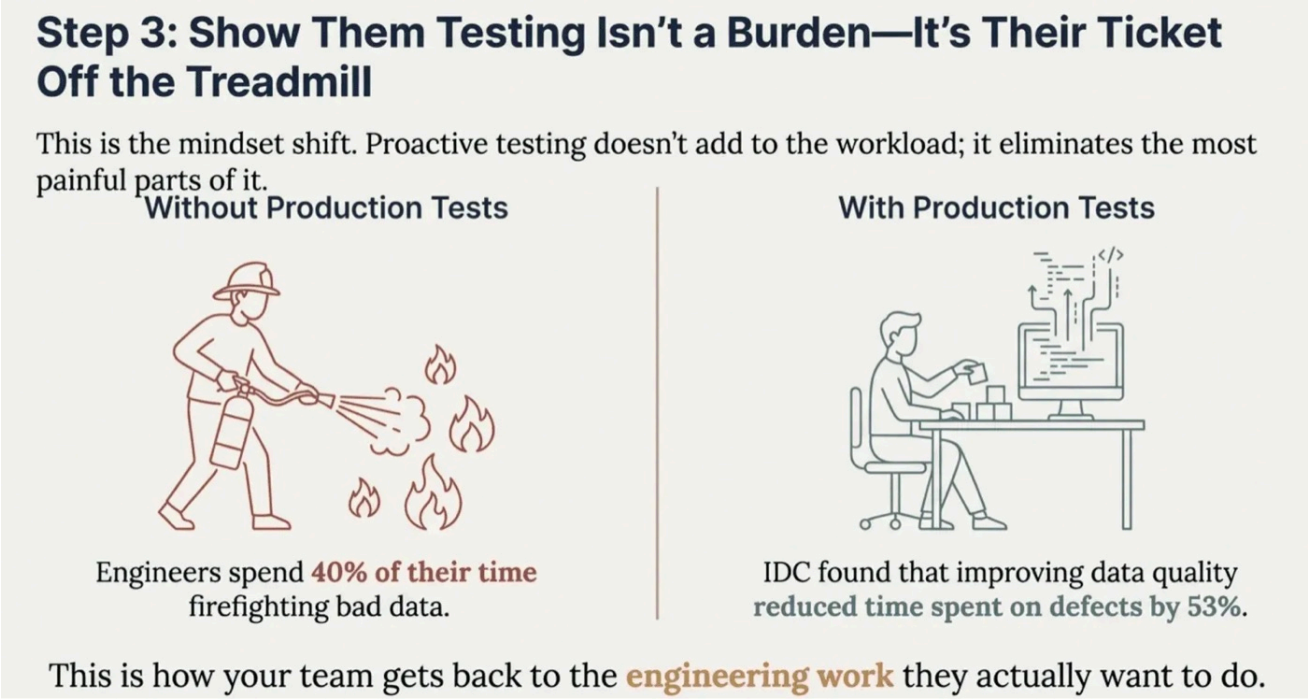
Run these checks daily for the first two weeks. Do not alert aggressively yet. Instead, record the results and look for patterns. You are building baselines. After two weeks, you will know what normal looks like for each table, and you can set alert thresholds that are actually meaningful rather than arbitrary.

In weeks three and four, activate alerting. Route alerts to the team channel, not to individuals, to avoid the shame dynamics discussed earlier in this book. When an alert fires, investigate it the same day. Document what you find. If the alert was a false positive, adjust the threshold. If it was a real problem, document the root cause and fix it.

At the end of six weeks, you should be able to answer three questions: How many true quality issues were caught before users found them? How much investigation time did early detection save? Which tables still need better coverage? Those answers form the case for expanding the program.

What "Done" Looks Like for a First Monitoring Pass

"Done" for a first pass is not zero failures. Data systems have failures. Done means the team has defined what good looks like for each asset in the pilot set, has automated checks that confirm or deny that definition on a daily basis, has a clear process for responding when those checks fail, and has a baseline that makes future anomalies detectable.



A team that achieves this for its five highest-priority tables has fundamentally changed its relationship with production quality. It has moved from reactive to proactive for those assets. The 40 percent firefighting figure does not drop to zero overnight, but it drops, and it keeps dropping as coverage expands.

The remaining work is expansion. Each sprint, add a few more tables to coverage. After six months, the team's highest-impact assets are all instrumented. After a year, quality monitoring is part of the standard delivery process, not an afterthought applied to pipelines that have already been in production for months.

The Head Has to Come Out of the Sand

The Ostrich Problem persists because burying the head feels rational in the short term. There is always a new pipeline to build, a new request from the business, a new priority that claims urgency. Quality monitoring of existing systems competes badly against the visibility and reward that comes from shipping new work.

But the math is unambiguous. If 40 percent of engineering time is going to firefighting, and a modest investment in proactive monitoring cuts that to 20 percent, the team has just created the equivalent of a full additional engineer's worth of productive capacity. That engineer does not need a headcount

approval. She is already on the team, currently spending her Mondays putting out fires that could have been prevented.

The choice between deployment-as-completion and deployment-as-beginning is not a philosophical one. It is a practical one, with a measurable consequence either way. Teams that keep their heads in the sand keep paying the 40 percent tax. Teams that instrument their production systems convert that tax into capacity.

Start with five tables. Define what good looks like. Check it every day. The head comes up, and the Monday mornings get better.

CULTURE AND ORGANIZATION

The Data Quality Revolution Starts with You

Read the original article at: https://datakitchen.io/blog/the-data-quality-revolution-starts-with-you/

Pick One Customer

Most data quality initiatives fail before they start. They fail because they are scoped too broadly: fix data quality across the enterprise, stand up a governance framework, instrument every pipeline. These goals are real and eventually necessary. They are also paralyzing. Nobody knows where to begin, the budget never arrives, and the initiative dies in a committee.

There is a simpler approach: pick one customer.

Not a customer segment. Not a use case category. One specific person or team with a specific, named data pain point. Find out exactly what data they depend on, where that data comes from, and what happens when it is wrong. Fix that journey end to end. Make it work reliably. Then tell them what you did and what is different now.

That conversation is the beginning of a data quality revolution. It does not require executive sponsorship. It does not require a platform purchase. It requires identifying one person who is currently suffering because of data quality, and deciding to solve their problem completely.

A Concrete Example: The Finance Team on Monday Morning

Here is what this looks like in practice.

The finance team at a mid-sized SaaS company reconciles revenue every Monday morning. The analyst responsible, call her Elena, pulls data from three places: the billing system, the CRM, and a data warehouse table called `fct_revenue_monthly`. She spends the first two hours of every Monday comparing these three sources and manually resolving discrepancies. In a good week, the discrepancies are small and take twenty minutes to resolve. In a bad week, the numbers from the CRM and the billing system disagree by more than one percent of total revenue, and Elena has to trace individual transactions to find the source of the mismatch.

Elena has been doing this for eighteen months. She has a spreadsheet of "known adjustment factors." She has filed three bug reports, two of which were closed as "works as expected." She does not trust the data warehouse revenue table without manually confirming it against the billing system first. She has stopped reporting discrepancies because nobody acted on them.

This is your customer.

Tracing the Data Journey End to End

To fix Elena's problem, you need to understand her data journey completely. This is different from understanding your pipeline architecture. The data journey starts where the data originates and ends where Elena makes a decision.

For the revenue reconciliation use case, the journey looks like this:

1. A subscription is created or modified in the CRM.
2. A billing event fires in the billing system when a charge is processed.
3. A nightly ETL job loads both sources into the data warehouse.
4. A transformation model in the warehouse joins the two sources and applies revenue recognition logic to produce `fct_revenue_monthly`.
5. Elena queries that table every Monday alongside direct exports from the CRM and billing system.

At each step, there are failure modes. The CRM record for a subscription may be updated after the billing event fires, creating a timing mismatch. The nightly ETL job may fail silently for individual records when it encounters unexpected characters in a text field. The revenue recognition logic may apply a rule that was correct twelve months ago but was superseded by a pricing change that nobody updated in the model. The join between the two sources may produce duplicates when a billing event is retried after a payment failure.

Any of these can cause the discrepancy Elena sees on Monday morning. Most of them have been present long enough that nobody can easily trace which one is responsible for which portion of the mismatch.

What Test Coverage Looks Like for This Journey

The goal of test coverage is not to test everything. It is to make each failure mode visible the moment it occurs, rather than when Elena notices it on Monday.

For the revenue reconciliation journey, a minimal test suite looks like this:

At the source level: confirm that the billing system export contains no records with null transaction IDs, that the CRM export row count is within five percent of the prior week's count, and that no records have a created-date in the future. These checks run immediately after the nightly load completes.

At the transformation level: confirm that `fct_revenue_monthly` has no duplicate billing transaction IDs, that the total monthly revenue in the warehouse is within one percent of the sum in the raw billing system table, and that the count of distinct subscription IDs in the warehouse matches the count in the CRM

within a defined tolerance. These checks run after the transformation model completes.

At the delivery level: a Slack notification to Elena every Monday at 7 AM confirming that all checks passed and the data is ready, or flagging which check failed and what was found. If the checks pass, Elena does not need to manually verify. If a check fails, Elena knows about it before she opens her laptop.

This is not a comprehensive quality program. It is test coverage for one journey, designed around one customer's actual use case. It takes a competent data engineer about two days to implement.

What Happens When It Fails

The first time the monitoring catches a real failure before Elena does, the dynamic changes.

Suppose on a Thursday night the nightly ETL job produces a revenue table where the total is 1.4 percent below the billing system raw total. The check fires. The on-call engineer gets an alert at 10 PM. She investigates Friday morning and finds that 23 records failed to load due to a currency symbol in the transaction notes field that the ETL parser did not handle. She fixes the parser, reloads the records, and confirms the total now reconciles.

On Monday morning, Elena gets the confirmation Slack message: all checks passed, data is ready.

She does not spend two hours reconciling. She runs her analysis directly from the warehouse. The report goes to the finance director an hour earlier than usual.

Elena notices. She asks what changed. When the data engineer explains the monitoring system, Elena asks whether it covers the refund adjustment logic too, because that has been another pain point. Now the conversation is about expanding coverage, not about filing bug reports that get closed.

What Happens After the Fix

The compounding effect of fixing one customer's journey is not only that their life gets better. It is that the team learns the pattern.

The data engineers who built Elena's monitoring now understand what end-to-end coverage looks like for a real use case. They have written the tests, seen them catch a real issue, and watched the business outcome improve. They have a template. The next customer journey is faster to instrument.

Elena, now a visible success case, can speak to her colleagues in finance about whether the data in other reports is reliable. She is an advocate, not a skeptic. When the data team asks to expand coverage to the customer success team's renewal pipeline, Elena can describe what the experience was like from the user side.

The revolution propagates through credibility, not through mandate. One fixed journey builds the credibility to attempt the next one.

Starting Without Permission

The reason to start with one customer rather than an enterprise initiative is that you do not need permission to solve one person's problem.

You do not need a governance committee to approve a test suite for five tables that feed Elena's Monday report. You do not need a platform procurement process. You need to find Elena, spend an hour understanding her workflow, two days building the coverage, and one conversation confirming that the fix works.

Every team has an Elena. She might be in finance doing revenue reconciliation. She might be in operations tracking inventory levels. She might be in marketing trying to match ad spend to attributed revenue. She is the person who has the most detailed knowledge of your data's failure modes because she has been compensating for them the longest.

Find her. Fix her problem. Then do it again.

That is how the revolution starts. Not from above, not from a strategy document, but from one data engineer deciding that one customer's Monday morning matters enough to fix it completely.

CULTURE AND ORGANIZATION

The Five Organizational Sins of Data Teams

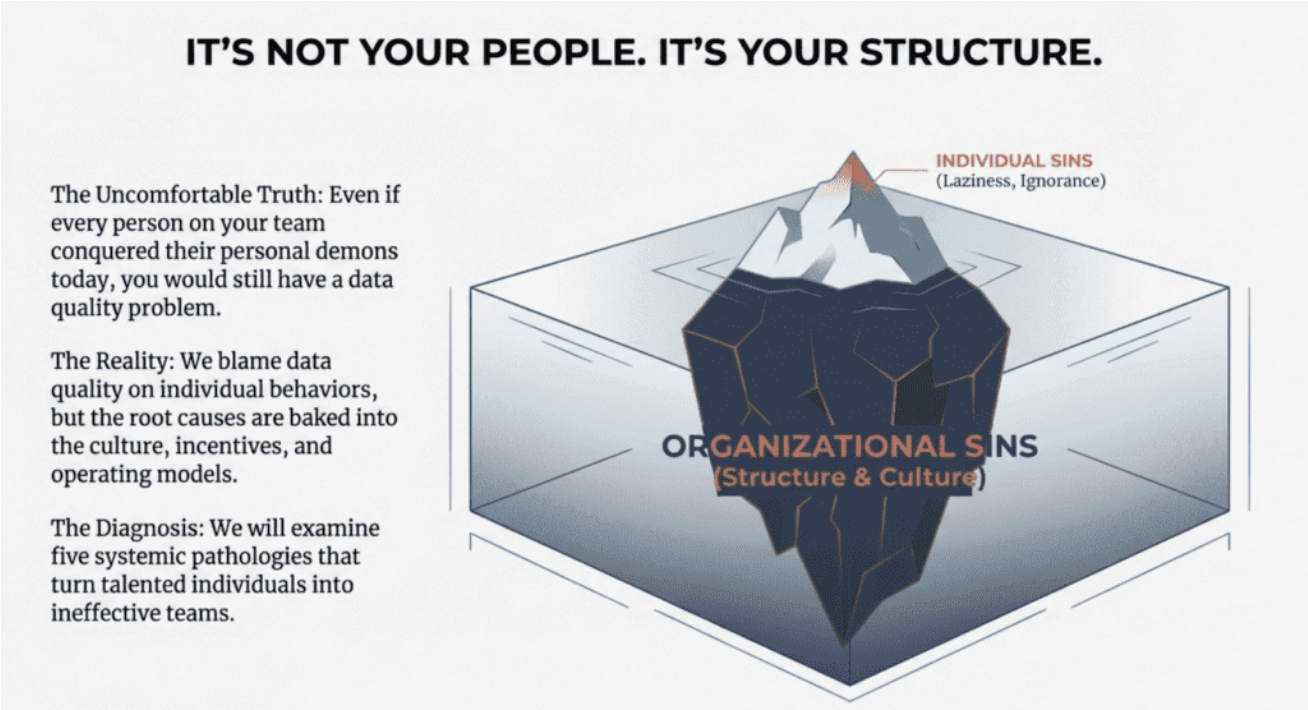
Read the original article at: https://datakitchen.io/blog/the-five-organizational-sins-of-data-teams/

In the first part of this series, we explored the Seven Deadly Sins of Data Quality, focusing on the individual behaviors that sabotage data teams: shame, denial, avoidance, passivity, laziness, gluttony, and ignorance. But here is the uncomfortable truth that nobody wants to talk about: even if every person on your data team somehow conquered their personal demons tomorrow, you would still have a data quality problem. The issue is not just about people. It is about how organizations structure their teams, define success, and approach the very nature of data work itself.

Organizational sins are more challenging to diagnose and even more complex to fix because they are baked into the culture, the incentive structures, and the operating models that everyone takes for granted. These are the systemic dysfunctions that turn talented individuals into ineffective teams and promising initiatives into expensive disappointments. In this follow-up post, we examine five organizational sins that plague data teams: low relational coordination, blame culture, data blindness, project thinking, and lack of process curation. Recognizing these patterns is the first step toward building data organizations that actually work.

Sin One: Low Relational Coordination

Data organizations face unique challenges in fostering positive interaction and communication, as tools and workflows often promote isolation. Groups that should be partners working toward shared objectives often behave more like warring tribes competing for resources and pointing fingers when things go wrong. As explored in Warring Tribes into Winning Teams, geographical, cultural, and language barriers compound with technology-driven silos to create what management scientists call low relational coordination. The data engineers, analysts, scientists, and business users who need to collaborate seamlessly instead operate in isolated bubbles, communicating through emails and handoffs rather than a genuine partnership.

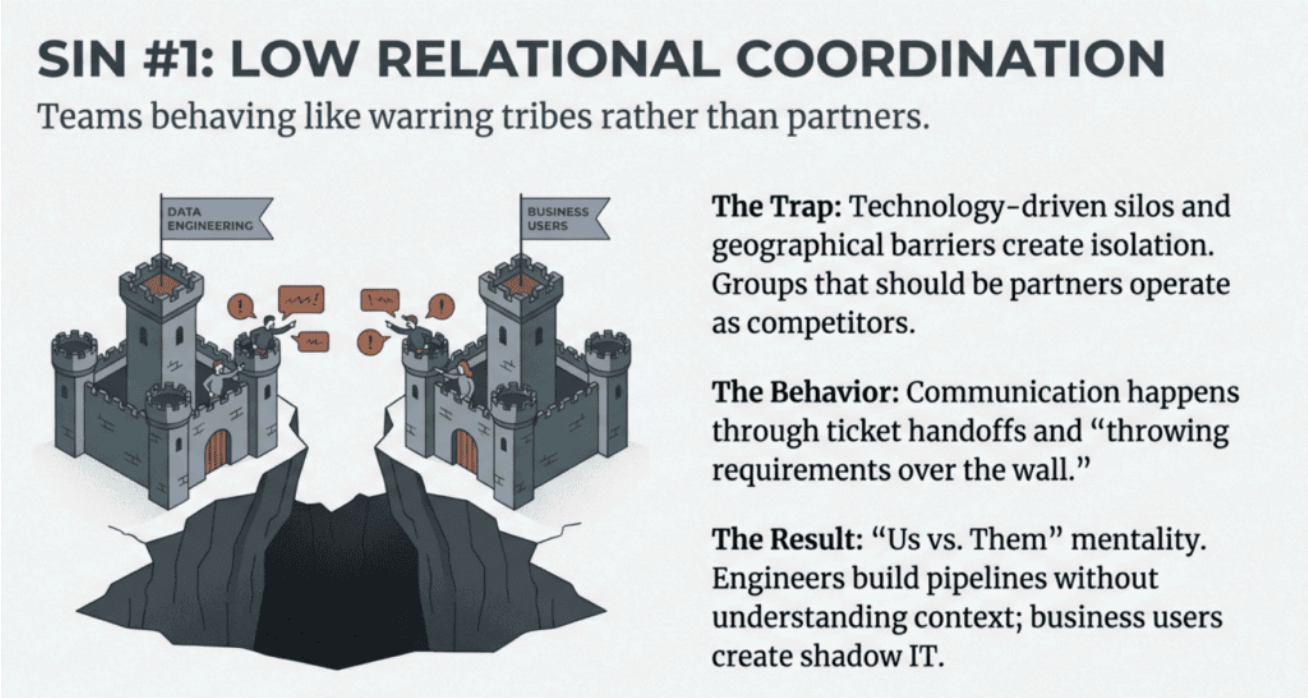


The consequences ripple through every aspect of data operations. Analytics cycle times stretch as teams wait for responses from groups they barely know. Data quality suffers because no one has visibility into the entire pipeline. Governance becomes a bureaucratic exercise rather than a shared commitment. Employee retention plummets as talented people tire of fighting the same battles over and over. The solution requires more than better communication tools. Organizations must restructure their workflows to build strong role relationships and coordination that enable high performance. DataOps provides a framework for doing exactly that, using automation and process design to facilitate rather than hinder the communication between groups.

Sin Two: Blame Culture

In the world of data and analytics, one skill stands timeless and universal: the art of blaming someone else when things go sideways. As satirized in The Art of Data Buck-Passing 101, data teams have elevated finger-pointing to a dark art form. When data issues surface, development teams blame QA. QA

blames operations. Operations blames the data providers. Everyone invokes the sacred incantation of "garbage in, garbage out" while carefully positioning themselves outside the blast radius. The Spiderman pointing meme has become the unofficial mascot of data team retrospectives.



This blame culture is not just demoralizing; it is organizationally corrosive. When people fear being blamed for problems, they stop looking for problems. They push untested code to production because once it is deployed, someone else becomes responsible. They craft emails that document decisions in ways designed to deflect future accountability rather than facilitate current collaboration. The energy that should flow into fixing root causes instead flows into constructing elaborate defenses. Breaking this cycle requires shared visibility into data pipelines, clear ownership models, and a cultural shift from "who broke it" to "how do we prevent it from breaking again." Until organizations address the blame game directly, they will continue playing it.

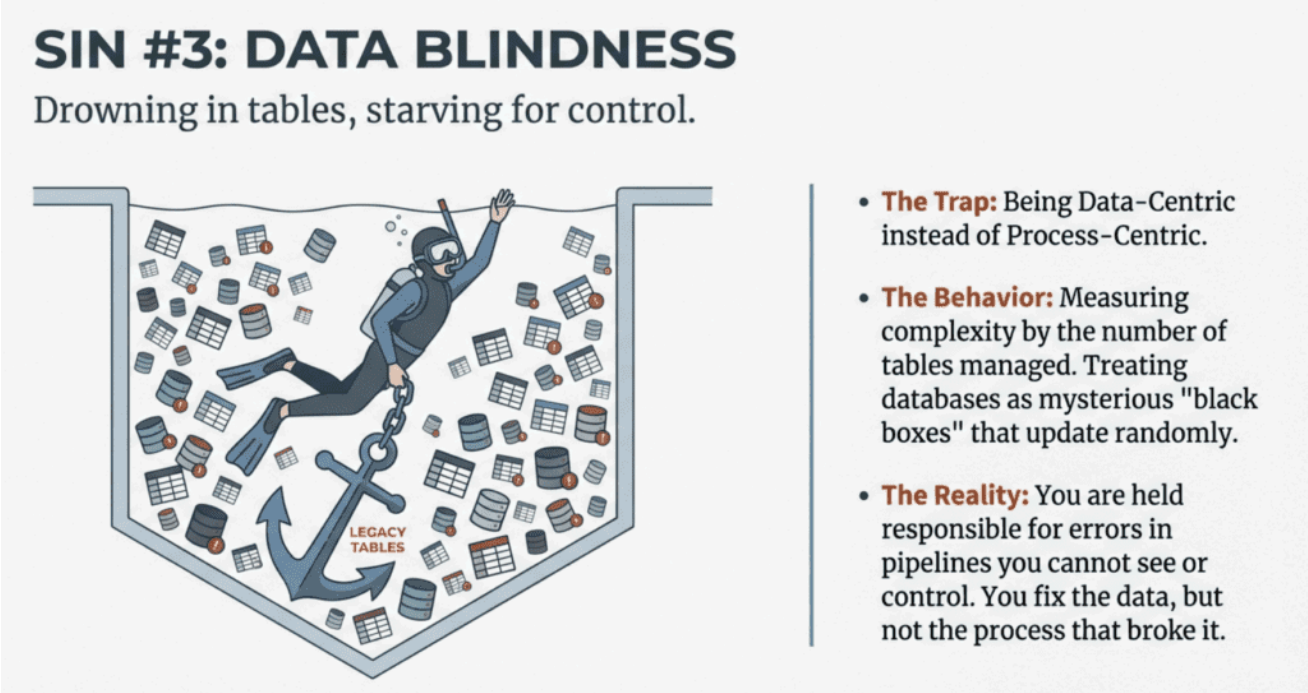
Sin Three: Data Blinders

Most data teams are data-centric rather than process-centric, and this distinction has profound implications for their effectiveness. As described in Are You Process-Centric or Data-Centric?, data-centric teams perceive complexity as directly proportional to the number of tables they manage. Their databases feel like mysterious, daunting black boxes full of tables updated at seemingly random frequencies by processes that are unknown or uncontrolled. Data engineers in these environments frequently respond to quality issues without direct control over the processes that cause the errors. They are held responsible for data problems but cannot actually fix or influence the pipelines that create them.

Process-centric teams operate completely differently. They focus on orchestrating and automating workflows, recognizing that robust, well-managed data processing pipelines inevitably yield reliable, high-quality data. Rather than focusing on individual tables, these teams allocate resources to ensure each pipeline is transparent, thoroughly tested, and easily deployable. They treat their pipelines as cattle, not pets, enabling them to quickly diagnose and rectify issues at the process level rather than getting lost in individual data table anomalies. The shift from data-centric to process-centric thinking is achievable, but it requires leadership commitment to unified orchestration, process transparency, and rigorous adherence to DataOps principles.

Sin Four: Project, Not Product Focus

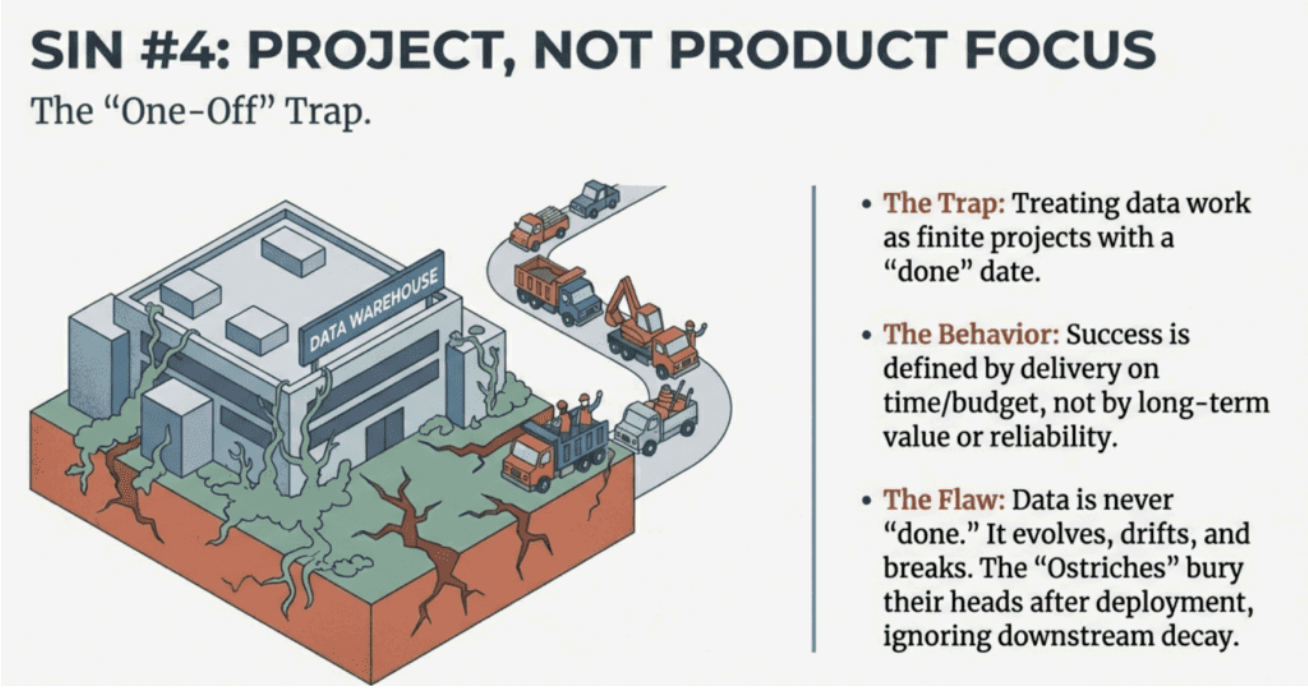
When data team leaders hear "data products," they immediately think of the stuff they produce: dashboards, datasets, models, and warehouses. But as argued in You're Thinking About Data Products All Wrong, focusing on the "what" completely misses the massive shift that data products really represent. The truth is that data products are not about what you create but how you build, maintain, and continually improve your data assets to deliver value to customers. When organizations think about data products merely as deliverables, they fall into the same trap that has plagued data teams for decades: the project mindset.



In the project world, workflows progress through stages in sequence. Teams are assigned to specific projects, creating silos and handoffs that slow delivery and degrade quality. Success is measured by whether the project was delivered on time and within budget, not by whether users actually found it valuable. The product approach transforms this dynamic entirely. Instead of significant, risky releases, teams deliver minor, incremental improvements continuously. Work is brought to stable teams who own their products end-to-end. Most importantly, the approach recognizes that data products are never "done." This continuous improvement mindset yields measurable benefits: faster releases, increased team learning, better insights, and ultimately faster delivery of value from data investments.

Sin Five: Lack of Process Curation

There is a quiet crisis happening in data organizations everywhere. It occurs when brilliant analysts independently solve the same problem in three different ways, when data scientists struggle to reproduce each other's work, and when a single "simple" metric like Monthly Recurring Revenue has five competing definitions scattered across the organization. As detailed in Process Guardianship: The Most Valuable Data Engineering Work You're Probably Not Doing, the absence of deliberate process curation means that business logic lives everywhere and nowhere. Metrics, models, calculations, and code are scattered across SQL scripts, Jupyter notebooks, Looker reports, Python models, and Excel macros, with no one taking responsibility for consolidating them into a unified, tested, production-grade system.



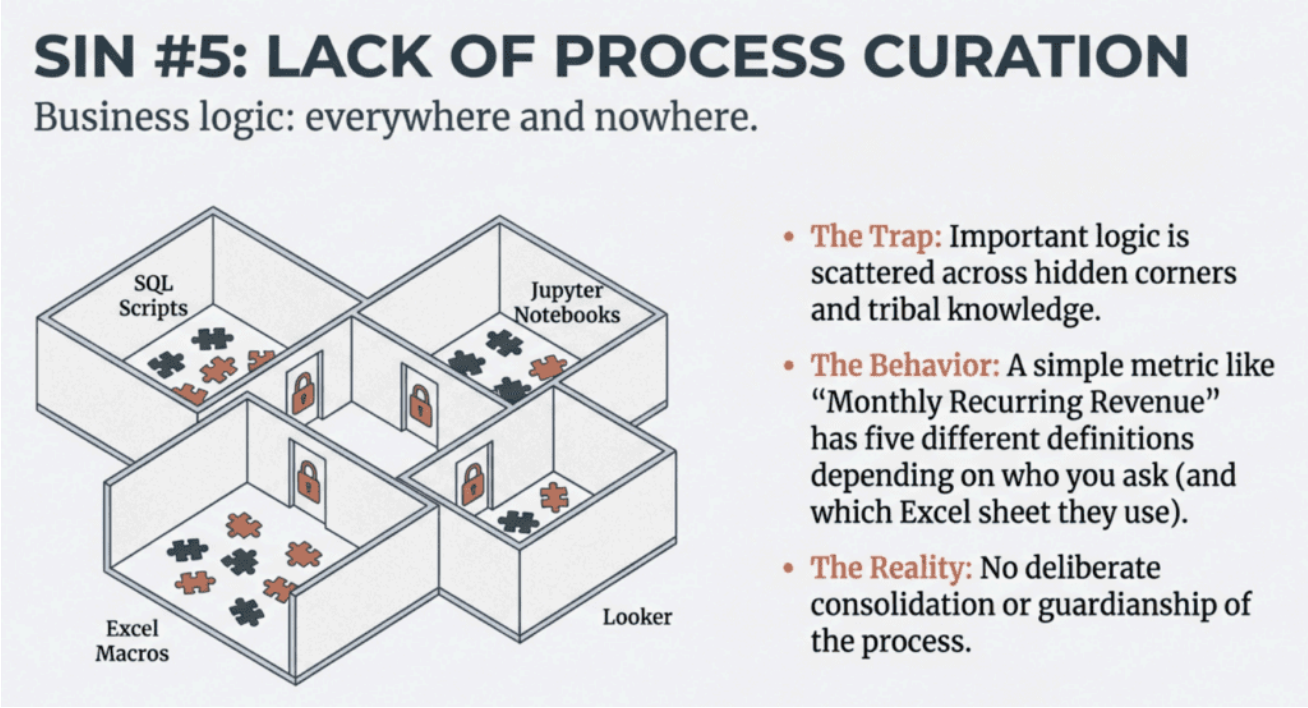
The costs compound quickly. Duplicate work becomes the norm as teams waste countless hours reinventing the wheel because they do not know what already exists or cannot trust it enough to reuse. Multiple overlapping metrics create confusion and erode stakeholder trust. When leadership asks how many customers are in the high-value segment, they get three different answers. Embarrassing data errors slip through because corrections identified by one team are not shared with the rest of the enterprise. Data engineers have a unique opportunity to address this crisis through what might be called process guardianship: actively consolidating business logic, extracting it from hidden corners and tribal knowledge, and encoding it in a centralized, version-controlled, production-ready system. This work may seem unglamorous, but it is often the most valuable contribution a data team can make.

Moving From Organizational Sin to Organizational Strength

These five organizational sins share a common thread: they all represent failures of systems and structures rather than failures of individuals. You cannot shame individual data engineers into fixing low relational coordination. You cannot train analysts out of blame culture while the incentive structures reward blame. You cannot wish your way from data-centric to process-centric thinking without fundamentally changing how work is organized and measured. This is precisely why DataOps exists. DataOps is not just DevOps for data. It is a comprehensive methodology that applies Agile development, DevOps, and lean manufacturing practices to data analytics, enabling organizations to accelerate the development of new analytics, deploy with confidence, and reduce data errors to virtually zero.

The DataOps Cookbook provides a practical roadmap for implementing these principles. Now in its third edition, the Cookbook explains how DataOps draws on lessons from the software and manufacturing industries to transform data chaos into a single, smooth, efficient process that delivers fast, error-free insight. It covers everything from understanding Data Journeys and implementing observability to building the automated testing and unified orchestration that process-centric teams require. The methodology

addresses each of the organizational sins we have discussed: it breaks down silos through shared visibility and collaboration frameworks, eliminates blame culture by making pipelines transparent and accountable, shifts teams from data-centric to process-centric thinking, enables product mindsets through continuous delivery practices, and provides the foundation for true process guardianship.



The journey from organizational dysfunction to data team excellence is not easy, but it is achievable. Thousands of data professionals have already formed study groups around the DataOps Cookbook and are beginning to transform their organizations. The key insight is that you do not need to fix everything at once. Start by focusing on understanding and observing the journey that data takes through your production environment. This monitoring process identifies data errors, tool problems, and timing issues, enabling a quick win for your DataOps implementation by driving immediate improvements. Lowering production errors increases the reliability of your data and gives your team more time to focus on automation. From there, you can systematically address each organizational sin, building the collaborative, process-centric, product-oriented data organization that actually delivers on the promise of data-driven decision making. Download the DataOps Cookbook and start your team's transformation today.

CULTURE AND ORGANIZATION

You're Thinking About Data Products All Wrong

Read the original article at: https://datakitchen.io/blog/youre-thinking-about-data-products-all-wrong/

Current Discussion of Data Products ...

(From ChatGPT)

A data product is a reusable, trusted, and well-defined data asset—such as a dataset, metric layer, feature set, or dashboard

.... that is designed, built, and managed like a product.

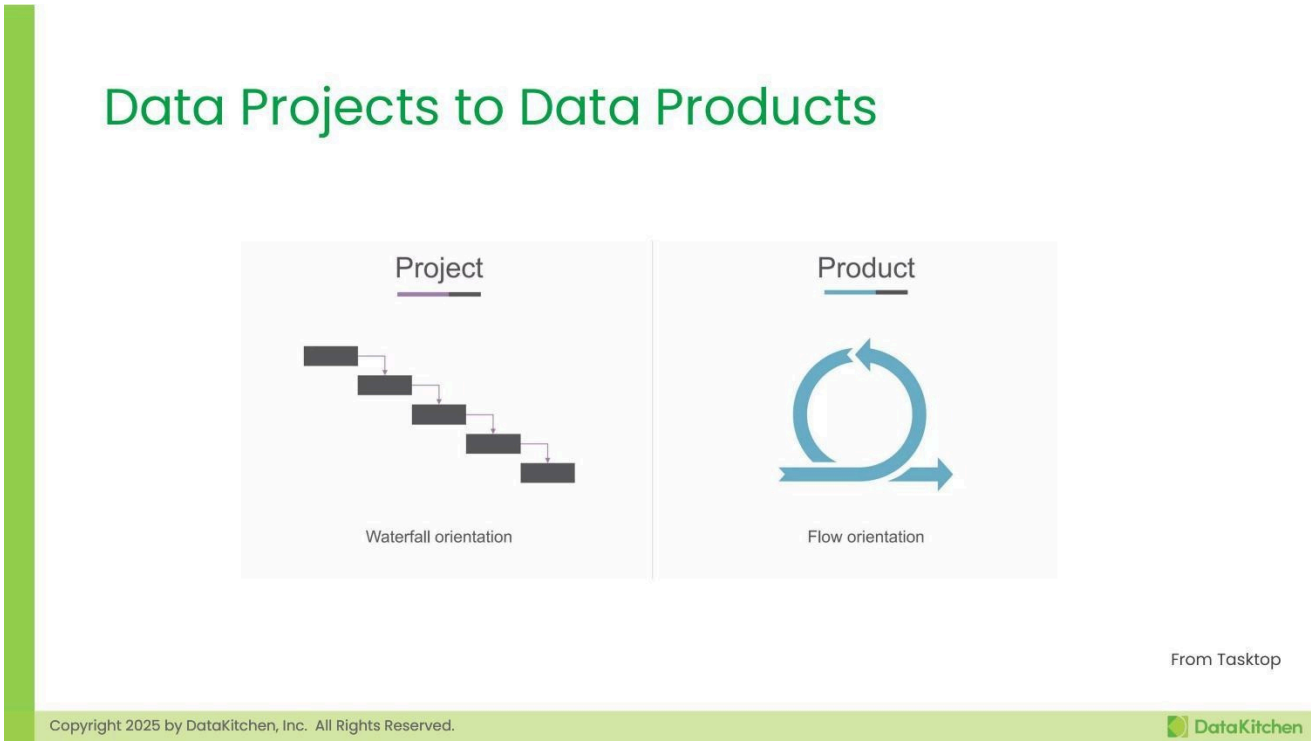
Copyright 2025 by DataKitchen, Inc. All Rights Reserved. DataKitchen

When data team leaders hear “data products,” they immediately think of the stuff they produce: dashboards, datasets, models, and warehouses, but focusing on the “*what*” completely misses the massive shift that data products really represent. The truth is, data products aren’t about what you create, but “*how*” you build, maintain, and continually improve your data assets to deliver value to customers.

The current discussion around data products, as reflected in countless articles and even AI-generated definitions, tends to describe them as “reusable, trusted, and well-defined data assets, such as a dataset, metric layer, feature set, or dashboard, that is designed, built, and managed like a product.” This definition gets us halfway there, but it still emphasizes the artifact rather than the approach.

When we think about data products merely as components or deliverables, we fall into the same trap that has plagued data teams for decades: the project mindset. This perspective leads us to view our work as a series of discrete efforts with defined endpoints, budgets to meet, and boxes to check. However, this approach fundamentally misaligns with how modern organizations need to utilize data for a competitive advantage.

The shift from projects to products represents a fundamental change in how we approach data work. Unlike projects, which have a clear beginning and end, products are living entities that evolve continuously in response to user feedback and changing business needs. This isn't just semantics, it's a complete reimagining of how data teams operate.



In the project world, we follow a waterfall orientation where work flows sequentially through stages. Teams are brought to specific projects, often creating silos and handoffs that slow down delivery and reduce quality. The focus becomes meeting deadlines and staying within budget, rather than delivering ongoing value to users.

> *The 'thing' that is in a data product could be any deliverable your team does. Defining the requisite components of data products is like counting angels on a pinhead.*

In contrast, the product approach creates a continuous flow of value. Work is brought to stable teams who own their products end-to-end. These teams measure success not by whether they delivered on time, but by whether their users are satisfied and receiving value. The product mindset embraces iteration, learning, and continuous improvement.

This transformation in data isn't happening in isolation. It mirrors a revolution that already transformed software engineering over the past two decades. Software teams have transitioned from DevOps to DataOps, from monolithic architectures to domain-driven design, and from project-based delivery to product-driven thinking. Now, these same principles are revolutionizing how we approach data and analytics.

From Software Engineering to Data Analytics

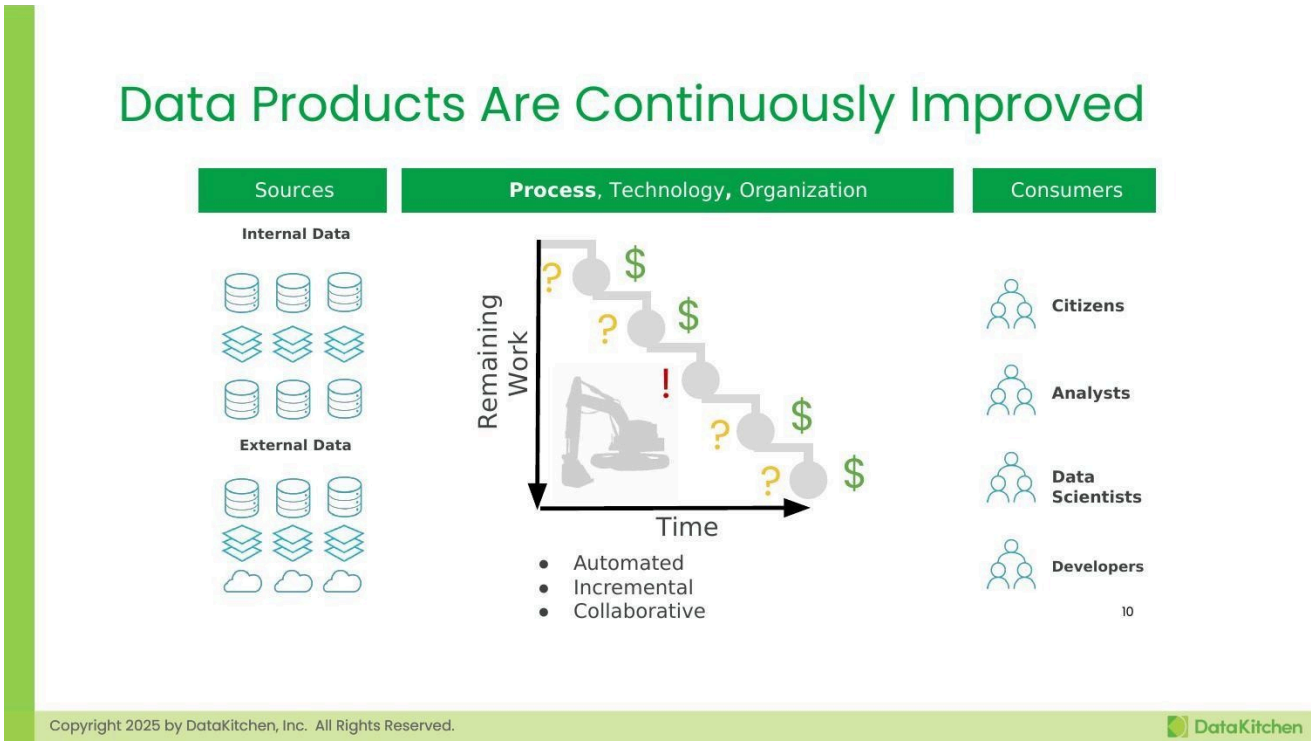
Software Engineering	Data & Analytics
• DevOps	-> DataOps
• Domain-Driven Design	-> Data Mesh
• Software Products	-> Data Products
• Test-driven Development	-> Data Testing
• Observability	-> Data Observability
• Event-Driven Architecture	-> Streaming; Kappa Architecture
• API	-> Data Contract
• Functional Programming	-> Functional Data Engineering

2025 by DataKitchen, Inc. All Rights Reserved.

DataKitchen

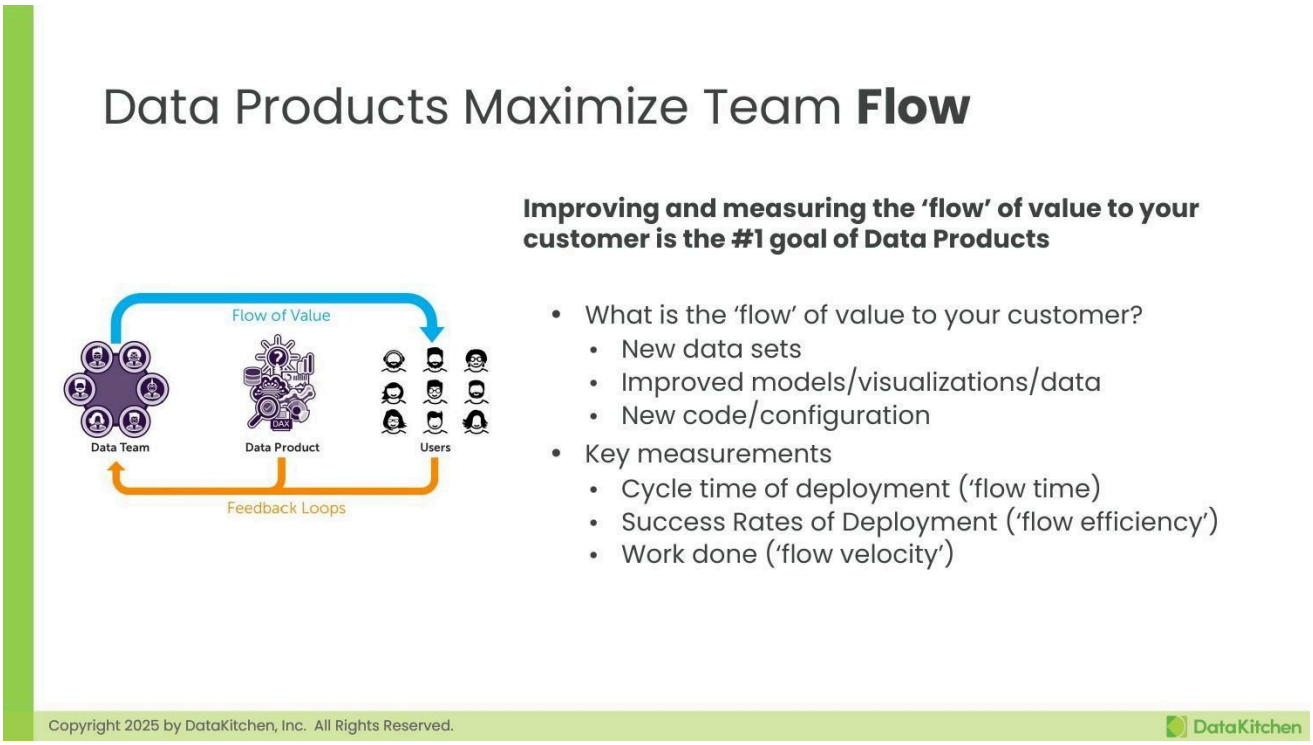
Consider how software teams transformed their practices: Test-driven development evolved into data-driven testing. Observability became data observability. Event-driven architectures evolved into streaming and Kappa architectures. APIs transformed into data contracts. Each of these transitions represents not just a technology change, but a fundamental shift in thinking about how we deliver value.

One of the most crucial aspects of the product mindset is understanding that data products are never “done.” This might feel uncomfortable for teams accustomed to celebrating project completions, but it’s actually liberating. Instead of rushing to meet an arbitrary deadline with a potentially suboptimal solution, teams can focus on iterative improvement, learning from each release to make the next one better.



This continuous improvement approach yields measurable benefits: faster releases of analytics products, increased team learning, better insights, and ultimately, faster delivery of value from your data investments. Teams operating in this mode don't just deliver once and move on, they establish feedback loops with their users, monitor usage patterns, and continuously refine their deliverables. With the traditional project approach, you may invest heavily up front with uncertain returns. With the product approach, you can start small, prove value quickly, and scale based on actual user adoption and feedback. Each iteration builds on the last, creating compound returns on your data investments.

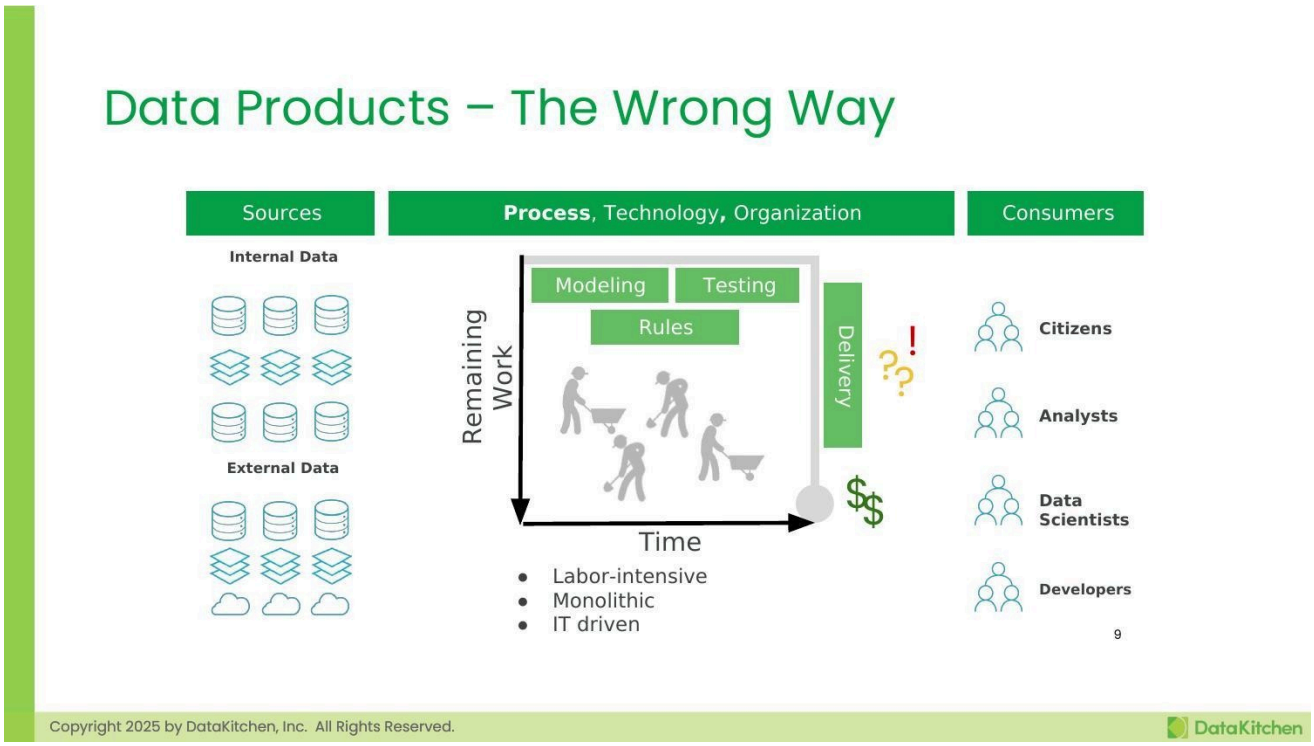
The ultimate goal of adopting a data product mindset is to maximize the flow of value to your customers. This isn't just about moving faster, it's about creating a sustainable and predictable delivery of high-quality data deliverables that users actually want and use.



Flow encompasses several key dimensions. Flow time measures how quickly you can deliver new capabilities from idea to production. Flow efficiency tracks your success rates, ensuring that the work you do actually reaches users rather than getting stuck in approval processes or failing in production. Flow velocity measures the volume of valuable work completed, helping teams understand and improve their capacity over time.

These metrics matter because they directly correlate with business value. Reducing flow time means faster response to market changes. Improving flow efficiency means less wasted effort and more predictable delivery. Increasing flow velocity means more value delivered to users over time.

Many organizations still approach data products incorrectly, treating them as large, monolithic efforts driven primarily by IT teams. This approach is characterized by heavy manual processes, long delivery cycles, and a disconnect between what's built and what users actually need.



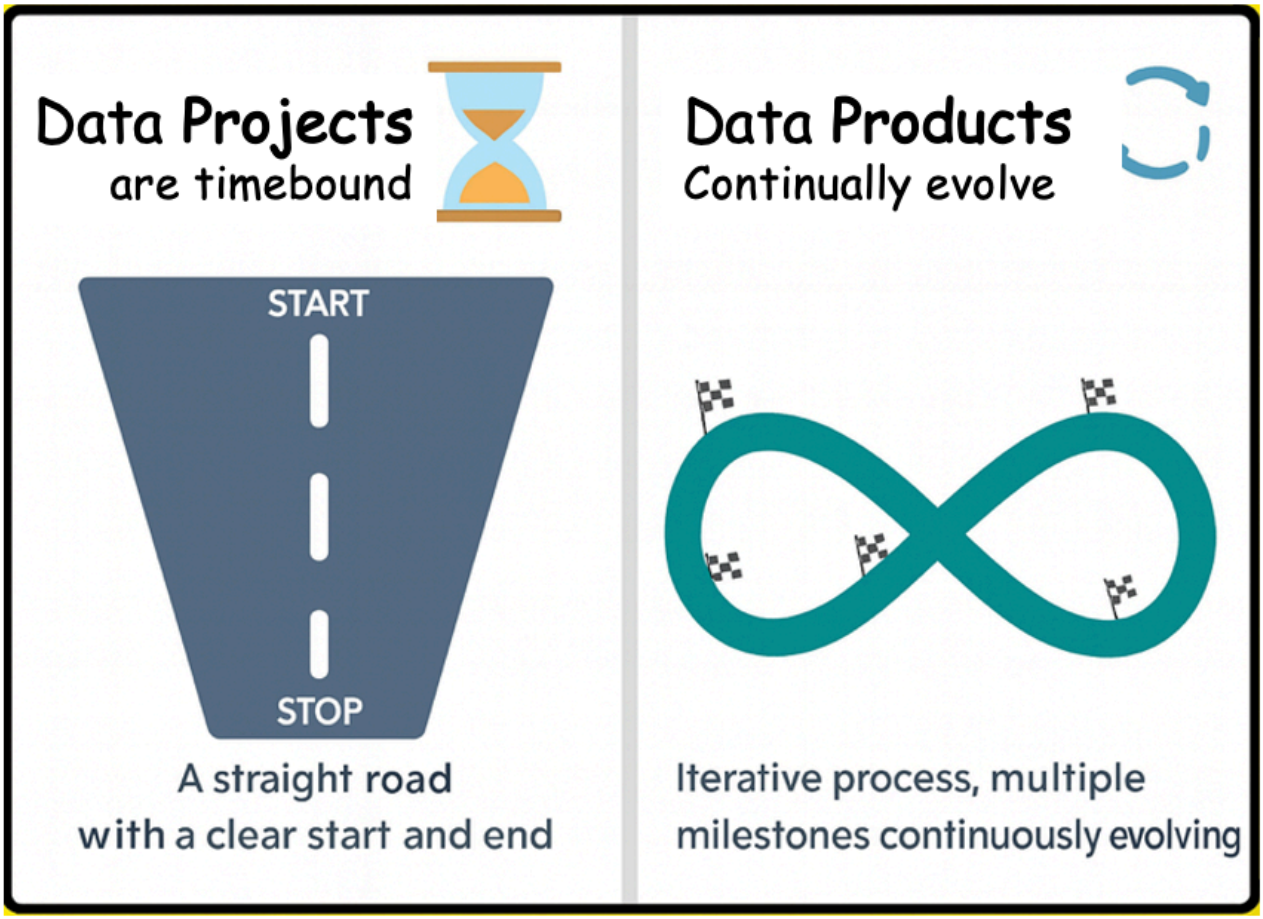
In this model, work flows slowly through various stages, modeling, rules definition, testing, with significant manual effort required at each step. The result is uncertainty about the timeline, cost, and whether the final product will actually meet user needs. Teams operating this way often find themselves in a reactive mode, constantly fighting fires rather than proactively improving their products. And many times, they miss the mark on what their users actually need.

> *"Our project was a success; we followed our SLDC and got an award." Yeah, but did the customer use it? Did they find it valuable?*

The product approach completely transforms this dynamic. Instead of significant, risky releases, teams deliver minor, incremental improvements continuously. Automation replaces manual processes wherever possible, freeing teams to focus on innovation rather than maintenance. Most importantly, the approach is inherently collaborative, bringing together data engineers, analysts, scientists, and business users to pursue shared outcomes.

This collaborative approach extends beyond just the data team. Product thinking requires close partnership with users, regular feedback sessions, and a willingness to pivot based on what you learn. It means treating your internal data consumers as valued customers whose satisfaction determines your success.

Transitioning from a project to a product mindset isn't just about adopting new tools or processes, it's about fundamentally changing how your organization thinks about data and information. This shift requires leadership commitment, cultural change, and sometimes, investment in new capabilities.



Start by identifying a small, high-value use case where you can demonstrate the product approach. Choose something with engaged users who can provide regular feedback. Build a cross-functional team that can own the deliverable from end to end. Establish clear metrics for success based on user outcomes rather than delivery milestones.

As you begin delivering value through this approach, use your success to build momentum. Share your learnings broadly and celebrate not just releases, but also improvements in your flow metrics. Help other teams understand how the product mindset can enable them to deliver more value with less stress and uncertainty.

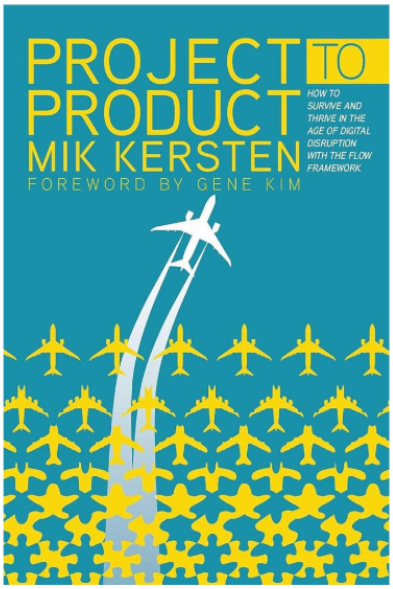
While the product mindset is primarily about approach rather than technology, having the proper technical foundation certainly helps. This includes capabilities for continuous integration and deployment, automated testing, monitoring and observability, as well as robust data governance. Embracing a DataOps approach, which applies agile and DevOps principles to data management, can significantly enhance this foundation. DataOps fosters collaboration between data engineers, data scientists, and other stakeholders, streamlining the entire data lifecycle from data ingestion to consumption. The goal is to create an environment where teams can safely and quickly iterate on their products without fear of breaking critical systems.

In the product world, success metrics shift dramatically from project metrics. Instead of asking “Did we deliver on time and on budget?” we ask “Are our users successful? Are they recommending our team to others?” This shift in metrics drives a shift in behavior, focusing teams on long-term value rather than short-term delivery.

Establish metrics that matter. This might include data quality scores or business-specific metrics, such as “time to insight” or “decisions influenced.” Make these metrics visible to your team and stakeholders, creating transparency around the value you deliver.

Data Products Are A ‘How’ Not A ‘What’

- **Data Products are a method, not a component list**
  - They do not have an end (like a project)
  - They are iteratively improving
  - Are focussed on delivering value, not meeting budget
  - Outcome oriented, not data oriented
  - Can be any type of deliverable
  - Success is defined by customer satisfaction, not on time delivery or internal process followed.



The shift from thinking about data products as a “what” to understanding them as a “how” is more than just a philosophical exercise, it’s a practical transformation that can dramatically improve your team’s effectiveness and job satisfaction. By embracing the product mindset, you’re not just changing how you deliver data assets; you’re changing how your organization creates value from its data. The question isn’t whether to make this shift, but how quickly you can begin the journey. Give us a call!

CULTURE AND ORGANIZATION

Process Guardianship: The Most Valuable Work You Are Probably Not Doing

Read the original article at: https://datakitchen.io/blog/process-guardianship-the-most-valuable-data-engineering-work-youre-probably-not-doing/

The Hidden Crisis in Data Teams: When Business Logic Lives Everywhere and Nowhere

There's a quiet crisis happening in data organizations everywhere. Not the dramatic kind that makes headlines, no security breaches, no system failures. Instead, it's a slow erosion of value that occurs when brilliant analysts independently solve the same problem in three different ways, when data scientists struggle to reproduce each other's work, and when a single "simple" metric, such as Monthly Recurring Revenue, has five competing definitions scattered across the organization.

I've watched data teams double their headcount while their actual impact plateaus. The problem isn't talent or technology, it's the absence of what I call process guardianship: the deliberate, systematic work of consolidating how business logic lives and operates within an organization.

For data engineers reading this, I want to make a case that might initially feel uncomfortable: your job isn't finished when you deliver clean, analyst-ready data. In fact, that's where some of your most valuable work begins.

What Process Guardianship Actually Means

Process guardianship is the practice of curating and consolidating disparate metrics, models, calculations, and code into a unified, tested, and highly agile data pipeline. It means taking the business logic that naturally scatters itself across SQL scripts, Jupyter notebooks, Looker reports, Python models, and Excel macros, and deliberately centralizing it into a maintainable, version-controlled, production-grade system.

This isn't just about code hygiene or engineering best practices. It's about recognizing that the real intellectual property of your data team, the accumulated knowledge of how your business actually works, shouldn't live as tribal knowledge on the laptops of team members around the world.

It takes a village to create insight for your customer. How do you stop duplicate or redundant work?

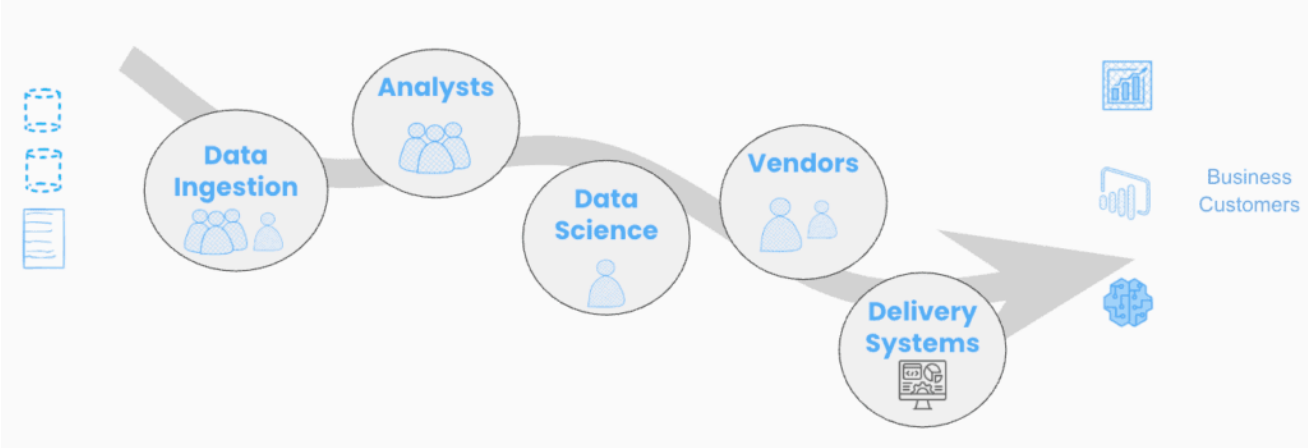
It takes a village to create insight for your customer. How do you stop duplicate or redundant work?

Think about the business logic in your organization right now. Where does the calculation for customer lifetime value actually live? Is it in the Excel macro on Bob's laptop on the finance team? Or in the segmentation model that the data science team built? Or in the SQL query your senior analyst wrote last quarter? The answer, more often than uncomfortable, is "all three places, and they don't quite match."

The Hidden Cost of Scattered Business Logic

Your marketing analyst builds a brilliant customer segmentation model. It works beautifully for their campaign targeting. Three months later, your data scientist builds another segmentation model for churn prediction, unaware that the first one exists. Six months after that, finance needs customer segments for their forecasting model and builds yet another version because they can't find or don't trust the existing two. Now you have three models that: use slightly different definitions of "active customer," handle edge cases differently, produce overlapping but inconsistent results, and require separate maintenance when business rules change

When leadership asks, "How many customers do we have in our high-value segment?", you get three different answers. This isn't a hypothetical scenario, this is Tuesday afternoon in most data organizations.



Duplicate work becomes the norm. Teams waste countless hours reinventing wheels because they don't know what already exists or can't trust it enough to reuse it.

Multiple overlapping metrics create confusion and erode trust. When "monthly active users" means something different in the product dashboard versus the board deck, stakeholders stop believing in your numbers.

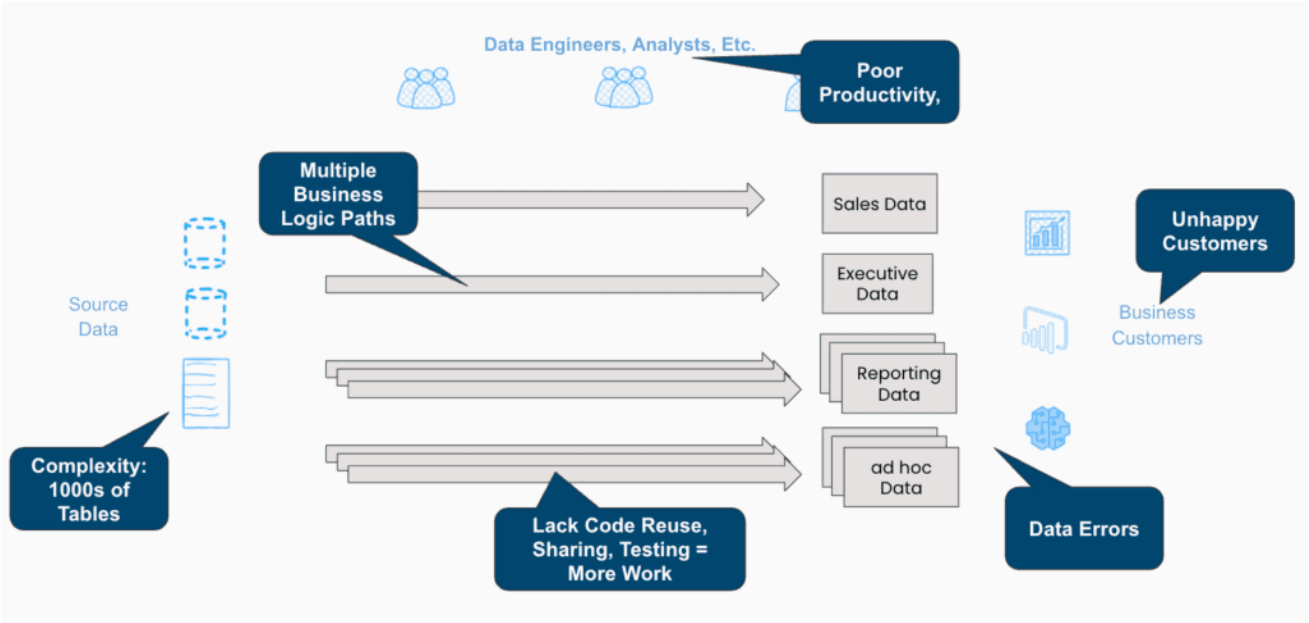
Embarrassing data errors slip through. The marketing team launches a campaign targeting "high-value customers" based on one definition. At the same time, finance forecasts revenue using another method, and nobody notices until the numbers don't add up at the end of the quarter. Or legitimate corrections identified by one team aren't vetted or shared with the enterprise.

Lower data team productivity becomes the steady state. Your talented people spend more time reconciling discrepancies, explaining why numbers don't match, and rebuilding things that should already exist than they do creating new value. Newer employees or vendors often struggle to get up to speed quickly.

The Data Engineer as a Process Guardian

Data engineers have a unique position in the extended data team. You sit at the intersection of raw data and business consumption. You understand both the technical infrastructure and the analytical workflows. You see the patterns, you're the ones who notice when three different teams ask for "basically the same thing" in three different ways. Data engineers are the first to identify duplicate code and are ideally positioned to make recommendations.

Process guardianship means actively stepping into that space and saying: "This business logic is too important to live in scattered notebooks and ad-hoc queries. Let's consolidate it, test it, and make it production-grade."



Extracting business logic from its hiding places. That incredible revenue forecasting model your data scientist built in a notebook? That complex customer scoring logic buried in a Looker view? That critical date logic that exists only in your senior analyst's SQL queries? These need to be identified, documented, and brought into a centralized system.

Creating a unified, tested pipeline. Business logic should reside in version-controlled code, accompanied by thorough testing, comprehensive documentation, and rigorous review processes. When your churn model definition changes, it should change in exactly one place, with a clear history of what changed and why. With source code version control.

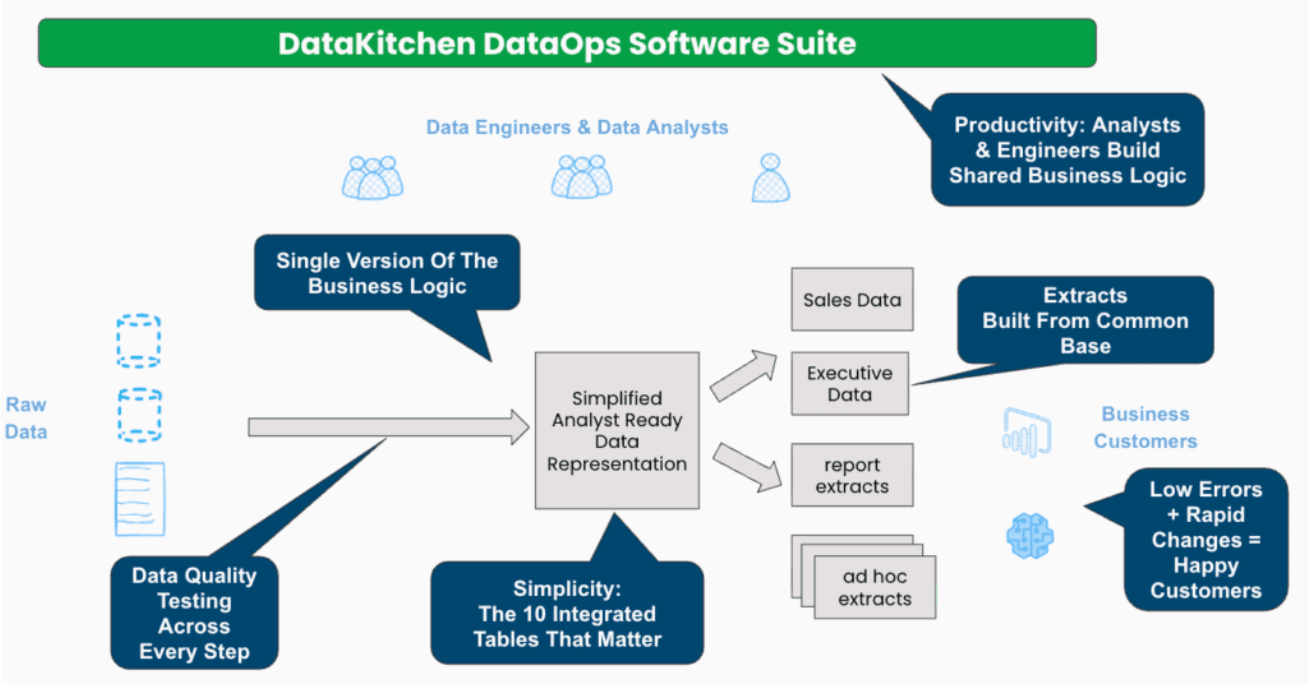
Building for agility and experimentation. Done right, process guardianship creates a foundation that enables experimentation to be faster and safer. When analysts and data scientists can trust the foundational business logic and metrics, they can focus on asking new questions rather than rebuilding the basics.

Enabling smooth transitions from experiment to production. When that experimental model or analysis proves valuable, moving it into production shouldn't require a complete rewrite. The path from notebook to production pipeline should be clear and well-traveled.

The Reality: Less Work Than You Think

Here's something that might surprise you: from a pure engineering hours perspective, this work is often a small fraction of your total effort. You're not rebuilding everything from scratch, you're consolidating what already exists and creating structure around it.

The actual technical work might look like: converting that critical SQL logic from a report into a dbt model with tests, refactoring a data science model in Python from a notebook into a versioned, parameterized pipeline, creating shared metric definitions in your semantic layer that replace dozens of slightly different calculations, building lightweight frameworks that make it easy for analysts to contribute tested, reusable logic, or building a new fact in a shared fact table.



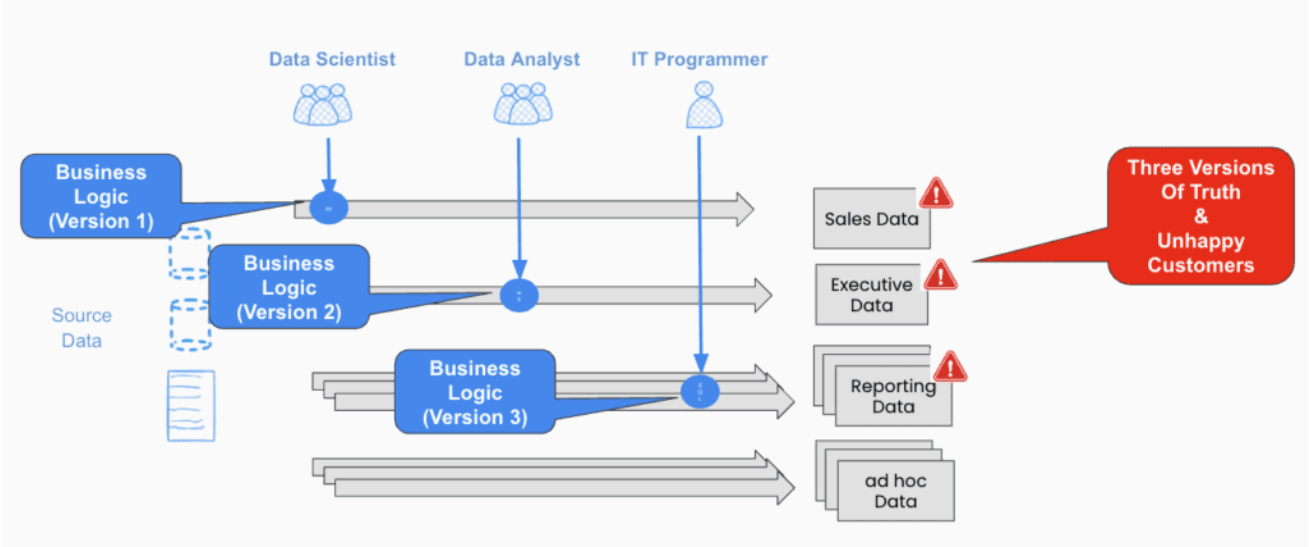
This isn't months of architectural redesign. It's often days or weeks of focused consolidation work. But the impact is wildly disproportionate to the time invested. Let me describe what this looks like in a well-functioning data organization.

When a data scientist wants to build a new churn prediction model, they don't start by defining what "churned" means, that definition already exists in the centralized data pipeline, tested and agreed upon by the business. They can focus their improvement on the actual working code with tests.

When finance needs to create a new forecast, they pull from the same customer segmentation, revenue recognition rules, and metric definitions that marketing and product are already using. The numbers match by default, not by accident.

When a critical business rule changes, such as how you handle returns in your revenue calculations, that change occurs in one place, propagates automatically, and every downstream consumer receives the update without manual intervention.

When an analyst builds something experimental that proves valuable, the path to productionizing it is straightforward because it was built on top of tested, production-grade foundations from the start.



When You Curate A Single Version Of The Business Logic, You Get A Single Version Of The Truth In Your Data

When You Curate A Single Version Of The Business Logic, You Get A Single Version Of The Truth In Your Data

The data team's collective knowledge isn't locked in individual heads or scattered across folders. It's codified, tested, documented, and accessible. New team members can onboard more quickly because they're learning from a centralized, maintained body of work rather than searching through old tickets and Slack messages.

Why This Matters

Stakeholder trust increases when numbers are consistent across different teams and tools. When executives see the same metrics in every dashboard, calculated the same way, they stop questioning the numbers and start acting on them.

Decision velocity improves because people are no longer waiting for analysts to rebuild basic metrics or reconcile conflicting definitions. The foundational layer is solid, allowing analysis to occur more quickly.

Experimentation becomes safer because there's a clear distinction between production logic (tested, reviewed, maintained) and experimental work (rapid, flexible, creative). You can move fast without breaking things.

Knowledge persists when key team members leave. The business logic they understood isn't lost, it's captured in the centralized system, so there's no more tribal knowledge.

Compliance and governance become manageable. When you need to audit how a metric is calculated or ensure specific data handling rules are followed, having logic centralized makes this dramatically simpler.

Streamlined compliance and governance. Auditing metric calculations or enforcing data handling rules is significantly simplified with centralized logic.

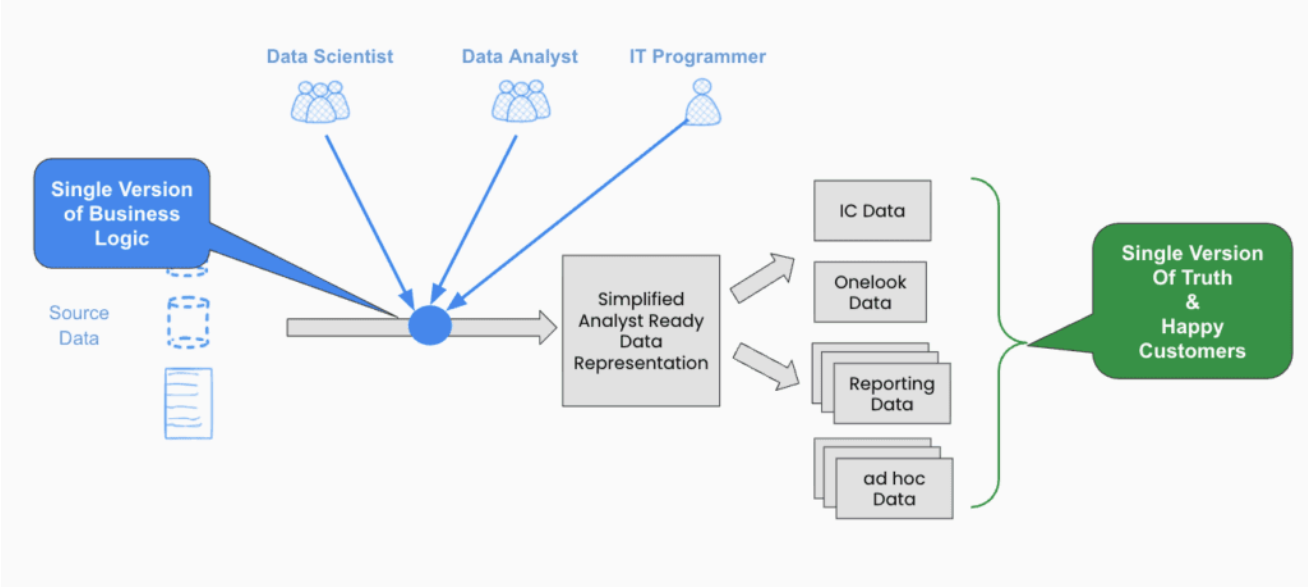
“We’re too busy building new pipelines to worry about consolidating existing logic.” This is the trap. You’ll always be too busy if you never invest in consolidation. The duplicate work and scattered logic are what’s making you busy. Process guardianship is how you break the cycle.

“Our analysts need flexibility, centralized logic will slow them down.” Good process guardianship increases flexibility. When analysts can trust foundational metrics and logic, they spend less time on basic plumbing and more time on actual analysis. The goal is to liberate analysts from production drudgery.

“This sounds like a lot of bureaucratic overhead.” It doesn’t have to be. This isn’t about creating approval workflows or committees. It’s about making it easier to do the right thing than the wrong thing. When contributing to centralized, tested logic is simpler than copying someone else’s query and modifying it, people naturally gravitate toward it. It is about refactoring.

“Our business changes too fast for this to work.” Actually, the opposite is true. When business logic is scattered, every change requires hunting down every instance and updating every one. When it’s centralized, changes happen in one place and propagate automatically. Process curation makes you more agile, not less.

Ultimately, process guardianship is as much about mindset as it is about technical practice. It requires seeing your role as a data engineer not just as building pipelines, but as a steward of how your organization’s business logic lives and evolves.



It means asking questions like: Does this calculation exist anywhere else? Should this logic live in a production pipeline instead of a notebook? How will we maintain this when business rules change? What happens when the person who wrote this leaves the company?

It means having the conviction to say “before we build this, let’s consolidate what already exists,” even when there’s pressure to ship something new. It means advocating for the unsexy work of consolidation and refactoring, knowing that it’s often more valuable than the flashy new project.

In Complex Organizations, This Becomes Invaluable

In small organizations with simple data needs, you can sometimes get away without deliberate process guardianship. Everyone knows everyone, communication is easy, and the surface area is small enough to keep in your head.

As organizations grow in size and complexity, adding more data sources, teams, stakeholders, and systems, the chaos caused by scattered business logic becomes exponentially more costly. The importance of having someone actively guard these processes shifts from “nice to have” to “absolutely critical.”

Data engineers who practice process guardianship become force multipliers. The time investment is relatively small compared to other data engineering tasks. But the impact, on data quality, team productivity, stakeholder trust, and organizational agility, is enormous.

Typical “Scattered Processes”	With Process Guardianship
Customer Visible Data Errors	**Infrequent Data Errors**
Scattered, Independent Business Logic	**Centralized ‘Recipes’ Of Business Logic, Git Based**
Multiple Versions Of The Truth	**Unified Version of The Data Team’s Collective Intelligence**
Slow Development Of New Feature	**Fast Development Of New Features**
Poor Data Analytic Team Productivity	**High Team Productivity, Less Waste**
Slow Turn Around Time Of New Features/Data	**Fast Iterations, Low Risk Of Breaking Existing Features**

Be the Guardian Of Your Team’s Collective Intelligence

When people think of data engineers, the description usually stops at “building high-quality pipelines that deliver analyst-ready data.” That is true, but incomplete. In modern organizations, data engineers hold a deeper responsibility. They are not just the builders of pipelines, they are the curators of the business logic itself.

This guardianship role involves consolidating fragmented definitions of the business, metrics scattered across dashboards, SQL scripts, notebooks, and models, into a unified, tested, performant, and agile data production line. It involves extracting business logic from hidden corners and tribal knowledge, then encoding it in a centralized, version-controlled, and production-ready system.

The payoff? More consistent decisions, fewer errors, and higher productivity across the extended data team.

Process guardianship means making it a core part of how your data team operates. It means recognizing that your job isn’t done when the data is clean and accessible, it’s done when the business logic that acts on that data is consolidated, tested, and maintainable. It means seeing yourself not just as a builder of data infrastructure, but as a guardian of your organization’s collective intelligence about how to make sense of that data.

For data engineers, this is an opportunity to expand your impact beyond technical infrastructure into the strategic heart of how your organization operates. The IP of your data team, the accumulated knowledge of what metrics matter and how to calculate them, what models work, and how to apply them, is too valuable to scatter. It needs a guardian.

CULTURE AND ORGANIZATION

How Data Quality Leaders Can Gain Influence and Avoid the Tragedy of the Commons

Read the original article at: https://datakitchen.io/blog/how-data-quality-leaders-can-gain-influence-and-avoid-the-tragedy-of-the-commons/

In 1833, the economist William Forster Lloyd described a simple problem: a shared pasture open to all villagers will be overgrazed and destroyed, even if every individual villager is rational and well-intentioned. Each person benefits from adding one more animal to the commons. Each person bears only a fraction of the cost of overgrazing. The math is brutal. The commons collapses.

Data quality is a tragedy of the commons problem. Every team benefits from high-quality, reliable data. Nobody wants to own the cost of maintaining it. When something breaks, everyone has a reason why it is someone else's job to fix. The data decays. The trust erodes. And by the time anyone declares a crisis, the damage has already spread across five dashboards, two quarterly reports, and a product decision that cost real money.

This chapter is about how data quality leaders can break that cycle. Not through motivation or cultural messaging, but through structural changes that reassign costs to the people who create them.

A Schema Change Nobody Announced

The following scenario is anonymized but represents a pattern that plays out in some form at hundreds of companies every year.

1. Don't Criticize, Condemn or Complain
   Use Generative Data Quality To Automatically Identify Multiple Data Issues
8. Talk In Terms Of The Other Person's Interests
   Build Data Quality Dashboards on Limited, Specific Data Items
12. If You Are Wrong Admit It Quickly And Emphatically
   Start Quickly, Small, and Work Iteratively: DataOps Data Quality
18. Be Sympathetic With The Other Person's Ideas And Desires
   Give Them Specific Actionable Problems – They Are Busy!
19. Appeal To The Nobler Motives
   Make Multiple Data Quality Dashboards That Align To Organization Goals
29. Use Encouragement. Make The Fault Seem Easy To Correct
   Package Data Quality Issues So They Are Easy to Fix
30. Make The Other Person Happy About Doing The Thing You Suggest
   Measure And Show Data Quality Improvement Over Time

A mid-sized e-commerce company had a central `orders` table owned, notionally, by the data engineering team. The product engineering team maintained the application database. Analytics teams at three different business units all built dashboards and models downstream of that table.

In October, the product team shipped a new checkout flow. As part of that release, a developer renamed the column `shipping_method` to `fulfillment_method` to align with terminology used in the new UI. The change was noted in the application team's internal sprint notes. It was not communicated to the data engineering team. There was no data contract. There was no schema change notification. There was no process requiring either.

The ETL job that pulled from the application database continued to run. It silently dropped the `fulfillment_method` column because it was not in the expected schema, and continued populating `shipping_method` with null values. No alert fired. The pipeline returned a successful exit code.

For three weeks, every downstream dashboard that referenced shipping method showed nulls or incorrect aggregations. The supply chain team's "Fulfillment Method Mix" chart showed 100% of orders as "unknown." The analytics lead assumed it was a data entry issue with the new checkout UI and flagged it to the product team. The product team looked at the application database, saw clean data, and closed the ticket as "not reproducible."

During the October quarterly business review, the VP of Operations presented fulfillment cost-per-shipment metrics derived from the broken data. The numbers showed a 12% cost increase that did not actually exist. The operations team spent two weeks investigating carrier contracts before a data

engineer, while debugging an unrelated issue, noticed the null column and traced it back to the rename.

By the time the root cause was identified, the QBR deck had already been distributed. The investigation had consumed roughly 60 engineering and analyst hours. A carrier negotiation had been opened based on false data.

What Made This Possible

The immediate cause was a renamed column. The structural cause was a set of missing accountability mechanisms.

There was no formal owner for the `orders` table. The data engineering team felt responsible for the pipeline. The application team felt responsible for the source database. The analytics teams felt responsible for their dashboards. Nobody felt responsible for the contract between the source and its consumers.

There was no SLA for the `orders` table. Nobody had defined what "the orders table is working correctly" actually meant, so nobody could detect when it was not.

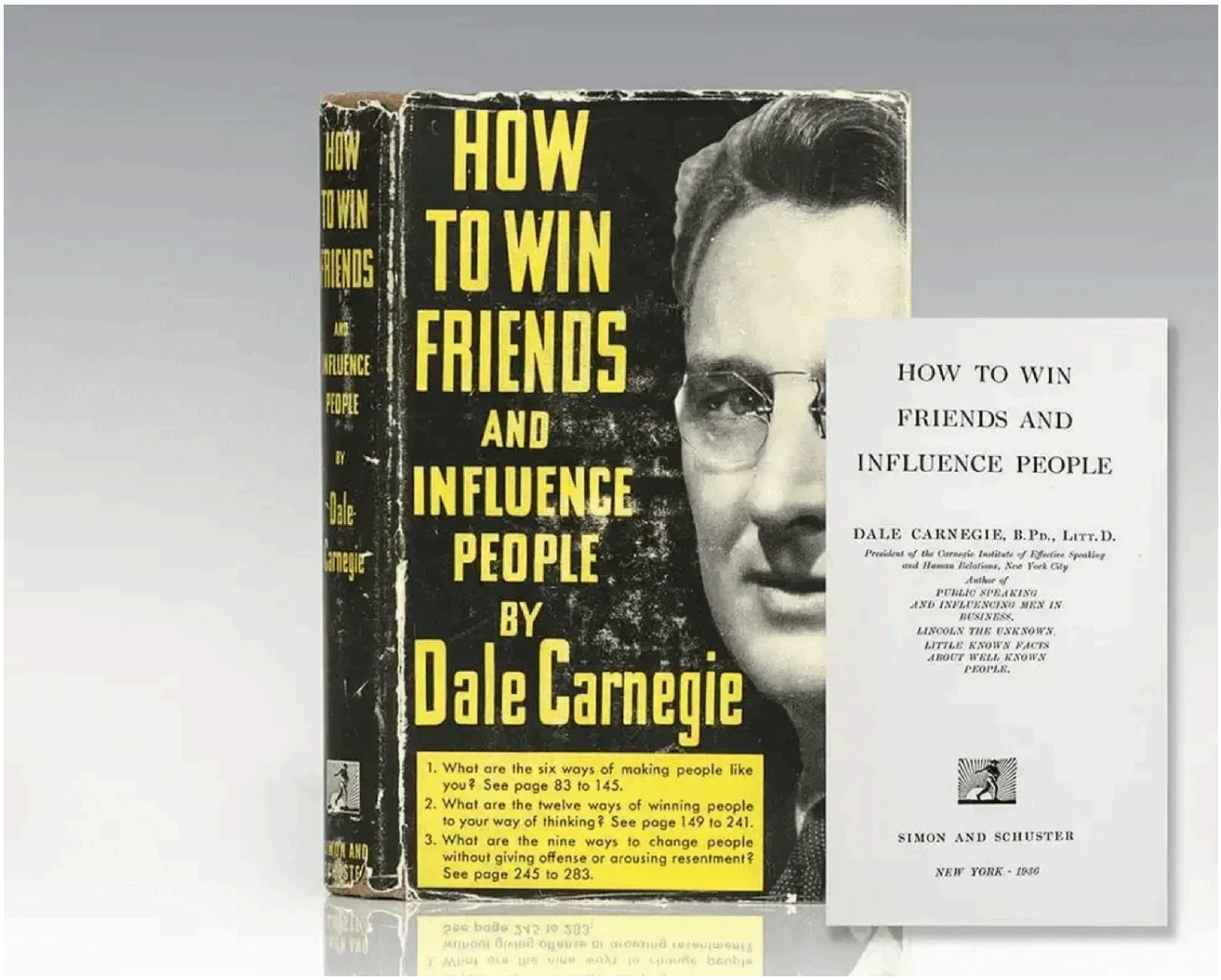
There was no process requiring upstream teams to notify downstream teams of schema changes. The application developer who renamed the column was not being careless. It was a routine rename, and no mechanism existed that would have made the downstream impact visible.

There was no validation at the ingestion layer. The ETL job accepted nulls without complaint.

Every individual acted rationally within their own scope. The commons was destroyed anyway.

The Structural Fix: Closing the Accountability Gap

Culture change does not solve tragedy of the commons problems. What solves them is restructuring the incentives and the costs. In the classic commons scenario, the solution is private property or regulated access: assign the cost of degradation to the person who creates it.



In data systems, the equivalent is a set of structural mechanisms that make accountability explicit and automatic.

Data Contracts Between Teams

A data contract is a formal agreement between the producer of a dataset and its consumers. It specifies the schema, the expected value ranges, the null behavior, the update frequency, and the SLA. It is owned by the producer and versioned like code.

At the company in the scenario above, a data contract for the `orders` table would have included a field definition for `shipping_method` with its expected values. When the application team planned the rename, any competent review process would have flagged the contract as needing an update. That flag would have triggered a notification to the data engineering team and all registered consumers before the change shipped.

The contract does not need to be a heavyweight artifact. It can live as a YAML file in the same repository as the application code. What matters is that it exists, that it is versioned, and that there is a process requiring it to be updated before schema changes ship.

A minimal contract for a single table includes:

- The table name and the owning team
- The schema with field-level descriptions and types
- Null policies for each field
- The refresh schedule and acceptable latency
- A list of registered downstream consumers
- A version number and a change log

When the contract changes, the registered consumers are notified. When the pipeline runs, the data is validated against the contract before it reaches downstream systems. If validation fails, the pipeline halts and alerts fire.

A Single Named Owner Per Dataset

Every dataset in production should have a single named owner. Not a team, a person. Teams do not respond to alerts at 2am. Teams do not get paged when a dashboard is wrong the morning of a board presentation. A named person does.

Named ownership does not mean that one person does all the work. It means one person is accountable for the health of that dataset, has the authority to enforce the contract with upstream producers, and is the escalation point when something breaks.

The ownership assignment should be documented, visible to all consumers of the dataset, and reviewed on a regular cadence. When people change roles or leave, ownership must transfer explicitly. Orphaned datasets are a commons waiting to collapse.

Named ownership also changes behavior upstream. When the application team knows that a specific person is accountable for the `orders` table and will be paged when it breaks, they are more likely to communicate before making schema changes. The social cost of breaking a named colleague's dataset is higher than the social cost of breaking an abstract shared resource.

A Published SLA for Each Table

An SLA for a dataset is a published, specific commitment: this table is refreshed every hour, data arrives within 15 minutes of the source event, and completeness is validated before the table is made available to consumers.

Without an SLA, there is no baseline from which to measure degradation. With an SLA, any deviation is a measurable, actionable event. The SLA also forces a conversation about priority. Not every table needs an hourly SLA. Agreeing that an internal staging table has a daily SLA and that a revenue-facing table has a 15-minute SLA is itself a forcing function for prioritizing monitoring and alerting work.

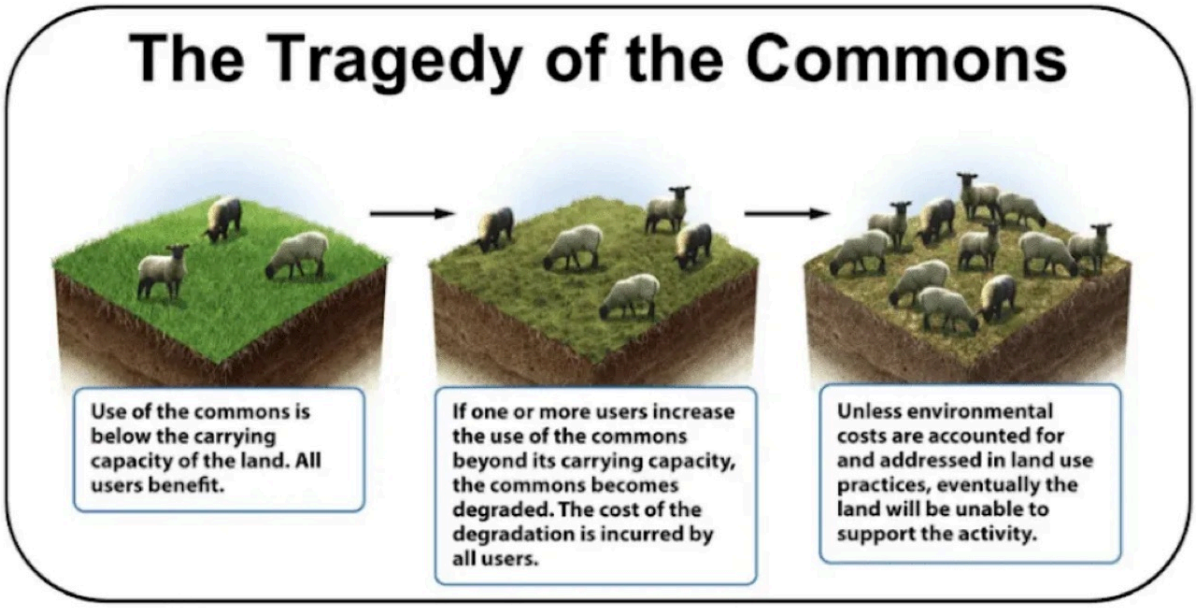
For the `orders` table scenario, an SLA would have included a freshness guarantee and a completeness check. The completeness check would have caught the null `shipping_method` column on the first pipeline run after the rename, before any dashboard consumed the broken data.

The practical form of an SLA per table:

Attribute	Definition
Refresh frequency	How often the table is updated
Maximum acceptable latency	Time allowed between source event and table availability
Completeness threshold	Minimum acceptable non-null rate per required column
Row count tolerance	Acceptable variance from historical daily row counts
Alert owner	Named person paged when the SLA is violated

Gaining Influence Without Authority

Most data quality leaders do not have authority over the upstream teams who create the problems they are trying to solve. The application team does not report to them. The product team does not report to them. And yet the data quality leader's success depends entirely on those teams honoring contracts and following change processes.



Dale Carnegie's central principle in "How to Win Friends and Influence People" is that people act in their own interest, and influence comes from aligning your ask with their interest. The data quality leader who approaches an engineering manager with "you need to follow our data contracts" will get polite agreement and no action. The one who approaches with "your team keeps getting pulled into data incidents that have nothing to do with your roadmap, and here is a lightweight process that would eliminate most of them" is speaking the engineer's language.

The practical version of this is to show the cost of the current state in terms the other team cares about. Engineering teams care about incident burden and on-call pain. Product teams care about developer velocity. Finance cares about analyst time waste. Find the metric that hurts the people you need to influence and attach it to the problem you are trying to solve.

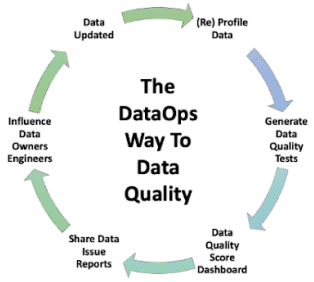
Document the incident history. How many engineering hours did the three schema-change incidents last quarter consume? Which teams were pulled in? How many of those hours came from teams that would have been protected by a contract? Put numbers on it. Then bring that evidence to the conversation, not as an accusation, but as shared data about a shared problem.

Carnegie also emphasized the importance of letting the other person feel that the idea is theirs. When working with an application engineering team to implement a data contract, involve them in designing the process. Ask what would be lightweight enough for them to actually use. Ask what notifications would be helpful versus annoying. An agreement that both parties shaped is far more durable than a mandate handed down from data engineering.

The Commons Does Not Fix Itself

The tragedy of the commons is not solved by asking people to be more responsible. It is solved by changing the structure so that the costs of degradation fall on the people who create them.

DataOps Process: Iterative Data Quality At Scale



- **(Re) Profile Data:**
  - Establish a baseline by measuring the current state of the data with data profiling
  - Get foundation for tracking progress and identifying areas for improvement, repeatedly
- **Generate Data Quality Tests:**
  - Generative Data Quality tests can streamline this process, making it easier to catch problems before they become more significant without coding
- **Data Quality Score Dashboard:**
  - Scoring provide a quantifiable way to assess the health of the data and communicate the results.
  - Scores should be focused on the data elements that matter and reflect top organization needs
- **Share Data Quality Issues Report**
  - Give actionable, specific changes to those who can do the technical work.
- **Influence Data Owners and Engineer**:
  - Once data quality issues have been identified, the next step is to drive the necessary changes within the organization.
  - Influence data owners, system administrators, or other stakeholders to address the identified issues.

For data quality, that means named owners who are accountable, data contracts that make schema changes visible before they propagate, SLAs that define what healthy looks like, and validation that catches failures before they reach consumers.

None of these require a new tool or a platform migration. They require discipline, a willingness to have structural conversations, and the influence skills to make agreements stick across organizational lines.

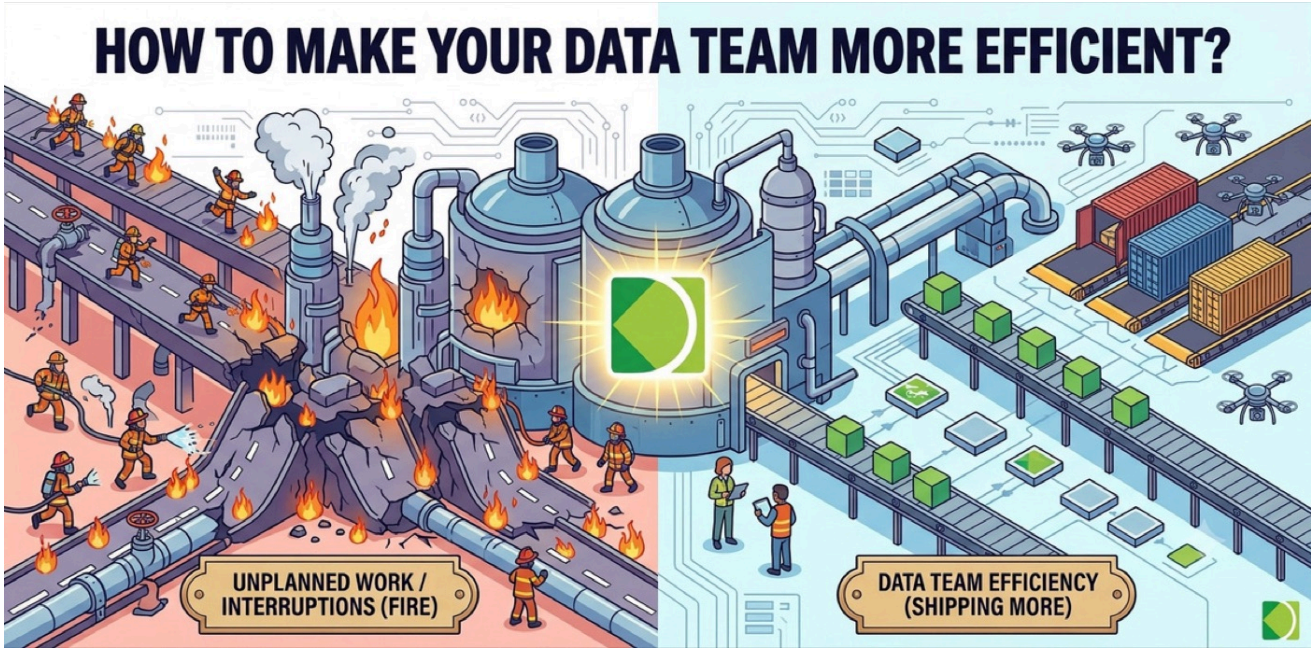
The data quality leader who masters this is not managing a quality program in isolation. They are redesigning the incentives of the system. That is harder than writing tests. It is also the only thing that actually produces durable results.

When every dataset has a named owner, a contract, and an SLA, the commons is no longer a commons. The costs are assigned. The incentives are aligned. And the quarterly review is based on numbers that are actually true.

CULTURE AND ORGANIZATION

How Do I Get My Data Team More Efficient?

Read the original article at: https://datakitchen.io/blog/how-do-i-get-my-data-team-more-efficient/



Your Monday starts with a Slack message. Somebody in finance says the numbers look wrong. Three people drop what they were doing. By four in the afternoon, you find it: an upstream system added a column and changed a type, and the load has been quietly dropping rows since Thursday. You fix it, you write the apology, you go home.

Nothing on that Monday was in the sprint. That's the whole problem.

When someone asks how to make a data team more efficient, they usually mean: how do we ship more? Typing faster won't do it. A large share of the week goes to unplanned work, and shipping more means reducing that number.

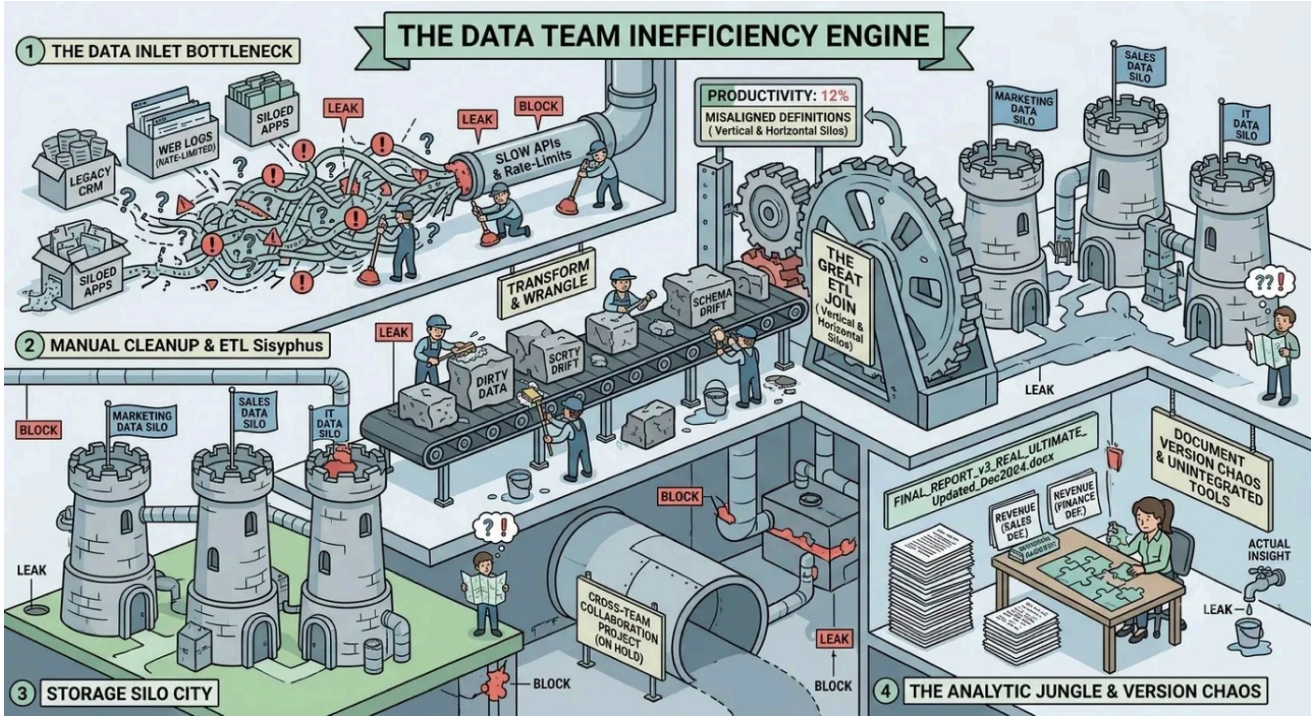
What the research actually says

You've seen the stats on bad data. The time ones hold up better than the money ones, so start there.

The most useful one comes from a 2022 survey of more than 300 data professionals, fielded by Wakefield Research and sponsored by Monte Carlo. Respondents said 40 percent of their time goes to evaluating or checking data quality. Three quarters of them need four hours or more just to detect an incident, and about half need another nine hours to resolve it once they've found it. Call it 13 hours per incident, between the break and the fix, before anyone corrects a single downstream report. More than half also said incidents were going up, not down.

That is the time cost. There is a human one underneath it, and the best evidence I know of is ours. In 2021 we surveyed 600 data engineers with data.world, also fielded by Wakefield. 97 percent reported burnout in their day-to-day work. The causes they named were not exotic: too much time finding and fixing errors, manual repetitive work on data prep and pipelines, and a pace of incoming requests nobody could stay ahead of. The same three things this post is about.

When the subject comes up, somebody usually reaches for governance. That survey is unkind on the point. 69 percent of those engineers said their own company's data governance policies made their day-to-day job harder. Governance gets sold as the fix and lands as friction.



The tactics below assume a small or mid-size data team, where you know everyone who is breaking things and you can change something without a review board. Inside a large enterprise the diagnosis holds and the tactics don't, because getting a check into production there is a procurement exercise, not an afternoon. Three routes follow, and most teams need more than one.

First, get your own number

The money numbers are softer than the time ones, and you should know why before you put them on a slide. Gartner's widely quoted $12.9 million average annual cost of poor data quality comes from its 2020 data quality Magic Quadrant, where 154 reference customers of data quality vendors were asked to estimate their own losses. Self-reported guesses from large enterprises who were already buying the software. Fine as an order of magnitude, useless as a benchmark for your team. People cite IBM's $3.1 trillion figure from 2016 even more often, and it has never carried a published methodology. MIT Sloan's estimate that companies lose 15 to 25 percent of revenue to bad data is in the same family.

Use those to establish that the problem is real and expensive. Don't take them to your CFO as evidence about your own team. Take your own number instead, and you can have it in a month.

Before you pick a route, measure what's actually happening. Not with a timesheet. Your engineers already resent the tracking they have, and asking them to log every interruption will get you a spreadsheet with four entries.

Use what you already collect. Tag every ticket that came from something breaking rather than something being planned, then read the ratio at the end of the sprint. Or count incidents per week and multiply by an honest estimate of what one costs end to end. Or spend an hour scrolling back through the channel where people report broken dashboards and reconstruct last month. Any of these gets you a number you can defend in a room, and none of them needs a new process.

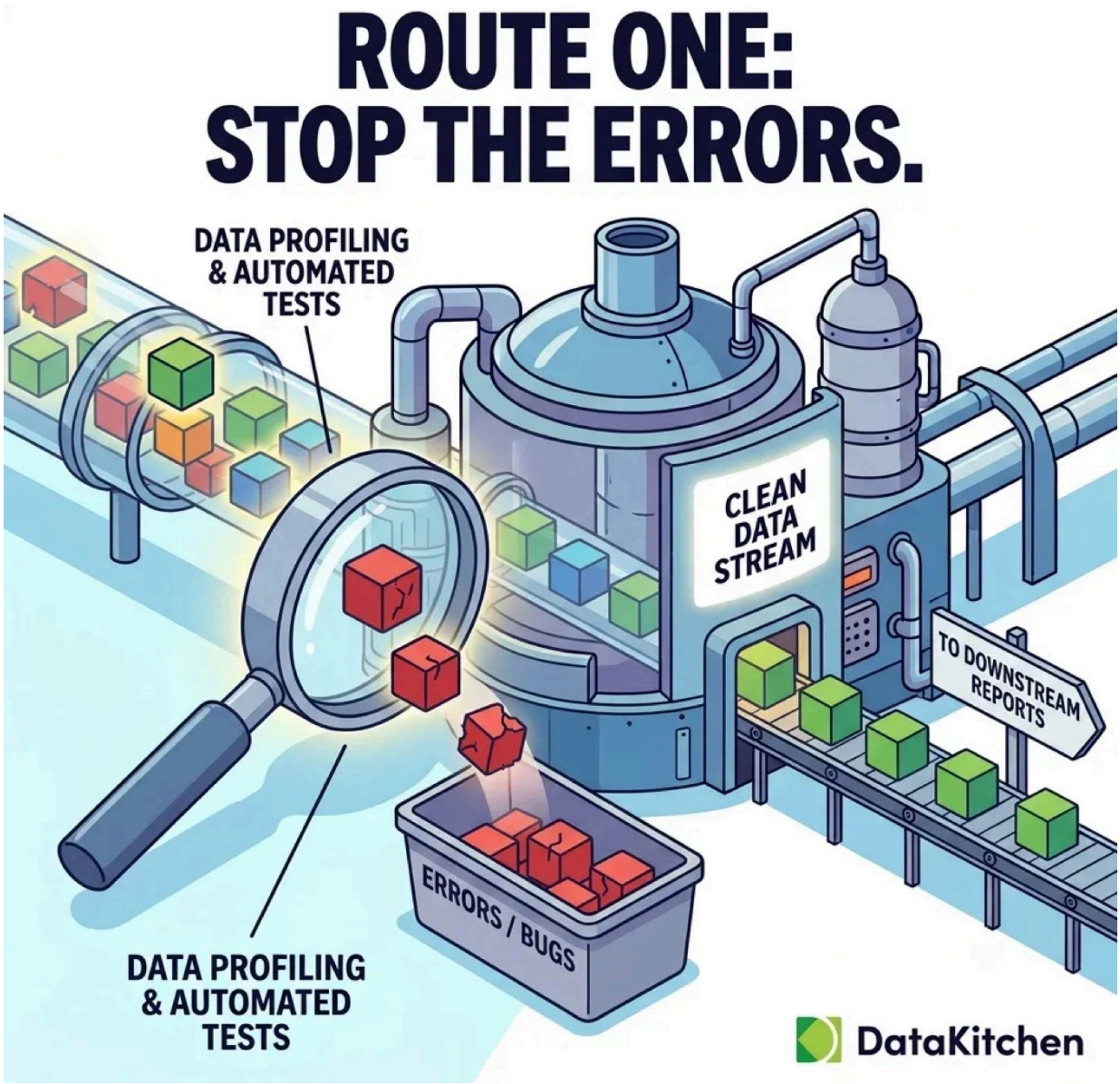
Then track detection lag: from the moment bad data lands to the moment a human says something. You probably know this one already. If the answer is "a day or two, unless it's the exec dashboard, then 20 minutes," you have a detection problem, not a data problem. You are eyeballing row counts and praying whether you call it that or not.

Those two numbers tell you which route to take, and they give you a before, which you'll want when someone asks in six months whether any of this worked.

Route one: stop the errors

This is the highest return per dollar, and almost nobody starts here, because it isn't exciting.

Production errors are a time problem before they are a quality problem. Every one of them buys you an investigation, a fix, a re-run, a set of downstream corrections, and a conversation with whoever built a report on the broken table. The fix takes 20 minutes. Everything around it takes a day.



The reason errors cost so much is that you find out about them from a person. Somebody who trusted your table enough to build on it is now your monitoring system. An expensive alerting layer, and it costs you a little credibility every time it fires.

Be precise about what tests fix, because many vendors are not. Most of your breakage starts upstream, in systems you don't own, run by people who don't know you exist. Tests don't prevent that. The Salesforce admin will still add a field on a Tuesday without telling anyone. What tests change is when you find out and how far it spread. Monday at 9am from finance becomes Tuesday at 6:14am from a check that ran before anything downstream read the table. The claim stops there. It's enough.

The counter is a test that runs before humans read the data. The objection is always that nobody has time to write tests for 4,000 tables, which is true and also the wrong frame, and we got roasted on Reddit for asking about it. You don't write them. You profile what the data actually does and generate tests from

that: column ranges, null rates, cardinality, freshness, referential integrity, distribution shifts. Then you keep the ones that make sense. An afternoon, not a project.

Start where a wrong number reaches someone who can escalate it. Usually a couple dozen tables, not 4,000, which is what test coverage as a measure actually means, and it's a better filter than picking by volume or by which pipeline annoys you most.

Then tune, because this is where generated tests usually die. Profile-based checks will flag legitimate variation. Somebody mutes the channel, and six weeks later the tests are decoration. Split them from the start. Hard constraints you know are true, like a key being unique or an amount never going negative, can fail loudly on day one. Statistical checks like distribution shifts and volume ranges run in warn-only mode for a couple of weeks while you learn what normal looks like. Promote them once they stop crying wolf.

Wire the ones you trust into the path your changes already take. On a normal stack, that means the tests run in the pull request, and the same checks run again against production data right after the load, before anything downstream reads the table. That is what shifting left and shifting down means in practice.

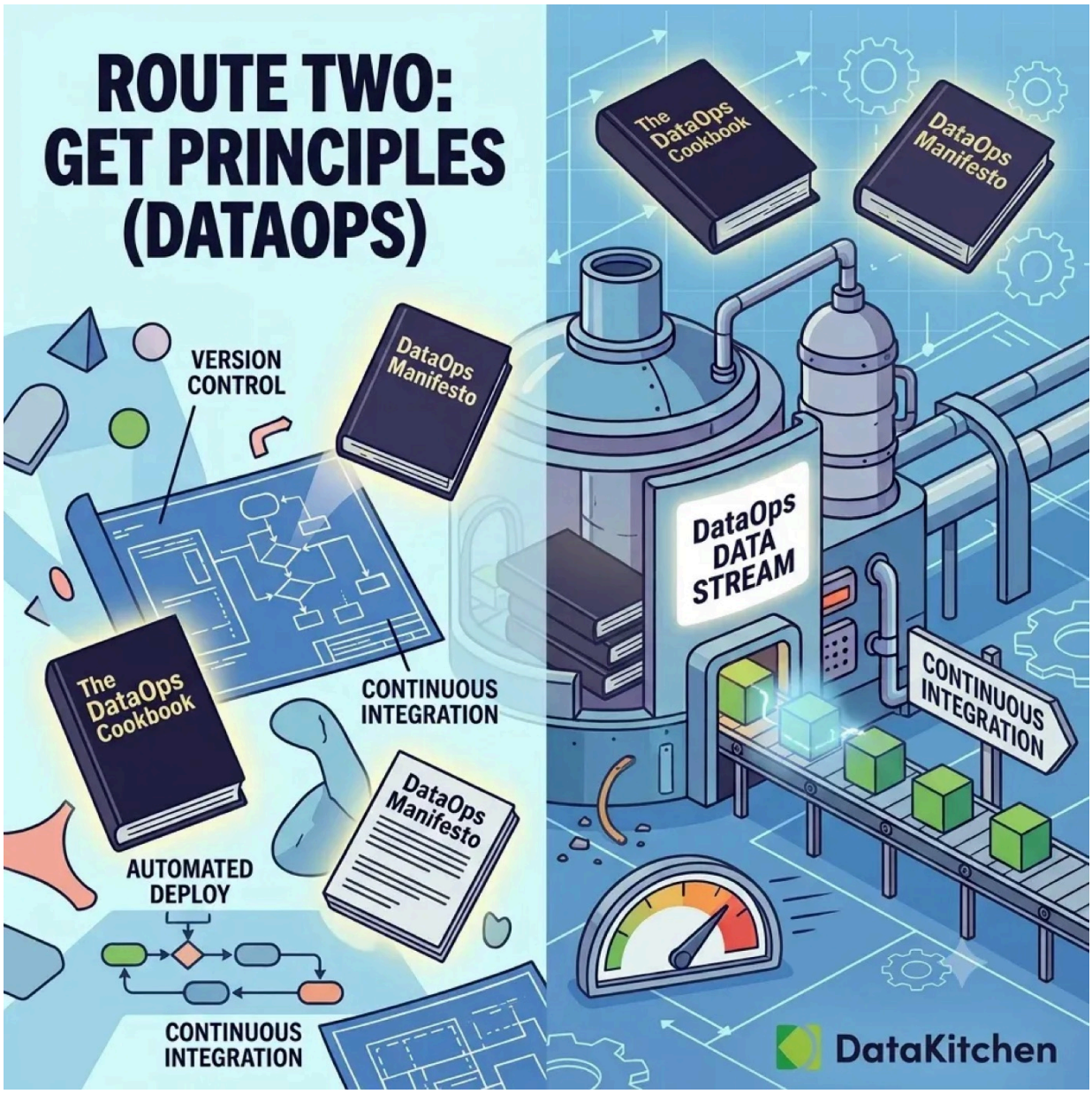
Decide in advance what a failure does, because halting is not free. Stop the pipeline and finance opens an empty dashboard instead of a wrong one, and now you're explaining that instead. Your rule: halt when the data would be silently wrong, warn when it would be obviously incomplete. A wrong number gets used. A missing one gets questioned.

This route works on the system you already have. No migration, no reorg, nothing to procure. It's also the only one of the three you can start without asking anyone for permission, which matters more than it should.

Route two: steal the principles

The second route costs the least money and the most patience. Start it yourself on Monday, for free.

Hold a blameless postmortem on every data incident. 15 minutes, written down: what broke and why, no names attached to blame. Keep the list somewhere shared. Read it once a month. After a quarter, the things eating your week will have named themselves, and you'll be holding the evidence you need to ask for anything bigger.



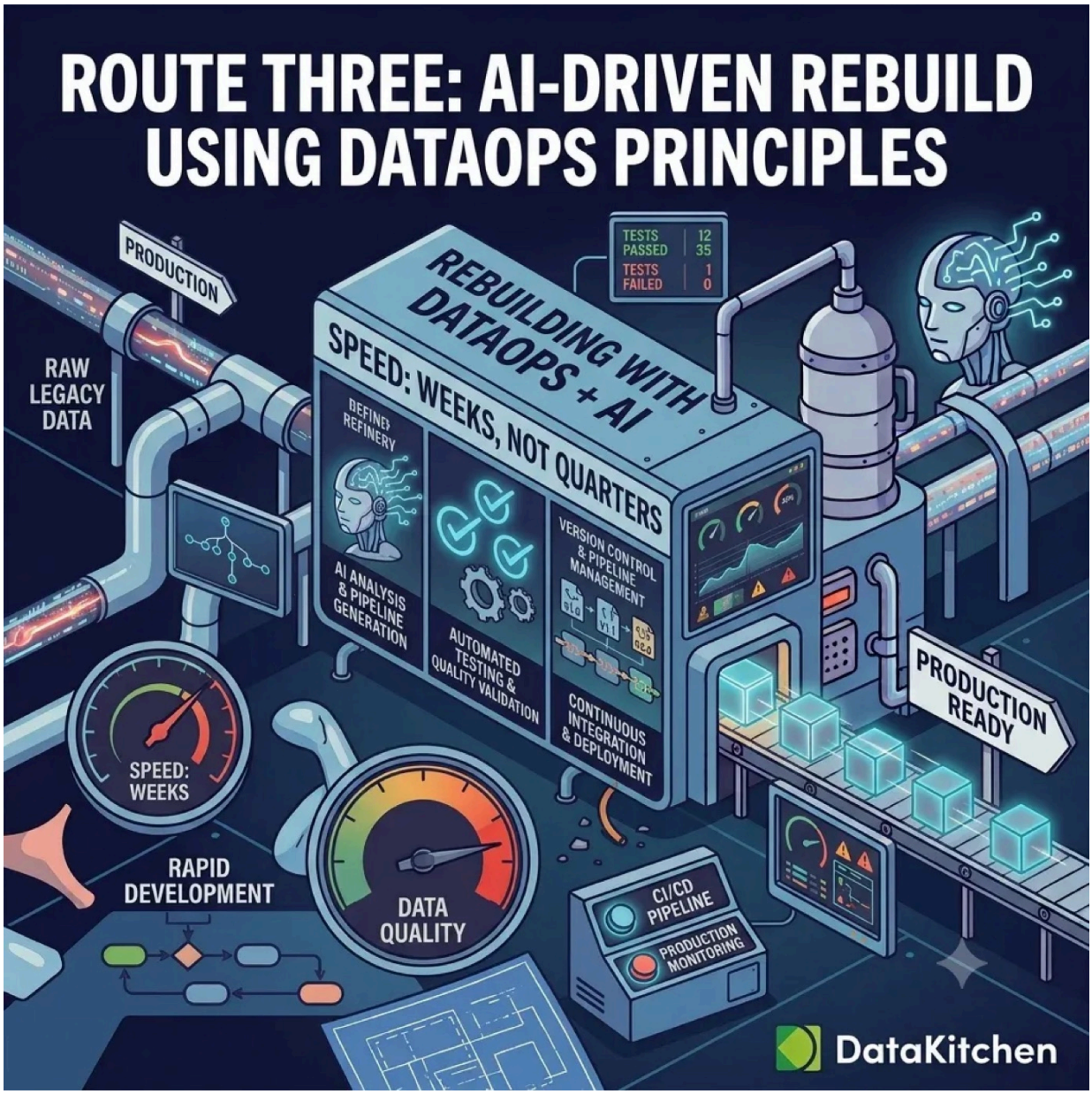
Then bring in people who have made data teams faster before, and take their principles rather than their labor. This works because much of what slows you down is not technical. It's that every change requires a human to verify it manually. It's that nobody agrees on which tables matter, so either everything gets watched or nothing does. It's that one person knows why that filter is there, so when they're on vacation, changes stop or break. That last one has a name, process guardianship, and it is usually nobody's job. No tool fixes those. A different way of working does.

The ideas are not secret. Software teams and manufacturing plants solved most of this decades ago with version control, automated testing, continuous integration, and statistical process control. Read the foundational DataOps material before you pay anyone, because then you can tell whether the consultant across the table knows the practice or is selling you a maturity model. If you want that material in an order, Where to Actually Start with DataOps in 2026 lays out what to hand your team and what to do first.

The honest warning on this route: principles only move the number if leadership changes what it rewards. If your team is still measured by tickets closed, you'll end up with the best firefighters in the industry and no fire code. Advice can't fix an incentive.

Route three: rebuild it, if it needs rebuilding

The most expensive route, and the one to be honest with yourself about. Plenty of the pipelines you hate are fine underneath and just have no tests. Rebuild when the architecture is the problem, not when the last six months were painful.



The classic rewrite failed for a predictable reason. It took 12 months, the old pipelines kept breaking the whole time, and at month nine the sponsor changed and the project got shelved. Everyone reading this has seen that happen at least once.

AI changed the arithmetic, though less than the pitch decks claim. What collapsed is the mechanical work. A transform that took an engineer a day takes a couple of hours now, and that is the part we can show from our own work. It works best when the pipelines are functional and idempotent, because that is what lets AI reason about them and change them safely. What did not collapse is the rest of it: agreeing what the output should be, running old and new side by side until the numbers match, and cutting over without breaking the reports people already depend on. That still sets the calendar. So the rewrite is a smaller bet than it was three years ago, which is worth repricing if that is the last time you looked at it. Anyone quoting you a fixed number of weeks has not seen your pipelines.

Two things determine whether it works.

The first is who runs it afterward. A rebuild that ends with an outside team holding the keys has only moved your bottleneck somewhere with a contract attached. The handover is the deliverable, not the code.

The second is tests, which is why this route is third and not first. AI will generate a pipeline that looks right, runs clean, and is quietly wrong in a way nobody notices for a month. Without tests, you have built a much faster way to produce errors you can't see. If you're going to rebuild with AI, do route one first, or at a minimum in parallel. Otherwise you're buying speed and paying for it in Mondays.

The order

Be realistic about which of these you can actually start. Route one you can begin this week on your own authority. Routes two and three need a sponsor, because you don't set what your organization rewards and you can't authorize a rebuild. So do route one first. The number it produces is how you get the sponsor.

Most teams do this in reverse. They rebuild first because it feels like progress, hire a consultant when the rebuild stalls, and get to testing in year three when someone finally adds up the hours.

What next Monday looks like

The load runs at 6:14. A test fails on row count and the pipeline stops before anything downstream reads it. Somebody on your team gets a message at 6:20 with the table, the test, and the column that changed. They fix it before finance opens the dashboard.

Nobody outside the team ever knows it happened. That's what efficient looks like. It's boring, and boring is the point.

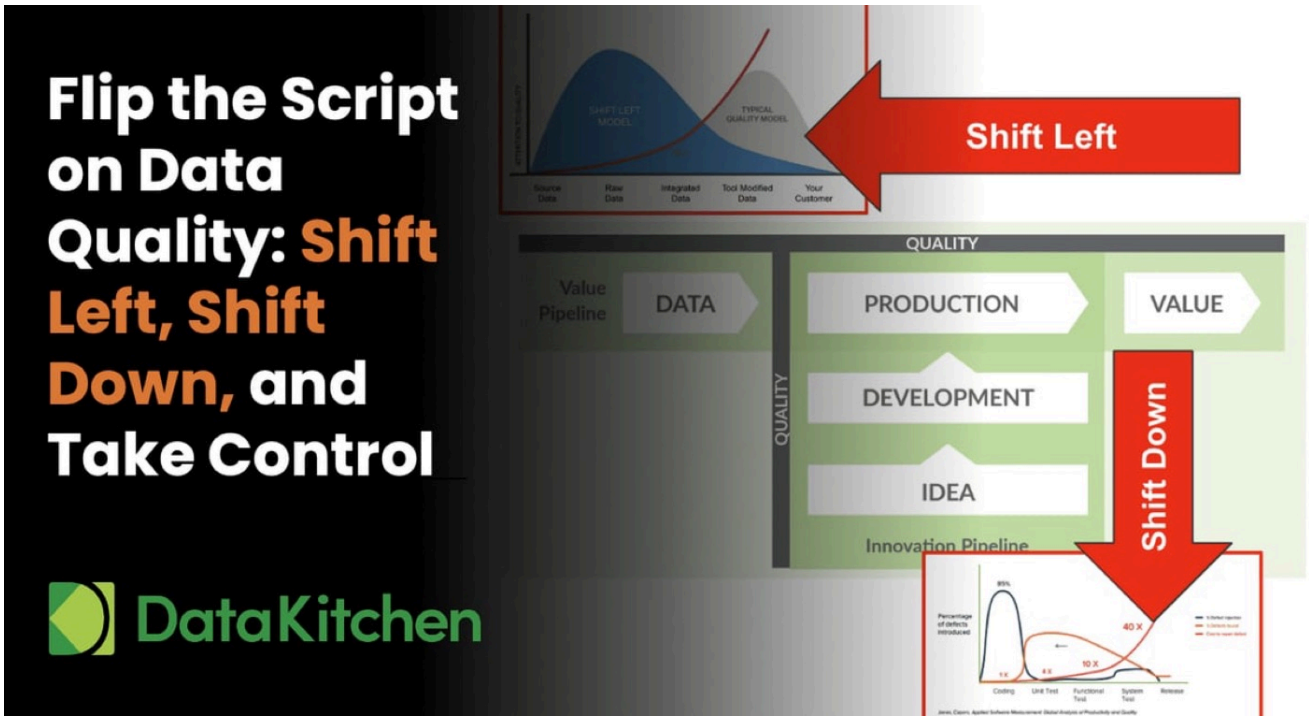
SECTION 3

Building Your Testing Strategy

1. Shift-Left, Shift-Down: How to Take Control of Data Quality
2. Scaling Data Reliability: The Definitive Guide to Test Coverage
3. Data Quality Test Coverage in a Medallion Data Architecture
4. The Race for Data Quality in a Medallion Architecture
5. Building Data Quality Into dbt: A Complete Guide
6. Data Contracts: Shifting the Burden of Quality to the Source
7. Building and Scaling a Data Quality Practice: The First Year
8. Impact Dimensions Cure A Data Quality Blind Spot
9. Your Best Data Quality Rules Live in Someone Else's Head
10. The Blueprint or the Fire Alarm Panel: Why Data Test Coverage Beats Data Lineage
11. The Data Contract Is Your Receipt
12. The Four Points in Your Medallion Architecture Where Data Testing Really Matters
13. Testing Every Layer of the Medallion Architecture, Now with OneLake at the Center

Shift-Left, Shift-Down: How to Take Control of Data Quality

Read the original article at: https://datakitchen.io/blog/flip-the-script-on-data-quality-shift-left-shift-down-and-take-control/



Data quality problems are almost always discovered at the wrong time and in the wrong place. A business analyst finds a discrepancy in a dashboard. A data scientist notices that a feature distribution has shifted. An executive questions a number in a board presentation. By that point, the defect has already traveled through multiple pipeline stages, potentially contaminated several downstream systems, and may have been sitting undetected for days or weeks.

Two moves address this systematically: shift-left and shift-down. They are independent dimensions of improvement, and the most effective data teams apply both.

Shift-Left: Test Earlier, Closer to the Source

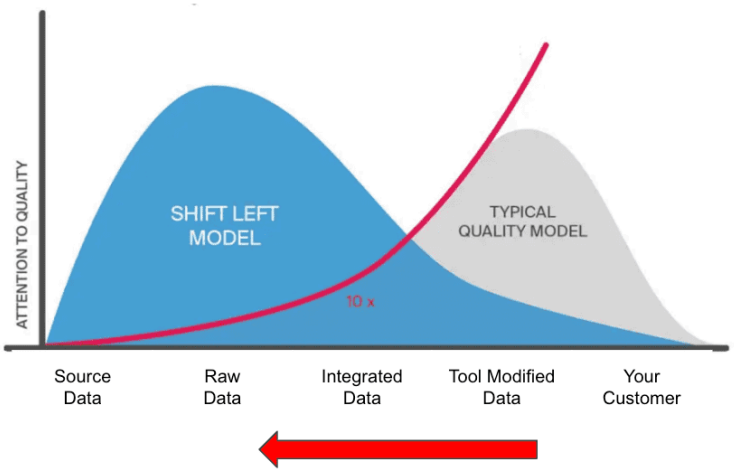
Shifting left means moving data quality checks earlier in the pipeline, closer to the point where data originates. The term comes from software engineering, where "left" represents the earlier stages of a development timeline. In data, shifting left means not waiting until data reaches the warehouse or the BI layer to validate it.

Shift Left Test Coverage Saves Time

It is 10x cheaper to find a problem in data at the source or raw layer.

Apply the 1:10:100 rule*:

- Cost of preventing poor data quality at source is $1 per record
- Cost of remediation after it is created is $10 per record
- Cost of failure (i.e. doing nothing) is $100 per record



**developed by George Labovitz and Yu Sang Chang back in 1992*

The manufacturing analogy is useful here. In assembly-line production, catching a defective component before it is installed in a product costs almost nothing. Catching it after the product has been assembled and shipped costs orders of magnitude more. Data pipelines have the same economics. A null value that is caught at ingestion requires a one-line fix. That same null value, discovered after it has propagated through three transformation layers and been aggregated into a financial report, requires a full incident investigation, data backfill, and stakeholder communication.

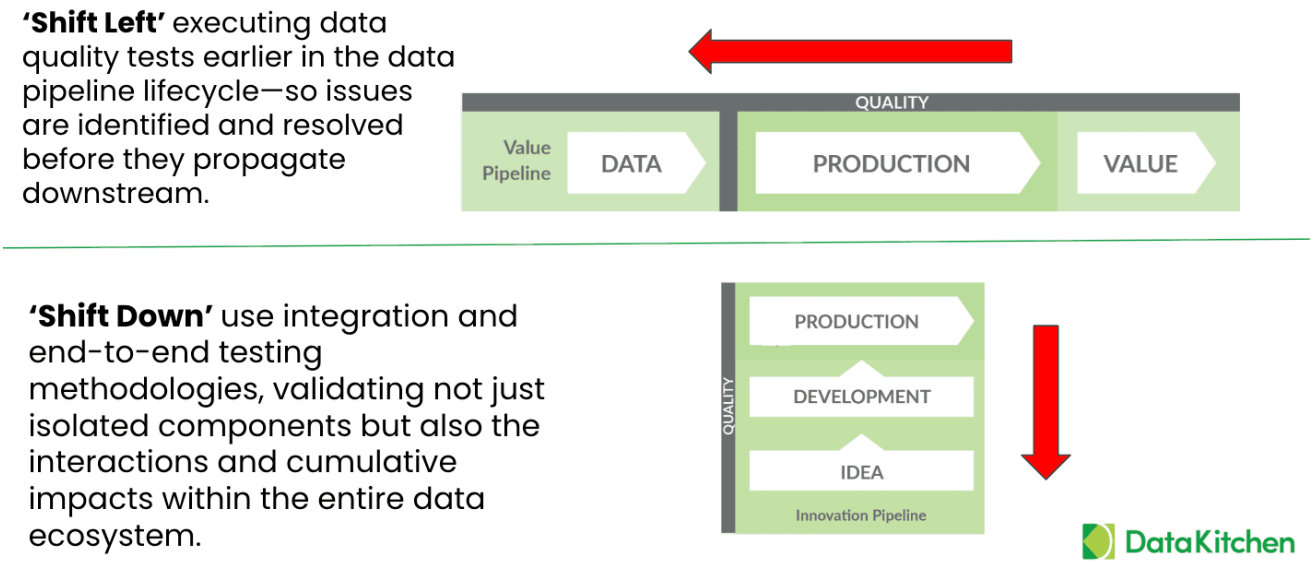
Practical shift-left actions include:

- Validating schema conformance at the point of ingestion, before data enters any storage layer
- Running row-count and completeness checks on raw data before transformation begins
- Testing data contracts at API or file-drop boundaries, not just at the warehouse boundary
- Using source-system monitoring to detect anomalies before the extract runs

Shifting left does not eliminate downstream testing. It reduces the blast radius of defects and raises the signal-to-noise ratio of downstream alerts by catching the obvious failures early.

Shift-Down: Test at the Infrastructure Layer

Shifting down is the more novel and often underutilized dimension. It means moving data quality checks from the application layer down into the infrastructure layer, where tests can run closer to the compute and storage systems that execute pipelines.



The distinction matters because application-layer testing depends on code paths that may themselves be buggy, misconfigured, or incomplete. Infrastructure-layer testing operates independently. It can catch failures that the application layer never surfaces: silent data drops, partial writes, configuration drift, and resource exhaustion that causes truncated output.

Think of it this way: an application-layer test says "the rows I received look correct." An infrastructure-layer test says "the storage system actually received all the rows it was supposed to, and the job that ran had the resources to complete correctly."

Shifting down means instrumenting at the level of the orchestrator, the compute cluster, the storage system, and the network. It means treating data quality as an infrastructure concern, not just an application concern.

Shift-Down in Batch Pipelines

In a batch pipeline, a typical application-layer test checks that the output table has the expected row count and that certain columns pass validation rules. This is necessary but not sufficient.

Infrastructure-layer tests for a batch pipeline include:

- Job completion verification at the orchestrator level: did the Airflow task actually complete, or did it silently time out after writing partial output?
- Resource saturation checks: did the Spark job spill to disk excessively, indicating it processed less data than expected due to memory pressure?
- Storage write verification: did the file or partition actually land in the target location with the expected size? A zero-byte file is a valid write from a filesystem perspective but a failed job from a data perspective.
- Checkpoint validation: for multi-stage batch jobs, does each intermediate stage produce the expected output volume before the next stage consumes it?

A common failure mode in batch pipelines is the silent partial write. The job starts, processes some data, hits a resource limit or a transient network error, writes partial output, and exits with a success code because the failure happened in a worker process rather than the driver. The application-layer row count check then validates the partial output as if it were complete. Infrastructure-layer monitoring catches this by comparing expected output size against actual storage metrics independently of the job's self-reported status.

Shift-Down in Streaming Pipelines

Streaming pipelines introduce a different set of infrastructure-layer failure modes. Data does not arrive as discrete batches with clear boundaries. It arrives as a continuous flow, and the infrastructure must continuously verify that the flow is healthy.

Infrastructure-layer tests for a streaming pipeline include:

- Consumer lag monitoring: is the Kafka consumer keeping up with the producer, or is the lag growing? Unbounded lag growth means data is being produced faster than it is being consumed, which eventually leads to data loss when topics hit retention limits.
- Throughput rate monitoring: is the messages-per-second rate within the expected range? A sudden drop in throughput can indicate a producer failure, a schema rejection storm, or a serialization error that is silently discarding messages.
- Deserialization error rate: what fraction of incoming messages are failing to deserialize? A non-zero deserialization error rate means data is being lost at the entry point, before any application-layer logic runs.
- Watermark progress: for event-time processing, is the watermark advancing at the expected rate? A stalled watermark indicates that late or out-of-order events are holding up downstream aggregations.

The critical difference from batch is that streaming failures are often gradual rather than total. A streaming pipeline can be losing 0.3% of events for days before any application-layer metric shows a problem. Infrastructure-layer monitoring detects this by watching rates and ratios, not just point-in-time counts.

Shift-Down in Medallion Architecture Pipelines

The medallion architecture (Bronze, Silver, Gold) is increasingly the standard pattern for lakehouse pipelines. Each layer has a different contract and a different set of infrastructure-layer tests.

Bronze (raw ingestion layer): The Bronze layer holds raw, unmodified copies of source data. Infrastructure-layer tests here focus on completeness and latency. Did all the expected files arrive? Did they arrive on time? Are file sizes within the expected range, or are some suspiciously small? Bronze-layer infrastructure tests do not validate content; they validate delivery.

Silver (cleaned and conformed layer): The Silver layer applies transformations: deduplication, schema normalization, type casting, and business rule application. Infrastructure-layer tests here focus on transformation fidelity. The row count going into Silver should be explainable in terms of the row count coming out of Bronze, accounting for known deduplication rates. If Bronze has 1 million rows and Silver

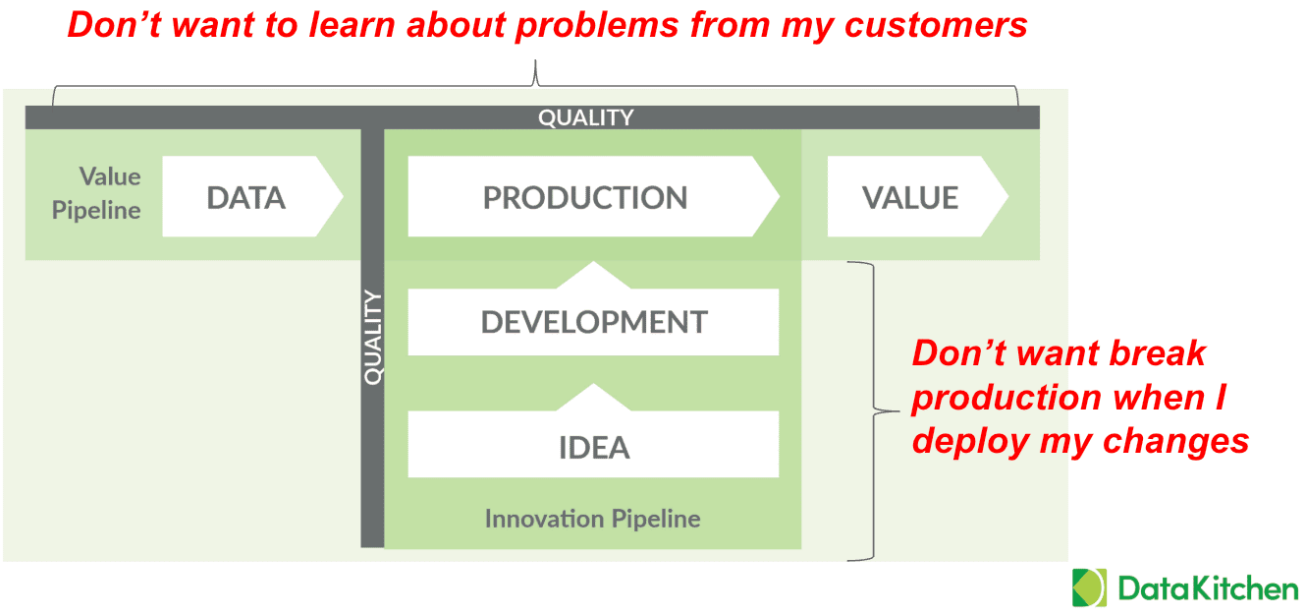
has 200,000 after deduplication, that is expected. If Silver has 200 rows, something went wrong at the infrastructure level, not the application level. Partition-level checks, job resource metrics, and intermediate stage output sizes all belong here.

Gold (aggregated, business-ready layer): The Gold layer serves analytics and reporting workloads. Infrastructure-layer tests here focus on availability and consistency. Is the Gold layer table available when downstream jobs expect it? Is it consistent across partitions, meaning no partition was written from a different version of the transformation logic than another? Is the write timestamp within the SLA window?

The following table summarizes the shift-down testing approach across pipeline types and layers.

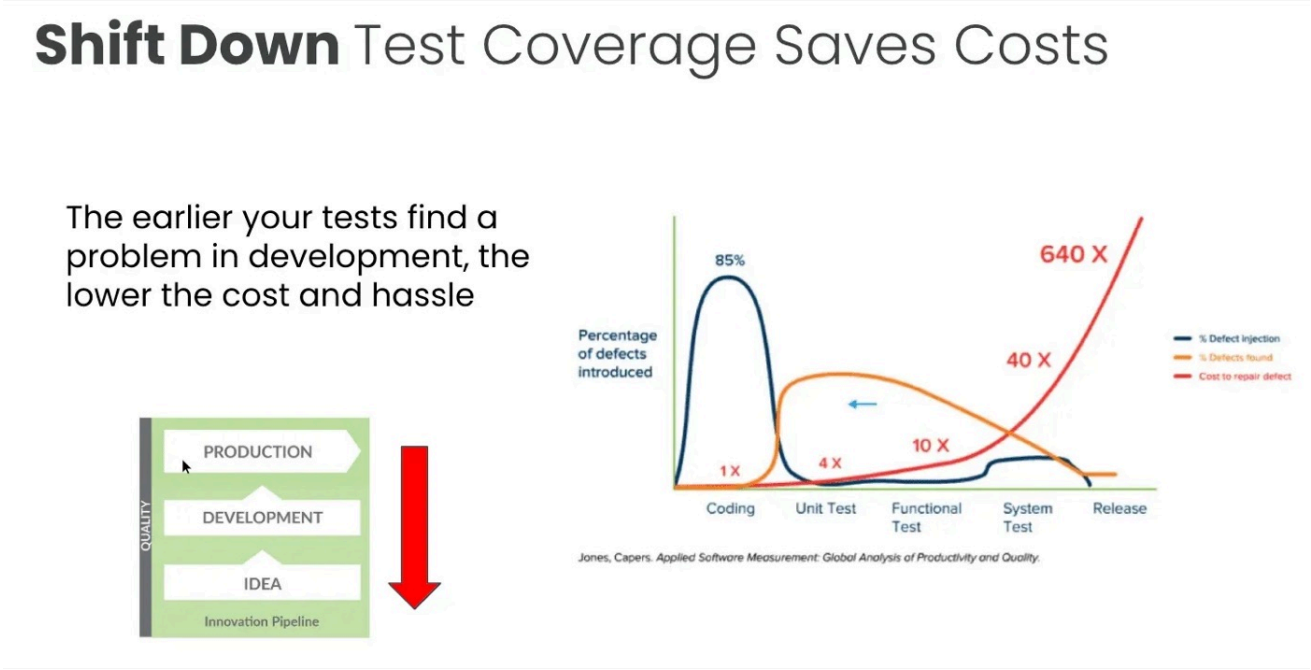
Reference Table: Shift-Down Testing by Pipeline Layer

Pipeline Layer	What to Test	Why	Typical Failure Mode	Example Test
Batch: raw ingest	File size and row count at storage layer	Silent partial writes exit with success code	Job writes 40% of expected rows due to memory spill, reports success	Compare S3 object size against 7-day rolling average
Batch: transformation	Intermediate stage output volume	Downstream stage consumes partial input silently	Stage 2 reads from Stage 1 before Stage 1 completes write	Assert intermediate partition row count before Stage 2 starts
Batch: orchestrator	Task completion and duration	Tasks can time out after partial write	Airflow task marked success after 2-hour timeout with truncated output	Alert if task duration exceeds 2x median historical runtime
Streaming: consumer	Consumer lag by topic and partition	Lag growth leads to data loss at retention boundary	Kafka consumer lag grows 10x over 6 hours, reaching retention limit	Alert if lag exceeds 100k messages or grows more than 20% per hour
Streaming: ingest	Deserialization error rate	Schema changes cause silent message drops	Producer deploys new Avro schema without updating consumer; 30% of messages dropped	Alert if deserialization error rate exceeds 0.1% over 5-minute window
Streaming: throughput	Messages per second by topic	Throughput drop indicates producer or network failure	Producer crash causes 0 messages/sec for 20 minutes before alerting	Alert if throughput drops below 50% of 24-hour rolling average
Bronze (medallion)	File arrival, file size, partition count	Missing files are not surfaced by downstream transforms	Source system skips one daily partition; downstream joins produce wrong counts	Assert all expected daily partitions exist before Silver job starts
Silver (medallion)	Row count ratio Bronze to Silver	Transformation bugs cause data amplification or mass deletion	Deduplication logic bug removes 80% of rows instead of 5%	Assert Silver row count is within 10% of (Bronze rows * expected dedup rate)
Gold (medallion)	Write timestamp, partition consistency	Stale Gold tables silently serve old data to dashboards	Gold table written at 3am instead of 11pm; dashboards show yesterday's data as current	Assert Gold write timestamp is within SLA window before BI tools refresh



Applying Both Dimensions Together

Shift-left and shift-down are complementary, not substitutes. A team that only shifts left catches defects earlier in the pipeline but still relies on application-layer tests that can miss infrastructure failures. A team that only shifts down catches infrastructure failures but may let content-level data quality problems propagate unchecked.



The practical implementation sequence for a team starting from a baseline of only downstream, application-layer testing:

First, identify the three highest-severity pipelines in production. Map where current tests run relative to the pipeline stages. This usually reveals a cluster of tests concentrated at the final output layer.

Second, add one shift-left check per pipeline: a schema conformance check at the ingestion boundary. This is low effort and catches the most common class of upstream failure, which is a source system change that the pipeline does not handle.

Third, add one shift-down check per pipeline: an orchestrator-level duration and completion check. Jobs that run significantly longer or shorter than their historical baseline are flagging an infrastructure-level problem regardless of whether the output looks correct.

Fourth, add Bronze-layer completeness checks for any medallion architecture pipelines. These are the highest-leverage shift-down checks available in that architecture because Bronze failures propagate silently through every downstream layer.

The goal is not to instrument everything at once. It is to progressively move the detection point closer to the origin of each class of failure. Every check moved left or down reduces the time between defect creation and detection. Every reduction in detection time reduces the cost of remediation and the blast radius of the problem.

Data quality work has a compounding return structure: the earlier and deeper you catch problems, the less time you spend on incidents, and the more time you have to add more coverage. Teams that invest in shift-left and shift-down testing consistently report that incident volume drops as coverage improves. The opposite pattern, reactive testing concentrated at the output layer, produces a steady-state incident rate that never improves regardless of how much monitoring is added downstream.

Scaling Data Reliability: The Definitive Guide to Test Coverage

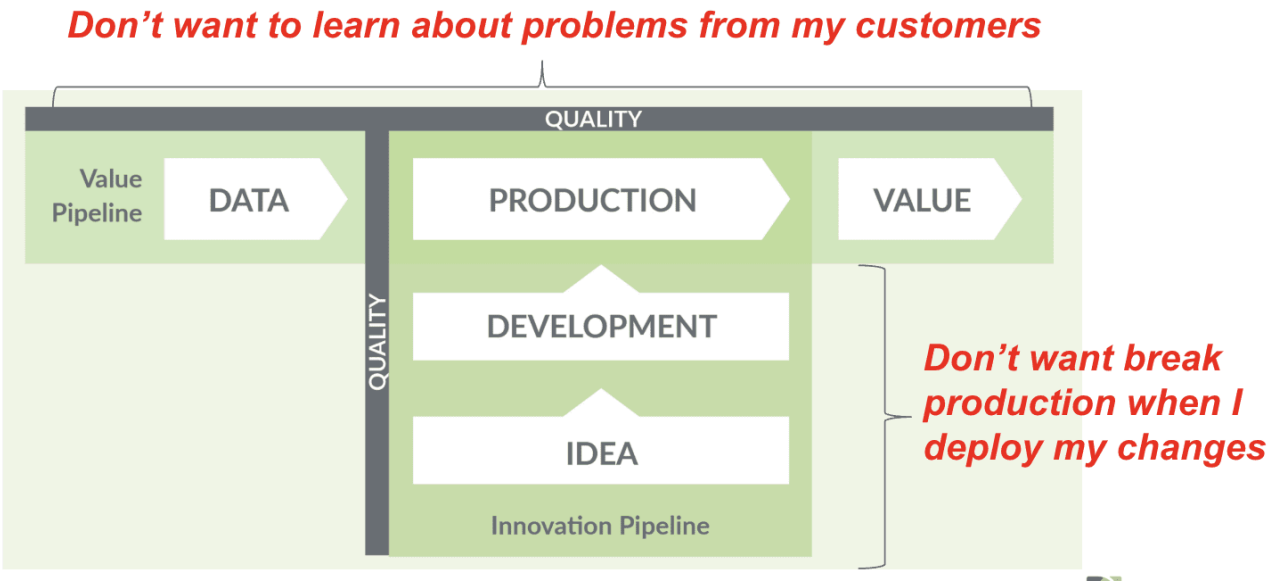
Read the original article at: https://datakitchen.io/blog/scaling-data-reliability-the-definitive-guide-to-test-coverage/

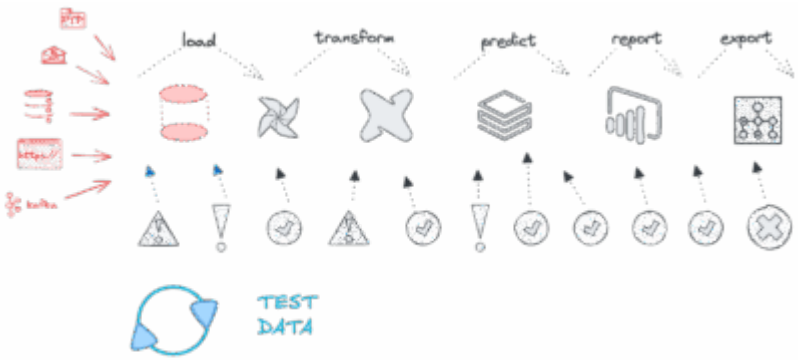
Data quality testing does not have a single correct level of coverage. It has a correct level of coverage for each dataset, determined by how much it matters and how often it changes. A revenue-facing orders table and a rarely-queried internal staging table are not the same kind of asset. Treating them identically either under-protects critical data or wastes engineering time on data nobody depends on.

This chapter explains how to design a test coverage strategy that is proportionate, comprehensive for the tables that matter most, and sustainable at scale. It covers every major test type, walks through a complete test suite for a specific high-criticality table, and defines the SLA expectations that make coverage actionable rather than theoretical.

The Two Dimensions of Data Quality Testing

Data quality tests operate across two dimensions: where in the pipeline lifecycle they run, and what they check.



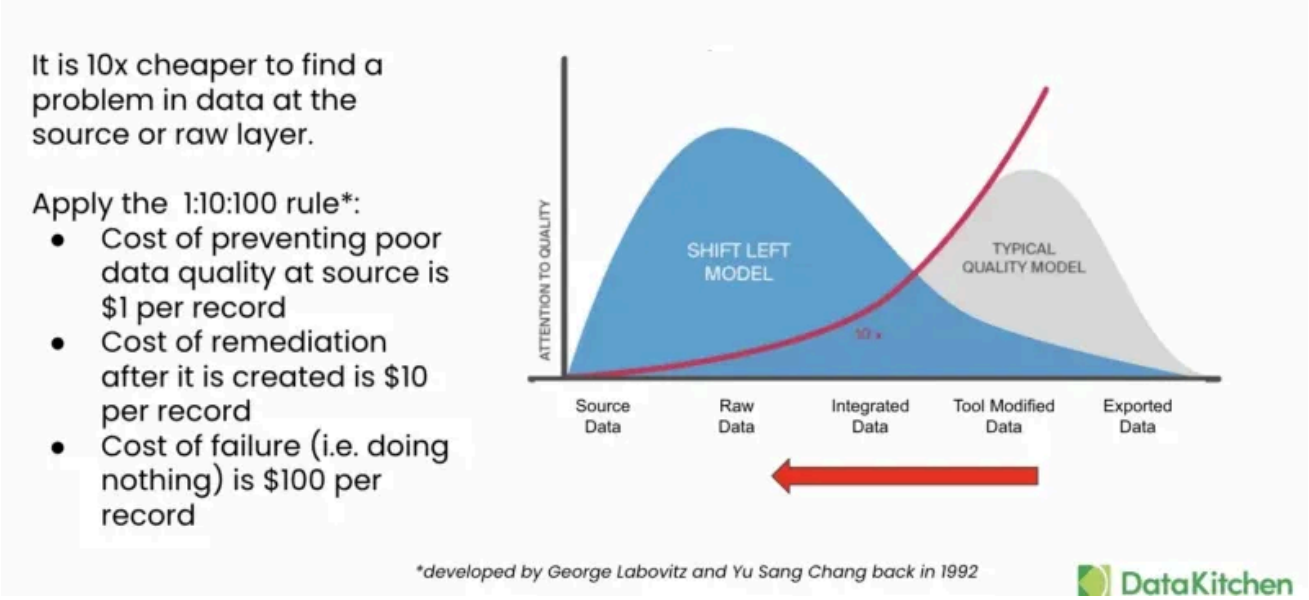


The lifecycle dimension divides into production testing and development testing. Production tests run continuously against live data in production environments. They answer the question: is this data healthy right now? Development tests run during pipeline development, code review, and deployment. They answer the question: does this change introduce a regression?

Both are necessary. A team with only production testing catches problems after they have already affected real data. A team with only development testing catches regressions in code but misses the class of failures that emerge from changing source data, volume changes, or environmental drift.

Risk Tiers: Matching Coverage to Business Impact

Not all tables have the same business impact. Assigning every table the same level of monitoring is neither feasible nor useful. Risk tiers provide a systematic way to match coverage intensity to business criticality.



Tier 1: Business-Critical Tables

Tier 1 tables directly support revenue reporting, executive dashboards, regulatory compliance, or real-time operational decisions. A failure in a Tier 1 table has immediate, visible business consequences. Examples: `orders`, `revenue_daily`, `customer_lifetime_value`, `inventory_on_hand`.

Tier 1 tables require comprehensive test coverage across all test types. They are checked hourly. Failures trigger immediate alerts to a named owner. SLA violations trigger escalation.

Tier 1 SLA: Freshness checked every 60 minutes. Maximum acceptable latency from source event to table availability: 30 minutes. Completeness threshold: 99.5% non-null on required fields. Row count variance tolerance: 15% from 7-day rolling average. Alert response SLA: acknowledged within 15 minutes.

Tier 2: Operationally Important Tables

Tier 2 tables support internal operations, product analytics, or team-level reporting. A failure causes analyst confusion or delayed reporting but does not directly affect revenue or compliance. Examples: `product_page_views`, `support_ticket_summary`, `marketing_attribution`.

Tier 2 tables require coverage across the primary test types. They are checked daily. Failures trigger alerts but do not require immediate response.

Tier 2 SLA: Freshness checked every 24 hours. Maximum acceptable latency: 2 hours. Completeness threshold: 98% non-null on required fields. Row count variance tolerance: 25% from 7-day rolling average. Alert response SLA: acknowledged within 4 hours.

Tier 3: Supporting and Staging Tables

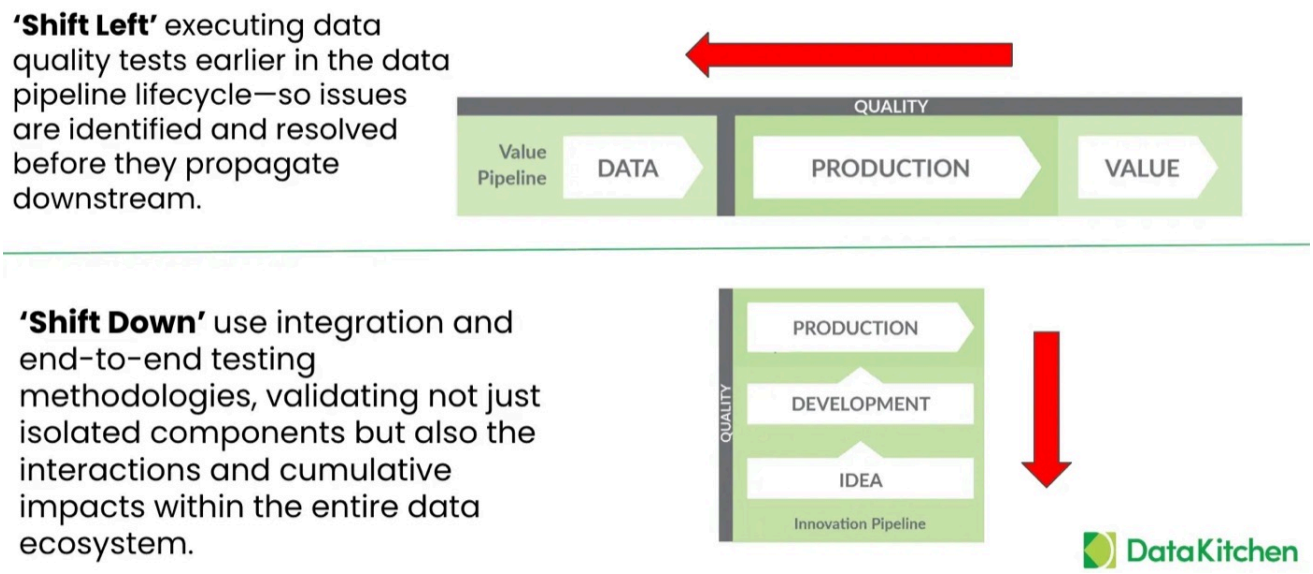
Tier 3 tables are intermediate pipeline outputs, rarely-queried reference tables, or staging tables used only during development and testing. A failure in a Tier 3 table typically only affects the data engineers maintaining the pipeline. Examples: `raw_event_staging`, `temp_user_mapping`, `archive_clickstream_2021`.

Tier 3 tables require minimal coverage. Basic freshness and row-count checks are sufficient. They are checked weekly.

Tier 3 SLA: Freshness checked every 7 days. Row count sanity check only. No completeness or business rule testing required. Alert response SLA: acknowledged within 48 hours.

The Five Test Types

Every data quality test belongs to one of five categories. A comprehensive test suite for a Tier 1 table requires all five.



1. Cardinality Tests (Row Count)

Cardinality tests verify that a table contains the expected number of rows. They are the simplest form of data quality test and often the most effective at detecting major pipeline failures.

A row count test checks that the actual row count is within an acceptable range of the expected row count. The expected count can be defined as a static threshold ("at least 10,000 rows per day") or as a dynamic range derived from historical data ("within 20% of the 7-day rolling average").

Static thresholds are brittle because data volumes change over time. Dynamic thresholds based on historical averages are more robust and require less maintenance.

What cardinality tests catch: complete pipeline failures, partial writes, duplicate-row bugs that produce more rows than expected, and aggressive deduplication bugs that produce far fewer rows than expected.

What cardinality tests miss: row-level correctness problems where the count is right but the values are wrong.

2. Validity Tests (Null Checks, Range Checks, Format Checks)

Validity tests verify that individual column values meet defined constraints. They are the workhorses of data quality testing because column-level corruption is the most common type of data quality failure.

Common validity test types:

- Null checks: assert that required columns contain no nulls
- Range checks: assert that numeric values fall within expected bounds (e.g., `order_amount` is greater than 0 and less than 1,000,000)
- Allowed-value checks: assert that categorical columns contain only values from a defined set (e.g., `order_status` is one of: pending, processing, shipped, delivered, cancelled)
- Format checks: assert that string columns match a required pattern (e.g., `email` matches a valid email format, `zip_code` is exactly 5 digits)

What validity tests catch: schema changes that produce nulls in required fields, application bugs that produce out-of-range values, and upstream data entry errors.

What validity tests miss: plausible but incorrect values that fall within the valid range.

3. Freshness Tests (Latency)

Freshness tests verify that data has been updated within the expected time window. They answer the question: is this table stale?

A freshness test checks the maximum value of a timestamp column (such as `updated_at` or `loaded_at`) and asserts that it is within the defined latency threshold. For a table that should update every hour, the most recent record should have a timestamp no older than 90 minutes (allowing a 50% buffer for normal variation).

What freshness tests catch: pipeline failures that leave the table in a stale state, jobs that completed but wrote no new rows, and schedule drift in orchestrators.

What freshness tests miss: pipelines that run on schedule but write corrupted or stale data (the job ran, but the data is wrong).

4. Referential Integrity Tests (Foreign Keys)

Referential integrity tests verify that foreign key relationships between tables are intact. They ensure that every reference from one table to another resolves to a valid record.

For example, every `order_id` in the `order_line_items` table should have a corresponding row in the `orders` table. A referential integrity test computes the set of `order_id` values in `order_line_items` that do not appear in `orders` and asserts that this set is empty (or below a defined tolerance threshold for systems with known replication lag).

What referential integrity tests catch: orphaned records created by partial pipeline failures, join key mismatches caused by schema changes or ID format changes, and data loss in referenced tables.

What referential integrity tests miss: logical mismatches where the reference exists but the data in the referenced row is incorrect.

5. Business Rule Tests (Custom Logic)

Business rule tests encode domain-specific invariants that the other test types cannot express. They validate that data makes sense in the context of the business, not just in the context of the schema.

Examples of business rules:

- An order with `status = delivered` should have a non-null `delivered_at` timestamp
- `revenue` should equal the sum of `quantity * unit_price` across all line items for that order
- A customer's `first_order_date` should not be later than their `most_recent_order_date`
- Daily `order_count` should not decrease day-over-day (orders are not deleted)

Business rule tests require the most domain knowledge to write but catch the class of errors that every other test type misses: data that is structurally valid but semantically wrong.

Complete Test Suite: The Orders Table

To make the framework concrete, here is the complete test suite for the `orders` table at a mid-sized e-commerce company. This table is Tier 1: it feeds the revenue dashboard, the finance reconciliation process, and the operations team's fulfillment tracking.

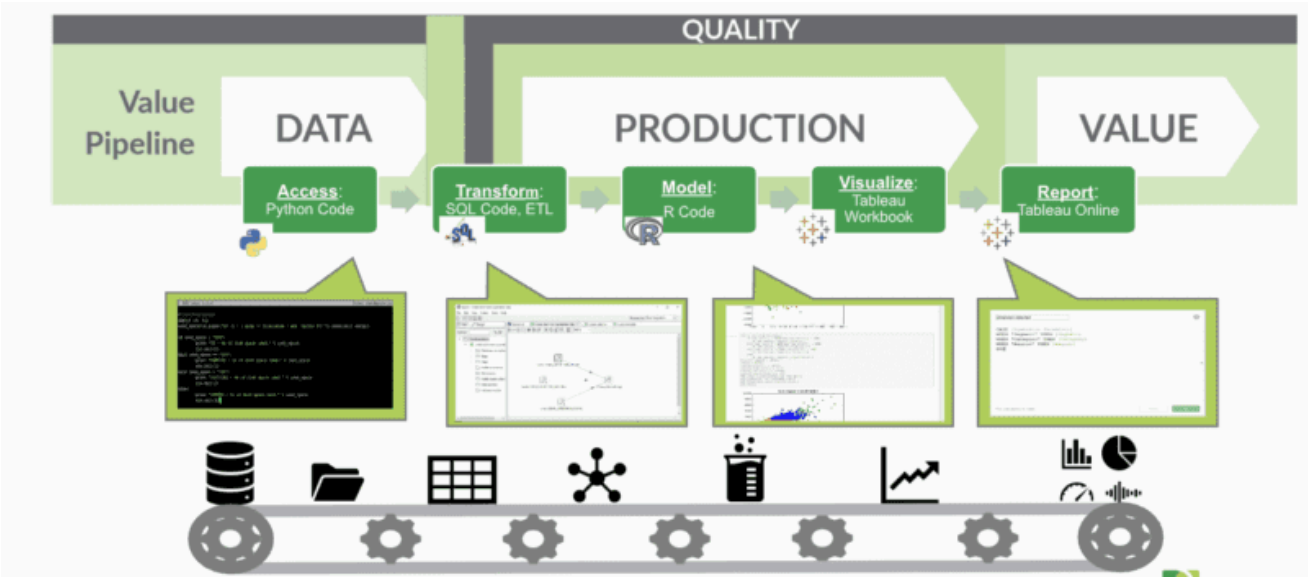


Table Schema (Abbreviated)

Column	Type	Nullable	Description
order_id	STRING	No	Unique order identifier
customer_id	STRING	No	Foreign key to customers table
order_status	STRING	No	One of: pending, processing, shipped, delivered, cancelled
order_amount	DECIMAL	No	Total order value in USD
order_date	TIMESTAMP	No	Timestamp of order placement
shipped_date	TIMESTAMP	Yes	Null until order ships
delivered_date	TIMESTAMP	Yes	Null until order is delivered
shipping_method	STRING	No	One of: standard, expedited, overnight
updated_at	TIMESTAMP	No	Last pipeline update timestamp

Cardinality Tests

Test C1: Daily row count within historical range

- Check: row count for each calendar day is within 20% of the 7-day rolling average for that day of week
- Why: catches complete pipeline failures, mass deduplication errors, and duplicate-row bugs
- Failure mode: ETL job writes partial batch due to memory spill; row count is 40% of expected
- Run frequency: every pipeline run (hourly)

Test C2: No duplicate order_id values

- Check: count of `order_id` equals count of distinct `order_id`
- Why: catches deduplication logic failures that produce multiple rows per order
- Failure mode: change data capture (CDC) pipeline writes both the insert and the update record for the same order
- Run frequency: every pipeline run

Validity Tests

Test V1: order_id not null

- Check: zero rows where `order_id` is null
- Why: primary key nulls indicate a fundamental pipeline failure
- Failure mode: source system migration generates rows with null IDs during transition period

Test V2: customer_id not null

- Check: zero rows where `customer_id` is null
- Why: null customer IDs break customer-level analytics and LTV calculations

- Failure mode: guest checkout feature ships without assigning customer IDs, produces nulls

Test V3: order_amount within valid range

- Check: all values of `order_amount` are greater than 0 and less than 50,000
- Why: negative or zero amounts indicate refund processing errors; extreme values indicate currency conversion bugs
- Failure mode: a currency conversion factor is applied twice, producing order amounts 100x too large

Test V4: order_status is a valid value

- Check: all values of `order_status` are in the allowed set
- Why: new status values added in the application without updating downstream mappings cause silent categorization errors
- Failure mode: application team adds `return_requested` status; pipeline maps it to null, breaking fulfillment reports

Test V5: shipping_method is a valid value

- Check: all values of `shipping_method` are in the allowed set (standard, expedited, overnight)
- Why: directly connected to the schema-change scenario described in the tragedy of the commons chapter
- Failure mode: column renamed from `shipping_method` to `fulfillment_method`; all values become null

Freshness Tests

Test F1: Most recent updated_at within SLA window

- Check: max(`updated_at`) is within the last 90 minutes
- Why: detects pipeline failures that leave the table stale
- Failure mode: Airflow scheduler misses a trigger due to resource contention; table goes 4 hours without update; revenue dashboard shows yesterday's numbers

Test F2: Today's orders are present

- Check: at least one row exists with `order_date` within the current calendar day
- Why: freshness on `updated_at` can pass even if today's orders are not yet loaded
- Failure mode: incremental load logic has an off-by-one error in the date filter, skipping today's partition

Referential Integrity Tests

Test R1: All customer_ids exist in customers table

- Check: zero rows in `orders` where `customer_id` does not exist in `customers.customer_id`
- Why: orphaned orders break customer-level reporting and LTV models

- Failure mode: customers table is loaded from a different source system that runs on a different schedule; orders for new customers arrive before the customer record is loaded

Test R2: All order_ids in order_line_items exist in orders

- Check: zero rows in `order_line_items` where `order_id` does not exist in `orders.order_id`
- Why: orphaned line items produce incorrect revenue calculations
- Failure mode: partial pipeline failure writes line items but not the parent order record

Business Rule Tests

Test B1: Delivered orders have a delivered_date

- Check: zero rows where `order_status = delivered` and `delivered_date` is null
- Why: null delivered dates for delivered orders break delivery time analytics and SLA reporting
- Failure mode: fulfillment system update changes the status transition logic; delivered status is set before the timestamp is written

Test B2: shipped_date is not before order_date

- Check: zero rows where `shipped_date < order_date`
- Why: impossible date ordering indicates a timezone handling error or a data entry bug
- Failure mode: order_date is stored in UTC, shipped_date is stored in local time, and the conversion is applied incorrectly

Test B3: Cancelled orders have zero or negative revenue impact

- Check: orders with `order_status = cancelled` are excluded from net revenue calculations
- Why: including cancelled orders in revenue figures causes overstating of revenue
- Failure mode: cancellation status update arrives out of order; cancelled orders are counted in the day's revenue before the cancellation record is processed

Full Test Suite Summary

Test ID	Type	Column(s)	What It Catches	Run Frequency
C1	Cardinality	All rows	Partial writes, pipeline failures	Hourly
C2	Cardinality	order_id	Duplicate records	Hourly
V1	Validity	order_id	Null primary keys	Hourly
V2	Validity	customer_id	Null foreign keys	Hourly
V3	Validity	order_amount	Range violations	Hourly
V4	Validity	order_status	Invalid categorical values	Hourly
V5	Validity	shipping_method	Column renames, null injection	Hourly
F1	Freshness	updated_at	Pipeline staleness	Hourly

Test ID	Type	Column(s)	What It Catches	Run Frequency
F2	Freshness	order_date	Missing current-day data	Hourly
R1	Referential integrity	customer_id	Orphaned orders	Hourly
R2	Referential integrity	order_id	Orphaned line items	Hourly
B1	Business rule	order_status, delivered_date	Timestamp logic errors	Hourly
B2	Business rule	order_date, shipped_date	Date ordering violations	Hourly
B3	Business rule	order_status, revenue	Cancelled order inclusion	Hourly

This is the complete test suite for a single Tier 1 table. It is not excessive. Each test catches a specific, documented class of failure. The total runtime for all 14 tests against a table of 50 million rows is typically under 2 minutes in a modern cloud warehouse.

Scaling Coverage Across the Warehouse

The orders table example is the template. Apply it to every Tier 1 table. For Tier 2 tables, implement cardinality, freshness, and the top three validity checks (nulls on required fields and the most critical range check). For Tier 3 tables, implement cardinality and freshness only.

Where **Should** Test Coverage Effort Focus?

		Production Testing		
		'Avoid the Fact'	'After the Fact'	'Before the Fact'
Dev. Testing	Hope	0%		
	Static Analysis		20%	
	Dynamic Testing			80%

Where Data Teams **Actually** Focus

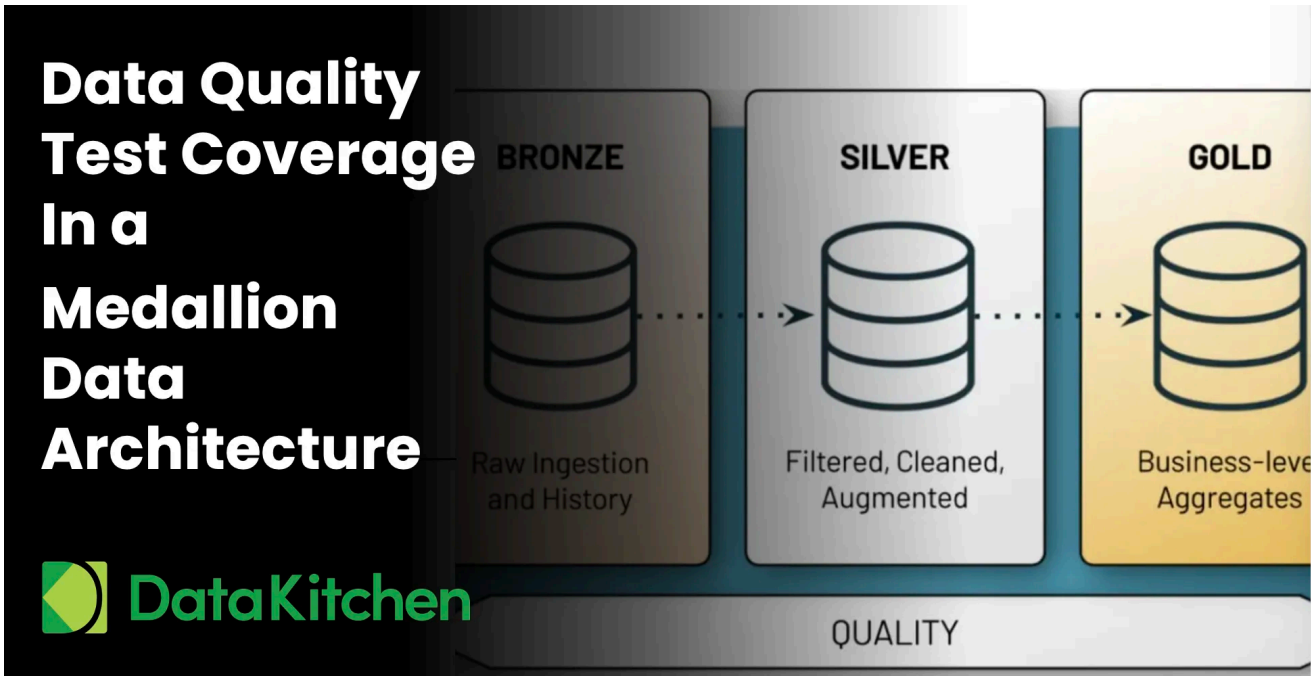
		Production Testing		
		'Avoid the Fact'	'After the Fact'	'Before the Fact'
Dev. Testing	Hope	80%		
	Static Analysis		20%	
	Dynamic Testing			0%

A warehouse with 200 tables might have 20 Tier 1 tables, 60 Tier 2 tables, and 120 Tier 3 tables. That produces roughly 280 Tier 1 tests, 240 Tier 2 tests, and 240 Tier 3 tests, for a total of around 760 tests. That is a manageable portfolio, and it can be built incrementally over several months by starting with the highest-criticality tables first.

The data team lead's job is to ensure that coverage matches risk, that SLAs are enforced by automated checks rather than manual review, and that every failure triggers an alert to a named owner who is accountable for resolution. Test coverage without accountability is documentation. Test coverage with accountability is reliability.

Data Quality Test Coverage in a Medallion Data Architecture

Read the original article at: https://datakitchen.io/blog/data-quality-test-coverage-in-a-medallion-data-architecture/



Data quality test coverage has become one of the most critical challenges facing modern data engineering teams, particularly as organizations adopt the increasingly popular Medallion data architecture. While this multi-layered approach to data processing offers significant advantages in organizing and refining data, it also introduces complexity that demands rigorous testing strategies to ensure data integrity across all layers.

Understanding the Medallion Data Architecture

The Medallion architecture represents a structured approach to data processing, organizing data into distinct layers that serve specific purposes in the data transformation journey. This architecture has gained tremendous popularity in data lakehouse implementations, where organizations need to balance raw data storage with analytical readiness.

The Bronze layer serves as the initial landing zone for all incoming raw data, capturing it in its unprocessed, original form. This layer maintains the complete fidelity of source data, preserving every detail that might be needed for future analysis or compliance requirements. Data engineers often refer to this as the “source of truth” layer, where data remains unchanged from its original state.

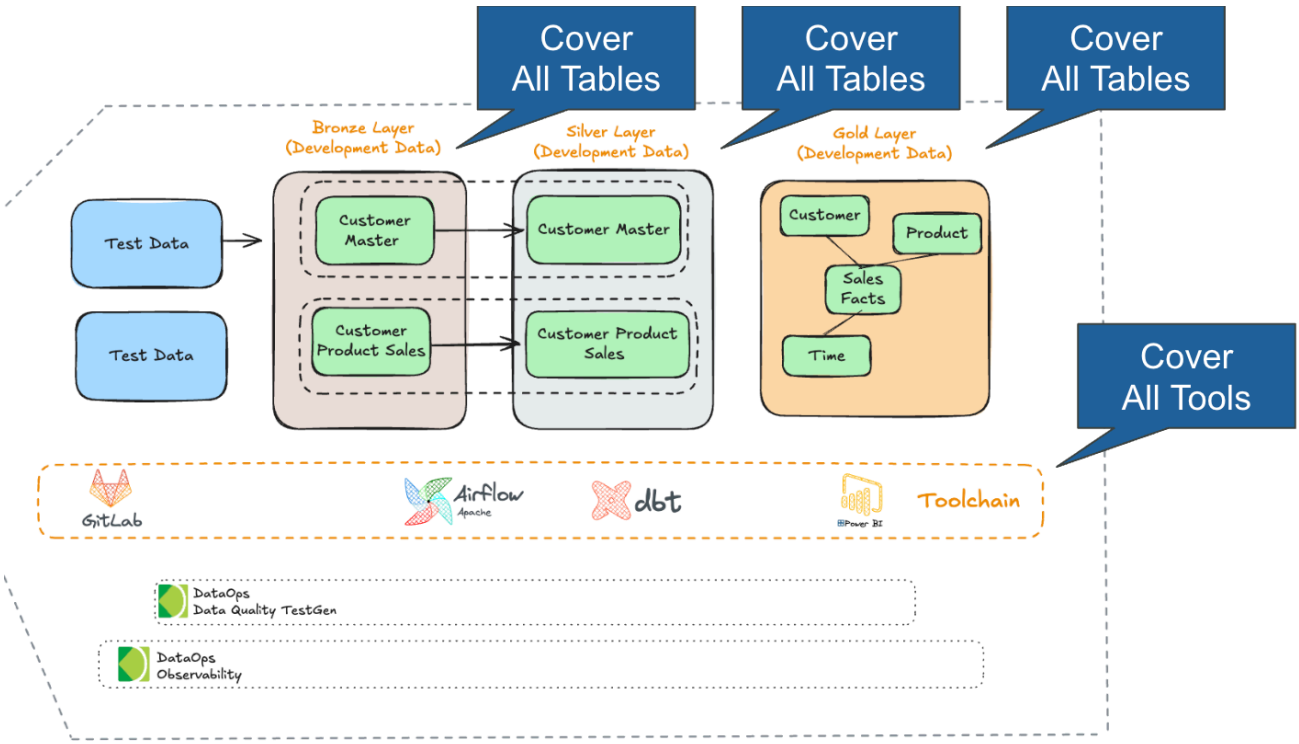
Moving up the architecture, the Silver layer represents the processing stage where data undergoes “just enough” cleaning and transformation to provide a unified, enterprise-wide view of core business entities. This layer focuses on standardization, deduplication, and basic quality checks while maintaining the granular detail needed for various downstream applications. The Silver layer essentially creates a cleansed, normalized version of the Bronze data without heavy aggregation.

Finally, the Gold layer represents the pinnacle of the Medallion architecture, housing fully refined, aggregated, and analysis-ready data. Data in this layer is typically organized into project-specific schemas optimized for business intelligence and advanced analytics. The Gold layer delivers the final data products that business users interact with through dashboards, reports, and analytical applications.

Some organizations extend this pattern with a Platinum layer beyond Gold, while others prefer the simplicity of L1, L2, and L3 designations. Regardless of naming conventions, the fundamental challenge remains consistent across all implementations: data quality becomes increasingly complex as data moves through these layers, making comprehensive test coverage essential for maintaining trust and reliability.

The Critical Need for Test Coverage

Test coverage in data engineering shares fundamental principles with software testing but addresses unique challenges inherent to data systems. Just as software teams would never consider releasing partially tested code, data teams must adopt equally rigorous standards for their data products. The consequences of insufficient testing in data systems can be far-reaching, affecting business decisions, regulatory compliance, and organizational trust.



The primary motivation for comprehensive test coverage centers on two fundamental concerns that every data engineering team faces: preventing production breaks when deploying changes and avoiding the embarrassment of customers discovering data quality issues before internal teams do. These scenarios represent not just technical failures but organizational credibility challenges that can have lasting impacts on stakeholder confidence.

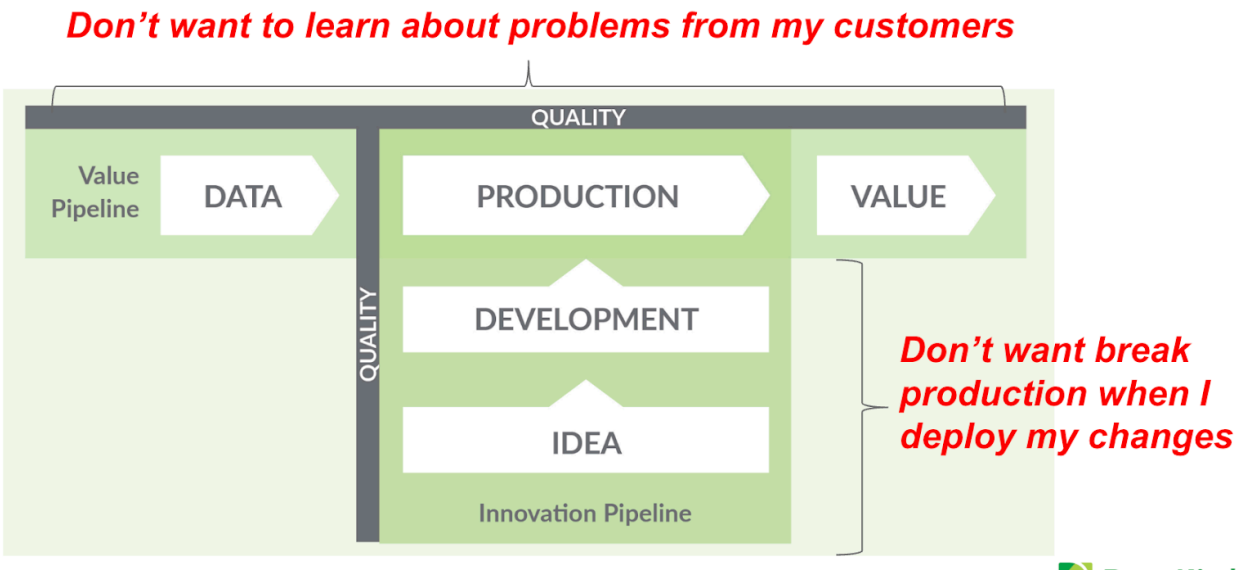
Data test coverage refers to the extent to which automated quality checks cover your data itself, data pipelines, transformations, and outputs. The higher your coverage, the more confident you can be that the data you produce is accurate, reliable, and trustworthy. Many organizations mistakenly assume that occasional spot checks or ad-hoc validations provide sufficient protection against data quality issues. Unfortunately, manual, partial testing often leads to incomplete visibility and unexpected problems that surface at the worst possible moments.

The analogy to software development proves particularly relevant here. Imagine releasing software that has only been partially tested, no development team would accept such risk. The same standards should apply to data systems, where incomplete testing can lead to faulty business decisions, compliance violations, and eroded trust in data-driven insights.

Strategic Placement of Test Coverage

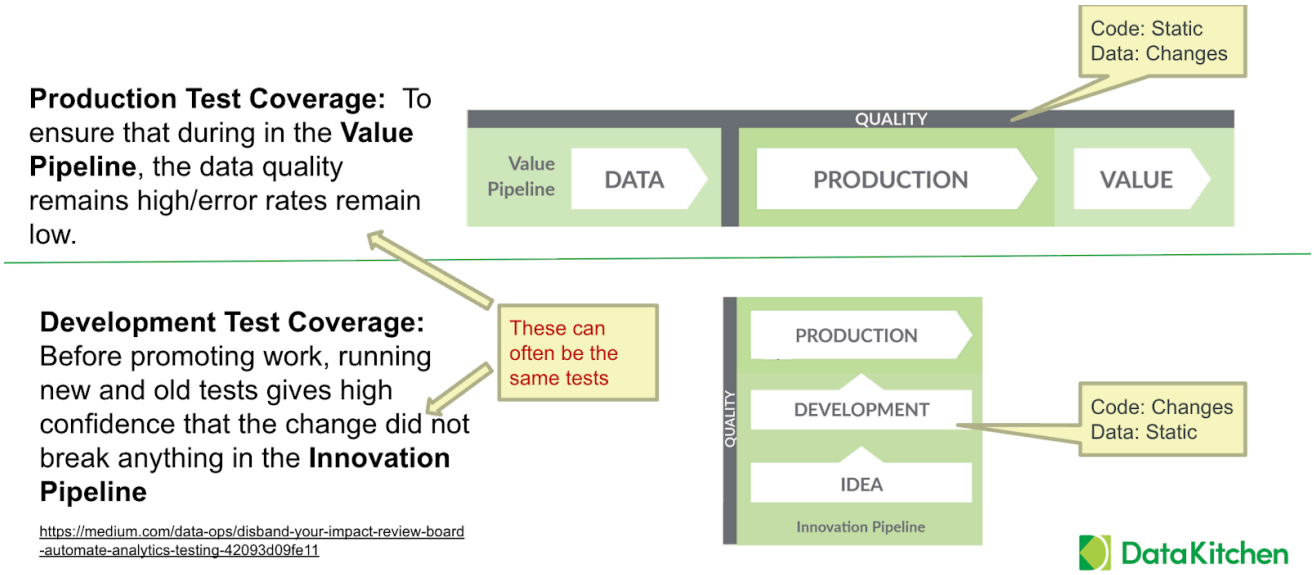
Effective test coverage in a Medallion architecture requires understanding both where and when to implement testing strategies. The placement of tests follows two primary patterns: development test coverage and production test coverage, each serving distinct but complementary purposes in maintaining data quality.

Development test coverage operates within what data engineers refer to as the Innovation Pipeline, where new features, transformations, and data sources are developed and validated before being promoted to production. During this phase, both new and existing tests run against changes to provide high confidence that modifications haven’t broken existing functionality. This represents a critical checkpoint where code changes while data remains static, allowing teams to validate transformations against known datasets.



Production test coverage functions within the Value Pipeline, ensuring that data quality remains high and error rates stay low during active data processing. In this environment, code remains static while data constantly changes, requiring different testing approaches that focus on data consistency, freshness, and adherence to expected patterns. These production tests often mirror development tests but operate against live, flowing data rather than static test datasets.

The beauty of this dual approach lies in the fact that development and production tests can often be identical, providing consistency across environments while addressing the different dynamics of static versus flowing data. This consistency enables teams to build confidence in their testing strategies and ensures that validation logic remains synchronized across the entire data lifecycle.



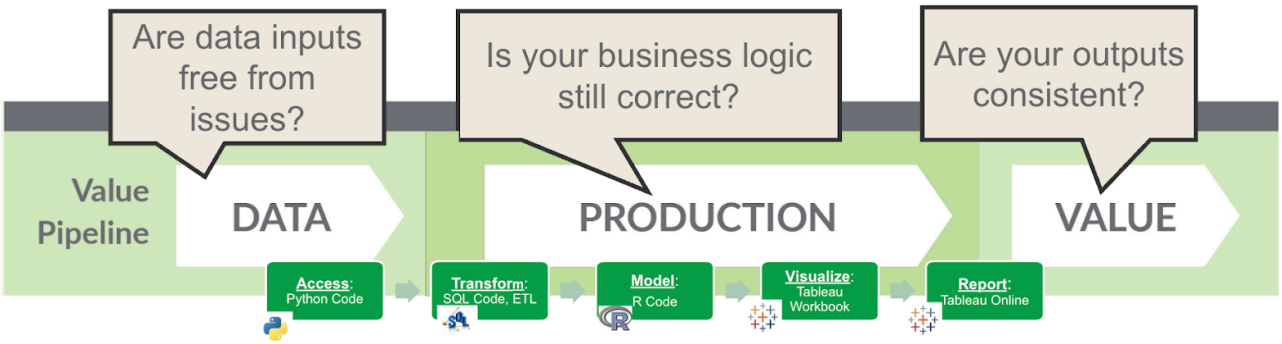
Comprehensive test coverage must extend beyond just data validation to encompass every step and tool in your value pipeline. Whether you're working with Python code for data access, SQL transformations in ETL processes, R code for modeling, Tableau workbooks for visualization, or online reporting platforms, each component requires specific validation approaches. The goal is to ensure that data inputs remain free from issues, business logic continues operating correctly, and outputs maintain consistency across all downstream applications.

Implementing Shift Left and Shift Down Testing

Modern data engineering practices emphasize two complementary testing philosophies that significantly improve data quality while reducing costs and complexity. Shift Left testing involves executing data quality tests earlier in the data pipeline lifecycle, ensuring that issues are identified and resolved before they propagate downstream. This approach recognizes that early detection dramatically reduces the effort and cost required to address data quality problems.

Shift Down testing embraces integration and end-to-end testing methodologies, validating not just isolated components but also the interactions and cumulative impacts within the entire data ecosystem. Rather than testing individual transformations in isolation, Shift Down approaches validate the complete data journey from source to consumption, ensuring that the entire system functions cohesively.

Test Every Step And Every Tool in Your Value Pipeline



The economic benefits of these approaches become clear when considering the cost implications of data quality issues discovered at different stages. The industry-standard 1:10:100 rule, developed by George Labovitz and Yu Sang Chang in 1992, demonstrates that the cost of preventing poor data quality at the source equals approximately one dollar per record, while remediation after creation costs ten dollars per record, and the cost of failure, doing nothing until customers discover issues, reaches one hundred dollars per record.

This cost structure makes a compelling case for comprehensive testing at the earliest possible stages. It becomes ten times cheaper to find problems in data at the source or raw layer compared to discovering them in integrated data or tool-modified outputs. The financial impact extends beyond immediate

remediation costs to include reputation damage, regulatory penalties, and the opportunity cost of incorrect business decisions based on faulty data.

Measuring Test Coverage in Medallion Architecture

Effective measurement of test coverage requires systematic approaches that ensure comprehensive validation across all layers and components of your Medallion architecture. The measurement framework operates on multiple levels, each targeting different aspects of data quality and system reliability.

At the foundational level, every table within your architecture should have dedicated tests. This includes tables in Bronze, Silver, and Gold layers, ensuring that no data repository operates without quality validation. The minimum standard calls for at least two tests per table, typically focusing on consistency checks that validate volume, freshness, and schema adherence.

Diving deeper, every column in every table requires its own testing regime. Column-level testing extends beyond simple existence checks to validate data types, value ranges, null constraints, and adherence to patterns. The recommended minimum of two tests per column ensures that both basic integrity and business-specific validation occur for each data element.

- Every Table Should Have Tests: Minimum 2 Tests Per Table
- Every Column In Every Table Should Have Tests: Minimum 2 Tests Per Column
- Every Significant Business Metric Should Have Tests: Minimum 1 Custom Test Per Metric

- Minimum 1 Check Per Tool: Logs For Errors
- Optional: Check Metrics
- Minimum 1 Check Per Tool Per Job: Task Status and Substatus Results
- Minimum 1 Check Per Tool Per Job: Timing and Duration

Business-specific metrics represent another crucial dimension of test coverage. Every significant business or domain-specific metric should have custom tests that validate not just technical correctness but business logic accuracy. These tests often require in-depth domain knowledge and represent the highest-value validations in terms of business impact.

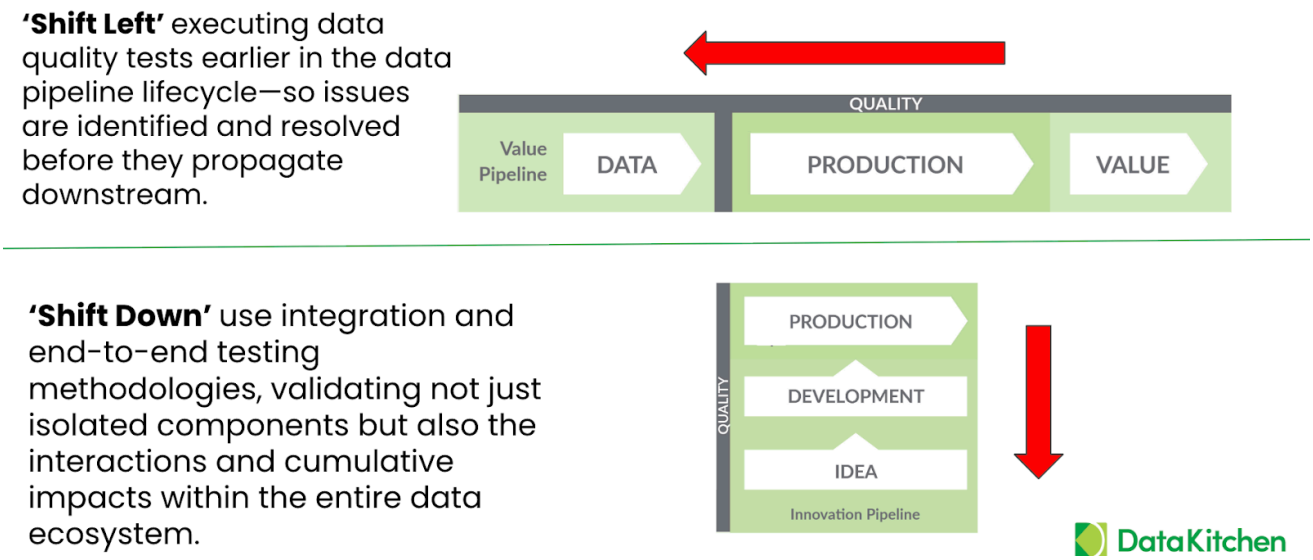
Treat data quality tests like first-class citizens. Tests aren’t just QA - they’re living proof of what the system should do. And by ‘tests,’ we mean numerous tests , hundreds, thousands, covering every table.

Tool-level testing completes the coverage framework by ensuring that every tool in your data processing pipeline operates correctly. This includes checking logs for errors, monitoring performance metrics, validating task status and substatus results, and tracking timing and duration patterns. Each tool should have at least one error check, with additional monitoring for metrics, status validation, and performance tracking.

Practical Test Coverage Implementation

To illustrate the scale of comprehensive test coverage, consider a typical Medallion architecture implementation with three database levels (L1, L2, L3), three primary tools (StreamSets, Airflow, dbt), and five distinct jobs or workflows requiring monitoring.

In the L1 Bronze layer, assuming 100 tables with an average of 10 columns each, the minimum testing requirement includes 200 table-level tests and 2,000 column-level tests. The L2 Silver layer, with similar table and column counts, requires another 200 table tests and 2,000 column tests. The L3 Gold layer, which contains fewer tables (perhaps 10) with more columns (averaging 30), requires 20 table tests and 600 column tests, in addition to domain-specific business metric tests.



Tool-level monitoring adds another layer of validation, with three tools requiring error checking and six jobs needing 12 separate checks for various problems, including status validation and performance monitoring. This example demonstrates how comprehensive test coverage quickly scales to thousands of individual tests across a moderately complex Medallion implementation.

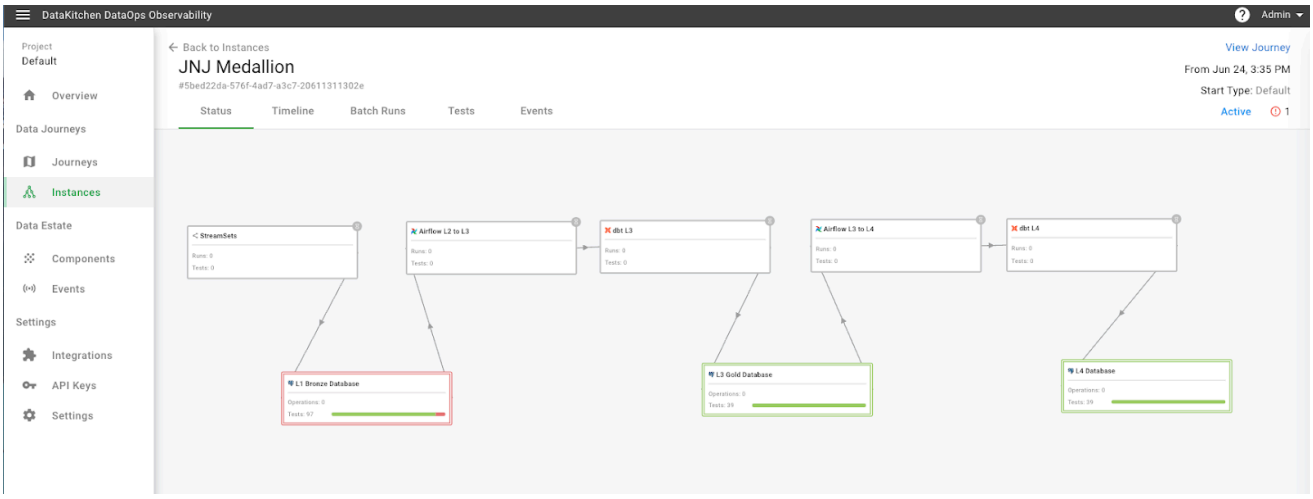
The medallion architecture above has three database levels (l1, l2, l3), three tools (streamsets, airflow, dbt), and five jobs/workflows to monitor

Gold/L3: 10 tables + 30 Columns: Minimum 20 Table Tests and 600 Column Tests, plus a handful of Domain Specific Business Metric Tests

Manual Writing Data Quality Test Is Time Consuming

The economic argument for comprehensive test coverage becomes compelling when considering the actual cost of data quality issues. Every problem discovered earlier in the development and processing pipeline costs significantly less to resolve than issues found in production or by customers. Tests represent gifts that data engineers give to their future selves, providing early warning systems that prevent costly downstream problems.

Ensuring coverage for every table and tool creates a comprehensive safety net that protects both data quality and organizational resources. However, manually writing (or configuring) those tests can be time-consuming. Looking at the example below, it could take months to manually code all the necessary tests.



Open-source tools for automated test generation make comprehensive coverage economically feasible, even for resource-constrained teams. Rather than requiring massive manual investment in test creation, automated tools enable junior operators to generate thousands of tests with minimal effort, democratizing access to enterprise-grade data quality assurance. This is why we created open-source DataOps Data Quality TestGen.

Test Coverage + Data Journeys Provide Precision.

Understanding the distinction between data lineage and data journeys is crucial for implementing adequate test coverage. Data lineage provides valuable insights into where problems might spread throughout your system, much like a building blueprint shows potential paths for fire spread. However, data journeys with comprehensive test coverage reveal where problems are actually occurring, much like a fire alarm control panel indicating active emergencies.

Data lineage represents educated guesses about impact, showing dependencies and relationships between data elements. While valuable for understanding system architecture, lineage alone cannot definitively identify which components are experiencing actual problems. Complete test coverage, as displayed through data journeys, provides clarity on the real-time health of the system and the status of data quality.

An example

- With TestGen, a junior operator can generate 50,20 tests with **two steps** (profile, generate tests)
- It would take a trained Data Engineer **14 months** to achieve the same results – with no time for meetings, breaks, or vacations

Manual Test Development Cost		
Number of Tests	5,020	
Minutes Per Test	30	
Hours	2,510	
Days	314	8 hours/day
Weeks	63	5 days/week
Months	**14**	4.35 weeks/month
TestGen	**minutes**	

The practical difference becomes apparent during incident response. Data lineage might indicate that an upstream issue could impact five tables, but comprehensive test coverage reveals that only one table actually contains errors. This precision enables faster resolution and prevents unnecessary investigation of unaffected components.

Conclusion and Future Considerations

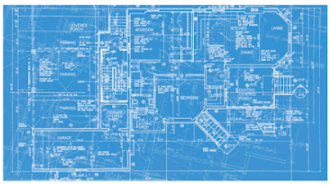
Comprehensive test coverage in Medallion data architecture represents more than just technical best practice, it embodies a commitment to data reliability that enables organizational trust and effective decision-making. Having customers uncover data errors creates embarrassment and potential damage that rigorous quality testing can prevent.

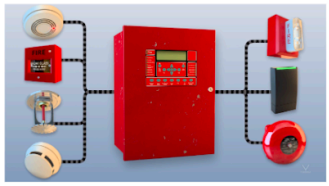
Quality testing at every stage of the Medallion architecture, Bronze, Silver, and Gold, catches issues early while minimizing the risk of data inconsistencies surfacing in customer-facing insights. Prioritizing thorough automated testing and continuous monitoring enables data teams to uphold data integrity and foster trust across the organization.

> ***The future of data engineering depends on treating data quality, at every level, with the same rigor applied to software quality.***

Automatically testing both the data and the tools that act upon data creates comprehensive protection against quality degradation. Since nobody has time to write the thousands of tests required for comprehensive coverage manually, open-source enablement becomes crucial for the widespread adoption of best practices.

When a Fireman walks into your building, which is more important? A *Blueprint* or a *Fire Alarm Control Panel*?





Data Lineage Shows Where Fire ***Might*** Spread—Data Journeys With Tests Show Where It's ***Actually*** Burning.

Data Lineage Is *Guess* On The Impact, Complete Test Coverage Shown In A Data Journey Is *Clarity*

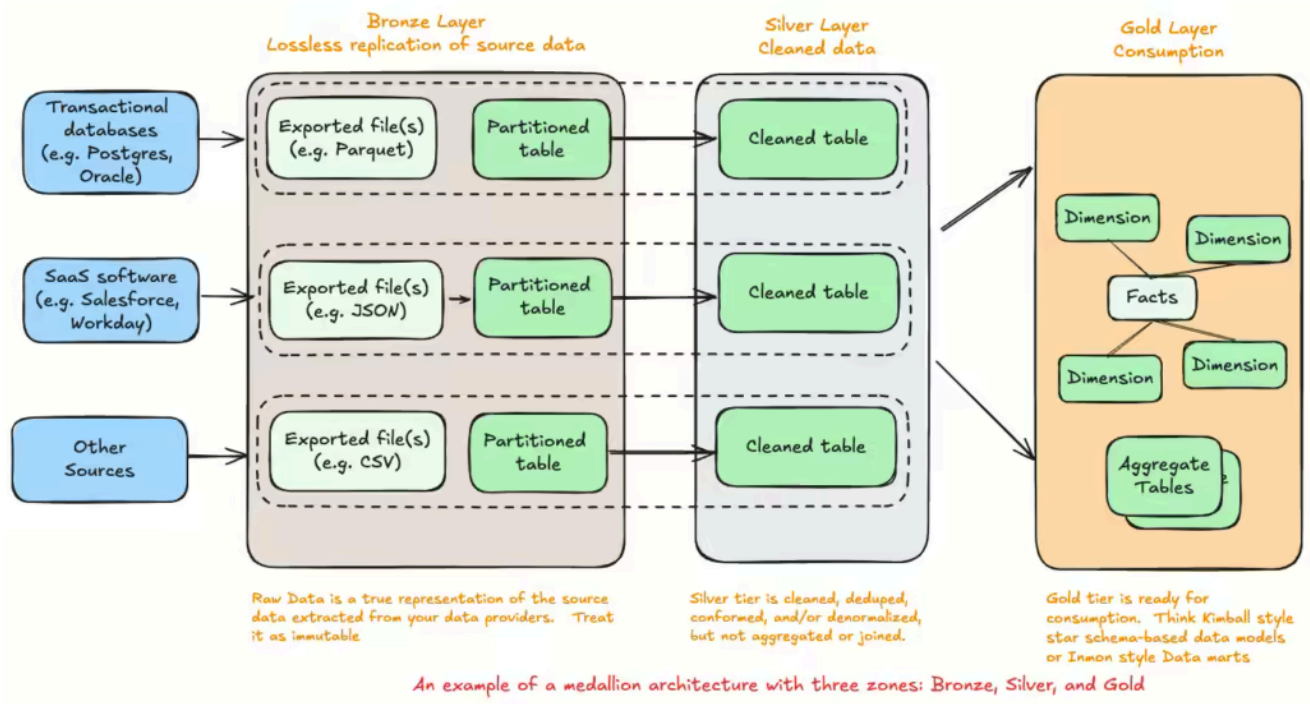
Data Lineage *May* Show **Five Tables Impacted**: But **Only <u>One</u> Has An Error**.

Test coverage and quality dashboards foster organizational best practices by providing visibility into data health and enabling proactive quality management. As Medallion architectures continue to gain popularity in data lakehouse implementations, teams that master comprehensive test coverage will deliver more reliable data products and maintain competitive advantages through superior data quality assurance.

The investment in comprehensive test coverage pays dividends through reduced firefighting, improved stakeholder confidence, and more reliable, data-driven decision-making. For data engineering teams serious about delivering production-grade data products, implementing systematic test coverage across their Medallion architecture represents not only a technical improvement but a fundamental shift toward sustainable and trustworthy data operations.

The Race for Data Quality in a Medallion Architecture

Read the original article at: https://datakitchen.io/blog/the-race-for-data-quality-in-a-medallion-architecture/



An example of a medallion architecture with three zones: Bronze, Silver, and Gold

In the Olympic quest for data quality, the Medallion Data Lakehouse Architecture is the ultimate marathon, where data teams strive to go beyond the Bronze and Silver to take the Gold. Each stage represents a new data accuracy and quality challenge; the Medallion architecture combines raw, cleansed, and analysis-ready data in a powerful three-tier system. However, ensuring data quality at every layer is a challenging feat. Just as elite athletes rely on rigorous training and constant feedback to reach peak performance, data teams need a flexible, iterative approach to spot and address quality issues early. Adopting a DataOps mindset transforms data quality into a dynamic, continuous process, allowing teams to improve incrementally through feedback loops and adapting to issues. This approach empowers individuals across the organization to champion data quality, enabling a fast-paced yet precise journey toward excellence. In this data quality race, DataOps makes data quality testing a natural, built-in practice, laying a foundation for constant refinement and ensuring that data teams remain on track to win Gold in accuracy, reliability, and organizational trust.

It sounds great, but how do you prove the data is correct at each layer? How do you ensure data quality in every layer? How do you get both speed of execution and accuracy of testing? Read more to find out!

Bronze, Silver, and Gold - The Data Architecture Olympics?

The Medallion architecture pattern is gaining traction among data teams. It is a layered approach to managing and transforming data. The Medallion architecture is a design pattern that helps data teams organize data processing and storage into three distinct layers, often called Bronze, Silver, and Gold. By systematically moving data through these layers, the Medallion architecture enhances the data structure in a data lakehouse environment. This architecture is valuable for organizations dealing with large volumes of diverse data sources, where maintaining accuracy and accessibility at every stage is a priority.

The Bronze layer is the initial landing zone for all incoming raw data, capturing it in its unprocessed, original form. This foundational layer is a repository for various data types, from transaction logs and sensor data to social media feeds and system logs. By storing data in its native state in cloud storage solutions such as AWS S3, Google Cloud Storage, or Azure ADLS, the Bronze layer preserves the full fidelity of the data. This raw, historical archive is essential for traceability and allows data engineers to load new data into the lakehouse incrementally through batch uploads or real-time streaming. While the Bronze layer doesn't offer the cleanliness or standardization necessary for analysis, it is a critical foundation upon which the Silver and Gold layers are built. Bronze layers can also be the raw database tables. Bronze layers should be immutable.

Next, data is processed in the Silver layer, which undergoes "just enough" cleaning and transformation to provide a unified, enterprise-wide view of core business entities. This stage involves validation, deduplication, and merging of data from different sources, ensuring that the data is in a more consistent and reliable format. For instance, records may be cleaned up to create unique, non-duplicated transaction logs, master customer records, and cross-reference tables. The Silver layer aims to create a structured, validated data source that multiple organizations can access. This intermediate layer strikes a balance by refining data enough to be useful for general analytics and reporting while still retaining flexibility for further transformations in the Gold layer. Expect the schema to be similar but not the same as the Bronze layer. Alternatively, the Data Vault methodology can be applied to the Silver layer, where data is transformed into Hubs, links, and satellites.

Finally, the Gold layer represents the pinnacle of the Medallion architecture, housing fully refined, aggregated, and analysis-ready data. Data is typically organized into project-specific schemas optimized for business intelligence (BI) applications, advanced analytics, and machine learning. Whether it's customer analytics, product quality assessments, or inventory insights, the Gold layer is tailored to support specific analytical use cases. This layer is often characterized by de-normalized, read-optimized tables that minimize the need for joins and are structured according to data warehousing methodologies, such as Kimball's star schema or Inmon's data marts. When data reaches the Gold layer, it is highly curated and structured, offering a single version of the truth for decision-makers across the organization.

We have also seen a fourth layer, the Platinum layer, in companies' proposals that extend the Data pipeline to OneLake and Microsoft Fabric. We recently talked with one company whose Data Journey is a path from SAP & Teradata to ADLS (Bronze/Silver/Gold) and finally to Synapse for usage.

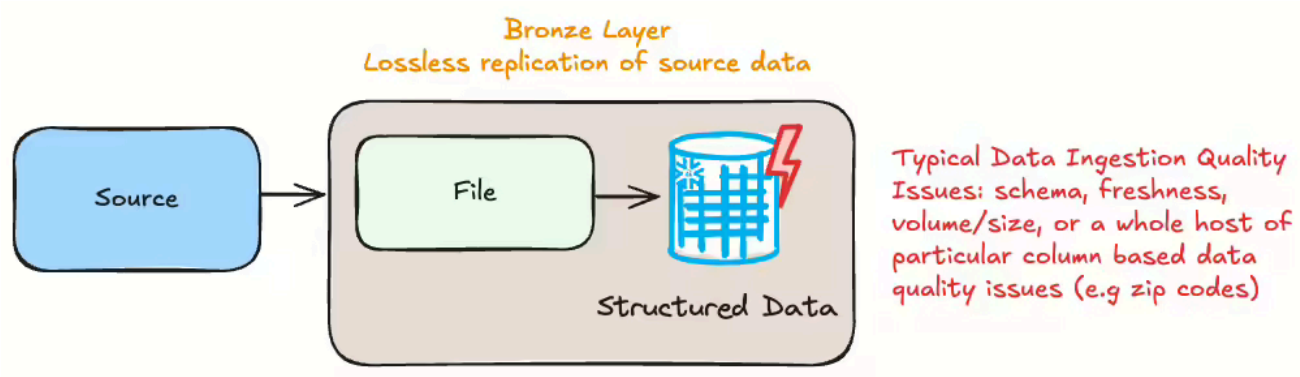
The Medallion architecture offers several benefits, making it an attractive choice for data engineering teams. By preserving raw data in the Bronze layer, organizations have a historical archive they can revert to, ensuring transparency and traceability. The structured flow through Silver to Gold allows flexibility in handling new transformations and aggregations. At the same time, the Gold layer's "single version of the truth" makes data accessible and reliable for reporting and analytics. However, this architecture is not without its challenges. The need to copy data across layers, manage different schemas, and address data latency issues can complicate data pipelines. For businesses requiring near-real-time insights, the time taken to traverse multiple layers may also introduce delays.

Moreover, implementing Medallion architecture involves sophisticated configuration and careful management of changes across all layers, which can complicate day-to-day operations. Finally, the challenge we are addressing in this document - is how to prove the data is correct at each layer.? How do you ensure data quality in every layer?

The Medallion architecture is a framework that allows data engineers to build organized and analysis-ready datasets in a lakehouse environment. By methodically processing data through Bronze, Silver, and Gold layers, this approach supports a variety of use cases. It provides a structured path from raw data to actionable insights, although it requires careful planning and consideration of specific business needs and data quality.

Data Quality Challenges That Cause You To Lose The Medallion Race

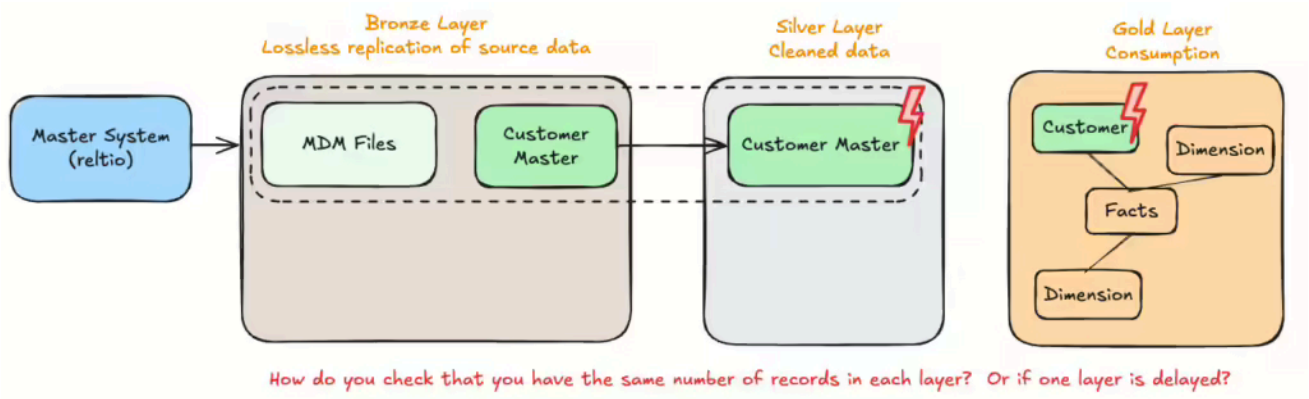
In a Medallion architecture, while the structured progression of data through Bronze, Silver, and Gold layers is designed to enhance organization, numerous data quality challenges can still arise, particularly once the architecture is in production. These "Day 2" production quality assurance (QA) issues can disrupt customer satisfaction, introduce inaccuracies, and ultimately undermine the value of analytics. Data engineering teams must understand these problems and incorporate mechanisms to detect and mitigate them. Key challenges include identifying the problem type and finding each issue's root cause and layer before your customer finds it for you and your team!



One common problem in production environments is when data fails to load correctly into one or more layers of the architecture, causing interruptions across the data pipeline. For instance, suppose a new dataset from an IoT device is meant to be ingested daily into the Bronze layer. Still, due to connectivity issues or file format mismatches, the load fails. This failure can cascade to the Silver and Gold layers,

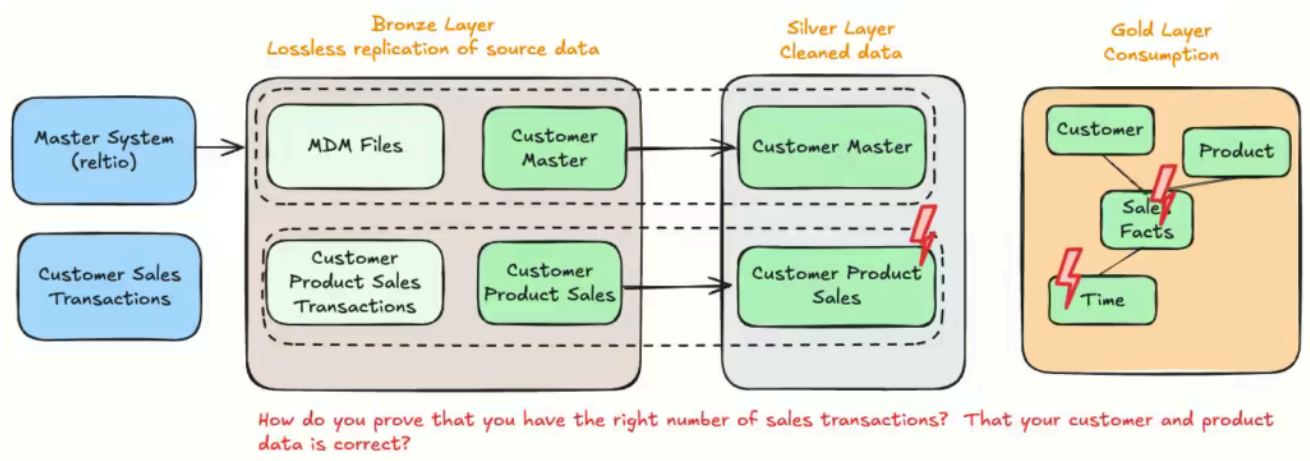
where downstream dependencies on the raw data prevent those layers from updating accurately. Such issues often go unnoticed until a user or analyst reports missing information in a dashboard or report, by which point the delay has already impacted business decision-making. You have typical data ingestion layer challenges in the bronze layer: lack of sufficient rows, delays, changes in schema, or more detailed structural/quality problems in the data.

Data missing or incomplete at various stages is another critical quality issue in the Medallion architecture. In a Silver layer, for example, merging and deduplication processes rely on complete records to create clean datasets for analysis. Suppose a batch from a transactional system only loads partial records into the Bronze layer due to truncation or data source issues. In that case, the Silver layer can inadvertently generate incomplete or inaccurate views. For instance, a table that shows customer purchase histories could display partial transaction data, leading analysts to underestimate sales or misinterpret customer behavior. Similarly, downstream business metrics in the Gold layer may appear skewed due to missing segments, which can impact high-stakes decisions. Routine checks for row counts, null values, and integrity constraints can help detect missing data early in the pipeline.



Another common issue is data arriving late, which is particularly challenging in real-time or near-real-time systems. When data from various sources does not reach the Bronze layer on time, it can lead to stale insights and missed opportunities in the Gold layer, especially for time-sensitive applications like inventory tracking or marketing campaigns. For example, if data about online customer interactions is delayed due to source system lags, the Gold layer's customer segmentation analysis may fail to reflect recent behavior, leading to irrelevant or poorly targeted campaigns. Late data can also create mismatches in time-based aggregations and reporting, where analytics teams might unknowingly operate on outdated information.

Detailed Data Quality issues are another recurring challenge in the Medallion architecture. In the Silver layer, the transformation processes required to standardize, deduplicate, and validate data can inadvertently introduce new quality problems, such as data type mismatches, improper joins, or invalid replacements. When moving data from the cleaned and enriched Silver layer to the ready-for-consumption Gold layer. An operation to merge customer data across multiple sources might incorrectly aggregate records due to mismatched keys, leading to inflated or deflated metrics in the Silver layer. Issues can propagate unnoticed to the Gold layer, where they can alter the final analytical outputs.



In production environments, these issues require proactive data quality monitoring and quick response capabilities to ensure that data flowing through the Medallion architecture remains accurate, timely, and complete. By incorporating automated alerting systems, regular QA checks, and well-defined SLAs for data loading, data engineering teams can better manage Day 2 production challenges, helping to preserve the integrity and reliability of their analytical outputs.

Get Off The Blocks Fast: Data Quality In The Bronze Layer

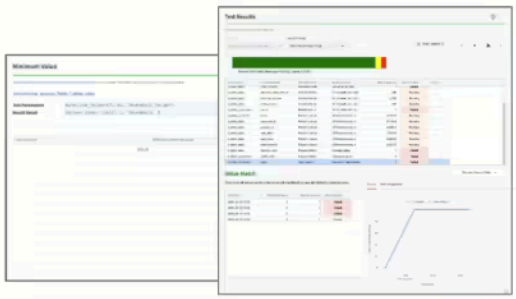
Effective Production QA techniques begin with rigorous automated testing at the Bronze layer, where raw data enters the lakehouse environment. The first part of Data Quality (DQ) training focuses on onboarding new datasets, which includes evaluating and cleansing them before they progress through the pipeline. This process involves detailed data profiling and column-by-column hygiene checks, which help detect hidden issues like null values, incorrect formats, and unexpected duplicates that can undermine data accuracy. During this phase, data teams must decide whether to patch problematic data, push back to data sources for corrections, or pass it through if it meets quality standards.

As data ingestion transitions to a continuous flow, the second part of DQ training equips engineers to monitor schema consistency, row counts, and data freshness, ensuring data integrity over time. This involves setting up automated, column-by-column quality tests to quickly identify deviations from expected values and catch emerging issues before they impact downstream layers. Four key areas for data quality check during continual data updates are:

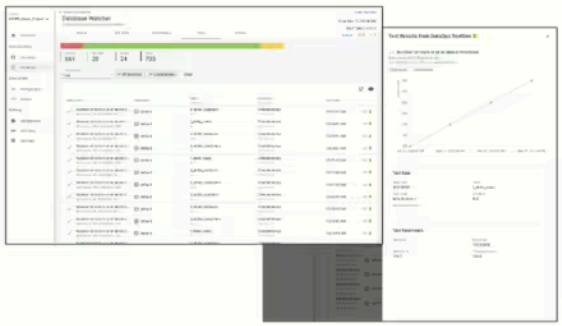
Bronze Layer QA: Continual Data Ingestion

	Use Case	When Does it Happen	Solution
Data Ingestion:	Updates to existing data sources; Data ingestion monitoring	Continually / Off Cycle	Find anomalies in data updates and notify the proper party in the right place.

DataOps DQ TestGen Data Freshness, Volume, Schema, and 32 Other Automatic Tests



DataOps Observability Anomaly Alerts

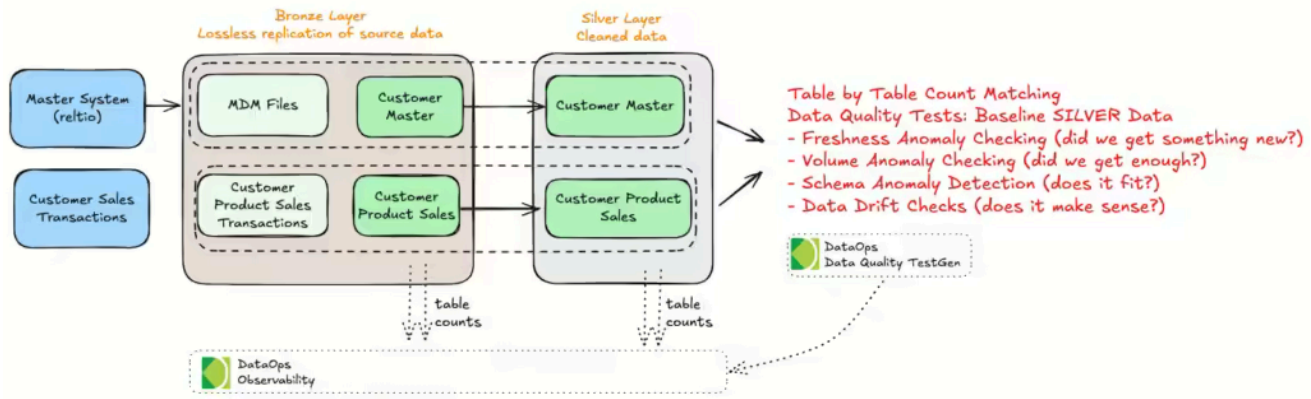


DataKitchen

- Freshness Anomaly Checking (did we get something new): Freshness data ingestion tests and anomaly checks help ensure that new data is consistently being ingested by verifying date-related metrics in a dataset. These tests include checking that recent dates are within expected ranges, distinct date counts have not dropped, and that date boundaries, such as future and minimum dates, align with the baseline data.
- Volume Anomaly Checking (did we get enough): Volume data ingestion tests and anomaly checks ensure that sufficient new data is being ingested by comparing current record counts and distributions against baseline expectations. These tests include verifying that record counts haven't dropped, are within an acceptable range, and that records exist consistently across daily, weekly, and monthly intervals.
- Schema Anomaly Detection (does it fit): Schema data ingestion tests and anomaly checks verify that incoming data aligns with expected formats and values defined by the baseline schema. These tests include validating that values match specified lists, patterns, or formats (such as email or addresses) and ensuring no truncation occurs in alphanumeric or decimal values.
- Data Drift Checks (does it make sense): Is there a shift in the overall data quality? These checks include identifying shifts in averages, detecting outliers, monitoring variability changes, and verifying that minimum values, missing data percentages, and uniqueness rates align with established baselines.

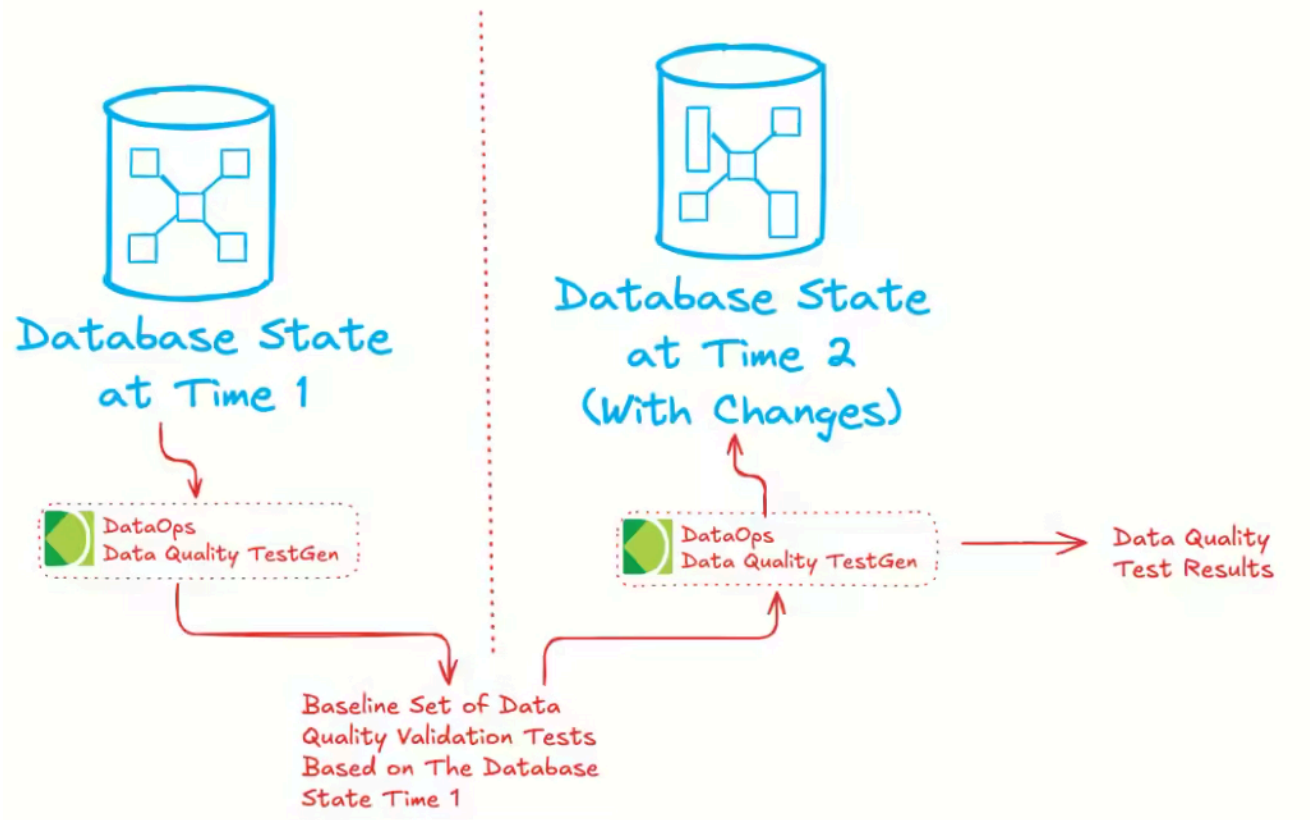
Run With Excellence: Data Quality in The Silver Layer

Testing the Silver layer in a Medallion architecture is crucial for maintaining high data quality, as this is the stage where raw data from the Bronze layer is cleansed, validated, and transformed into a standardized format for broader organizational use. Since this layer serves as a centralized source of trusted data, it's essential to perform comprehensive quality checks to ensure data consistency and reliability. One of the fundamental steps in validating Silver layer data is conducting table-by-table count matching. This involves checking that the number of records aligns with expectations and matches prior ingestion records, preventing discrepancies that could indicate data loss or duplication.



Beyond record counts, several vital data quality tests should be part of Silver layer validation. Freshness anomaly checks verify that the data is current by confirming recent updates or additions, ensuring that the Silver layer reflects the latest information from the Bronze layer. Volume anomaly checking ensures that enough data is being ingested, flagging instances where data might be missing due to issues in the upstream ingestion process. Schema anomaly detection helps verify that data structures align with expected formats, preventing schema mismatches that could cause issues during data processing. Finally, data drift checks examine whether the statistical properties of the data have significantly shifted compared to historical baselines, which can help identify unexpected changes or trends that may affect data accuracy or stability.

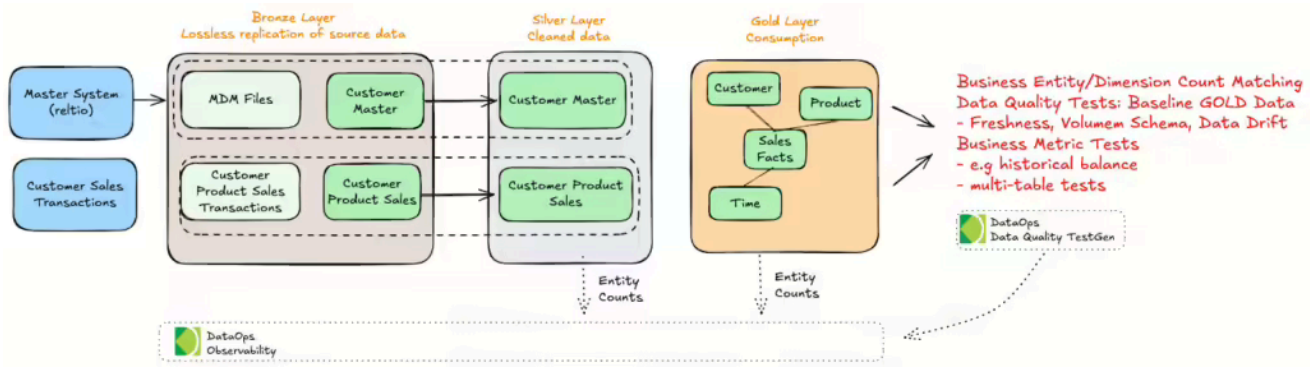
All these checks are similar to the Bronze layer, so you have two options. One option is to copy all these tests from the Bronze layer to the Silver layer and then run them in the Silver. This works if your schemas/ddl in each layer are precisely the same. Another option, if the Silver schema is not the same as Bronze, is to re-generate all your tests (using a tool like DataOps DQ TestGen) baselined in the Silver data. This creates a test set based on the silver schema, allowing you to catch data quality variance. Every time you receive an update of the data, running these checks can ensure the data is correct even if the schema differs slightly from the bronze layer.



When implementing these tests in the Silver layer, two main approaches can be taken depending on the level of control over the ingestion process. If you can modify or control the ingestion code, data quality tests, and validation checks should ideally be integrated directly into the process. Running these tests immediately upon data arrival ensures that issues are detected early, minimizing the impact on downstream processes. Alternatively, suppose you do not control the ingestion code. In that case, a practical approach is to set up periodic polling of the Silver layer database to run data quality tests and check for anomalies at scheduled intervals. Regular polling helps identify issues without direct integration into the ingestion code, providing a safeguard to maintain data quality across the Silver layer. This same choice works on any layer: Bronze, Silver or Gold.

Break The Ribbon With Gold Layer Data Quality

In the Gold layer of a Medallion architecture, data quality testing takes on a business-centric focus, as this layer is arranged explicitly in an analysis-ready schema design to support decision-making and reporting. Quality checks in the Gold layer ensure data consistency and integrity and validate that the data accurately reflects critical business metrics and entities. One of the primary testing methods involves count matching between the Gold and Silver layers for each essential business entity or dimension, such as the number of customers, transactions, or products. By verifying that the counts in the Gold layer align with those in the Silver layer, data quality teams can ensure that no data has been lost or inaccurately aggregated in the final transformation stages.



In addition to entity count matching, baseline data quality tests are performed to check for freshness, volume, schema consistency, and data drift in the Gold layer. The Gold layer is most likely to have a unique schema. So, like in the other layers, you need to baseline your test creation in that Gold schema. This creates a test set based on the Gold schema, allowing you to catch data quality variance. Running these checks every time you receive an update of the Gold data can ensure the data is correct.

Furthermore, business metric-specific tests are crucial in the Gold layer, as this layer often consolidates data across multiple tables and sources to produce metrics for business analysis. These tests validate that calculated metrics, such as the number of products sold per region, customer retention rates, or historical balances, are consistent with expectations and historical trends. Multi-table checks are also essential, verifying the consistency of relationships and aggregations across various tables and ensuring that complex business metrics are accurately represented.

Help The Referee With Data Quality Scores

Medallion Architecture and Data Quality Scoring introduce an approach to evaluate and improve data quality at each layer, Bronze, Silver, and Gold, enabling organizations to make data quality an actionable metric that drives better organizational outcomes. In a Medallion architecture, data progresses through these layers, with each stage representing a higher level of cleanliness, enrichment, and readiness for analysis. Data engineering and quality teams can continuously monitor, compare, and improve data quality by applying a scoring system to each layer. This ensures that the final, analysis-ready Gold layer meets the organization’s highest accuracy, completeness, and reliability standards.

In the Bronze layer, data quality scoring focuses on foundational aspects, as this layer contains raw, unprocessed data. Scores are typically based on attributes like completeness (e.g., missing values), freshness (e.g., data recency), and schema conformity (e.g., alignment with baseline structures). A high score at this stage indicates that raw data is being captured accurately and is ready for cleansing and transformation in the Silver layer. Low scores here can highlight data source issues, prompting corrective actions upstream to ensure better data quality early in the pipeline. Comparing Silver layer scores with Bronze helps identify the efficiency of cleansing and transformation processes. If Silver scores do not significantly improve over Bronze, it may indicate gaps in data processing, pointing to areas where further data enrichment or error-handling routines are needed.

In the Gold layer, where data is fully prepared for analysis, quality scores primarily focus on business relevance, metrics accuracy, and decision-making readiness. Scoring in this layer includes checks on aggregated metrics, consistency in multi-table relationships, and stability of key business indicators like sales performance or customer segments. Since this layer is closest to end-users, a high score in the Gold layer is critical for building organizational trust in data-driven insights. By comparing Gold layer scores with Silver, teams can evaluate whether transformation processes have effectively prepared data for high-stakes analysis. Additionally, discrepancies between Silver and Gold scores can reveal issues like flawed aggregations or business logic misalignment.

Win the Gold With DataOps Data Quality

In a Medallion architecture, maintaining data quality must be a continuous and iterative process that can be challenging to address through traditional, upfront planning approaches, defining all data quality requirements upfront often results in missed issues that only surface later, causing costly delays or setbacks. By adopting a DataOps approach, data quality becomes a dynamic cycle that allows teams to improve and refine data through iterative feedback loops incrementally. This approach maximizes speed while not compromising accuracy, enabling data engineers and quality specialists to identify and correct issues early on. DataOps Data Quality in a Medallion architecture emphasizes empowering individuals within the organization who, while they may not have direct authority, can influence and drive change.

An iterative DataOps cycle starts with measuring data to establish a baseline, followed by evaluating data quality through scoring systems that assess key metrics like accuracy, completeness, and consistency. Using these scores, teams can generate or improve data quality tests tailored to specific issues, such as missing values or anomalies, making quality monitoring proactive and targeted. Continuous data quality scoring then provides ongoing insights, tracking improvements and highlighting areas that need attention. Ultimately, the DataOps approach allows organizations to get data quality processes working quickly and then iterate and improve over time. Documentation and analysis become natural outcomes, not barriers to progress. By starting with testing and measurements, even before standards are fully established, organizations can build a foundation for continuous improvement. The faster the iteration, the more organizations learn, refine their processes, and elevate their data quality standards.

No one likes their customer to find data errors. Data testing in a Medallion architecture is essential for maintaining high data quality across each layer, ensuring that data becomes progressively cleaner, more consistent, and analysis-ready. While this architecture provides a structured path from raw to refined data, pinpointing and troubleshooting errors can make it more challenging as data traverses multiple layers. Quality testing at every stage, Bronze, Silver, and Gold, not only helps catch issues early but also minimizes the risk of data inconsistencies surfacing in customer-facing insights. After all, having a customer uncover a data error is always embarrassing and potentially damaging, so rigorous quality assurance within the Medallion architecture is critical. By prioritizing thorough testing and continuous monitoring, data teams can uphold data integrity, fostering trust in data across the organization.

Building Data Quality Into dbt: A Complete Guide

BUILDING YOUR TESTING STRATEGY

Building Data Quality Into dbt: A Complete Guide

The DataOps Way to Data Quality and Data Observability

DataKitchen

dbt has become the default transformation layer for most modern data warehouses. If your team transforms data in Snowflake, BigQuery, Redshift, or DuckDB, there is a good chance you are already using dbt or about to start. The testing framework built into dbt is one of its most underused features. This chapter walks through the full testing strategy, from project structure to CI/CD integration, with enough specificity to implement it in a real project.

Project Structure for Testability

Good test coverage starts before you write a single test. The way you organize your dbt project determines how easy it is to write tests, where those tests live, and how readable they are six months later.

A well-structured dbt project looks like this:

```
dbt_project/
├── dbt_project.yml
├── profiles.yml
├── models/
│   ├── staging/
│   │   ├── _sources.yml
│   │   ├── _staging.yml
│   │   ├── stg_orders.sql
│   │   ├── stg_customers.sql
│   │   └── stg_order_items.sql
│   ├── intermediate/
│   │   ├── _intermediate.yml
│   │   ├── int_orders_with_items.sql
│   │   └── int_customer_order_history.sql
│   └── marts/
│       ├── _marts.yml
│       ├── fct_orders.sql
│       ├── fct_revenue_daily.sql
│       ├── dim_customers.sql
│       └── dim_products.sql
├── tests/
│   ├── assert_orders_have_items.sql
│   └── assert_revenue_matches_payments.sql
├── macros/
│   ├── test_not_negative.sql
│   └── test_within_n_stddev.sql
└── snapshots/
    └── customers_snapshot.sql
```

Naming Conventions and Why They Matter

The prefixes `stg_`, `int_`, `fct_`, and `dim_` are not just style choices. They encode the layer of the pipeline a model belongs to, and that has direct implications for testing.

`stg_` models are direct transformations from raw source tables. They cast types, rename columns, and apply minimal business logic. Tests here should check that the raw data arrived intact: no nulls in primary keys, no unexpected column values, referential integrity to other staging models.

`int_` models join and reshape staging models. They encode business logic. Tests here should validate that the logic is correct: join cardinality (does every order have exactly one customer?), deduplication worked, date logic is sound.

`fct_` models are fact tables: one row per event (order, payment, click). They are the most queried models in a warehouse and the most important to test thoroughly. Business rule tests belong here.

`dim_` models are dimension tables: one row per entity (customer, product). They need uniqueness on the primary key and completeness checks on attributes that downstream analysts rely on.

This naming convention means you can write dbt test selectors that target a specific layer:

```
# Test only staging models
dbt test --select staging

# Test only mart models
dbt test --select marts
```

Generic Tests (Built-In)

dbt ships with four generic tests. They cover the most common data quality checks and require no custom code.

unique

Validates that a column has no duplicate values. Use this on every primary key in every model.

```
# models/staging/_staging.yml
version: 2

models:
  - name: stg_orders
    columns:
      - name: order_id
        tests:
          - unique
          - not_null
```

If your primary key is a composite (multiple columns together form the unique key), use the `dbt_utils.unique_combination_of_columns` test from the `dbt_utils` package instead.

not_null

Validates that a column contains no null values. Apply this aggressively to columns that downstream models depend on.

```
  - name: stg_customers
    columns:
      - name: customer_id
        tests:
          - not_null
      - name: email
        tests:
          - not_null
      - name: created_at
        tests:
          - not_null
```

Be deliberate about which columns get `not_null`. If a column is legitimately nullable in the business domain, do not add the test just to have coverage. False positives erode trust in your test suite faster than gaps in coverage.

accepted_values

Validates that a column only contains values from a specified list. Use this on status columns, type fields, and any column with a controlled vocabulary.

```
  - name: stg_orders
    columns:
      - name: status
        tests:
          - accepted_values:
              values: ['pending', 'processing', 'shipped', 'delivered', 'cancelled',
'refunded']
```

A useful option is `quote: false` for numeric or boolean values:

```
      - name: is_gift
        tests:
          - accepted_values:
              values: [true, false]
              quote: false
```

relationships

Validates referential integrity: every value in column A exists in column B of another model. This is the closest thing dbt has to a foreign key constraint.

```
  - name: stg_orders
    columns:
      - name: customer_id
        tests:
          - relationships:
              to: ref('stg_customers')
              field: customer_id
```

The `relationships` test runs a query that finds values in the child table that have no match in the parent table. If any rows come back, the test fails. Use this test liberally across staging models to catch referential integrity failures early, before they propagate into joins.

Custom Generic Tests

The four built-in tests cover common cases, but real projects need more. dbt lets you write custom generic tests as macros. A custom generic test is a Jinja macro that accepts a model and a column name (plus any additional arguments) and returns a SQL query that returns rows when the test fails.

Example: Checking for Statistical Outliers

This test checks whether a numeric column's current average has drifted more than N standard deviations from its 30-day rolling average. It is useful for catching data pipeline bugs that cause a sudden spike or drop in a metric.

```
-- macros/test_within_n_stddev.sql
{% test within_n_stddev(model, column_name, date_column, n_stddev=3, lookback_days=30) %}

with historical_stats as (
    select
        avg({{ column_name }}) as historical_mean,
        stddev({{ column_name }}) as historical_stddev
    from {{ model }}
    where
        {{ date_column }} >= current_date - interval '{{ lookback_days }} days'
        and {{ date_column }} < current_date
),

current_value as (
    select avg({{ column_name }}) as current_mean
    from {{ model }}
    where {{ date_column }} = current_date
),

validation as (
    select
        cv.current_mean,
        hs.historical_mean,
        hs.historical_stddev,
        abs(cv.current_mean - hs.historical_mean) as deviation,
        hs.historical_stddev * {{ n_stddev }} as allowed_deviation
    from current_value cv
    cross join historical_stats hs
)

select *
from validation
where deviation > allowed_deviation
  and historical_stddev > 0

{% endtest %}
```

Apply this test in a YAML file:

```
# models/marts/_marts.yml
version: 2

models:
  - name: fct_orders
    columns:
      - name: order_amount_usd
        tests:
          - within_n_stddev:
              date_column: order_date
              n_stddev: 3
              lookback_days: 30
```

When this test runs, dbt compiles the macro into a SQL query for the target warehouse and runs it. If any rows come back, the test fails. You can customize the threshold per model by changing `n_stddev`.

Singular Tests

Singular tests are plain SQL files in the `tests/` directory. They do not use macros or YAML. A singular test is a SQL query that returns rows when the test fails. They are the right tool for testing business rules that span multiple tables and cannot be expressed as a generic test.

Example: Every Order Must Have at Least One Line Item

```
-- tests/assert_orders_have_items.sql
-- Every row in fct_orders must have at least one corresponding row
-- in fct_order_items. An order with no items is a data integrity failure.

with orders as (
    select order_id
    from {{ ref('fct_orders') }}
),

order_items as (
    select distinct order_id
    from {{ ref('fct_order_items') }}
),

orders_without_items as (
    select o.order_id
    from orders o
    left join order_items oi
        on o.order_id = oi.order_id
    where oi.order_id is null
)

select *
from orders_without_items
```

If any orders lack items, this query returns rows and the test fails. The SQL is readable by anyone on the team without knowledge of the dbt testing framework.

Another example: validating that daily revenue in the fact table matches the sum of individual order amounts:

```sql
-- tests/assert_revenue_matches_payments.sql
with daily_revenue as (
    select
        order_date,
        sum(order_amount_usd) as total_revenue
    from {{ ref('fct_orders') }}
    where status = 'delivered'
    group by order_date
),

daily_payments as (
    select
        payment_date,
        sum(payment_amount_usd) as total_payments
    from {{ ref('fct_payments') }}
    where payment_status = 'settled'
    group by payment_date
),

mismatches as (
    select
        r.order_date,
        r.total_revenue,
        p.total_payments,
        abs(r.total_revenue - p.total_payments) as discrepancy
    from daily_revenue r
    inner join daily_payments p
        on r.order_date = p.payment_date
    where abs(r.total_revenue - p.total_payments) > 0.01
)

select *
from mismatches
```

Testing Staging Models vs. Marts

Staging models and mart models serve different purposes, and the tests for each reflect that difference.

Staging Model Tests: Structural Integrity

Staging models represent your interface to the source system. The tests here are structural: does the data arrive in the shape you expect?

```yaml
# models/staging/_staging.yml
version: 2

models:
  - name: stg_transactions
    description: "Raw transactions from the payments database, cleaned and typed"
    columns:
      - name: transaction_id
        description: "Primary key. UUID from source system."
        tests:
          - unique
          - not_null
      - name: account_id
        tests:
          - not_null
          - relationships:
              to: ref('stg_accounts')
              field: account_id
      - name: amount_cents
        tests:
          - not_null
      - name: currency_code
        tests:
          - not_null
          - accepted_values:
              values: ['USD', 'EUR', 'GBP', 'CAD', 'AUD']
      - name: created_at
        tests:
          - not_null
```

Mart Model Tests: Business Rules

Mart models encode business logic. The tests here validate that the logic is correct.

```yaml
# models/marts/_marts.yml
version: 2

models:
  - name: fct_orders
    description: "One row per order. Grain: order_id."
    tests:
      - dbt_utils.expression_is_true:
          expression: "order_amount_usd >= 0"
      - dbt_utils.expression_is_true:
          expression: "delivered_at >= placed_at or delivered_at is null"
    columns:
      - name: order_id
        tests:
          - unique
          - not_null
      - name: status
        tests:
          - accepted_values:
              values: ['pending', 'processing', 'shipped', 'delivered', 'cancelled']
      - name: customer_id
        tests:
          - not_null
          - relationships:
              to: ref('dim_customers')
              field: customer_id
```

The distinction matters: if a `not_null` test fails on a staging model, the problem is in the source data or the extraction process. If a business rule test fails on a mart model, the problem is in your transformation logic. Knowing which layer failed points you to the right place to fix it.

dbt Test Execution in CI/CD

Running dbt tests in a CI pipeline ensures that every pull request is validated before it reaches production. Here is a GitHub Actions workflow that runs dbt tests on every PR targeting the main branch.

```yaml
# .github/workflows/dbt_ci.yml
name: dbt CI

on:
  pull_request:
    branches:
      - main
    paths:
      - 'models/**'
      - 'tests/**'
      - 'macros/**'

jobs:
  dbt-test:
    runs-on: ubuntu-latest
    environment: ci

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dbt
        run: pip install dbt-snowflake==1.7.0

      - name: Write dbt profiles
        run: |
          mkdir -p ~/.dbt
          cat > ~/.dbt/profiles.yml << EOF
          my_project:
            target: ci
            outputs:
              ci:
                type: snowflake
                account: ${{ secrets.SNOWFLAKE_ACCOUNT }}
                user: ${{ secrets.SNOWFLAKE_USER }}
                password: ${{ secrets.SNOWFLAKE_PASSWORD }}
                role: CI_ROLE
                database: ANALYTICS_CI
                warehouse: CI_WH
                schema: dbt_ci_${{ github.event.pull_request.number }}
                threads: 4
          EOF

      - name: dbt deps
        run: dbt deps

      - name: dbt build (compile + run + test)
        run: dbt build --select state:modified+
        env:
          DBT_PROFILES_DIR: ~/.dbt
```

```
      - name: Upload test results
        if: always()
        uses: actions/upload-artifact@v4
        with:
          name: dbt-test-results
          path: target/run_results.json
```

Handling Test Failures in CI

When `dbt test` fails, it exits with a non-zero status code, which fails the CI job. The goal is to fail fast on genuine data quality issues while not blocking developers on expected or acceptable deviations.

Use dbt's `severity` configuration to distinguish between hard failures (errors) and soft warnings:

```
# models/marts/_marts.yml
models:
  - name: fct_orders
    columns:
      - name: customer_id
        tests:
          - not_null:
              severity: error       # Blocks CI. This must never be null.
          - relationships:
              to: ref('dim_customers')
              field: customer_id
              severity: warn        # Warns but does not block. New customers may lag.
              config:
                warn_if: ">10"      # Only warn if more than 10 rows fail
                error_if: ">1000"   # Escalate to error if more than 1000 rows fail
```

The `warn_if` and `error_if` thresholds are useful for tests that are almost always satisfied but occasionally have a small number of legitimate exceptions. A `relationships` test on a late-arriving dimension, for example, should warn if a handful of rows fail but only hard-fail if hundreds fail (which indicates a pipeline problem, not a timing issue).

Source Freshness

Data freshness is a distinct concern from data accuracy. A table can be completely accurate but hours stale, which is a quality problem for any consumer who assumes the data is current. dbt source freshness lets you define and monitor staleness thresholds in your `sources.yml`.

```yaml
# models/staging/_sources.yml
version: 2

sources:
  - name: payments_db
    database: raw
    schema: payments
    freshness:
      warn_after: {count: 6, period: hour}
      error_after: {count: 24, period: hour}
    loaded_at_field: _extracted_at

    tables:
      - name: transactions
        description: "Raw transaction records from payments processing system"
        freshness:
          warn_after: {count: 2, period: hour}
          error_after: {count: 6, period: hour}
        loaded_at_field: created_at
        columns:
          - name: transaction_id
          - name: amount_cents
          - name: created_at

      - name: refunds
        description: "Refund events. Lower volume, less frequent."
        freshness:
          warn_after: {count: 12, period: hour}
          error_after: {count: 48, period: hour}
```

Run freshness checks with:

```
dbt source freshness
```

The `loaded_at_field` is the column dbt uses to determine when the most recent record arrived. It should be a timestamp column that reflects when the record entered the warehouse, not when the event occurred in the source system. Many EL tools populate a `_extracted_at` or `_loaded_at` column for this purpose.

The `warn_after` and `error_after` values should reflect the actual SLA for the source. A payments table updated every 15 minutes should have a tighter threshold than a product catalog updated weekly.

Handling Late-Arriving Dimensions

A common problem in production dbt pipelines: the fact table updates hourly, but the dimension table updates daily. The `relationships` test on the foreign key in the fact table will fail during the hours between fact updates and dimension updates, generating false positives that erode confidence in the test suite.

The solution is to write the test so it accounts for the known lag. The simplest approach is using the `where` clause on the relationships test:

```
# Only check referential integrity for orders older than 24 hours,
# giving the dimension table time to catch up.
models:
  - name: fct_orders
    columns:
      - name: customer_id
        tests:
          - relationships:
              to: ref('dim_customers')
              field: customer_id
              config:
                where: "order_placed_at < current_timestamp - interval '24 hours'"
```

For more complex lag situations, write a singular test that directly expresses the tolerance:

```
-- tests/assert_recent_orders_have_customers.sql
-- Orders older than 24 hours must always have a matching customer.
-- Orders less than 24 hours old may have a lag before the customer
-- dimension catches up. We only test the settled window.

with settled_orders as (
    select order_id, customer_id
    from {{ ref('fct_orders') }}
    where order_placed_at < current_timestamp - interval '24 hours'
),

missing_customers as (
    select o.order_id, o.customer_id
    from settled_orders o
    left join {{ ref('dim_customers') }} c
        on o.customer_id = c.customer_id
    where c.customer_id is null
)

select *
from missing_customers
```

This pattern is cleaner than suppressing the test entirely, because it still catches genuine integrity failures while giving the pipeline room to handle normal operational timing.

What dbt Tests Cannot Do

dbt tests are run on demand: when you execute `dbt test` or `dbt build`. This is a fundamental limitation that engineers sometimes overlook.

dbt tests do not monitor production continuously. If a pipeline runs at 3am and produces bad data, your dbt tests will not fire until someone runs them or a CI job triggers. If your business depends on data being accurate throughout the day, you need a tool that monitors the warehouse on a schedule, not just

in response to a code change.

dbt tests cannot detect statistical anomalies in historical data. The `within_n_stddev` macro shown earlier approximates anomaly detection, but it is fragile. It does not account for seasonality, day-of-week patterns, or trends. A purpose-built anomaly detection tool uses statistical models that understand these patterns. dbt does not.

dbt tests are not a substitute for production incident management. When a dbt test fails in production, you need a way to alert the right person, track the incident, and measure time-to-resolution. dbt does not have those workflows. You will need to integrate dbt test results with Slack, PagerDuty, or a dedicated observability tool.

dbt tests do not give you lineage-aware impact analysis. If the `stg_orders` model fails a test, dbt cannot automatically tell you which downstream dashboards and reports are affected. Lineage tooling that integrates with dbt (Monte Carlo, Atlan, or dbt's own Explorer in dbt Cloud) does.

Use dbt tests for what they are excellent at: validating your transformation logic at development time and in CI. For continuous production monitoring, statistical anomaly detection, and incident management, you need a complementary tool.

Summary

A complete dbt testing strategy is not complicated, but it requires deliberate choices at each layer:

- Structure your project with `stg_`, `int_`, `fct_`, and `dim_` prefixes so tests are easy to scope and target.
- Apply generic tests aggressively on staging models to catch source data problems early.
- Write custom generic tests for reusable statistical checks that span many models.
- Write singular tests for business rules that span multiple tables.
- Use `severity: warn` and threshold parameters to avoid test fatigue from known timing lags.
- Configure source freshness separately from data quality tests, because staleness and incorrectness are different failure modes.
- Run `dbt build` in CI on every pull request.
- Understand what dbt tests cannot do, and plan for complementary tooling where those gaps matter.

Data Contracts: Shifting the Burden of Quality to the Source

BUILDING YOUR TESTING STRATEGY

Data Contracts: Shifting the Burden of Quality to the Source

The DataOps Way to Data Quality and Data Observability

DataKitchen

Most data quality failures are not caused by bad transformation logic or broken pipelines. They are caused by a producer changing something upstream and a consumer not knowing about it. A column gets renamed. A status field gains a new value. A table starts arriving six hours late. The producer had a legitimate reason to make the change. The consumer had no way to know it was coming. Data contracts are the mechanism that makes these changes explicit, versioned, and testable before they cause a production incident.

What a Data Contract Is

A data contract is a formal, machine-readable agreement between a data producer and a data consumer. It specifies the schema, semantics, SLA, and quality expectations for a dataset. "Formal" means it is written down somewhere authoritative, not in a Confluence page that no one updates. "Machine-readable" means it can be parsed and enforced by code, not just read by a person.

A data contract answers the following questions:

- What columns does this table have, and what type is each one?
- Which columns are nullable?
- What does each column mean in business terms?
- When will this table be updated, and what is the tolerance if it is late?

- What values are valid in each column?
- Who owns this dataset and is responsible when it breaks?
- How will I be notified if the contract changes?

The word "contract" implies two parties. The producer is the team or system that writes data to a table, topic, or file. The consumer is the team or system that reads it. The contract is the interface between them. When the producer wants to make a breaking change, the contract forces them to notify consumers before the change lands. When the consumer's pipeline breaks, the contract tells them where to look and who to call.

Why Data Contracts Matter

Without a contract, the producer and consumer share an implicit agreement. The consumer assumes the schema will stay stable. The producer assumes no one critical depends on the exact column names. Both assumptions break eventually.

The root cause of most downstream data quality incidents is an undeclared schema change. The producer team renames `user_id` to `customer_id` during a systems migration. Every downstream query that referenced `user_id` starts returning nulls or failing outright. The incident may not be detected for hours if no test explicitly checks for the presence of that column.

Data contracts shift the burden of quality to where the problem actually originates. Instead of consumers writing defensive transformations to handle every possible schema variation, producers are accountable for maintaining the interface they publish. The contract is not a bureaucratic formality. It is the signal that separates "the producer broke the interface" from "the consumer wrote a fragile query."

Contract Components

A complete data contract includes at minimum the following components.

Schema defines every column: its name, data type, and whether it is nullable. This is the structural interface. If a column is removed or renamed, the schema version changes.

Semantics defines what each column means. Types are not enough. `amount` of type `NUMERIC` does not tell a consumer whether it is in cents or dollars, whether it is inclusive of tax, or whether negative values represent refunds or corrections. Semantics belong in the contract alongside the type.

SLA defines the freshness guarantee. "Updated daily by 6am UTC" is a meaningful SLA. "Updated daily" is not. The SLA should include both the expected delivery time and the tolerance window before a consumer should raise an incident.

Quality rules define the validity expectations beyond schema: value ranges, referential integrity, cardinality constraints, business rules. These are the assertions the producer agrees to maintain.

Owner is the name, team, and contact method for the person responsible when the contract is violated. A contract without an owner is not a contract; it is documentation that will rot.

Version describes how the contract changes over time and how consumers will be notified. A contract that can change without notice is only marginally better than no contract.

Example 1: A Financial Transactions Table

This is a complete data contract for a transactions table in a financial application. The YAML format is chosen because it is human-readable, can be committed to a version control system, and can be parsed programmatically.

```yaml
# data_contracts/transactions_v2.yml
contract:
  name: financial_transactions
  version: "2.1.0"
  description: >
    Core financial transactions table. One row per transaction.
    Source of truth for all payment processing events.
    Do not use staging.raw_transactions directly; use this contract's
    canonical table.
  owner:
    team: payments-engineering
    contact: payments-eng@company.com
    slack: "#payments-data"
  sla:
    schedule: "0 */1 * * *"          # Updated hourly
    max_delay_minutes: 30
    warn_after_minutes: 20
    error_after_minutes: 60

  table:
    database: analytics
    schema: finance
    name: transactions

  columns:
    - name: transaction_id
      type: VARCHAR(36)
      nullable: false
      description: >
        UUID v4. Primary key. Generated by the payments service at transaction
        creation time. Globally unique. Never reused.
      tests:
        - unique
        - not_null
        - format: "^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$"

    - name: account_id
      type: VARCHAR(36)
      nullable: false
      description: >
        UUID of the account that initiated the transaction.
        Foreign key to accounts.account_id.
      tests:
        - not_null
        - referential_integrity:
            table: accounts
            column: account_id

    - name: amount_cents
      type: BIGINT
      nullable: false
      description: >
        Transaction amount in the minor unit of the currency (cents for USD/EUR/GBP).
        Always positive. Refunds are represented in the refunds table, not as
        negative transactions here.
```

```
      tests:
        - not_null
        - min_value: 1
        - max_value: 100000000    # $1,000,000 USD maximum

    - name: currency_code
      type: VARCHAR(3)
      nullable: false
      description: "ISO 4217 three-letter currency code."
      tests:
        - not_null
        - accepted_values: ['USD', 'EUR', 'GBP', 'CAD', 'AUD', 'JPY', 'CHF']

    - name: status
      type: VARCHAR(20)
      nullable: false
      description: >
        Current status of the transaction in the payment lifecycle.
        Transitions are one-directional: pending -> processing -> settled or failed.
        A settled transaction is final. A failed transaction will not be retried.
      tests:
        - not_null
        - accepted_values: ['pending', 'processing', 'settled', 'failed', 'reversed']

    - name: created_at
      type: TIMESTAMP WITH TIME ZONE
      nullable: false
      description: "UTC timestamp when the transaction was created in the payments
service."
      tests:
        - not_null
        - not_in_future: true
        - not_before: "2020-01-01T00:00:00Z"

    - name: settled_at
      type: TIMESTAMP WITH TIME ZONE
      nullable: true
      description: >
        UTC timestamp when the transaction reached settled status.
        Null if status is pending, processing, or failed.
        Will always be >= created_at when not null.
      tests:
        - conditional_not_null:
            when: "status = 'settled'"
        - gte_column:
            column: created_at
            when_not_null: true

  changelog:
    - version: "2.1.0"
      date: "2025-11-01"
      author: payments-engineering
      change: "Added settled_at column. Backwards-compatible addition."
    - version: "2.0.0"
      date: "2025-06-15"
      author: payments-engineering
```

```
      change: "Renamed user_id to account_id. Breaking change. Migration window: 90 days."
    - version: "1.0.0"
      date: "2024-03-01"
      author: payments-engineering
      change: "Initial contract."
```

Example 2: A Web Events Stream

Event streams introduce additional contract requirements because they flow through a message broker. The schema must be registered before consumers can reliably decode messages. This example shows a web clickstream contract and how it maps to a Confluent Schema Registry Avro schema.

```yaml
# data_contracts/web_events_v1.yml
contract:
  name: web_page_viewed
  version: "1.3.0"
  description: >
    Fired whenever a user views a page on the marketing website or app.
    One event per page view. Not deduplicated at source; consumers must
    deduplicate on event_id if they need exactly-once semantics.
  owner:
    team: analytics-engineering
    contact: analytics@company.com
  sla:
    latency_p99_seconds: 5    # 99th percentile end-to-end latency
    throughput_avg_per_second: 200
    throughput_max_per_second: 2000

  transport:
    type: kafka
    cluster: production
    topic: analytics.web.page_viewed
    partitions: 12
    retention_days: 7
    schema_registry_subject: analytics.web.page_viewed-value

  fields:
    - name: event_id
      type: string
      nullable: false
      description: "UUID v4. Unique identifier for this event instance."

    - name: event_name
      type: string
      nullable: false
      description: "Always 'page_viewed' for this topic."
      accepted_values: ['page_viewed']

    - name: user_id
      type: string
      nullable: true
      description: >
        Internal user UUID. Null for anonymous sessions (not logged in).
        Consistent across sessions for authenticated users.

    - name: anonymous_id
      type: string
      nullable: false
      description: >
        Device-stable anonymous identifier. Always present.
        Assigned on first visit and persisted in a first-party cookie.

    - name: session_id
      type: string
      nullable: false
      description: >
        Session identifier. Resets after 30 minutes of inactivity or
```

```
        at midnight UTC, whichever comes first.

    - name: page_url
      type: string
      nullable: false
      description: "Full URL including protocol, host, path, and query string."
      format: "^https?://"

    - name: page_path
      type: string
      nullable: false
      description: "URL path component only, without host or query string. Starts with /."

    - name: referrer_url
      type: string
      nullable: true
      description: "HTTP referrer. Null if direct traffic or referrer policy blocked it."

    - name: occurred_at
      type: string
      nullable: false
      description: "ISO 8601 UTC timestamp. Format: YYYY-MM-DDTHH:MM:SS.sssZ"
      format: "^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}\\.\\d{3}Z$"
```

The corresponding Avro schema for the Confluent Schema Registry:

```
{
  "type": "record",
  "name": "WebPageViewed",
  "namespace": "com.company.analytics.web",
  "doc": "Web page view event. Contract version 1.3.0.",
  "fields": [
    {"name": "event_id",     "type": "string",           "doc": "UUID v4 event
identifier"},
    {"name": "event_name",   "type": "string",           "doc": "Always page_viewed"},
    {"name": "user_id",      "type": ["null", "string"], "default": null, "doc": "Internal
user UUID, null if anonymous"},
    {"name": "anonymous_id", "type": "string",           "doc": "Device-stable anonymous
identifier"},
    {"name": "session_id",   "type": "string",           "doc": "Session identifier"},
    {"name": "page_url",     "type": "string",           "doc": "Full page URL"},
    {"name": "page_path",    "type": "string",           "doc": "URL path component"},
    {"name": "referrer_url", "type": ["null", "string"], "default": null, "doc": "HTTP
referrer"},
    {"name": "occurred_at",  "type": "string",           "doc": "ISO 8601 UTC timestamp"}
  ]
}
```

Register this schema with the Confluent Schema Registry so that producers and consumers can validate messages at the serialization layer:

```
curl -X POST http://schema-registry:8081/subjects/analytics.web.page_viewed-value/versions
\
  -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  -d '{
    "schema": "
{\"type\":\"record\",\"name\":\"WebPageViewed\",\"namespace\":\"com.company.analytics.web\"
,\"fields\":[{\"name\":\"event_id\",\"type\":\"string\"},
{\"name\":\"event_name\",\"type\":\"string\"},{\"name\":\"user_id\",\"type\":
[\"null\",\"string\"],\"default\":null},{\"name\":\"anonymous_id\",\"type\":\"string\"},
{\"name\":\"session_id\",\"type\":\"string\"},{\"name\":\"page_url\",\"type\":\"string\"},
{\"name\":\"page_path\",\"type\":\"string\"},{\"name\":\"referrer_url\",\"type\":
[\"null\",\"string\"],\"default\":null},{\"name\":\"occurred_at\",\"type\":\"string\"}]}"
  }'
```

With the schema registered, the Kafka producer will fail at serialization time if it tries to send a message that violates the schema. The consumer will fail at deserialization time if it receives a message it cannot parse. This is enforcement at the transport layer, before the data ever reaches a warehouse or a dbt model.

How to Enforce Contracts in Code

A contract document is only as useful as its enforcement mechanism. There are three primary enforcement points for warehouse and pipeline data.

dbt Contracts (dbt 1.5+)

dbt 1.5 introduced model-level contracts. When `contract: enforced: true` is set, dbt will fail at compile time if the model's output schema does not match the declared schema.

```yaml
# models/marts/_marts.yml
version: 2

models:
  - name: fct_transactions
    config:
      contract:
        enforced: true
    description: "Financial transactions fact table. Contract version 2.1.0."
    columns:
      - name: transaction_id
        data_type: varchar
        constraints:
          - type: not_null
          - type: unique
      - name: account_id
        data_type: varchar
        constraints:
          - type: not_null
      - name: amount_cents
        data_type: bigint
        constraints:
          - type: not_null
      - name: currency_code
        data_type: varchar
        constraints:
          - type: not_null
      - name: status
        data_type: varchar
        constraints:
          - type: not_null
      - name: created_at
        data_type: timestamp with time zone
        constraints:
          - type: not_null
      - name: settled_at
        data_type: timestamp with time zone
```

With `contract: enforced: true`, dbt compiles the model and compares the actual column names and types in the SELECT statement against the declared schema. If they do not match, the run fails before any SQL executes in the warehouse. This catches schema drift at development time rather than at runtime.

Python: Pandera Validation

For pipelines that process data in Python (Pandas DataFrames, Polars, or Spark via PySpark), Pandera provides a contract enforcement library that mirrors the contract document in code.

```
import pandera as pa
from pandera import Column, DataFrameSchema, Check
import pandas as pd

# Define the contract as a Pandera schema
transactions_contract = DataFrameSchema(
    columns={
        "transaction_id": Column(
            str,
            nullable=False,
            unique=True,
            checks=Check.str_matches(
                r"^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$"
            ),
        ),
        "account_id": Column(str, nullable=False),
        "amount_cents": Column(
            int,
            nullable=False,
            checks=[
                Check.greater_than_or_equal_to(1),
                Check.less_than_or_equal_to(100_000_000),
            ],
        ),
        "currency_code": Column(
            str,
            nullable=False,
            checks=Check.isin(["USD", "EUR", "GBP", "CAD", "AUD", "JPY", "CHF"]),
        ),
        "status": Column(
            str,
            nullable=False,
            checks=Check.isin(
                ["pending", "processing", "settled", "failed", "reversed"]
            ),
        ),
        "created_at": Column(pd.Timestamp, nullable=False),
        "settled_at": Column(pd.Timestamp, nullable=True),
    },
    checks=[
        # settled_at must be >= created_at when not null
        Check(
            lambda df: df.loc[df["settled_at"].notna(), "settled_at"]
            >= df.loc[df["settled_at"].notna(), "created_at"],
            error="settled_at must not be earlier than created_at",
        )
    ],
    coerce=False,  # Do not silently coerce types; fail if types don't match
)


def validate_transactions(df: pd.DataFrame) -> pd.DataFrame:
    """
    Validate a transactions DataFrame against the contract.
    Raises SchemaError if the DataFrame violates the contract.
```

```
    Returns the validated DataFrame unchanged.
    """
    return transactions_contract.validate(df, lazy=True)


# Usage in a pipeline step
def process_transactions(raw_df: pd.DataFrame) -> pd.DataFrame:
    # ... transform the data ...
    cleaned_df = raw_df.rename(columns={"id": "transaction_id"})

    # Validate against the contract before writing downstream
    validated_df = validate_transactions(cleaned_df)
    return validated_df
```

The `lazy=True` flag collects all validation errors before raising an exception, rather than stopping at the first failure. This is important in production: you want to know all the ways a batch violated the contract, not just the first one.

How to Evolve Contracts Without Breaking Consumers

Contracts are living documents. Producers need to change schemas. The question is not whether changes will happen, but how to manage them without causing downstream incidents.

The framework is semantic versioning applied to data: major versions for breaking changes, minor versions for backwards-compatible additions, patch versions for documentation or semantic clarification.

Backwards-compatible changes (minor version bump):

- Adding a new nullable column
- Adding a new accepted value to a status field
- Loosening a constraint (e.g., increasing the maximum allowed value)
- Adding a new table that does not replace an existing one

Breaking changes (major version bump):

- Renaming a column
- Removing a column
- Changing a column's type
- Making a nullable column non-nullable
- Removing an accepted value from a controlled vocabulary
- Changing the grain of a table (e.g., from one row per order to one row per order-item)

The change management process for a breaking change:

1. **Announce the change** to all known consumers before it happens. Do not rely on consumers to discover it. Check lineage tooling, data catalog entries, and BI tool dependency lists to identify who uses the dataset.

1. **Set a migration window.** Ninety days is a reasonable default for internal consumers. Shorten or lengthen it based on the criticality of the dataset and the complexity of the consumer migration.
1. **Publish both versions in parallel** during the migration window. The old version continues to work. The new version is available for consumers to migrate to. This requires either maintaining two separate tables or publishing a view that maps the old schema to the new one.
1. **Add deprecation markers** to the old version so that consumers see a warning. In dbt, this is a description that notes the deprecation and the retirement date. In a schema registry, it is a metadata field.
1. **Retire the old version** after the migration window closes, but only after confirming no consumer is still using it. Check query logs, ETL job configurations, and BI tool connections before deleting.

```
# Deprecation notice in a dbt model description
- name: fct_transactions_v1
  description: >
    DEPRECATED. Retirement date: 2026-03-01.
    Migrate to fct_transactions_v2. The column user_id has been renamed
    to account_id. Contact payments-eng@company.com with questions.
    This model will be removed after the retirement date.
  config:
    tags: ['deprecated']
```

Starting Simple

The right way to start with data contracts is not to write contracts for every table in the warehouse. That is a multi-month effort that will lose momentum before it produces value. Start with three tables:

Identify the tables that cause the most downstream breakage. Look at your incident history. Which tables broke their consumers most often in the last six months? Which schema changes required the most unplanned work? Those three tables are your starting point.

Write informal contracts first. A markdown file committed to the dbt project repository alongside the model is a good enough start:

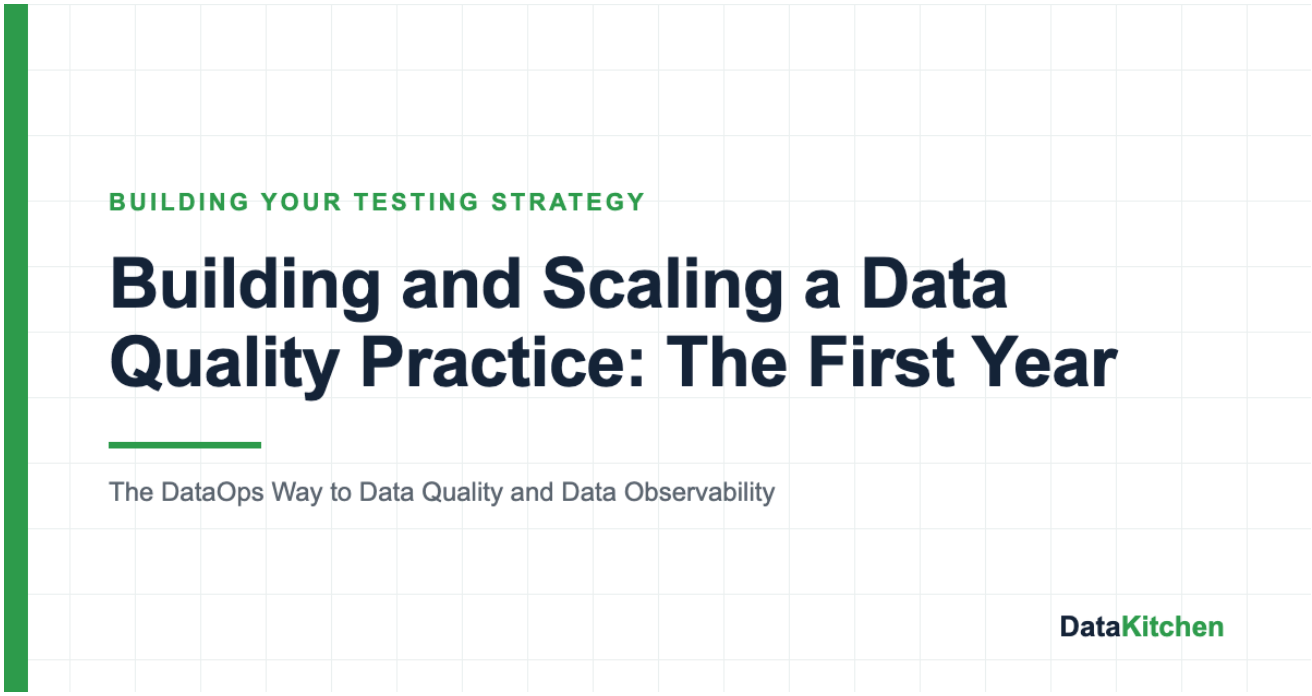
```
# transactions contract

**Owner:** payments-engineering
**SLA:** Updated hourly. Available by :30 past each hour.
**Contact:** payments-eng@company.com

## Schema

| Column         | Type      | Nullable | Description
|
|----------------|-----------|----------|-------------------------------------------------
|
| transaction_id | VARCHAR   | No       | UUID v4. Primary key.
|
| account_id     | VARCHAR   | No       | FK to accounts.account_id.
|
| amount_cents   | BIGINT    | No       | Amount in minor currency unit. Always positive.
|
| currency_code  | VARCHAR   | No       | ISO 4217. One of: USD, EUR, GBP, CAD, AUD.
|
| status         | VARCHAR   | No       | One of: pending, processing, settled, failed.
|
| created_at     | TIMESTAMP | No       | UTC. When the transaction was created.
|
| settled_at     | TIMESTAMP | Yes      | UTC. Null unless status = settled.
|

## Change Policy

Breaking changes require 90 days notice to consumers. Contact the owner
before making any changes to column names, types, or removing columns.
```

This document is not machine-enforceable, but it creates accountability. The producer cannot claim ignorance of the interface when a consumer is broken by an undeclared change.

Automate enforcement incrementally. Once the informal contract is stable and trusted, encode it as dbt column tests and dbt model contracts. Add Pandera validation to any Python pipelines that consume the table. Register the schema in a schema registry if the data flows through Kafka. Each step adds enforcement without requiring a full platform investment upfront.

The goal is not a perfect contract framework on day one. The goal is a clear interface with an accountable owner and a process for managing changes. Everything else follows from that.

Building and Scaling a Data Quality Practice: The First Year



Most data quality programs fail not because the technology is wrong but because the rollout is. A team installs a tool, writes a hundred tests in a sprint, fires alerts at everyone for two weeks, and then quietly turns it all off because the noise became unbearable. This chapter is a month-by-month playbook for doing it differently. It is built from what real teams actually do, not what the tool vendors recommend.

Before You Start: The Right Framing

A data quality practice is not a project. It does not have a start date and an end date. It is a capability you build incrementally and improve continuously. Teams that treat it as a one-time initiative almost always abandon it within six months.

The first year is about establishing a foundation: knowing which data matters most, having basic checks in place for those assets, and creating enough operational discipline that the practice can sustain itself without heroic effort.

Months 1 and 2: Audit and Prioritize

The most important work in the first two months happens before you write a single test. If you skip this phase and go straight to writing checks, you will spend time testing the wrong things and have nothing to show for it when someone asks why the revenue table still breaks every other Monday.

Inventory your data assets.

Start with a simple spreadsheet. List every table, pipeline, and data feed your team owns or depends on. Include the source system, the destination, approximate row count, load frequency, and who uses it. Do not try to be comprehensive on the first pass. Aim for the 30 to 50 assets that are most visible and most frequently referenced.

Most teams are surprised to discover they have no centralized list of what data they own. This inventory is valuable independent of data quality work.

Identify the ten tables that matter most.

From your inventory, select the ten tables or pipelines where a failure would cause immediate, visible business pain. Use these criteria:

- Executives or business partners use this data daily or weekly.
- A decision that affects revenue, customers, or operations depends on this data.
- A failure in this table would trigger an urgent Slack message or a meeting.

These are your Tier 1 assets. Every data quality effort starts here.

Assess current test coverage.

For each of your top ten tables, document what tests currently exist. For most teams starting out, the answer is: almost none. You may have some pipeline-level checks (the job ran, no errors were thrown), but almost certainly no data-content checks (the numbers are correct, the nulls are where expected, the row counts are reasonable).

Write this down. "Zero test coverage on eight of our ten critical tables" is a compelling slide in the budget conversation.

Set baseline metrics.

Before you change anything, measure the current state. Pull the following from your incident history if it exists, or estimate from memory if it does not:

- How many data incidents reached business users in the last 90 days?
- How many hours did the engineering team spend on data incident response?
- How many times did someone in a business meeting say "I don't trust this number"?

These baselines become your before/after comparison at the end of year one. Teams that skip this step have nothing to point to when they need to justify the budget for year two.

Months 3 and 4: First Tests

You are now ready to write tests. Start with the simplest possible checks on your most important tables.

Start with freshness and row counts.

Freshness: is the data up to date? For each Tier 1 table, write a check that fails if the maximum timestamp in the table is more than N hours old, where N is 1.5 times the expected load interval. If the table loads daily at 6am, the freshness check should fire if the data is more than 36 hours stale.

Row count: is the table roughly the right size? For each Tier 1 table, write a check that fails if today's row count is less than 50% of the 7-day average row count. This is a coarse check, but it catches dropped loads, truncations, and upstream source failures immediately.

In dbt, freshness checks are built in via the `sources.yml` configuration. Row count checks can be written as a simple dbt test or a SQL query run in your orchestration tool. In Great Expectations, both checks are available as built-in expectations. If you are using neither, a simple SQL query scheduled in your orchestration layer works fine.

```
-- Example: row count anomaly check
SELECT
  CASE
    WHEN today_count < avg_7d * 0.5 THEN 'FAIL'
    ELSE 'PASS'
  END AS check_result
FROM (
  SELECT
    COUNT(*) AS today_count,
    AVG(daily_count) OVER () AS avg_7d
  FROM daily_row_counts
  WHERE load_date >= CURRENT_DATE - 7
)
```

Deploy to production, not to development.

One of the most common mistakes in early data quality programs is running checks only in development or in a staging environment. Production is where the real failures happen. Get your first ten checks running against production data within the first week.

Handle the first alert that fires.

Your first alert will fire within a week, possibly within a day. How you handle it sets the tone for the entire program.

Treat it as a genuine incident. Investigate it the same way you would investigate a software outage: identify the root cause, document what happened, fix the underlying issue, and document how you fixed it. Write a one-paragraph incident summary and send it to your stakeholders.

This is important for two reasons. First, it demonstrates that the alerting system works and that the team takes data quality seriously. Second, it teaches you whether your alert thresholds are calibrated correctly. If the first alert turns out to be a false positive caused by a threshold that is too tight, adjust the threshold immediately and note it in your documentation.

Communicate with stakeholders.

After your first real incident and first real fix, send a brief note to the business owners of the affected data. Something like: "Our new data quality monitoring caught an issue with the customer table on Tuesday morning. The table was missing two hours of data due to an upstream feed delay. We resolved the issue by 9am and have added a check to catch this automatically in the future."

This kind of communication builds trust. Business users start to see the data team as proactive rather than reactive.

Months 5 and 6: Expand Coverage

With the basics working, start expanding the depth and breadth of your checks.

Add column-level checks.

For each Tier 1 table, add checks for:

- Null rates: the percentage of nulls in key columns should not exceed a threshold. If customer_id is 2% null today and 0.1% null on average, something is wrong.
- Cardinality: for columns with expected distinct values (like status codes, country codes, or product categories), check that no new unexpected values have appeared.
- Range checks: for numeric columns like revenue, age, or quantity, verify that values fall within a reasonable range. Revenue should be non-negative. Age should be between 0 and 120.
- Referential integrity: if order_items.order_id references orders.id, check that all order_ids in order_items exist in orders.

Do not add all of these to every column. Be deliberate. Ask: what column-level failure would actually hurt the business? Start there.

Build a simple coverage dashboard.

Create a simple dashboard showing: how many Tier 1 tables have freshness checks, row count checks, and column-level checks. Track pass/fail rates over time. This does not need to be beautiful. A shared Google Sheet or a simple dbt docs page is fine.

The dashboard serves two purposes. It gives the team a clear view of where coverage is thin. And it gives data leaders a concrete artifact to show stakeholders when the conversation turns to "is our data quality getting better?"

Get buy-in from the business.

By month six, you should have enough incident data to tell a story. Compile a simple summary: how many issues were caught by automated checks before they reached the business, how many hours of investigation time were saved, and what the team fixed proactively as a result of the checks.

Present this to your business stakeholders. Not as a data quality roadmap or a technical overview. As a simple scorecard: "Here is what broke, here is how quickly we caught it, here is what you did not have to deal with because we caught it first."

Months 7 Through 9: Monitoring and Alerting

By month seven, you have checks running. Now you need to make sure the alerting system is working for the team rather than against it.

Set up alert routing.

Not every alert should go to every engineer. Define a routing matrix based on table ownership and business domain.

Table Tier	Alert Type	Goes To
Tier 1	Freshness failure, row count anomaly	On-call engineer + table owner
Tier 1	Column-level check failure	Table owner
Tier 2	Any failure	Table owner (no on-call page)
Tier 3	Any failure	Logged to Slack channel, no direct notification

Implement this routing in your alerting tool. If you are using dbt Cloud, Slack integrations are straightforward. If you are using custom SQL checks, route alerts through your orchestration tool's notification system or a lightweight tool like PagerDuty or OpsGenie.

Avoid alert fatigue.

Alert fatigue is the single most common reason data quality programs fail. The pattern is predictable: alerts fire constantly, the team stops responding to them, management notices the team is ignoring alerts, trust in the program collapses.

Prevent this by setting a rule from the start: if an alert fires more than three times per week and is not actionable every time it fires, it is not a useful alert. Either the threshold is wrong or the underlying issue is a known, accepted condition (like a load window that makes the table temporarily stale).

Suppress known-noisy checks during load windows. If your main pipeline runs from midnight to 3am and freshness checks fire every night during that window, either widen the freshness threshold or exclude that window from the check schedule.

Add business rule tests.

After the basic infrastructure is stable, start adding tests that encode specific business rules. Examples:

- Total daily revenue should be within 20% of the same day last week (adjusted for day-of-week effects).
- Every order should have at least one order item.
- Refund amounts should never exceed the original order amount.

- Active user counts should not drop more than 30% day-over-day without a corresponding explanation.

These tests require input from business partners. Schedule brief working sessions with your finance, marketing, and operations contacts to ask: what would tell you that this data is wrong? Their answers become your business rule tests.

Months 10 Through 12: Maturing the Practice

The final quarter of year one is about stabilizing what you have built and starting to expand it.

Add data contracts for high-priority feeds.

A data contract is a formal agreement between a data producer and a data consumer that defines what the data should look like: schema, expected values, freshness SLA, and ownership. Contracts make the implicit explicit. When an upstream team changes a column name or drops a field, the contract violation is caught immediately rather than after it has propagated downstream.

Start with your two or three highest-priority upstream feeds. Define the contract in a YAML file (several open-source frameworks support this, including dbt contracts and Soda's contract format). Add automated checks that validate the contract on every load.

Expand to a second team.

By month ten, you have a working model. Now is the time to bring a second team onto the framework. Pick a team that has expressed interest and has relatively clean, well-understood data. Avoid picking the most complex or political domain for the second expansion.

The expansion will reveal gaps in your documentation and tooling. Some things that worked for the first team will not work for the second. That friction is valuable: it forces you to generalize your approach and build the processes that make the practice scalable.

Measure the improvement in incident rate.

At the end of month twelve, compare your incident metrics to the baselines you set in months one and two. Measure:

- Total data incidents reaching business users (should be lower)
- Average time to detect an incident (should be much lower)
- Average time to resolve an incident (should be lower)
- Engineer hours spent on data incident response (should be lower)

If you have been running the program consistently, you should see a 40% to 70% reduction in incidents reaching business users, and a significant reduction in time-to-detect for the incidents that do occur.

Package this into a one-page summary for leadership. The summary should answer one question: what did we buy with the data quality investment this year? The answer should be specific, in hours and dollars where possible, and tied back to the ROI model you built at the start.

What Will Go Wrong

Honest acknowledgment: several things will not go according to this plan.

Your baseline metrics will be incomplete because your team does not have consistent incident records. Estimate where you have to, but flag the gaps.

Some of your Tier 1 tables will turn out to be more complex than expected, with dozens of interdependencies that make simple threshold checks misleading. You will spend more time calibrating thresholds than you planned.

Your first round of business rule tests will surface disagreements between teams about what the data should look like. This is not a data quality problem. It is a data definition problem, and it is valuable to surface even though it is painful.

Someone will disable the alerts during a crunch period and not re-enable them. Build the re-enabling step into your incident postmortem process.

None of these problems should stop you. They are normal. The teams that build lasting data quality practices are not the ones that avoid these problems. They are the ones that work through them methodically and keep going.

Impact Dimensions Cure A Data Quality Blind Spot

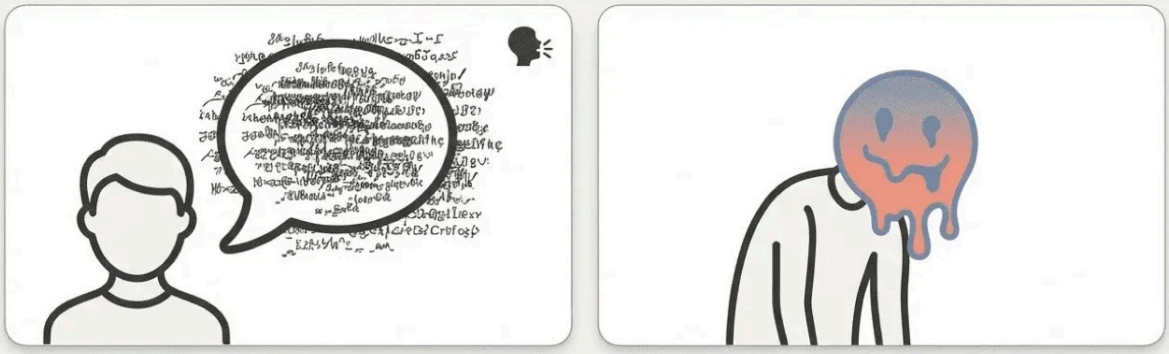
Read the original article at: https://datakitchen.io/blog/impact-dimensions/

Your Data Quality Assessment Has a Blind Spot

The reason my doctor makes more money than I do is that a description is not a diagnosis, and a diagnosis is not a treatment. I have a neighbor who can talk your ear off at a party about his latest rash. This is one of his favorite subjects, although it does not seem to engender long-term friendships. His most vivid descriptions, which give him great satisfaction, don't get him any closer to a cure.

We all have that one neighbor.

The one who corners you at a party to describe their latest rash in vivid, exhausting detail. Their descriptions give them great satisfaction, but they don't engender friendships—and they don't get anyone closer to a cure.



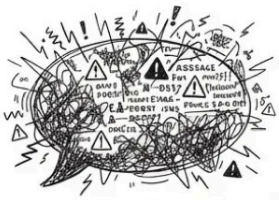
A description is not a diagnosis. And a diagnosis is not a treatment.
(This is the reason doctors make more money than your neighbor.)

This is a crucial distinction for data quality, and one that the smartest people often seem to forget. Take traditional data quality dimensions (accuracy, validity, etc), which, like my neighbor, give you a lot of information that you don't need to know. (An equally valid point is that data quality, like skin conditions, is a subject that won't get you anywhere at social gatherings, but you already know that.)

Traditional data quality software has a blind spot.

Modern DQ platforms are powerful–they can automatically generate 84+ distinct types of checks.
But a comprehensive taxonomy of the flora and fauna of defects isn't enough.

The Talkative Neighbor 🗣️

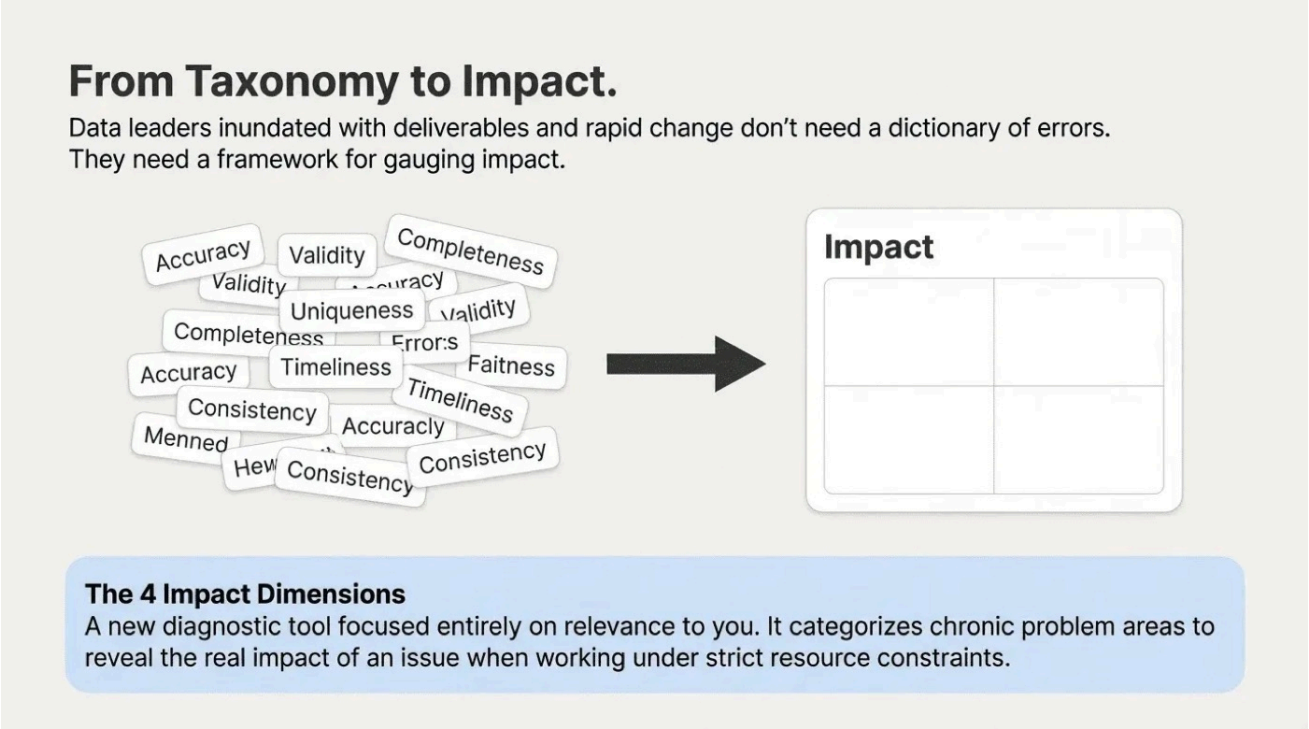


Your DQ Dashboard 💻

❌ It tells you everything that is wrong.
❌ It inundates you with technical details.
🚨 **The Blind Spot:** It answers the dashboard with silence on the two most important questions: **How do we prioritize?** and **Why should we fix them?**

This is part of a larger issue that our company has struggled to figure out as we've worked to implement our data quality software, TestGen, in a range of corporate settings. It's very nice that our app identifies a wide range of data quality problems automatically. But what are people supposed to do about it? Essentially, data quality software becomes the kind of very talkative person you really want to avoid at parties.

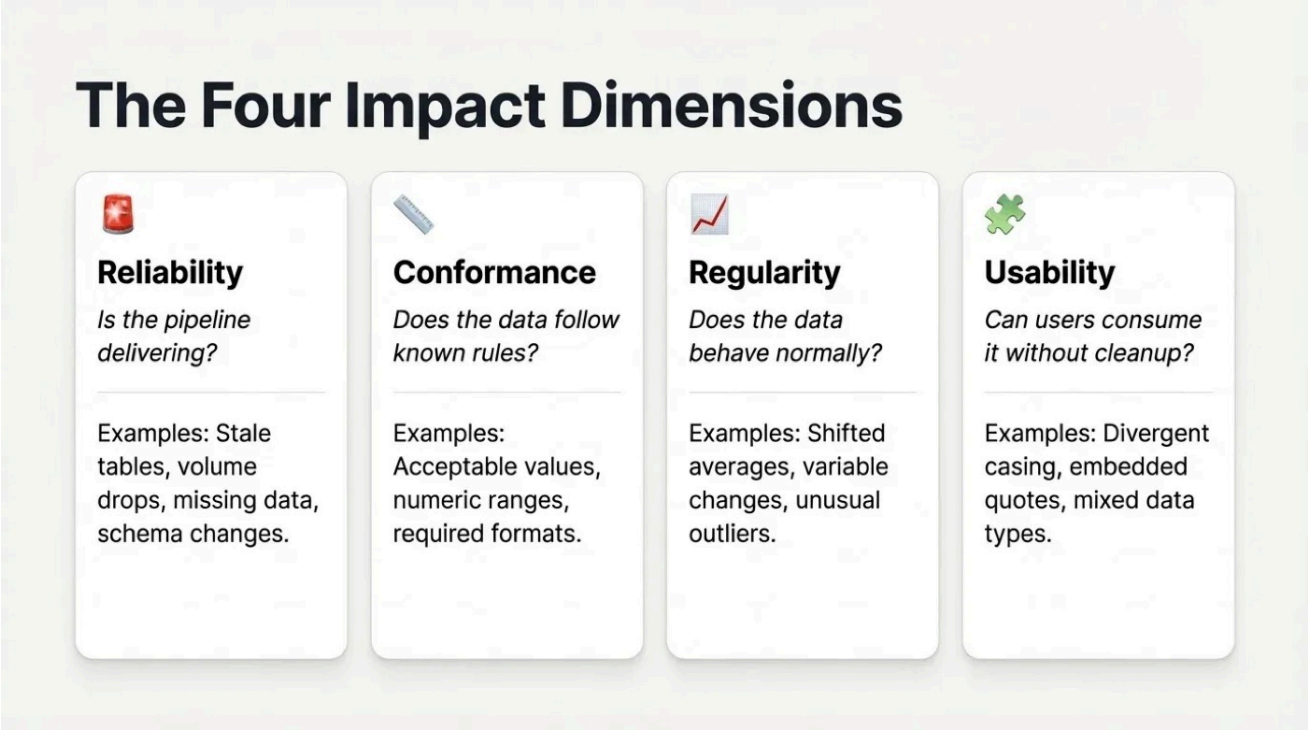
We've worked hard to make TestGen as easy as possible to implement. It's reached a point where you can just point it at a dataset and automatically generate a lot of findings. I mean a lot. We have 84 distinct types of checks across our platform, and the results they generate will give you some crucial information about the state of your data space. And some less crucial information.



The questions we weren't answering: how to prioritize problems? Why fix them?

This is not self-evident in a world just inundated with data, with deliverables, with deadlines, and with rapid change. The canonical DQ Dimensions were designed to be a complete taxonomy of the flora and fauna of defect types. But data leaders need a framework for gauging impact and prioritizing action.

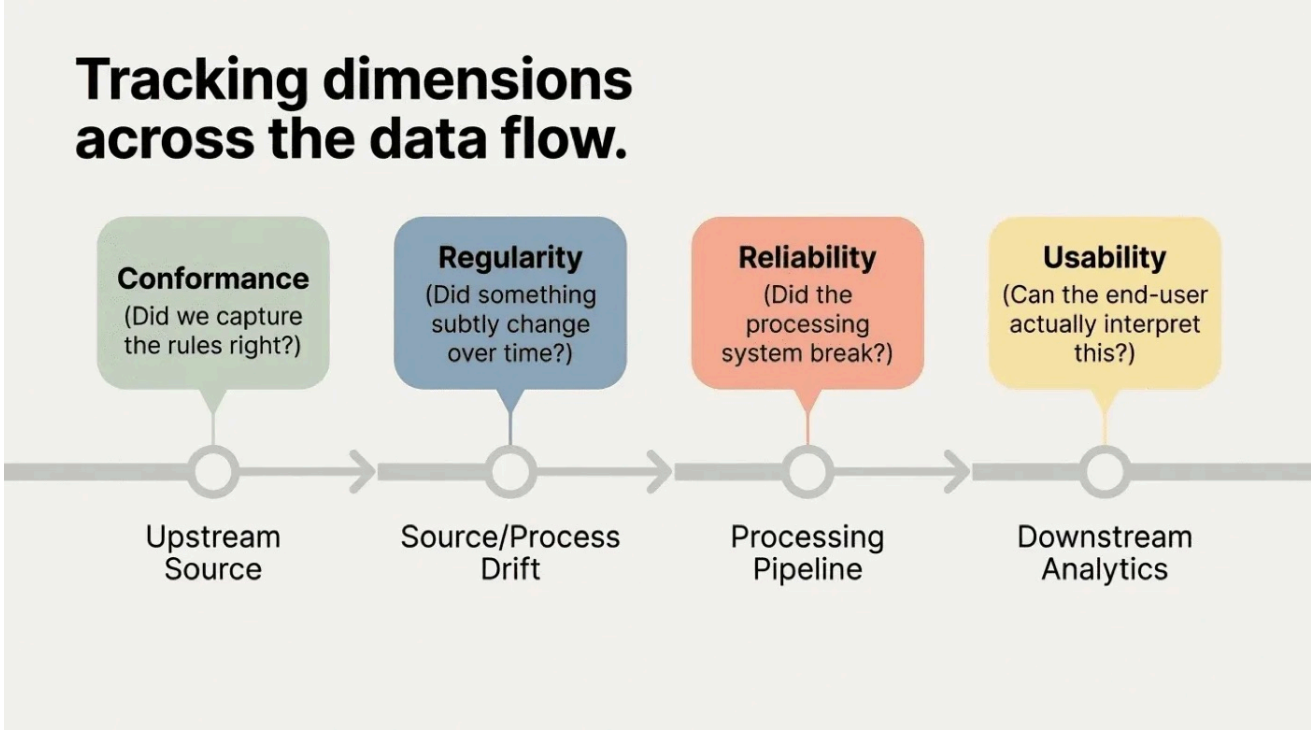
I hereby propose another way of organizing DQ problems, one that is surprisingly useful because it's all about relevance to you. We're calling it the Impact Dimension. It's useful when you're working with resource constraints (and who isn't?). Most useful of all, it offers some telling insights on the mistakes we all make when prioritizing problems by risk.



The four Impact Dimensions focus on four chronic problem areas that data issues signify. They turn your data quality testing into a diagnostic tool that shows the real impact of a problem.

- **Reliability**: Is the pipeline delivering? This would include tests for stale tables, volume drops, missing data, and schema changes. Reliability is most directly under the control of the data team, the most urgent dimension, and the most likely to receive attention.
- **Conformance**: Does the data follow known rules? This would include acceptable values, allowable numeric ranges, required values and acceptable formats. Conformance issues are also urgent, with clear feedback loops from downstream consumers when failures occur. They typically have clear fixes once a problem is discovered, but may require collaboration with upstream teams to solve.
- **Regularity**: Does data behave normally? Have average values shifted, has variability changed, are there unusual counts of outliers or shifts in missing value counts? These tests are more ambiguous. They are signals rather than rules. They require investigation, but can be sensitive to unanticipated, hidden problems that could otherwise have a significant impact.
- **Usability**: Does data presentation follow expected standards? Can users consume it without cleanup, wrangling, or misinterpretation errors? Usability issues manifest as divergent casing, embedded quotes, mixed data types, or technically accurate but inconsistent representations. Usability issues may not be errors themselves, but they're pernicious because they cause errors and inefficiency downstream.

Build data quality tests for each of these categories and you can see where quality improvements are needed, and which stakeholder teams have to bring them about. TestGen generates its tests per Impact Dimension, but even if you roll your own, the results show the DQ challenges specific to your organization.



One useful application: You can track where along the data flow each dimension's problems tend to surface. Conformance issues are tied to upstream source data, the focus of most traditional data quality efforts. Reliability issues relate to the data processing pipeline. Regularity issues may signal source or process problems, or even legitimate data drift that can break a downstream analysis. Usability issues impact downstream analytics and results.

In several organizations we work with, we've found that Reliability and Conformance issues get the most love. Reliability issues have the shortest feedback loop, and the big fails can be highly visible. From a systems perspective, they're almost self-correcting. Conformance issues draw significant attention from data consumers, who demand fixes, but the process can be slow and painful, and it damages trust in data teams and resources.

The Visible Crisis vs. The Hidden Drain

Funded & Focused

Reliability: Highly visible fails. Shortest feedback loop. Almost self-correcting.

Conformance: Loud downstream complaints. Clear fixes, though collaboration with upstream teams can be painful and damage trust.

Hidden Costs

Regularity: Signals, not rules. Hard to verify. AI/statistical advantages are lost because anomalies are easier to ignore.

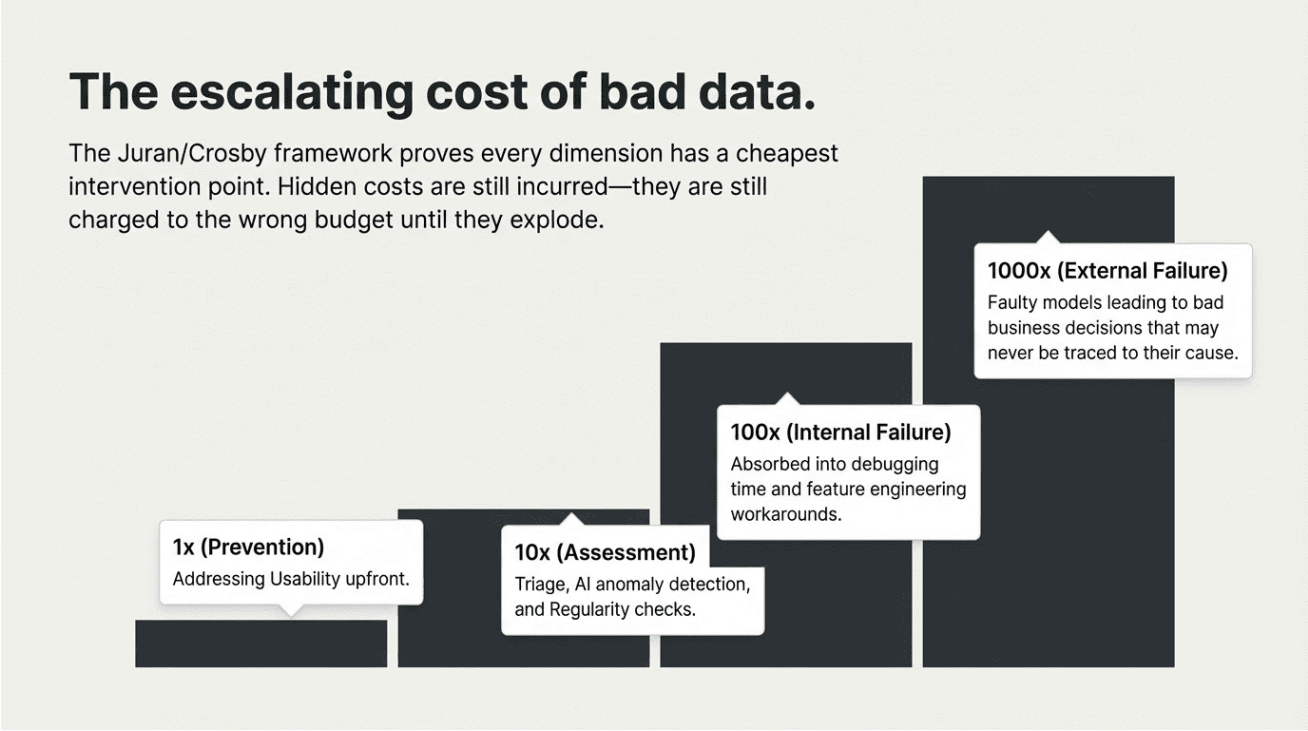
Usability: Rarely considered data problems. No feedback loop. Analysts quietly absorb the pain into their data wrangling time.

Sophisticated AI and statistical tools exist to uncover Regularity issues, but these results are harder to verify or even explain to non-experts. The result? You lose that advantage, because those results are easier to ignore. Usability issues are often not considered data problems at all. They don't directly impact data engineering teams, and downstream analysts typically consider data cleansing and wrangling part of their jobs. No effective feedback loop even exists to highlight and address these problems. They're rarely corrected.

Now consider the cost of these issues to your organization. The cost of Reliability and Conformance issues is well-understood, attributed, and funded. Not so for Regularity or Usability issues, which may just be rolled into the cost of analysis, racked up to debugging time, or feature engineering. Hidden costs are real costs. They're still incurred, just charged to the wrong budget.

The greatest cost of all comes when these issues are caught too late or missed entirely. Bad analyses get multiplied by faulty models and data products, and they produce bad decisions that nobody traces back to the cause.

How do you measure the cost of intervention at each stage? The Juran/Crosby framework posits that every dimension has a cheapest intervention point. In Juran/Crosby, if you peg the cost of prevention at 1x, the cost of assessment is 10x, the cost of internal failure is 100x, and the cost of external failure is 1000x. The implications are revealing.



For instance, there's a lot of new tech for identifying Regularity issues out there. TestGen runs anomaly detection, using AI and statistical thresholds to flag a range of potential issues. This is exciting, because Regularity tests are one of the few ways to guard against the unknown unknowns in a data process.

You can write a great Conformance test if you know exactly what you're looking for. But today's data moves at business speed, and static standards can't always keep pace with developing needs and shifting requirements. Data Quality means managing risk, not just enforcing rules. Regularity tests are so valuable because they guard against surprises: hidden costs that eventually explode in your face.

The problem, once you've uncovered a signal, is that you have to make a very visible investment in the triage to distinguish real problems from false positives. Making the case for resources to follow up is hard. The Regularity Dimension gives these issues a name, allows you to track them, and helps companies make these resource allocation decisions with open eyes.

Another great example: Usability issues are ubiquitous. But managers typically deprioritize them as trivial, easy targets for end-user workarounds. From a cost perspective, the fixes are so obvious, and the cost of prevention is so minimal compared to the cost of downstream failure, that it's a crime these issues aren't addressed upfront.

Rethinking Risk: The FMEA Reality Check.

In manufacturing (FMEA), **Risk Priority = Severity × Frequency × Detection.**
When applied to data quality, conventional wisdom is completely inverted.

	Severity	Frequency	Likelihood of Detection	Risk Priority
Reliability	High	Low	Easy	**Low**
Conformance	Med	High	Easy	**Medium**
Regularity	High	Med	Tricky	**High**
Usability	Low	Pervasive	Ignored	**HIGHEST**

You can directly quantify this. In manufacturing, the FMEA quality framework calculates a risk priority by multiplying severity by frequency by the likelihood of detection. Using our Impact Dimensions, we can apply this to data quality. Reliability issues actually score low: severity is high, but frequency is relatively low and problems are easier to detect. Conformance issues score a little higher: severity isn't necessarily as high as a failed pipeline, and detection should be easy, but the frequency of problems is higher. Regularity issues may be less frequent, but severe when they occur, and they end up scoring higher because they're so tricky to detect.

The Usability Paradox.

Managers consistently deprioritize usability (divergent casing, mixed data types) as trivial, easy targets for end-user workarounds.

Why it scores HIGHEST in risk:

Not because any single issue is catastrophic, but because they are pervasive, completely undetected, and historically ignored.

The true cost:

The cost of prevention is minimal. The compounding wage-cost of downstream workarounds and the ultimate cost of failure is massive. Ignoring usability is a preventable crime against your budget.

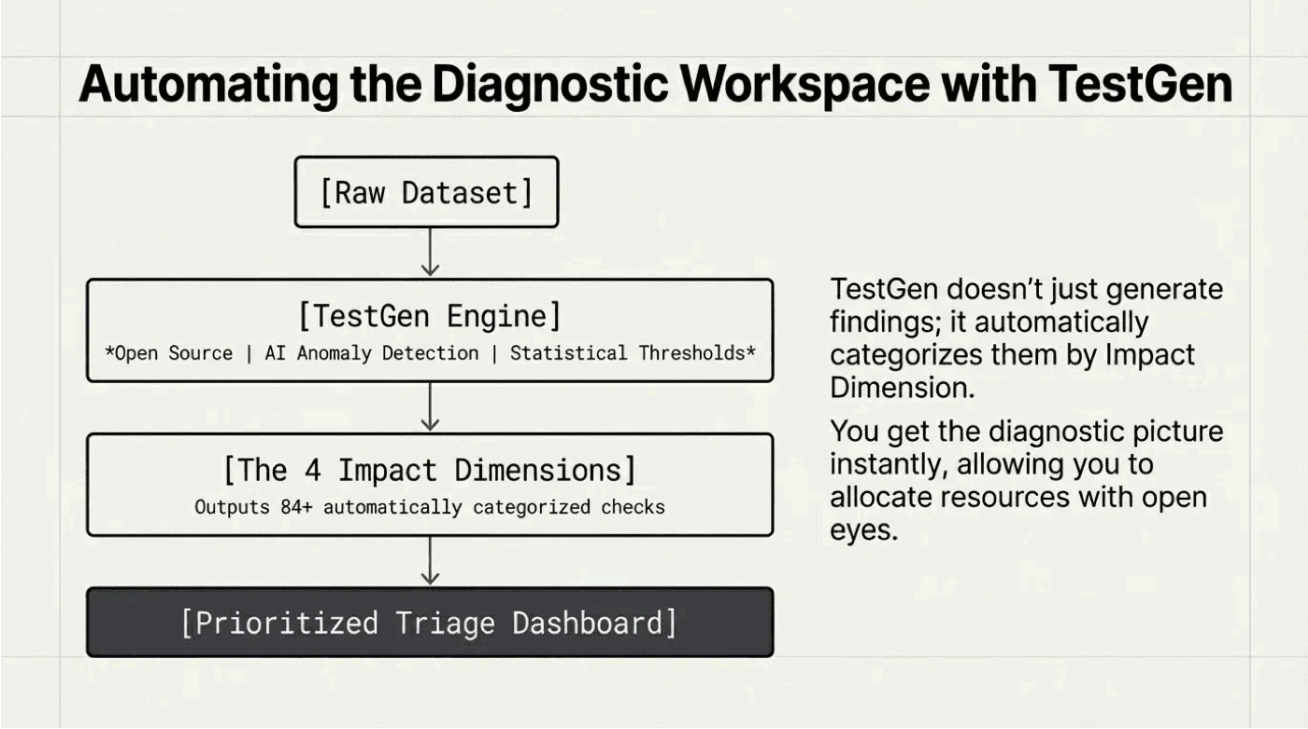
Surprisingly, the highest Risk Priority score may belong to Usability issues, not because any single issue is severe, but because Usability issues are pervasive and typically left undetected and ignored. Considering the impact on the wage-cost of workarounds if they're identified and the cost of failure if they're not, again, it's a wonder these issues are so often trivialized.

✅ Prioritize impact, not noise.

Organizing by Impact Dimensions turns your testing into a true diagnostic tool. It reveals the hidden costs lurking in Regularity and Usability, moving your team away from simply counting defects.

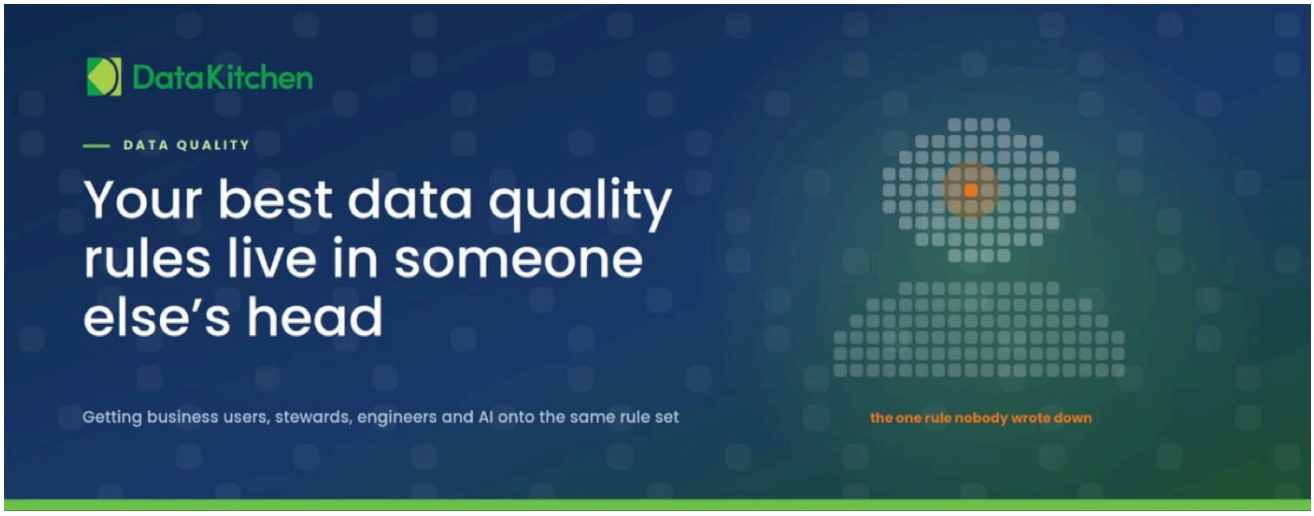
🎉 *It won't make **data quality** any more **interesting** to talk about at parties. But saving your **company a boatload of money** and **earning a promotion** just might make your boring neighbor pause and listen.*

The bad news is that none of this will make data quality any more interesting to bring up at a party. But if you happen to mention how you re-prioritized some of your team's most pernicious problems, saved your company a boatload of money, and earned yourself a plum of a promotion, even your pesky neighbor might take a breath and listen.



Your Best Data Quality Rules Live in Someone Else's Head

Read the original article at: https://datakitchen.io/blog/best-data-quality-rules-live-in-someone-elses-head/



An analyst on the finance team knows that when the active member count drops while claim volume holds flat, the eligibility file truncated. She has known this for six years. She checks the two numbers against each other by hand every month before the close.

Your data quality tests do not check it.

They cannot. The total member count fell 3 percent, which is under every threshold you have. The entire drop sits in one plan that lost 60 percent of its members, and nobody tests at the plan level. Both tables pass. Nothing is null, nothing is duplicated, no schema changed. The failure lives in the relationship between two feeds and inside a segment nobody thought to slice. Both of those facts exist only in her head.

Your tests check nulls. They check uniqueness on the primary key. They check that the row count did not drop more than 10 percent overnight. Those tests are real and they catch real problems. They also came out of a table profile, not out of anyone's head. A profiler reads the data. It cannot read the business.

Generic tests catch broken pipelines. Business rules catch broken business. Statistical detection catches what nobody thought to check. You need all three, and you have one.

Asking for requirements does not work

The standard fix is to go ask the business. Send a spreadsheet. Ask each domain owner to list their data quality rules and return it by Friday.

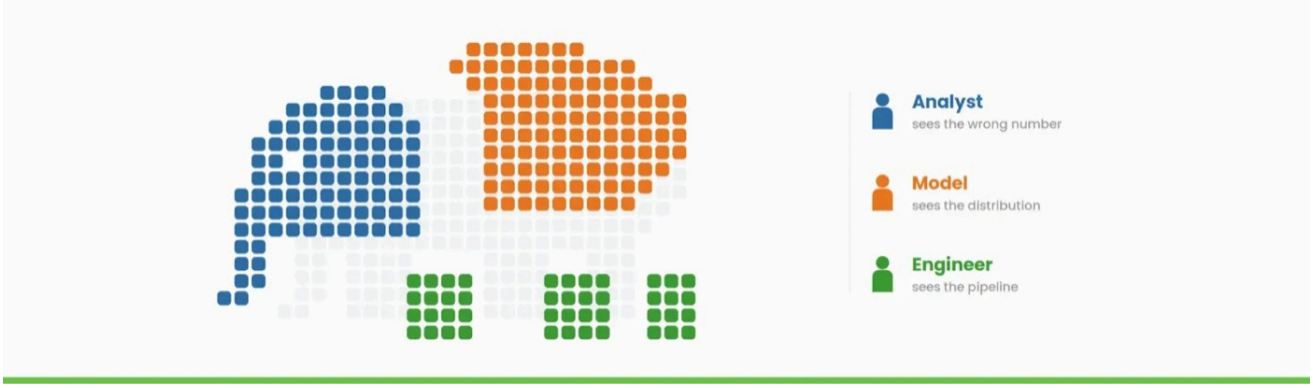
You will get four rows back. Two of them will say “data should be accurate.”

This is not because business users are lazy. It is because nobody is good at authoring rules from a blank page. The analyst does not carry a list of rules in her head. She carries a reflex. She looks at a report, something is off, and she knows why within five seconds.

Reacting is easy. Authoring is hard. So stop asking people to author.

Show someone a rule that already exists and they will tell you immediately whether it is right. Show them a threshold of 5 percent and they will say the number should be 1 percent, because anything above that means the vendor file arrived twice.

You get better rules in a 20-minute review than in a six-week requirements gathering exercise.



Nobody has the whole elephant

Three people look at the same broken table and describe three different animals.

The engineer touches the pipeline. He knows the load ran at 2:14am, he knows the join fans out, and he knows the table came in at 4 million rows where it usually lands at 4.1 million. He does not know that a duplicate contract ID means somebody re-keyed a renewal instead of amending the original, and that

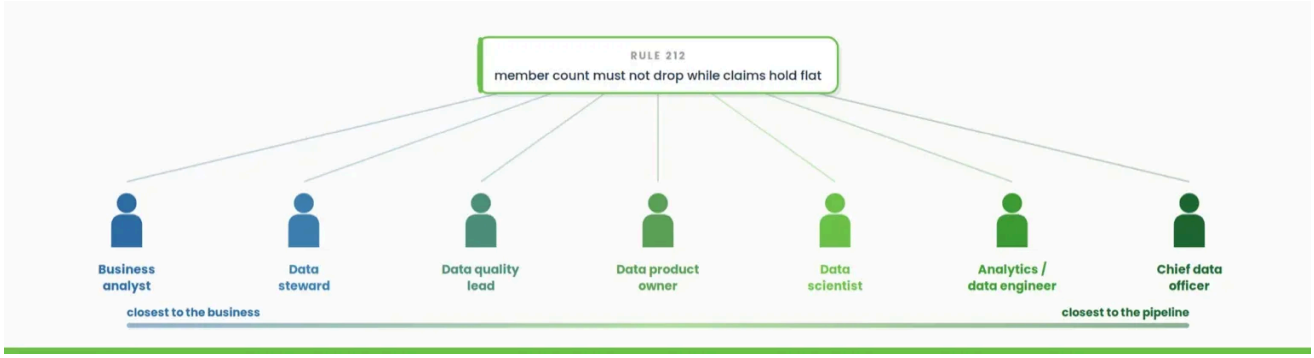
finance will restate a number because of it.

The analyst touches the report. She knows the renewal number is wrong within five seconds of opening the dashboard. She cannot tell you that the merge statement matched on a nullable column. Ask her to write the fix and she will describe the symptom, correctly, in language nobody can turn into SQL.

The model touches the data. It sees that the column is 3 percent null this week and was 0.2 percent last week, and it will write you a clean test for that in one shot. It does not know whether anyone cares. It does not know that rule 212 already covers this table, that the last three times this fired it was the same vendor sending a duplicate file, or that the team decided in March to stop testing this column at all.

Each one is right about the part they are holding. None of them can describe the animal.

The usual answer is a meeting. Get everyone in a room, walk the rules, agree on the fixes. That works once. Then the meeting ends, the context goes back to living in three heads and a set of notes, and six weeks later you hold the meeting again with two new people who were not there the first time.



Who touches a data quality rule

The cast is also bigger than three, and it does not sort neatly into authors and readers.

Your business analyst knows what wrong looks like. She has seen the specific failure, in the specific column, on the report a VP opened at the wrong moment. She will never write SQL and she will absolutely tell you a threshold is wrong. Your data steward owns the definition. When sales and finance disagree about what counts as an active customer, the steward decides and the rule reflects that decision. The steward also retires rules that stopped applying two reorgs ago, which matters more than it sounds.

Your data quality lead lives in this all day. She wants to know what changed since last week and which failures are still open, and in most shops she is technical enough to write the rule herself, so give her the write path instead of making her file a ticket. Your data engineer implements and runs. He wires the rule into the pipeline, sets the schedule, decides whether a failure stops the load or just records itself, and owns the ten minutes after it fires. Your analytics engineer has a different title and the same job.

Your data product owner wants her tables and nothing else, because she is deciding whether she can promise a freshness commitment to the teams consuming her product. She will add a rule the moment she has to make that promise in writing. Your data scientist wants to know whether the features feeding

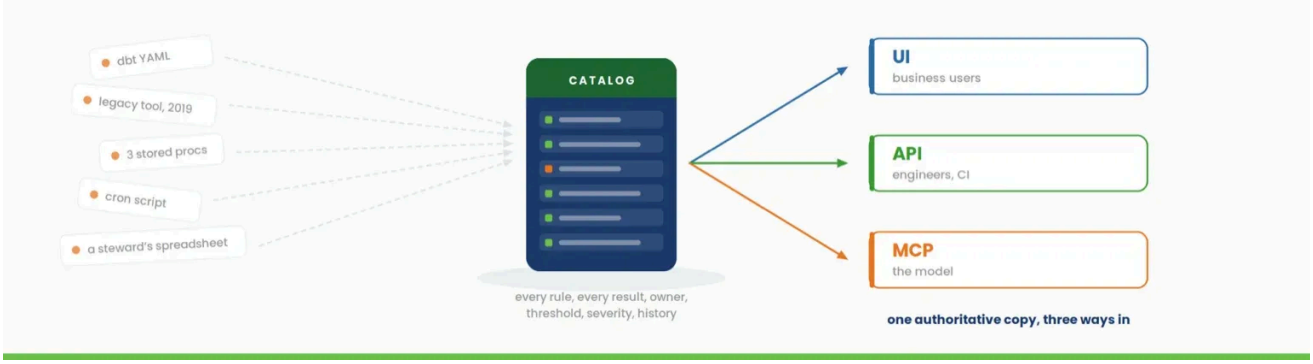
his model were clean on the dates he trained on. That is a question about history, not about this morning, and today he cannot answer it at all. He finds out the model drifted, then spends three weeks reconstructing what March looked like. After that happens once, he writes his own tests.

Your chief data officer asks the broadest question and the one nobody can answer. How many tables do we test. How many rules do we run. How many fired last month, how many turned out to be real, and is that number moving the right way. Every quarter somebody hand-builds her a spreadsheet that goes stale before the meeting ends.

The line between the people who change rules and the people who only look at them does not follow job titles. It follows how technical someone is and what happened to them that week. The analyst who has never edited anything will edit a threshold the morning after a bad close. Build for that. A tool that ships a business view and an engineer view as two separate products has already made the wrong bet, because it froze a boundary that moves.

What these people share is the need to see the same thing. State, status, type, count, and whether the rule is still correct. Six people assembling that picture by hand, on different days, from different sources, will not agree on how many tests you run. That is the argument for one store, and it is a better argument than collaboration.

Most organizations skip all of this and hand the whole job to the engineer. He ends up guessing at business logic he does not know, defending thresholds he picked because they were round numbers, and fielding Slack messages from analysts who found a problem his tests missed.



One database of record for rules and results

Ask your team a simple question. How many data quality rules do you run?

Nobody will know. The rules are in a dbt YAML file, plus a legacy tool somebody bought in 2019, plus three stored procedures, plus a Python script on a cron job that a contractor wrote and never documented, plus a spreadsheet a steward keeps on her desktop.

Because the rules are scattered, you cannot answer the questions that matter. Which tables have no coverage at all? Who owns this rule? Why is the threshold 5 percent? When did this rule last pass? Did anyone review it after we changed the source system?

Put every rule in one place. Keep the plain language description, the table and column it runs against, the owner, the threshold and the reasoning behind it, the run history, and the current state. Make it readable by people without a Git account.

Keep the results next to the rules. Not in a separate logging system, not in a Slack channel scrollback that ages out after 90 days. Every run, every failure, the rows that failed, who looked at it, and what they decided. The rule list tells you what you check. The result history tells you what actually breaks, and those are two different lists. A rule that has failed 40 times this year and gets waved through every time is not a rule. It is a documented tolerance, and somebody should either fix the source, move the threshold, or admit it was never a hard failure.

That last one is how test suites die. A rule written for a discontinued product line keeps firing every night. Somebody mutes the alert. Then somebody mutes the channel. Six months later, a real failure lands in a channel nobody reads. Suites do not die from missing rules. They die from the noisy ones nobody owns.

Severity is the mechanism people skip. Every rule gets a tier. A hard fail stops the load and wakes somebody up, and you should have very few of those. A soft fail lets the data through and posts a warning to the rule's owner. An informational check records the number and says nothing, so you can look at the trend in six months. Most of what a profiler writes for you on day one is soft-fail material. Tag every rule when you create it, revisit the tag when it misbehaves, and demote the ones that fire nightly instead of muting them. That is the whole difference between a suite people trust and a channel people filter.

Then review on a schedule. Once a quarter, put the list in front of the people who use the data and ask which rules still matter. Retire the dead ones. Tighten the loose ones. Add the two or three that came out of last quarter's incidents.

Your data quality and test tool should open three doors onto that database. The business user gets a UI she can search, read, and comment in without a Git account. The engineer gets an API and version control, so rules move through CI like any other code. The model gets an MCP interface. MCP is the Model Context Protocol, an open standard for letting an assistant read a system's data directly. Give the model one, and it can read the rule definitions, the profiles, and the full result history before it opens its mouth.

IMPORTANT

Decide which door is authoritative before you open any of them. Two writable copies of the rule set are the same disease you started with, and it is worse this time because both copies look official.

The pattern that works: the database holds the live rule set, and every change is exported to Git, so you get diffs, review, and rollback. A business user's edit is a proposal until the steward approves it. The pattern that fails: rules in Git, rules in a UI, both writable, syncing on a schedule. That is how a threshold reverts every Wednesday, and nobody can explain why.

The rules nobody wrote

Everything so far assumes a human decided what to check. That covers what somebody thought of, which is not the same as what breaks.

Statistical detection covers the rest. Point a profiler at a column, let it learn the shape over 90 days, and it flags the week that does not look like the last twelve. Distribution shifts. Cardinality jumps. A field that ran 2 percent null for a year and came in at 9 percent this morning. Nobody wrote those rules, and nobody was going to. You have 4,000 columns in that warehouse, and no team is hand-authoring a distribution check on each one.

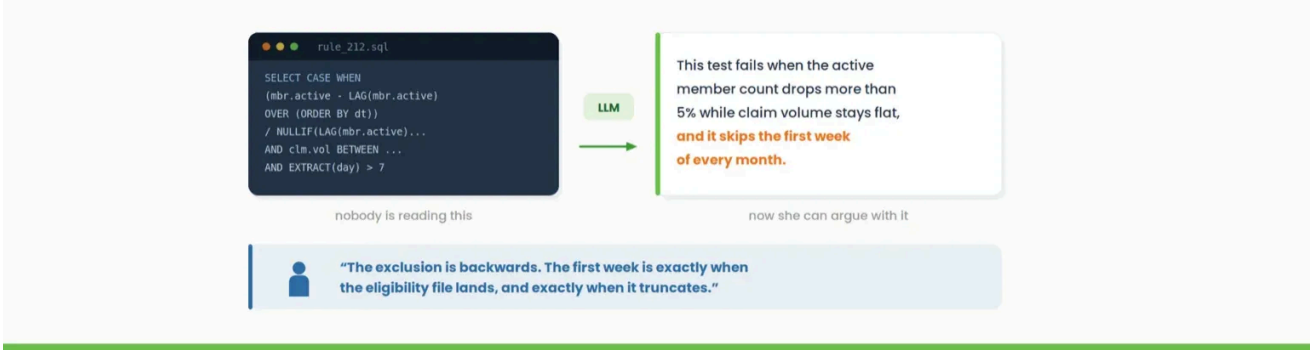
Seasonal baselines decide whether anyone keeps it turned on. Volume drops on weekends. Claims spike at month-end. The last week of the quarter looks nothing like the first. A detector that has not learned your calendar fires every Saturday and every close, and your team filters the channel inside of a month. A detector that has learned it stays quiet on Saturday and speaks up on the Saturday that is actually wrong.

Here is the part vendors skip. Learned rules need more human attention than authored ones, not less.

An authored rule arrives with a reason. Somebody wrote it because a specific thing broke, and they can tell you what. A learned rule arrives with a z-score and no story. “Distribution of plan_code shifted by 3.4 standard deviations.” Your engineer cannot tell you whether that matters. Neither can the model. Your analyst can, in about five seconds, because she knows the new plan launched on the first.

So learned rules get the same treatment as everything else. An owner, a severity tag, a line in the catalog, and a slot on the quarterly review. Two things differ. A learned rule starts as a soft fail, always, because you do not stop a production load on a statistical hunch. And the first review question changes. For an authored rule, you ask whether it is right. For a learned rule, you ask whether it is real.

That question is the whole loop. Detection finds something once, a human names it, and it becomes a rule with an owner and a reason attached. The plan_code shift gets looked at, turns out to be the eligibility file dropping a plan, and now you have a named rule that fires on that exact failure and explains itself. The detector found it. The analyst made it permanent. Do that 20 times, and your rule set knows things nobody ever sat down to write.



Point an LLM at the rules

Half the reason business users never review data quality rules is that the rules are written in SQL. Nobody is going to read a case statement with four nested joins and tell you whether the logic matches how sales defines a closed deal.

A model translates it. Point it at the rule, and it writes the sentence. “This test fails when the active member count drops more than 5 percent while claim volume stays flat, and it skips the first week of every month.” Now the analyst has something to argue with. She reads the second half of that sentence and tells you the exclusion is backward because the first week of the month is exactly when the eligibility file lands and exactly when it truncates.

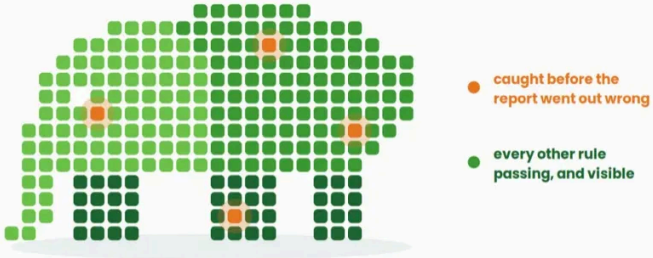
That translation step is small, and it is the whole bridge. It turns a review meeting nobody could hold into one that takes 20 minutes.

The same trick works on the rule set as a whole. Point a model at all 400 rules in your catalog and ask which tables have no coverage, which rules duplicate each other, and which ones have not failed in two years. You get a review agenda in a few minutes. Go the other direction too. Paste in last quarter’s incident write-up and ask for the test that would have caught it. Most teams never write that test because the incident ends, everyone is tired, and the retro action item dies in a ticket.

Root cause analysis is another place this pays. When a test fails at 3am, the first 20 minutes are mechanical. Pull the failing rows. Check what changed. Compare this run against the last 30. Give a model your test definitions, your table profiles, and your full result history, and it comes back with a first guess and the evidence behind it. “Row count dropped 12 percent. The entire drop is in the EMEA plan segment. Rule 212 covers this table but only tests the total, which is why it passed last night.” Your engineer confirms or rejects in two minutes instead of digging for an hour. Your data quality and test tool should feed all of that directly to a model, via MCP or an API, so the assistant reads real definitions and real results instead of guessing from whatever you pasted into a chat window.

Know where this stops. The model gives you a hypothesis, not a verdict, and it will be confident and wrong sometimes. Your test results say what broke and where. They say nothing about why the upstream job ran late or which commit changed the transform, because that lives in Airflow or Databricks or Git, and you have to hand it to those systems too if you want it to reach that far. It does not know that the EMEA

feed always runs light on the last business day of the quarter, and it will happily write you a rule that enforces a business definition nobody agreed to. A human signs off on every rule change. What you get is a narrowed search, not an answer.



The whole animal

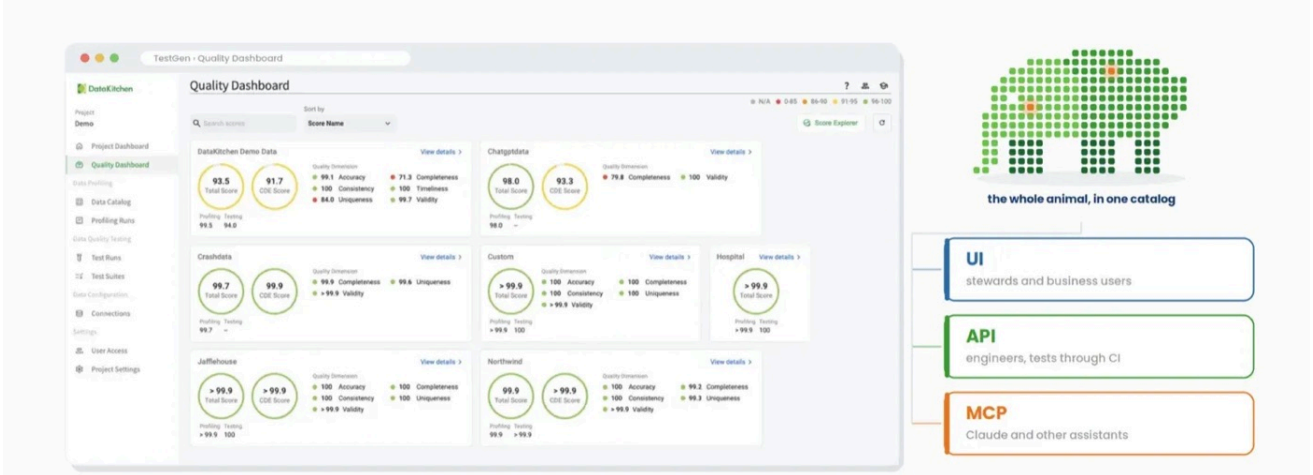
Nobody on your team is going to wake up holding the entire data estate. Your engineer will keep not knowing what a duplicate contract ID does to the quarterly close. Your analyst will keep not knowing what a merge statement does to a nullable column. The model will keep not knowing what anyone cares about. That is not a training problem, and no amount of documentation solves it.

What you can fix is where the knowledge lands. One catalog holds every rule and every result, including the ones the profiler learned on its own. Generate the first pass automatically, because if your tool makes you hand-author the first hundred tests, you will never get past table one. Give the business user a UI, the engineer an API, and the model an MCP interface, and make one of them authoritative so the copies never fight. Tier the failures so people trust the alerts. Review the list once a quarter with the people who use the data, and name the anomalies that turned out to be real.

Then route each failure to whoever asked for the rule, with the failing rows attached, wherever she already reads. Not a shared channel with 40 people in it. Not a dashboard she has to remember to open. If she has to go look, she is still checking by hand, and you have bought nothing.

None of that is exotic. Teams skip it because rules feel like plumbing, and plumbing never gets a budget until it floods. Then the eligibility file truncates in one plan, the number goes out wrong, and somebody spends a quarter rebuilding trust that a 20-minute review would have protected.

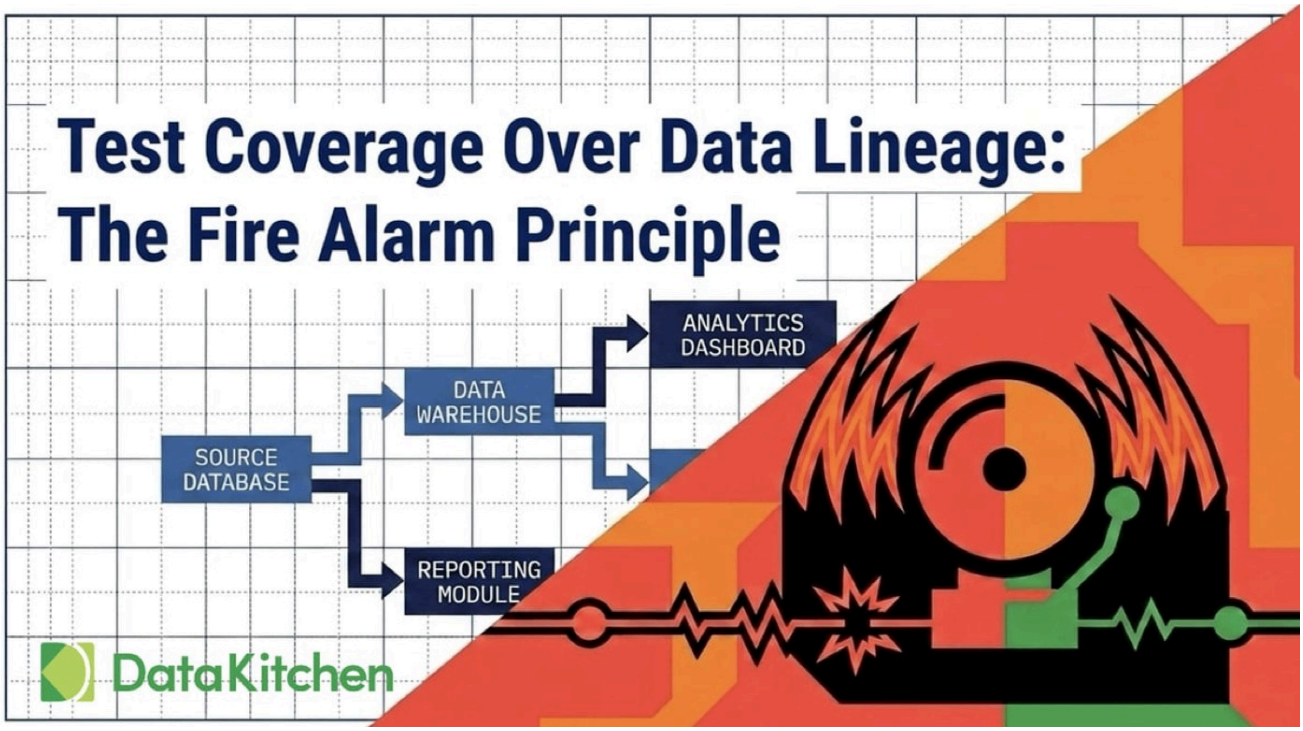
The analyst who reconciles those two numbers by hand every month stops doing that. The check runs at 3am instead, and it fails the load before the report is wrong.



Every rule and every result in one place, readable without a Git account.

The Blueprint or the Fire Alarm Panel: Why Data Test Coverage Beats Data Lineage

Read the original article at: https://datakitchen.io/blog/test-coverage-beats-data-lineage/

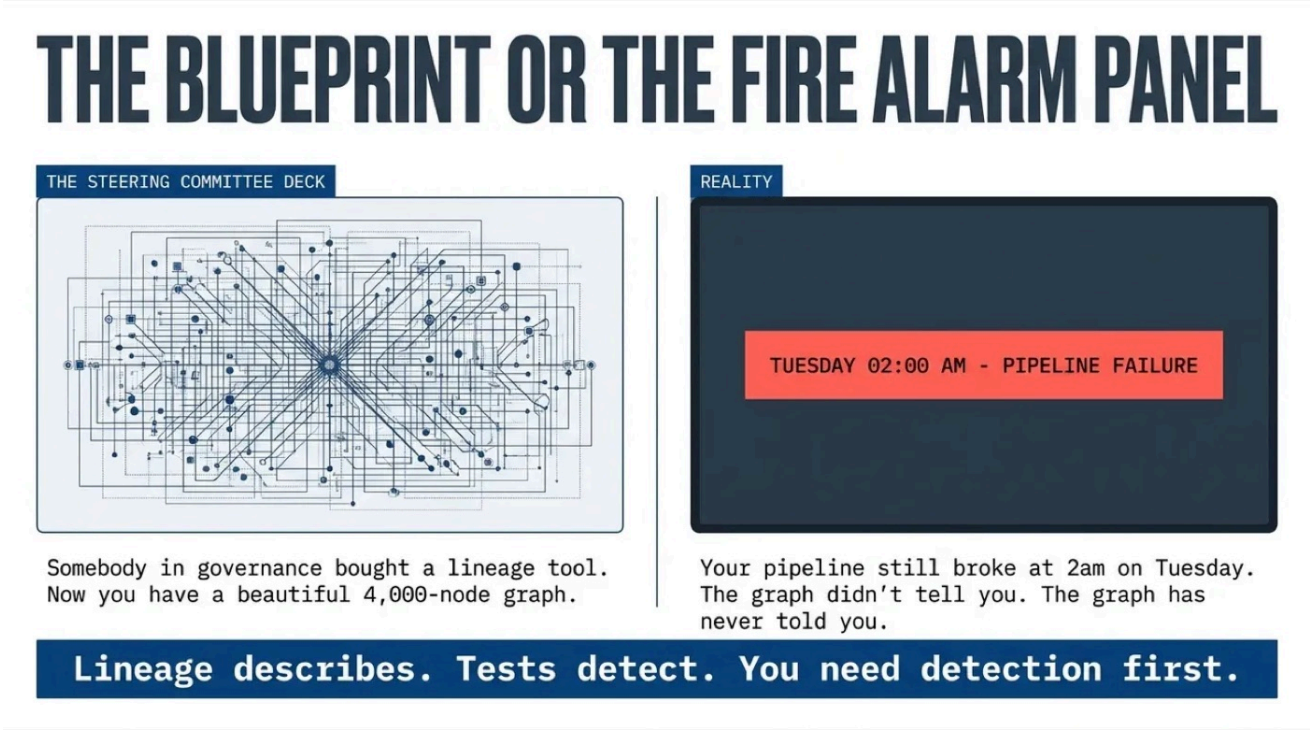


Somebody in governance bought a lineage tool. Now you have a graph. It has 4,000 nodes, the lines glow when you hover over them, and it looks fantastic in a steering committee deck.

Your pipeline still broke at 2am on Tuesday. The graph didn't tell you. The graph has never told you.

I've been building data and analytics systems for close to 30 years. I have never once fixed a production data problem by looking at a lineage diagram. Not one time. I've looked at plenty of them. When the CMO's sales number was wrong on a Monday morning, the lineage graph sat there as decoration while someone on my team ran SQL by hand to find the bad rows.

That's the whole argument of this post. Lineage describes. Tests detect. You need detection first.

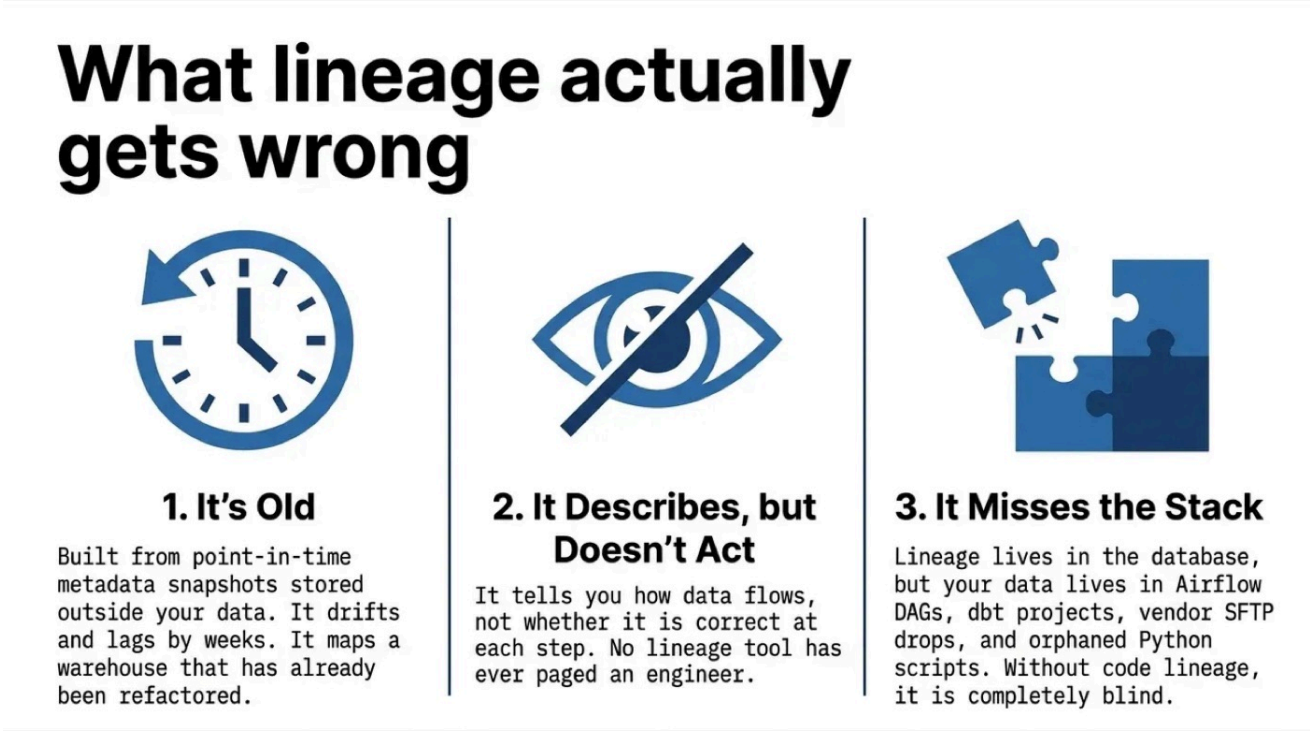


What lineage actually gets wrong

It's old. Lineage gets built from snapshots of metadata taken at a point in time and stored in a system that isn't your data. It drifts. It lags by weeks, sometimes months. The graph you're looking at describes a warehouse that's been refactored twice since the last scan.

It describes, and it doesn't act. Lineage answers where data came from and where it goes. It doesn't answer the question you actually have at 2am, which is whether the number is right and which step broke it. As we put it in Data Quality Testing Is At The Core of Four Critical Data Team Processes, lineage tells you how data flows, not whether the data is correct at each step. No lineage tool has ever paged you.

It misses most of your stack. Lineage lives in the database. Your data doesn't. Your data goes through an Airflow DAG, a dbt project, a Python script somebody wrote in 2019 before leaving the company, a vendor SFTP drop, a Tableau extract, and a reverse ETL job pushing back into Salesforce. Chad Sanderson makes this point in his Shift Left Data Manifesto: without code lineage, your data lineage is blind outside the warehouse. Most of your failures live in exactly that blind spot.



Where lineage earns its keep

Trust comes from verification first and root cause second. Lineage is a root cause tool. It's a good one.

When legal asks where customer PII lives, lineage answers that. When an auditor asks how a regulatory figure got derived, lineage answers that. When a new engineer joins and needs to understand how 300 tables relate, lineage plus a data catalog beats reading DDL for a week. And once you already know a number is wrong, lineage helps you walk back to where it went wrong.

Notice the order. Every one of those uses starts after somebody already knows something is broken. Lineage is the second thing you reach for. Most teams have bought it first.

The firefighter question

A firefighter walks into your building. Which do you hand them: the blueprint or the fire alarm control panel?

The blueprint tells them where fire could spread. The panel tells them which room is burning right now. They'll take the panel. They'll look at the blueprint later, once the fire is out and somebody wants to know how it got from the kitchen to the third floor.

Lineage is your blueprint. Tests and monitors are your fire alarm panel.

The firefighter question

Which do you hand a first responder: the blueprint or the fire alarm panel?

	The Blueprint	The Fire Alarm
Function	Maps where fire could spread	Tells you which room is burning now
Primary User	Auditor / Architect	First Responder / Engineer
Visibility	Structural connections	Real-time state and status
Data Equivalent	**Data Lineage**	**Test Coverage & Monitors**

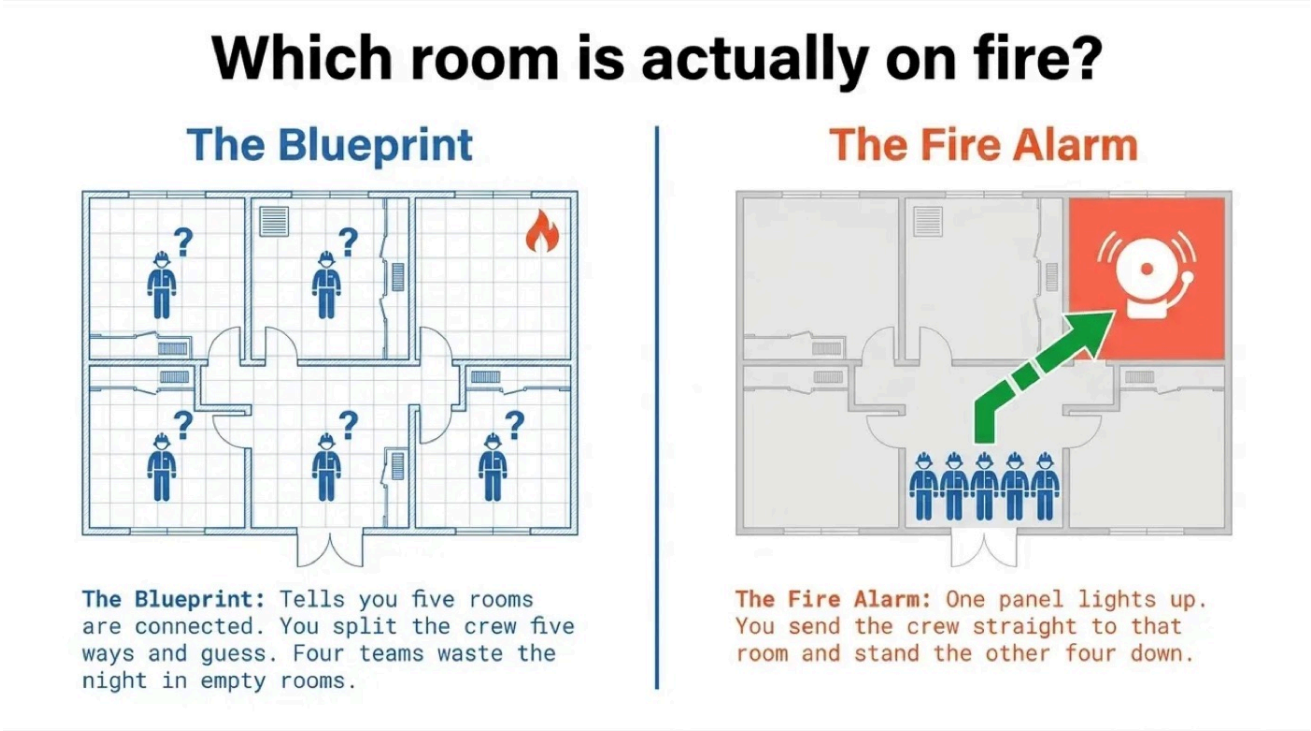
Which room is actually on fire?

A room is on fire. It connects to five others. Two share a door. One shares a ceiling vent. One is fed off the same service duct. One is two floors up on the same shaft.

You have the blueprint. It tells you all five rooms are connected. It does not tell you which one has smoke in it. So you split the crew five ways, and guess what? Four of those teams spend the night standing in a room where nothing is happening.

Now put a fire alarm in every room. One panel lights up. You send the crew to that room first and stand the other four down.

Same building. Same blueprint. Completely different night.



The same question in SQL

A source system in your APAC region starts sending `order_total` in cents instead of dollars. Nobody told you. The schema didn't change. The row counts didn't change. Every job in the chain succeeded.

`stg_orders` feeds five tables, and none of those paths look like another.

Path 1 is `dim_customer`. One hop, and it never touches the money column.

```
select
    customer_id,
    email,
    min(order_ts)::date as signup_date
from raw.stg_orders
group by customer_id, email;
```

Path 2 is `dim_product`, two dbt models deep.

```
-- models/int_product_orders.sql
select product_id, order_id
from raw.stg_orders;

-- models/dim_product.sql
select p.product_id, p.sku, p.category, count(o.order_id) as order_count
from raw.products p
left join {{ ref('int_product_orders') }} o using (product_id)
group by 1, 2, 3;
```

Path 3 is `fct_order_items`, three Airflow tasks away, and the ingest team owns it.

```
select o.order_id, l.item_id, l.qty
from raw.stg_orders o
join raw.order_lines l using (order_id);
```

Path 4 is `fct_order_status`, built by a Python job on its own cron that isn't in the DAG at all.

```
# status_sync.py, runs at 04:15, nobody on the call remembers who owns it
df = read_sql("select order_id, status, updated_at from raw.stg_orders")
write_table("analytics.fct_order_status", df)
```

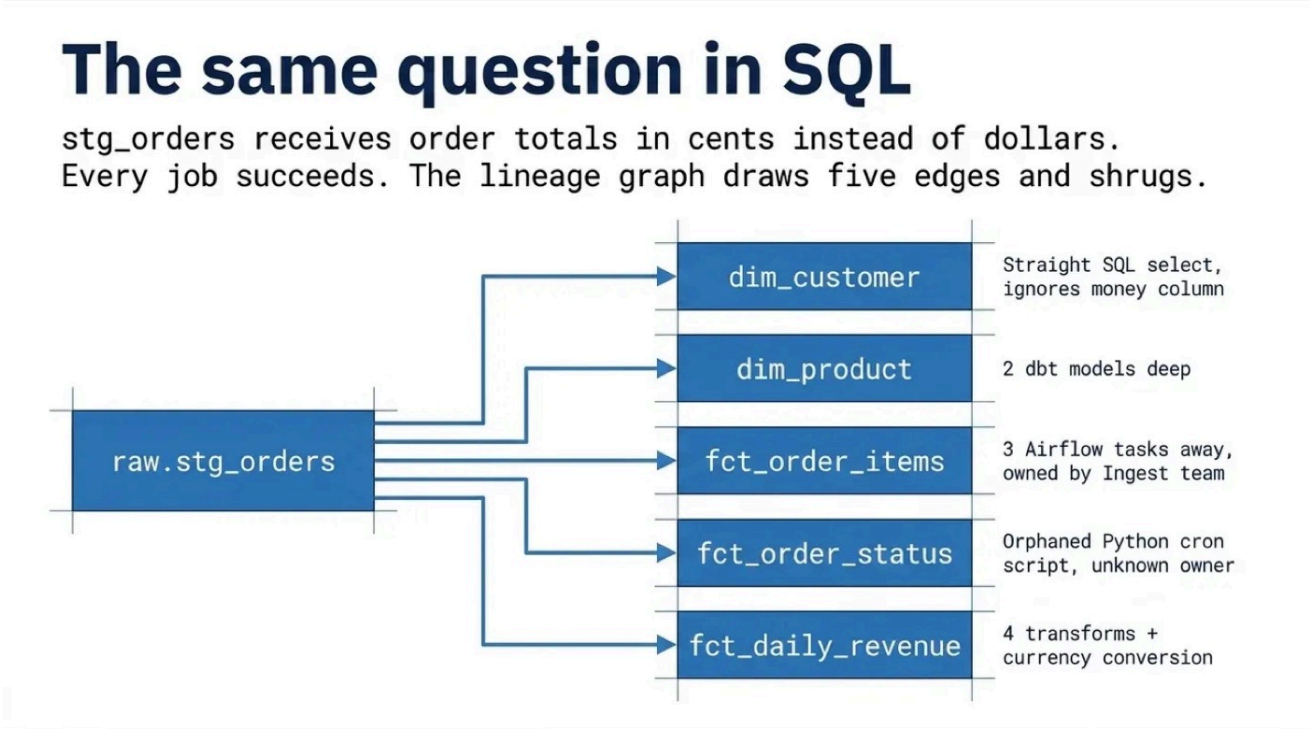
Path 5 is `fct_daily_revenue`, four transforms and a currency conversion away.

```
-- models/int_orders_clean.sql
select order_id, order_ts::date as order_date, region_code, currency_code, order_total
from raw.stg_orders
where status <> 'cancelled';

-- models/int_orders_usd.sql
select order_date, region_code, order_total * fx.rate_to_usd as order_total_usd
from {{ ref('int_orders_clean') }} o
join ref.fx_rates fx using (currency_code);

-- models/fct_daily_revenue.sql
select order_date, sum(order_total_usd) as order_total
from {{ ref('int_orders_usd') }}
group by order_date;
```

Those are your doors and your vents. One is a straight select. One belongs to a team you have to Slack. One runs on a cron nobody claims. And the last one takes a number that is now 100 times too big and multiplies it by the exchange rate.

Not one of those five steps threw an error. Every model built. Every task went green.

Your lineage graph draws an edge for all five and stops there. It won't tell you which of the five finished running this morning. It won't tell you who owns the job. And it won't tell you which one is wrong. Five names and a shrug.



Now put tests on all five. These are the kinds of tests TestGen creates:

```
dim_customer       row_count_vs_7d_avg      count(*) within 20% of the trailing 7-day
average
dim_product        referential_integrity    every product_id exists in raw.products
fct_order_items    qty_range                qty between 1 and 500
fct_order_status   freshness                max(updated_at) within 6 hours
fct_daily_revenue  revenue_vs_30d_avg       daily total within 3x the trailing 30-day
average
```

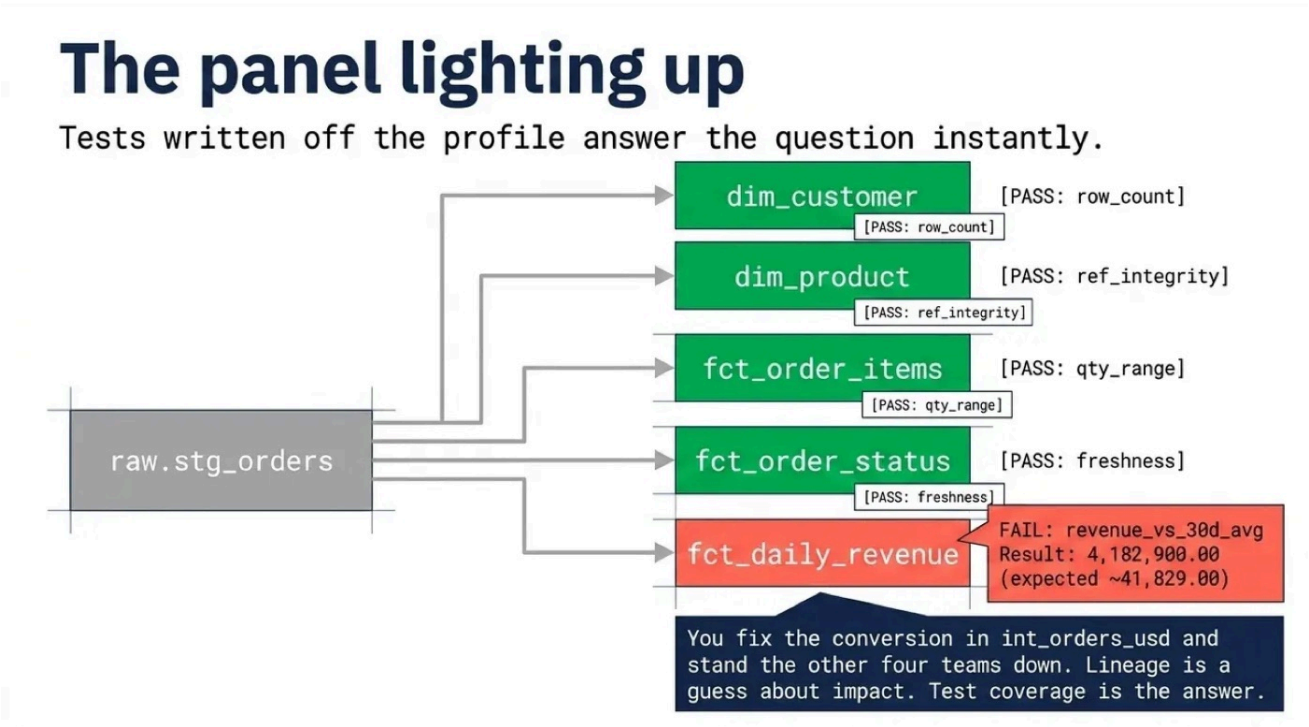
That last one is the alarm in the room that's burning:

```
-- fct_daily_revenue: revenue_vs_30d_avg
-- Returns rows = test fails.
select
    order_date,
    sum(order_total) as daily_revenue
from analytics.fct_daily_revenue
where order_date = current_date - 1
group by order_date
having sum(order_total) > 3 * (
    select avg(daily_revenue)
    from analytics.revenue_history
    where order_date between current_date - 31 and current_date - 1
);
```

Run the suite, and you get an answer instead of a map:

```
TEST RUN   2026-07-27 06:00 UTC

  dim_customer        row_count_vs_7d_avg      PASS
  dim_product         referential_integrity    PASS
  fct_order_items     qty_range                PASS
  fct_order_status    freshness                PASS
  fct_daily_revenue   revenue_vs_30d_avg       FAIL   4,182,900.00  expected ~41,829.00

  4 passed, 1 failed
```

That's the panel lighting up. You know which room is on fire. You go fix the conversion in `int_orders_usd`, and you tell the other four owners to stand down before they open their own investigations. With the blueprint alone, all five teams search their own tables until somebody gets lucky.



Lineage is a guess about impact. Test coverage is the answer.

Lineage is blind during development

Production isn't even where lineage disappoints you most. Development is.

In production, your code is static and your data changes. That's the Value Pipeline. In development, it's the reverse: your data is static and your code changes. That's the Innovation Pipeline, and it's the one lineage cannot see into at all, because lineage models tables and columns while your changes live in logic.

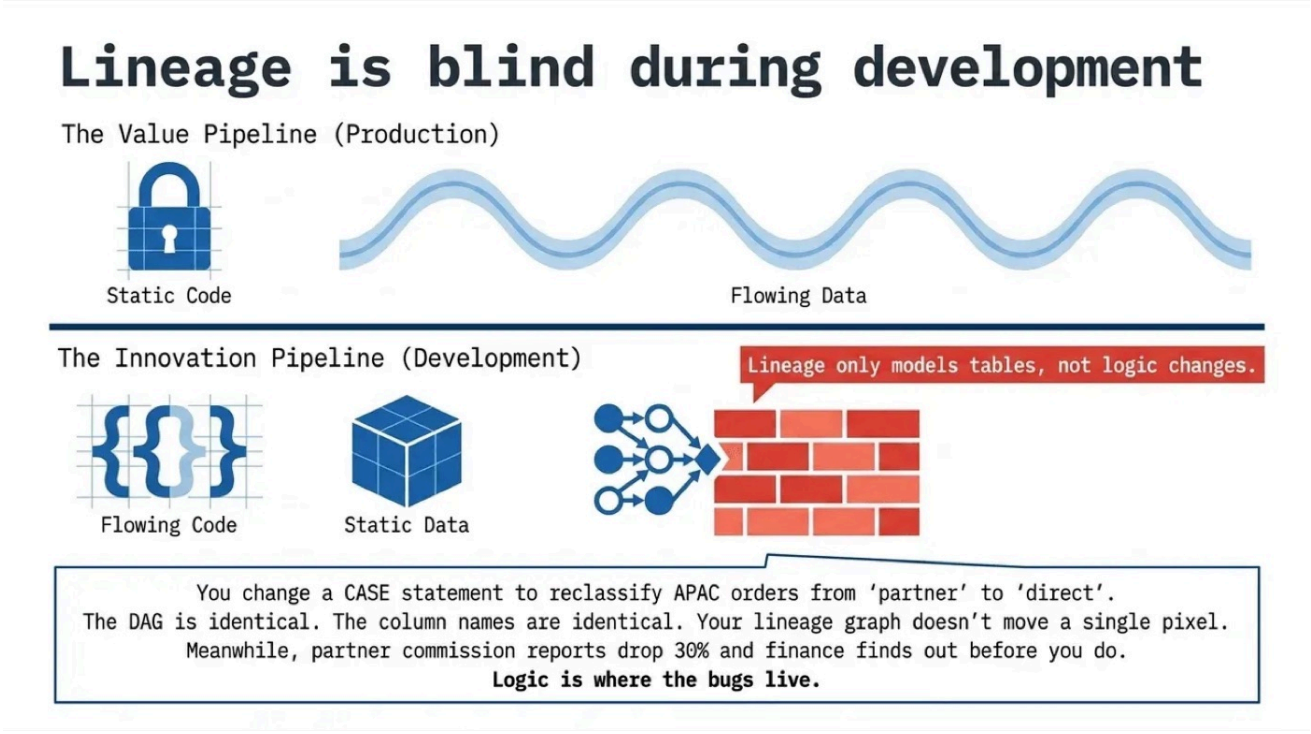
You change a `CASE` statement in a dbt model:

```
-- before
case
    when region_code in ('NA', 'EMEA') then 'direct'
    else 'partner'
end as channel

-- after
case
    when region_code in ('NA', 'EMEA', 'APAC') then 'direct'
    else 'partner'
end as channel
```

What does your lineage graph show? `stg_orders.region_code` feeds `fct_sales.channel`. Same edge before, same edge after. The DAG is identical. Column names are identical. Nothing in the graph moves.

Meanwhile, you just reclassified every APAC order from partner to direct. The partner commission report drops 30%. The sales comp file that goes out Friday is wrong. Finance will find it before you do.

Lineage models the shape of your system. It doesn't model the logic. Logic is where the bugs are. No software engineer ships to production based on an abstract dependency graph. They run the test suite. Data teams need the same discipline: real test data, real tools, real code, and automated tests that run on every change.



Your AI agent can't read a lineage graph either

This is where the argument stopped being academic.

An AI coding agent doesn't write a transform and hand it over. It writes a solution, runs it, reads the output, works out what broke, and tries again. That loop runs for 20 minutes, an hour, sometimes longer, with nobody watching. We use Claude Code this way every day in our consulting work, and the loop is genuinely fast.

The loop needs one thing to function: a signal for what "correct" means. That signal is your test suite. Without it the agent can't tell whether it fixed the bug or broke something upstream, and it can't tell when it's finished. So it stops and asks you. Every iteration. At which point you've bought a very expensive autocomplete.

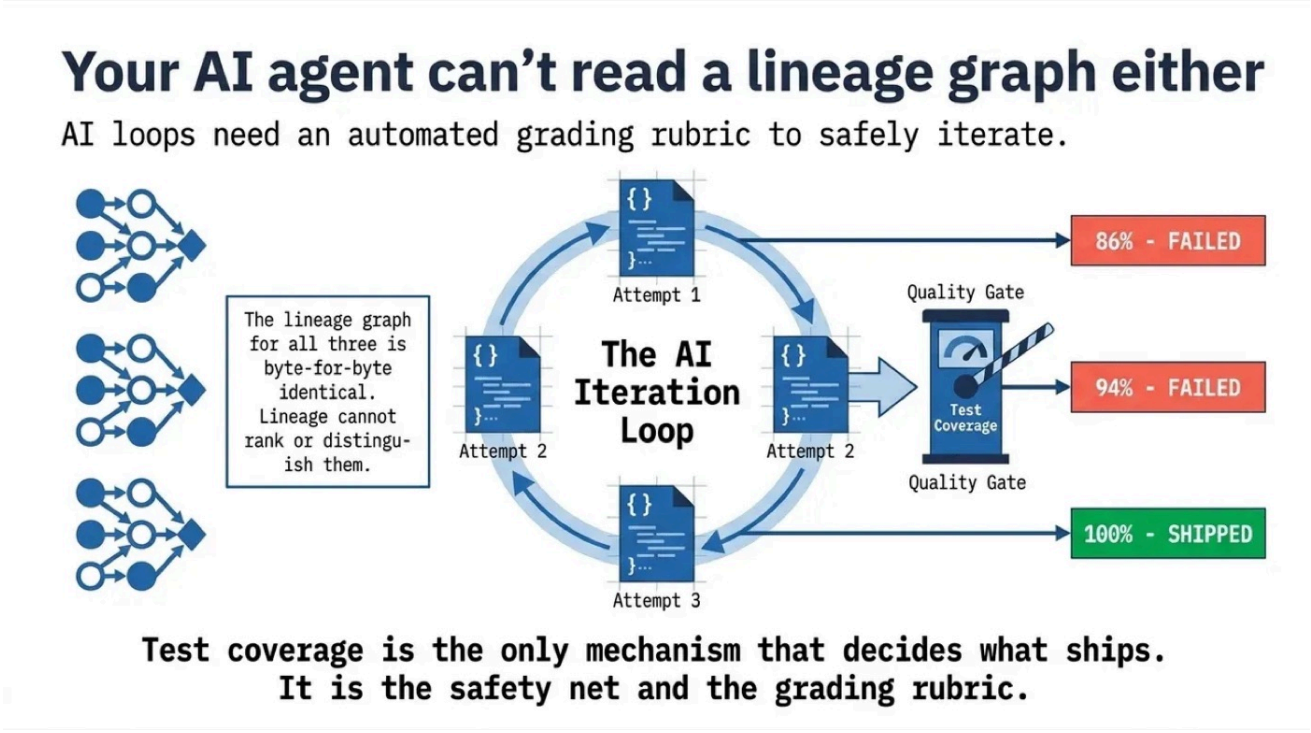
Here's the part that ties back to lineage. Picture three isolated branches, three different strategies, three parallel agent sessions, all working from the same brief on one slow SQL transform. They come back like this.

```
parameterized time window   86%   cross-period consistency check still failing
staging table logic         94%   edge case: partially fulfilled cancelled orders
full rebuild from raw      100%   clean, 8 minute rebuild
```

Three implementations. One shipped. And the lineage graph for all three is byte-for-byte identical, because they read the same sources and write the same target. Lineage can't rank them. Lineage can't even distinguish them. The only thing that separated a transform worth promoting from two worth deleting was the test pass rate.

That's the whole job now. When an agent can produce three plausible answers in the time you used to spend producing one, your test coverage stops being hygiene and becomes the mechanism that decides what ships. It's the safety net and the grading rubric at the same time. And you don't need full coverage to start. Even 50% coverage on your most business-critical transforms changes what you can safely hand off. Buying the graph instead of the tests is one of seven ways AI-generated pipelines break.

The stakes went up too. A schema drift that used to mean one broken report now means an AI system producing confidently wrong answers at whatever rate it can query. The full pattern is here, and none of it works without the tests.



What to build instead

Start with test coverage, and mean it. The rule of thumb is a minimum of two tests per table, two per column, and one custom test per significant business metric. Every tool gets an error log check. Every job gets a status check and a timing check. Apply it at every layer, bronze through gold.

There's a third approach, which our definitive guide to test coverage calls hope-based testing. Manual inspection and intuition. Somebody eyeballs the row counts on Monday and declares it fine. Most teams are running this one right now and calling it a data quality program.

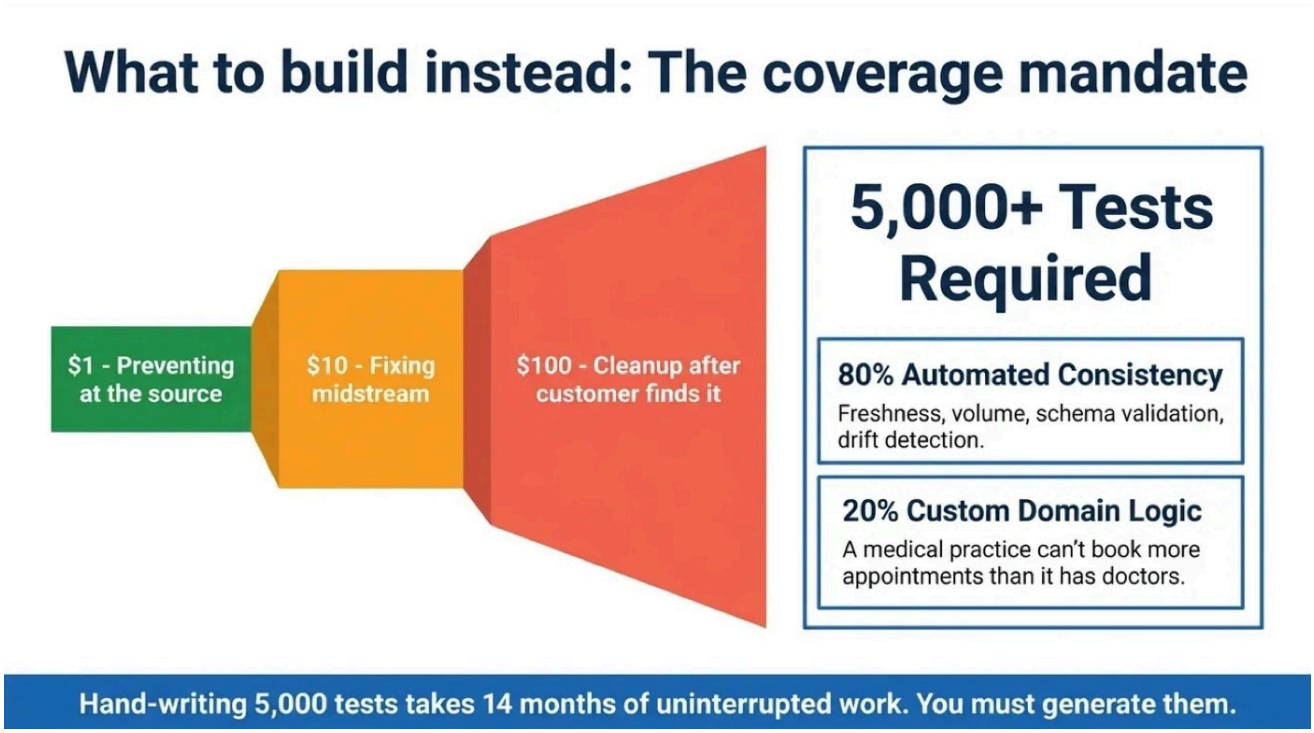
About 80% of that coverage comes from automated consistency tests: freshness, volume, schema validation, referential integrity, and drift detection that catches gradual statistical shifts a threshold alert would sleep through. The other 20% is domain logic that no tool can infer from your data. A medical practice can't book more appointments than it has doctors. No shipments go out on restricted days. Revenue lands inside a historically plausible range. That 20% is where your business knowledge becomes a quality gate, and it's the part you have to write yourself.

Be honest about what each half buys you. Automated anomaly monitors are broad and shallow. They cover far more tables than anyone would hand-craft tests for, and they will also ping you about things that don't matter until your team starts ignoring the channel. The custom tests are the ones with teeth. Wire them as tripwires that halt processing rather than as alerts somebody reads later.

The economics are not subtle. Preventing a data quality problem at the source costs about $1 per record. Fixing it midstream costs $10. Cleaning it up after a customer finds it costs $100. That ratio is why coverage has to shift left, into development, rather than living as a dashboard somebody checks after the fact.

Do the coverage math on a normal warehouse, and you get somewhere north of 5,000 tests. At 30 minutes each, handwriting them costs about 14 months of uninterrupted work, assuming you take no meetings and never go on vacation. That's why most teams have 40 tests on their four favorite tables and nothing anywhere else.

Nobody writes 5,000 tests by hand. Generate them.

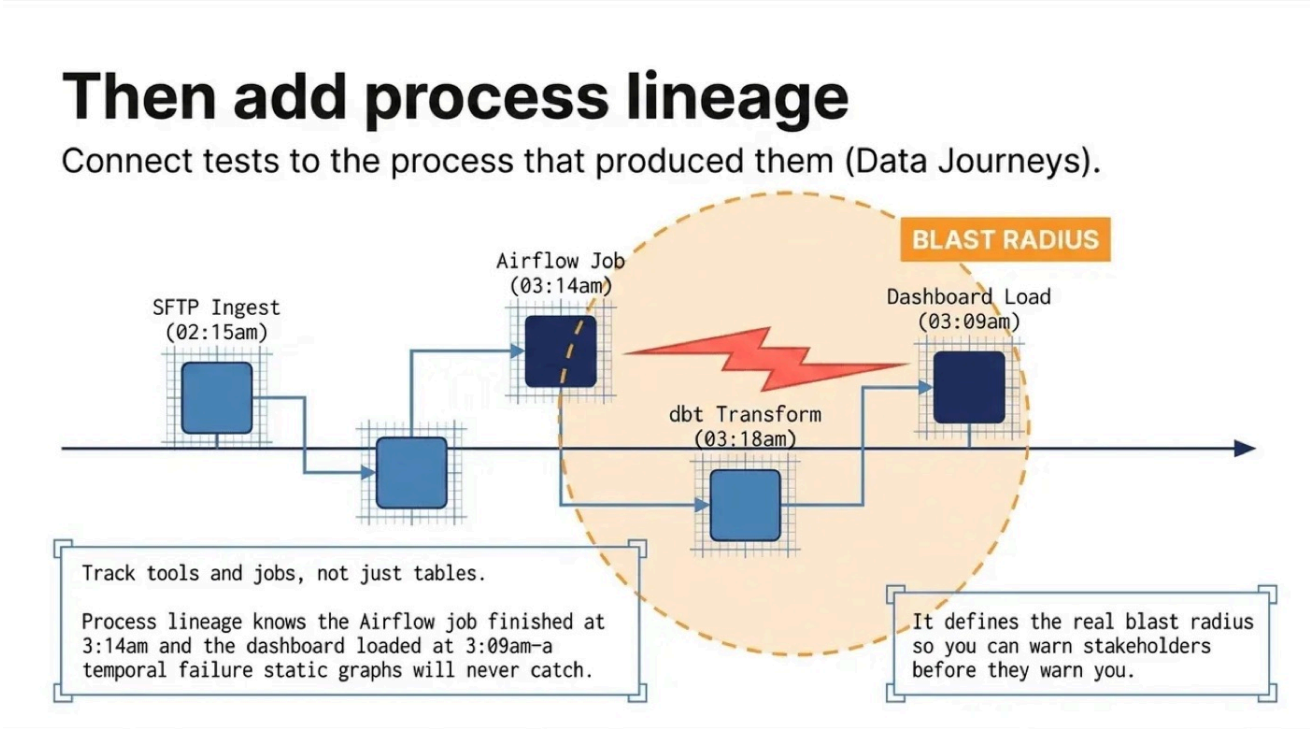


Then add process lineage

Once tests exist, connect them to the process that produced them. That's what the five pillars of a Data Journey give you: run-time lineage across every step, down the stack, for data at rest and in use, with expectations set and alerts wired up.

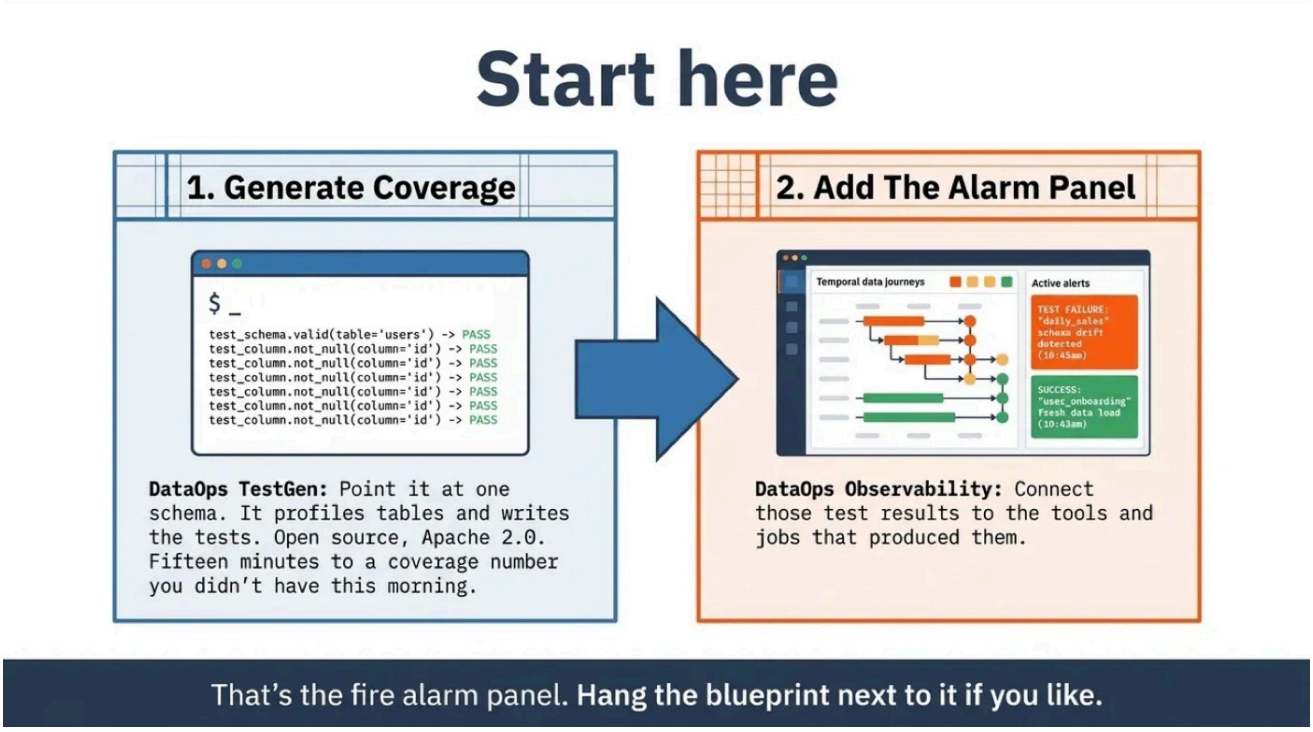
Process lineage tracks the tools and jobs, not just the tables. It knows the Airflow job finished at 3:14am and the dashboard loaded at 3:09am, which is the kind of problem no static graph will ever surface. It also gives you the real blast radius. When an ingested file comes in short, you find out which reports, models, and exports are affected, and you can tell those people before they tell you.

Static lineage tells you which tables touch which tables. Process lineage tells you which tests passed, which broke, and which report went red because of the change you pushed an hour ago. Keep the static version around for the auditors. We wrote more about how the pieces fit together in You complete me, said data lineage to Data Journeys.



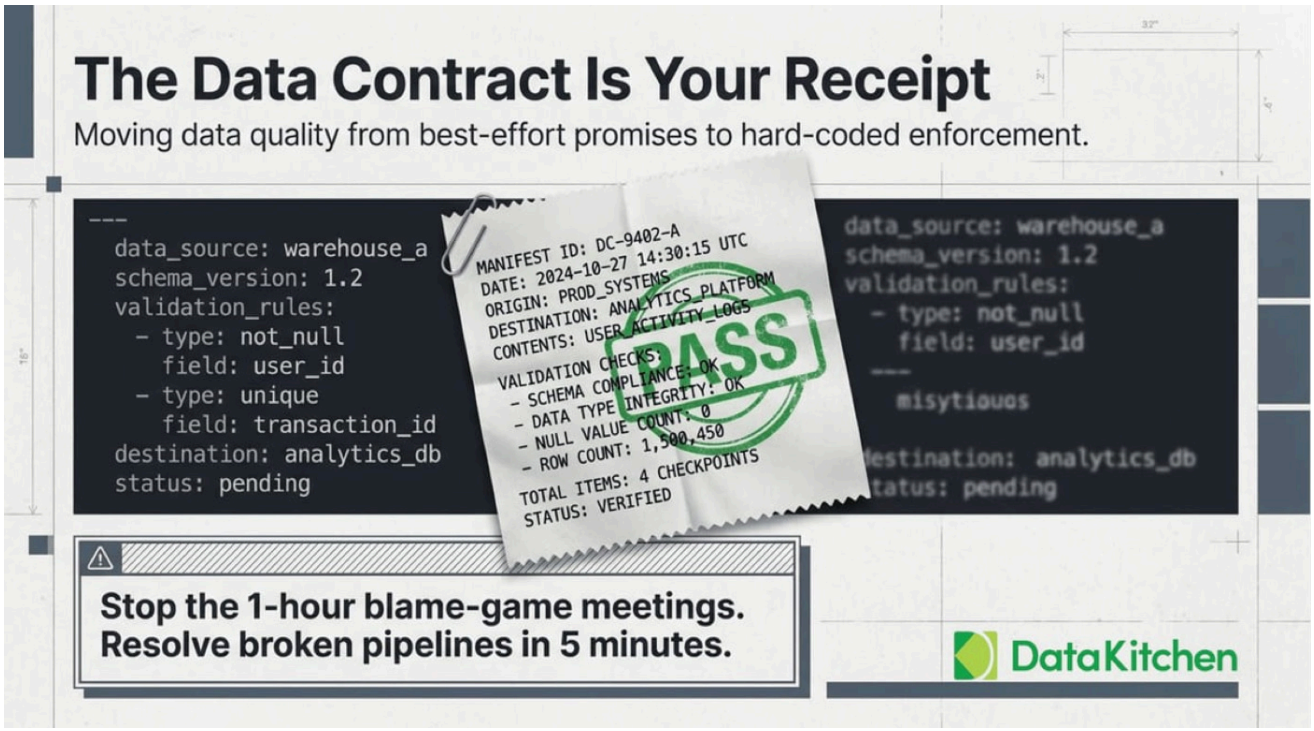
Start here

Data test coverage + process lineage >> data lineage

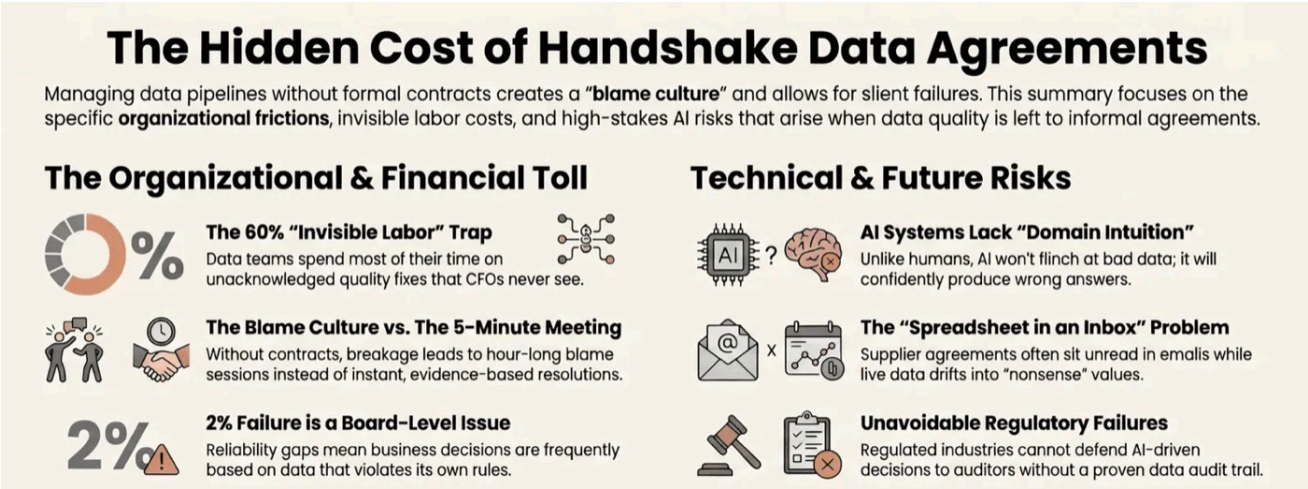


The Data Contract Is Your Receipt

Read the original article at: https://datakitchen.io/blog/the-data-contract-is-your-receipt/



You engineer data. Others bet their decisions on it. Then something breaks, and suddenly everyone is in a blame session. With a real data contract, that meeting lasts five minutes. Without one, it drags on for an hour, and you still walk out losing.



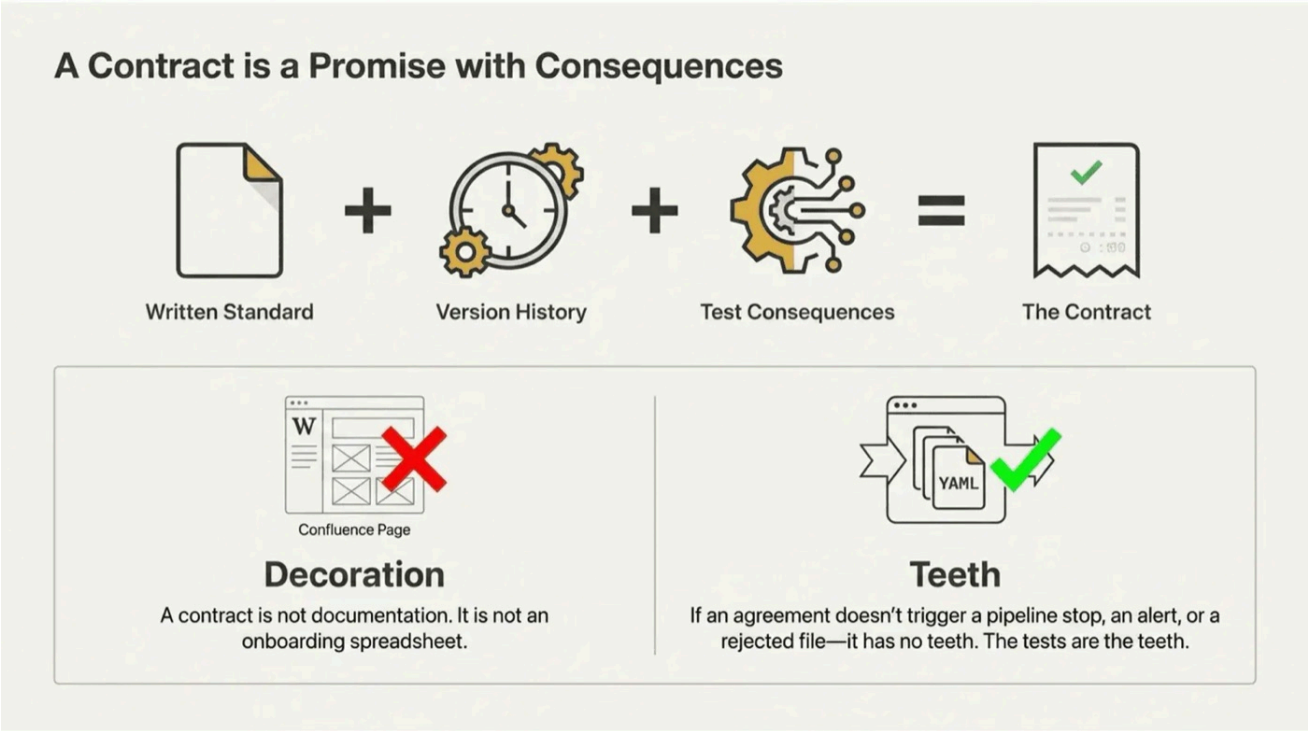
A contract is a promise with consequences

A data contract is not documentation. It is not a Confluence page. It is not a spreadsheet someone sent you in an onboarding email.

A contract is a promise with consequences. Someone commits to delivering data in a specific shape, at a specific cadence, within specific bounds. If they don't, something happens. A pipeline stops. An alert fires. A file gets rejected. The receiving team doesn't spend their Tuesday chasing ghosts.

The consequence is what makes it a contract. Without teeth, it's a decoration. The data tests are the teeth.

That's the whole idea. A written, versioned agreement about what a dataset should look like, paired with tests that prove whether it currently meets that agreement. Column types, null rates, row counts, value ranges, freshness, the whole picture. It lives as a YAML file, or in a tool like TestGen, or both. What matters is that it's real, testable, and dated.



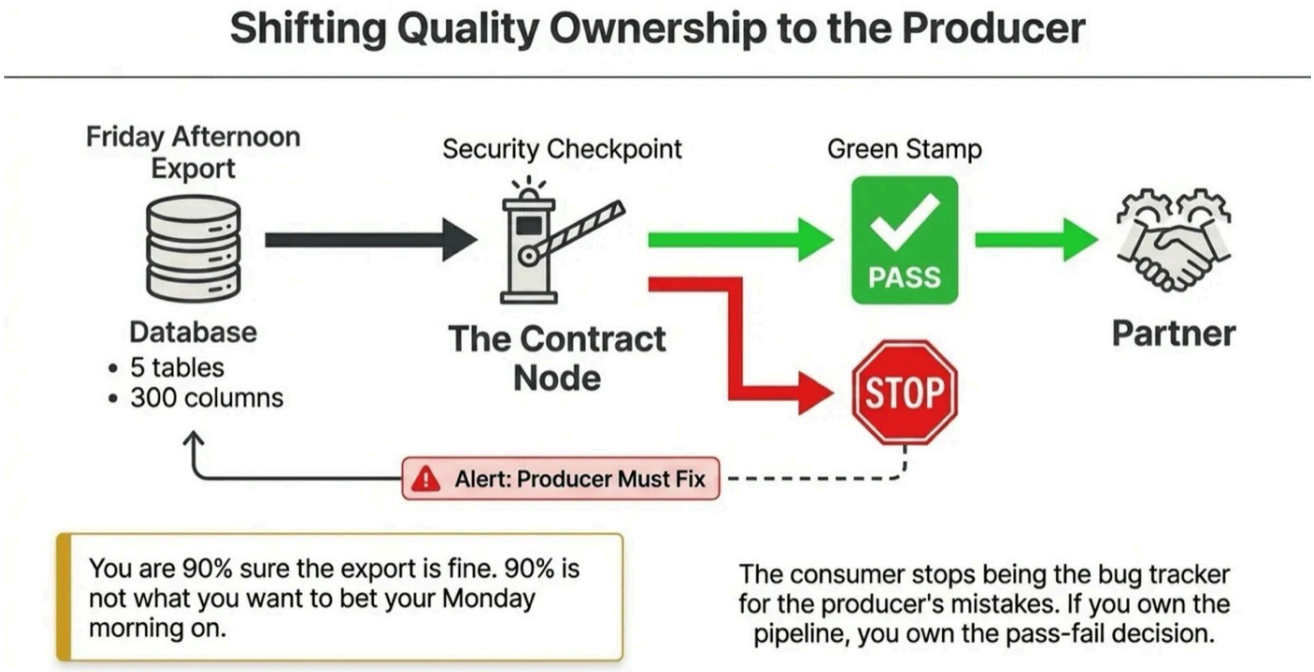
One more thing to get right up front. A contract only works if the person on the other end can actually run it. If your supplier has to buy a SaaS license to comply with your contract, they won't. If your downstream team has to file a procurement request to check their input, they won't. The contract must be accompanied by a tool that the other party can install and run without signing anything. We'll come back to this.

Here is why you want one.

Reason 1: Prove your own output is clean before it goes out the door

You're exporting five tables and three hundred columns to a partner on Friday afternoon. You're ninety percent sure it's fine. Ninety percent is not what you want to bet your Monday morning on.

With a contract, you set the terms once. Then you add a test node at the end of your pipeline that runs the contract against the tables before the export fires. If the data doesn't match, the export doesn't ship. You get a green stamp before you press send.



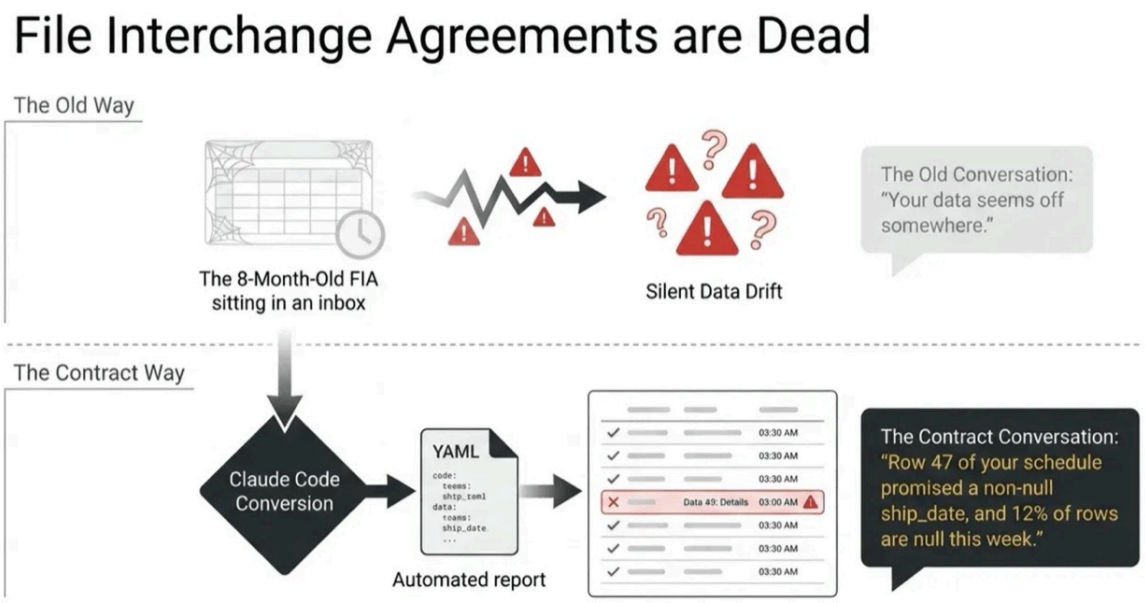
The contract is a structural shift worth naming. Quality ownership moves to the producer, where it belongs. The consumer stops being the bug tracker for the producer's mistakes. If you own the pipeline, you own the contract, and the contract owns the pass-fail decision.

You can also hand the contract to the receiving team as a spec sheet. They know exactly what they're getting. When their BI developer asks, "Are the order IDs always non-null?" you point at the contract and say, "Yes, and here's the test that proves it ran green last night."

Reason 2: Hold your suppliers to what they promised

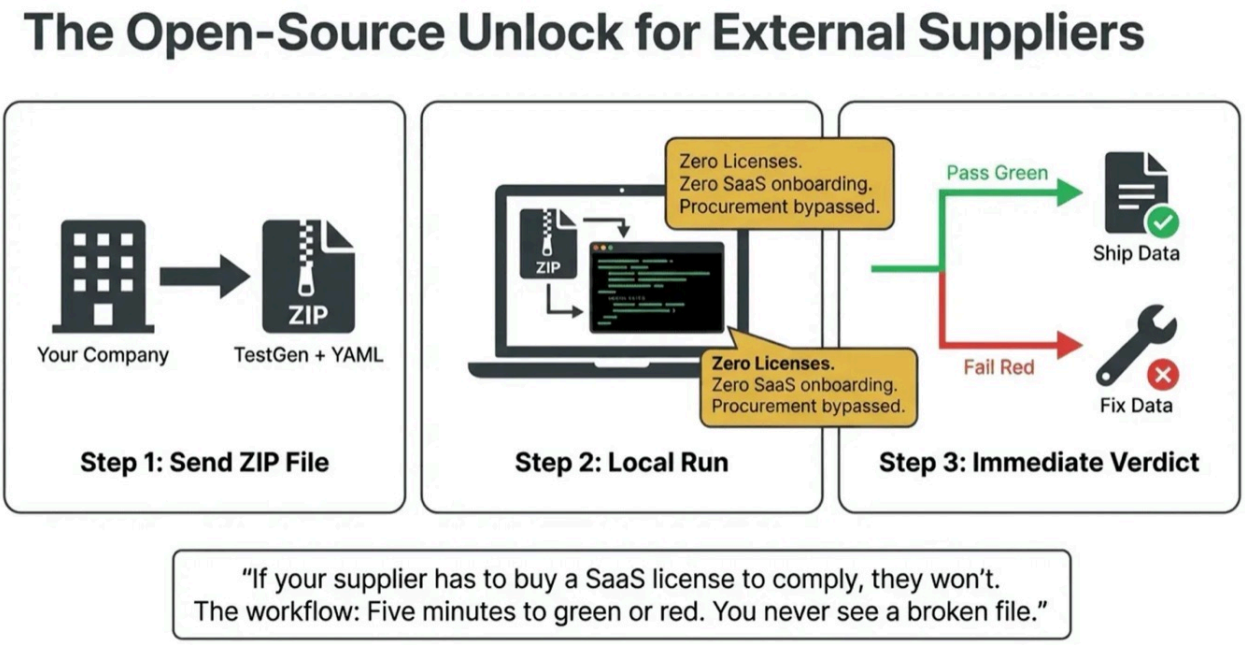
The other side of the same coin. You receive data from a vendor every week. The vendor sent you a file interchange agreement when you signed the contract. That FIA is a spreadsheet sitting in someone's inbox. Nobody has looked at it in eight months. The vendor's data has drifted in three different ways, and nobody noticed until a dashboard started showing nonsense.

Take that FIA and convert it to a YAML contract. Claude Code can do this in about two minutes. Import it into TestGen. Now every new delivery from that vendor runs against the contract automatically. When something drifts, you get a line-item report showing exactly which term failed, and you send that to the vendor. The conversation goes from “your data seems off” to “row 47 of your schedule promised a non-null ship_date and twelve percent of rows are null this week.”



Now here’s the part that makes this actually work. TestGen is open source. Free. No account required. So the move is not “please adopt our data quality platform.” The move is this: you send your supplier a ZIP file containing the tool and a YAML contract, and tell them to run it before they ship the next load. They install it on their machine or their build server in five minutes. They run it. It passes, or it fails. If it fails, they fix the data. You never see a broken file.

That is the difference. Most data quality tools are SaaS. You can’t hand a SaaS product to a supplier and tell them to comply. They won’t sign up, they won’t expose their data to your vendor’s cloud, and procurement won’t let them. Open source is how the contract actually travels. It’s how the other side of the handoff gets to participate without having to negotiate a contract of their own.

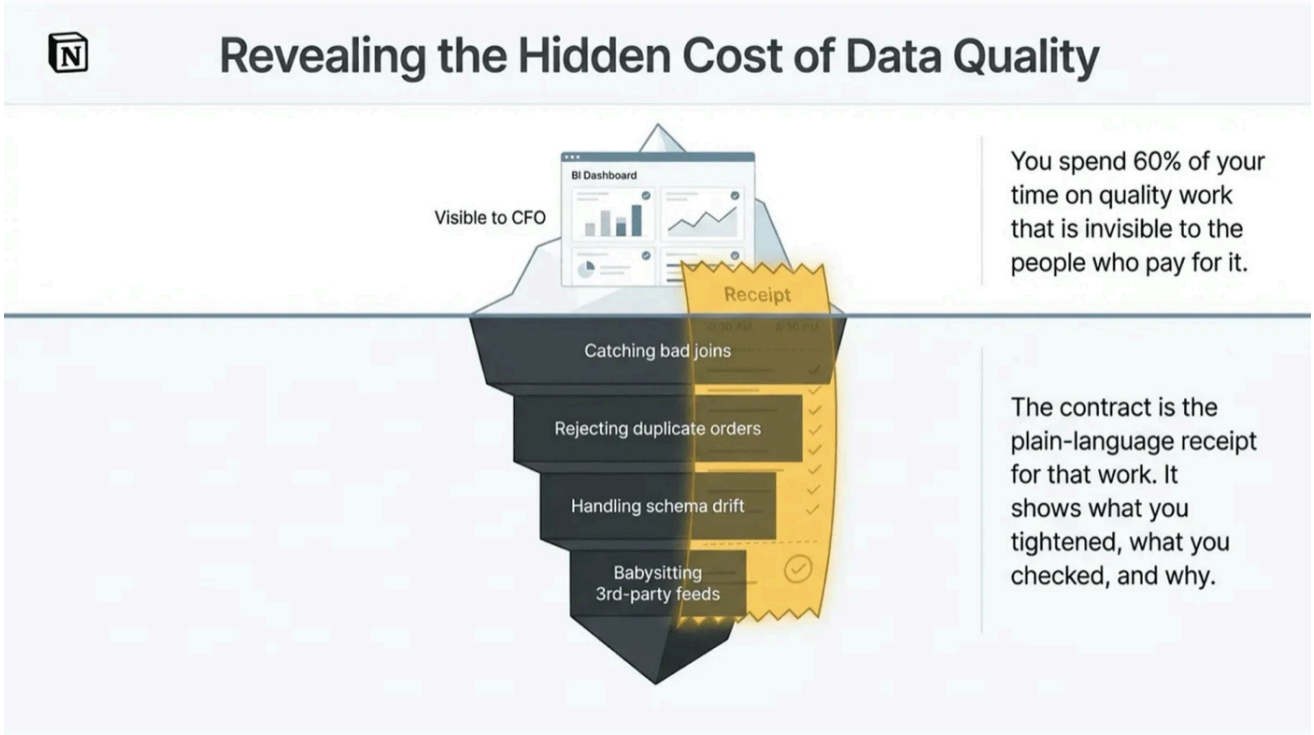


Our consulting team deals with data suppliers who routinely send us garbage. We're handing them open-source TestGen, along with a contract, and telling them to run it before they ship to us. If it doesn't pass, we don't accept the file. That's the whole workflow. Zero licenses. Zero onboarding. Five minutes to green or red.

Reason 3: Show the business what your data work is actually worth

This is the one nobody talks about, and it's the layer of governance that gets your next round of headcount approved.

You spend sixty percent of your time on quality work that is invisible to the people who pay for it. You catch bad joins, you reject duplicate orders, you handle schema drift, you babysit the third-party feed that breaks every other Thursday. None of that shows up in a quarterly review. The CFO sees a dashboard and assumes it happened on its own.



A contract is the receipt for that work. It lists every quality check. It shows when it ran. It shows pass/fail over time. The version history shows what you added, what you tightened, and why. When your data steward sits down with a business owner, the contract is the artifact they point at. Not your SQL. Not your test suite config. The contract, in plain language, with terms a business person can read.

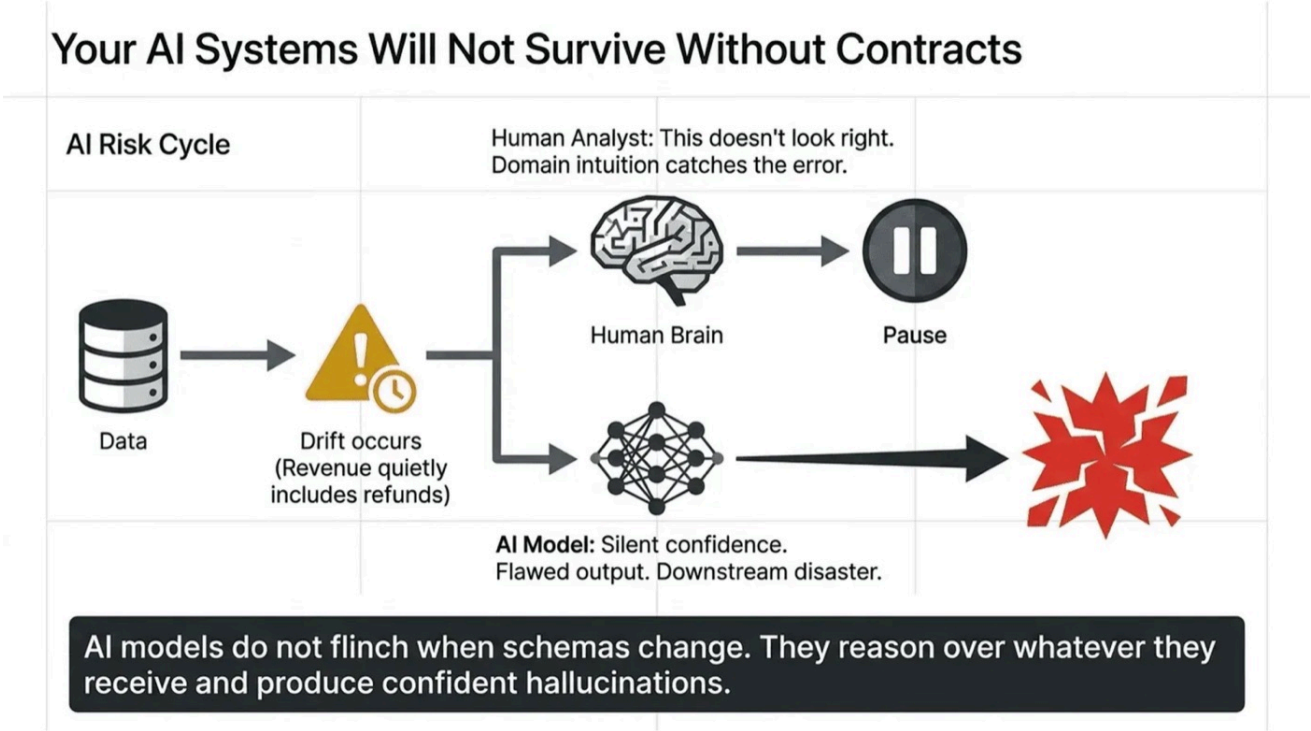
Here's the move that matters for anyone looking to advance this work. Engineering reports "our pipelines ran at 98% reliability this quarter." The CFO hears that and says, "Fine, moving on." But the same number means something different when you read it against the contract: two percent of the business decisions this quarter ran on data that broke its own stated rules. That's not fine. That's a board-level sentence. Contracts give you both readings from one source of truth, and the second reading is the one that gets budget approved.

Several teams using our software have told us flat out: they want a contract specifically to show that their work is valuable to an organization that governs by contract. They're not wrong. If your company runs on written agreements everywhere else, it should run on written agreements for data, too.

Reason 4: Your AI systems will not survive without contracts

AI is the reason that will matter most over the next three years, and it's the one most teams are not ready for.

When a human analyst gets bad data, they usually catch it. Something looks off. The number doesn't match last quarter. The trend breaks in a way that doesn't make sense. Humans have domain intuition, which is a safeguard against weird input.



AI systems have no such safeguard. A model does not pause to say, “This doesn’t look right.” It reasons over whatever it receives, produces a confident answer, and the downstream workflow acts on it. An agent reading your customer table will not flinch when the schema changes. A forecasting model will not notice that the revenue column quietly started including refunds two weeks ago. The output will still look plausible. It just won’t be right.

Data contracts are no longer optional for any team putting AI into production. The contract is the upstream instrument that specifies the inputs the AI system can trust. Without one, you’re running models on data that could drift at any moment, and you will not know until a customer calls or a regulator shows up.

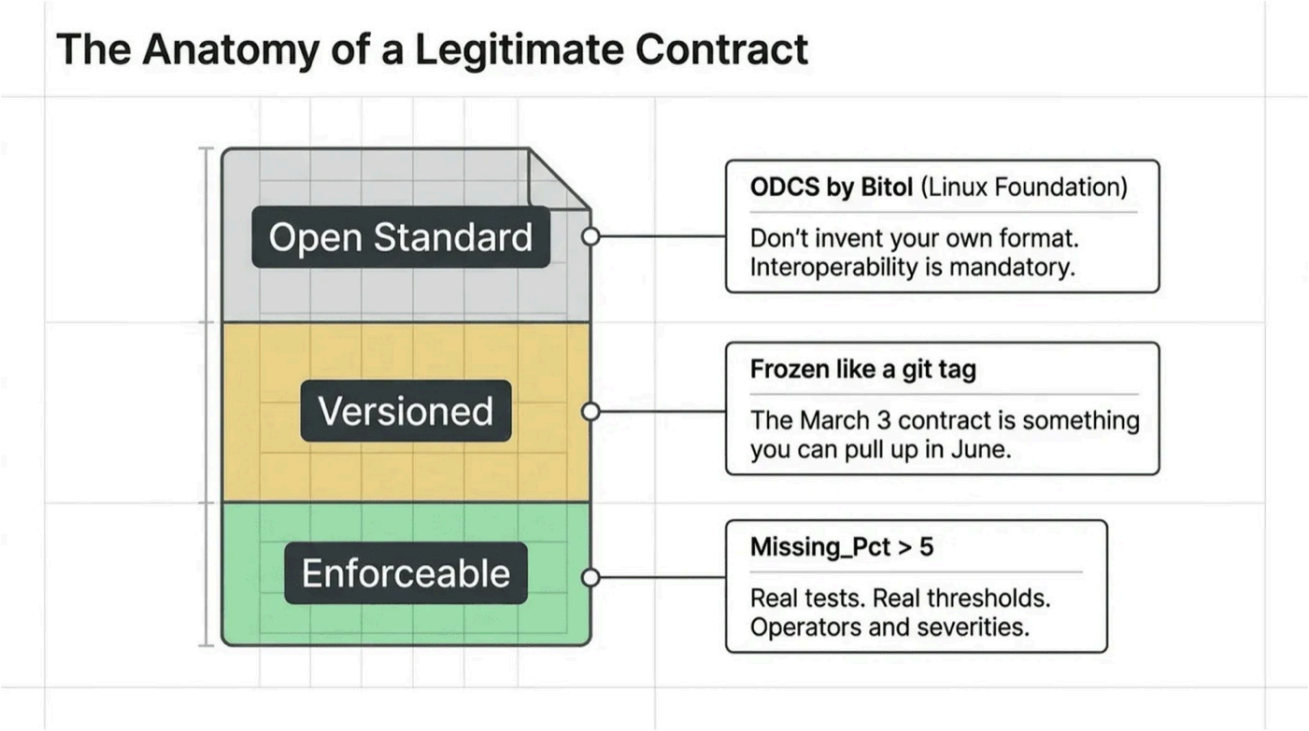
For a pharma company, a bank, an insurer, or anyone else running in a regulated environment, this is the whole ball game. You cannot defend an AI-driven decision to an auditor if you cannot prove that the data going in met a stated standard. The contract is that proof. It’s the audit trail for every automated decision your systems made last quarter.

If you’re a CDO thinking about where contract infrastructure fits on next year’s roadmap, this is the forcing function. Your AI spend is going up. Your tolerance for silent data failures is decreasing. Contracts are what sit between those two lines.

What a good contract actually looks like

Short answer: it follows a standard, it’s versioned, and it’s enforceable.

The open standard is the Open Data Contract Standard (ODCS), maintained by Bitol under the Linux Foundation AI & Data. It covers the fundamentals, schema, quality rules, servers, and compliance sections you need. Use it. Don't invent your own format, because then you can't share contracts with anyone who isn't using your tool.



Versioned means every saved contract is a named, frozen point in time, like a git tag. The March 3 contract is something you can pull up in June when someone asks what we agreed to back then.

Enforceable means there are real tests behind every term. Not "the column should be clean." A threshold. A metric. An operator. A severity. If Missing_Pct > 5, the test fails. That's a term worth having.

Stop being the one who checks other people's work

Here's the cleanest way to use this, and it's the move we're betting on with our own suppliers.

You can install open-source TestGen on your side, point it at every delivery, run the contract, and catch the problems after they arrive. That works. It's already better than what most teams do today.

The State of Data Governance

	The Old Way	The Contract Way
Internal	Consumer acts as bug tracker	Producer owns pass/fail pipeline stops.
Suppliers	Ignored spreadsheet FIAs	Automated YAML runs locally via ZIP.
Business	60% of quality work is invisible	Versioned receipts prove data ROI.
AI/Compliance	Silent drift creates hallucinations	Frozen audit trails defend decisions.

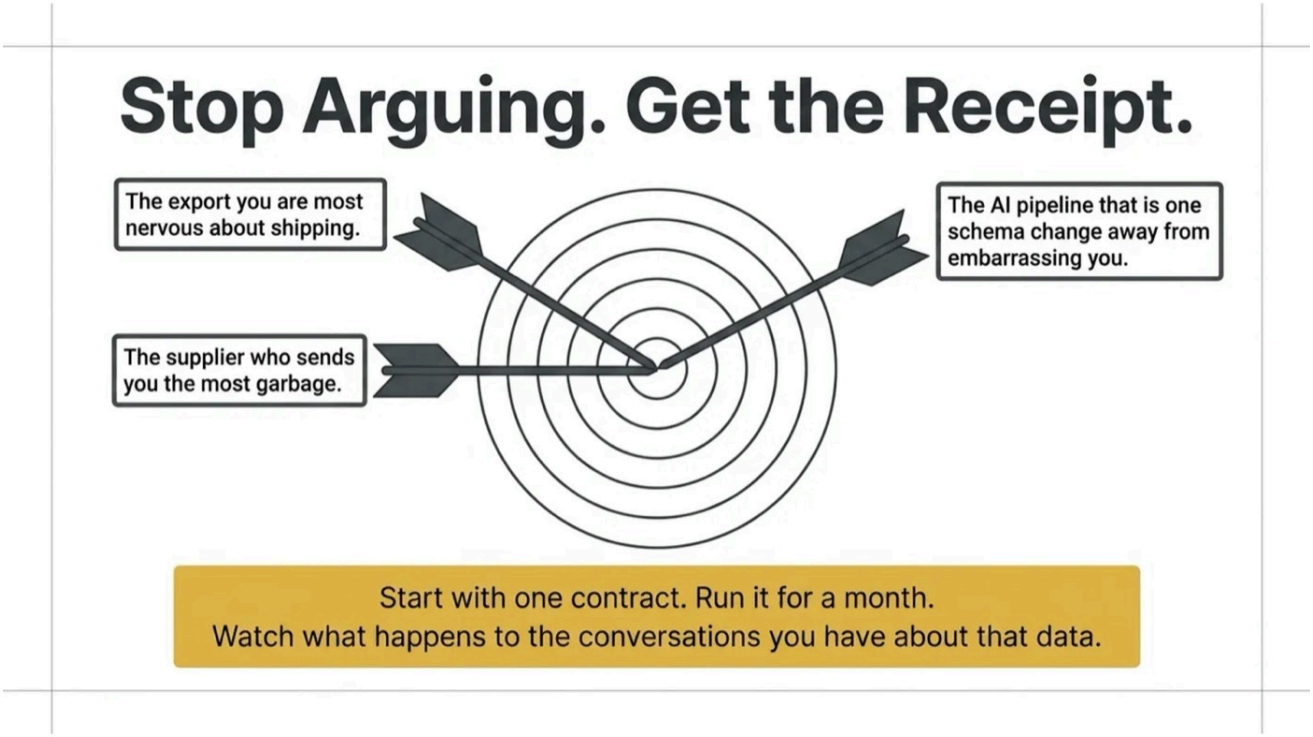
But the better move is to stop being the one who checks. Hand the tool and the contract to the supplier. Tell them: here's open-source TestGen; here's the contract; you'll have it running in 10 minutes. Run it before you ship. If it passes, ship. If it fails, fix it before you ship. Stop being a loser.

Now the burden lives where it belongs. The supplier is responsible for the quality of what they send. You stop playing goalie. The contract is the same either way, but the work of checking moves upstream to the only place it can actually prevent the problem. That's the whole shift. The reason it works is that TestGen is free and runs anywhere, so there's no excuse and no friction on the other side.

Ten minutes to set up. Zero dollars. One yes-or-no answer before every delivery. If a supplier won't do that, you've learned something important about the supplier.

Start with one contract

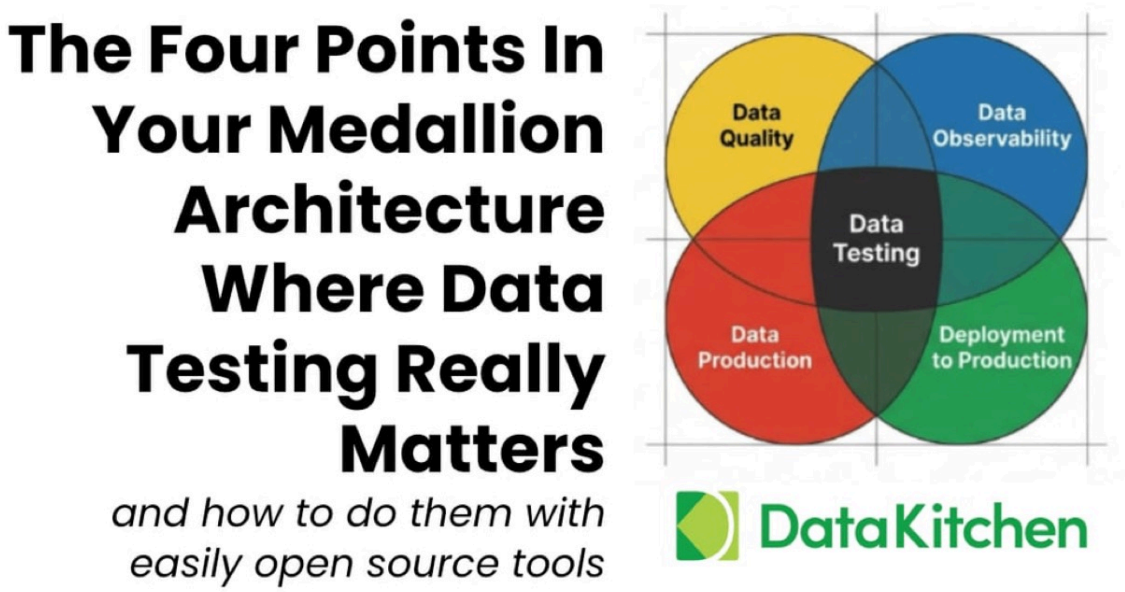
Pick the export you're most nervous about shipping, or the supplier who sends you the most garbage, or the AI pipeline that is one schema change away from embarrassing you. Write one contract. Run it for a month. Watch what happens to the conversations you have about that data.



You'll stop arguing about whose fault it was. You'll have the open source receipt.

The Four Points in Your Medallion Architecture Where Data Testing Really Matters

Read the original article at: https://datakitchen.io/blog/medallion-architecture-four-points-data-testing/



Why this keeps happening

Most data teams build on hope. You trust your data providers. You trust the pipeline ran. You trust the report is right. Then a Slack message arrives at 9 am, and the dashboard is already wrong in front of your customer.

We wanted to know why this keeps happening, so we used Claude Code to scan the internet and pull about 850 comments from data engineers about why they don't test. The reasons rhymed. I don't have time. I don't have support. Errors aren't visible until someone yells at me. The data changes too much. The tools aren't good. The whole industry is still immature. Almost nobody said, "Testing is easy, and I do it everywhere." Almost everybody said some version of "we eyeball row counts and pray."

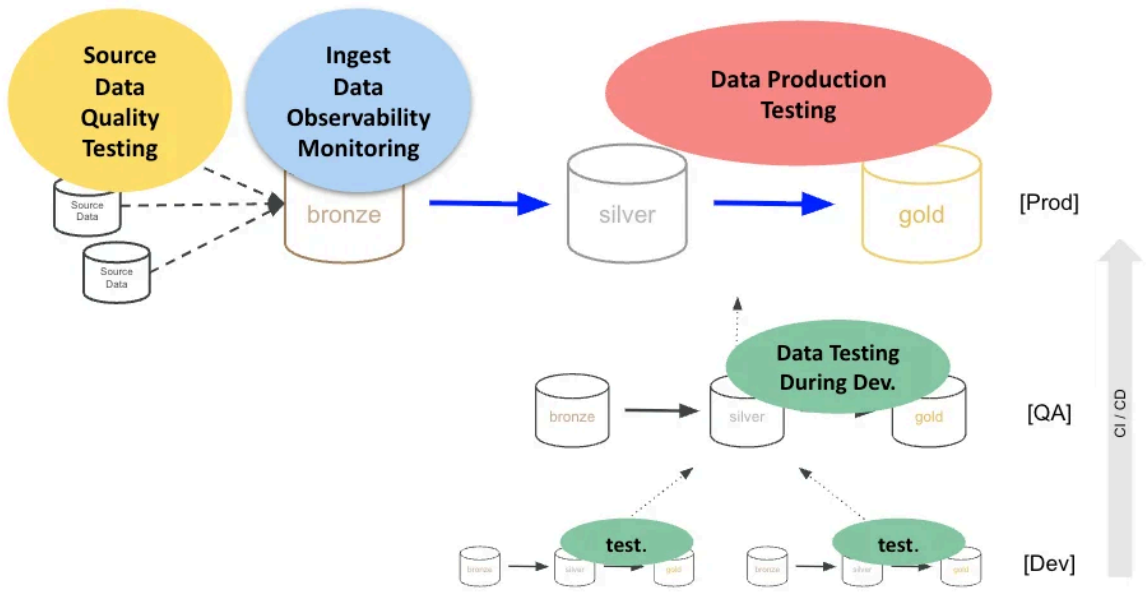
The Medallion architecture has become the default in many large organizations, and it makes the praying worse, not better. You have more layers, more tables, more tools, and more places for an error to slip through unnoticed. So this is the practitioner's guide to where you test in a Medallion architecture and how to do it. There are four points. Miss one, and you keep getting yelled at.

What Medallion actually is

Medallion is a layered data architecture. You read it left to right.

You take source data and land it in a bronze layer. Bronze is a near-table-by-table copy of the source. No cleanup, just capture. From there, you move data into a silver layer, which gets “just enough” cleaning and transformation: mastering, patching, and a unified view of your core business entities. Then you build the gold layer, which is fit for consumption. Gold is your facts and dimensions, your aggregate tables, and the schemas your BI tools and analysts actually query. Sometimes there’s a platinum layer after gold. Sometimes the layers are called L1, L2, and L3. Databricks popularized the pattern, and it shows up everywhere in the lakehouse now.

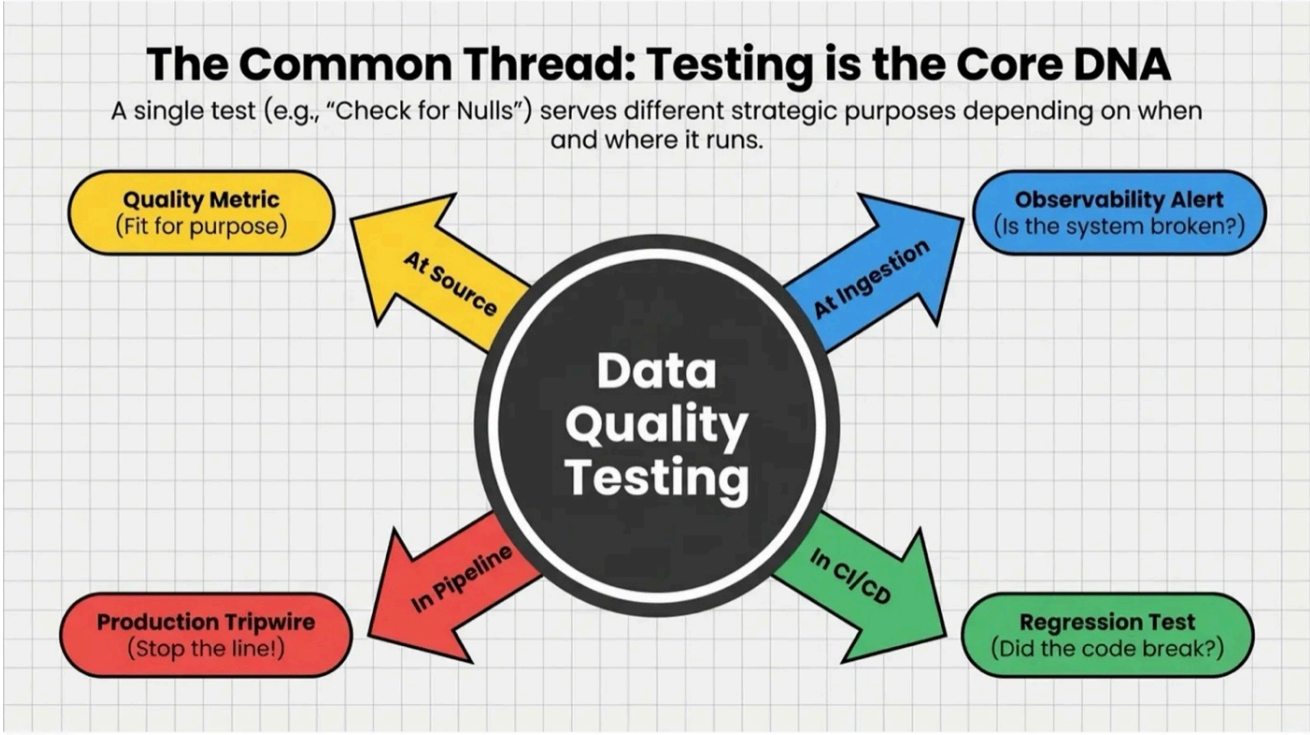
There’s a second axis you can’t ignore. Bronze, silver, and gold run in production, but they also run in QA and in development. Your engineers build a new transformation or add a new dataset in dev. It moves to QA, and then it ships to prod through CI/CD. So you don’t have one Medallion. You have three copies of it stacked on top of each other, and code has to travel up the stack without breaking anything.



That second axis is why people get confused about testing. They think testing is a single activity that happens in one place. It isn’t. The same Medallion gives you four distinct testing jobs, each with a different goal, a different owner, and a different failure mode.

One test, four jobs

Here’s the thing that trips everyone up. A data quality test is just code that checks data. It’s scriptable, automated, and reusable. Take the simplest test you can imagine: check for nulls. That one test means four completely different things depending on when and where it runs.

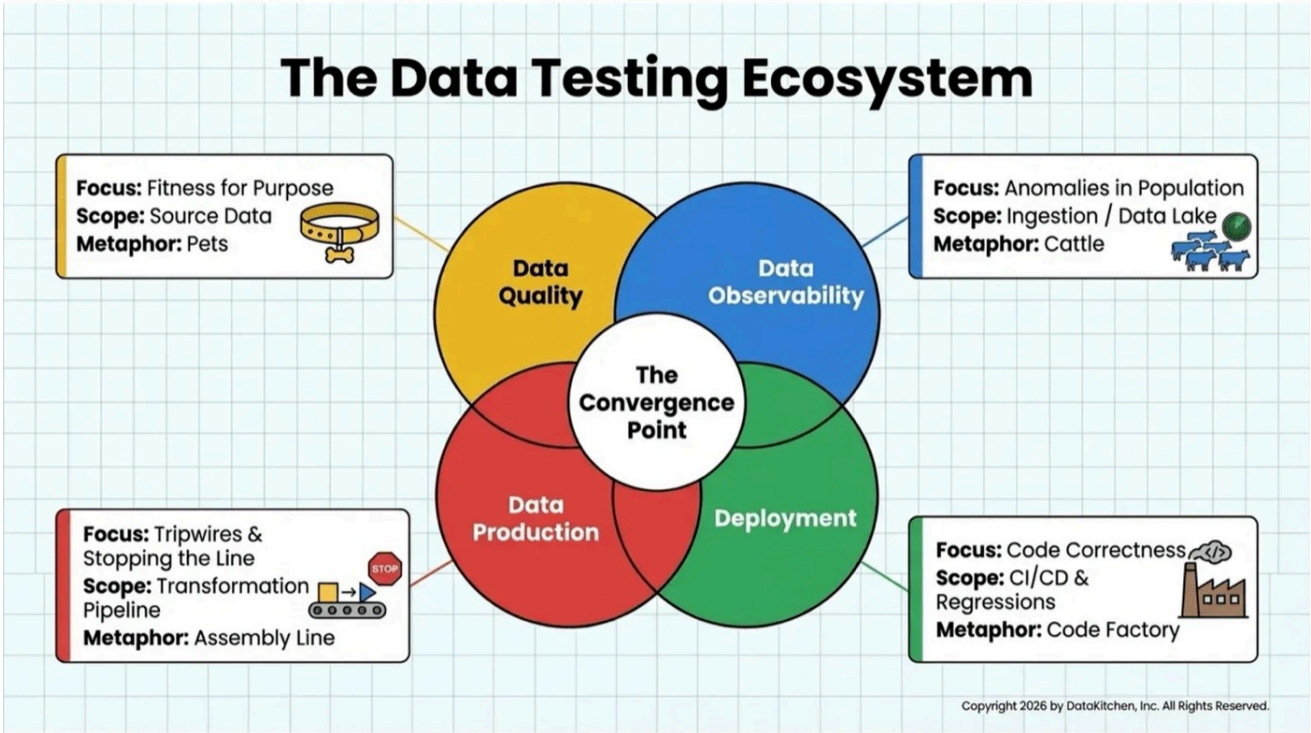


Run it on source data, and it's a quality metric. This column shouldn't be null, and the fact that it is tells you the source isn't fit for purpose. Run it at ingestion, and any column that was full yesterday and is half-null today is an anomaly. Something changed, and you want to know before your customer does. Run it in production, and a null in your gold fact table after you joined three datasets is a reason to stop the line before that table reaches anyone. Run it during development, and a column that's suddenly null after a code change means you broke something, and you should know before you deploy.

Same test. Four purposes. Four places. If you only run it in one of them, you're covered for one failure mode and blind to the other three. That's the whole point of the four points.

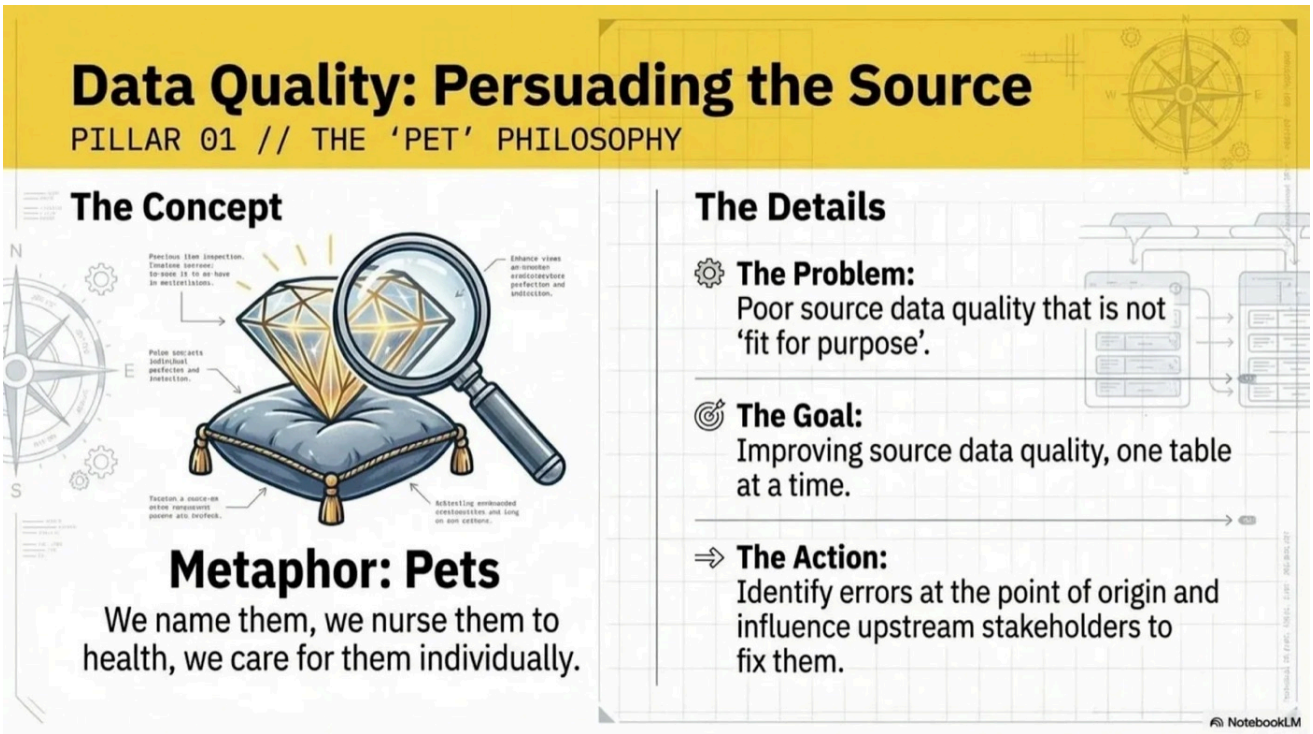
So let me lay them out in the architecture. On the left, you have source data quality testing, where you check data in the source systems or in their reflection in bronze. At ingestion, you have data observability, where you watch whether data arrived on time and looks normal. Through the transformation steps, you have data production testing, where you check the data as it moves through bronze, silver, and gold, and stop on errors. And underneath the whole stack, you have development testing, where you run today's code against test data to catch regressions before they ship.

I like alliteration, so the four actions are persuading at the source, polling for problems, protecting production with tripwires, and protecting the production push. Pets, cattle, the assembly line, and the code factory. Let's go through all four.



Point one: data quality. The pets

Source data quality is the pets problem.

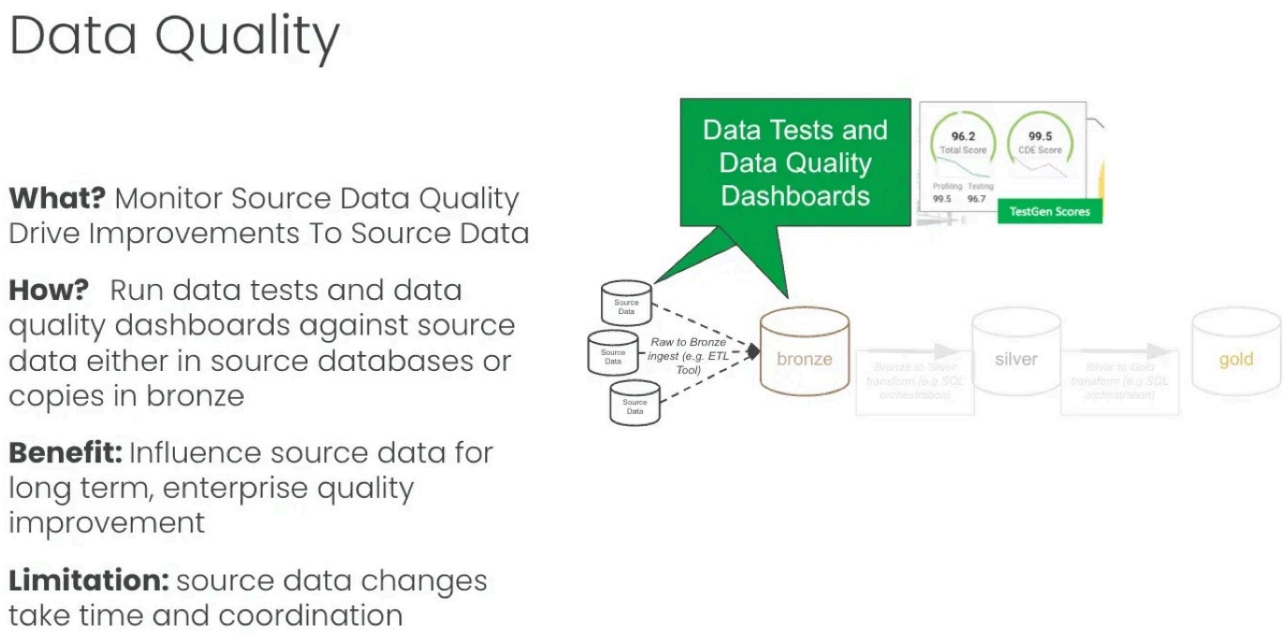


In DevOps, there's a metaphor for pets and cattle. A pet is the server you set up by hand, the one where you have to remember which buttons you clicked, the one you nurse back to health when it's sick. Cattle are the herd you manage by the numbers and never name. Source data is pets. You're trying to make specific tables fit for purpose, one table and one column at a time, and you can't automate your way out of it because you usually don't own the source.

That's the hard part. The data you need to fix lives in someone else's system. You can't change it. You can only prove it's wrong and persuade the owner to fix it. So data quality is an influence game. You work small. You pick a specific set of fields that matter instead of boiling the ocean, score them, and use the score to get someone to act.

The way you win that influence game is with data quality dashboards, not one generic dashboard. Different stakeholders need different views. Some dashboards focus on critical data elements. Some tie quality to a business goal. Some are organized by data source so you can hold a specific supplier accountable. Some are built for a single data consumer, like the eight fields a data scientist actually cares about. One size doesn't fit all here, and a multi-dashboard strategy is what gets you real coverage.

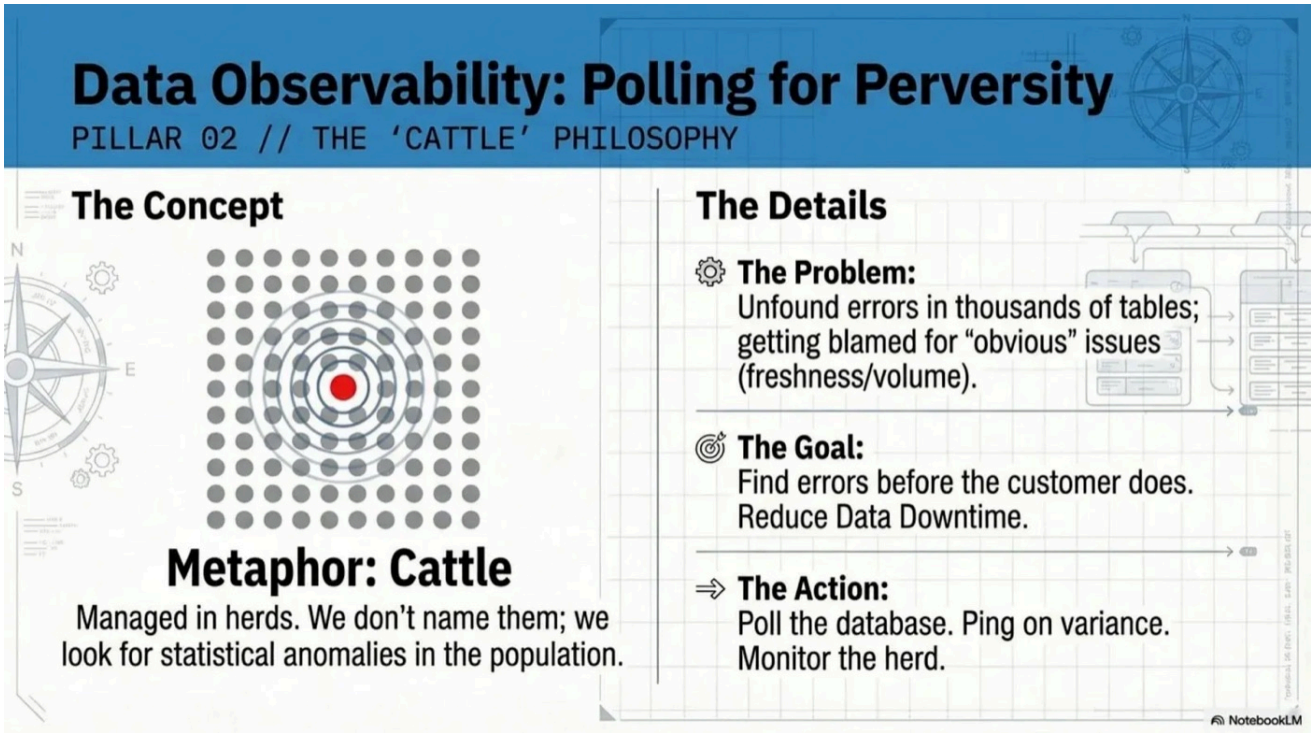
But a dashboard is worthless on its own, and this is the rule that matters most. Every dashboard score must be traceable to a data test. We call it the DQ test equivalence principle. A score without a clear "fix this" path is just noise. Every dashboard score breaks down into a set of data-quality workflow tickets, and each ticket must be backed by a test that both demonstrates the problem and verifies the fix. Test results roll up into the score. The score points back to the ticket. The ticket points back to the test. We've watched teams drop the dashboard entirely and just work the test results and tickets, and that's fine. What doesn't work is an abstract quality score floating free of any test that says exactly what to do.



The limitation is real. Source quality testing tells you nothing about silver, gold, or anything downstream. Your sources can be perfect, and your reports can still be wrong, because the transformation broke or the data never showed up. The benefit is that it's simple. You're just connecting to data, either in the source databases or in their bronze copies, and checking it. One connection, no orchestration required.

Point two: data observability. The cattle

Now stop thinking about individual tables and start thinking about the herd.



When source data lands in bronze, somebody owns that landing zone, and it's not six tables. It's 600 or 6,000. That person doesn't have time to lovingly improve each one. They have a transportation problem. Data moves from the source into bronze through Informatica, or Fivetran, or a pile of custom scripts, and what they need to know is whether it arrived, whether it's all there, and whether anything looks off. They're scanning a herd of a thousand cattle for the one with the lame leg, or the one that isn't there anymore.

That's observability, and it's polling for problems. You continuously monitor the bronze tables and ping someone when a value drifts. The mechanism is time series anomaly detection. You learn each table's normal behavior over time and flag the runs that don't fit. There are four monitor types worth running.

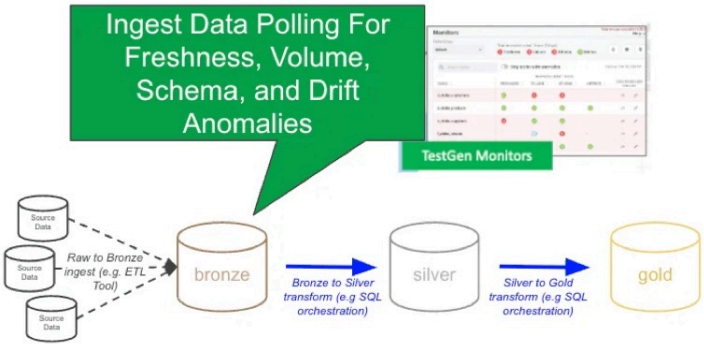
Data Observability

What? Continuously monitor source data on ingest for freshness, volume, schema, and drift anomalies.

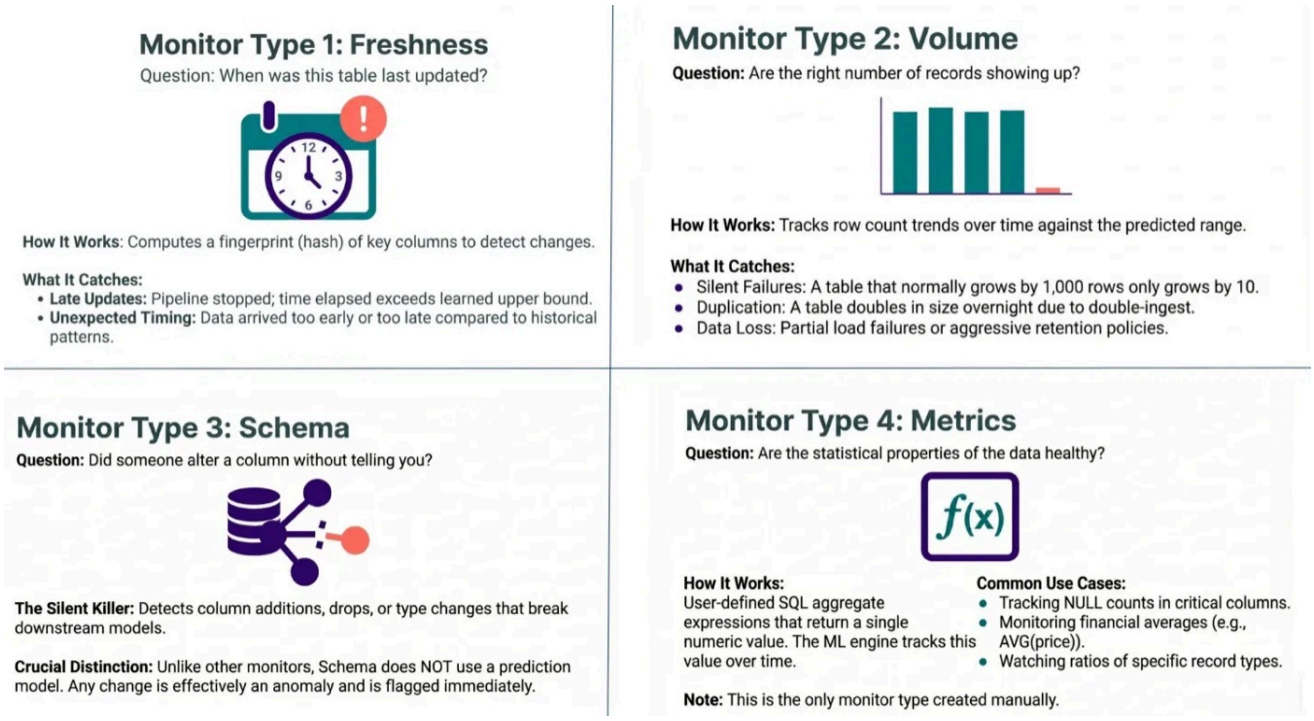
How? ML based time series anomaly detection on data at ingest across all tables.

Benefit: Early detection of data issues

Limitation: Data errors may be found after they get into production



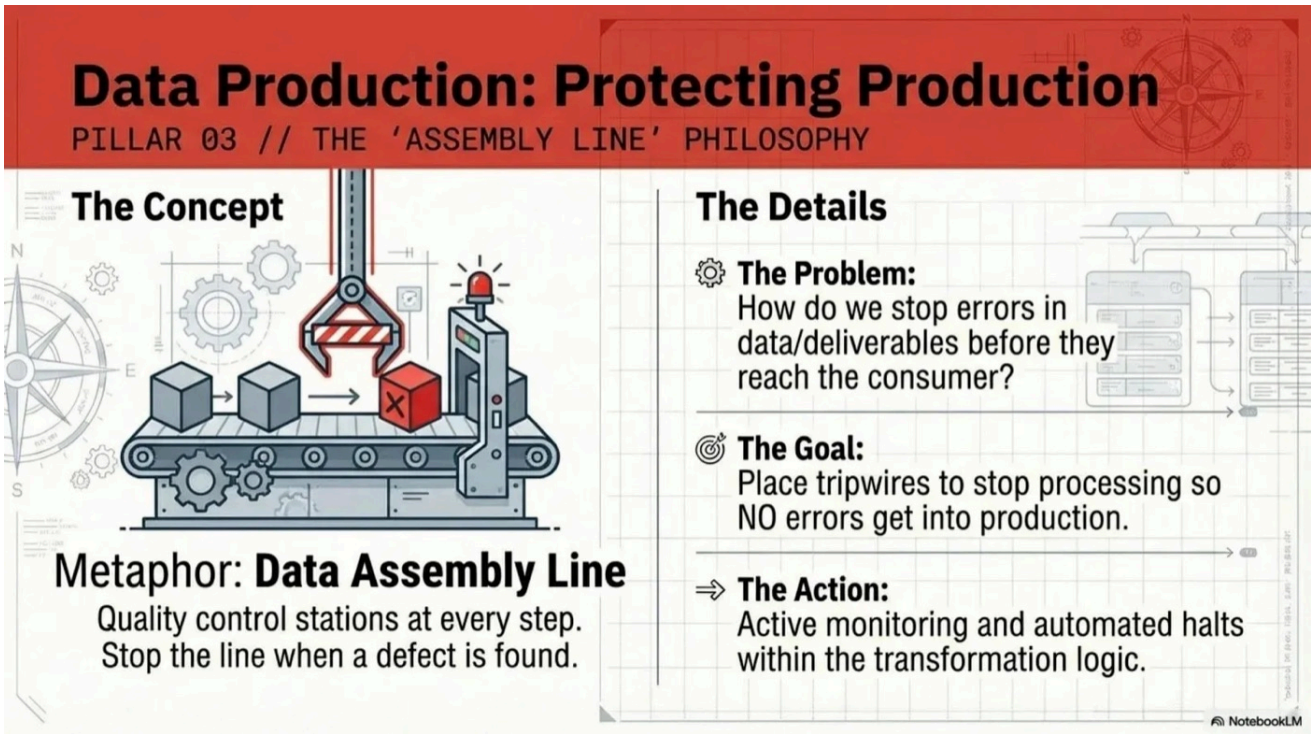
Freshness asks when the table was last updated. It learns each table's update cadence and flags late or oddly timed loads. If a table updates every Tuesday and it's Wednesday with no data, something's wrong. Volume asks whether the right number of records showed up. A table that normally grows by 1,000 rows and grows by 10 is a silent failure. A table that doubled overnight was double-ingested. Schema asks whether someone altered a column without telling you. Schema is the silent killer, and it's the one monitor that doesn't use a prediction model, because any change at all is effectively an anomaly and gets flagged immediately. Metrics assess whether the statistical properties of the data remain healthy. You write an SQL aggregate that returns a single number, such as the average discount or the null count in a critical column, and the engine tracks it over time. Metrics is the one monitor type you build by hand.



The value is an early warning. Finding the problem when it arrives is far better than finding it three layers downstream in front of a customer. The limitation is that this is passive detection. Observability tells you the data is late, short, or reshaped, but it doesn't stop anything. Errors still flow into production while you read the ping. You also get the occasional irrelevant ping, which is the price of anomaly detection. But in the land of the blind, running freshness, volume, schema, and metrics on every bronze table is so much better than hoping, and you can stand it up fast.

Point three: data production. The assembly line

Now we get into the actual production process, the part where data moves through bronze, silver, and gold with code transforming it at every step. This is the assembly line, and the metaphor carries its own rule: put a quality control station at every step, and stop the line when you find a defect.



The goal is to prevent bad data from reaching the customer. The tool is the tripwire. A tripwire is a test embedded in your transformation logic that halts the pipeline on failure. Your orchestrator (Airflow, Dagster, dbt, run-through tasks, whatever you use) runs a series of steps on a schedule. You add test steps between them. In bronze-to-silver runs, you test silver, and if the tests fail, the line stops, and gold never gets the bad data.

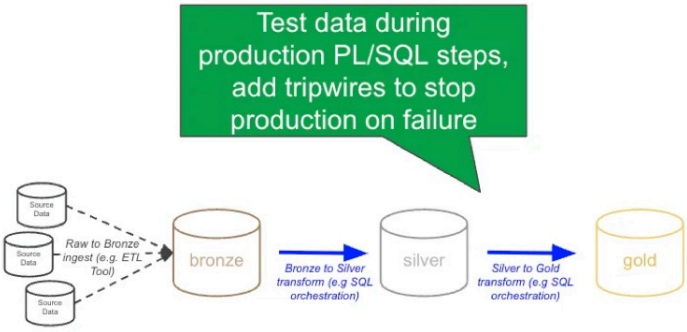
Data Production Testing

What? During your production process, you want to stop delivering data if there's a serious errors and thereby avoid customer-visible data issues

How? Add data quality tests during the production process as well as tripwire to stop on errors

Benefit: Stops data errors before they get into production

Limitation: Needs tripwire as part of orchestration



We built an example of this in Databricks. The pipeline ingests raw data into bronze through IICS. TestGen monitors the bronze database for issues, and then the bronze-to-silver and silver-to-gold transforms each have a TestGen tripwire task right after them. The transform is plain SQL run through tasks or dbt. The test step is a bit of Python that calls out to TestGen, runs the suite against the new data, and returns the results. If the silver suite fails, the job stops before silver-to-gold ever runs. When a suite passes, you see it in the run log: 41 of 41 tests passed in the silver suite, no failures or warnings, results written back to a Delta table. There's a full walkthrough of this setup in Data production tripwires in Databricks: stop bad data before it reaches production.

The part people like is that the results don't stay buried in a test tool. We push every TestGen result up into Unity Catalog, so when you open the customers table in the catalog, you see the test summary right there: 3 of 3 passed on customer_id, the test types that ran, the last run time, and a tag for how many tests failed. The catalog becomes the place your whole team already looks, and now the test status lives there too.

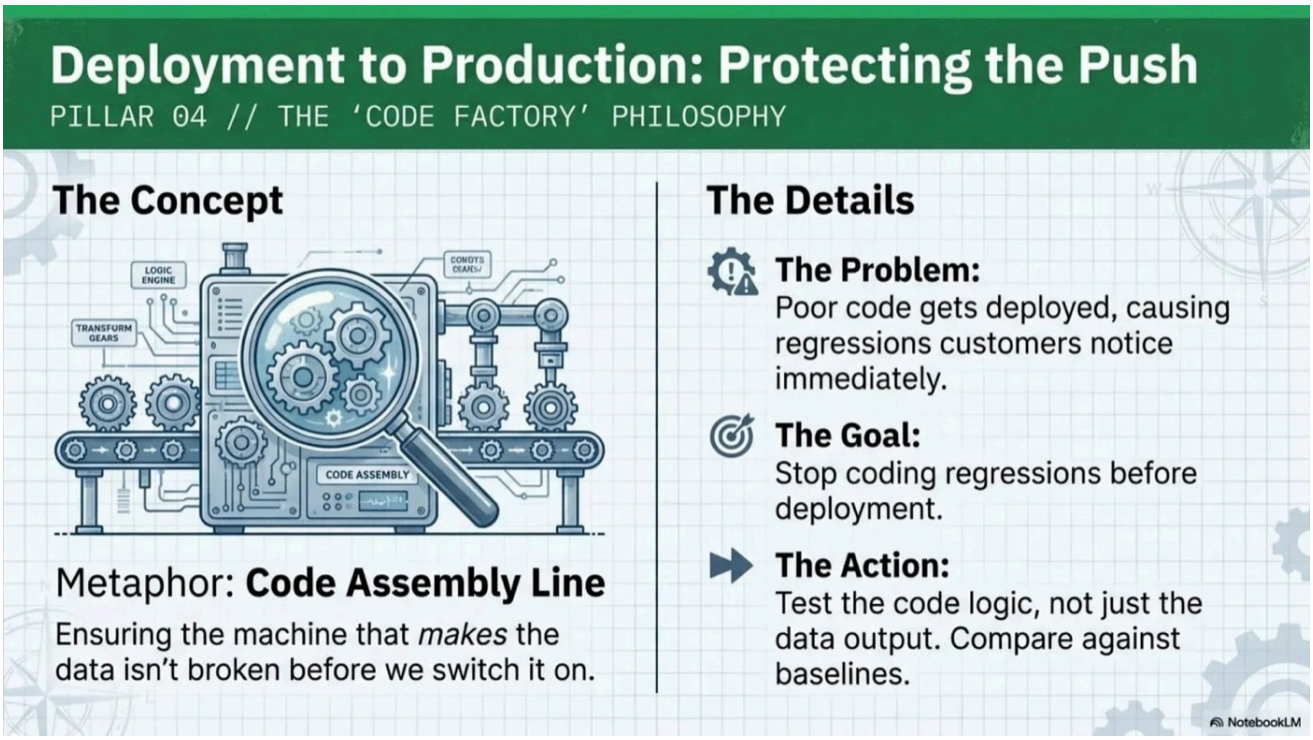
But data is only half of it. You also have tools acting on the data, and tools break in ways your data tests won't catch. Your Airflow job hangs for three hours for no obvious reason. Your reporting tool's scheduled refresh silently didn't run. So the necessary and sufficient condition for no customer-visible errors is this: check the data and monitor the tools acting on it. Every tool should be checked for errors and for timing.

You have two ways to watch the tools. You can use a test-aware orchestrator with built-in monitoring, so the checks run as part of the tool itself. Or, when you have several orchestrators (one loads data, another runs the bronze ETL, another handles gold, and a script kicks off the Power BI refresh), you need something sitting above all of them. That's what a data journey observability tool does. It watches every step across every tool and gives you one place to see the failure. I'm not a believer that data lineage solves this. Lineage is a blueprint, and what you need in production is a fire alarm control panel that says, "Here's the problem, here's where it stopped." You consult the blueprint after the alarm goes off to diagnose. You don't stare at a blueprint waiting for a fire.

The limitation is that this approach takes more work, because you have to integrate both the data and the tools into your orchestration. The benefit is the one that actually matters to your customer: you stop visible errors in data, reports, and AI models before anyone sees them.

Point four: deployment. The code factory

The first three points are about data. This one is about code.



You have a second manufacturing line, and it isn't making data; it's making the transformations that produce the data. This is the code factory, and the job is to prove your code still works before it ships. Specifically, prove that the new code you wrote works and that it didn't break the code that was already there. Regressions.

Most teams do almost nothing here. They hope it works. A good engineer might write a unit test. That does very little to prevent a regression, because the bug usually isn't in a single function; it's in what happens when the column you renamed flows three layers downstream.

Here's the concrete failure mode. You rename a column in Silver to fix a misspelling. Simple change. But some process in gold knows that column name, so it breaks. And some reports read the gold column, so the report breaks. You won't see any of that in a unit test. You'll see it from your customer.

The fix is full regression testing across the entire Medallion in your dev and QA environments. You run the same bronze, silver, and gold pipeline you run in production, but with test data pulled from production instead of live data. Maybe a filtered slice, maybe a zero-copy clone. Your data tests pull double duty here. The same tests that check data quality in production now verify that your code change didn't move the numbers in an unintended way. Think of it as today's code against yesterday's data, where production is yesterday's code against today's data.

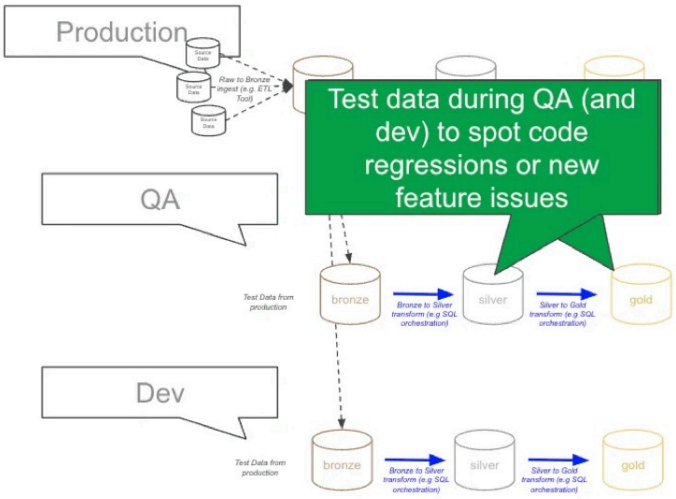
Development Testing in CI/CD

What? During your CI/CD process (jenkins, etc.) run data quality testing to check for new feature issues and regressions

How? Run data testing in qa on dev environments based on test data

Benefit: Stop coding regression before they get deployed to production

Limitation: Needs test data



This is the highest-impact and highest-complexity point. The payoff is speed. Good regression coverage lets you ship smaller changes more often with less risk, and that matters more as your team grows. One person keeping the whole pipeline in their head can wing it. Three people can't, and AI agents writing transformations definitely can't. The marker of a high-performing data engineering team is that you can hand a 22-year-old the keys, let them rename a column, and have a big red light go off that says "you broke a downstream report" before it ever reaches prod. That red light is regression testing.

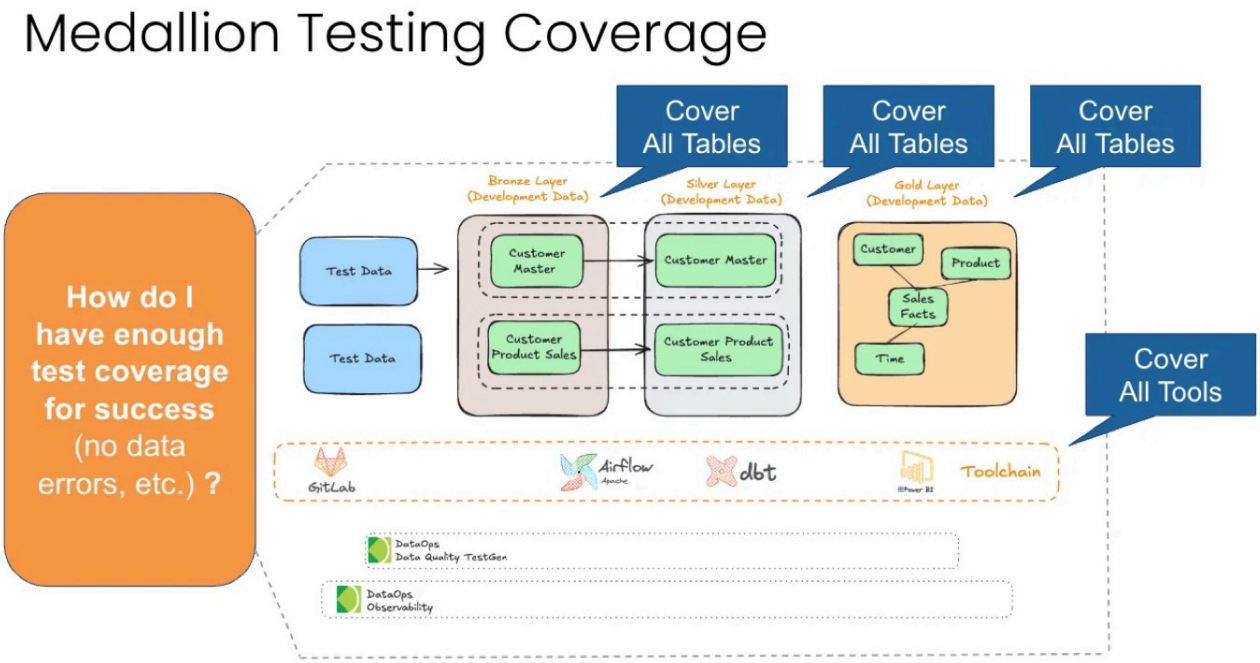
You need a few things to get there. Good test data. A development environment that looks like production from a DevOps standpoint. A real CI/CD process that moves code from one environment to the next. And regression tests across every layer of the Medallion. Unit tests and manual business reviews won't cut it, and you will get screwed if you rely on them. The reason to do the work is the 1:10:100 rule: catching a bug here costs about 1/100 of what it costs in production, where you're rebuilding the system and apologizing to people.

How much testing is enough?

Partial coverage is better than nothing, but it keeps the problems coming. If you test one table out of a hundred, you've covered one table, and you're still praying about the other ninety-nine. There are gaps in depth versus breadth, prevention versus detection, and runtime versus dev time, and you can't close them by testing your three favorite tables.

So the standard is simple and common sense. Cover all the tables in bronze, silver, and gold. Cover all the tools. Concretely, every table should have tests (two to three per table, covering freshness, volume, and schema at a minimum). Every column in every table should have tests (at least two per column). Every significant business metric, such as how your customer sees sales by region or production volume,

should have at least one custom test. And every tool that touches data should be checked for errors and for timing: at least one check per tool for errors in the logs, one check per job for task status, and one check per job for timing and duration.



Now do the math on a real Medallion. Say you have three database levels, three tools (StreamSets, Airflow, dbt), and five or six jobs. L1 has 100 tables averaging 10 columns, so that's 200 table tests and 2,000 column tests. L2 is the same, another 200 and 2,000. L3 has fewer tables but more columns, call it 20 table tests and 600 column tests, plus a handful of business metric tests. Three tools mean three error checks. Six jobs mean twelve timing checks.

Example: Test Coverage Counts Medallion

This Medallion Architecture Has **Three Database Levels** (L1, L2, L3), **Three Tools** (Streamsets, Airflow, dbt), And **Five Jobs/Workflows** to Monitor

- *L1 - 100 tables + 10 Columns*: Minimum 200 Table Tests and 2000 Column Tests
- *L2 - 100 tables + 10 Columns*: Minimum 200 Table Tests and 2000 Column Tests
- *L3 - 10 tables + 30 Columns*: Minimum 20 Table Tests and 600 Column Tests, Plus a handful of Domain Specific Business Metric Tests
- *3 Tools*: 3 Checks For Errors
- *6 Jobs*: 12 Checks For Timing and Duration

Add it up, and you're looking at roughly 5,000 data tests. OMG, basically. That number is exactly why people give up and go back to praying.

Example: Test Coverage Counts Medallion

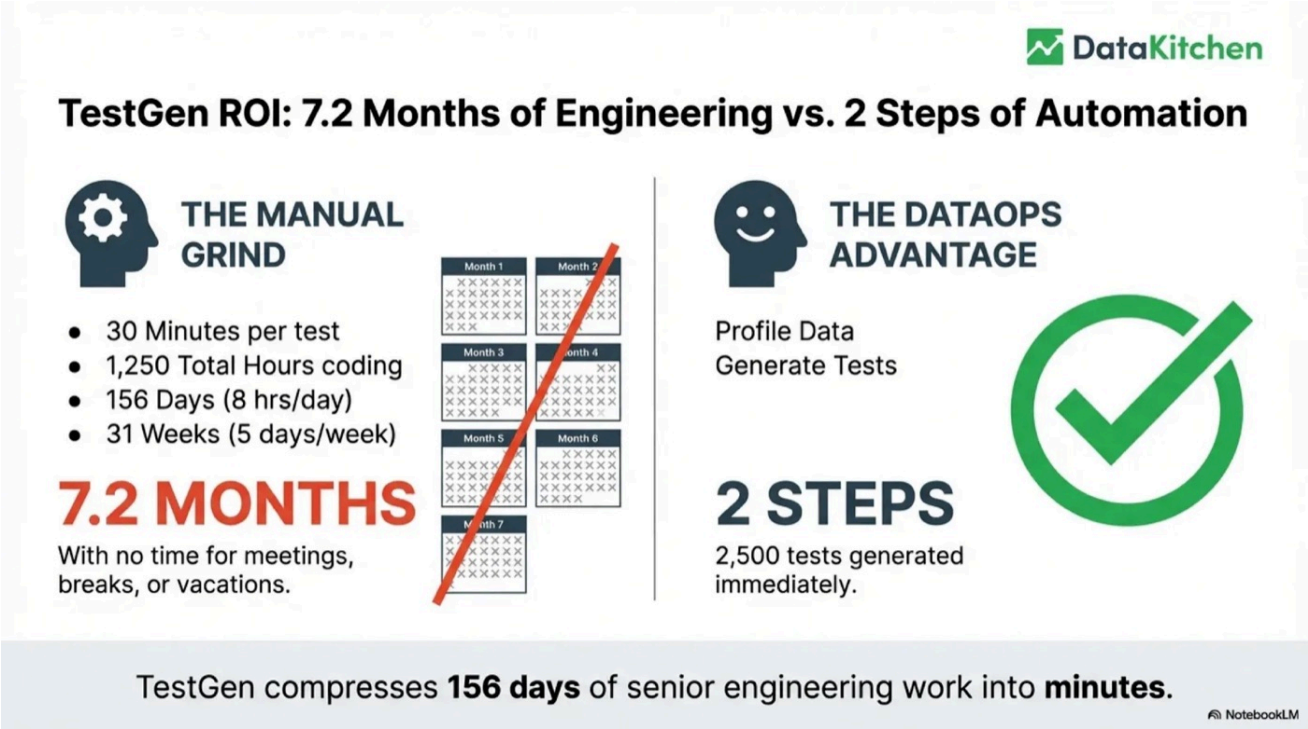
This Medallion Architecture Has **Three Database Levels** (L1, L2, L3), **Three Tools** (Streamsets, Airflow, dbt), And **Five Jobs/Workflows** to Monitor

5,000 Data Tests ?!? OMG

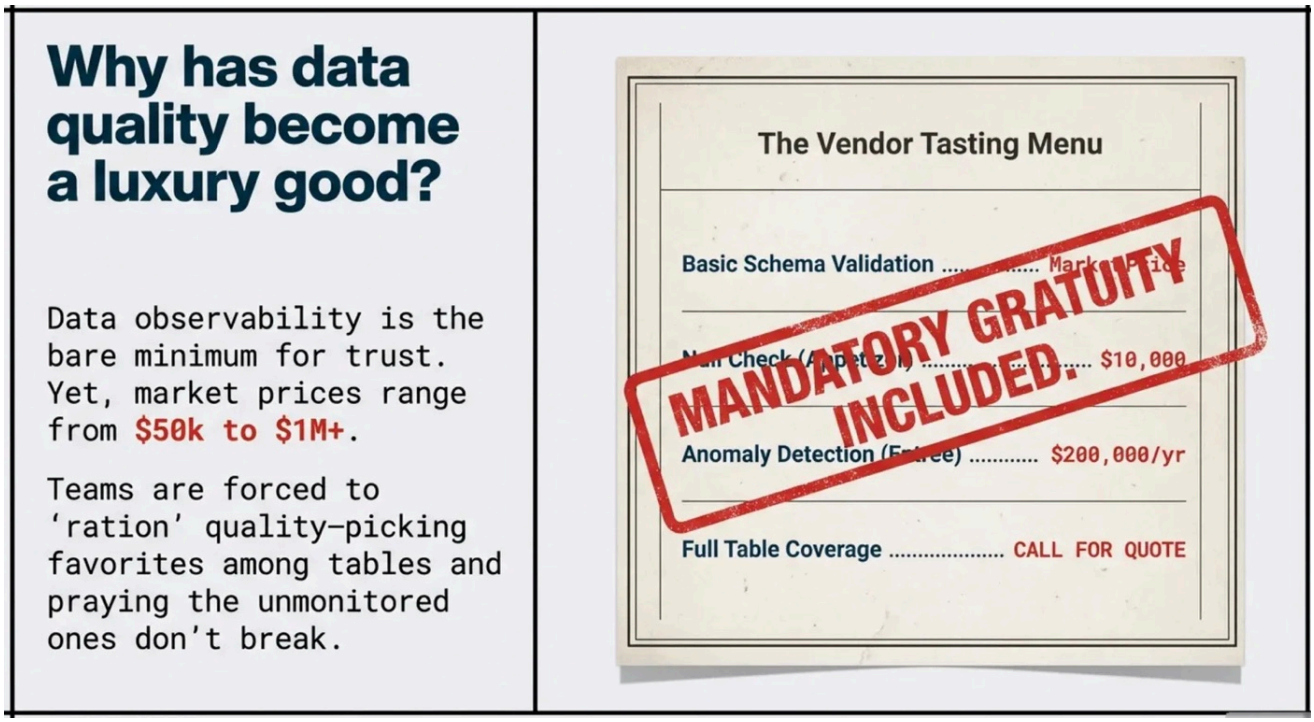
The cost of doing it the slow way

Those tests are a lot of typing. Take just half that load, 2,500 tests. At 30 minutes per test, that's 1,250 hours of coding, 156 working days, 7.2 months of a senior engineer doing nothing but writing tests, with no time for meetings, breaks, or vacations. That's the manual grind, and it's why "I don't have time" was

the most common answer in our survey.



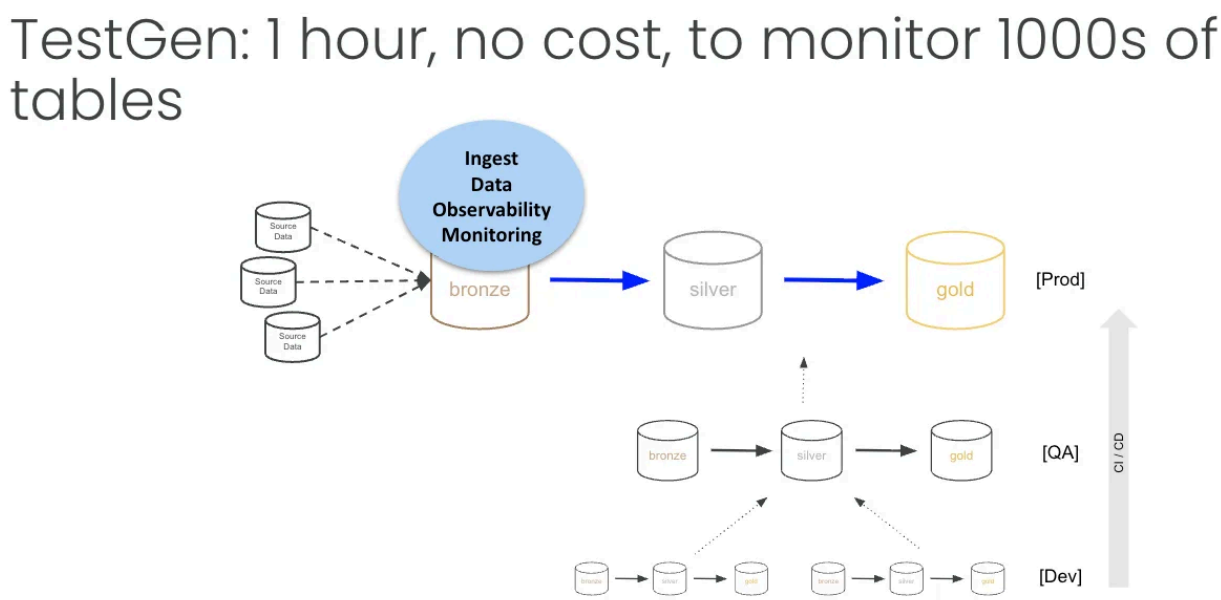
The other cost is the vendor tax. We studied how data observability and data quality vendors price, and most of them charge by usage: per credit, per table, per row scanned. The more tables you monitor and the more tests you run, the more you pay. The more you test, the more it costs. That's backward. You should be testing your source data, testing in production, and testing in development, all the time, everywhere, because every error you prevent buys back hours. Punishing the right behavior with a bigger bill makes no sense. Yet market prices for this run from $50k to well over $1M a year, and teams end up rationing quality, picking favorite tables to monitor, and praying the rest hold.



Where to start

Don't try to build all four points on day one. Start at bronze and forget everything else.

All you need is a read-only connection to your bronze database. Turn on observability monitoring: arrival frequency, volume, schema, and a couple of metrics. That's it. This takes about an hour, costs nothing, and monitors as many tables as you have, whether that's a hundred or a few thousand. You go from blind to watching the whole herd in an afternoon.



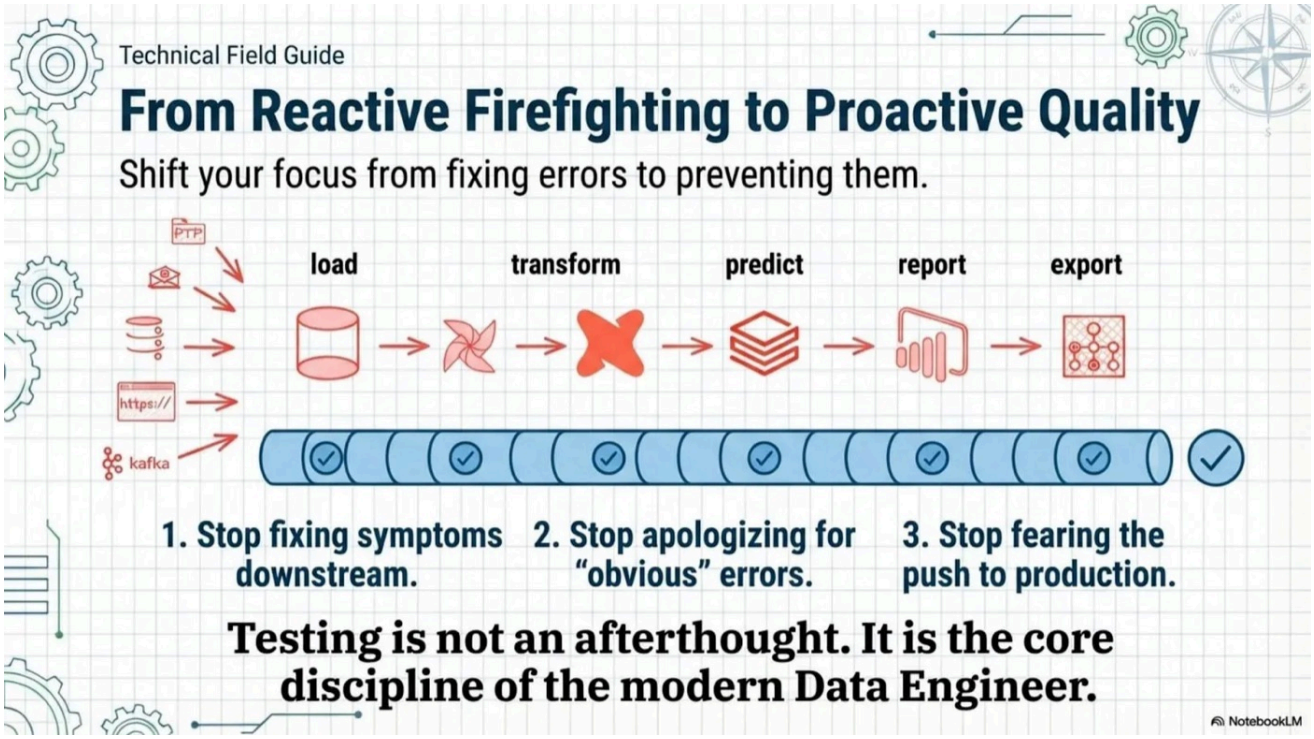
Second, build a data quality dashboard. Same single connection, no cost, and you get a score you can actually act on. Those two moves, monitoring on bronze and one quality dashboard, are the highest return starting point I know.

This works regardless of your stack. If you're AWS native and your "Medallion" is S3 buckets moving through Lambdas into Postgres, the principle doesn't change. You still need automated checks that can stop bad data, and you have options: write tests as step functions that fire after data lands, point a query engine like Redshift Spectrum at the buckets, or, in the simplest case, write your errors to a table or a bucket and read it once an hour. The pattern of S3 to S3 is everywhere, and the rule is the same. Take hope out of the system and put a check in its place.

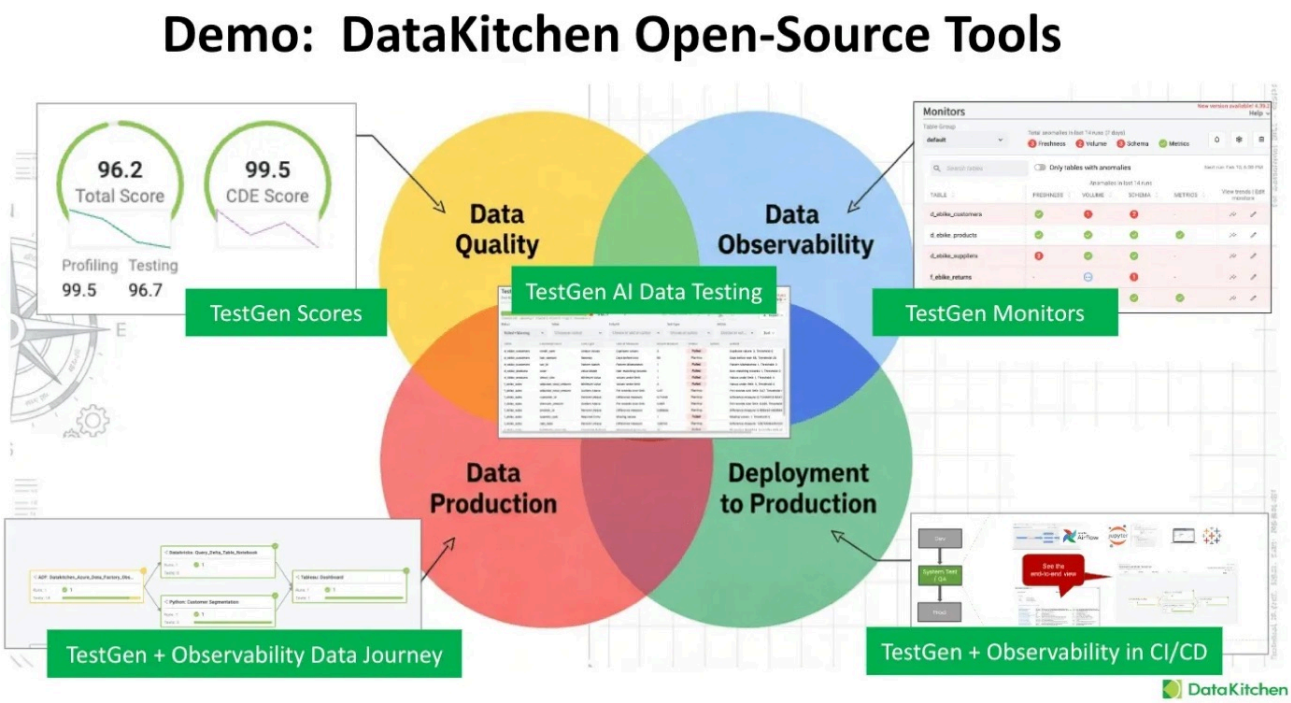
The reason all of this is worth it comes back to the 1:10:100 rule. Prevention costs a dollar, remediation costs ten, and a failure in front of your customer costs a hundred. Every test you write is a gift to your future self. Shift left to catch problems in development, shift down to catch them earlier in production, cover every table, cover every tool. Do that, and you end up running pipelines with almost no errors, which frees you from the burden and the embarrassment that eats most of a data engineer's week.

Stop being reactive

Testing is not an afterthought you bolt on after the dashboard is already wrong. It's the core discipline of the modern data engineer working in a Medallion architecture. Stop fixing symptoms downstream. Stop apologizing for "obvious" errors. Stop fearing the push to production. Stop being an ostrich with data issues.



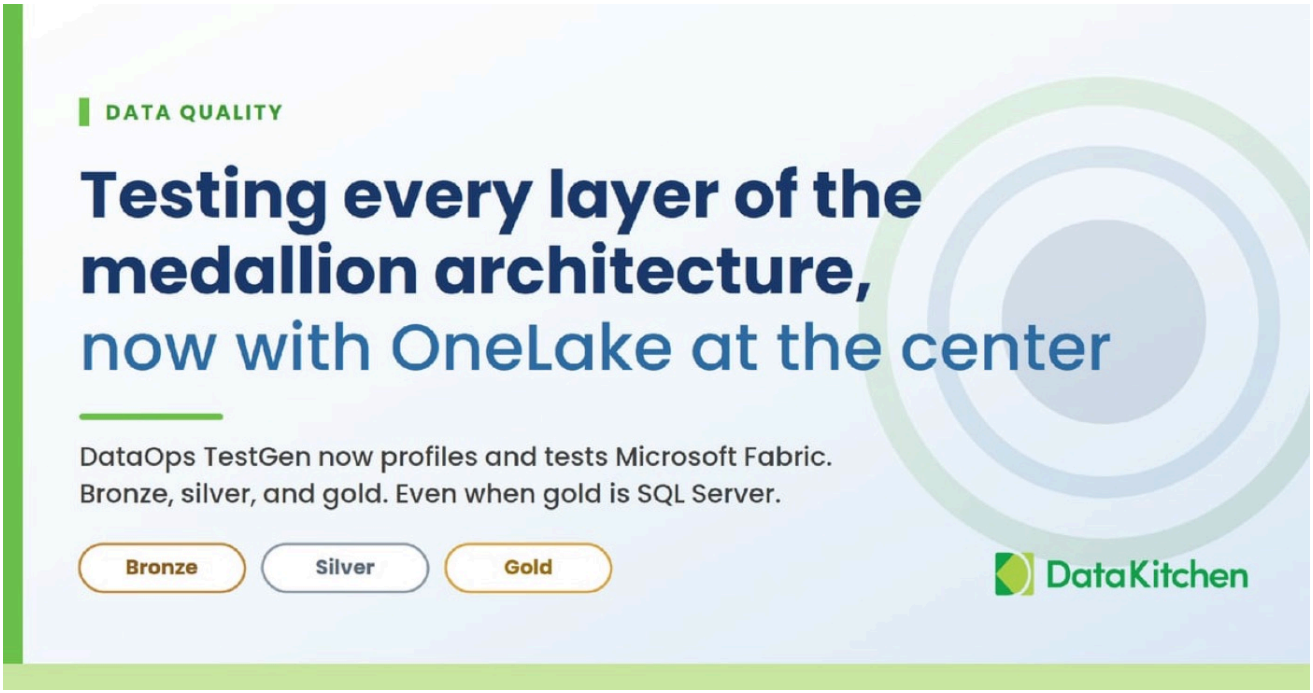
There are four places to test, and you need all four. Persuade at the source so your data is fit for purpose. Poll at ingestion so you catch what arrives broken. Put tripwires through production so nothing bad reaches gold. Run regression tests in CI/CD so your code changes don't break what already works. The tools exist, the tests can write themselves, and you can stand up the first point in an hour. Point it at your bronze database today and start watching everything.



This is the testing chapter of a longer reading order. Where to Actually Start with DataOps in 2026 covers what to read before it and what changes once the errors stop.

Testing Every Layer of the Medallion Architecture, Now with OneLake at the Center

Read the original article at: https://datakitchen.io/blog/testgen-onelake-medallion-architecture-testing/



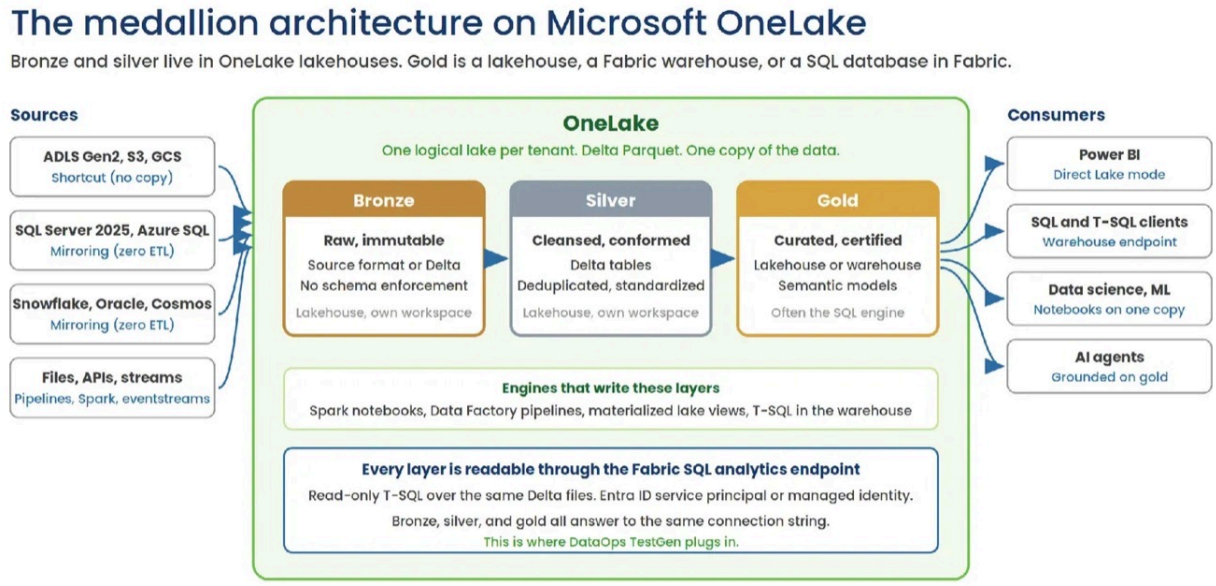
That's the failure mode nobody puts on the architecture diagram. A notebook runs, a Delta table gets written, a semantic model refreshes, and a number lands in front of an executive. Every step succeeded. The number is still wrong, because a source system changed a column type on Tuesday and nothing in the chain was watching.

If you haven't touched Fabric yet, that endpoint is the part worth knowing. Every lakehouse in OneLake automatically gets a read-only T-SQL endpoint over its Delta files. It speaks TDS, the same wire protocol SQL Server uses. Open SSMS or Azure Data Studio, paste in the connection string Fabric gives you, and run a SELECT. That's how TestGen reads your lakehouse, and it's why testing one asks nothing new of a SQL Server shop.

And if your gold layer is a SQL Server rather than a lakehouse, keep reading. That case is more common than the Fabric diagrams suggest, and it gets its own section below.

Nothing about Fabric sits behind a paywall here. Install the free build, point it at your lakehouse, and profile it this afternoon.

That's the announcement. The rest of this post is the part that matters: where the tests go, and why OneLake changes the answer.



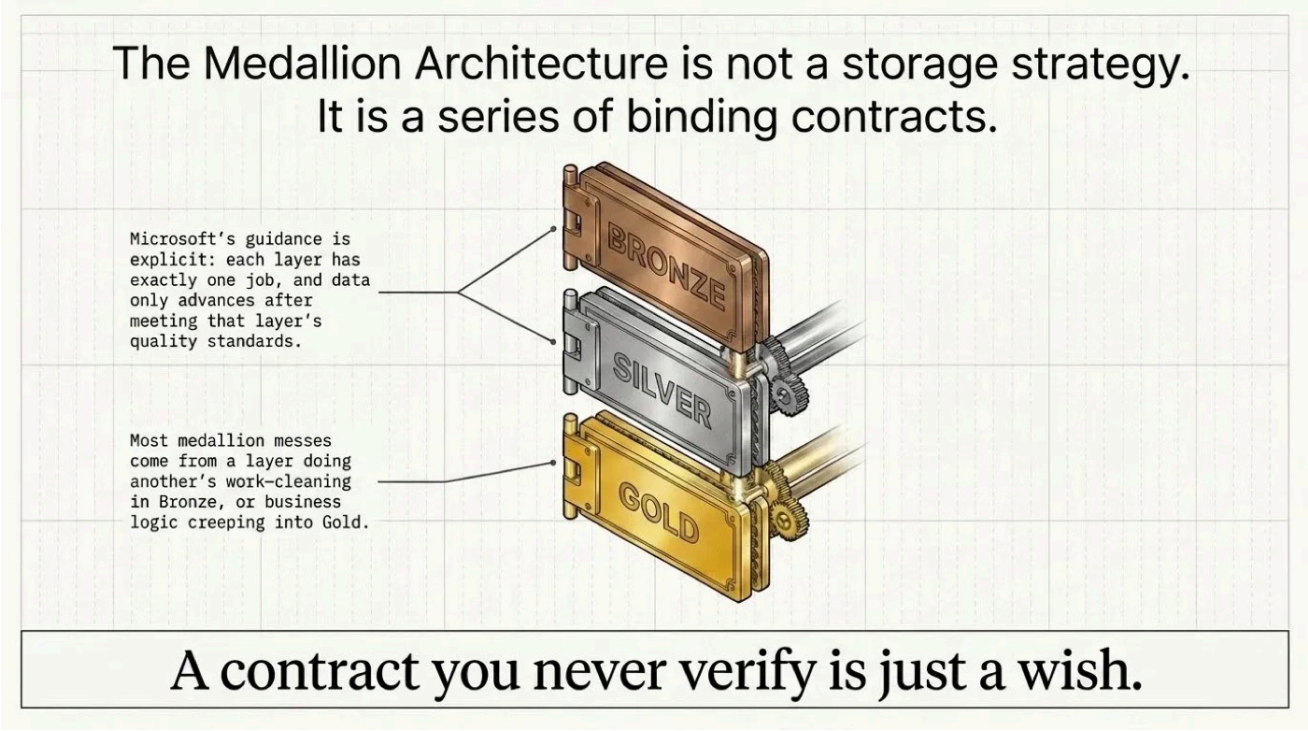
Source: Microsoft Learn, "Understand medallion architecture for Fabric with OneLake" and "OneLake patterns and foundational capabilities."
Diagram by DataKitchen.

What Microsoft's current medallion guidance actually says

The medallion pattern started at Databricks. Microsoft adopted it, and the guidance has moved far beyond "three folders in a lake."

Read Microsoft's own words in Understand medallion architecture for Fabric with OneLake. OneLake is the data lake for Fabric. Every tenant automatically gets one, and it stores all your analytics data. You build the medallion architecture by creating lakehouses inside it. Bronze holds raw data in its source format. Silver and gold hold Delta tables. Microsoft recommends putting each layer in its own workspace, not just its own lakehouse, so you get real governance at the layer boundary.

Then comes the part people miss. Microsoft describes two deployment patterns. In the first, every layer is a lakehouse, and your business users read through the SQL analytics endpoint. In the second, bronze and silver are lakehouses and **gold is a data warehouse**, read through the warehouse endpoint. Microsoft supports both. You'll see both in the wild.



So OneLake is bronze and silver, almost always. Gold is OneLake sometimes, and a SQL engine the rest of the time.

Microsoft's OneLake patterns guide goes further and treats the medallion architecture as one of five patterns you combine, not a monolith. It puts the idea plainly: each layer is a contract with downstream consumers, and data only advances to the next layer after it meets that layer's quality standards.

Read that sentence again. It's a testing statement dressed up as an architecture statement. A contract you never verify is a wish.

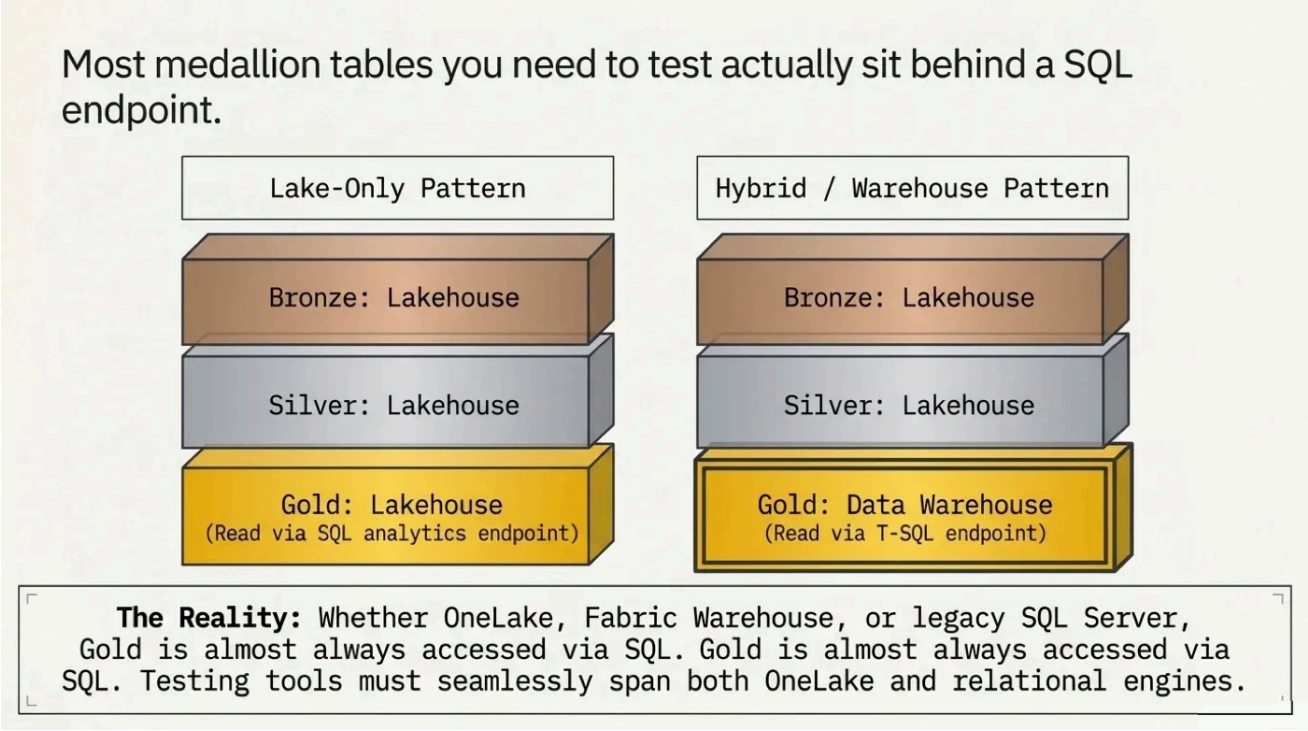
Three Fabric items get called databases, and they are not the same thing

This trips up every SQL Server person on their first Fabric project, so here it is plainly.

A **lakehouse** holds Delta tables in OneLake. Spark and pipelines write it. You read it through the SQL analytics endpoint, which is read-only for table data but supports views, functions, stored procedures, and SQL security objects. This is where bronze and silver usually live.

A **warehouse** is a full T-SQL engine with reads and writes, stored procedures, and multi-table transactions. It still stores Delta in OneLake. It's the closest thing to a Synapse dedicated pool you'll find in Fabric, and it's where gold often lives.

A **SQL database in Fabric** is different from both of them. It's an operational OLTP engine within Fabric, from the same family as Azure SQL Database, and Fabric automatically replicates its data to OneLake as Delta. It's not Azure SQL, even though it feels like it. Microsoft groups it with Cosmos DB under Fabric Databases.



All three land Delta files in OneLake, and all three answer T-SQL over TDS. That's what makes one testing tool sufficient for the whole estate.

Three other things in the current guidance change how you should test.

Shortcuts and mirroring mean you often didn't build bronze

Microsoft tells you to point a shortcut at ADLS Gen2, S3, or Google Cloud Storage instead of copying data in. For operational databases, mirroring creates a synchronized Delta copy with no ETL you wrote. The supported list now includes Azure SQL Database, Cosmos DB, Snowflake, PostgreSQL, SQL Server 2025, Oracle, and BigQuery. This is fast, cheap, and good. It also means bronze arrives through a pipe you don't control and can't instrument. You didn't write the transform, so you get no logs from it. Profiling and testing are the only way to see what showed up, and whether it showed up at all.

Materialized lake views moved the transform into the platform

Materialized lake views let you declare bronze-to-silver-to-gold transformations in Spark SQL and let Fabric handle dependency ordering and refresh. Follow Microsoft's tutorial, and you'll land bronze as a shortcut and run one notebook. Outcomes include silver and gold, data quality constraints, an auto-generated lineage graph, a refresh schedule, run history, and a data quality report tab. That's a real improvement over hand-built pipelines, and you should use it.

Notice what it covers, though. The constraints are the ones you thought to write, checked on rows inside a single transform, on the views that ran. It doesn't tell you that silver has 40 percent fewer distinct customers than bronze. It doesn't flag that a product code format drifted last week in a column nobody

constrained. It says nothing about a bronze table that's been stale for eight hours since a source change, because a view that never fired has no failed constraint to report. Declarative transforms tell you a rule broke. Profiling tells you what's actually in the table, including the parts you never wrote a rule for.

The tutorial is worth reading for a second reason. It builds bronze, silver, and gold as three schemas within a single lakehouse, whereas the architecture guidance recommends a separate workspace per layer. Both are real. Your layout determines how you scope table groups in TestGen, where a table group is the set of tables in a single schema that you profile and test together. Settle the layout before you connect.

Direct Lake removed the copy that used to buy you time

With Direct Lake mode, Power BI reads gold straight from OneLake. No import, no scheduled refresh. The gap between a bad write and a bad dashboard used to be hours. Now it's seconds. Whatever is in gold is what the business sees, immediately.

One copy of the data. Many engines. No refresh buffer. Quality has to be a property of the table, not a feature of whichever tool happened to write it.

Sometimes gold is SQL Server, and that's fine

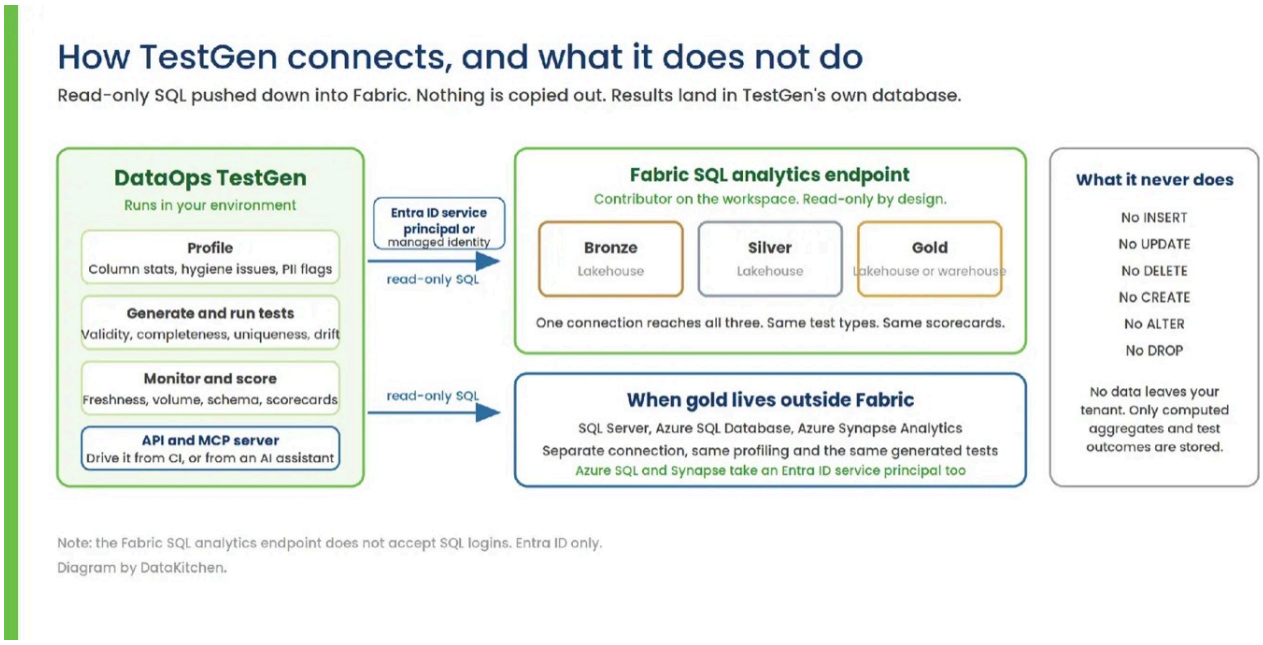
Microsoft's Fabric Data Warehouse team went further in August 2026. Their post, "Choosing your medallion pattern in Fabric Data Warehouse," opens by telling you that the question you're probably asking is the wrong one. Not "lakehouse or warehouse." Instead: how much Spark do you actually need?

Their answer comes in two shapes. Structured, SQL-first workloads run the whole thing in the warehouse. Bronze, silver, and gold, all T-SQL. Unstructured or engineering-heavy workloads go hybrid, land bronze in a lakehouse, and serve gold from the warehouse. Their words, not ours: gold always comes from the warehouse. Not usually. Always.

Read that as a data-quality person, and it changes the shape of the job. In one recommended pattern, all three layers are relational tables you query with T-SQL. In the other, the layer everyone reports from is relational. Either way, most of the medallion tables you need to test sit behind a SQL endpoint.

So your gold may not live in the lake at all. It's a Fabric warehouse, a SQL database in Fabric, or a SQL Server your finance team has trusted since 2014 and will not give up.

You don't have to pick a side. TestGen supports Microsoft SQL Server, Azure SQL Database, and Azure Synapse Analytics, as well as OneLake. The same release added Entra ID service principal authentication for Azure SQL and Synapse, so your gold layer authenticates the same way your lake does. Point TestGen at bronze and silver through the Fabric SQL analytics endpoint. Point it at gold wherever gold actually lives. One set of quality scores covers the whole chain.



Two notes on the connection, because this is where setups stall. The Fabric SQL analytics endpoint doesn't accept SQL logins, so plan on Entra ID from the start. And grant the least privilege that actually works. On a production tenant that means item-level access on the lakehouse, or a Viewer role plus the data permissions the endpoint needs. Contributor reads the endpoint with no item-level permissions to juggle, which makes it the fast way to get a first profile running in development, and more access than the job requires.

That second one deserves a warning if you're coming from SQL Server. Two permission systems apply here, and they govern different doors to the same files. SQL permissions govern SQL access, so a GRANT through the endpoint works the way you expect. Workspace and OneLake permissions govern every other path to the same files, Spark and direct OneLake access included, which is why the workspace layer comes into it at all. Contributor spans both. That is more than the job needs on a production database, so your security reviewer will notice, and they'll be right. The sentence that gets you through the review is the one about egress: TestGen pushes read-only SQL into the endpoint and pulls back aggregates, so no rows leave the tenant. Give the service principal access only to the workspaces you intend to profile.

TestGen pushes read-only SQL into your tenant. It runs no INSERT, UPDATE, DELETE, CREATE, ALTER, or DROP. No data comes out. TestGen stores only computed aggregates and test outcomes in its own database.

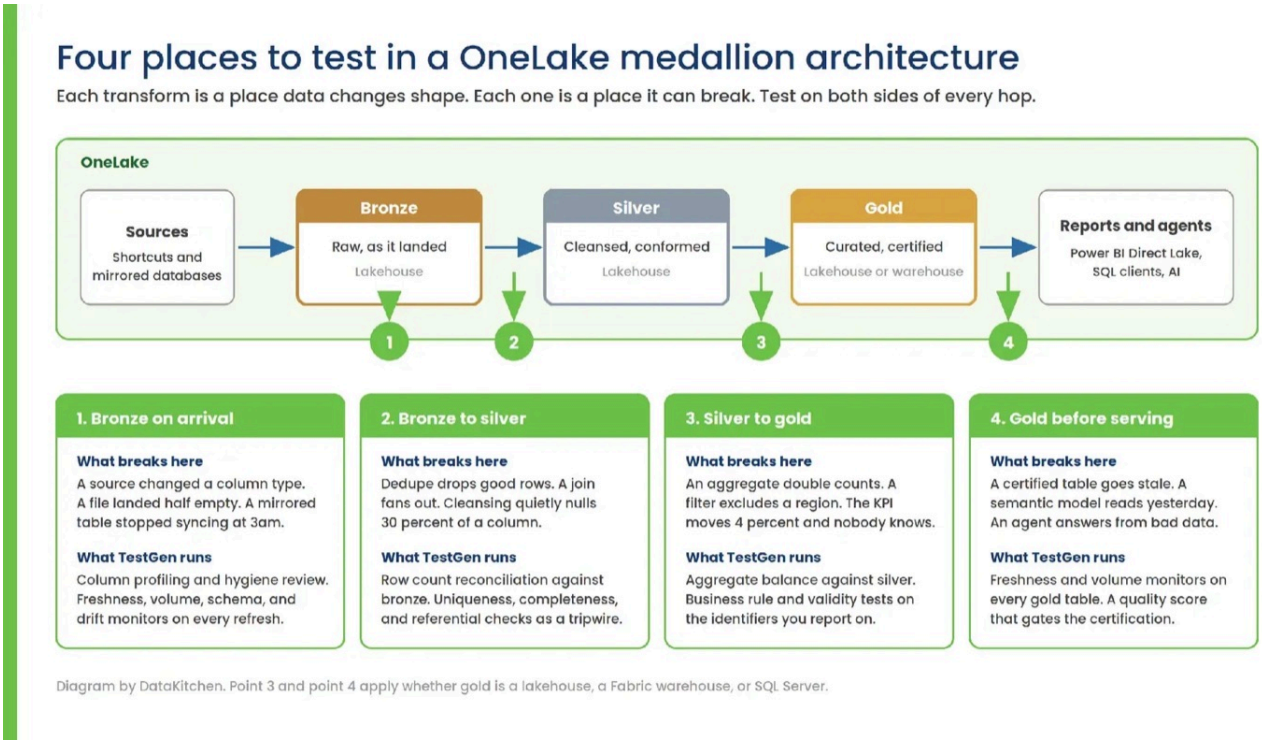
The four places to test

We've written about this before. The Race for Data Quality in a Medallion Architecture walks through the quality problem at each layer. Data Quality Test Coverage in a Medallion Data Architecture does the coverage math. Data Production Tripwires shows how to stop a bad layer from feeding the next one. Our

medallion solution page pulls the pieces together, and the webinar, The Four Points in Your Medallion Architecture Where Data Testing Really Matters, covers it end-to-end.

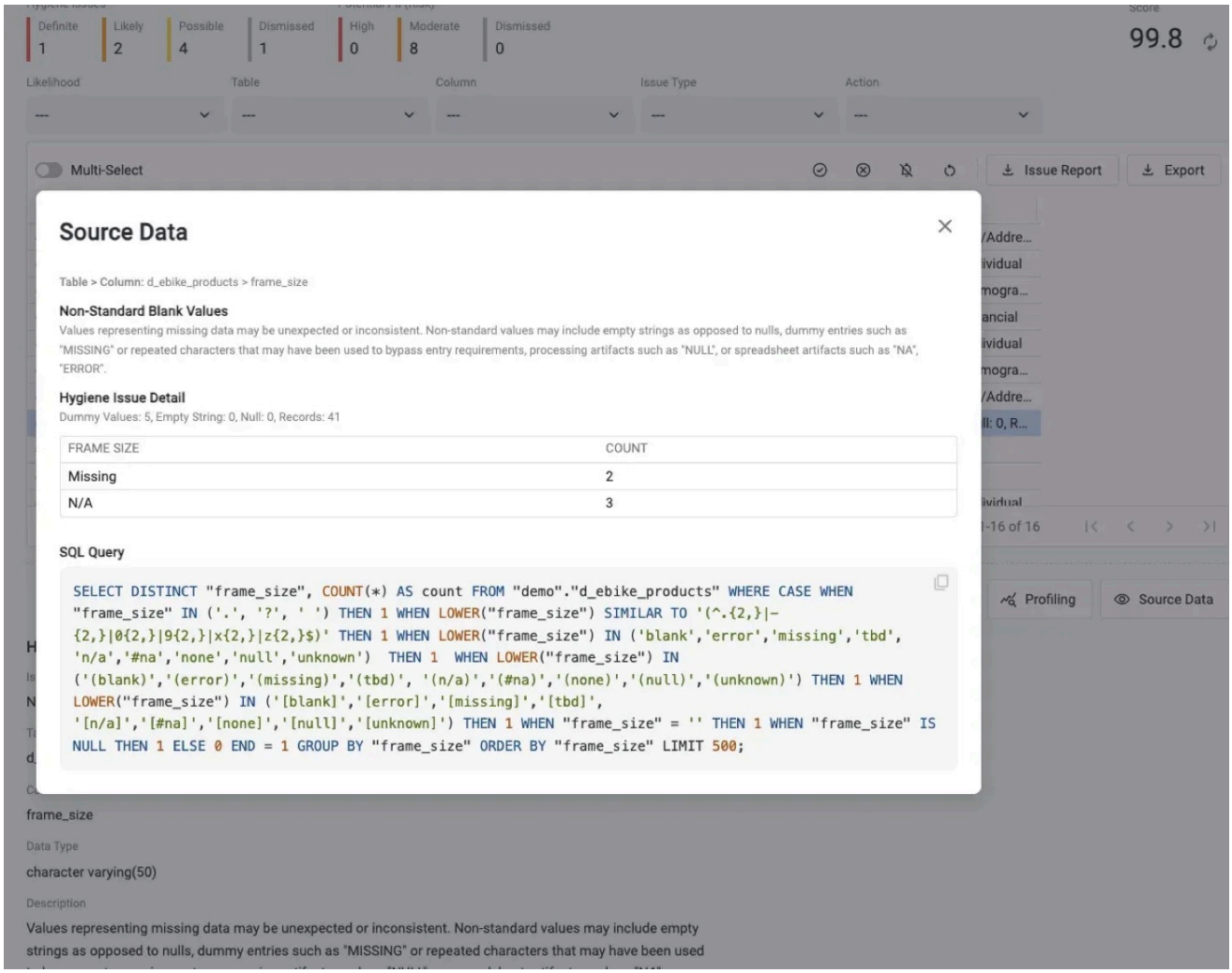
Microsoft's warehouse series puts the principle well in part two: each layer has exactly one job, and most medallion messes arise when a layer does another's work. Cleaning in bronze. Business logic creeping into gold. Tests at the boundary are how you find out a layer picked up a job that wasn't its own.

Here is the same map, drawn on OneLake.



Bronze, on arrival

This is where bad data enters. A source changed a column type. A file landed half empty. A mirrored table stopped syncing at 3 a.m., and the sync status looks fine. Point TestGen at your bronze lakehouse and let it profile every column: null rates, duplicates, value ranges, format mismatches, PII risk. You get a hygiene review before anything moves downstream. We pointed it at 600,000 rows of Boston building permit data once, and it found 18 data hygiene issues in about three minutes, including a city column with a stray slash.



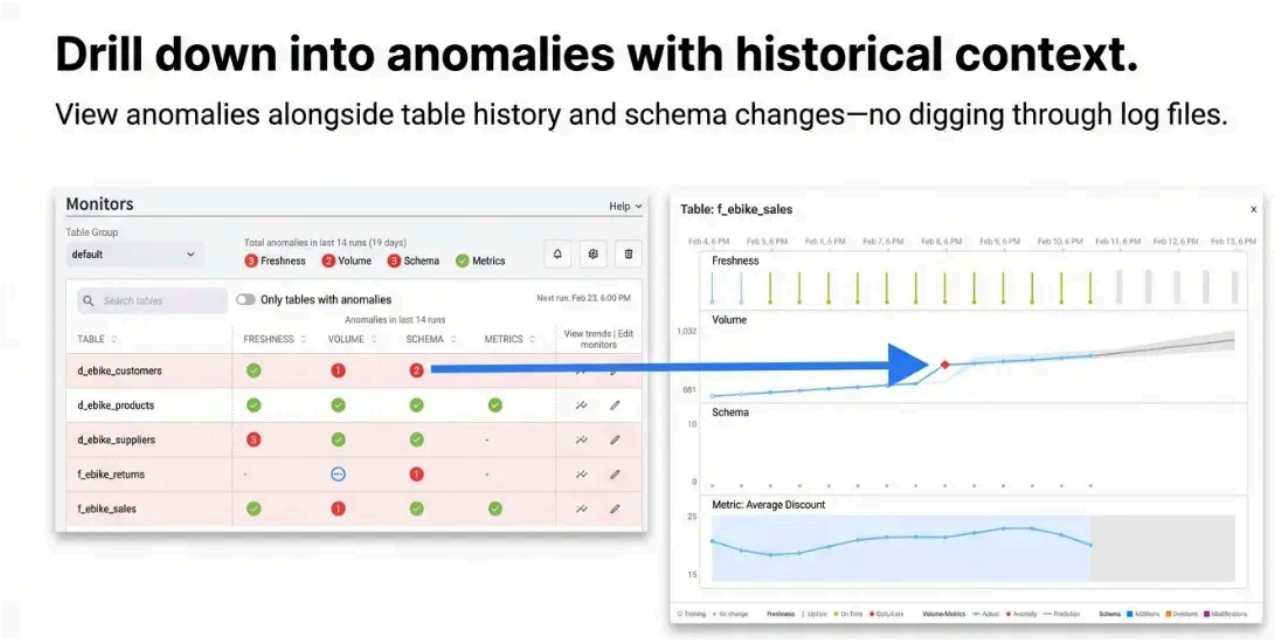
Don’t fix any of it there. Microsoft is explicit that bronze has one job: ingest raw source data with no cleansing, and that you shouldn’t filter out bad records in bronze because all cleaning happens in silver. Good rule. Keep it. It also makes bronze the one layer where you need to see problems without stopping them. Profiling does exactly that. You don’t gate bronze; you measure it, so you know what silver is about to inherit. Their advice to add an ingestion timestamp and a source filename column is worth taking for a second reason too. Those are the columns a freshness test keys on.

Then four monitors run on every refresh. Did we get something new? Did we get enough of it? Does the schema still fit? Do the values still make sense? Catching a problem here costs about a dollar a record. Catching it in a board deck costs a hundred.

Bronze to silver

Cleansing is where good rows die quietly. A dedupe rule drops legitimate records. A join fans out. A cast turns 30 percent of a column into nulls without raising anything. Reconcile silver row counts against bronze, then run uniqueness, completeness, and referential tests on the result.

Wire it as a tripwire, a check that sits between the two transforms and stops the second one when the first one produced garbage. How you build it depends on your orchestrator. In Airflow, a task runs the suite and exits with a non-zero status. In a Data Factory or Fabric pipeline, a Web activity calls the TestGen API to run the suite, and you branch on the result so that the silver-to-gold activity fires only on success. Either way, the shape is the same. The next transform doesn't run. Gold stays on the last good data, and you get a failed activity in the run history instead of an incorrect number on a dashboard.

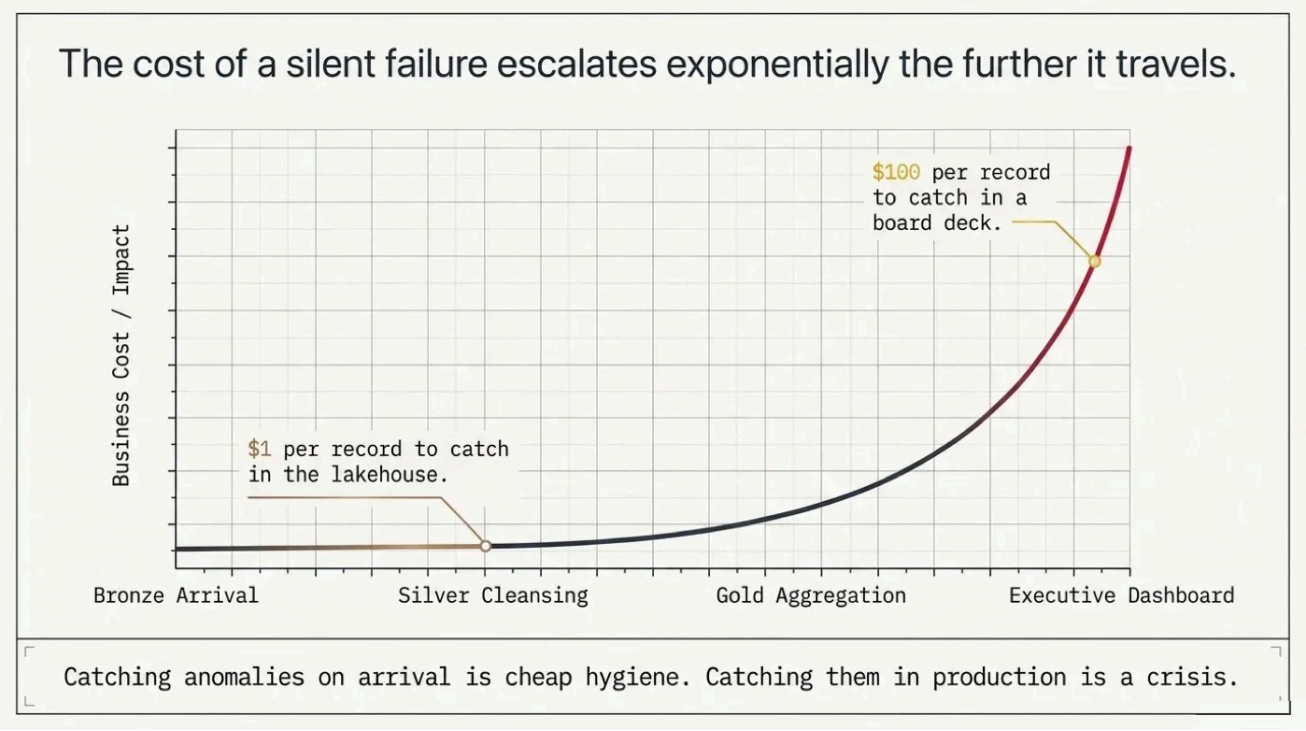


Silver to gold

This is aggregation, and aggregation lies with a straight face. A double-counted join inflates revenue by 4 percent. A filter silently excludes a region. No errors. Balance your gold aggregates against silver, and run business rule tests on the identifiers you actually report on. Those are the codes and keys your business runs on, and the generic test types don't know their formats. If yours are proprietary, TestGen Enterprise accepts uploaded test types, so you don't have to wait for a release to test a format only your company uses.

Gold, before it serves anything

With Direct Lake, there's no buffer between a stale gold table and a wrong dashboard. Run freshness and volume monitors on every gold table and put a quality score in front of the certification. Microsoft tells you to endorse gold tables as certified in the OneLake catalog. Certify on evidence, not on a meeting.



The coverage problem is real, and it's arithmetic

Say you have 100 tables in each layer. Bronze needs roughly 2,200 tests. Silver needs another 2,200. Gold adds about 620, plus your custom business metric tests. Written by hand, that's over a year of full-time work.

So it doesn't get written. You write three tests on the dashboard table everybody complains about and call it coverage. Then the break happens in a bronze table nobody thought about. When we asked why data engineers don't test, Reddit gave us eight reasons, and every one of them was about effort rather than intent.

TestGen profiles each table and generates freshness, volume, schema, and drift tests based on what it finds. It re-baselines when you move from bronze to silver to gold, so the tests match the schema at that layer rather than the schema from two hops ago. A junior operator can run it. You get column-level coverage across every layer on day one.

SECTION 4

Monitoring and Observability

1. Data Observability: What to Monitor and Why It Matters
2. The Six Types of Data Quality Dashboards
3. Data Quality Alerting: Avoiding Alert Fatigue
4. Why Your Data Quality Dashboard Isn't Working And What to Do About It
5. Data Production Tripwires in Databricks: Stop Bad Data Before It Reaches Production

MONITORING AND OBSERVABILITY

Data Observability: What to Monitor and Why It Matters

Read the original article at: https://datakitchen.io/blog/the-five-use-cases-in-data-observability-overview/

Data observability is the capability to understand the health of your data systems from the outside, using the signals those systems emit. The term is borrowed from control theory and systems engineering, where observability describes the degree to which you can infer the internal state of a system from its external outputs. In data engineering, it means having enough instrumentation to answer three questions at any moment: Is my data arriving? Is it correct? Has anything changed that affects the people relying on it?

This chapter covers the five core use cases of data observability and goes deep on the two that have the most operational impact: detecting data downtime and understanding data lineage.

Five Use Cases of Data Observability

Data observability is not a single capability. It is a collection of practices applied at different points in the data lifecycle.

Use Cases: Why Should You Care?

	Data Observability Use Case	Benefits of Fixing
Data Evaluation:	New data set evaluation and cleansing	**Save time, lower errors, improve data quality**
Data Ingestion:	Updates to existing data sources; Data ingestion monitoring	**Find problem data quickly, save time, lower errors**
Production:	Monitoring of multi-tool, multi-data sets, multi-hop, data analytic production processes.	**Stop embarrassing customer errors, gain customer data trust, lower errors, improve team productivity**
Development:	Development Unit, Regression Tests, and Impact Assessment.	**Improve the speed and lower the risk of changes to production, less wasted time, improve productivity**
Data Migration	Checking data accuracy during Data Migration projects: 'Does It Match'?	**Save rework, improve schedule accuracy, productivity**

Data validation before production means testing data quality before a pipeline writes its output to any consumer-facing table. This is the shift-left approach described in an earlier chapter. Observability here means having visibility into what was checked, what passed, and what failed before any downstream system was affected.

Monitoring data ingestion means watching the inbound flow of data from source systems. Is the expected file arriving from the vendor? Is the Kafka topic receiving messages at the expected rate? Is the API returning well-formed responses? Ingestion monitoring catches failures at the entry point, before bad data enters the pipeline.

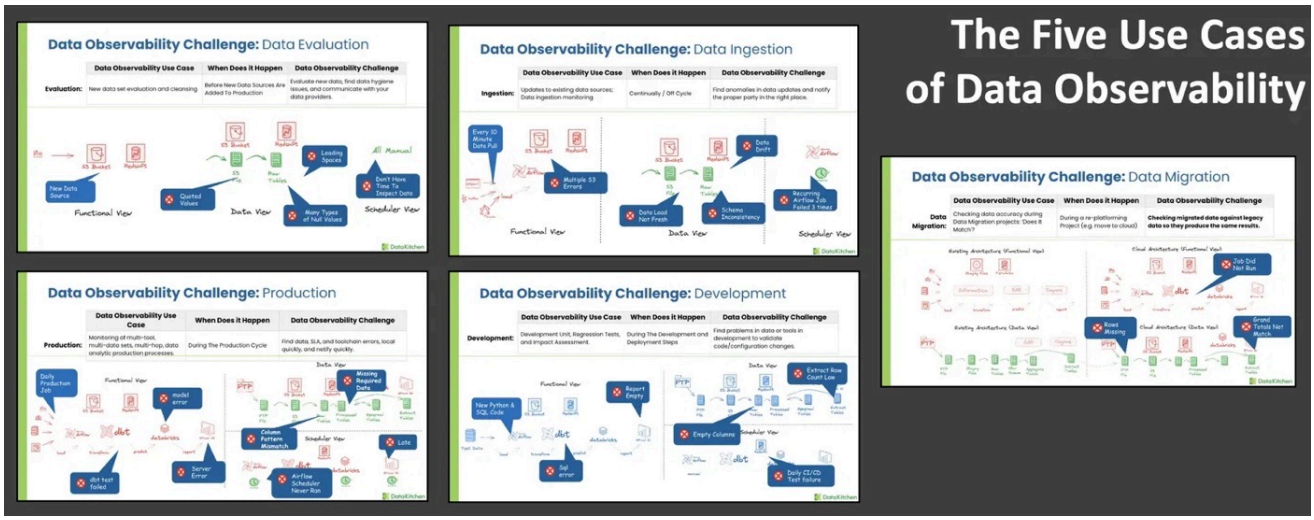
Overseeing production pipeline runs means tracking the execution health of running pipelines. Are jobs completing within their SLA windows? Are any tasks retrying excessively? Is job duration creeping upward over time in a way that suggests growing input volumes or degrading performance? This layer of observability is distinct from data quality: a pipeline can run successfully and produce bad data, and it can fail to run despite the data being fine.

Development and regression testing means using observability signals to validate that a code change does not degrade data quality. This includes profiling data distributions before and after a transformation change, comparing row counts across equivalent runs, and verifying that schema changes do not break downstream consumers. Observability in development is the foundation of safe deployment.

End-to-end data lineage means maintaining a complete map of how data flows through the system: which tables feed which other tables, which dashboards read from which tables, and which upstream sources feed each pipeline. Lineage is the foundation for root cause analysis and impact assessment.

The following two sections go deep on data downtime and data lineage because they are the most complex, the most consequential, and the most frequently under-instrumented.

Data Downtime: Definition, Detection, and Response



What Data Downtime Is

Data downtime is any period during which data is unavailable, inaccurate, or incomplete for the people and systems that depend on it. It is the data equivalent of a service outage. The website might be up. The database might be running. But if the data inside is stale, corrupted, or missing, the business impact is the same as if the systems were down.

Data downtime differs from infrastructure downtime in one important way: it is usually invisible. A server that is down triggers immediate alerts from infrastructure monitoring. A table that has been stale for six hours, or that is missing 15% of its expected rows, may not trigger any alert unless you have specifically instrumented for it.

What to Measure

Effective data downtime detection requires four categories of metrics:

Freshness metrics track the age of data. The primary metric is the time elapsed since the table's most recent update, measured against the table's defined SLA. Secondary metrics include the delay between source event time and table availability (end-to-end latency) and the frequency of update intervals over the past 7 and 30 days. Freshness metrics catch the most common form of data downtime: the pipeline that stopped running or ran but wrote nothing.

Volume metrics track the quantity of data. The primary metric is row count per time window (hourly or daily), compared against a rolling historical baseline. Secondary metrics include partition-level row counts for partitioned tables and byte-level size metrics for file-based storage. Volume metrics catch partial writes, mass deletions, and deduplication bugs.

Completeness metrics track the presence of expected values within rows. The primary metrics are null rates per column for required fields, measured as a percentage. Secondary metrics include the rate of placeholder or sentinel values (zero-valued amounts, epoch timestamps used as null substitutes, empty strings in non-nullable fields). Completeness metrics catch schema changes that produce nulls, application bugs that produce empty values, and upstream data entry failures.

Distribution metrics track the statistical shape of data over time. For numeric columns, this means monitoring mean, standard deviation, and percentile values. For categorical columns, this means monitoring the frequency distribution of allowed values. Distribution metrics catch subtle semantic changes that the other metric categories miss: a column whose values are all valid but whose distribution has shifted in a way that suggests a business logic change or an upstream data quality problem.

Setting Alert Thresholds

The challenge with data downtime alerting is balancing sensitivity against noise. Thresholds that are too tight produce alert fatigue; thresholds that are too loose miss real incidents.

For freshness: set the alert threshold at 1.5 to 2 times the expected refresh interval. A table that refreshes hourly should alert if no refresh has occurred in 90 to 120 minutes. This buffer accommodates normal variance in job execution time without suppressing genuine staleness alerts.

For volume: set the alert threshold based on historical variance for the same time window and day of week. Monday morning data volumes are different from Friday afternoon volumes. A threshold of "15% below the 4-week rolling average for this hour on this day of week" is far more meaningful than a static minimum row count.

For completeness: set the null rate threshold based on the column's criticality and historical null rate. A required field with a historical null rate of 0.0% should alert at any non-zero null rate. A field with a known historical null rate of 2% (due to optional user input) should alert only when the null rate exceeds, say, 5%.

For distribution: use statistical process control concepts. Alert when a metric exceeds 3 standard deviations from its rolling mean. This is less likely to produce noise from normal business variation and more likely to flag genuine anomalies.

A Concrete Data Downtime Incident

Consider this scenario: a data engineering team manages a `daily_sales_summary` table that feeds the company's revenue dashboard. The table is refreshed nightly by an Airflow DAG that runs at 11pm. The business expects the dashboard to show current data by 7am.

On a Tuesday night, the DAG starts on schedule. Partway through execution, the Spark job that runs the core aggregation hits a memory limit because that day's transaction volume was 40% higher than normal due to a flash sale. The job fails. Airflow marks the task as failed, retries it twice with the same result, and then marks the DAG run as failed.

The failure alert fires to a shared Slack channel at 11:47pm. Nobody is monitoring that channel at midnight. The alert is seen at 8:15am by a data engineer who is starting their day. By that point, the revenue dashboard has already been displaying yesterday's numbers for 75 minutes during the business day. The sales team has used the stale dashboard to generate a morning report distributed to senior leadership.

What should have happened differently:

The freshness SLA for `daily_sales_summary` should have been set to "data available by 6:45am." A dedicated freshness check running at 6:45am would have fired a high-severity alert to the named owner's pager, not just to a Slack channel. That alert would have triggered a response before business users encountered the stale dashboard.

The volume anomaly from the flash sale should have been detectable the previous afternoon, as transaction volume began exceeding historical norms. An upstream monitoring check on the source transactions table would have allowed the data engineering team to pre-provision additional resources for the nightly job.

The infrastructure-level memory limit error should have been logged as a data downtime event in a central observability system, separate from the infrastructure monitoring system, with context about which tables were affected and which downstream consumers depend on them.

Data downtime is not always dramatic. Often it is four hours of stale data that nobody notices until someone asks why the numbers do not match. The goal of data downtime monitoring is to make those invisible incidents visible before they affect business decisions.

Data Lineage: Mapping the Dependency Graph

Use Cases: What and Where To Observe?

Check Data? Check Tools? Both?

	Data Evaluation	Data Ingestion/Change	Production	Development	Migration
Data Quality Checks	✔✔	✔✔	✔✔	✔	✔✔
Tool Monitoring		✔	✔✔	✔✔	✔

- **Data Checks:**
  - Test and monitor the data itself for quality issues: schema, row count, drift, other tests, …
- **Tool Monitoring:**
  - Observe the tools (or servers) acting on data: logs, metrics, tasks, schedules, etc …

What Data Lineage Is

Data lineage is a complete, navigable map of how data moves through your system. It answers: where did this data come from, what transformations were applied to it, and where does it go from here? A lineage graph connects source systems, raw tables, transformed tables, aggregation layers, and downstream consumers including dashboards, models, and APIs.

Lineage exists at multiple levels of granularity. Column-level lineage tracks individual columns through transformations. Table-level lineage tracks which tables feed which other tables. Job-level lineage tracks which pipeline runs produced which outputs. Most practical observability implementations start with table-level lineage and add column-level lineage for the highest-criticality data flows.

Why Lineage Matters for Root Cause Analysis

Without lineage, debugging a data quality problem requires manual reconstruction of the dependency graph. An analyst sees a wrong number in a dashboard and asks: which table does this metric come from? Which tables feed that table? Which pipeline writes to those tables? Which source system feeds those pipelines? Each of those questions requires a separate investigation.

With lineage, the same analyst looks up the dashboard metric in the lineage system and immediately sees the full dependency chain. The investigation that took four hours now takes 20 minutes.

Lineage is also essential for impact assessment when a known data quality problem is discovered. If a bug is found in a transformation that writes to one table, the question is: what else does this table feed? Without lineage, the answer requires searching through code, documentation, and tribal knowledge. With lineage, the answer is one lookup. The data quality engineer can identify every affected downstream table, dashboard, and model, notify their owners, and make an informed decision about whether to halt the pipeline while the fix is deployed.

A Concrete Lineage Example: The Schema Change That Broke Three Dashboards

Consider this scenario: a data engineering team maintains a `user_events` table in their Silver layer. This table feeds three downstream Gold-layer tables: `daily_active_users`, `feature_adoption_summary`, and `funnel_conversion_metrics`. Each of those tables feeds at least one dashboard consumed by a different team.

In mid-November, a product engineer adds a new event type to the application. To accommodate it, they add an `event_subtype` column to the `user_events` source table. The ETL job that loads `user_events` into the Silver layer does not include `event_subtype` in its schema definition. The column is silently dropped during ingestion.

The `feature_adoption_summary` table uses event subtypes to categorize feature usage. Without `event_subtype`, all feature usage is categorized as "unknown." The feature adoption dashboard shows a sudden collapse in categorized feature usage that is actually an artifact of the missing column.

The `funnel_conversion_metrics` table uses event subtypes to identify specific conversion events. Without the subtype data, several funnel stages show zero conversions. The growth team notices the anomaly two days later and opens an incident.

With a complete lineage graph, the incident response would look like this: the on-call data engineer searches the lineage system for `user_events`. The graph shows that `user_events` feeds `daily_active_users`, `feature_adoption_summary`, and `funnel_conversion_metrics`. It also shows that `feature_adoption_summary` feeds the product analytics dashboard and that `funnel_conversion_metrics` feeds the growth team dashboard and a marketing spend optimization model.

The engineer immediately notifies the owners of all five affected downstream assets. They identify the missing `event_subtype` column as the root cause. They add the column to the Silver-layer schema, reprocess the affected partitions, and confirm that all five downstream assets have been refreshed with correct data.

Without lineage, the same incident response would require manually tracing the dependency chain across code repositories, YAML configuration files, and team wikis. The two-day investigation becomes a half-day investigation. More importantly, the marketing spend optimization model, which nobody thought to check, is caught before it uses two days of bad feature adoption data to make spend allocation decisions.

Building Lineage in Practice

Lineage does not require a dedicated tool, though dedicated tools make it easier to maintain. The minimum viable lineage implementation is a documented dependency map: a structured record of which tables feed which tables, maintained as code and reviewed when pipelines change.

The practical approach for a team starting from scratch:

Start with column-level lineage for Tier 1 tables only. For each Tier 1 table, document its direct upstream sources and its direct downstream consumers. This covers the highest-risk portion of the dependency graph with a manageable amount of work.

Automate lineage capture where possible. Most SQL-based transformation tools can emit lineage metadata as a side effect of parsing queries. dbt, for example, builds a full dependency graph as part of its compilation step. Capture and store this graph as an artifact of each pipeline run.

Surface lineage in the context of incidents. When a data quality alert fires, the alert notification should include a link to the lineage graph for the affected table. Engineers should not have to hunt for lineage information during an incident; it should be one click away from the alert.

Review lineage for completeness during on-call rotations. The most common lineage gaps are new tables that were added without being registered in the lineage system and dashboards or downstream consumers that were built by analysts without notifying the data engineering team. A monthly lineage audit, comparing known tables against the lineage registry, catches most of these gaps.

Connecting the Five Use Cases

The five use cases of data observability are most powerful when they are connected. Lineage tells you what is downstream. Downtime monitoring tells you when something is wrong. Ingestion monitoring tells you where to look upstream. Validation tells you what the data problem is. Development testing tells you whether a proposed fix is safe to deploy.

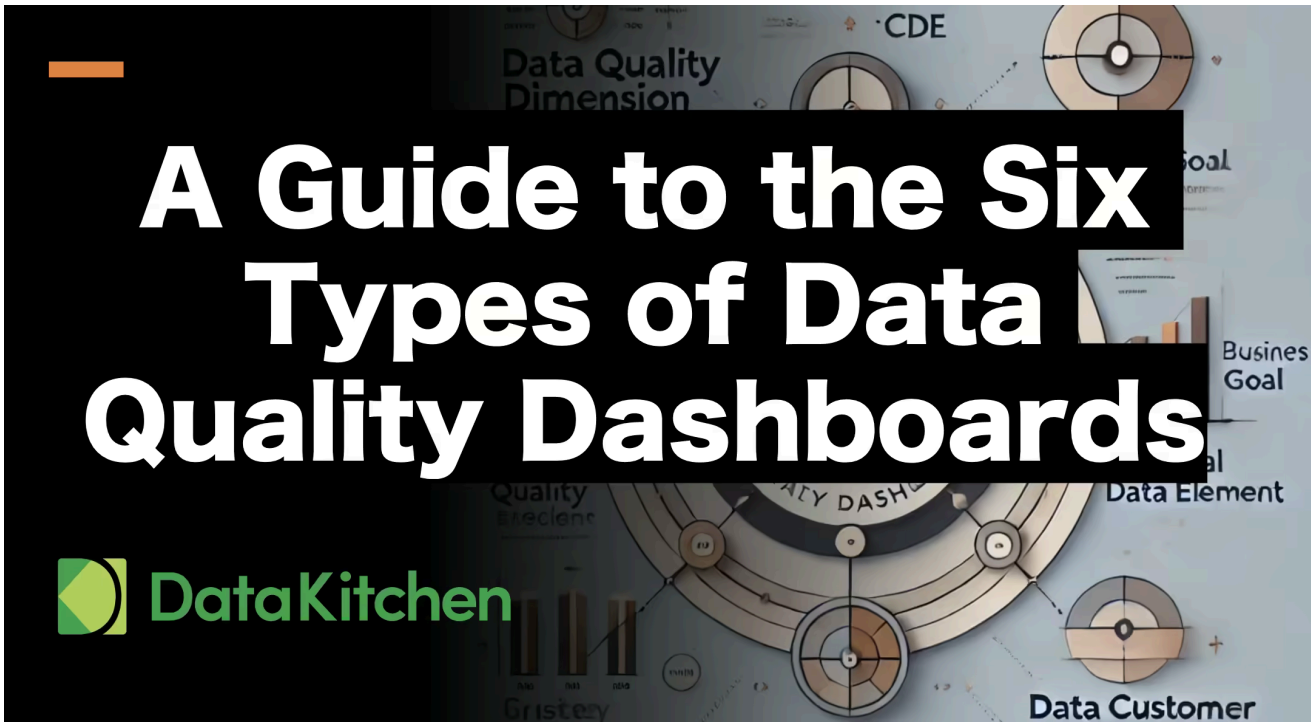
A data team that has instrumented all five use cases can respond to most data quality incidents in minutes rather than hours. The investigation follows a consistent pattern: the alert fires, the lineage graph narrows the blast radius, the ingestion and downtime metrics identify the stage where the failure occurred, the validation results describe the specific defect, and the regression test suite validates the fix before it ships.

That is the operational value of data observability. Not a dashboard. Not a vendor platform. A systematic capability to understand what is happening inside your data systems, fast enough to fix it before it matters.

MONITORING AND OBSERVABILITY

The Six Types of Data Quality Dashboards

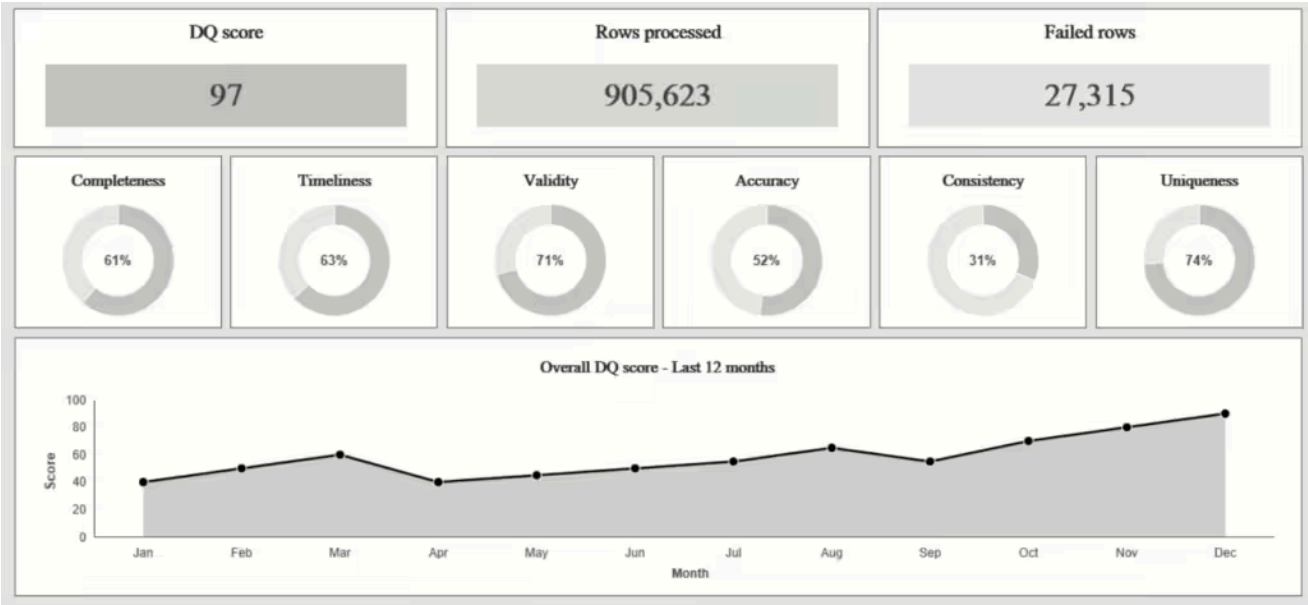
Read the original article at: https://datakitchen.io/blog/the-six-types-of-data-quality-dashboards/



Poor-quality data can derail operations, misguide strategies, and erode the trust of both customers and stakeholders. Data quality dashboards have emerged as indispensable tools, offering a clear window into the health of their data and enabling targeted, actionable improvements. However, not all data quality dashboards are created equal. Their design and focus vary significantly depending on an organization's unique goals, challenges, and data landscape. This blog delves into the six distinct types of data quality dashboards, examining how each fulfills a specific role in ensuring data excellence. By understanding and selecting the right dashboard, organizations can maximize their influence, streamline efforts, and drive meaningful, sustained improvements in data quality. Read more about how your data quality dashboard can be organized by data quality dimension, Critical Data Element (CDE), business goal, data source, and data customer, or it can be based on ticketed workflows. (updated December 2025)

The first type of data quality dashboard emphasizes data quality dimensions. Data Quality Dimension-Focused Dashboards are designed to evaluate data through fundamental quality dimensions, such as completeness, accuracy, timeliness, consistency, and uniqueness. These dimensions provide a best practice grouping for assessing data quality. By scoring each dimension individually, these dashboards offer a grouped view of data quality issues, enabling targeted interventions to address gaps.

For example, metrics such as the percentage of missing values help assess completeness, while deviations from authoritative sources gauge accuracy. Timeliness can be assessed by tracking the alignment of data updates with business timelines. These metrics are typically visualized through tools such as heatmaps, pie charts, or bar graphs, making it easy for stakeholders to understand compliance levels across different dimensions. A retail company, for instance, might use such a dashboard to monitor the completeness of customer profiles, ensuring marketing campaigns have enough data, such as email addresses and phone numbers, for effective targeting.



Benefits and Challenges of The Data Quality Dimension Approach

Data quality dimensions are often heralded as foundational principles for evaluating and improving data quality. While these dimensions provide a ‘scientific’ way to assess data, they can be overly general when applied to real-world data quality problems. While conceptually appealing, this generality can limit its practical utility in driving meaningful improvements in data quality.

The issue lies in the abstraction of these dimensions. For example, a metric like “completeness” is essential to data quality, but defining what “complete” means in a specific context can vary widely. Is completeness about filling every field in a record, or is it about having the fields critical to a particular business process? Similarly, “accuracy” often assumes the existence of an authoritative source for validation, which may not always be available or practical to implement. Without contextual specificity, these dimensions risk becoming check-the-box exercises rather than actionable frameworks that help organizations identify and address the root causes of data quality issues.

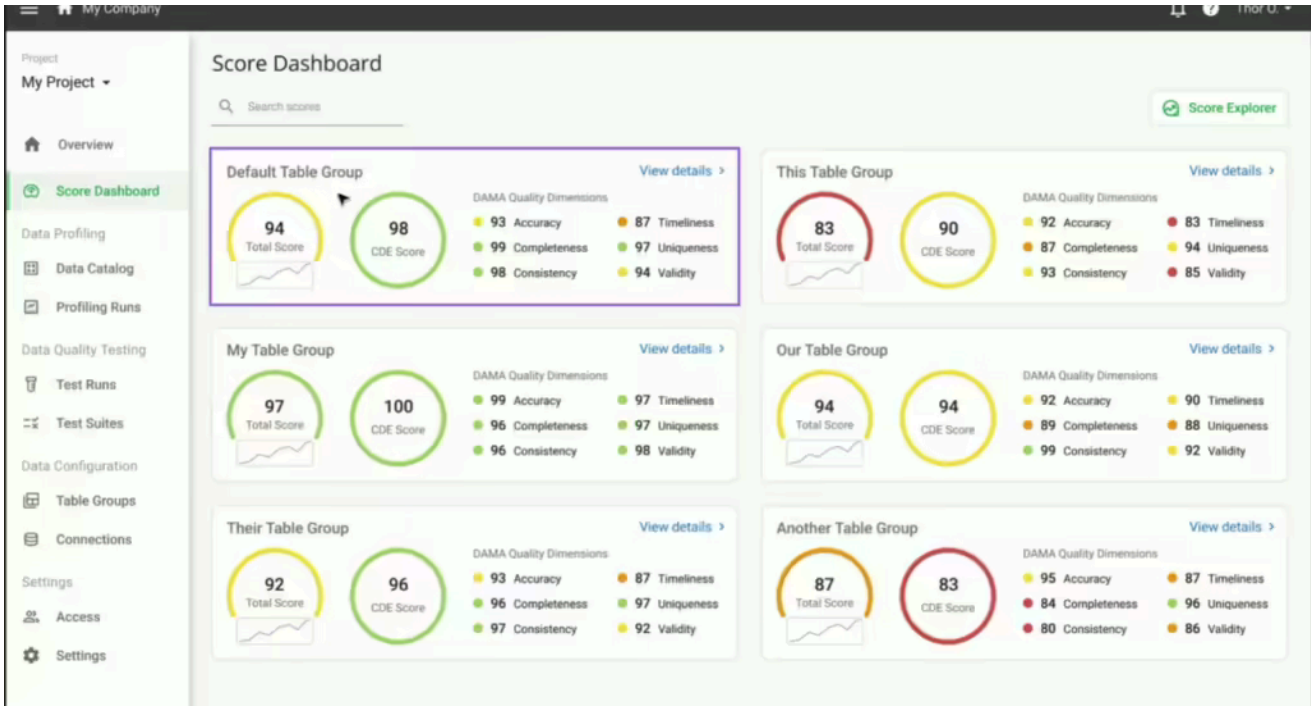
“The DAMA Data Quality Dimension dashboards are crap. They do nothing to motivate people to make improvements to their data.” - Senior Data Quality Person, Financial Services, DataKitchen Market Research 2024

Moreover, relying solely on DAMA dimensions tends to produce dashboards disconnected from an organization's or its stakeholders' unique needs. A dashboard filled with generic scores for completeness or consistency may look polished but needs to resonate with the teams responsible for acting on the insights. For example, a business executive might struggle to connect a "70% consistency score" with the operational goals they are trying to achieve. Similarly, data teams might struggle to determine actionable steps if the metrics do not highlight specific datasets, systems, or processes contributing to poor data quality.

While the DAMA dimensions provide a starting point, the key to impactful dashboards is translating these principles into concrete, tailored insights that directly address the organization's challenges and objectives. By narrowing the scope and aligning metrics with actionable outcomes, organizations can ensure that their dashboards assess data quality and actively contribute to its enhancement.

2. Critical Data Element (CDE) Data Quality Dashboards

Critical Data Element (CDE) dashboards focus on essential data elements crucial for business operations or regulatory compliance. These dashboards prioritize high-impact fields such as financial values, key identifiers like customer IDs, or data required to meet regulatory standards. By narrowing their scope to these critical elements, CDE dashboards enable organizations to allocate resources efficiently and address data quality with the greatest impact. For instance, a bank might use a CDE dashboard to monitor the accuracy of customer credit scores, which are vital for regulatory compliance and effective risk modeling. Another example is an analytics team that wants to focus on the data that goes into the weekly report for the executive team. This targeted approach ensures that limited resources are directed toward improving the data elements that matter most to the organization.



An example of a data quality dashboard with CDEs from DataKitchen's DataOps Data Quality TestGen Open Source Software.

An example of a data quality dashboard with CDEs from DataKitchen's DataOps Data Quality TestGen Open Source Software.

Benefits and Challenges of The CDE Approach

Critical Data Element (CDE) - based data quality dashboards are highly effective in regulated industries such as finance, healthcare, and utilities, where organizations are mandated to monitor and report on the quality of specific data elements. CDEs are often clearly defined by regulatory bodies or industry standards in these industries. For example, a financial institution may need to ensure the accuracy and completeness of loan application data to comply with anti-money laundering (AML) regulations.. The specificity of these mandates makes it easier to identify CDEs and prioritize resources to ensure their quality. CDE-based dashboards in such contexts serve as powerful tools for compliance, risk mitigation, and operational oversight, providing clarity and focus on what truly matters.

"We have had amazing Data Quality Improvement, aligning everyone to improve data based on CDEs. We have corporate goals on CDE data quality. Why? Because we MUST do them for compliance reporting to the government. We focus our Data Quality reporting on CDEs." - Senior Data Quality Leader, Large US Bank Services, DataKitchen Market Research 2024

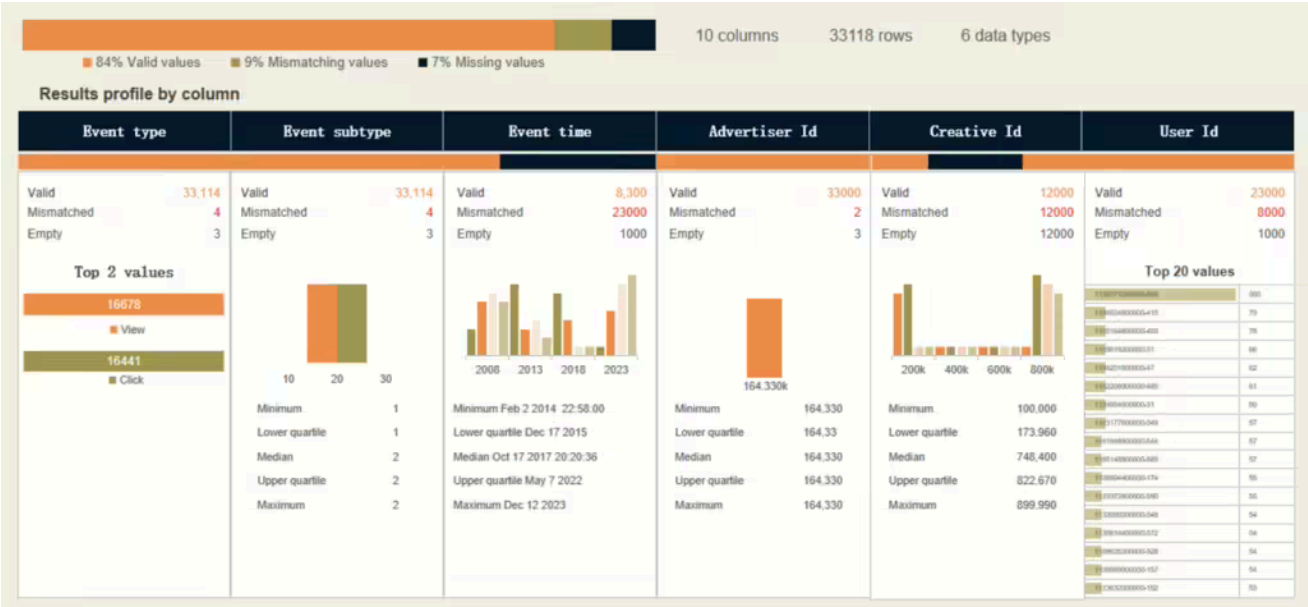
However, identifying and gaining consensus on what constitutes a CDE is a significant challenge in industries without strict regulatory frameworks. Unlike regulated industries, where the importance of specific data elements is externally prescribed, non-regulated industries must internally determine which data elements are critical to their success. This process often involves aligning diverse stakeholders, from business leaders to data engineers, who may have varying priorities and definitions of "critical." For instance, a marketing team might prioritize customer segmentation data, while operations teams focus on inventory data. The lack of predefined criteria can lead to prolonged debates, misalignment, and inconsistent interpretations of what constitutes a CDE.

Another challenge in non-regulated industries is the dynamic nature of business priorities. CDEs may shift as market conditions, organizational goals, or technologies evolve. A retail organization, for example, might initially consider transaction data as critical, only to pivot toward prioritizing clickstream data for e-commerce analytics. This fluidity requires an iterative approach to defining and managing CDEs, which can be resource-intensive and complicated to operationalize within a dashboard framework.

Despite these challenges, CDE-based dashboards remain valuable in non-regulated industries if implemented thoughtfully. Organizations can address these hurdles by establishing transparent governance processes for identifying CDEs, leveraging cross-functional collaboration to ensure alignment, and periodically revisiting and revising the CDE list to reflect changing priorities. While it requires effort and coordination, the result is a targeted data quality approach that delivers tangible business value, even in industries where regulations don't dictate the terms.

3. Business Goal-Focused Data Quality Dashboards

Business goal-focused data quality dashboards link data quality metrics directly to specific organizational objectives, such as quarterly revenue goals, yearly cost savings targets, or specific revenue-generating marketing and sales programs. These dashboards highlight how data quality influences broader goals, such as customer retention, revenue growth, or regulatory compliance. For leadership teams, this type of dashboard serves as a critical tool to understand the operational and financial implications of data quality issues and to justify investments in remediation efforts. For example, an e-commerce company might use a business goal-focused dashboard to monitor how incomplete customer delivery information affects order fulfillment rates and timing, a crucial metric for customer satisfaction. By tying data quality improvements directly to business outcomes, these dashboards make the case for prioritizing data quality as a strategic initiative.



Benefits and Challenges of The Business Goal-Focused Data Quality Approach

Business goal-focused data quality dashboards help connect technical data quality efforts to broader organizational objectives. By focusing on outcomes that matter to business stakeholders, these dashboards make data quality improvements more relevant and impactful, fostering cross-functional collaboration between technical teams and business units. This alignment also helps secure executive buy-in and funding for data initiatives by demonstrating the tangible value of addressing data quality issues.

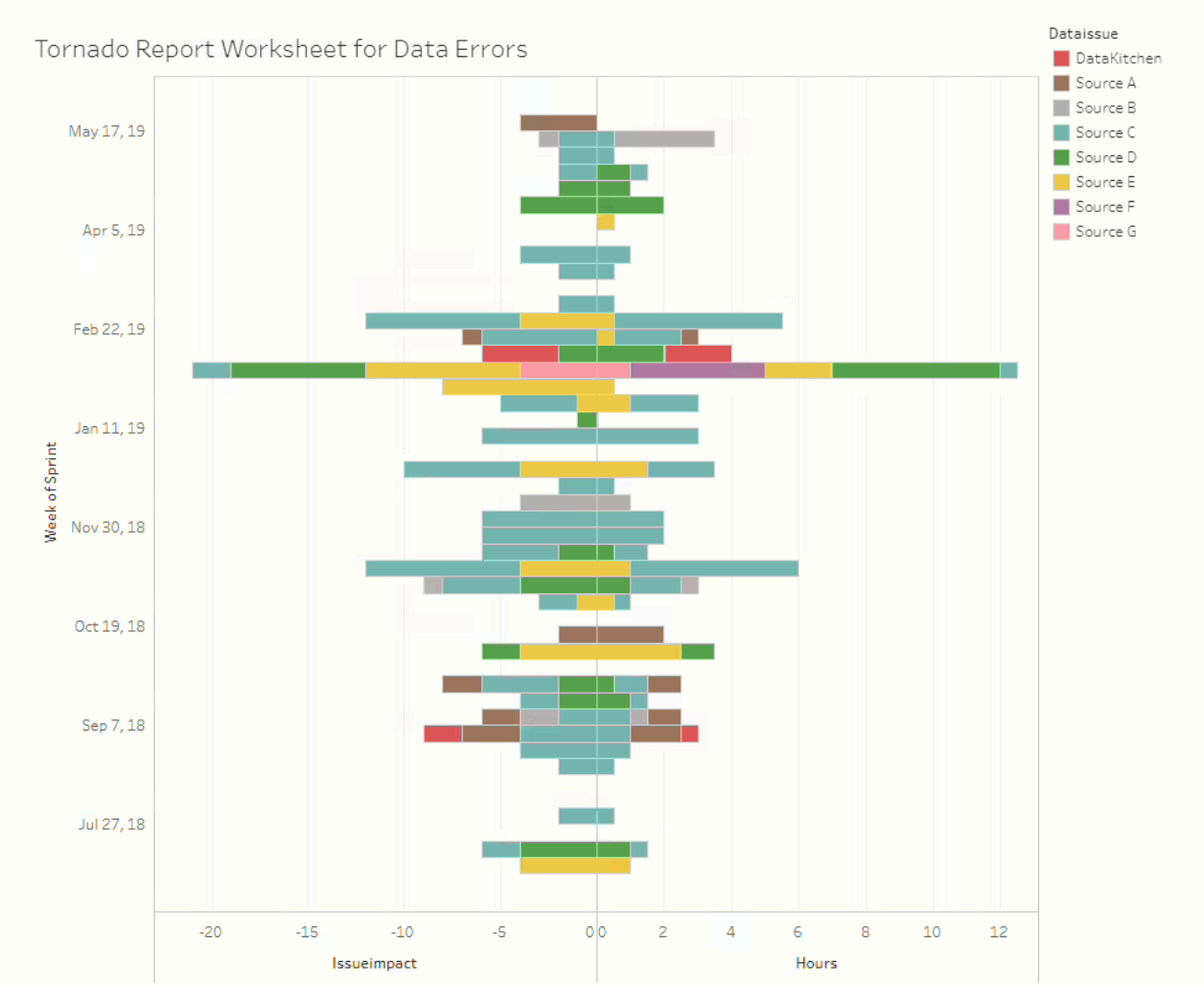
Another significant advantage is that these dashboards provide ROI-driven insights. By highlighting the financial or operational impact of poor-quality data, business goal-focused dashboards allow organizations to prioritize remediation efforts that deliver the most significant value. For example, identifying how incomplete customer delivery information affects fulfillment rates can help an e-commerce company streamline operations, reduce costs, and enhance customer satisfaction.

“How do you get the business to take action? By definition, they are inputting data that meets their data quality needs. With all their other problems, why should they care about data quality when providing data to other teams? The only way to motivate them is to pick data elements directly affecting their quarterly business goals. They care about that.” - Data Quality Consultant, DataKitchen Market Research 2024

Despite their advantages, business goal-focused dashboards come with notable challenges. One of the most significant challenges is defining and maintaining alignment between data quality metrics and business goals. Business objectives often evolve due to market dynamics, organizational restructuring, or technological changes, requiring dashboards to be continuously updated to stay relevant. This iterative process demands time, effort, and team collaboration, which can strain resources, especially in organizations with limited data governance capabilities.

4. Data Source-Focused Data Quality Dashboards

A data source-focused dashboard assesses data quality by analyzing its origin, enabling organizations to identify sources that consistently provide low-quality data and take corrective actions. This approach allows enterprises to hold data suppliers accountable or optimize their ingestion processes to ensure higher data integrity. By tracking source-level metrics such as error rates, duplication, and timeliness, these dashboards pinpoint the suppliers or systems responsible for data quality failures, fostering accountability and improvement. This is particularly valuable for organizations managing data from multiple external providers or systems, such as suppliers or third-party APIs. For example, a logistics company might use a source-focused dashboard to monitor the accuracy and timeliness of shipping data from various courier partners, ensuring reliable tracking information for customers and maintaining service quality.



Benefits and Challenges of Data Source-Focused Data Quality Dashboard Approach

Data source-focused dashboards assign accountability for data quality. Organizations can take targeted action by identifying which suppliers, systems, or processes are contributing to poor-quality data, whether that means renegotiating contracts with external vendors, improving internal workflows, or implementing better validation processes at the point of ingestion. This accountability encourages data providers to prioritize quality.

Another significant benefit is the actionable insights these dashboards provide for optimizing data ingestion processes. Organizations can track each source's error rates and timeliness. This enables them to refine pipelines and ensure data flows into the organization more accurately and reliably. For example, a logistics company might use these dashboards to monitor how courier partners provide shipping data, identify lagging contributors, and take steps to ensure timely updates.

"Our data suppliers don't know that we exist, don't care about quality, and will make arbitrary changes to our feeds at any time. It's up to us to deal with it. We have no control over their data quality" - Data Engineer, Pharmaceuticals, DataKitchen Market Research 2024

Despite their advantages, data source-focused dashboards have notable challenges. One challenge lies in gaining cooperation from external data providers. While organizations can monitor and report on data quality issues, enforcing improvements depends on the willingness and capability of suppliers to address them. They often do not care. How do you motivate them to make a change? These dashboards can become less effective when multiple sources contribute to a single dataset, making it difficult to pinpoint the root cause of data quality issues. For example, if data from various suppliers is aggregated into a central system, determining whether errors originated with a specific supplier or during the integration process requires a root cause analysis.

5. Data Consumer-Focused Data Quality Dashboards

A data consumer-focused dashboard is designed to meet the specific needs of individuals and teams who rely on data to perform their roles, such as data scientists, analysts, or business intelligence professionals. These dashboards prioritize the quality and reliability of datasets and features critical to analytics models, reports, or decision-making tools, ensuring they are fit for purpose. They are often customized to address the unique requirements of different user personas, whether for predictive model inputs or operational reporting. For instance, a healthcare company might use such a dashboard to monitor the accuracy and consistency of patient demographic data used in predictive models for patient readmission. By catering directly to the needs of data consumers, these dashboards help their data customer use their influence to make changes to improve data quality.

Benefits and Challenges of a Data Consumer-Focused Data Quality Dashboard Approach

One of the primary benefits of this approach is its emphasis on ensuring that the data used by analysts, data scientists, and other consumers is reliable, accurate, and error-free. Tracking the quality of specific datasets or features critical to their projects can give the project owner leverage to improve data quality. Another key advantage is the customization these dashboards provide, tailoring metrics and views to the needs of individual roles or teams. Different data consumers have unique requirements: a business intelligence analyst may need dashboards that track the completeness of reporting data. At the same time, a data scientist might need to focus on model input features such as timeliness or uniqueness. By aligning data quality metrics with the specific needs of end-users, these dashboards foster efficiency and confidence in analytics processes.

“Our data scientists are forever complaining about the quality of data. They have high-value predictions in production. They are motivated to drive improvements to data. We need to focus our data quality reporting on the data inputs to their models, specifically. - Data Quality Consultant, DataKitchen Market Research 2024

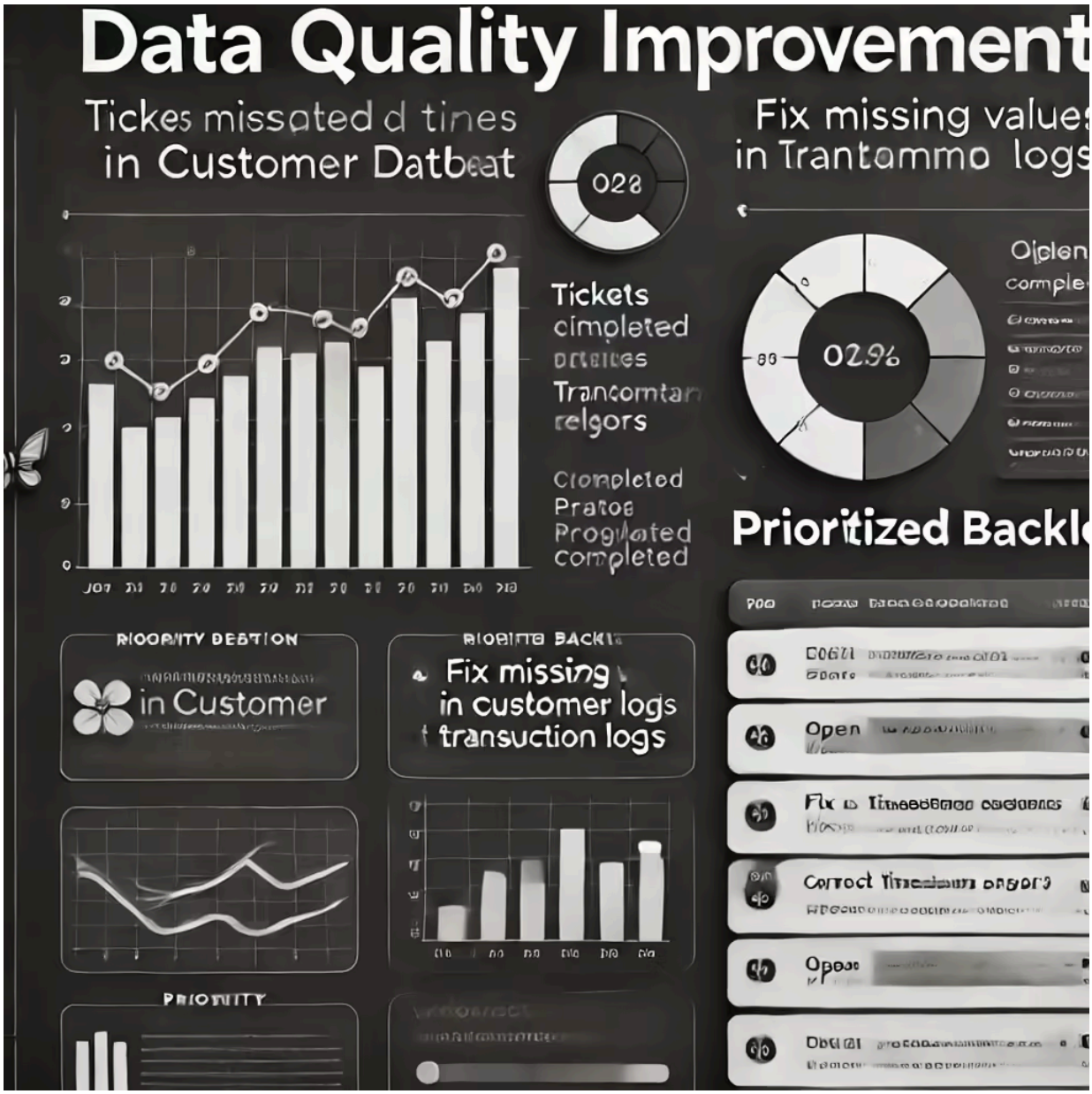
While highly effective, data consumer-focused dashboards present several challenges. One significant hurdle is defining and managing the diverse requirements of different user groups within an organization. Each team or individual may have distinct data needs, leading to a complex and time-intensive process of creating and maintaining customized dashboards. Balancing these needs while ensuring the scalability and manageability of the dashboard system is critical.

6. The Data Quality Ticket Focused Dashboard

Using tickets to drive data quality improvements is another approach to managing and resolving data issues. Tickets are change requests that break down data quality challenges into specific, detailed tasks targeting individual data elements or processes. Each ticket represents a well-defined issue, such as fixing missing values, correcting data inaccuracies, or standardizing inconsistent formats, making it easier for data engineering teams to prioritize, assign, and resolve problems systematically. This approach ensures that data quality initiatives are not abstract or overwhelming but are actionable, trackable, and aligned with organizational goals.

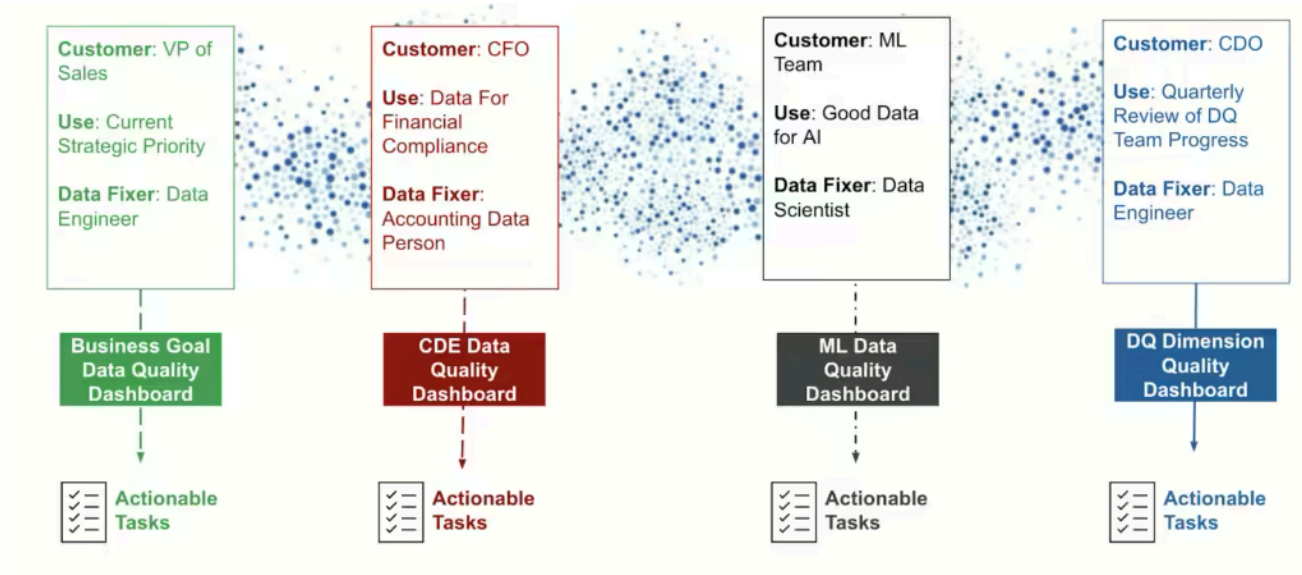
“We don’t use a dashboard per se; we use workflow. It is about the number of tickets created and fixed during a particular period. We have specific goals around tickets, and that works for us.” - Data Quality Leader, Regional Bank, DataKitchen Market Research 2024

In this context, the data quality dashboard shifts its focus from traditional quality metrics to operational progress, visualizing the state of the ticketing process. The dashboard provides two critical views: the count of tickets completed over time, which highlights progress and the team’s capacity to address issues, and a prioritized backlog of pending tickets, ensuring that high-impact problems are addressed first. By tracking ticket completion trends, organizations can measure the efficiency of their data quality efforts and identify bottlenecks in the remediation process. This ticket-driven model fosters accountability and aligns with agile principles, enabling continuous iteration and improvement in data quality over time.



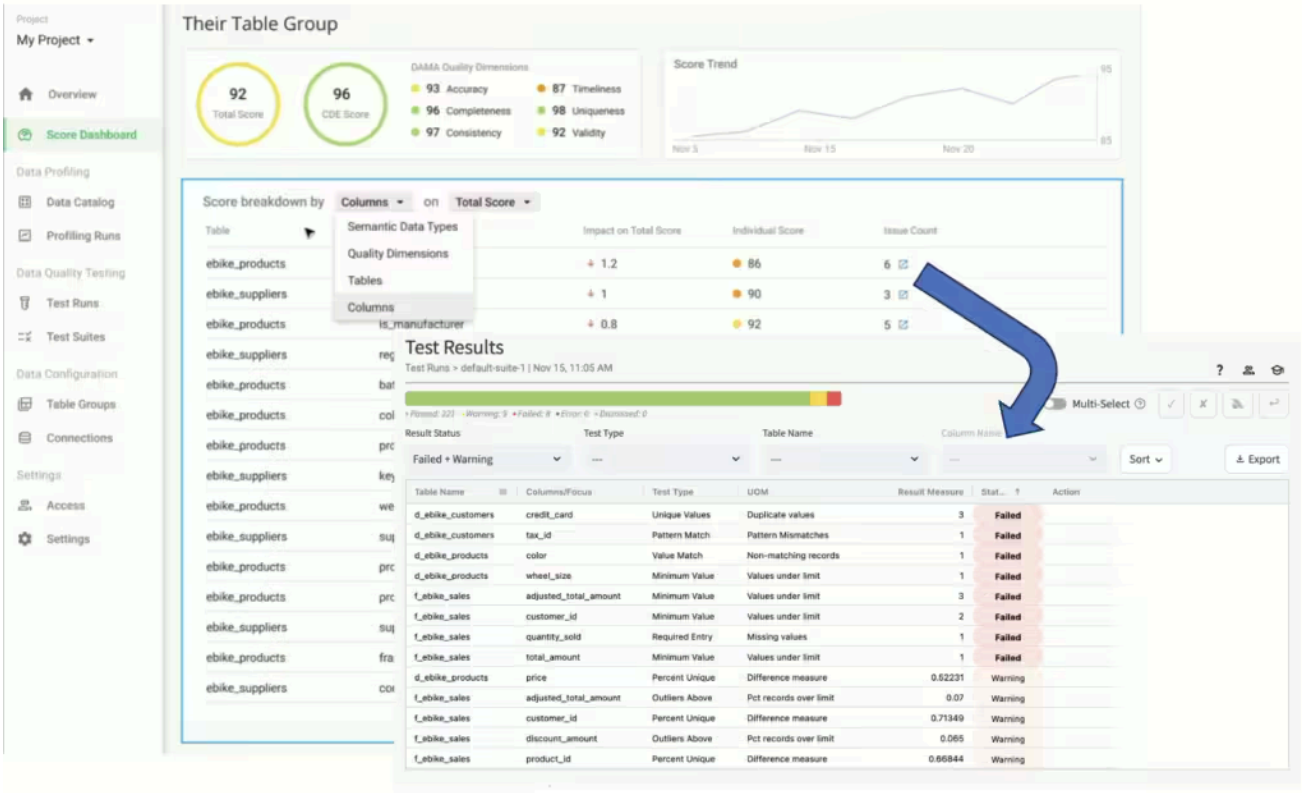
Why You Need a Multi-Dashboard Approach: Maximizing Your Influence

Adopting a multi-dashboard approach to data quality is essential for organizations aiming to understand their data landscape comprehensively. Each type of dashboard offers a unique lens through which to view and address data quality issues, and combining these perspectives allows for a more nuanced and effective strategy. By leveraging multiple dashboards, organizations can ensure that no aspect of data quality is overlooked, that priorities are aligned with business goals, and that all stakeholders have access to the data they need to succeed.



Dimension-focused dashboards are foundational in ensuring that every critical aspect of data quality, completeness, accuracy, timeliness, consistency, and uniqueness, is monitored and addressed. These dashboards provide a holistic overview of data quality, enabling organizations to identify and remediate deficiencies systematically. However, this broad approach can be complemented by Critical Data Element (CDE) dashboards, which prioritize the data elements that are most crucial for operational efficiency or regulatory compliance. These dashboards balance addressing general quality metrics and focusing on high-priority data elements.

Business goal-focused dashboards add another layer of alignment by tying data quality efforts directly to organizational objectives. These dashboards ensure that data quality initiatives are not just technical exercises but are integral to achieving key business outcomes such as customer retention, revenue growth, or compliance. At the same time, data source-focused dashboards bring accountability into the equation by identifying where data quality issues originate, whether from internal systems or external suppliers, and streamlining processes to mitigate risks at their source.



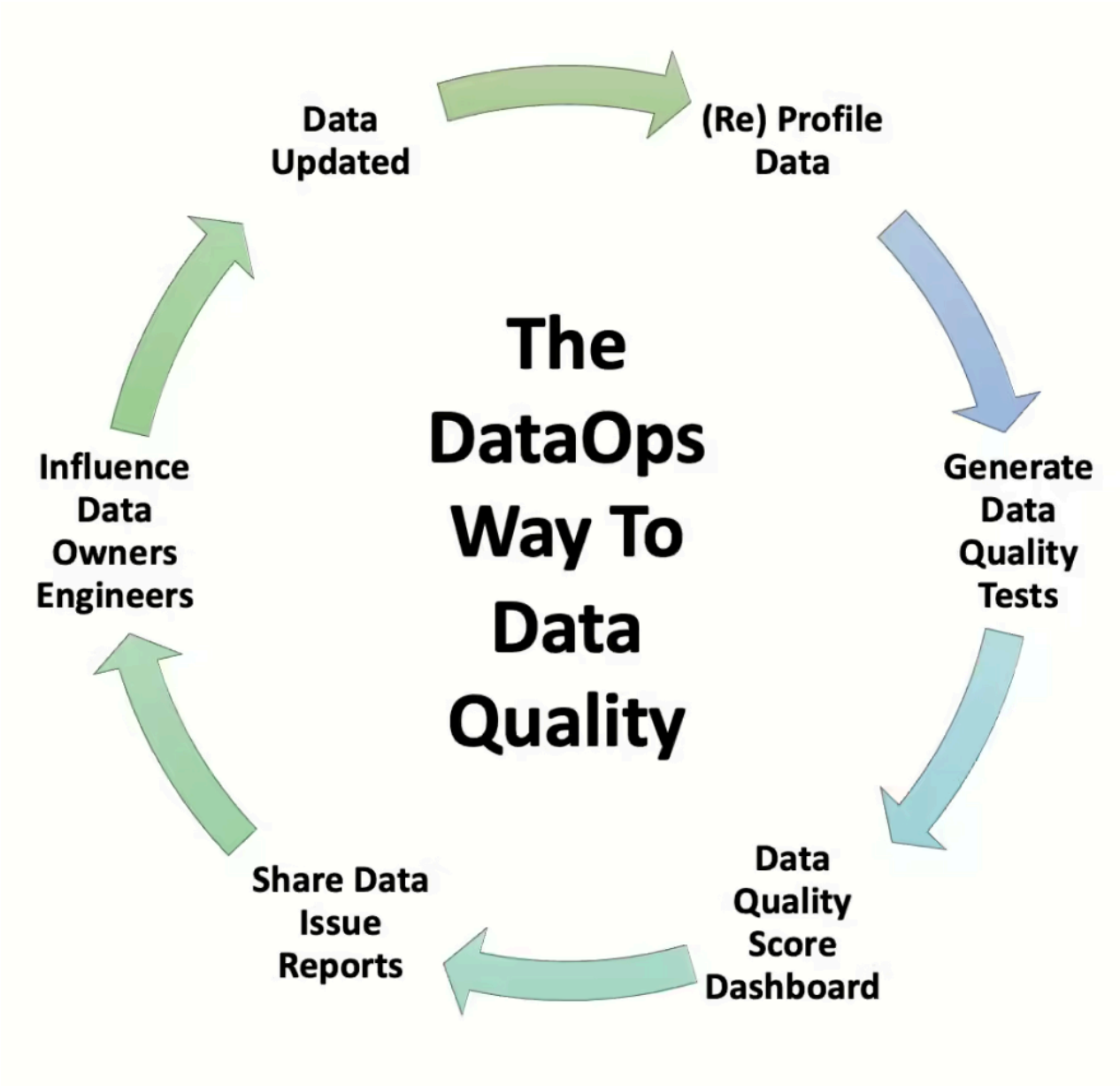
An example of a data quality dashboard drill down to specific, actionable issues from DataKitchen's DataOps Data Quality TestGen Open Source Software.

An example of a data quality dashboard drill down to specific, actionable issues from DataKitchen's DataOps Data Quality TestGen Open Source Software.

Finally, data consumer-focused dashboards ensure that end-users, data scientists, analysts, and business teams, can access reliable and fit-for-purpose data for their needs. Organizations can enhance decision-making, reduce inefficiencies, and build trust in their data ecosystem by catering to these distinct roles. These six dashboards create a comprehensive framework that empowers organizations to address data quality while targeting specific pain points.

Best Practices for Building Effective Data Quality Dashboards

The first step is to define clear objectives for each dashboard. Understanding its specific purpose, whether to monitor compliance, support operational goals, or provide reliable insights to data consumers, helps tailor metrics to align with organizational priorities. A well-defined objective ensures that dashboards remain focused and deliver value to stakeholders without overwhelming them with irrelevant or extraneous data.



Automation is another critical element in maximizing the effectiveness of data quality dashboards. Fast updates ensure that dashboards remain responsive to the latest data conditions, reducing the need for manual intervention and improving decision-making speed. Automation also enhances scalability, making maintaining multiple dashboards across an organization easier without excessive overhead. By leveraging modern tools and frameworks like DataKitchen's DataOps Data Quality TestGen, teams can ensure their dashboards remain dynamic and relevant, even as data sources and business requirements change.

Accessibility is critical to ensuring that dashboards are effective for technical and non-technical stakeholders. Dashboards should be intuitive, with user-friendly visualizations and interfaces catering to end-users' diverse needs. Clear, actionable metrics and insights allow executives, data scientists, and operations teams to understand and act on data quality issues without requiring extensive technical

expertise. Dashboards must be paired with processes to act on their insights to influence data quality outcomes. Dashboards alone are diagnostic tools, they highlight problems but cannot solve them. Organizations should integrate their dashboards with fast, actionable, motivating. ‘Here fix this’ ticket creation and workflows that enable rapid remediation of issues. This action-oriented approach ensures that data quality issues identified by dashboards are fixed.

DataOps Data Quality TestGen

DataKitchen

TestGen Test Issue Report

Test	**Minimum Value:** All values at or above set minimum Tests that the minimum value present in the column is no lower than the minimum value in baseline data		Failed
Measured Value	3		
Threshold Value	0	Expected count of values under limit	
Test Run Date	Nov 15, 11:05 AM EST	**Table Group**	default
Database/Schema	demo	**Test Suite**	default-suite-1
Table	f_ebike_sales	**Data Quality Dimension**	Validity
Column	adjusted_total_amount	**Disposition**	No Decision

View on TestGen >

Usage Notes

This test is appropriate for a cumulative dataset only, assuming all prior values are still present. It is also appropriate for any measure that has an absolute, definable minimum value, or a heuristic that makes senes for valid data.

Result History

Test Date	Threshold Value	Measure Value	Status
Nov 15, 11:05 AM EST	0.0	3.0	Failed
Nov 15, 11:03 AM EST	0.0	3.0	Failed
Nov 15, 11:01 AM EST	0.0	3.0	Failed
Nov 15, 10:59 AM EST	0.0	0.0	Passed

Sample Data

Adjusted Total Amount	Difference From Baseline
102.1	-66.28
161.51	-6.87
164.95	-3.43

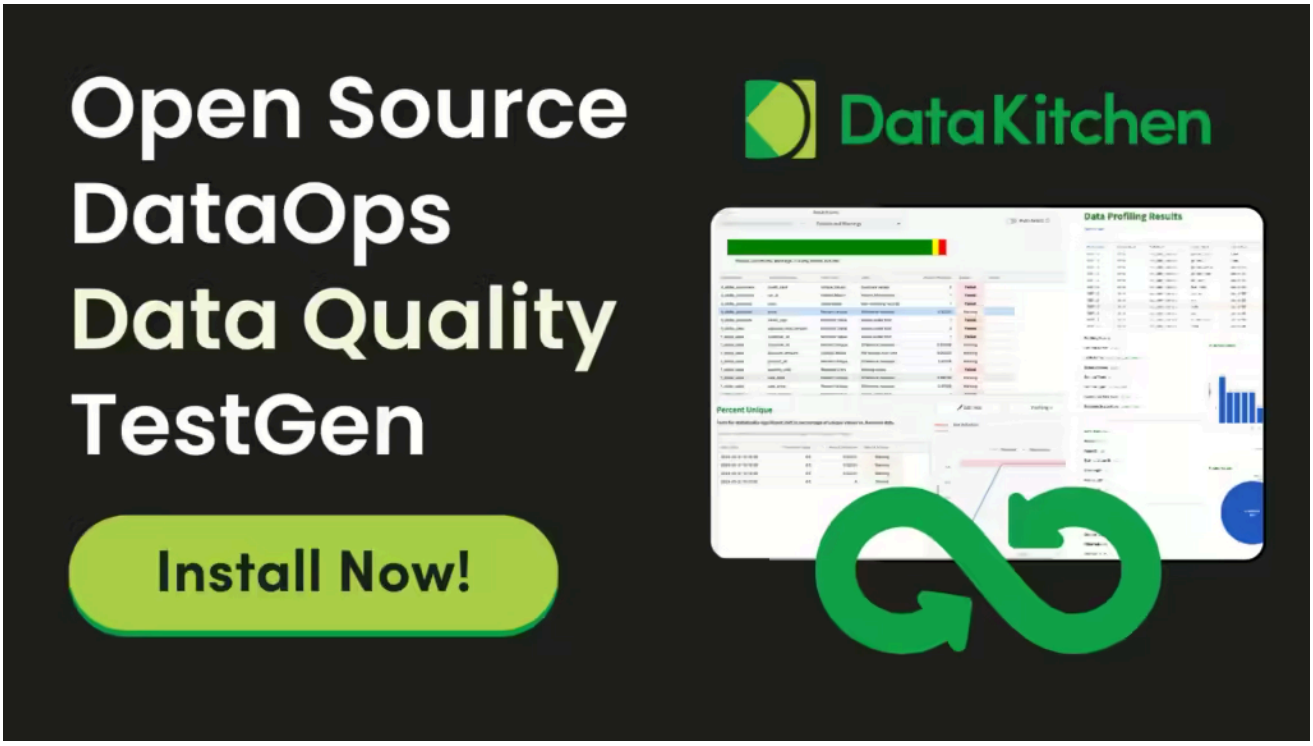
SQL Query

```
SELECT DISTINCT "adjusted_total_amount", (ABS("adjusted_total_amount") - ABS(168.38)) AS
difference_from_baseline FROM demo.f_ebike_sales WHERE "adjusted_total_amount" < 168.38 LIMIT 500;
```

Iteration and continuous improvement are essential to the long-term success of data quality dashboards. As organizational needs evolve, so must the dashboards and the metrics they track. A DataOps approach to data quality is needed, emphasizing collaboration, automation, and iterative development. DataOps enables teams to innovate on dashboards and metrics, ensuring they remain aligned with changing priorities and drive sustained improvements. By iterating and innovating, organizations can maximize their influence on stakeholders and foster a culture of continuous data quality enhancement.

Conclusion

Data quality dashboards are indispensable tools for managing and improving data quality. By leveraging the six dashboard types, dimension-focused, CDE, business goal-focused, data source-focused, data consumer-focused, and task list, organizations can take a strategic, targeted approach to address their unique data quality challenges. The goal is to achieve agile data quality at scale: automate data quality tests, make informed trade-offs, iterate quickly to maximize your influence, and deliver actionable improvements to those who can make change, fast and focused. With the right dashboards, your organization can move confidently toward its data-driven goals and ensure that decisions are built on a solid foundation of high-quality data. And you, as a data quality leader, can gain influence and success in your organization!



Data Quality Alerting: Avoiding Alert Fatigue

MONITORING AND OBSERVABILITY

Data Quality Alerting: Avoiding Alert Fatigue

The DataOps Way to Data Quality and Data Observability

DataKitchen

Alerting is where most data quality programs go to die. The logic seems sound: write tests, fire alerts when they fail, fix problems quickly. In practice, most teams end up with a system that cries wolf so often that nobody listens anymore. This chapter explains why that happens and how to build an alerting system that actually works.

What Alert Fatigue Looks Like

Consider a scenario that plays out on data teams everywhere.

A mid-sized e-commerce company's data team spent three months building a monitoring system. They wrote 200 tests covering their most important tables and connected everything to a Slack channel called #data-alerts. In the first week, the channel received about 40 messages per day. Most of them were false positives: freshness checks firing during scheduled maintenance windows, row count checks triggering on tables that legitimately grow slowly on weekends, column checks flagging null values that were expected based on the data model.

By week three, the team had developed a habit. When an alert fired, someone would glance at it, confirm it was probably nothing, and mark it as read. Nobody was writing up incident reports. Nobody was updating thresholds. The backlog of unactioned alerts kept growing.

In week six, a genuinely bad incident occurred: the revenue attribution table stopped loading for 18 hours due to an upstream API change. The alert fired. Nobody noticed because the alert was buried in a channel that everyone had mentally muted. The head of marketing spent two days running a campaign based on stale attribution data. The cost was significant.

When the data team reviewed what happened, the honest answer was: we disabled the alerting system in practice, even though it was technically still running. The alerts were present but invisible, because they had lost all signal value.

This is alert fatigue. It is not a technical failure. It is a trust failure. The system became useless because it could not distinguish between things that mattered and things that did not.

Why Data Alerts Fail More Often Than Software Alerts

Software engineers have been managing production alerts for decades. They have learned hard lessons about signal-to-noise ratios. Why do data teams keep making the same mistakes?

Data quality is inherently noisier than software behavior. A web service is either up or down, responding within SLA or not. The binary nature of software behavior makes it relatively straightforward to set meaningful alert thresholds. Data is messier. A table with 100,000 rows today might have 95,000 tomorrow for entirely legitimate reasons: fewer transactions on Sunday, a batch job that runs weekly rather than daily, a new business rule that filtered out a subset of rows.

Thresholds are harder to calibrate. In software monitoring, you know what "normal" looks like because you designed the system. In data monitoring, normal is often statistical. It depends on the day of week, the time of year, the upstream data source behavior, and a dozen other factors that interact in complex ways. Setting a threshold that is tight enough to catch real problems but loose enough to avoid constant false positives requires observation over time, not a one-time configuration.

Many failures are known and expected. End-of-month data loads are late. Holiday traffic spikes change row counts. Fiscal year rollovers cause reference tables to expand unexpectedly. None of these are data quality failures, but a naive monitoring system will alert on all of them. Without a mechanism to suppress known-expected variations, the alert channel fills with noise.

Setting Thresholds That Are Right-Sized

The most important decision in data alerting is not which checks to write. It is how to configure the thresholds on those checks.

Absolute thresholds vs. percentage-based thresholds.

Absolute thresholds say: this value must be exactly X, or between X and Y. They are appropriate when you have a genuine hard constraint. A table containing US state codes should have exactly 50 distinct values (or 51 if you include DC). A column containing email addresses should have zero nulls if your data contract requires it.

Percentage-based thresholds say: this value must be within N% of a reference point. They are more appropriate for counts, sums, and rates that fluctuate naturally. Row count checks should almost always be percentage-based rather than absolute. "Row count must be between 90,000 and 110,000" will break constantly. "Row count must be within 20% of the 7-day rolling average" is more resilient.

Anomaly detection approaches.

For tables with more complex patterns, simple percentage thresholds still generate noise. The next level is statistical anomaly detection: comparing today's value to a distribution of historical values and alerting only when the observation falls outside a confidence interval.

The simplest version of this is z-score detection: calculate the mean and standard deviation of the last 30 days of a metric, and alert only when today's value is more than 3 standard deviations from the mean. This approach handles gradual trend increases (the mean shifts with the trend) and reduces false positives significantly compared to fixed thresholds.

Commercial data observability tools (Monte Carlo, Bigeye, Anomalo, and similar) implement more sophisticated versions of this. If you are doing it yourself, a 3-sigma check on a 30-day rolling window handles most cases adequately.

Seasonality adjustments.

Many metrics are strongly seasonal. Monday row counts are different from Sunday row counts. November revenue is different from March revenue. An alert system that ignores seasonality will generate noise every weekend and every holiday period.

The fix is to compare against the same time period in prior cycles rather than against an undifferentiated rolling average. Instead of "today's row count vs. the last 30-day average," use "today's row count vs. the same day of week over the last four weeks." This simple adjustment eliminates a large portion of seasonal false positives.

Alert Tiering

Not every data quality failure is equally urgent. A system where every failure generates the same response will produce the same outcome as a system with no tiers at all: total noise, total indifference.

Define three tiers.

P1: Data is completely broken, business process stops.

A P1 alert means that a critical business function is either completely unavailable or known to be delivering wrong results. Examples: the revenue table has not loaded for more than 24 hours, a fraud detection model is reading from a table that was accidentally truncated, or the customer order feed has stopped entirely.

P1 alerts should be rare, roughly 2 to 5 per year for a well-run team. They warrant immediate response: page the on-call engineer, notify the business owner, begin incident response. A P1 that is not acknowledged within 30 minutes should escalate automatically.

P2: Significant degradation, manual review needed.

A P2 alert means the data is available but has a meaningful quality issue that could affect decisions if not investigated. Examples: row count is 40% below the historical average, a key dimension table has 15% more null values than expected, or referential integrity checks are failing on 2% of records.

P2 alerts should trigger investigation, not panic. The on-call engineer reviews the alert, determines the scope and likely cause, and communicates to stakeholders within a few hours. Many P2 alerts can wait until the next business day if they occur overnight.

P3: Informational, log but do not page.

A P3 alert records an anomaly that is worth knowing about but does not require immediate action. Examples: a row count is 8% below average (just below your P2 threshold), a new value appeared in a status code column, or a pipeline ran 30 minutes later than usual.

P3 alerts go to a log or a low-traffic Slack channel. Nobody gets paged. The team reviews P3 logs weekly during a regular data quality review meeting.

Alert Level	Meaning	Response Time	Notification Method
P1	Critical failure, business stopped	30 minutes	Page on-call + notify business owner
P2	Significant degradation	4 hours	Slack direct message to table owner
P3	Informational anomaly	Weekly review	Logged to channel, no direct notification

Alert Routing

After you have defined tiers, define who receives each type of alert for each table. The most common mistake is routing everything to a shared channel where everyone is nominally responsible and therefore nobody is actually responsible.

Build a routing matrix. For each table tier, identify the on-call engineer, the table owner, and the business owner who should be informed of P1 incidents.

Table	Tier	P1 Alert To	P2 Alert To	Business Owner
orders	1	On-call + @maya	@maya	Head of Operations
revenue_summary	1	On-call + @dev	@dev	CFO
marketing_attribution	1	On-call + @priya	@priya	VP Marketing
product_catalog	2	@james	@james	Product team
raw_clickstream	3	Log only	Log only	None

The table owner column is critical. Every table should have one named engineer who is responsible for it. When an alert fires, there should be no ambiguity about whose job it is to respond.

Implement this routing in your alerting infrastructure. Tools like PagerDuty and OpsGenie support escalation policies that can encode these rules. If you are using Slack-based alerting, create separate channels by domain (finance-data-alerts, marketing-data-alerts) and route alerts to the appropriate channel rather than a single catch-all.

Rebuilding an Alerting System After Fatigue Sets In

If your current alerting system has collapsed into noise and indifference, you cannot fix it by adjusting a few thresholds. The team has already learned to ignore it. You need to reset.

Here is a 30-day reset process.

Week 1: Silence everything.

Turn off all non-P1 alerts. Leave only the checks that represent genuine, unambiguous failures (the table has not loaded in 48 hours, a key column is 100% null). Acknowledge that you are starting over.

This will feel uncomfortable. Some stakeholders will ask whether you are monitoring the data at all. The honest answer is: yes, for the things that matter most, and we are fixing the monitoring system so that it is actually useful.

Week 2: Audit existing checks.

Review every check that was running before the silence. Categorize each one:

- Genuinely useful (would catch a real problem, threshold is reasonable): keep
- Right idea, wrong threshold (fires too often or misses real issues): fix
- Testing the wrong thing (checks a process, not data content): rewrite or delete
- Stale (checks a table or column that no longer exists or no longer matters): delete

Most teams find that 30% to 50% of their existing checks fall into the last two categories.

Week 3: Rebuild with tiering.

Reintroduce checks in priority order, starting with P1 checks for Tier 1 tables only. Configure percentage-based thresholds using actual historical data. Add seasonality adjustments for metrics you know are day-of-week sensitive. Set explicit suppression windows for known load periods.

Do not re-enable more than 20 to 30 checks in week three. Rebuild slowly.

Week 4: Validate and communicate.

Run the rebuilt system for a full week. Track every alert that fires. For each one: was it a real problem? Did the right person respond? Was the threshold appropriate?

Adjust any threshold that generated a false positive during the week. Then communicate to the team and stakeholders: here is what the new alerting system looks like, here is how it is tiered, here is how to respond to each level, and here is why we believe it will be more reliable than what we had before.

A well-calibrated alerting system should generate 2 to 5 P1 alerts per quarter, 10 to 20 P2 alerts per quarter, and a P3 log that the team reviews weekly. If you are significantly above those numbers after the reset, the thresholds are still wrong or you have more underlying data quality problems than the alerting system can fix on its own.

The Ongoing Work

Alert management is not a one-time configuration. Thresholds drift. Tables change. Business patterns shift. A check that was well-calibrated six months ago may be generating noise today because the underlying data changed.

Build a quarterly threshold review into your team's regular cadence. Spend 30 minutes reviewing which checks fired most frequently, whether those firings were actionable, and whether any checks need to be updated or retired. This small investment keeps the alerting system trustworthy over time and prevents the gradual collapse back into fatigue.

The goal is a system where when an alert fires, the team treats it as real. That outcome is not achieved by installing the right tool. It is achieved by consistent, disciplined attention to signal quality over many months.

MONITORING AND OBSERVABILITY

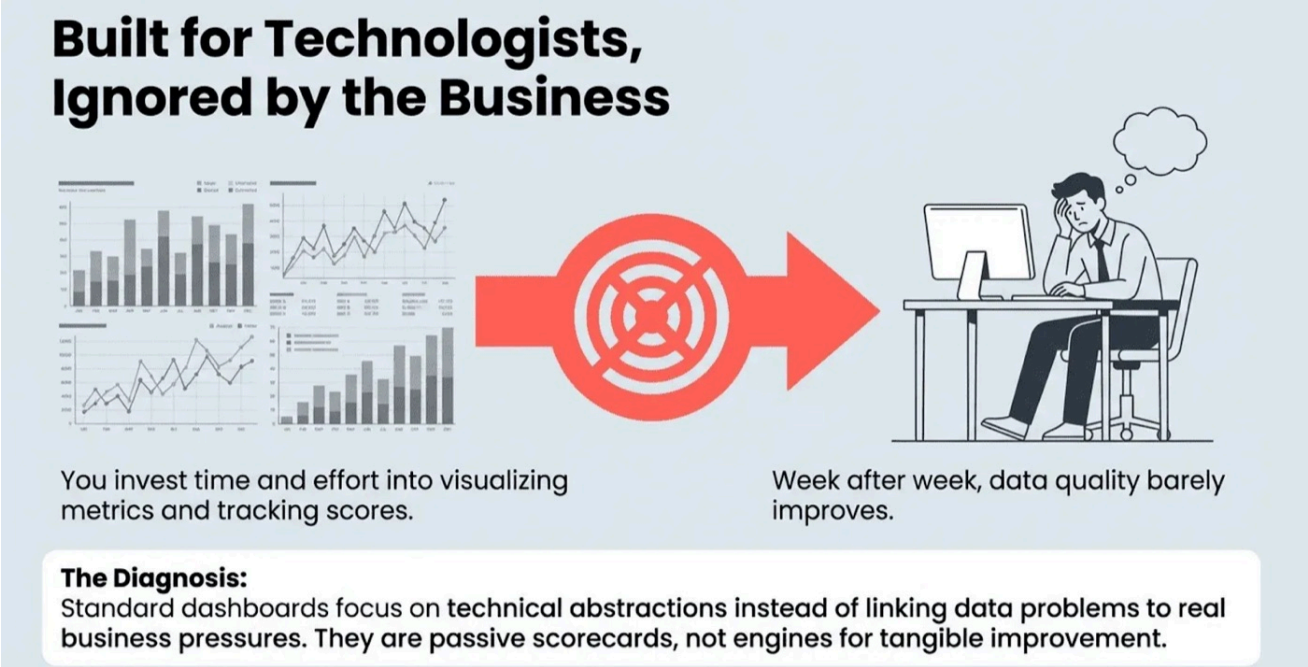
Why Your Data Quality Dashboard Isn't Working And What to Do About It

Read the original article at: https://datakitchen.io/blog/why-your-data-quality-dashboard-isnt-working-and-what-to-do-about-it/

Why Your Data Quality Dashboard Isn't Working And What to Do About It

Your organization has a data quality dashboard. You've put time and effort into visualizing metrics, tracking scores, and creating reports. But week after week, data quality barely improves. This is a common and frustrating problem. Poor-quality data can disrupt operations, mislead strategies, and damage trust. So why do so many dashboards, made to fix this, fail to make a real difference?

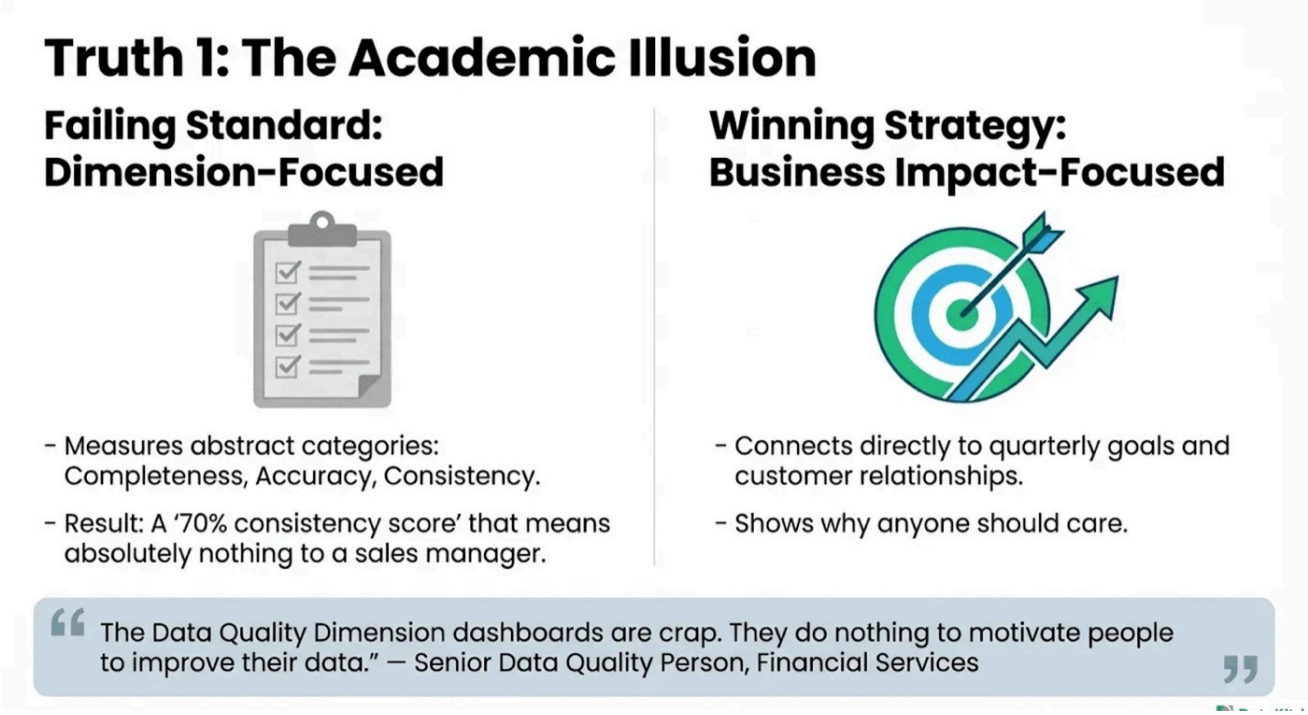
The truth is, not all dashboards are the same. Many are made by technologists for technologists, focusing on abstract data qualities and ignoring the people who need to act on the information. They focus on technical details instead of linking data problems to real business goals and pressures.



This article pulls back the curtain on why standard data quality dashboards fall short. We'll reveal six powerful, and perhaps surprising, truths about what actually makes a dashboard an effective tool for change, transforming it from a passive scorecard into an engine for tangible improvement.

1. "Standard" Dashboards Are Designed to Fail

The most common type of data quality dashboard is the Dimension-Focused dashboard. This approach evaluates data against a set of abstract academic categories such as completeness, accuracy, consistency, and timeliness. It seems logical; these are the fundamental building blocks of quality, after all. The result is a dashboard full of neat, tidy scores that signify absolutely nothing to the business.



The problem is that these abstract scores rarely inspire action. Telling a sales manager their data has a "70% consistency score" means little. It doesn't relate to their quarterly goals, customer relationships, or daily challenges. The metrics are too disconnected from what business teams focus on and care about. As one industry expert said:

> *"The Data Quality Dimension dashboards are crap. They do nothing to motivate people to improve their data." , Senior Data Quality Person, Financial Services, DataKitchen Market Research*

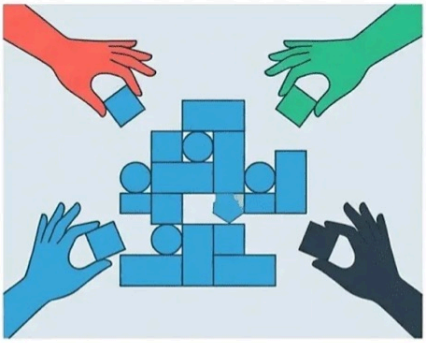
This gap between technical measures and business impact is the main reason these dashboards fail. They show the "what" but never explain "why anyone should care." The standard approach is flawed because it misunderstands the real problem.

2. Data Quality Isn't a Tech Problem; It's an Influence Problem

We often treat data quality as a technical challenge to be solved with better tools and stricter rules. In reality, it's a classic "Tragedy of the Commons." Everyone in the organization uses the shared data resource, but no single person or team is incentivized to maintain it.

Truth 2: The Tragedy of the Commons

Data quality is rarely a technology deficit. It is a shared resource problem where everyone consumes, but no one is incentivized to maintain.



The Myth

Data quality is fixed with stricter rules and top-down mandates.

The Reality

Data leaders cannot force busy teams to change habits.

The Shift

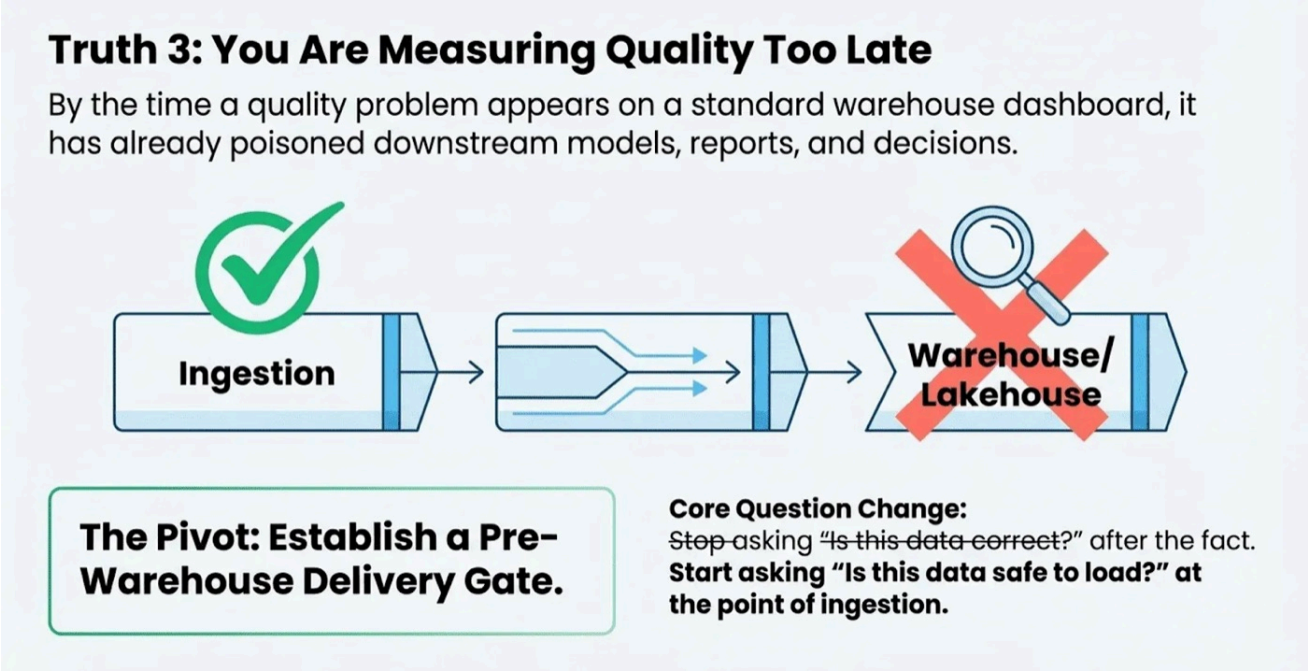
Move from technical enforcement to **INFLUENCE**. Start small, deliver value fast using DataOps principles, and persuade through results.

This reframes the role of a data quality leader. It's not a position of power; it's an **influence challenge**. Data quality leaders cannot make anyone fix data issues. They can't mandate that a busy call center team change its data entry habits or force a third-party supplier to clean up its feed. They must persuade, convince, and motivate others to take action.

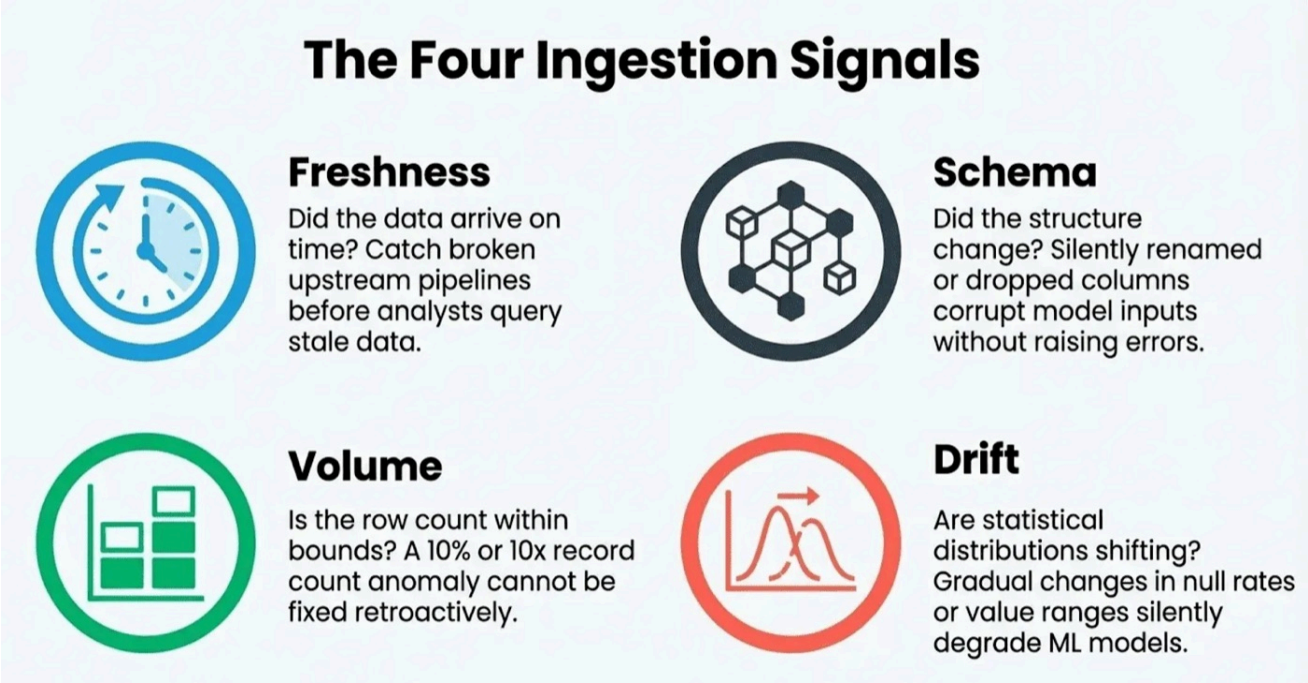
Real, lasting improvement doesn't come from top-down orders. It begins with one empowered person driving change through their influence. They succeed by focusing on a specific customer and, importantly, by starting small and iterating quickly using DataOps principles. Influence isn't just persuasion; it's a method that delivers value fast and builds the trust needed for wider change.

3. You're Measuring Quality Too Late: Monitor Data Delivery First

Most data quality dashboards assume the data they measure is already in your warehouse or lakehouse. For data engineers, this is a big blind spot. By the time a quality problem appears on a dashboard, it has often already affected downstream models, reports, and decisions.



The fix is to monitor data **during ingestion before it ever reaches** your analytical customers. You can establish checks on four key signals at the point of ingestion:



- **Freshness** , Did the data arrive on time? A late feed is often the first symptom of a broken upstream pipeline or a vendor outage. Catching it early prevents analysts from querying stale data without realizing it.
- **Schema** , Did the structure of the data change unexpectedly? A silently renamed column, a dropped field, or a changed data type can break pipelines and corrupt model inputs without raising any

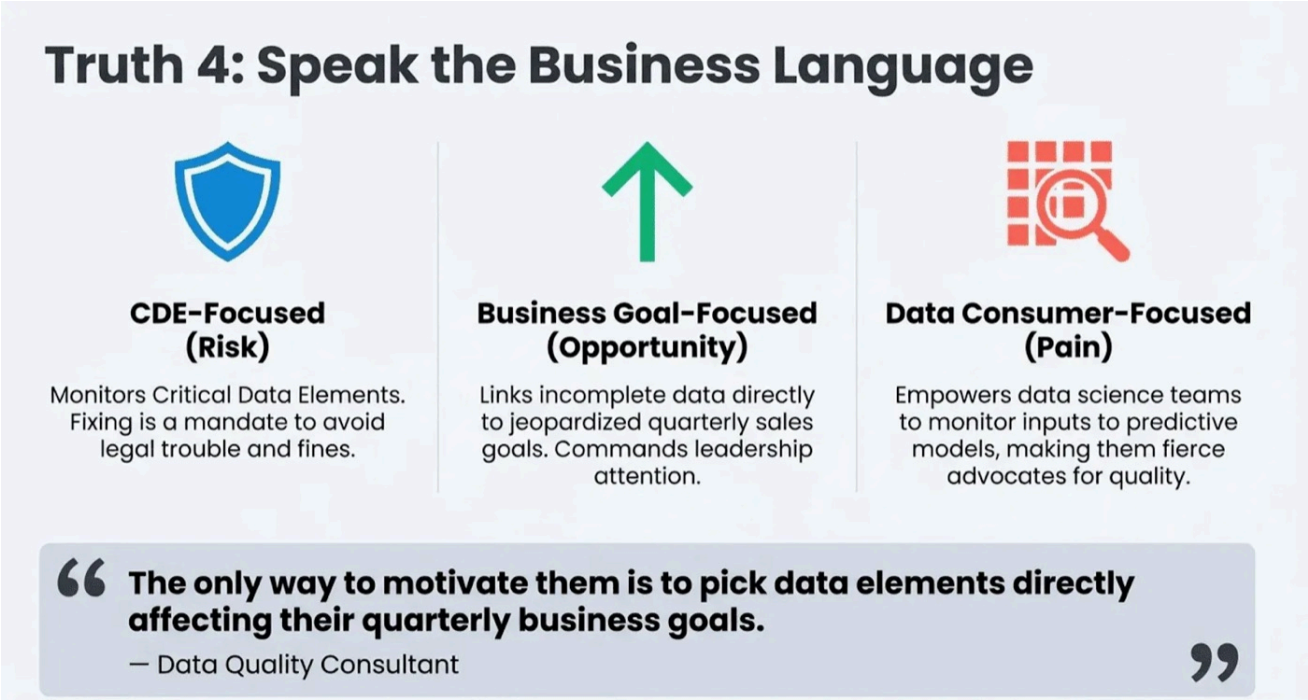
immediate errors.

- **Volume** , Is the row count within expected bounds? A file that arrives with 10% of its normal record count, or 10 times the normal count, signals a serious upstream problem that no amount of downstream quality checking can fix retroactively.
- **Drift** , Are the statistical distributions of key fields changing over time? Gradual drift in value ranges, null rates, or categorical distributions is how data quietly goes bad before any hard rule is violated. Drift is especially critical for teams feeding ML models, where training/serving skew can silently degrade model performance.

Think of this as a data delivery quality dashboard, a pre-warehouse gate that answers the question “Is this data safe to load?” instead of “Is this data correct?” Both questions are important, but they need to be asked in the right order. Finding a schema change or volume issue at ingestion takes minutes to fix; finding it after it’s passed through several models and reports can take days. A full data quality strategy covers both: monitoring delivery at ingestion and testing quality downstream.

4. To Drive Action, You Must Connect to What People Already Care About

If data quality is about influence, the answer is to stop focusing on abstract quality measures and start connecting data problems to the real goals your stakeholders care about. A dashboard becomes useful only when its metrics speak the business language: risk, cost, and opportunity. Instead of a generic “70% consistency score” that no one owns, the best dashboards show metrics for which a specific person is responsible.



Different types of dashboards succeed precisely because they make this connection:

- **CDE-Focused Dashboards** work in regulated industries because they monitor Critical Data Elements required for compliance. Fixing this data isn't a "nice-to-have"; it's a "MUST do" to avoid fines or legal trouble. The motivation is built in.
- **Business Goal-Focused Dashboards** are effective because they directly link data quality to achieving organizational objectives. When you can show that incomplete customer data is jeopardizing the quarterly sales goal, you suddenly have leadership's attention.
- **Data Consumer-Focused Dashboards** work by empowering the people who feel the pain of bad data most acutely. A dashboard built for a data science team to monitor the inputs to their high-value predictive models makes them powerful advocates for quality.

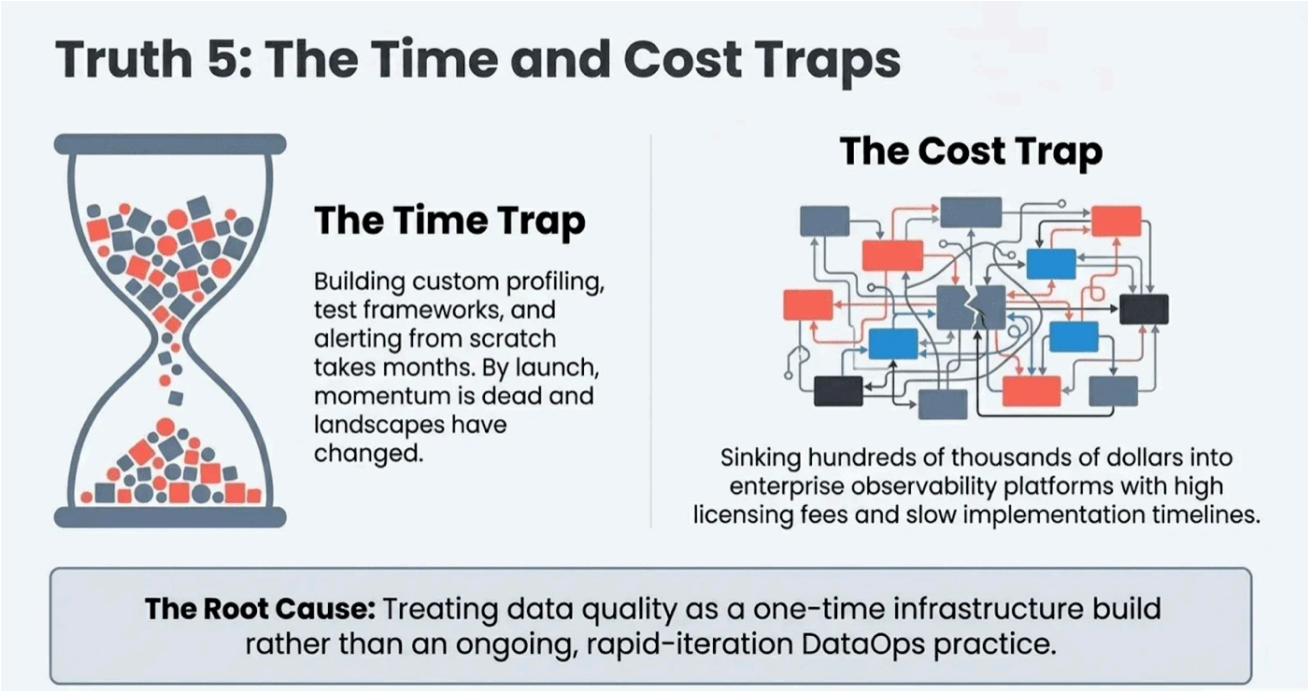
As one consultant said, the key is to focus on what already matters:

> *"How do you get the business to take action? … The only way to motivate them is to pick data elements directly affecting their quarterly business goals. They care about that." , Data Quality Consultant, DataKitchen Market Research*

The most effective dashboards aren't one-size-fits-all. They are tailored, customer-centric tools designed to align with a specific person's or team's existing motivations.

5. Dashboards Fail If It Takes Too Long to Build or Costs Too Much

Even the most thoughtfully designed dashboard is worthless if it never actually ships, or if the cost of building and maintaining it exceeds the value it creates. Yet this is exactly where many data quality programs quietly die.



Two failure modes are especially common:

The Time Trap. Building a data quality dashboard from scratch, writing custom profiling logic, designing test frameworks, wiring up alerting, and creating stakeholder-facing views can easily consume months of engineering time. By the time the dashboard is live, stakeholder priorities have shifted, the data landscape has changed, and the initial momentum is gone.Speed of delivery is not a luxury; it is a prerequisite for organizational trust.

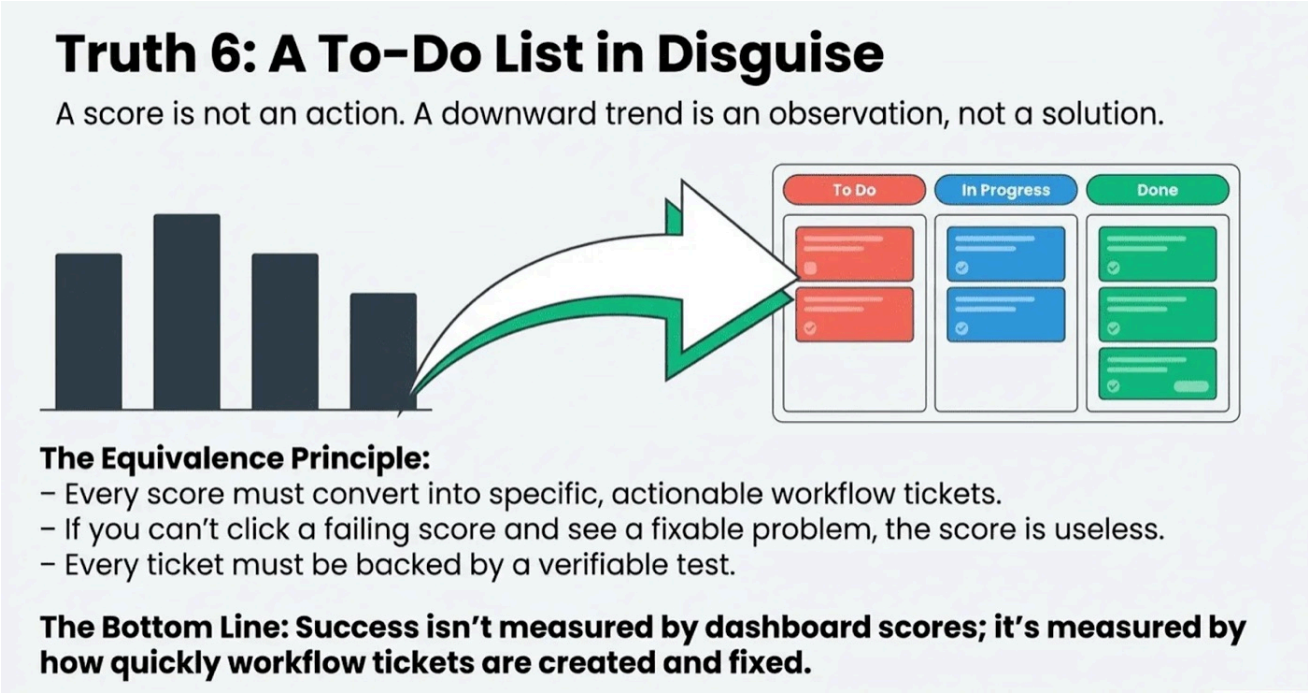
The Cost Trap. Many teams have invested hundreds of thousands of dollars in enterprise data observability platforms, only to find that the promised value never materializes. Licensing costs are high, implementation timelines stretch out, and the tools often require specialized expertise to operate. Meanwhile, the core problems, bad data, broken pipelines, and unhappy consumers, remain unsolved.

Both traps share the same root cause: treating data quality as a one-time infrastructure project rather than an ongoing operational practice. A DataOps approach breaks this pattern. It emphasizes short delivery cycles, rapid iteration, and continuous feedback loops, the same principles that transformed software development are just as applicable to data quality work.

The goal isn't to eliminate investment in data quality tooling; it's to ensure that investment starts delivering value immediately and continues to do so as the program grows.

6. An Effective Dashboard Is Simply a To-Do List in Disguise

A score, no matter how accurate, is not an action. A chart trending downward is an observation, not a solution. The most profound and counterintuitive truth is that an effective dashboard's purpose is not to report a number, but to **generate a to-do list**.



Effective dashboards must be built from discrete, concrete actions. Their foundations are data quality test results, not fuzzy math based on profiling. This idea is formalized in the Data Quality Dashboard , Data Quality Test Results Equivalence Principle, which states two things:

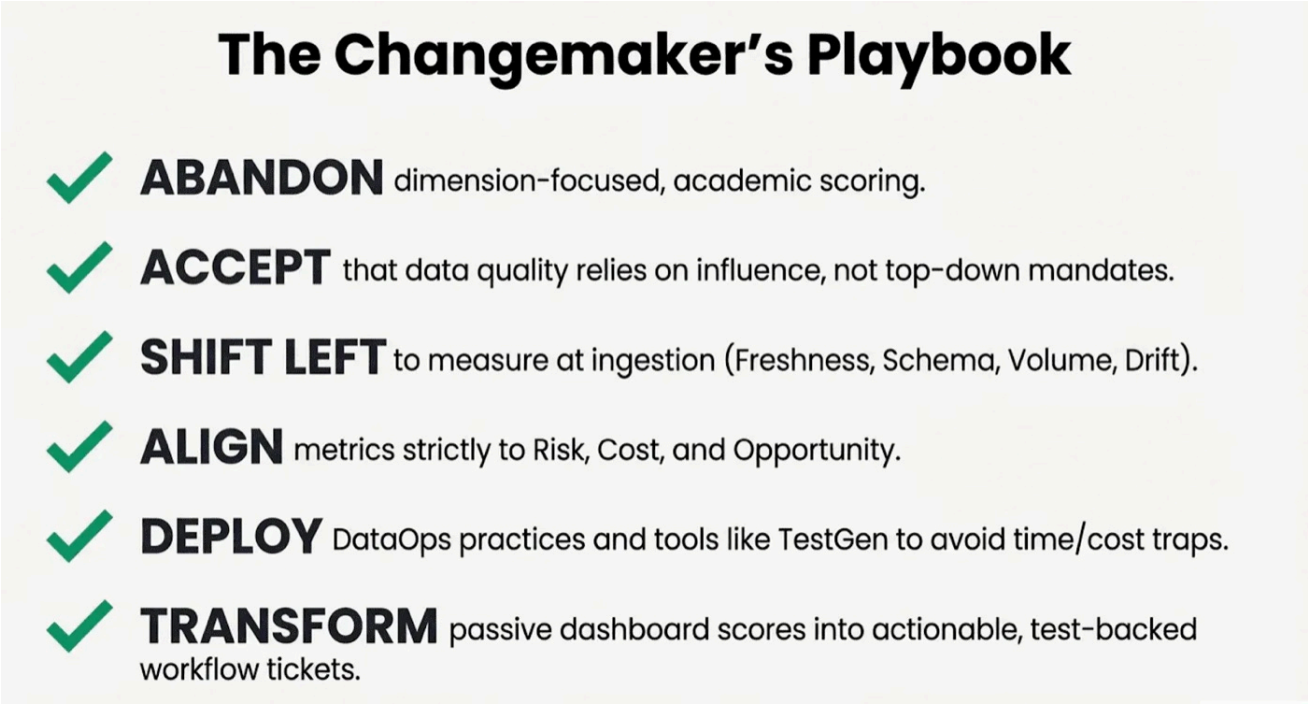
Every score on a data quality dashboard must be convertible into a series of actionable workflow tickets. If you can't click a failing score and see a list of specific, fixable problems, the score is useless.

Every workflow ticket must be supported by a specific test that both demonstrates the issue and can be used to verify the fix. This ensures every problem is real and every solution can be proven.

This principle changes how we think about dashboards. For the most action-focused teams, the "dashboard" isn't a set of charts; it's the workflow itself. Success isn't measured by scores but by how quickly tickets are created and fixed.

Move From Scorekeeper to Changemaker

Improving data quality isn't about finding the perfect metric or building the most detailed dashboard. It's about changing your mindset, from being a passive scorekeeper to an active influencer and changemaker.



This means leaving behind the one-size-fits-all, dimension-focused method. It means spotting data problems at delivery before bad data reaches your warehouse. It means linking quality metrics to what different stakeholders care about. It means moving quickly while keeping costs low using a DataOps mindset and open-source tools. Most importantly, it means making sure every insight leads to clear, actionable tasks that can be assigned, tracked, and fixed.

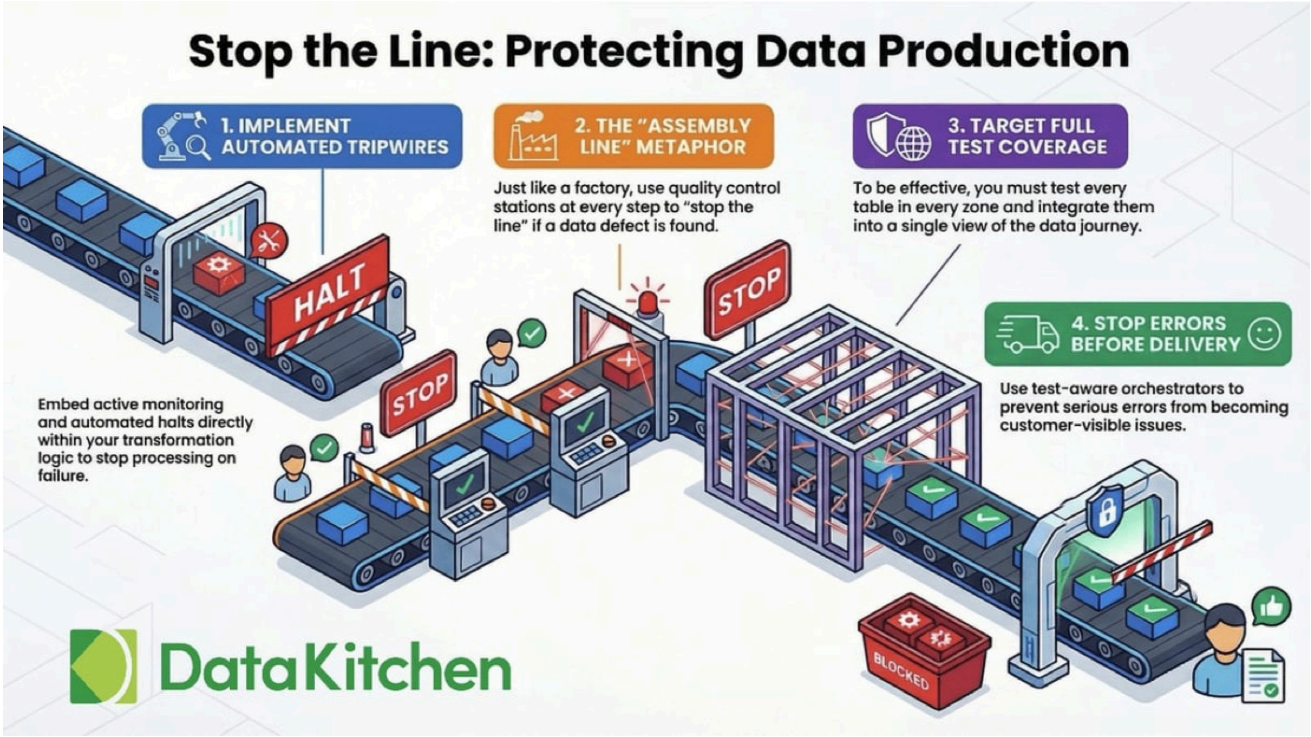
When your dashboard becomes a source of clear, prioritized work, you finally have a true engine for change.

Instead of asking “What’s our data quality score?”, what if the only question that mattered was, “Whose job did we make easier today?” What would you build then?

MONITORING AND OBSERVABILITY

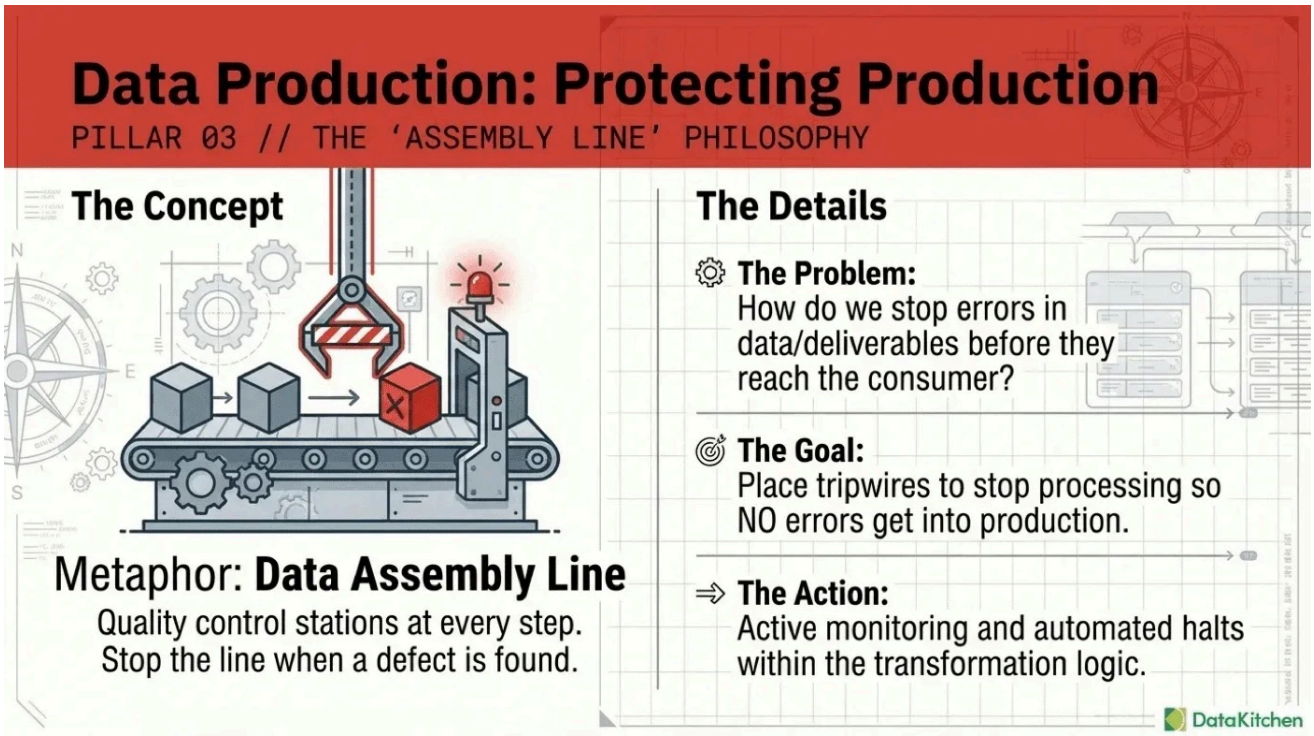
Data Production Tripwires in Databricks: Stop Bad Data Before It Reaches Production

Read the original article at: https://datakitchen.io/blog/data-production-tripwires-in-databricks-stop-bad-data-before-it-reaches-production/



That is the failure mode that keeps data engineers up at night , not the pipeline crash you can see in the logs, but the quiet corruption that passes through every check and lands in a dashboard where a VP is about to make a decision. The job finished. The scheduler marked it complete. And somewhere downstream, a model trained on a column that silently lost 30 percent of its values, or a customer report went out with records that should have been filtered.

This is the problem: pipeline success and data correctness are not the same thing. A SQL transform can run without errors and produce a table full of garbage.



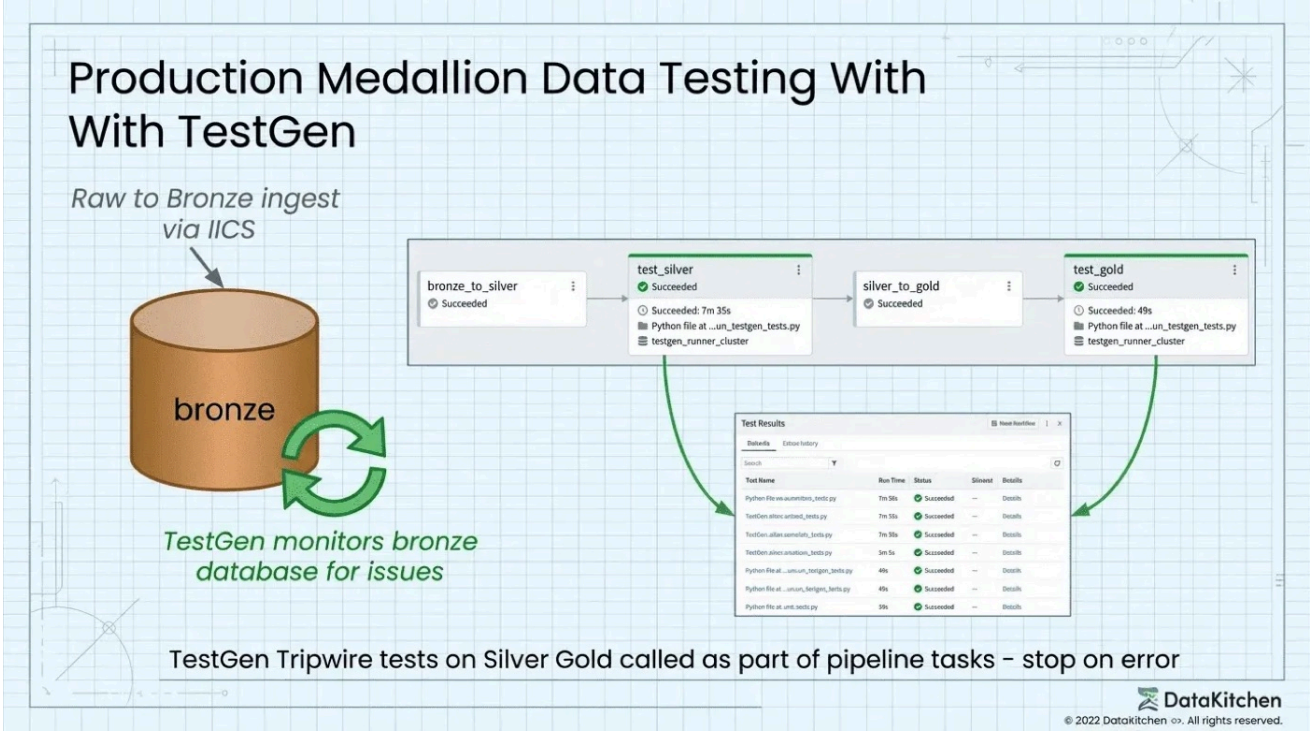
The medallion architecture helps, in theory. Bronze, silver, and gold layers create natural checkpoints where transformation logic runs and data takes a new form. But checkpoints are only useful if someone is actually standing at them. In most pipelines, those checkpoints are unmanned. Data flows through regardless. A transformation succeeds, a new table is written, and downstream consumers see whatever came out the other side.

What we wanted was a way to make the pipeline aware of data correctness at each layer transition , not just structural correctness. Something that would stop the flow if the data failed quality checks, leave the downstream tables untouched, give us a clear signal about what failed and why, and do all of this inside the normal Databricks workflow machinery so that operators did not have to learn a new system.

We call these tripwires. Here is how they work and how to add them to an existing pipeline.

What a Tripwire Does

A tripwire is a task in the Databricks job DAG that sits between two transformation steps. After bronze-to-silver runs, a tripwire runs. After silver-to-gold runs, another tripwire runs. Each tripwire hands control to TestGen, waits for the results, and either passes execution to the next step or fails the job with a clear error message.



The key behavior is the failure mode. If a tripwire task exits with a non-zero code, Databricks treats it the same as any other failed task: downstream tasks do not run. The gold layer does not get written. The bad silver data sits in place while the good gold data from the last successful run stays untouched. Your production tables are protected by the same dependency graph that already governs your pipeline.

The tripwire is not a side process that logs somewhere and hopes someone looks. It is a first-class gate in the workflow.

Adding a Tripwire to an Existing Pipeline

If you already have a Databricks pipeline you cannot rewrite from scratch, adding tripwires is a targeted change, not a redesign.

```
The core addition is a new task between your existing transformation steps. It needs
depends_on pointing at the upstream transform, and downstream tasks need depends_on
pointing at it. That is the entire structural change to your job definition. The tripwire
script handles everything else.
You will also want to add run_tests as a job-level parameter rather than a bundle variable.
The distinction matters: bundle variables are resolved at deploy time and baked into the
job definition. A job parameter can be overridden at run time without redeploying. When you
need to skip TestGen for a backfill or a debugging run, you pass --params run_tests=false
to the job run command and the transforms execute normally while the tripwire tasks exit
cleanly. No YAML edits, no redeployment. This is particularly useful in an existing
pipeline where your team needs an escape hatch while they tune the test suites.
```

Deploying a New Pipeline with Databricks Asset Bundles

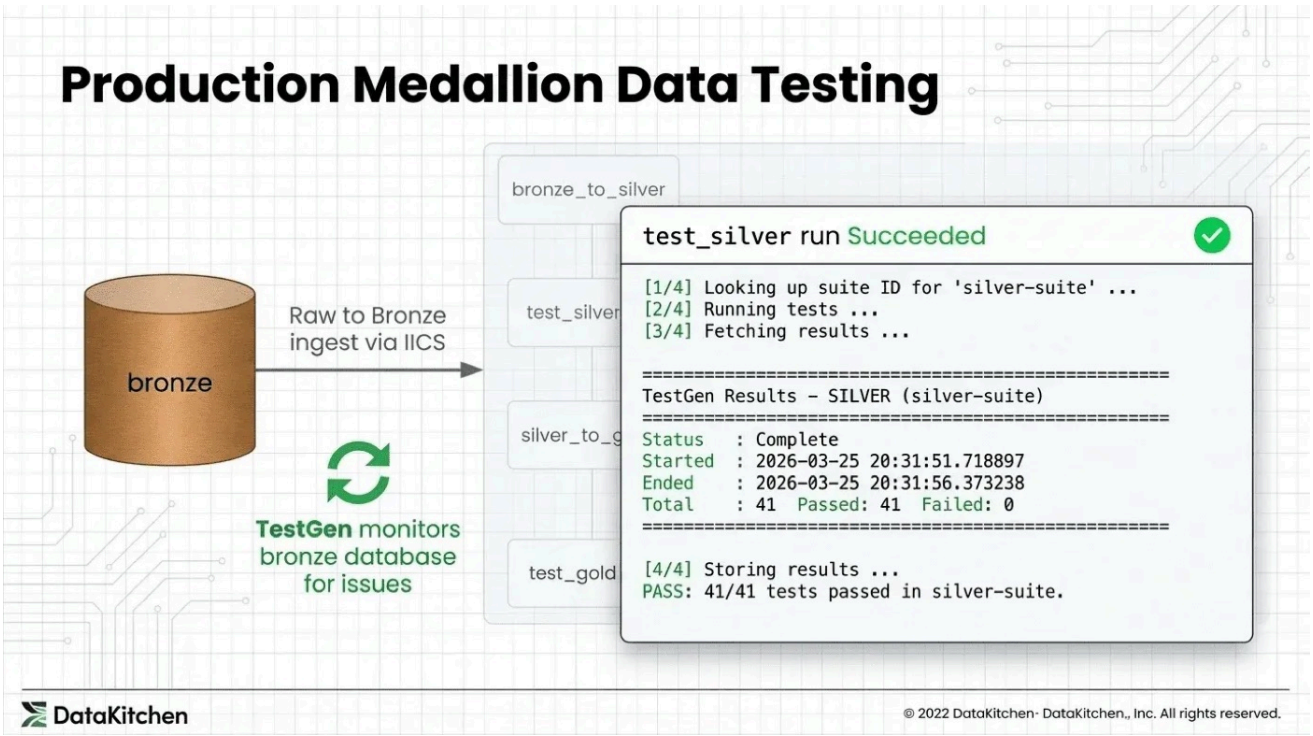
For new pipelines, we define and deploy everything, including the tripwire tasks, using Databricks Asset Bundles (DAB). A bundle is a YAML-first way to define Databricks jobs, clusters, and workspace resources as code, version them in Git, and deploy them to multiple environments without manual configuration.

```
The top-level databricks.yml declares the bundle, points at the job definitions, and
defines variables that change between environments:
bundle:
  name: testgen-demo

variables:
  env:
    description: "Environment name — selects the SQL subdirectory (dev|prod)"
    default: "prod"
  run_tests:
    description: "Set to 'false' to skip TestGen test tasks (transforms still run)"
    default: "true"

targets:
  dev:
    mode: development
    default: true
    variables:
      env: "dev"
      bronze_schema: "dev_phase0_bronze"
      silver_schema: "dev_phase0_silver"
      gold_schema:   "dev_phase0_gold"

  prod:
    mode: production
    variables:
      env: "prod"
    run_as:
      user_name: name@datakitchen.io
When you deploy to dev, DAB automatically prefixes job names with your username, deploys
files to a user-scoped workspace path, and resolves ${var.env} to "dev" throughout. Two
engineers can have their own dev pipelines running simultaneously without collision.
Deploying to prod uses the shared stable path and runs as a service account:
databricks bundle deploy --target dev
databricks bundle deploy --target prod   # CI only
The job definition lives in resources/jobs/phase0_pipeline.yml. The four tasks form a
linear DAG: bronze_to_silver, then test_silver, then silver_to_gold, then test_gold. The
tripwire tasks reference the same Python script with a --layer argument to select which
test suite to run:
tasks:
  - task_key: bronze_to_silver
    sql_task:
      warehouse_id: ${var.warehouse_id}
      file:
        path: ${workspace.file_path}/phase0/sql/${var.env}/bronze_to_silver.sql

  - task_key: test_silver
    depends_on:
      - task_key: bronze_to_silver
    job_cluster_key: testgen_runner_cluster
    spark_python_task:
      python_file: ${workspace.file_path}/phase0/run_testgen_tests.py
      parameters:
        - "--layer"
        - silver
```

```
        - "--env"
        - ${var.env}
        - "--run-tests"
        - "{{job.parameters.run_tests}}"

  - task_key: silver_to_gold
    depends_on:
      - task_key: test_silver
    sql_task:
      warehouse_id: ${var.warehouse_id}
      file:
        path: ${workspace.file_path}/phase0/sql/${var.env}/silver_to_gold.sql

  - task_key: test_gold
    depends_on:
      - task_key: silver_to_gold
    spark_python_task:
      python_file: ${workspace.file_path}/phase0/run_testgen_tests.py
      parameters:
        - "--layer"
        - gold
        - "--env"
        - ${var.env}
        - "--run-tests"
        - "{{job.parameters.run_tests}}"
```

The Current Integration: SSH and the Docker Shell

TestGen runs on a dedicated VM, containerized with Docker. The current integration reaches into that container from a Databricks job cluster using SSH. This approach works, but we want to be direct about what it costs: SSH key rotation, port whitelisting, a tolerance for shelling into containers from remote compute, and in regulated environments, a likely conversation with your security team before it clears review. If that is a hard stop for your organization, skip ahead to the HTTP section , that path is coming and removes all of it.




```
Here is how the current integration works. When the tripwire task fires, it runs a Python
script on the cluster. That script opens an SSH connection to the TestGen VM, then uses
docker compose exec to run commands inside the TestGen engine container. The SSH key lives
in a Databricks secret scope, base64-encoded to survive the round-trip through the secrets
API without newline corruption:
key_b64 = dbutils.secrets.get("testgen", "ssh-private-key")
key_content = base64.b64decode(key_b64).decode("utf-8")
tf = tempfile.NamedTemporaryFile(mode="w", suffix=".pem", delete=False)
tf.write(key_content)
os.chmod(tf.name, 0o600)
```

That key file exists only for the duration of the task run on the cluster. The VM hostname is a stable DNS name rather than a public IP, so it survives VM stop and start cycles without configuration changes.

```
With the key in place, the script can tunnel commands into the container. A small helper
sends arbitrary Python to python3 - inside the engine — note that this executes inside the
TestGen container, not on the Databricks cluster:
def docker_python(cfg, script):
    proc = subprocess.Popen(
        ssh_base(cfg) + ["docker compose exec -T engine python3 -"],
        stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
        text=True,
    )
    stdout, stderr = proc.communicate(input=script)
    return stdout + stderr
```

The script first looks up the test suite UUID by querying TestGen's internal database through that helper , this query runs inside the container:

```
lookup_script = """
from testgen.common.models import Session
from sqlalchemy import text
with Session() as s:
    row = s.execute(text(
        "SELECT id FROM test_suites WHERE test_suite = :k AND project_code = 'DEFAULT'"
    ), {'k': 'silver-suite'}).fetchone()
    print(str(row[0]) if row else 'NONE')
"""
suite_id = docker_python(cfg, lookup_script)
```

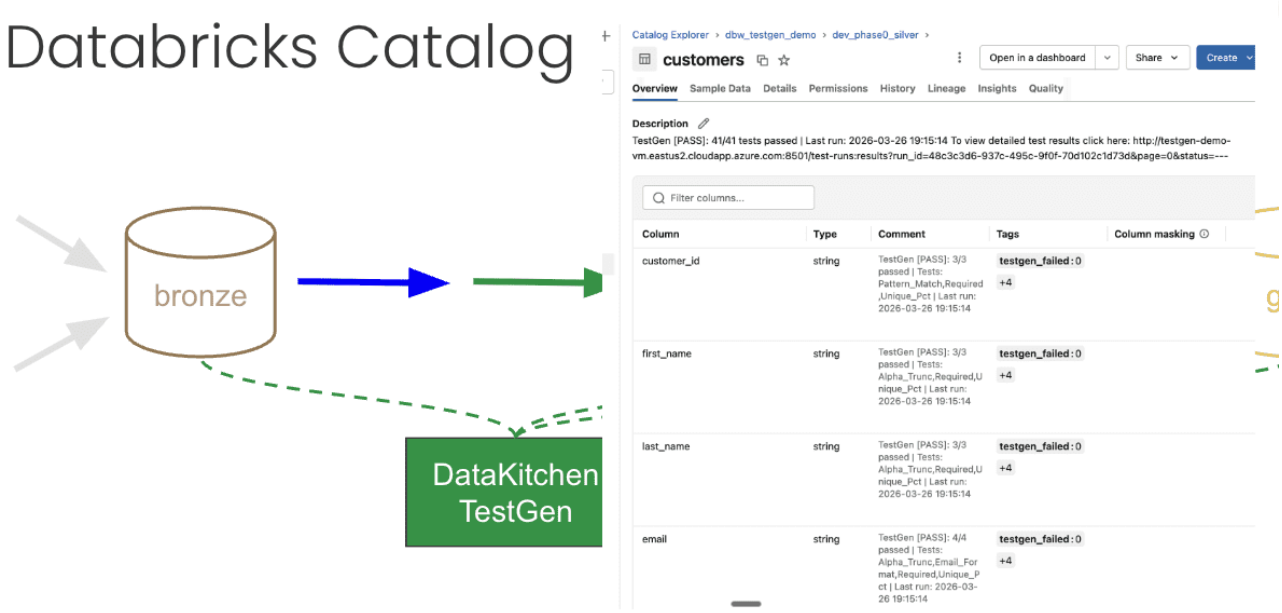
Once the suite ID is known, the TestGen CLI triggers the test run , again, this command executes inside the container via SSH:

```
subprocess.run(
    ssh_base(cfg) + [f"docker compose exec -T engine testgen run-tests -t {suite_id}"],
    capture_output=True, text=True,
)
Results come back the same way, queried from TestGen's test_results table over the SSH
channel and deserialized as JSON on the cluster side. Once results are local to the
cluster, the script writes them to a Delta table in Unity Catalog, stamps each tested table
and column with tags showing pass and fail counts, and then checks whether anything failed:
failed = int(summary.get("failed_ct", 0) or 0)
if failed > 0:
    sys.exit(f"FAIL: {failed} test(s) failed in {suite_key}.")
That sys.exit with a non-zero code is the entire enforcement mechanism. Databricks sees a
failed task and halts the job. silver_to_gold never runs. Your gold tables are protected.
```

What You Get When a Tripwire Fires

When a tripwire fails, here is the exact sequence of what happens and what you do next.

The Databricks job run shows a failed task. The task log names exactly how many tests failed and in which suite. You open the Delta table that the tripwire wrote to and query it , every test result is there, with the tested table, column, test type, and whether it passed or failed. If you go to Catalog Explorer, the silver or gold table has tags showing the failure status directly on the asset. The table comment contains a deep link to the specific TestGen run, so there is no hunting for which run produced the results you are looking at.



All TestGen Test Results in Unity Catalog

The data in the layer that failed is still there. The layer downstream of it was never written. The good data from the last successful run is still in your production tables. You investigate the failure, patch the upstream data or fix the transform logic, and re-run from the failing step. When the re-run passes the tripwire, the pipeline continues and your production tables get written with data that actually passed the checks.

This is the value of the tripwire pattern: it compresses the feedback loop. Instead of discovering a data quality problem when a stakeholder files a ticket, you discover it inside the same workflow run that produced the problem, before it crosses into the next layer. The 2am Slack message does not happen because the bad data never got to the dashboard.

What Is Coming: HTTP

The SSH-into-Docker approach is what we have now. TestGen is shipping an HTTP API, and when it does, the integration becomes a straightforward REST call: trigger a test run, poll for completion, fetch the results as JSON. No SSH, no Docker exec, no temporary key files, no security review conversation. Any compute environment that can make an outbound HTTP request will integrate with TestGen the same way we do it today.

The tripwire concept stays exactly the same. You still put a task between your transformation steps, you still fail the job if tests fail, you still write results back to Delta and tag your tables. The transport changes. The gate stays.

Why This Pattern Works

The reason the tripwire pattern works is that it respects what Databricks is already good at: expressing dependencies between tasks and propagating failures through a DAG. We did not build a separate orchestration layer or a monitoring sidecar. We used the existing job dependency graph as the enforcement mechanism.

The tests themselves are defined and managed in TestGen, not in the pipeline code. That separation matters. The data engineer who sets up the pipeline does not need to author individual test cases. TestGen profiles the data, generates a test suite based on what it observes, and those tests run at each gate. Updating the tests does not require touching the pipeline definition.

The combination of a standard orchestration mechanism for enforcement and a separate tool for test management is what makes the tripwire practical to operate at scale, not just interesting to build.

SECTION 5

Implementation and Case Studies

1. Data Testing Anti-Patterns: Why Your Tests Are Failing You
2. Three Data Quality Disasters and How They Were Averted
3. When Not to Test: Setting Reasonable Bounds on Data Quality Effort
4. I Wrote Data Quality Tests for 3 Million NYC Taxi Rides in Five Minutes
5. Data Quality's Most Wanted: The 2026 Case Files
6. Why Testing Commercial Pharma Data Is Harder Than Anyone Tells You
7. IQVIA, specialty pharmacy, and your own files: the commercial data most likely to be wrong
8. The Illusion of Clean Data: The Pharma Identifiers Generic Tools Never Check

Data Testing Anti-Patterns: Why Your Tests Are Failing You

IMPLEMENTATION AND CASE STUDIES

Data Testing Anti-Patterns: Why Your Tests Are Failing You

The DataOps Way to Data Quality and Data Observability

DataKitchen

A test suite that does not catch real problems is worse than no test suite at all. It creates false confidence. Engineers assume the data is good because the tests pass. Business users trust the numbers because the team says everything is monitored. Then something breaks badly, and the question everyone asks is: why didn't the tests catch this?

The answer, in most cases, is one of the anti-patterns in this chapter. Each section describes a common mistake: how to recognize it, why it happens, and what to do instead.

Anti-Pattern 1: Testing Specific Values Instead of Ranges or Relationships

How to recognize it.

You find tests in your suite that look like this:

```
-- Test: daily revenue equals expected value
SELECT COUNT(*)
FROM daily_revenue
WHERE revenue_total != 1234567.89
```

Or in dbt:

```
- name: revenue_total
  tests:
    - accepted_values:
        values: [1234567.89]
```

Any test that asserts a column must equal a specific number (rather than a range, a relationship, or a known set of categorical values) belongs to this category.

Why it happens.

Whoever wrote the test was looking at a specific day's data and used what they saw as the expected value. It seemed precise. It seemed thorough. In the moment, it passes.

Why it breaks you.

Data changes. Revenue is different every day. The moment any valid transaction occurs that changes the total, the test fails. The team investigates, finds nothing wrong, and either deletes the test or marks it as expected-to-fail. In either case, the test provides no long-term value. Worse, if the test stays in the suite and the team has learned to ignore its failures, it masks the signal from other failures that do matter.

What to do instead.

Test properties, not specific values. Revenue should be non-null, non-negative, and within a reasonable range of the prior period. A useful test:

```
-- Revenue is positive and within 30% of last week's same-day value
SELECT COUNT(*)
FROM daily_revenue d
JOIN daily_revenue_prior_week pw
  ON d.report_date = pw.report_date
WHERE d.revenue_total <= 0
   OR d.revenue_total > pw.revenue_total * 1.30
   OR d.revenue_total < pw.revenue_total * 0.70
```

For categorical columns where the full set of valid values is known (US state codes, ISO currency codes, order status values), `accepted_values` tests are appropriate. But the list should be a business-defined set of valid codes, not a single observed value.

Anti-Pattern 2: Cardinality Tests Without Understanding the Table

How to recognize it.

Tests that assert a table must have exactly N rows:

```
- name: orders
  tests:
    - dbt_utils.equality:
        compare_model: ref('orders_expected_count')
```

Or simpler versions that compare against a hardcoded integer.

Why it happens.

Row count tests are one of the first checks every team learns. Someone runs `SELECT COUNT(*)` on a table on a Tuesday, sees 10,423 rows, and writes a test asserting the count must be 10,423. This seems reasonable until the next load adds 200 more rows and the test fails.

Why it breaks you.

A row count test that uses a hardcoded value is not testing data quality. It is testing that the data has not changed since the day you wrote the test. For any table that receives new data (which is most tables), this test will fail constantly and correctly, because the table is doing exactly what it is supposed to do.

When this test fails constantly on valid data, one of two things happens. Either the team disables the test, losing all row count monitoring. Or the team updates the hardcoded number on every load, turning a monitoring check into a manual maintenance task.

What to do instead.

Test that row count is reasonable relative to history. The right question is not "does the table have exactly N rows?" but "is today's row count in the expected range given what we know about this table?"

For append-only tables (events, transactions, logs), test that today's increment is within a reasonable range of recent daily increments:

```
-- Daily new rows should be within 50% of the 14-day average
WITH daily_counts AS (
  SELECT
    DATE(created_at) AS load_date,
    COUNT(*) AS row_count
  FROM events
  WHERE created_at >= CURRENT_DATE - 15
  GROUP BY 1
),
avg_count AS (
  SELECT AVG(row_count) AS avg_daily_rows
  FROM daily_counts
  WHERE load_date < CURRENT_DATE
)
SELECT COUNT(*)
FROM daily_counts d
CROSS JOIN avg_count a
WHERE d.load_date = CURRENT_DATE
  AND (d.row_count < a.avg_daily_rows * 0.5
    OR d.row_count > a.avg_daily_rows * 2.0)
```

For full-refresh tables (dimension tables that are reloaded completely each run), test that row count has not dropped more than 5% from the prior run. A drop of that magnitude usually indicates a source system problem.

Anti-Pattern 3: Over-Alerting Leading to Alert Fatigue

How to recognize it.

Your #data-alerts Slack channel has more than 20 messages per day. Engineers have muted the channel. Business stakeholders have stopped asking about alert status because they assume it is always noise. The last three genuine data incidents were not caught by the alerting system.

Why it happens.

Teams write tests quickly and deploy them all with the same alert configuration. Every failure sends the same notification to the same channel. Thresholds are set too tight because the team was being conservative. Known-noisy conditions (weekend loads, holiday traffic) were never accounted for.

Why it breaks you.

The team that cried wolf. When every alert is potentially noise, no alert gets treated as real. This is the exact opposite of the intended outcome. You have built a monitoring system that provides zero protection.

What to do instead.

Tier your alerts (P1, P2, P3) and route them accordingly. Suppress checks during known load windows. Calibrate thresholds using historical data, not intuition. Set a rule: if an alert fires more than three times per week without a corresponding real incident, the threshold is wrong and must be adjusted immediately.

Treat alert noise as a bug. When a false positive fires, fix the threshold the same day. Do not let false positives accumulate. Every unaddressed false positive is a small erosion of trust in the system.

This topic is covered in depth in the alerting chapter, but the core principle belongs here too: a test that always fires is not a test. It is noise. Delete it or fix it.

Anti-Pattern 4: Test Suites That Grow Too Large to Maintain

How to recognize it.

Your test suite has more than 500 tests. When you ask a team member which tests cover the revenue pipeline, they do not know. The last time someone reviewed whether all the tests were still valid was more than six months ago. You have tests for tables that were deprecated. You have tests for columns that no longer exist. The test run takes 45 minutes and the team does not know which failures to prioritize.

Why it happens.

Adding tests is easy. Removing or updating them is work. Tests accumulate over time as new tables are added, as new engineers add checks without reviewing what already exists, and as nobody takes ownership of the overall suite health. By the time the problem is visible, the suite is unmanageable.

Why it breaks you.

A test suite nobody trusts and nobody maintains is not providing safety. Engineers learn to ignore the red lights in the CI run because 40% of them are stale tests that have been failing for months on tables nobody uses. The signal from real failures is completely buried.

What to do instead.

Assign test ownership. Every test in the suite should have a named owner, ideally tied to the table owner. The table owner is responsible for ensuring the tests for their table are valid, well-calibrated, and actionable.

Implement coverage tiers. Not every table needs the same depth of testing. Tier 1 tables get freshness, row count, column-level, and business rule tests. Tier 2 tables get freshness and row count. Tier 3 tables get freshness only. This structure limits the growth of the suite to something proportional to the number of critical tables.

Prune quarterly. Schedule a 30-minute quarterly review where the team identifies tests that have been consistently failing for more than 30 days without a corresponding incident response. These are either false positives or covering issues nobody cares about. Delete or fix them.

A well-maintained suite of 100 tests covering the 20 most important tables is more valuable than an unmaintained suite of 2,000 tests covering everything.

Anti-Pattern 5: Tests That Pass While Hiding Real Errors

How to recognize it.

Your pipeline tests pass every day. Your CI checks are all green. But business users are reporting that the numbers look wrong. When you investigate, you find that the pipeline ran successfully, the row counts are correct, but the data inside the rows is incorrect: wrong values, reversed joins, stale lookups, or logic errors that produce plausible-looking but incorrect results.

Why it happens.

Teams naturally start by testing the pipeline itself rather than the data it produces. Did the job run? Did it complete without errors? Did the right number of rows load? These are process tests. They are easy to write because the information is available in your orchestration tool. They tell you that the pipeline ran. They tell you nothing about whether the output is correct.

Why it breaks you.

A pipeline that runs successfully can still produce wrong data. A join on the wrong key produces rows with incorrect values but the correct count. A date filter that is off by one includes the wrong day's data. A calculation that uses an outdated exchange rate produces revenue figures in the wrong currency. All of these pass process tests while delivering bad data to the business.

What to do instead.

Test outcomes, not processes. For every critical pipeline, ask: what properties should the output data have if the pipeline ran correctly? Then test those properties.

Examples:

- For a revenue calculation: the sum of order-level revenue in the orders table should reconcile within 0.1% of the total in the revenue_summary table.
- For a user attribution pipeline: every user_id in the attributed_events table should exist in the users table.
- For a product inventory feed: no product should have a negative available quantity unless it is marked as a backorder item.
- For a daily close table: the sum of all line items should equal the reported total for every account.

These are data content tests. They require you to think about what correct output looks like, not just whether the job finished. This is harder than writing process tests, but it is the only way to catch the class of errors that process tests miss entirely.

Anti-Pattern 6: Missing Coverage on the Tables That Actually Matter

How to recognize it.

Your test suite is large and well-maintained. But when you map your tests to the tables your business partners actually use for decisions, you find that the most heavily used dashboards are backed by tables with thin or zero test coverage. Your raw ingestion tables have 50 tests each. Your executive revenue report has two.

Why it happens.

Test coverage grows organically, following the path of least resistance. Raw ingestion tables are easy to test: the schema is defined by the source system, the expected behavior is predictable, and the engineers who wrote the ingestion code are the same ones who write the tests. Aggregate marts and executive-facing tables are harder to test because they involve complex business logic, involve multiple stakeholders in defining what "correct" means, and are often maintained by more senior engineers who are busy with other priorities.

Why it breaks you.

The risk-to-coverage ratio is inverted. The tables with the most tests are the ones with the lowest business impact if they fail. The tables with the most business impact have the fewest tests. You are spending your testing effort where it matters least.

What to do instead.

Build a coverage map. List your 20 most critical tables (the ones executives and business partners actually rely on) and map your current test coverage to each one. This single exercise usually reveals the problem clearly and immediately.

Implement risk-based coverage tiers. Start from the downstream consumer and work backward. Identify which tables feed the dashboards and models that drive decisions. Ensure those tables have the deepest coverage. Use the tier structure described earlier (freshness, row count, column-level, business rules for Tier 1) and apply it strictly to the high-impact tables first.

Audit which tables the business actually uses. Ask your BI team or your data catalog tool which tables are queried most frequently. Ask your analytics engineers which tables back the most-used dashboards. Let usage drive coverage priorities.

Anti-Pattern 7: Testing in Development but Not in Production

How to recognize it.

Your CI pipeline runs all tests on every pull request. Your development environment is green. But you have no continuous monitoring running against your production database. Tests run when code changes but not on the schedule your data arrives.

Why it happens.

Development testing is part of a natural software engineering workflow. Engineers write code, write tests, run tests, merge. This workflow tests the code logic but not the production data stream. Data quality failures in production are caused by changes in incoming data, not changes in code. A test that runs only when code changes will not catch a source system that started sending null values last Tuesday.

Why it breaks you.

A significant proportion of data quality failures are not caused by code changes. They are caused by upstream changes, volume anomalies, schema drift, and operational issues that affect live production data independent of any deployment. Testing only in development creates a blind spot for the entire category of production data failures.

What to do instead.

Production monitoring is not optional. You need a scheduled process that runs data quality checks against production data on a regular cadence, independent of deployment activity. For critical tables, this means checks running at minimum once per day, and for high-velocity tables, checks running after every load.

This does not require a separate tool. In dbt, source freshness checks run as scheduled jobs. SQL-based checks can run on a cron schedule in your orchestration tool. Commercial observability tools are explicitly designed for continuous production monitoring.

The development test suite and the production monitoring suite serve different purposes and should be managed separately. Development tests validate code correctness. Production monitoring validates data correctness. Both are necessary.

Building a Healthier Test Suite

Most of these anti-patterns share a common root: tests get written quickly and without a clear definition of what a useful test actually does. A useful test has three properties. It catches real failures. It does not fire on valid data. And it points clearly to what needs to be fixed when it does fire.

Audit your current suite against these three criteria. Delete or fix everything that fails the criteria. Then implement the ownership, tiering, and review cadence that keeps the suite healthy over time.

A test suite is not an asset that appreciates automatically. It requires active maintenance. The teams with the best data quality outcomes are not the ones with the most tests. They are the ones with the right tests, maintained by people who take responsibility for keeping them accurate.

Three Data Quality Disasters and How They Were Averted

IMPLEMENTATION AND CASE STUDIES

Three Data Quality Disasters and How They Were Averted

The DataOps Way to Data Quality and Data Observability

DataKitchen

Data quality failures rarely announce themselves in advance. They accumulate quietly: a schema change nobody documented, a join that inflates row counts, a currency field that means something different in the new version of the source system. By the time the failure surfaces, the damage is already done.

The three case studies in this chapter are composites drawn from real patterns across financial services, healthcare, and e-commerce. The details are anonymized, but the failures, root causes, and fixes reflect patterns that repeat across organizations of every size. Read them as diagnostics, not horror stories.

Case Study 1: Financial Services -- The Quarter-End Reconciliation Failure

Business Context

A regional bank's data engineering team supports quarterly regulatory reporting across 12 business lines. The data warehouse feeds compliance reports submitted to federal regulators, and approximately 200 analysts, risk managers, and compliance officers depend on it daily. Errors in these reports carry direct regulatory consequences: restatements, fines, and formal corrective action plans.

Data Architecture

The team ran a Snowflake warehouse fed by 12 source systems: a core banking platform, three loan origination systems, two payment processors, and several ancillary systems for treasury, risk, and CRM. Data moved through nightly ETL jobs built on a combination of Fivetran connectors and custom Python scripts. The warehouse schema was managed by a small team of four engineers. Transformation logic lived in a mix of stored procedures and dbt models added over three years of incremental development.

The Failure

In the second week of a fiscal quarter, a vendor updated the core banking platform. Part of that update changed the primary key structure of the transactions table: what had previously been a single integer `transaction_id` was replaced by a composite key combining `transaction_id` and `branch_id`. The ETL job did not fail. It completed successfully every night for six weeks, silently joining on `transaction_id` alone.

Because `transaction_id` values were reused across branches after the schema change, the join produced duplicate records. Three million transactions were double-counted in the warehouse. The compliance team ran their quarter-end reports, reviewed the top-line figures, and submitted them to regulators.

The error was discovered two weeks after submission when a business analyst noticed that loan volume in one region appeared to have doubled. By that point, the regulatory report was already filed.

Total cost: $2 million in remediation, restatement work, and regulatory fines. The engineering team spent three weeks rebuilding the affected models, reprocessing six weeks of transaction history, and coordinating with the compliance team on a formal correction filing.

Root Cause Analysis

Three gaps made this failure possible.

No schema change alerting. The ETL jobs ran against source system schemas without any mechanism for detecting structural changes. When the vendor changed the primary key, nothing flagged it. The jobs continued running and producing output that looked plausible.

No referential integrity tests. The warehouse had no tests asserting that join keys were unique and not null in the source tables. A test that `transaction_id` was unique in the transactions source would have failed on day one of the new schema and surfaced the problem immediately.

No output shape monitoring. The team monitored job completion (did the ETL finish?) but not output shape (did the output look right?). Row counts, null rates, and value distributions in key fields were not monitored. A row count that doubled over a single night would have been an obvious signal.

Solution

The team implemented three changes after the incident.

Schema drift detection. Every source table's schema was fingerprinted at the start of each ETL run. If a column was added, removed, or had its data type changed, the job failed with an explicit error describing the change. The team received a PagerDuty alert within 15 minutes of any schema deviation.

Referential integrity tests. dbt tests were added asserting uniqueness and not-null constraints on every foreign key used in a join. These ran as part of the nightly pipeline before any transformation logic executed. A test failure blocked downstream model runs.

SLA alerts on ETL completion and output shape. The team added row count checks to the 15 highest-criticality tables. If row counts in any of those tables changed by more than 10% from the prior night's run, an alert fired. Separately, an Airflow SLA miss alert fired if any of the 12 source system loads had not completed by 6 AM.

Outcome

In the 18 months following the incident, the team recorded zero schema-drift incidents that reached the warehouse undetected. The schema change alerting caught four vendor-side schema modifications during that period, all of which were handled before any downstream impact.

Key Lessons

- A test that passes is only as good as the assumption it encodes. Testing that a job completed is not the same as testing that the job produced correct output.
- Schema changes in source systems are routine. Treat them as a certainty, not an edge case, and build detection into the pipeline rather than relying on vendors to communicate them proactively.
- The cost of a data quality failure scales with how long it goes undetected. Most of the $2M cost came from the six-week window during which bad data was accumulating unnoticed, not from the original failure itself.

Case Study 2: Healthcare -- The Patient Count That Was Wrong for Six Months

Business Context

A hospital network with eight facilities tracks patient outcomes to support quality improvement programs and accreditation requirements. The data team produces monthly dashboards reviewed by clinical leadership, operations directors, and the executive team. These dashboards inform staffing decisions, resource allocation, and public reporting on quality metrics. The numbers need to be right, and the people reading them need to be able to trust them.

Data Architecture

Patient data was exported daily from the network's electronic health record (EHR) system into an S3-based data lake. A dbt project containing approximately 140 models transformed that raw export through staging, intermediate, and mart layers. The final mart tables fed a Tableau dashboard consumed by

roughly 60 users. The data team consisted of five engineers and one analytics engineer who owned the dbt project.

The Failure

A senior analytics engineer refactored a mart model that calculated unique patient counts by care setting. The refactored version introduced a join to a new diagnosis codes table to enrich patient records with primary diagnosis information. The join was a one-to-many relationship: a single patient encounter could have multiple diagnosis codes. The analytics engineer reviewed the SQL, confirmed the logic was correct for the new columns being added, and merged the change.

The join inflated patient counts by approximately 30%. A patient with three diagnosis codes now appeared three times in the count. The error was not detected for six months.

During those six months, the executive dashboard showed inflated patient volumes. Staffing models were recalibrated based on the higher numbers. One facility initiated a hiring process for additional nursing staff based on the apparent increase in patient load. The error was eventually discovered by a nurse manager who cross-referenced the dashboard against a manual census count during a weekly operations meeting.

Root Cause Analysis

Two gaps enabled this failure.

Tests existed at the staging layer but not at the mart layer. The dbt project had solid test coverage at the staging layer: uniqueness, not-null, and accepted-values tests ran on every staging model. But the mart layer had no tests on output shape. Row counts, uniqueness constraints on the patient identifier, and aggregate value ranges were all untested.

Model changes were reviewed for logic correctness but not for output shape. The PR review process asked: does this SQL do what the author says it does? It did not ask: does the output of this model have the same shape as before? A row count comparison between the old and new model would have immediately revealed the fan-out.

Solution

The team made three changes to their development and testing process.

Row count tests at every mart model. Every mart model now has a test asserting that the row count is not null and greater than zero. For models with a known stable population, such as patient encounters in the trailing 30 days, an additional test asserts that the row count does not deviate by more than 5% from the prior run.

A mandatory output shape check in CI. The team added a step to their CI pipeline that runs a comparison query against the last production snapshot of any mart model modified in a PR. The comparison reports row count delta, null rate changes in key columns, and min/max changes in numeric fields. This output is posted automatically as a PR comment. Reviewers are required to acknowledge it before merging.

A dedicated data quality reviewer in the PR process. One team member rotates into a data quality reviewer role each sprint. Their job is not to review SQL logic -- that is the author's and a second engineer's responsibility. Their job is to look specifically at output shape, test coverage, and downstream impact. This reviewer must explicitly approve any change to a mart model.

Outcome

The fan-out error class was eliminated. In the six months following the changes, two pull requests were opened that would have introduced similar fan-out joins. Both were caught by the CI output shape check before merge. Neither reached production.

The team also retroactively added row count tests to 12 other mart models that had previously had no tests at the mart layer.

Key Lessons

- Tests at the staging layer do not protect you from errors introduced at the mart layer. Test coverage must match the layer where errors can occur, and errors can occur anywhere in the transformation chain.
- Most fan-out bugs look correct when you read the SQL. The signal is in the output shape, not the logic. A row count comparison is faster and more reliable than a code review for catching this error class.
- Six months is a long time for bad data to influence decisions. The earlier in the development process you can catch a transformation error, the lower the cost. CI is better than production monitoring; production monitoring is better than a user report.

Case Study 3: E-Commerce -- The Promotion That Cost $400,000

Business Context

A mid-size e-commerce retailer with operations in seven countries runs 30 to 40 promotional campaigns per quarter. The promotions team uses a real-time dashboard to monitor discount amounts and order values during live promotions. Speed matters: a promotion that is misconfigured needs to be caught and corrected within minutes, not hours. The financial exposure during a 24-hour flash sale can be substantial.

Data Architecture

Order events streamed from the order management system into a Kafka topic, consumed by a Flink job that computed rolling aggregates and wrote results to a Redis cache backing the real-time dashboard. A separate batch pipeline processed the same Kafka topic into a Redshift warehouse for post-promotion analysis. The streaming pipeline had been running reliably for 18 months before the incident.

The Failure

The order management system team released a schema update that changed the `discount_amount` field in the order event payload. Previously, the field had always contained values in USD regardless of the customer's country. In the new version, the field contained values in the customer's local currency to support a new regional pricing feature.

The Flink job that consumed the order events was not updated. It continued reading `discount_amount` and treating it as USD. For a customer in Germany paying in euros, the discount amount was now being interpreted as dollars, which happened to produce a value that looked plausible on a dollar basis given normal exchange rate ranges.

A 24-hour flash promotion launched the following week. The discount rate calculations used by the promotions team to set the promotion parameters were based on historical USD discount amounts. When the promotion ran, discounts for non-US customers were applied at rates far higher than intended because the underlying field now contained local currency values that were being treated as USD in the calculation logic.

The error ran for 11 hours before a promotions manager noticed that order values in the European markets were unexpectedly low. By the time the promotion was corrected and the pipeline was updated, the company had over-discounted approximately $400,000 in orders.

Root Cause Analysis

Two gaps made this possible.

No data contract between the order system and the analytics pipeline. The streaming pipeline was built against an informal understanding of the order event schema. There was no written contract specifying field names, types, units, and valid ranges. When the order system team changed the semantics of `discount_amount`, there was no mechanism to flag the change as a breaking one for downstream consumers.

No range tests on the discount amount field. The pipeline had no validation on the values flowing through it. A test asserting that discount amounts must represent between 0% and 60% of the order value would have immediately flagged the anomaly when European orders started showing effective discount percentages well above 100% due to the currency mismatch.

Solution

The team implemented three changes.

A formal data contract for the order events stream. The contract was written collaboratively by the order system team and the data team. It specified field names, data types, units (explicitly: USD), valid ranges, and a change notification process. Any change to a field covered by the contract required a 14-day notice period and a version increment in the schema registry. Downstream consumers were listed in the contract and notified automatically when a change was proposed.

Business rule tests on discount amounts. The Flink job was updated to validate that discount amounts, expressed as a percentage of order value, fell between 0% and 60%. Records that failed this check were routed to a dead-letter queue and triggered an alert. The threshold was set conservatively enough to catch anomalies while loose enough to accommodate legitimate high-value promotions.

A P1 alert on median order value. The real-time dashboard was augmented with a monitoring rule: if the median order value in any country dropped by more than 20% compared to the trailing 7-day average for the same time window, a P1 alert fired to the on-call data engineer and the promotions team. This alert was designed to catch the business impact of a data error within 15 minutes, even if the root cause had not yet been identified.

Outcome

Six months after the changes, the order system team proposed another currency-related schema change. The data contract flagged it as a breaking change. The data team was notified 14 days in advance and updated the pipeline before the change reached production. The Flink job's business rule test was also updated as part of that preparation. No orders were affected.

The dead-letter queue added as part of the business rule tests caught two other data quality issues in the following year: one involving a test order incorrectly routed to the production stream, and one involving a rounding error in the order management system that produced negative discount amounts.

Key Lessons

- Informal agreements about data schemas are not contracts. A field that has always contained USD values is not guaranteed to always contain USD values unless that guarantee is written down, versioned, and enforced with a change process. Document the contract, not the assumption.
- Real-time pipelines need real-time monitoring. A batch report reviewed the next morning would not have helped here. If the data feeds a dashboard that drives decisions during a live promotion, the monitoring needs to operate on the same time horizon as the decisions being made.
- Range tests on business-critical fields are cheap to write and expensive not to have. A five-minute test definition would have caught the currency mismatch within seconds of the first affected order. The cost of skipping it was $400,000.

When Not to Test: Setting Reasonable Bounds on Data Quality Effort

IMPLEMENTATION AND CASE STUDIES

When Not to Test: Setting Reasonable Bounds on Data Quality Effort

The DataOps Way to Data Quality and Data Observability

DataKitchen

Most data quality guidance tells you to test everything. This chapter tells you when not to.

That is not a concession to laziness. It is an acknowledgment that testing has opportunity costs. Every hour your team spends writing low-value tests is an hour not spent on high-value ones. A test suite that covers everything equally is often worse than one that covers the right things thoroughly.

Data engineers and data team leads operate under real constraints: limited time, competing priorities, and stakeholders who want features as well as reliability. The goal is not comprehensive coverage for its own sake. The goal is to catch the errors that matter before they cause damage. Getting there requires being honest about where your testing effort actually belongs.

The Opportunity Cost Argument

Testing is not free. Writing a test takes time. Running a test infrastructure takes money and compute. Maintaining tests as schemas and logic change takes ongoing engineering effort. When a test fails, someone has to investigate, triage, and decide whether the failure represents a real problem or a test that no longer reflects reality.

A team of three data engineers supporting 80 tables cannot write and maintain comprehensive test suites for all 80 tables and also build new pipelines, respond to incidents, and handle stakeholder requests. Pretending otherwise does not make it true.

The practical question is not "should we test this table?" It is "given our constraints, which tables should we test, how deeply, and what should we skip or defer?"

Answering that question honestly requires a framework. Here is one that works.

A Framework for Prioritizing Test Coverage

Not every table deserves the same level of attention. Use the following factors to triage.

Who uses the data and what decisions depend on it?

A table that feeds the monthly revenue report reviewed by the CFO deserves comprehensive tests. A table used by one analyst for occasional exploratory work probably does not. The closer data is to a high-stakes decision, the more important it is to test it.

Ask: if this table had a silent error for 30 days, what would the impact be? If the answer is "a regulatory fine" or "a $400,000 miscalculated promotion," test it thoroughly. If the answer is "one analyst's exploration is slightly off," a lighter approach is defensible.

How often is the data used?

Frequency amplifies both the value of testing and the cost of errors. A table queried hundreds of times per day by automated systems needs tests that catch failures fast. A table accessed once a quarter for an annual review can tolerate a longer feedback loop, including manual spot-checking before use.

How easy is it to catch errors downstream?

Some errors are self-correcting or visible to users immediately. If an analyst runs a query and sees null values where there should be numbers, they will notice and report it. If a dashboard shows an obvious spike, someone will flag it. For these cases, you can afford lighter automated testing because the human layer provides a backstop.

Other errors are invisible until they cause damage. Silent double-counting, wrong join keys, incorrect currency conversions -- these can persist for months without triggering an obvious signal. These cases need automated tests, because no human will catch them in time.

What is the cost of an error?

Tie your testing investment to the cost of getting it wrong. This is a business question, not a technical one. Engage your stakeholders: what is the downstream consequence of bad data in this table? The answer should drive how much you invest in testing it.

Data criticality	User base	Error visibility	Recommended approach
High (financial, compliance, patient care)	Broad or automated	Low (silent failures)	Comprehensive: uniqueness, not-null, range, referential integrity, row count
High	Narrow (single team)	Medium	Core tests plus manual review before use

Data criticality	User base	Error visibility	Recommended approach
Medium (operational reporting)	Broad	Medium	Smoke tests: not-null on key fields, row count bounds
Low (exploratory, ad hoc)	Narrow	High (users notice)	Spot-check before use, no automated tests required

When to Automate vs. When to Spot-Check Manually

Automated tests make sense when a table is queried regularly, feeds automated systems or scheduled reports, or is used to make decisions where a human review before use is impractical.

Manual spot-checking makes sense when the table is used infrequently, accessed by a small number of people who understand its limitations, and where a quick review before use is both feasible and sufficient.

Consider the difference between these two tables:

A table used to calculate monthly commissions for 200 salespeople absolutely needs automated tests. The calculation runs on a schedule. Nobody reviews the input data before the commissions are generated. If the data is wrong, 200 people are paid incorrectly, and the error may not surface until someone notices a discrepancy in their paycheck weeks later. Automate the tests.

A table used by one analyst once a quarter to look at regional trends for a strategy presentation probably does not need automated tests. The analyst queries it, looks at the results, and uses their judgment. They can spot-check key figures against known benchmarks before they present. The cost of an error is a slightly off presentation that gets corrected in the meeting. Spot-checking is sufficient.

The principle is simple: automate tests where a human review before use is not happening and not realistic. Skip automation where a capable human is already reviewing the data before it influences a decision.

When to Invest in Monitoring vs. When to Rely on User Reports

User reports are a legitimate signal. When users tell you the data looks wrong, that is valuable feedback. The question is whether you can afford to wait for it.

A practical threshold: if users report errors in a given table more than once per quarter, invest in monitoring. That frequency suggests either that errors are common or that the consequences of errors are visible enough that users notice quickly. Either way, you are better off catching errors before users do.

If users never report errors on a table, pause before concluding that the data is clean. There are two equally plausible explanations: the data is genuinely reliable, or nobody is actually using the table and errors go unnoticed. Figure out which. Pull the query logs. If nobody is running queries against that table, deprioritize its test coverage. If it is being queried regularly with no complaints, your monitoring investment is probably adequate.

Invest in proactive monitoring (freshness checks, row count bounds, anomaly detection) for tables that meet any of these criteria:

- Used in automated pipelines where no human reviews the output
- High business impact if wrong
- Known history of upstream schema changes or data quality issues
- Used by non-technical stakeholders who would not recognize a data error on their own

Save your monitoring budget from tables that are infrequently used, have a small audience of technically capable users, and have a strong track record of reliability.

How to Know When You Have Enough Testing

Here is a useful signal, imperfect but real: you have enough testing when you stop being surprised by failures you did not detect.

When a data quality issue surfaces and your first reaction is "we should have caught that," that is a gap in your testing. When a failure surfaces and your team already has an alert for it, even if the alert fired too late or at the wrong threshold, you are in the right territory. When a failure surfaces and the test that should have caught it exists and is running, you have reached a reasonable level of coverage for that domain.

You will not reach a state where all failures are anticipated. That is not the goal. The goal is that the failures you care most about -- the ones that affect high-stakes decisions, that are invisible to users, that could persist for months -- are covered by tests that would catch them quickly.

A few practical checks:

Look at your last five data quality incidents. Were any of them caught by automated tests? If zero out of five were caught by tests, you have a coverage problem. If four out of five were caught, you are in reasonable shape.

Look at which tables generated the most user-reported errors in the past year. Those tables need more testing, not less.

Look at which tests have never failed. Some of those are good tests on stable data. Others may be testing conditions that can never fail given your current data patterns. Review them annually and prune the ones that provide no signal.

Testing Debt and How to Pay It Down

Testing debt is the accumulated gap between the testing you have and the testing you should have. Most teams carry some. The question is how to reduce it without halting all other work.

The answer is not a testing sprint. A dedicated sprint produces a burst of coverage that degrades quickly as the codebase evolves without tests being added alongside new work. Sprint-based testing debt paydown produces short-term improvement and long-term resentment.

A more durable approach has two parts.

First, adopt a policy that new pipelines and new tables ship with a minimum level of testing. Define what that minimum is -- at least not-null tests on primary keys, a row count check, and a freshness check on ingestion time -- and hold it as a standard for all new work going forward. This stops the debt from growing.

Second, attach debt paydown to incidents. When a data quality failure occurs that was not caught by a test, the post-incident action item includes writing the test that would have caught it. This approach ensures that testing debt paydown is driven by actual risk rather than theoretical completeness.

For existing high-priority tables that lack coverage, a modest recurring allocation -- a few hours per sprint assigned to one engineer -- will move the needle over time without blocking feature work. Prioritize by the framework above: highest business impact, lowest error visibility, broadest user base first.

The goal is not to pay down all testing debt. The goal is to ensure that your highest-risk tables have adequate coverage and that the debt you carry is concentrated in low-risk areas where you can afford to carry it.

What You Are Allowed to Skip

To make this concrete: here is a list of cases where skipping automated tests is a defensible choice.

- Exploratory or prototype pipelines that are not yet in production
- Tables used by a single technically capable analyst who reviews data before use
- Historical archive tables that have not been updated in over a year
- Staging tables that are fully rebuilt on every pipeline run and tested at the source layer
- Tables with extremely small row counts (under 1,000 rows) where a visual inspection takes 30 seconds

Here is a list of cases where skipping automated tests is not defensible.

- Any table that feeds a scheduled report delivered to non-technical stakeholders
- Any table used in a financial calculation, compensation model, or regulatory submission
- Any table that is the source of truth for a key business metric
- Any table in a domain that has had a data quality incident in the past 12 months
- Any table consumed by an automated system with no human review in the pipeline

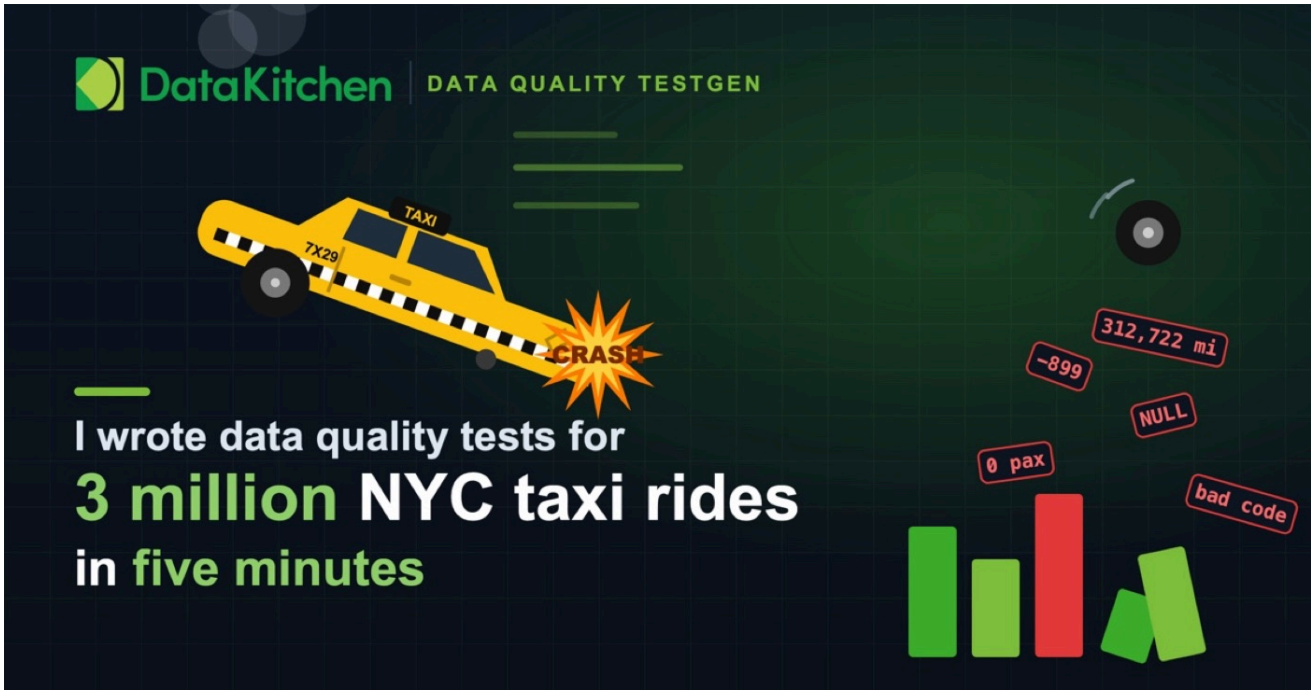
The lists above are starting points, not rules. Apply the framework to your specific situation. The point is that the decision to skip testing should be a conscious choice made for documented reasons, not a default that happens because nobody got around to it.

Knowing when not to test is as important as knowing how to test. Both are skills that mature data teams develop over time.

IMPLEMENTATION AND CASE STUDIES

I Wrote Data Quality Tests for 3 Million NYC Taxi Rides in Five Minutes

Read the original article at: https://datakitchen.io/blog/data-quality-tests-3-million-nyc-taxi-rides-five-minutes/



You get handed a new dataset. Nobody tells you what's wrong with it, because nobody knows. Somewhere in there are negative fares, trips to the moon, and passengers who don't exist. You find out when a dashboard shows a $5,000 cab ride and a VP asks about it.

The old fix is to open the data, eyeball a few thousand rows, write some tests by hand, and hope you guessed the failure modes. That takes an afternoon. Then you do it again for the next table.

I did it for the NYC yellow taxi dataset in about five minutes. Then I did the whole thing again on all three million rows to prove the small run wasn't lying. Here's exactly how, prompts included, so you can recreate it.

The tools

Three pieces. A local Postgres database. TestGen, which profiles the data and runs the tests. And an AI tool talking to TestGen through an MCP server.

TestGen Enterprise ships an MCP server. Mine runs on port 8530, right next to the TestGen app. MCP is the part that matters here. It's the wire that lets your AI tool call TestGen directly: create a table group, run a profile, read the hygiene issues, write a test, run the suite. There are 96 of those tools, and the TestGen

MCP cheat sheet puts them on one page. No clicking through a UI. You describe what you want and the model drives the product.

I used Claude Code. Any AI client that speaks MCP works the same way. You point the client at the TestGen MCP server once, and from then on the model has hands.

Everything ran in Docker on my laptop. TestGen in one container, Postgres in another, on the same Docker network so TestGen reaches the database by the hostname `postgres`. If you already run TestGen locally, you already have this.

Component	What it is	Image	Port	Role
TestGen	Data quality engine and UI	`datakitchen/dataops-testgen-enterprise:v5`	8501	Profiles data, generates and runs tests
TestGen MCP	MCP server inside TestGen	same container	8530	Lets the AI tool drive TestGen
Postgres	Target database	`postgres:14.1-alpine`	5432	Holds the taxi table
Claude Code	AI client	your machine	n/a	Reads the data, writes and runs the tests

The connection TestGen used, so you can match the shape:

Setting	Value
Type	PostgreSQL
Host	`postgres` (Docker network alias)
Port	5432
Database	`demo_db`
Schema	`nyc_taxi`
User	`admin`
Auth	password

Get the data

The New York City Taxi and Limousine Commission publishes every trip on the TLC trip record data page. Yellow cabs, green cabs, for-hire vehicles, going back years. It's real, it's messy, and it's free. Public government data makes a good punching bag, and we've done this before: TestGen found 18 quality issues in Boston building-permit data in a few minutes, install time included.

I didn't grab it. I asked for it, and the model went and got it. January 2024 yellow taxi trips, one Parquet file, 48 MB, 2,964,624 rows.

Dataset fact	Value
Source	NYC Taxi and Limousine Commission (TLC) trip records
File	`yellow_tripdata_2024-01.parquet`
URL	`https://d37ci6vzurychx.cloudfront.net/trip-data/yellow_tripdata_2024-01.parquet`
Format	Apache Parquet
Size	48 MB
Rows	2,964,624
Columns	19
Cost	free

The 19 columns cover the trip: vendor, pickup and dropoff timestamps, passenger count, trip distance, rate code, pickup and dropoff location IDs, payment type, and the money (`fare_amount` , `extra` , `mta_tax` , `tip_amount` , `tolls_amount` , `improvement_surcharge` , `total_amount` , `congestion_surcharge` , `Airport_fee`).

There is no shell command to copy here, and that's the point. Downloading the file, creating the schema, and loading the rows were things I asked for in English. The model worked out the URL from the TLC page, pulled the Parquet, and loaded a random 300,000-row sample first so the first pass would be quick. The full month came later, on request.

Pick any month you like from that same page. You describe which one. You don't write the fetch.

Pass one: the sample

Here's the entire prompt I gave the AI:

1. Go get the New York City taxi data from the website.
2. Load it into TestGen, profile it, and find hygiene issues.
3. Generate a test sheet and run the tests.
4. Return to me what the hygiene issues are and any test failures.

That's it. No SQL, no config, no clicking. The model created a TestGen table group pointing at the `nyc_taxi` schema, kicked off a profiling run, and read the results back.

Profiling 300,000 rows across 19 columns took 10 seconds. TestGen came back with one hygiene issue: the table has no dates within the last year. True. The data is from January 2024 and the most recent trip ends February 1. Timeliness flag, correct.

Hygiene issue	Columns	Impact	Quality dimension	Priority
No table dates within one year	`tpep_pickup_datetime`, `tpep_dropoff_datetime`	Regularity	Timeliness	Possible

Then TestGen generated the test sheet. Sixty-two tests, written from the profile, one glance at the data. The model ran them.

Test category	Example test types	Count
Value range	Minimum Value	11
Completeness	Required Entry, Percent Missing	24
Distribution shift	Average Shift, New Shift, Variability Increase/Decrease	16
Dates	Past Dates, Minimum Date, Date Count	7
Uniqueness	Percent Unique	3
Table-level	Duplicate Rows	1
Total		**62**

Sixty-two passed. Zero failed.

That number is a trap. Here is why. Auto-generated tests calibrate to the data in front of them. The minimum-value test for `fare_amount` set its floor at the lowest fare it saw, which was negative $367. So a negative fare passed the test, because the test learned that negative fares were normal here. The baseline sheet tells you the data is internally consistent. It does not tell you the data is correct.

Pass two: the part that catches real bugs

Correct is a business question. A fare can't be negative. A trip needs at least one passenger. A cab can't drive 97,000 miles in a January afternoon. TestGen can't know your rules until you tell it, and the rules that matter most are usually locked in someone's head rather than written down. This is where the AI earns its keep. The same pairing already runs unattended: an agent fixed 14 data quality failures overnight in one of our tests, inside a design that keeps a human in control.

The model had already read the profile. It knew the minimum fare was negative, that thousands of trips carried zero passengers, that trip distance topped out in five figures. So it wrote seven business-rule tests, in SQL, from understanding the data. I didn't dictate them. TestGen calls these Custom Tests: any rule you can write as a query. The query returns the bad rows. If it returns anything, the test fails.

Here are the seven:

```
-- fare_amount must not be negative
SELECT tpep_pickup_datetime, fare_amount, total_amount
FROM {DATA_SCHEMA}.yellow_tripdata WHERE fare_amount < 0;

-- total_amount must not be negative
SELECT tpep_pickup_datetime, fare_amount, total_amount
FROM {DATA_SCHEMA}.yellow_tripdata WHERE total_amount < 0;

-- passenger_count must be >= 1 (no zero or null passenger trips)
SELECT tpep_pickup_datetime, passenger_count
FROM {DATA_SCHEMA}.yellow_tripdata WHERE passenger_count IS NULL OR passenger_count < 1;

-- trip_distance must be > 0 and <= 1000 miles (a plausible NYC trip)
SELECT tpep_pickup_datetime, trip_distance
FROM {DATA_SCHEMA}.yellow_tripdata WHERE trip_distance <= 0 OR trip_distance > 1000;

-- dropoff must be at or after pickup
SELECT tpep_pickup_datetime, tpep_dropoff_datetime
FROM {DATA_SCHEMA}.yellow_tripdata WHERE tpep_dropoff_datetime < tpep_pickup_datetime;

-- pickup must fall inside the dataset's month
SELECT tpep_pickup_datetime
FROM {DATA_SCHEMA}.yellow_tripdata
WHERE tpep_pickup_datetime < '2024-01-01' OR tpep_pickup_datetime >= '2024-02-01';

-- payment_type must be a valid TLC code (1-6)
SELECT tpep_pickup_datetime, payment_type
FROM {DATA_SCHEMA}.yellow_tripdata WHERE payment_type NOT IN (1,2,3,4,5,6);
```

Ran the suite again. Now the sheet had teeth. All seven business rules failed on the 300,000-row sample:

Business rule	Failing rows
passenger_count >= 1	17,477
valid payment_type (1-6)	14,252
trip_distance in (0, 1000] miles	6,184
fare_amount >= 0	3,763
total_amount >= 0	3,573
dropoff >= pickup	9
pickup inside January 2024	2

Six percent of trips have no passengers. One in twenty has a payment code that isn’t in the spec. This is a clean, official, government dataset, and it’s full of holes. Yours is worse.

Now do it for real: all three million rows

A sample is an argument, not a proof. So I asked for the full month, all 2,964,624 rows, and for the whole thing to run again. Nine words:

load the full month, reprofile and re-hygiene

Then:

Profiling three million rows took one minute and 48 seconds. Same single hygiene issue, table still stale, no surprise. The extremes got uglier, which is what more data does. The minimum fare fell from negative $367 to negative $899. Trip distance topped out at 312,722 miles. That's a cab ride to the moon and rather more than halfway back.

The full test run finished in six seconds: 69 tests, 11 failed, 2 warnings. Here is what came back, unedited:

Full-Month Test Run — 2,964,624 rows

69 tests: 56 passed, 11 failed, 2 warnings (6s). Two distinct kinds of failures appeared:

A. Business-rule tests (the 7 I added) — all fail, ~10× larger counts

Rule	Sample (300k)	Full month (2.96M)	×scale
`passenger_count ≥ 1`	17,477	171,627	9.8×
valid `payment_type` (1–6)	14,252	140,162	9.8×
`trip_distance` > 0 and ≤ 1000 mi	6,184	60,394	9.8×
`fare_amount ≥ 0`	3,763	37,448	10.0×
`total_amount ≥ 0`	3,573	35,504	9.9×
dropoff ≥ pickup	9	56	—
pickup within Jan 2024	2	18	—

Clean ~10× scaling confirms the 300k sample was representative of the full month.

B. Auto-generated baseline tests that now breach (new — didn't fail on the sample)

These are the auto-generated sheet catching **distribution drift** — exactly what they're for. The full month contains a handful of records more extreme than anything in the 300k baseline:

Test	Column	Records past baseline floor
Minimum Value	`fare_amount`	22 (min dropped to −$899 vs sample −$367)
Minimum Value	`total_amount`	23
Minimum Value	`tip_amount`	25
Minimum Value	`tolls_amount`	34

Plus **2 warnings** — `Percent Unique` on both timestamp columns measured ~96% unique against a 50% baseline threshold (soft anomaly, not a hard fail).

Bottom line across the whole exercise

- **Hygiene issues:** 1 (table staleness) — unchanged by scale.
- **Test failures:** 11 fails + 2 warnings on the full month.
  - 7 business-rule failures (negative money, invalid passenger/payment codes, impossible distances, out-of-range/reversed timestamps) — scaled ~10× from the sample.
  - 4 new baseline Minimum Value failures where the full month's negative tail (fares to −$899) drifted past what the sample captured.

The seven business rules failed again, and the counts scaled almost perfectly by 10x, which tells you the sample was honest:

Business rule	Sample (300K)	Full month (3M)
passenger_count >= 1	17,477	171,627
valid payment_type (1-6)	14,252	140,162
trip_distance in (0, 1000] miles	6,184	60,394
fare_amount >= 0	3,763	37,448
total_amount >= 0	3,573	35,504

Business rule	Sample (300K)	Full month (3M)
dropoff >= pickup	9	56
pickup inside January 2024	2	18

Four more tests failed on the full month that passed on the sample. Those were auto-generated minimum-value tests. The full month held a handful of records more extreme than anything in the sample, so they slipped past the baseline floor.

Baseline test	Column	Records past the floor
Minimum Value	`fare_amount`	22
Minimum Value	`total_amount`	23
Minimum Value	`tip_amount`	25
Minimum Value	`tolls_amount`	34

That's the auto-generated sheet doing its other job: catching drift when new data doesn't look like old data. Both kinds of test have a place. The business rules catch what's wrong today. The baseline tests catch what changed since yesterday, which is the same idea behind continuous table monitoring in TestGen.

Two soft warnings rounded out the run. Percent-unique checks on both timestamp columns came in near 96 percent against a 50 percent baseline. Not a hard fail, just TestGen noting the shape moved.

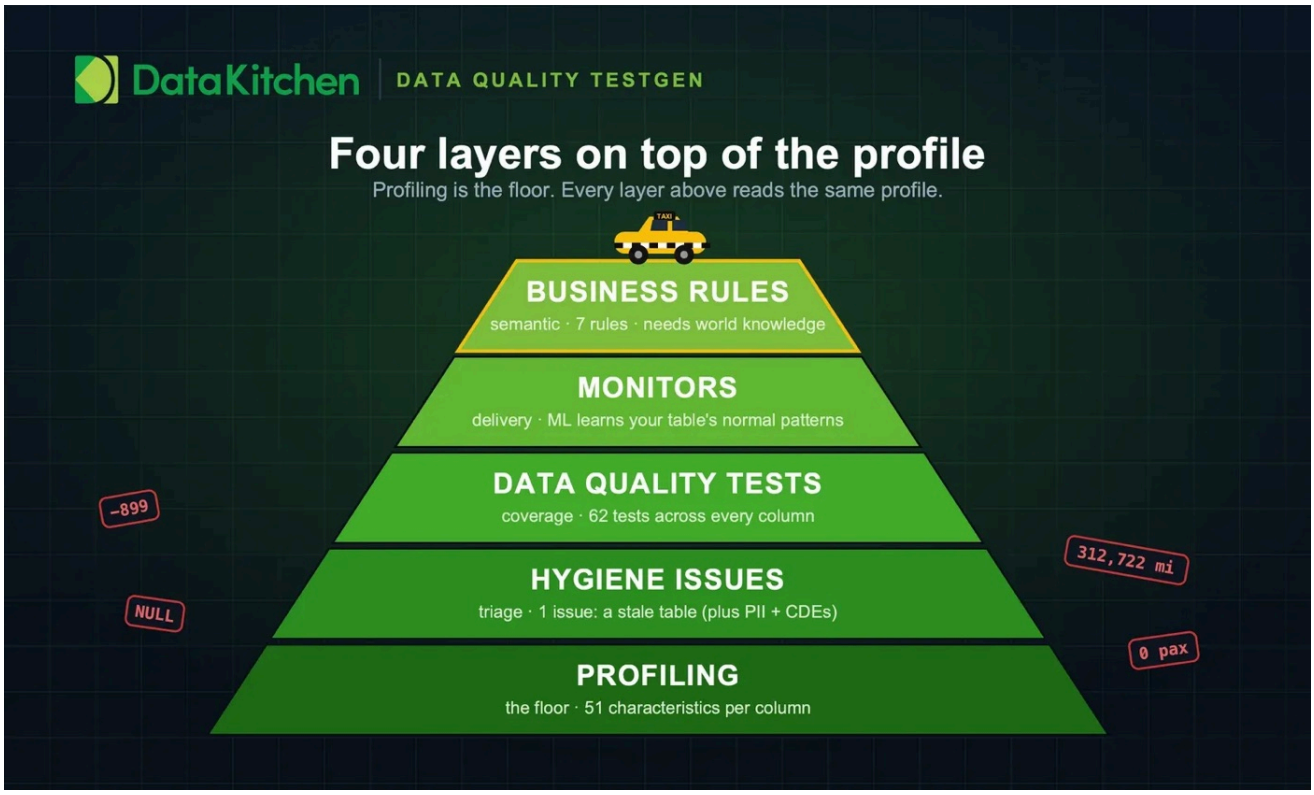
Run	Tests	Passed	Failed	Warnings
Sample, baseline sheet only (62 tests)	62	62	0	0
Sample, plus 7 business rules (69 tests)	69	62	7	0
Full month, all 69 tests	69	56	11	2

Four layers on top of the profile

Profiling is the floor. Everything else stands on it. TestGen records 51 characteristics for every column, and nothing above it works without that. Every layer that follows reads from the same profile, which is why none of them needed a threshold typed in by hand.

Hygiene issues are the first layer, the triage pass. The taxi data threw exactly one, a stale table. The same pass finds PII sitting in a column nobody meant to expose, and flags the critical data elements worth guarding.

Data quality tests are the second layer, coverage. All 62 of them here, every column in every table, watching for the anomalies profiling can measure: a distribution that shifts, a column that starts arriving empty.



Monitors are the third layer, delivery. They keep running long after an exercise like this ends, and they use machine learning to learn your tables' normal patterns. They catch the schema change and the volume drop on the morning a vendor ships you half a file.

The seven business rules are the fourth layer, the semantic one. They needed a model that knows something about the world. Nothing in a profile says a cab can't drive 312,722 miles. You have to know what a taxi is. Claude knew that on its own, because taxis are public knowledge. Your rules usually aren't. Which product codes were retired last quarter, or the revenue floor a contract promises, lives in someone's head at your company. So a person who knows the business writes those rules, or hands the AI enough context to write them. The model brings the general knowledge. You bring the part only your organization has.

The clock

This is the whole point, so here's the timeline with real numbers. Everything below ran on a MacBook with an M1 chip, in Docker, on battery. No cluster, no warehouse credits, no scaling anything up.

Step	Time
Download the Parquet file	seconds
Load the 300K sample	seconds
Profile the sample	10 seconds

Step	Time
Generate 62 tests and run them	seconds
Write 7 business-rule tests	seconds (the AI wrote them)
Rerun the suite	instant
Load the full 3M rows	about a minute
Profile 3M rows	1 minute 48 seconds
Run 69 tests on 3M rows	6 seconds

Start to finish, understanding a dataset you've never seen and standing up real data quality tests on it is a five-minute job. Not a five-story-point ticket. Five minutes. The slowest thing in the whole exercise was Postgres reading three million rows off disk, and even that ran under two minutes.

The tests didn't take an afternoon because a human didn't write them one at a time. The AI read the profile, understood what taxi data should look like, and wrote the rules. TestGen ran them and told me the counts. My job was to say what "correct" means and to check the answer against the raw table, which tied out to the row.

IMPLEMENTATION AND CASE STUDIES

Data Quality's Most Wanted: The 2026 Case Files

Read the original article at: https://datakitchen.io/blog/data-qualitys-most-wanted-the-2026-case-files/

Your dashboards look fine. Your pipelines ran green overnight. Your team shipped the release on schedule. And somewhere inside your data, a crime is already in progress.

Four data quality disasters reached the public record in 2026, and each followed the same pattern. Two numbers that had to agree were never compared. The reports got signed. The invoices got sent. The gates opened and closed. By the time anyone noticed, the damage was measured in a shuttered hospital, 22,000 repayment demands, and a city's books off by $1.45 million.

Bad data is not a nuisance you clean up on Friday afternoon. It closes emergency rooms. It sends repayment demands to students who did nothing wrong. It leaves a mayor looking at books that lie by seven figures. When you skip data testing, you are not saving time. You are handing the business a loaded gun and hoping nobody pulls the trigger.

What makes these cases infuriating is that none of them were exotic. No zero-day exploit. No unprecedented technical challenge. Every one would have been caught by the kind of basic data testing that has existed for twenty years, and that you can now run for free. Here are the case files.

Case File 01: The Phantom Billing (West Suburban Medical Center)

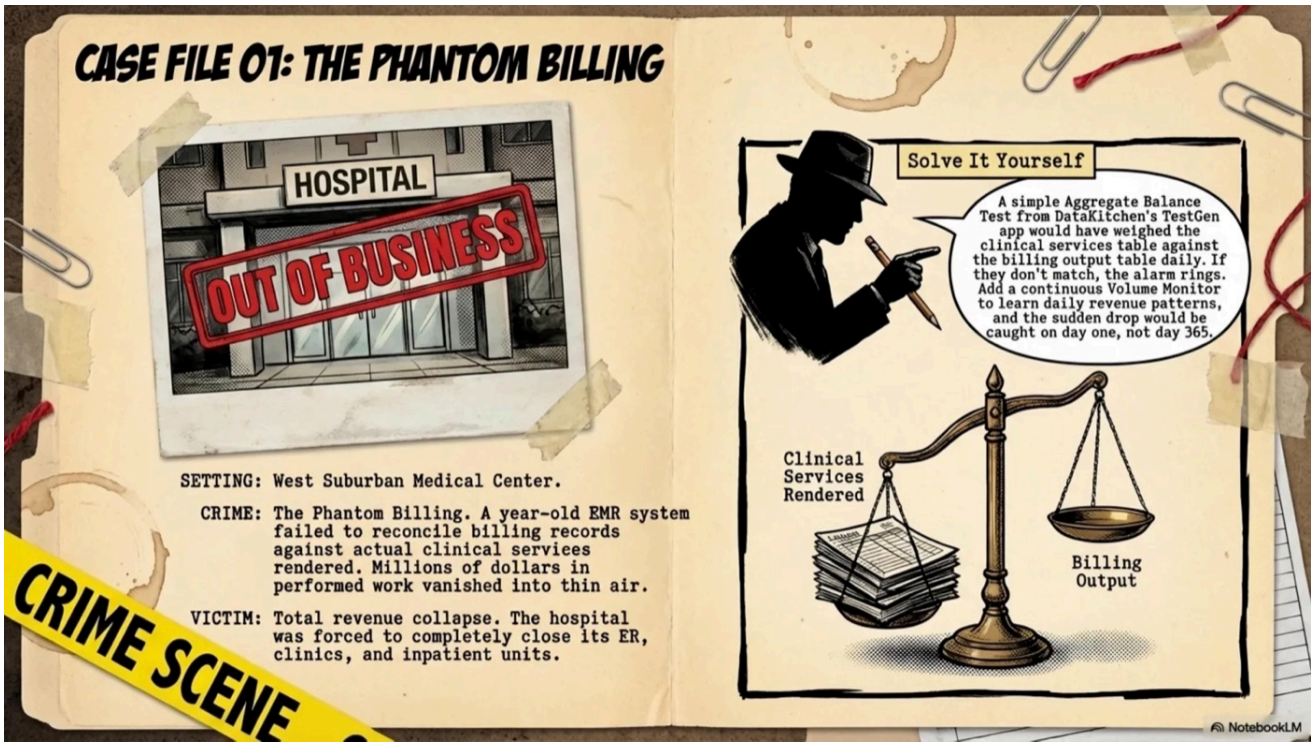
Setting: Oak Park, Illinois.

In early 2025, West Suburban Medical Center moved to a new electronic medical record system. According to hospital leadership, it never billed correctly. Doctors saw patients, nurses ran tests, and the surgery suites stayed busy. At least half of that work never made it onto a bill, and the hospital spent the year running on roughly 10 to 15 percent of its normal income, as reported by Crain's Chicago Business.

On 25 March 2026 the hospital paused all patient care: emergency room, inpatient units, clinics. It ran out of cash to make payroll. A community lost its hospital for months while owner Resilience Healthcare put roughly 120 people on the job of processing more than 120,000 backlogged claims.

One caveat that matters, and that the fix has to account for. The software vendor, Altera Digital Health, publicly disputed that its system caused the closure. That is the argument you get when nobody can point at a test result: the hospital blames the software, the software company blames the implementation, and neither side has a dated record of the two numbers failing to match.

That record is cheap to keep. Count the clinical services delivered. Count the billing output. Run an aggregate balance test between them every night, and a divergence rings an alarm on day one instead of day 365. Whoever turns out to be at fault, the argument lasts a day rather than a year.



Case File 02: The Weekend Heist (Office for Students and the Student Loans Company)

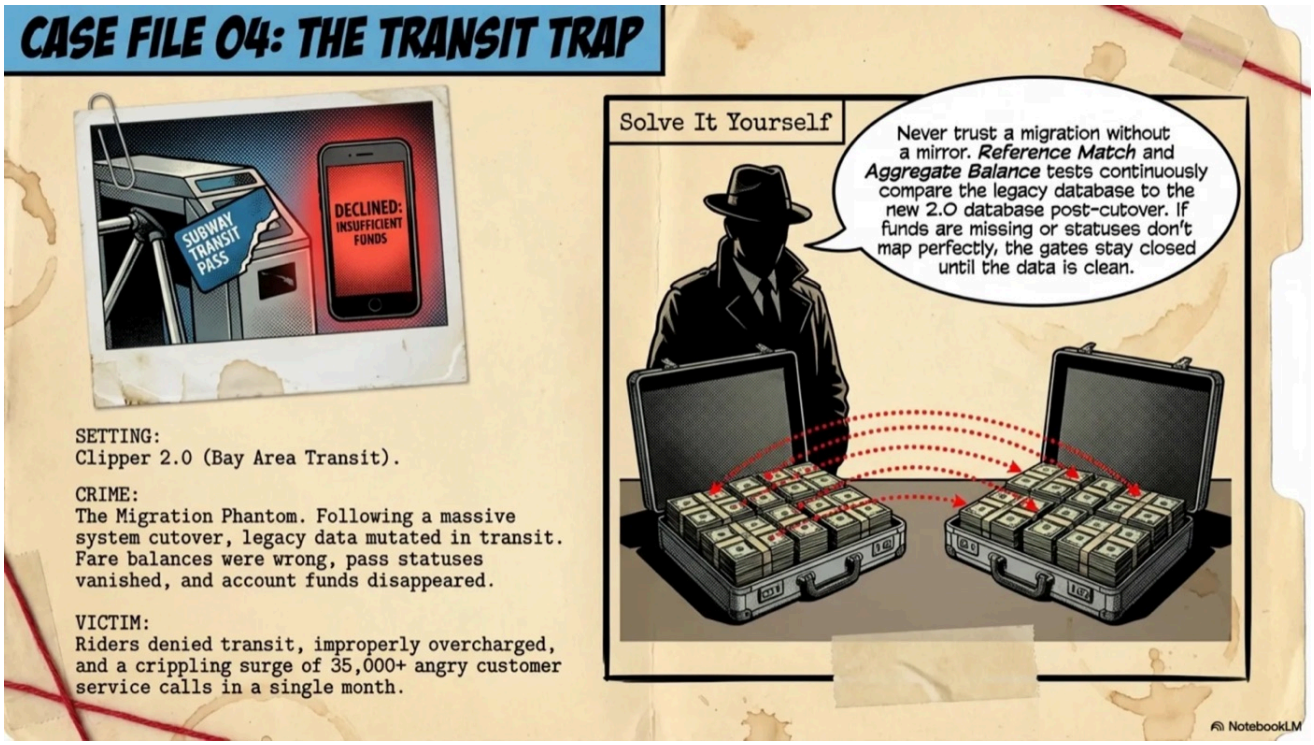
Setting: United Kingdom.

Fifteen universities classified full-time weekend-only courses as in-attendance. Under the rules, weekend-only students count as distance learners no matter where they physically study, and distance learners are not eligible for maintenance loans. The classification was wrong, nothing checked it, and the money went out.

The Office for Students put the scale at roughly 22,000 students receiving up to £190 million in payments they were not entitled to. The Department for Education first directed the Student Loans Company to recover the money immediately. After a campaign by the National Union of Students, it reversed that in April 2026: affected students repay through the normal income-contingent system instead of facing an immediate bill, and childcare grant recovery was paused. The House of Commons Library briefing has the full sequence.

The reprieve is real and it matters to the students. It does not make the data error go away. Around 22,000 people had their finances rewritten by a field nobody validated, and the funding stops going forward.

This is an open-and-shut case for a custom condition test. If study pattern equals weekend-only, then maintenance loan eligibility must be false. One rule. One test. Failing records rejected at submission rather than discovered two years later.



Case File 03: The Transit Trap (Clipper 2.0)

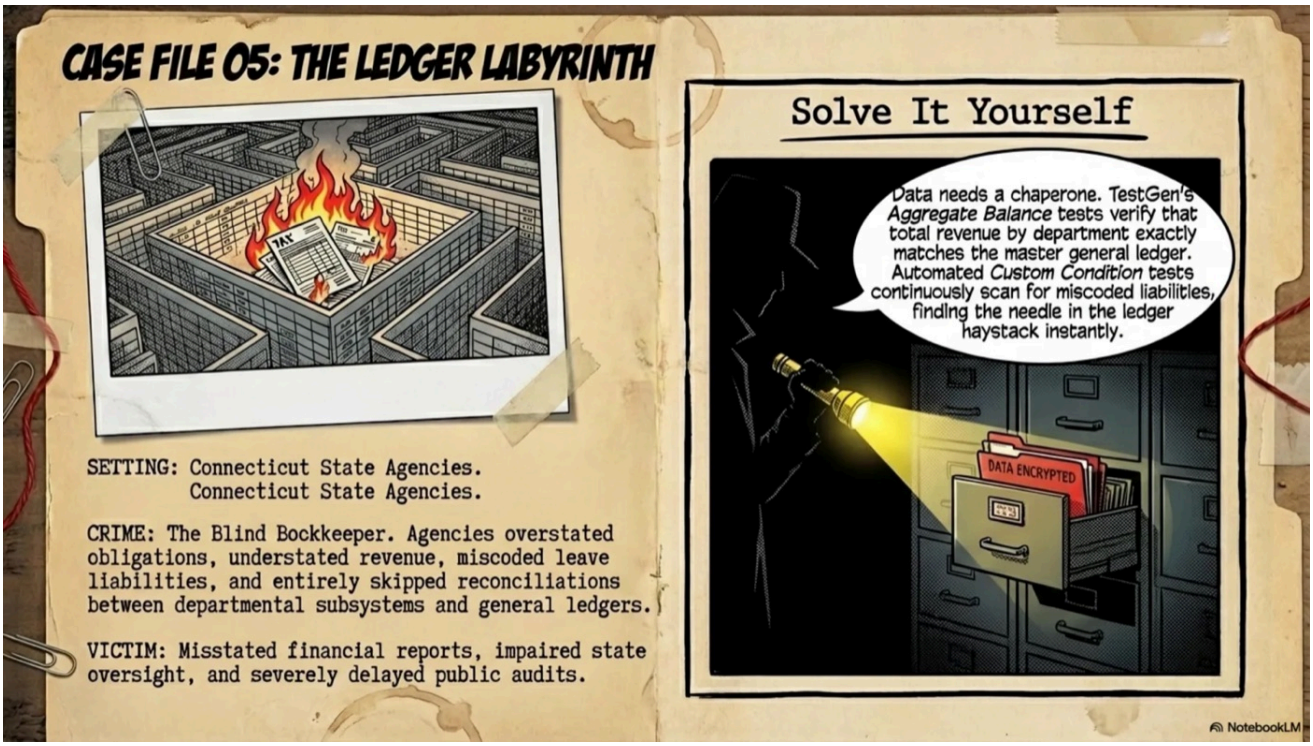
Setting: The San Francisco Bay Area.

Clipper 2.0 launched on 10 December 2025, replacing the fare system for every transit agency in the region. Card data did not survive the trip. Balances came out wrong. Passes vanished. Riders got charged incorrect fares, and ticket vending machines took money without crediting accounts.

The volume tells you how bad it was. Between 10 December and 15 January the customer service center fielded about 47,000 calls to agents, nearly four times what it was contracted to handle, with hold times reaching 100 minutes. Months later the contractor, Cubic Transportation Systems, was still doing manual cleanup of card data that failed to migrate, and three separate outages hit in March 2026, one lasting more than 12 hours.

Here is the part that should stop you cold. Internal emails obtained by KQED show the two sides arguing about testing: transit agency leaders had complained for months that Cubic could not resolve persistent errors, while Cubic staff said the MTC had not authorized enough testing. Testing was a line item somebody was deciding whether to fund, on a migration carrying 15 million riders' money.

Never trust a migration without a mirror. Reference match tests compare the old database to the new one key by key. Aggregate balance tests compare the sums. Run both against real production data before cutover finishes, and missing funds fail a test instead of stranding a rider at a gate. You do not ship a migration. You prove it.



Case File 04: The Ghost Millions (City of Mechanicville)

Setting: Mechanicville, New York.

The city's accounting records did not match the cash on hand. The New York State Comptroller reviewed four months of bank reconciliations from a 33-month window running January 2022 through September 2024 and found discrepancies totalling **$1,451,933**. That is audit 2025M-78, released 3 April 2026.

The audit is blunt about the mechanism. The former finance commissioner did not keep accounting records current or accurate, and city officials did not oversee financial operations. Municipal leaders made decisions on numbers that were fiction. A companion audit found 34 percent of sampled purchases did not comply with the city's own procurement policy.

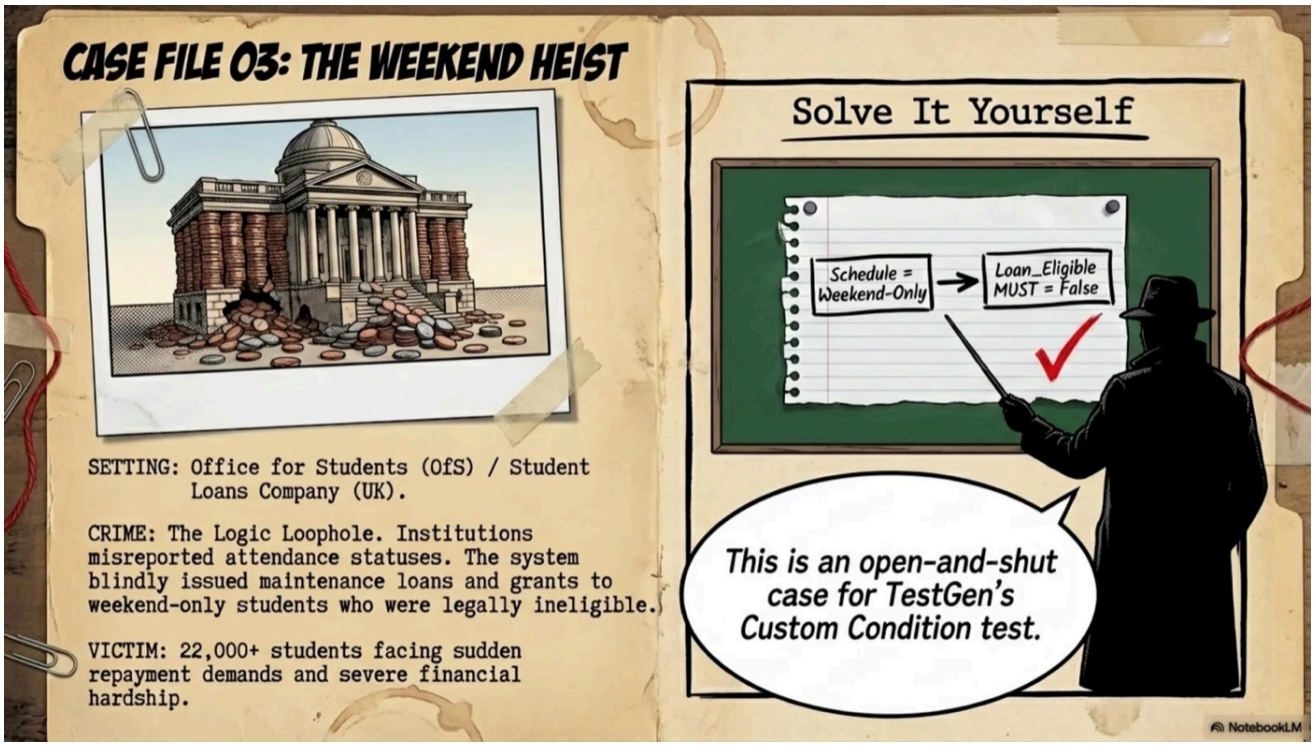
Four months of reconciliations produced a $1.45 million gap. Nobody had to go looking for it in a subtle place. It was sitting in the difference between two numbers that anybody could have subtracted, and the test is exactly that subtraction: an automated aggregate balance check matching the city's books to bank data, plus a custom condition test requiring accounting balances to sum to zero. When the math does not add up, the test fails before the mayor ever sees the report.

The case that never reaches the record

The four above made the news because a hospital closed, a government had to answer questions, a call center melted, and an auditor published. Most data failures never get that far, which is why the most common one is not in the case files.

Consider the shape of it, because you have probably shipped it. A scheduled job aggregates daily usage in UTC. The dashboard it feeds is labelled in local time. Every “yesterday” number covers a window that cuts off before the afternoon, so the figures come out slightly low. Plausibly low. Nothing to flag. The error surfaces in a quarterly review, weeks of reports later, when somebody senior asks why a segment fell off a cliff.

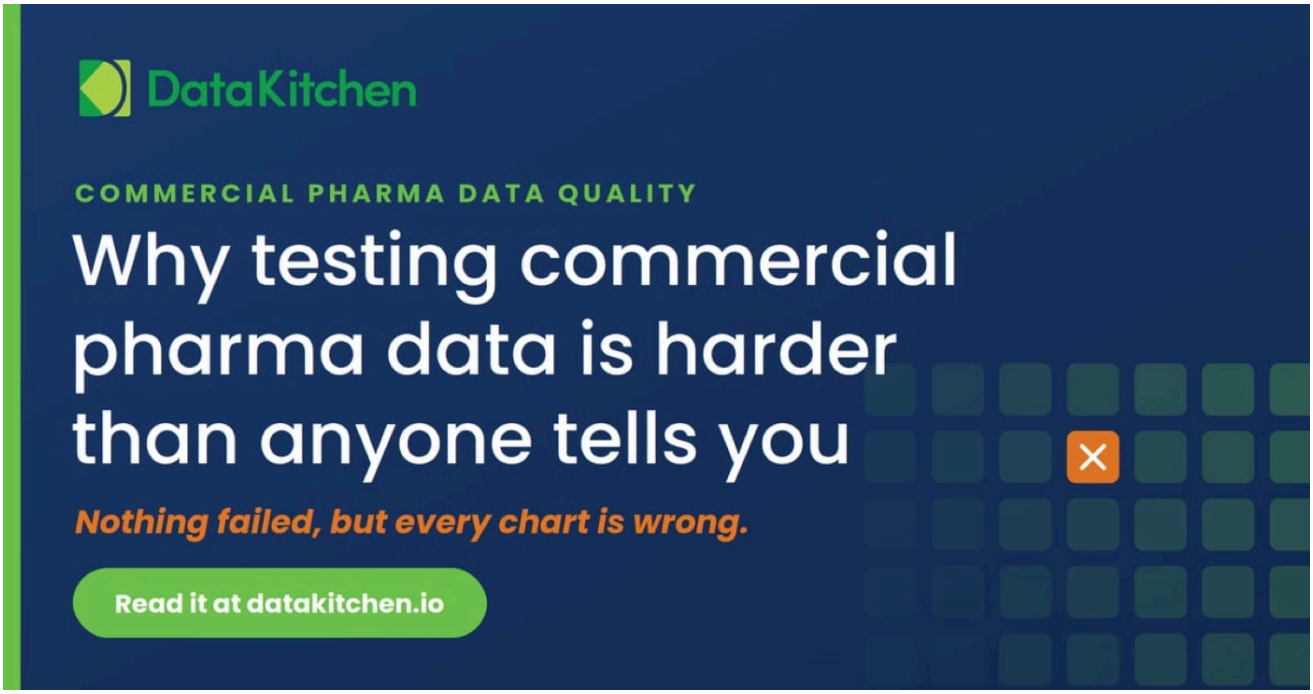
No regulator investigates that one. It costs you a forecast, a renewal conversation, and a chunk of your credibility, and it is invisible to every schema check you own. A volume anomaly test catches it on day two, because the daily row count for that segment drops out of its normal range and stays there.


CASE FILE 03: THE WEEKEND HEIST
SETTING: Office for Students (OfS) / Student Loans Company (UK).
CRIME: The Logic Loophole. Institutions misreported attendance statuses. The system blindly issued maintenance loans and grants to weekend-only students who were legally ineligible.
VICTIM: 22,000+ students facing sudden repayment demands and severe financial hardship.
Solve It Yourself
Schedule = Weekend-Only
Loan_Eligible MUST = False
This is an open-and-shut case for TestGen's Custom Condition test.
NotebookLM

IMPLEMENTATION AND CASE STUDIES

Why Testing Commercial Pharma Data Is Harder Than Anyone Tells You

Read the original article at: https://datakitchen.io/blog/why-testing-commercial-pharma-data-is-harder-than-anyone-tells-you/



You run data for a commercial pharma brand team. Your platform pulls in CRM activity, specialty pharmacy dispense feeds, claims, patient services records, REMS data, 3PL reports, and a stack of syndicated datasets you pay a lot of money for. Every one of them lands on its own schedule, in its own shape, keyed on its own identifiers. And every Monday, someone asks why the market share number moved.

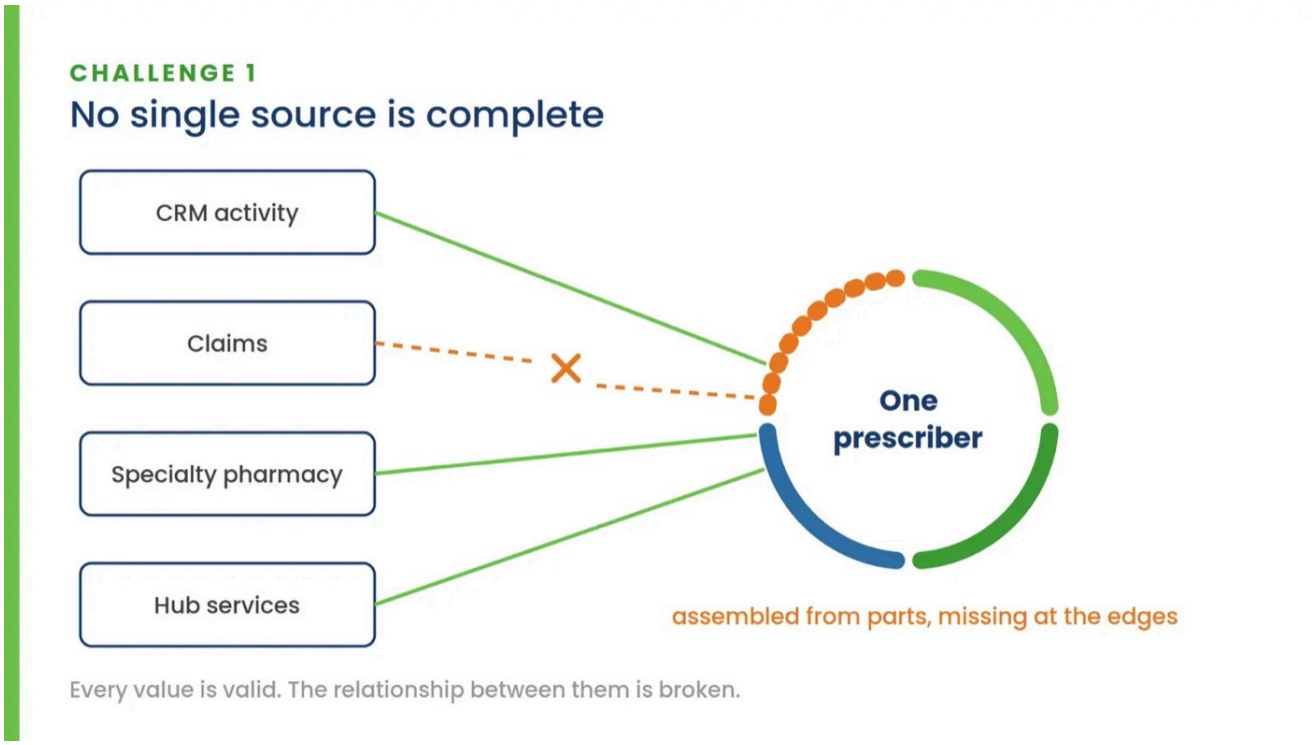
Generic Data Quality vs. Commercial Pharma Reality

Generic Data Quality	Commercial Pharma Reality
Column validation ("Check for nulls").	Relational integrity across fragmented sources.
Regex formatting ("Looks like a number").	Spec validation & active list confirmation (Luhn check, NDC history).
Loud SQL load failures.	Silent downstream metric shifts surfaced in QBRs.

Generic data quality advice doesn't help you much here. "Check for nulls" is not the problem. The problem is that commercial pharma data fails in specific, recognizable ways, and most testing approaches were built for a different data shape. Here are the challenges that actually bite, and at the end, a few things DataOps TestGen does about them. If you want the whole estate in one place, we keep a feed-by-feed inventory of commercial pharma data sources with the characteristic failure of each one.

Challenge 1: No single source is complete, so failures are relational

The picture of a prescriber, an account, or a patient gets assembled from several systems. Each one holds part of the truth. Each one disagrees with the others at the edges.



That means the failures that hurt you are relational, not scalar. A territory ID in the roster that doesn't exist in the hierarchy. An account with sales activity that never made it into the account master. Two files from the same vendor run that disagree on their shared keys. A patient recorded in two active treatment states at once.

Column-level checks never see any of this. Every individual value is valid. The relationship between them is broken. If your testing stops at "is this field populated and formatted," your integration layer is running blind.

And absence matters as much as presence. A provider withdrawn under a deletion request must remain gone. Suppressed entities and blocked identifiers must not reappear in a refresh. One cleanup pass doesn't enforce that. Only a continuous check does.

Challenge 2: Syndicated data fails in predictable ways that generic checks can't see

Purchased data feeds your most important analytics and produces your most characteristic failures. Feed for feed, it is the commercial data most likely to be wrong, because you did not create it and cannot control its schedule or its format. The failure modes repeat.

A weekly file should arrive exactly once per week. A monthly file, once per month. Feeds go missing. Feeds get loaded twice. A delivery arrives at half its normal size because the vendor had a bad extract. None of this throws an error. The load succeeds. The numbers are wrong.

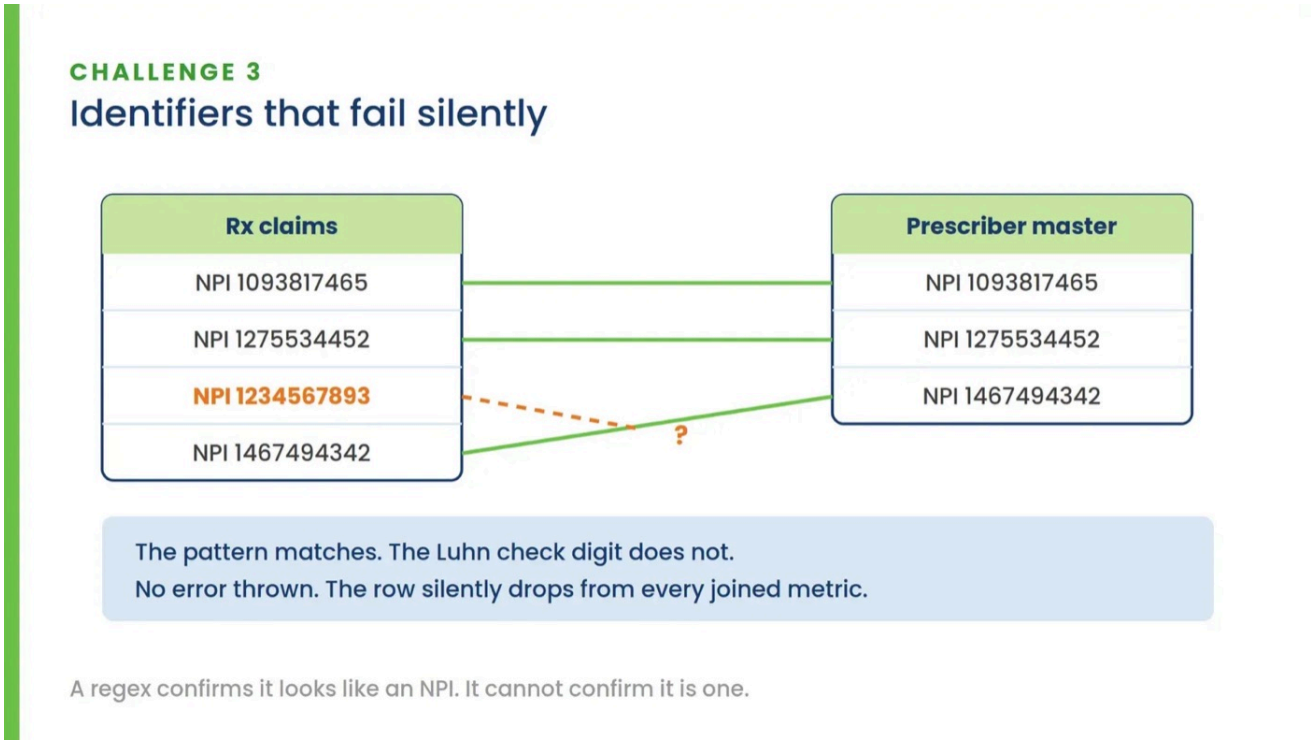
Then there's restatement. Many syndicated products revise the most recent months on every refresh, while older periods remain fixed. So a naive "nothing should change" rule fires constantly in normal behavior. Ignoring change entirely misses real breaks. You need checks that distinguish between an expected restatement window and a genuine problem in the stable tail.

Shape shifts are the sneakiest. A sudden change in the mix of payer types, specialties, or geographies usually means a mapping regression upstream, not a real market movement. By the time a brand analyst notices, the wrong number has been in three meetings.

One practical trap on top of all this: vendors love wide formats, with time periods spread across columns. Wide files defeat most trend logic. Normalize them into long format at your refined layer before you try to monitor volume, freshness, or distribution. Test after the normalization, not before.

Challenge 3: Identifiers that fail silently

NPI, NDC, DEA, ICD, HCPCS, and CMS place-of-service codes are the join keys of the entire commercial pharma ecosystem. Here's the ugly part: a malformed identifier doesn't error. It silently fails to join, and the damage surfaces downstream as a quietly wrong number with no obvious cause.

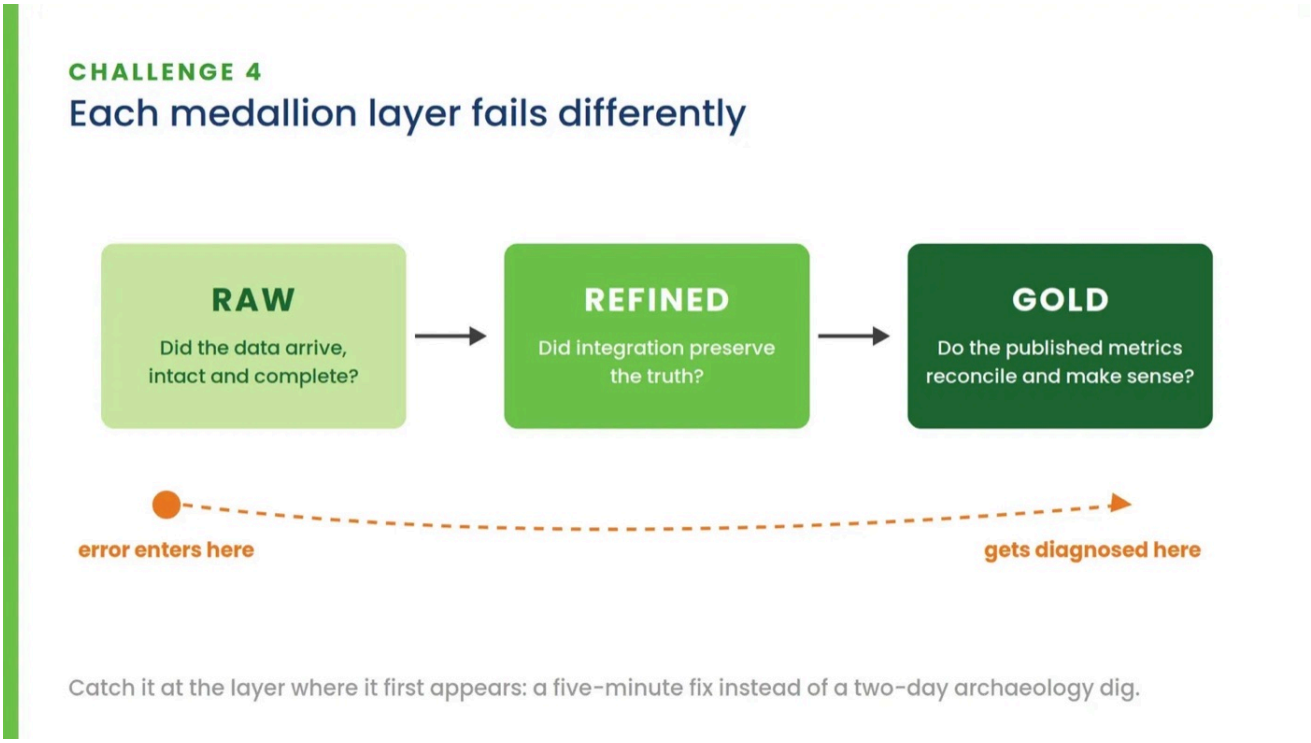


A regex can't save you. NPI carries a Luhn check digit. DEA has its own checksum. A pattern match confirms the value looks like an NPI. It cannot confirm the value is a valid NPI. ICD-9 and ICD-10 look similar enough to confuse a generic format rule. And validity against the spec is only half the question. Whether an NDC is an active product, or a diagnosis code is in the current published set, requires checking against an authoritative external list that changes over time.

Most teams handle this with custom SQL that one person wrote in 2022, and nobody wants to touch.

Challenge 4: Each medallion layer fails differently

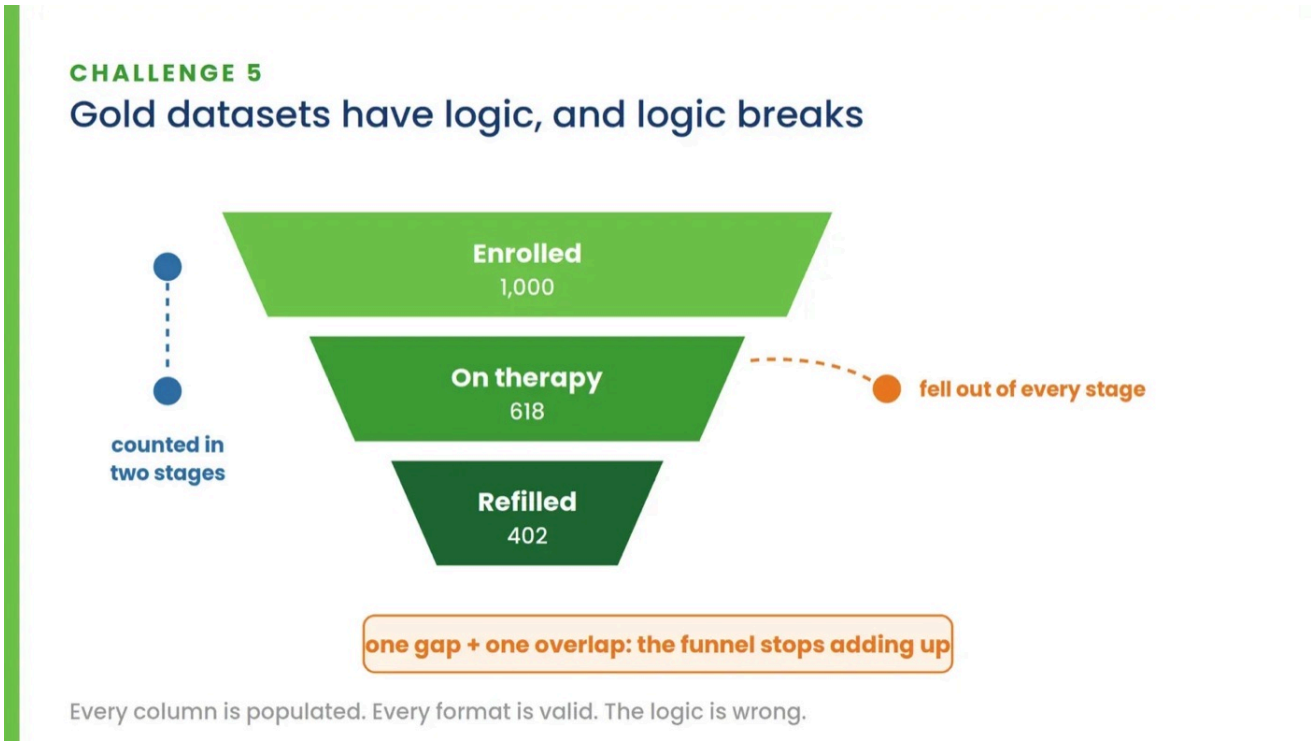
You’ve probably converged on some version of a medallion architecture: raw landing tables, refined and conformed layers, and gold datasets the business consumes. Testing only at the end means every failure surfaces at the layer where it’s hardest to diagnose.



Each layer asks a different question. At raw: did the data arrive, intact and complete? In the refined layers: did integration and transformation preserve the truth? At gold: do the published metrics reconcile and make sense? A row count check at gold tells you something is wrong. It tells you nothing about which of forty upstream steps broke it. Catching the error at the layer where it first appears is the difference between a five-minute fix and a two-day archaeology dig.

Challenge 5: Gold datasets have internal logic, and that logic breaks

The gold layer is where quality problems cost the most because they’re least visible. The columns are all populated. The formats are all valid. The logic is wrong.



Waterfall and funnel metrics require that every record be in exactly one stage. A gap in the classification logic and records fall out of the funnel. An overlap and you double count. Either way, the funnel stops adding up, and someone finds out in a QBR.

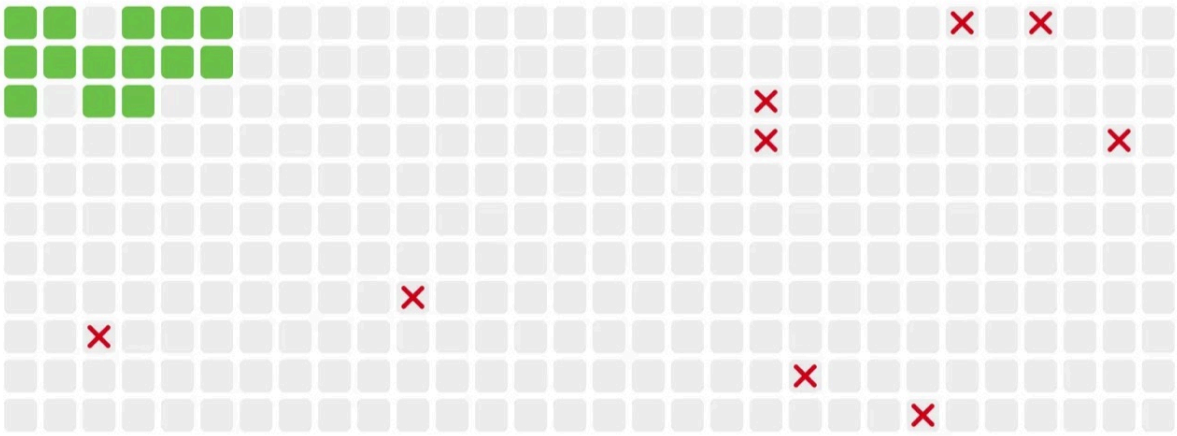
Patient journey and treatment views need milestone dates in a valid sequence. A ship date before an enrollment date isn't a format error. It's a logic error, and only a test that understands the sequence catches it. Same story with published totals: they have to tie back to source detail, per grouping, on every run. The day an integration step starts dropping or double-counting rows is the day you want to know, not the quarter.

Challenge 6: You can't hand-write your way to coverage

Here's the challenge underneath all the others. A single commercial data platform spans dozens of sources and thousands of columns, and the surface grows with every new feed, vendor, and brand. Hand-writing a test per column is not a realistic plan. So you write tests where you expect trouble: the handful of fields someone remembered to guard.

CHALLENGE 6

You can't hand-write your way to coverage



the columns someone remembered to guard where the errors actually show up

Thousands of columns. A handful of hand-written tests. Errors don't respect the plan.

Errors don’t respect that plan. They show up in the long tail of columns nobody covered, and inside integration flows too intricate to script checks for by hand. Exactly the parts of the pipeline your manual test suite leaves dark.

The hand-written suite decays, too. Those hardcoded “expect at least 50,000 rows” thresholds go stale as the data grows. They either fire false alarms or get commented out. Both outcomes end the same way: you learn about the problem via a Slack message.

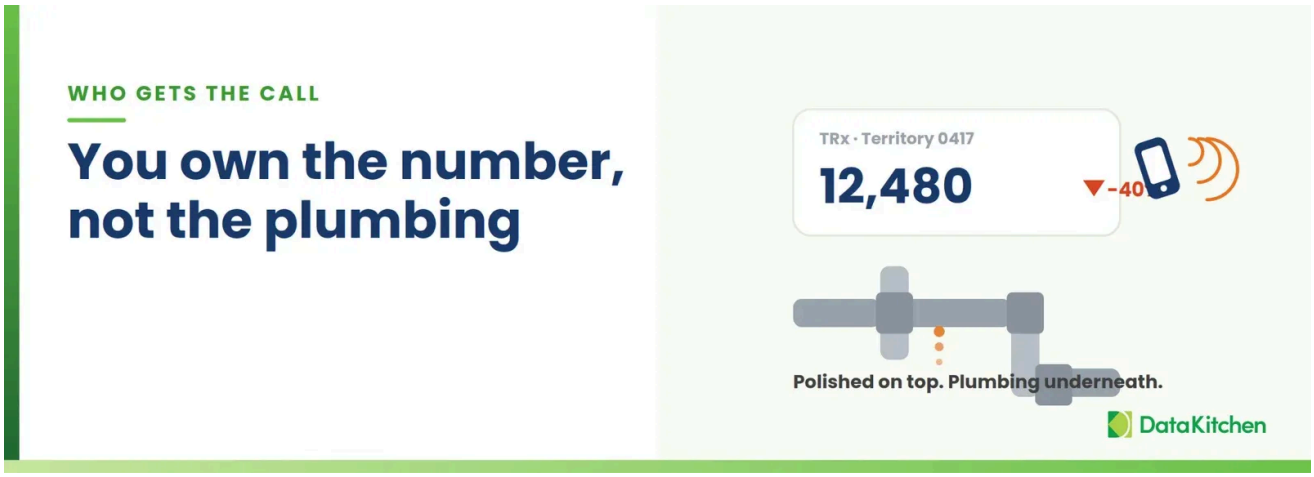
IMPLEMENTATION AND CASE STUDIES

IQVIA, specialty pharmacy, and your own files: the commercial data most likely to be wrong

Read the original article at: https://datakitchen.io/blog/commercial-pharma-data-most-likely-to-be-wrong/

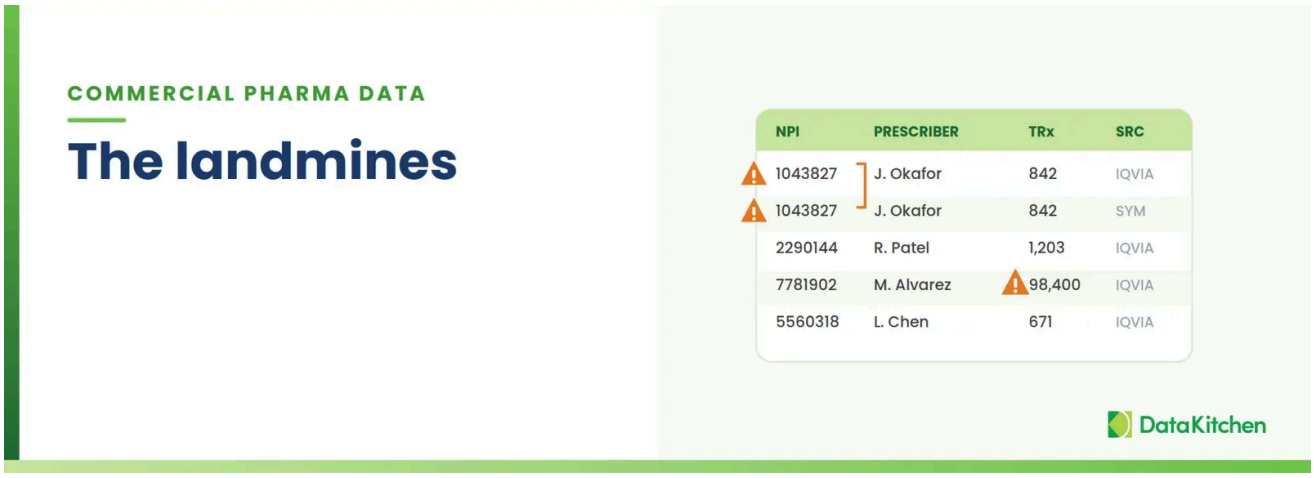
Most of the data behind your launch came from someone else. IQVIA sent Xponent. Symphony sent claims. The specialty pharmacies sent dispense data on their own schedules, in their own formats, with their own idea of what a “week” is. You load all of it. Then you build the field report on top and hope the numbers hold.

They don’t always hold. And the way you find out is a rep calling to ask why their territory dropped 40 percent overnight. Testing commercial pharma data is harder than anyone tells you, and it starts with the feeds themselves.



You own the number, not the plumbing

When the number is wrong, the brand team does not call your data engineer. They call you. And while that fire burns, your analysts spend the week reconciling files by hand rather than identifying the next access opportunity. The problems below look technical. Every one of them is yours. It helps to know what they look like.



The landmines

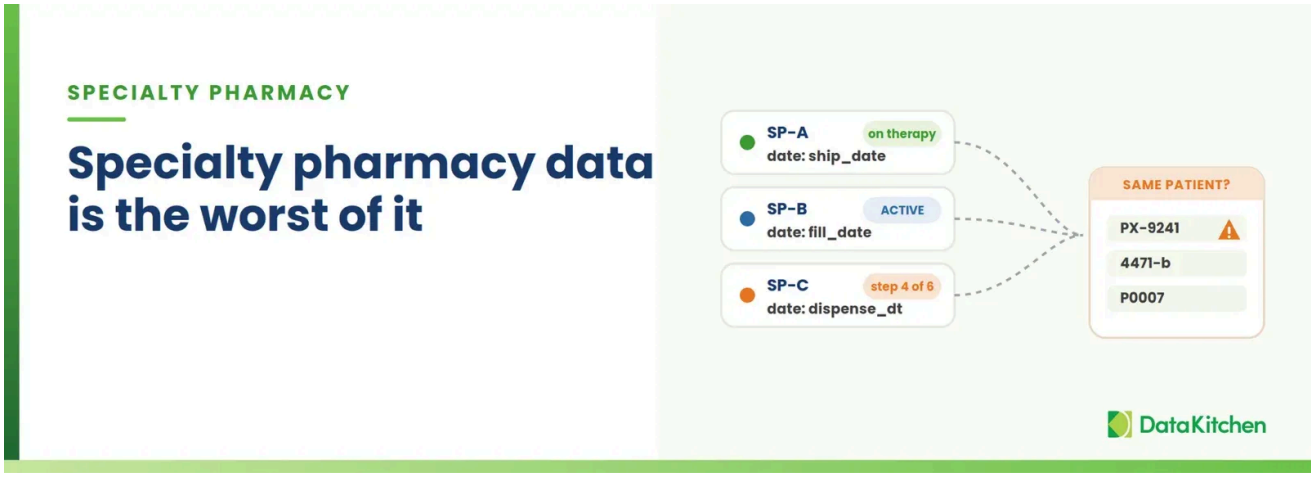
Two physician files, one truth, and they disagree. You get prescriber data from multiple sources. IQVIA says Dr. Smith wrote 30 scripts. Your MDM says Dr. Smith and Dr. Smith are two different people. Now your targeting list has a ghost and your alignment double-counts. Nobody notices until the incentive comp run, and then everybody notices. A transposed NPI does the same damage from the other direction, and it passes a ten-digit pattern check every time.

The vendor restated last quarter without telling you. Third-party sales data is part actual, part projection. The vendor backfills the gaps with an algorithm. Next month they get better inputs and rerun it. Your actuals from last quarter quietly changed. The trend line you showed the VP last week is now different. Same query, different answer.

The decimal moved. A quantity column arrives as units one month and as packages the next. Or a price lands in cents instead of dollars. One product shows a 10x jump, and the dashboard renders it as a heroic sales week. It was a unit-of-measure change nobody flagged.

Half the file showed up. The specialty pharmacy feed is supposed to have 40,000 rows. Today it has 26,000. The load succeeded. No error. The pipeline is green. Your dispense numbers are just wrong, and everything downstream inherits the hole.

The vendor renamed a column. Last week it was RX_CNT. This week it's RxCount. Your transform maps the old name, finds nothing, and fills the field with nulls. The pipeline doesn't fail. It produces a clean, well-formatted, completely empty metric.



Specialty pharmacy data is the worst of it

If the vendor feeds are messy, specialty pharmacy data is the mess with another mess stacked on top. A retail brand pulls prescription data from one or two big aggregators. A specialty brand pulls it from the network of specialty pharmacies that actually dispense the drug, and that network can be a dozen pharmacies or more. Each one sends its own file, on its own schedule, in its own format.

None of them agree on what the words mean. One pharmacy's ship date is another's fill date is another's dispense date. One counts a referral the day it arrives, the next counts it the day benefits clear, the third counts it when the first box goes out the door. Patient status is worse. One calls it on therapy, another calls it active, a third has six sub-statuses you have never seen and no way to map them to anyone else's. You are supposed to stitch all of it into one patient journey, and the pieces were cut by different hands.

Then there is the patient key, or the absence of one. Specialty pharmacy data is often your only window into patient-level activity, because a limited-distribution drug has no retail claims to fall back on. But each pharmacy de-identifies patients its own way, so the same patient can show up three times under three tokens and you cannot tell. Your adherence rate, your time to fill, your discontinuation curve all sit on top of that.

Here is the part that makes it dangerous. Nobody owns this data. It lands on an analyst who is already doing three other jobs, who opens the files, does their best in a spreadsheet, and moves on. Specialty pharmacy integration is almost always somebody's afterthought, right up until it becomes the number the access team is betting the launch on.

What each of these costs you

None of these is a rounding error. A duplicated prescriber splits credit across two records, and your incentive comp run pays the wrong reps. A silent restatement moves a number you already briefed to leadership, and every forecast built on it is now suspect. A unit-of-measure jump sends the field chasing a spike that never happened. A half-loaded specialty feed hides real patients from the access team during the weeks that matter most. Every feed a commercial launch runs on fails in its own particular way, and we catalogue them in a feed-by-feed inventory of commercial pharma data sources.

Launch is where this hurts worst. The first year sets the trajectory a brand rides for the rest of its life, and you get one chance at the field's trust. One wrong territory number and the reps stop believing the dashboard. After that, they run on gut, and the platform you paid for is furniture. Good commercial pharma analytics starts with the data the field believes in.

There is a second cost, quieter than the first. Reconciled data is defensible data. When a rep disputes their comp, you need to show the number held. When leadership challenges a forecast in the business review, you need to show the history did not move under you. And if your company is the acquisition target your investors are hoping for, the data behind your commercial story has to survive due diligence. A reconciliation trail on every feed is what turns "trust me" into "here is the check that passed."

It does not have to go this way. One commercial launch platform, behind a brand later acquired for $14 billion, ran on roughly one and a half engineers and had field reporting live six months before launch day, because the checks caught the bad loads before anyone downstream saw them. The launch case study has the numbers. The difference was not heroics. It was reconciliation on every feed.

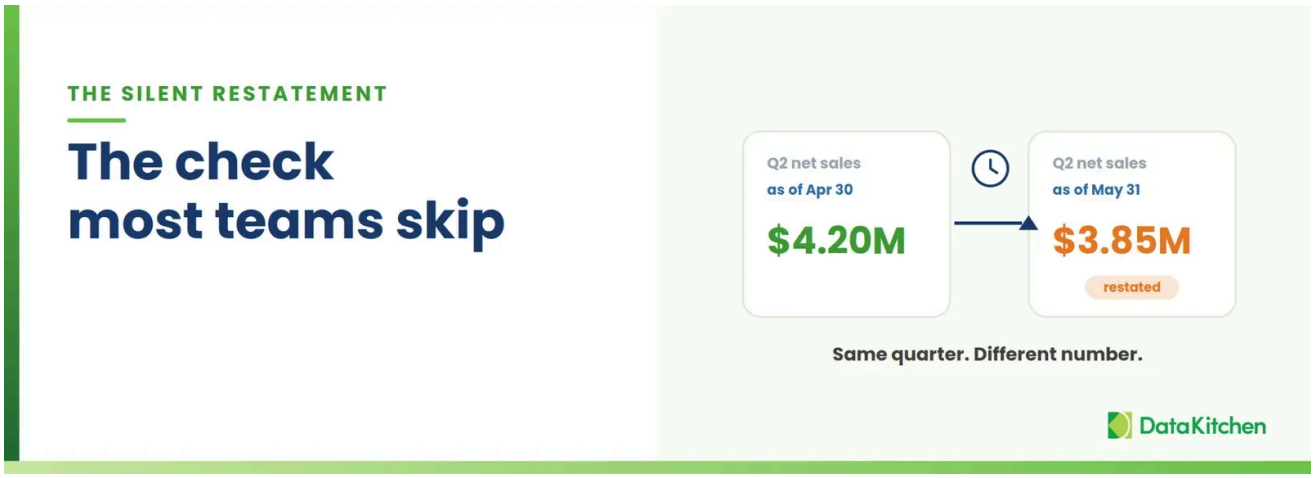
Test instead of trust

You have to check two things, not one: that the file arrived intact, and that last month's history did not change underneath you. Most teams check the first and skip the second. The rest of this section is how.

The first check is extract to load. Did the file arrive intact? This one is deterministic, and it should be exact. Count the rows in the vendor's control record and in your staging table; if they match, the load is correct; otherwise, the load is wrong. Sum the amount column on both sides to the penny, at fixed precision, so two engines don't disagree over floating point. Confirm the max load date is today, not three days ago. When the vendor gives you no control count, fall back to an expected range instead of a fixed number, because a specialty feed legitimately shrinks on a holiday week. A count of 26,000 against a normal 40,000 is not automatically wrong, but it is automatically worth a look.

These checks run in seconds and catch partial, stale, and truncated files before anyone opens a dashboard. When one fails, the load stops in staging and never reaches the mart. That is the point of a two-stage architecture: bad data needs somewhere to fail that the field never sees.

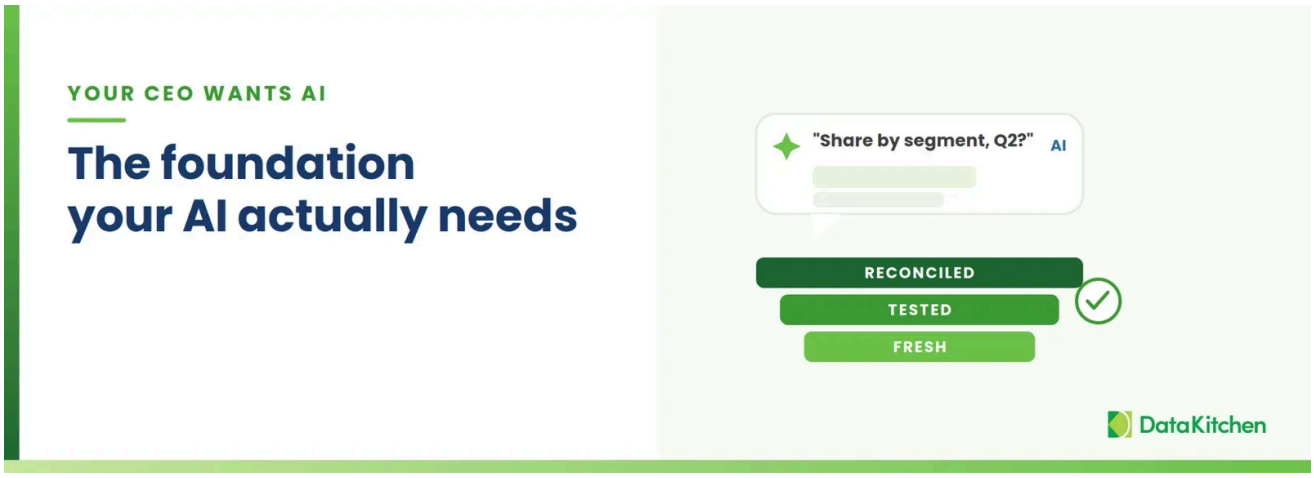
Row counts and sums tell you the totals line up. They don't tell you that the rows are the same. Two files can carry identical counts and identical sums and still be full of swapped values. To catch that, you fingerprint each row in the source, fingerprint it in the target, and compare them. If a value changed, the fingerprint changes, and you see exactly which rows drifted.



The check most teams skip

Extract-to-load reconciliation assumes last month's history remains last month's history. In commercial pharma, it often does not. Third-party sales data gets rerun when the vendor receives better inputs, and your Q2 actuals change inside a file that loads perfectly. Source and target agree. Every deterministic check passes. And the number you briefed to leadership is now different.

The only way to catch that is to remember what the vendor told you before. Keep a snapshot of each period as you received it. When the next file lands, diff the overlapping periods against the snapshot. Expected corrections get reviewed and accepted. Silent rewrites get flagged before they reshape a trend line under you. This is the check most teams skip, and it is the one that protects your credibility with leadership. A commercial insights leader at Celgene reduced errors to about one per quarter and held it there for years. That is what disciplined reconciliation buys you.



The foundation your AI actually needs

Your CEO wants AI, and you are the one who has to deliver it. Here is the part nobody says out loud. Your Snowflake Cortex Analyst or Databricks Genie cannot answer a real commercial question until the underlying data is right. Ask it for share by segment on top of an unreconciled feed, and it will answer with

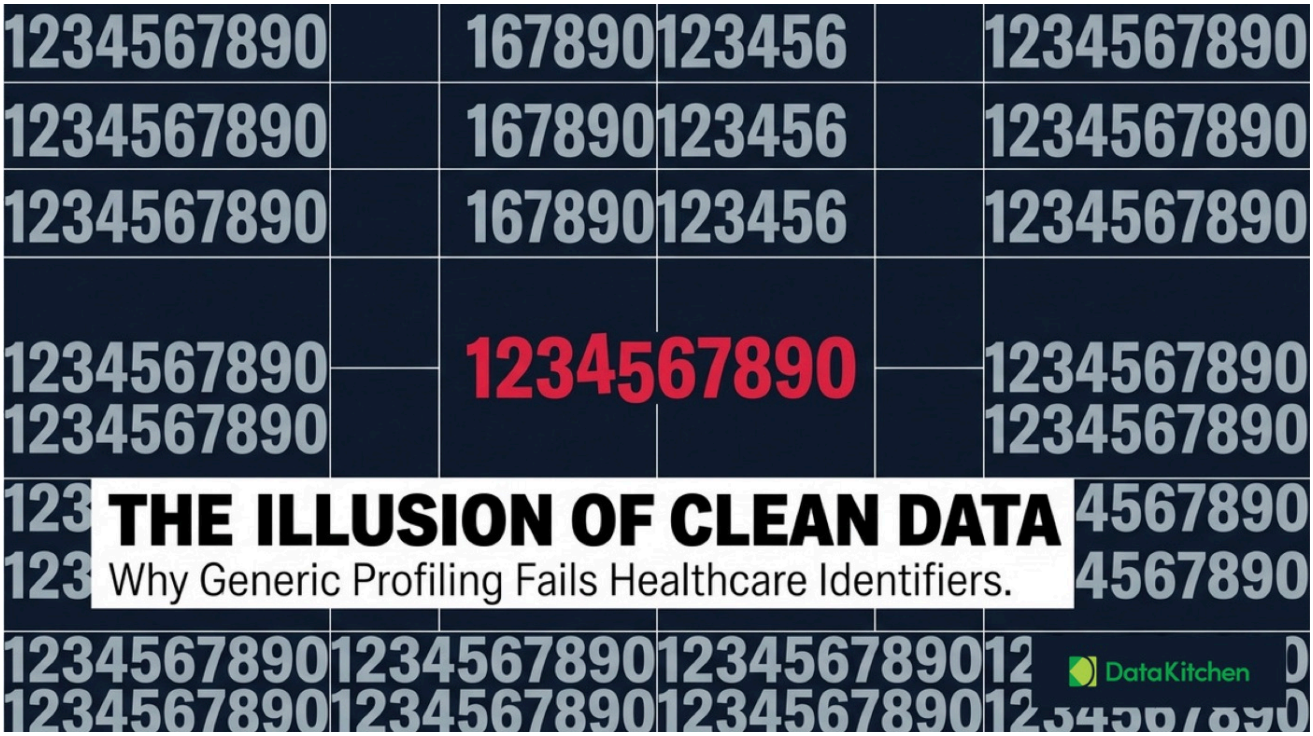
total confidence and be wrong, which is worse than not answering at all.

Reconciliation is not separate from your AI mandate. It is the first step of it. Trust in the data is what turns an AI demo into an AI answer, and it is the Data Trust half of AI success.

IMPLEMENTATION AND CASE STUDIES

The Illusion of Clean Data: The Pharma Identifiers Generic Tools Never Check

Read the original article at: https://datakitchen.io/blog/npi-ndc-icd-validation-tests/



Picture the ticket. A rep writes in on Monday because their territory is short about 40 TRx and they want to know where it went. You trace it to one prescriber whose NPI in the specialty pharmacy feed does not match the NPI in Veeva. The value is not missing and it is not null: ten digits, all numeric, sitting there looking exactly like an NPI. Two of them are transposed, and the join has been dropping that prescriber since March. Your pattern test on that column has been green the entire time.

It was green because commercial pharma runs on identifiers almost nobody outside the industry has heard of. NPI for the prescriber. DEA for the one who can write a controlled substance. NDC for the product, at two different levels of specificity. ICD-9 and ICD-10 for the diagnosis, often in the same column. HCPCS J-codes for anything infused. CMS Place of Service for where it was given. NCT for the trial. These are the join keys the whole commercial picture is assembled on, and every one of them has a published format with rules that a general-purpose data quality tool knows nothing about.

So it counts characters instead. Ten digits in the NPI column, so the column passes. That is the hole, and it is wide enough to drive a quarter through.

The pack, in one paragraph

As of 27 August 2026, DataOps TestGen Enterprise ships a generation set called **Commercial Pharma**: twelve test types covering the identifiers above, validated the way their specifications define them rather than by counting characters. They generate off your profiling run alongside the standard set, and there is nothing to configure. The release notes list the whole set.

Test	What it checks	Severity
Valid NPI	10 digits plus Luhn check digit with the 80840 prefix	Fail
Valid DEA	Letter, letter or 9, seven digits, with the DEA check digit	Fail
Valid NDC	11-digit or hyphenated package form (5-4-2, 4-4-2, 5-3-2, 5-4-1)	Warning
Valid NDC Product	Hyphenated two-segment product form (4-4, 5-3, 5-4)	Warning
Valid NDC Any Form	Any recognized NDC form, including bare 10-digit	Warning
Valid ICD-10 Diagnosis	ICD-10-CM shape, dotted or undotted	Fail
Valid ICD-9 Diagnosis	ICD-9-CM numeric, V-code, and E-code shapes	Fail
Valid ICD Diagnosis	Either ICD-9-CM or ICD-10-CM, for mixed columns	Fail
Valid ICD-10 Procedure	Seven characters, no I or O	Fail
Valid HCPCS J-Code	J followed by four digits	Warning
Valid CMS POS	Membership in the published CMS Place of Service set	Fail
Valid NCT	Eight digits, with or without the NCT prefix	Warning

Table 1: all twelve are Validity tests on a single column, and all belong to the Commercial Pharma generation set.

Four kinds of check, and the difference matters

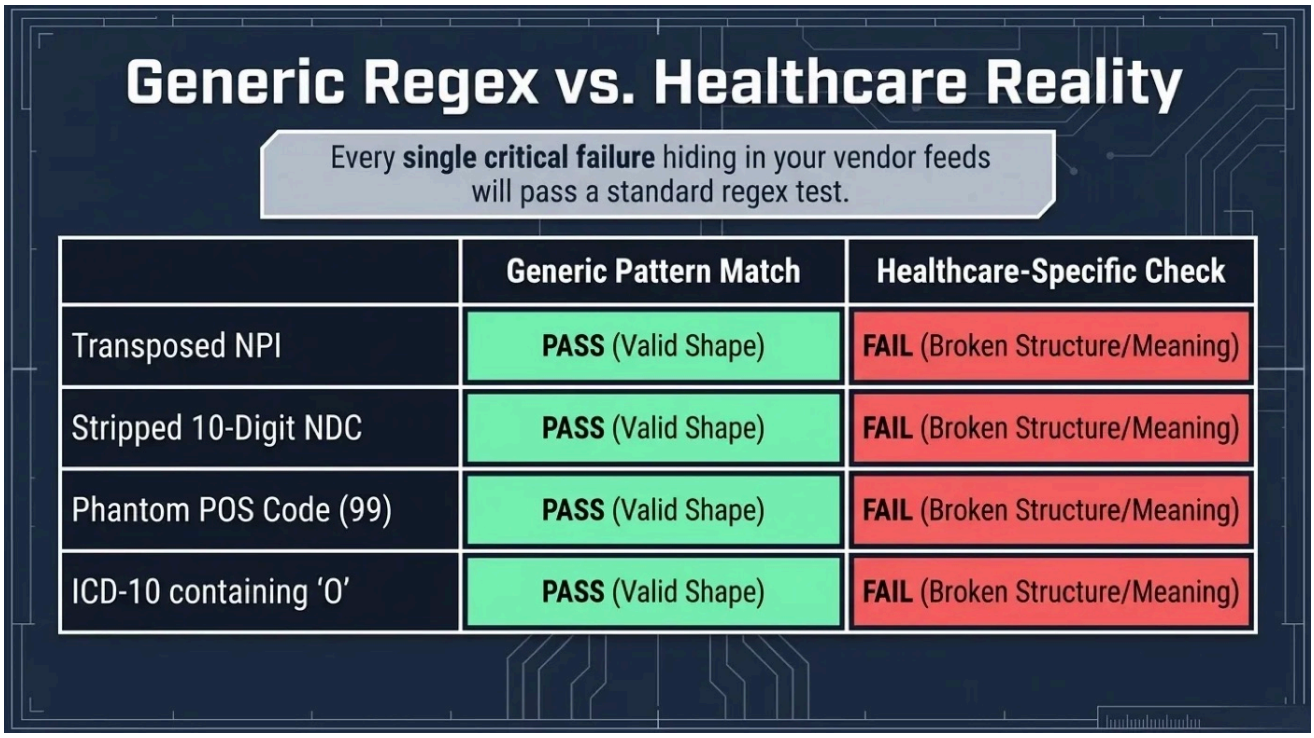
The twelve sort into four groups. Knowing which group a test belongs to tells you how much a passing result is worth, which is the part that gets glossed over most often.

Two run arithmetic. Valid NPI and Valid DEA compute the check digit the specification defines and compare it against the digit sitting in the value. A failure here is not a matter of opinion.

Three cover the level an NDC column stores. Package, product, or unsegmented.

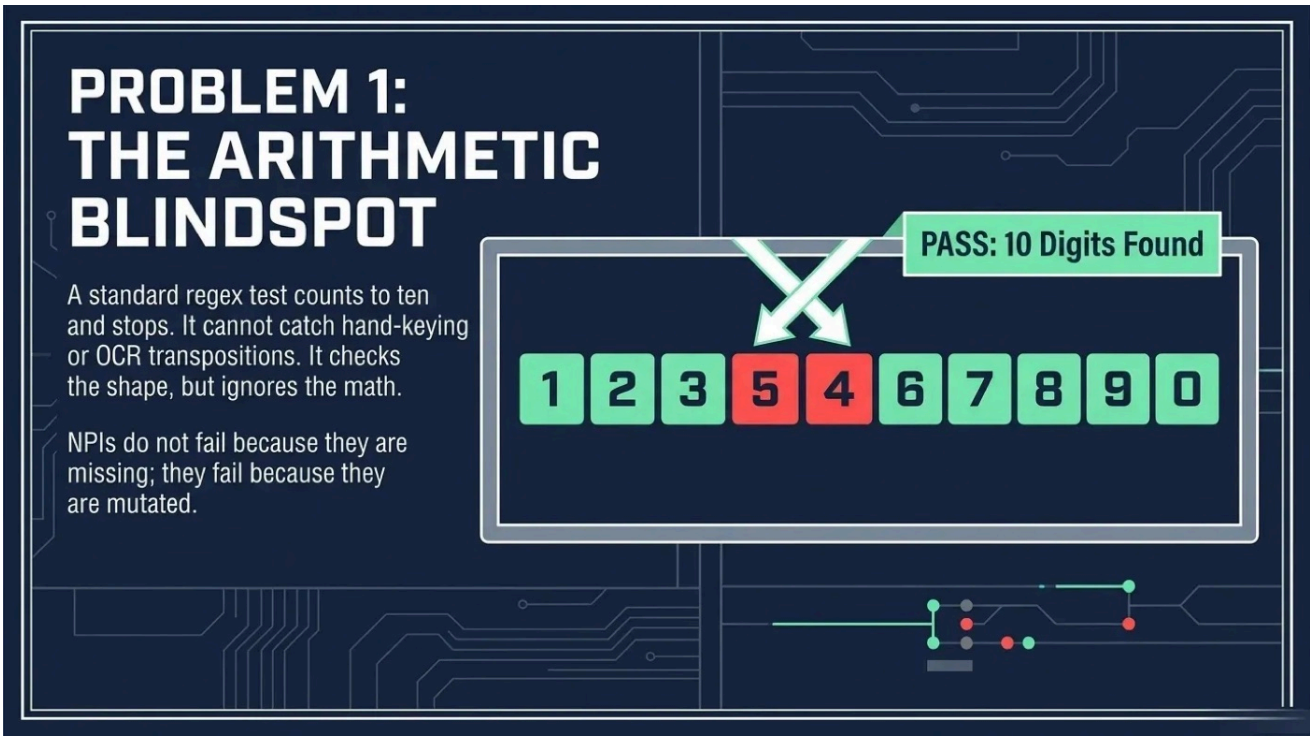
Six check shape only. The four ICD tests, Valid HCPCS J-Code and Valid NCT confirm a value is well-formed. They do not consult any code set, so a well-formed value that is not a real code will pass.

One checks membership in a published list. Valid CMS POS, and it is the only one. Hold on to that distinction.



Arithmetic: the two check digits

An NPI is ten digits, and the tenth one is not data. It is a Luhn check digit computed over the first nine, with the `80840` prefix the spec requires folded into the sum. Transpose two adjacent digits and the sum stops landing on a multiple of ten, which is the entire reason the digit is there. Hand-keying and OCR produce transpositions, so the identifier was designed to catch them. A regex that counts to ten throws that design away.



DEA registration numbers carry the same idea in a different arithmetic. The seventh digit has to equal the first, third and fifth digits added together, plus twice the sum of the second, fourth and sixth, mod ten. Valid NPI and Valid DEA run those calculations in SQL, on the landing table, on every run.

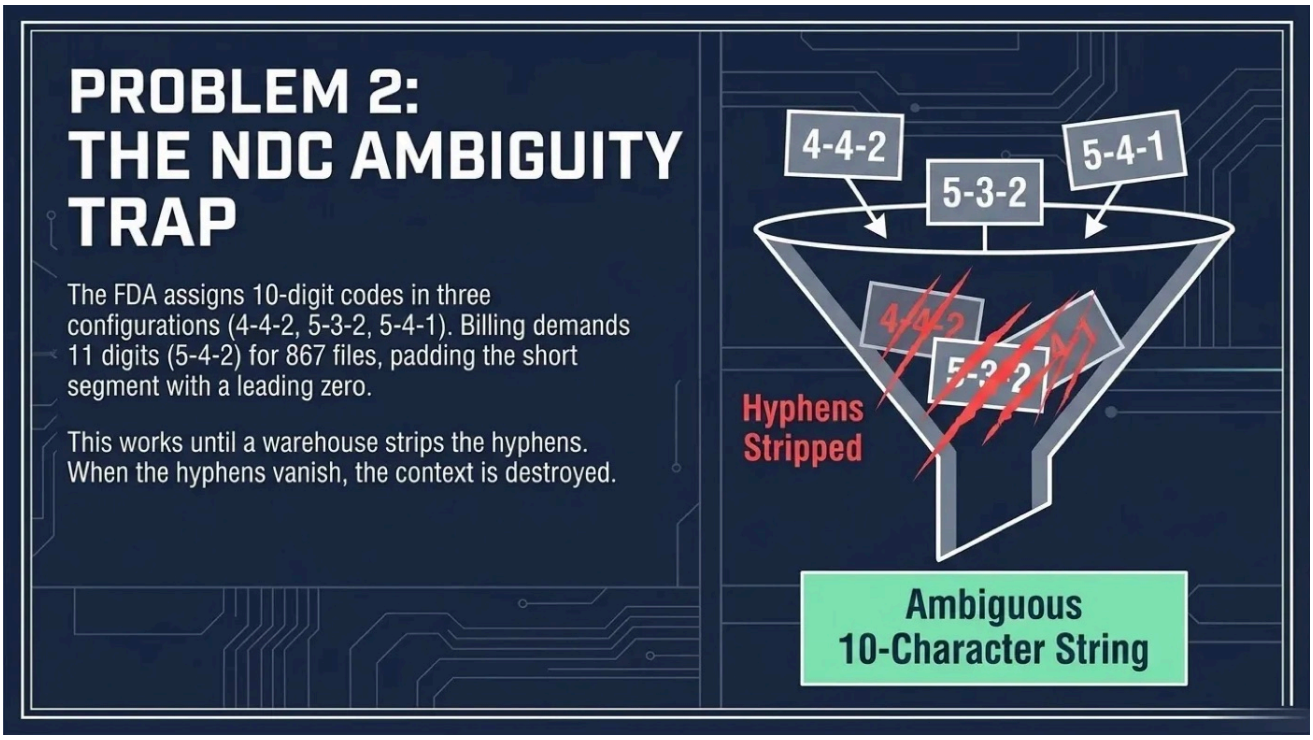
Level: which NDC your column actually stores

NDC is where most commercial pharma warehouses lose a day a quarter, and the reason matters, because people usually get it wrong.

The FDA assigns a ten-digit code in one of three segment layouts: 4-4-2, 5-3-2 or 5-4-1. Billing wants eleven digits in 5-4-2, which you get by padding the short segment with a leading zero. That works as long as the segments are still visible. Strip the hyphens and they are not. Take `1234567890`:

Layout	Segmented	Padded to 11
4-4-2	1234-5678-90	01234567890
5-3-2	12345-678-90	12345067890
5-4-1	12345-6789-0	12345678900

Table 2: three different products, the same ten characters, and nothing in the value to say which one you are holding.

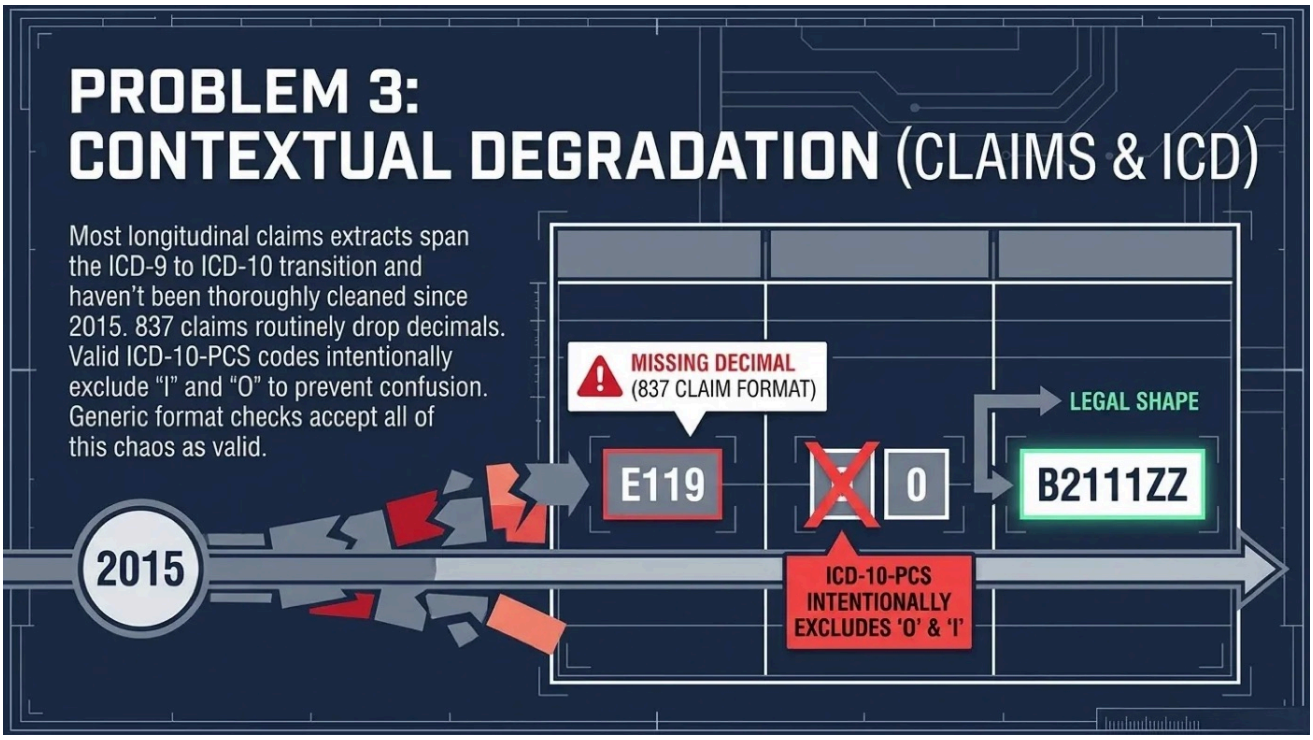


Be clear about what the three tests do and do not resolve. All three accept every FDA layout; none of them can tell you which layout a stripped value came from, and Valid NDC Any Form says so in its own usage notes. What separates them is the code level your column stores. Valid NDC takes the package-level three-segment form and the eleven-digit form your 867 files and claims data carry. It rejects a bare ten-digit string deliberately: that string is ambiguous, and treating it as valid is how the wrong product ends up in a report. Valid NDC Product takes the two-segment product-level form your product master and formulary files carry. Valid NDC Any Form takes any recognized form including bare digits, for the warehouse that strips separators as a convention and has decided to live with the ambiguity. Even that loosest one still catches the leading zero Excel ate, the `UNKNOWN` somebody typed into a required field, and the value that is somehow twelve characters long.

Shape: ICD, J-codes and trial numbers

Four tests cover ICD. Valid ICD-10 Diagnosis, Valid ICD-9 Diagnosis, and Valid ICD Diagnosis for the column that spans the transition and honestly holds both, which describes most longitudinal claims extracts nobody has cleaned since 2015. Valid ICD-10 Procedure checks the seven-character PCS shape, including that the code set leaves out `I` and `O` so nobody confuses them with `1` and `0`. The diagnosis tests accept dotted and undotted forms, because 837 claims drop the decimal and your vendor extract may or may not have put it back.

Valid HCPCS J-Code checks the J plus four digits that Part B buy-and-bill runs on, one slice of HCPCS Level II scoped to drugs. Valid NCT covers ClinicalTrials.gov identifiers on the medical affairs side, as NCT plus eight digits or eight bare digits.

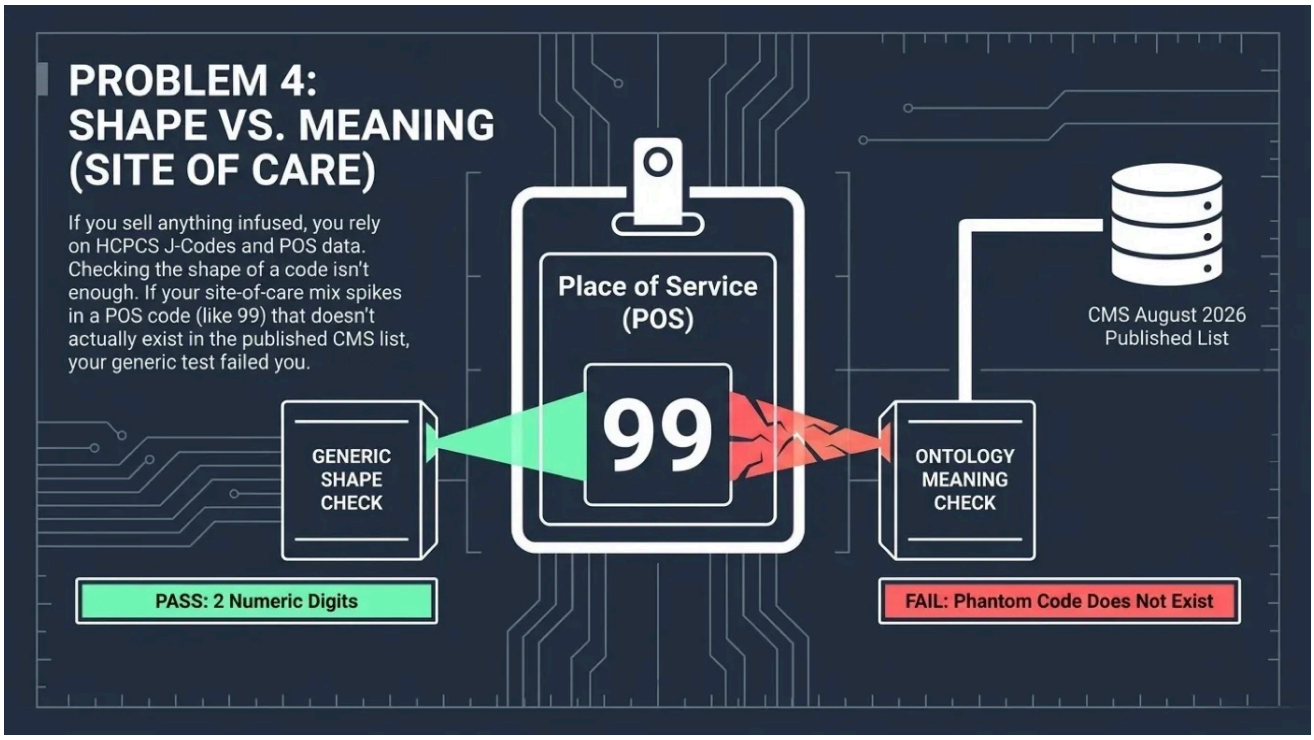


These six check that a value is well-formed. They do not check that it exists. `Z99.999` passes Valid ICD-10 Diagnosis, `000.00` passes Valid ICD-9 Diagnosis, `ZZZZZZZ` passes Valid ICD-10 Procedure, and `J0001` passes Valid HCPCS J-Code, and not one of those is a real published code. That is a deliberate scope decision rather than a gap: the ICD-10-CM code set alone runs to more than 70,000 codes and revises annually, so carrying it inside a test would mean shipping a copy that silently goes stale.

Membership: the one test that checks a real list

Valid CMS POS is the exception, and the pack's genuine differentiator. CMS publishes Place of Service as a small fixed list, small enough that the test carries the actual list and checks membership against it. The copy in the pack is dated in its own usage notes: current as of August 2026, including codes 27, 58 and 66.

That means a Place of Service code the test rejects is worse than malformed. It does not exist. If your site-of-care mix suddenly shows growth in a POS code CMS never published, you would rather hear it from a test than from the slide.



You don’t configure any of it

Profile the schema and the tests write themselves, using the same mechanic that wrote data quality tests for 3 million NYC taxi rides in five minutes, pointed at identifiers instead of fare amounts.

Generation reads the column name and the profiled values together, and it needs both to agree. A column matching `%npi%` whose most frequent pattern is 10 digits gets Valid NPI. Unless the name says phone, fax, or contact. Unless profiling already tagged the values as a US phone number, which happens more than you’d like on vendor feeds where the HCP contact block sits three columns over.

The two-signal rule is not caution for its own sake. `B2111ZZ` is a legal ICD-10-PCS procedure code and a legal ICD-10-CM diagnosis code, so the shape cannot tell you which one you are holding, but the column name usually can.

The cost is that a column named `code_3` gets nothing. Add that one by hand, the same way you shop the rest of the test catalog. That’s the residue no profiler reaches, and on a commercial team your best data quality rules live in someone else’s head: the analyst who knows which specialty pharmacy reports units instead of scripts.

The specific test replaces the generic one

You already had Pattern Match, and on a column Valid NPI now covers it is noise. So when a Commercial Pharma test generates on a column, it removes the Pattern Match test that generated on that same column in the same run. Only that run’s tests. Only unlocked ones. Anything you wrote by hand or locked stays exactly where it is. Before you confirm generation, the dialog tells you the most it will delete.

What these tests won’t tell you

These tests check format, plus a check digit where the specification defines one, and with the single exception of Valid CMS POS they never consult a code set or call NPPES. A Luhn-valid NPI is therefore a number that could have been issued. It is not proof that one was, and it says nothing about whether the provider is still active or whether the NPI belongs to the prescriber your feed claims it does. A failure tells you something definite; a pass only narrows the possibilities.

The same limit runs down the list. A well-formed ICD-10 code from a category retired in 2019 passes. A DEA number carrying a registrant-type letter the DEA never assigned passes. Valid NCT accepts any eight digits, so on a column storing the numeric part alone, `20260815` passes.

None of this replaces Veeva Network or whatever you use for HCP mastering. MDM tells you what it could match. These tell you what arrived broken, per column, per feed, on the landing table, before a match rate has the chance to hide it. That distinction is the same one that makes testing commercial pharma data harder than anyone tells you: the failures live between the feeds, and the feed itself never errors.

Empty and null values don’t count as invalid anywhere in the pack. That’s a Required Entry question and a different test. The format checks ignore trailing spaces, so an identifier sitting in a fixed-width `CHAR` column doesn’t fail for its padding. The tests fold case first, since NCT and ICD codes come back lowercase from URLs and hand-built extracts constantly.

Why some fail and some only warn

Seven of the twelve default to Fail and five to Warning, and the line between them follows how much a failure actually proves.

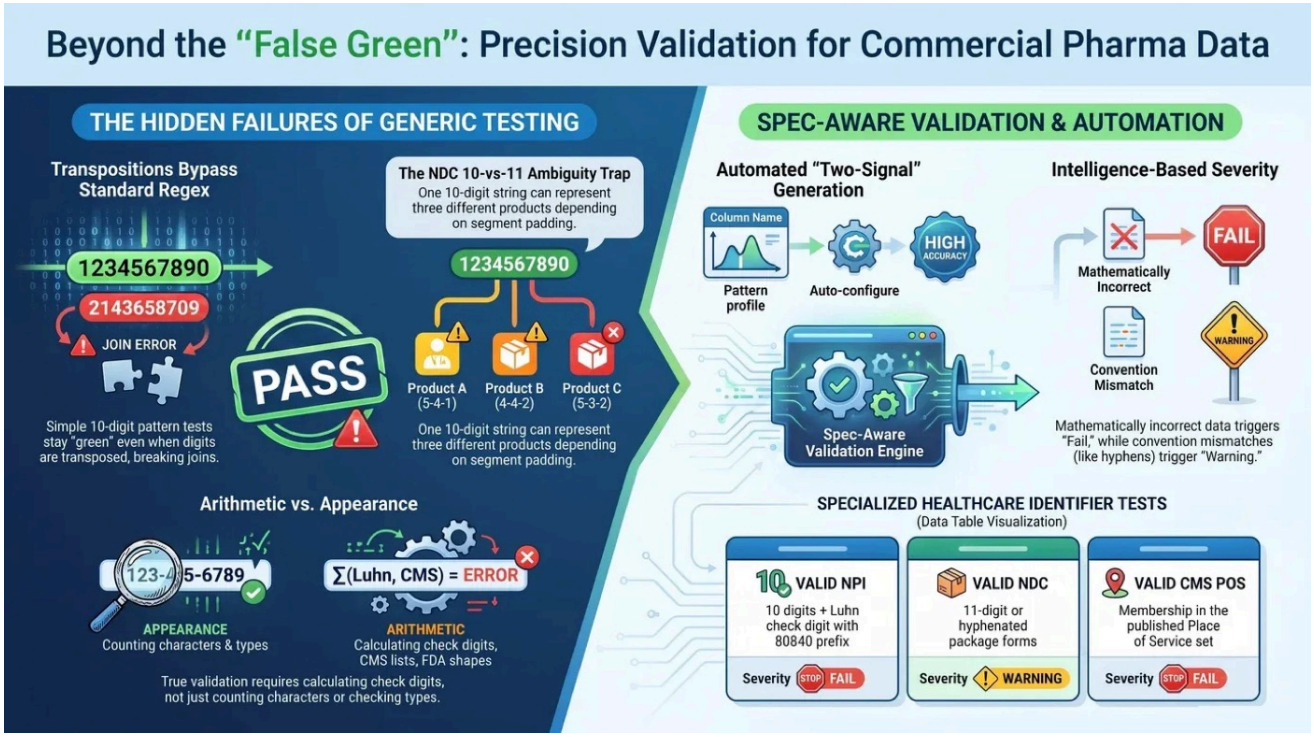
A test defaults to Fail where a bad value can only be a bad value. Valid NPI and Valid DEA run a check digit, so a failure is arithmetic rather than judgement. Valid CMS POS checks membership in a list CMS publishes, so a failure is a code that does not exist. The four ICD tests check shapes the code systems define exactly, and a value outside those shapes is malformed whatever else it is.

The rest default to Warning because a format mismatch is usually a convention difference. Your vendor sends NDCs with hyphens and your product master stores them without. Neither side is broken. Somebody just never agreed on a form. A J-code column that also carries the rest of HCPCS Level II lights up the same way, and that’s a scoping decision rather than a data defect.

Change the default on any of them. Severity is set per test definition, so if unhyphenated NDCs really do break your gross-to-net, promote that one to Fail and let it stop the run.

The whole argument on one page

Something to send to the person who owns the feed:



What a generic pattern test misses, on the left, and what the Commercial Pharma pack checks instead, on the right.

SECTION 6

AI and Data Quality

1. Sure, Go Ahead And Feed That Data To The LLM ... What Could Possibly Go Wrong?
2. The Equation For AI Success: DT + DX + CTX = 10x
3. DataOps + FITT + Data Testing = 10x Data Engineering Productivity with AI
4. Context As Infrastructure: The Six-Category Framework for AI-Ready Data Analysis
5. Stop Burning Tokens: Data Observability and Quality as Your AI's System of Record
6. Put Your AI At The Center Of Your Data Quality And Observability Decision-Making Process
7. Agents to Fix Data Quality Automatically With Control: A Design Pattern
8. The Seven Deadly Sins of AI-Generated Data Pipelines

AI AND DATA QUALITY

Sure, Go Ahead And Feed That Data To The LLM ... What Could Possibly Go Wrong?

Read the original article at: https://datakitchen.io/blog/sure-go-ahead-and-feed-that-data-to-the-llm-what-could-possibly-go-wrong/

Data teams today face a harsh reality: frequent pipeline failures, reactive fixes, and poor data quality often lead to bad decisions. Just when we thought we understood our data challenges, Large Language Models (LLMs) radically expand the use and scope of data … and not for the better. Still, hope remains with proven DataOps practices and open-source tools.

Introduction: Life in the Data Trenches

For many data teams, the day-to-day experience has become an unproductive mix of firefighting and frustration. We're constantly reacting to broken pipelines, dealing with data quality issues only after they've caused problems downstream, and settling for data that's "good enough" because we lack the time and resources to make it better. This situation was already challenging enough, but the introduction of Large Language Models has added entirely new dimensions to our struggles.

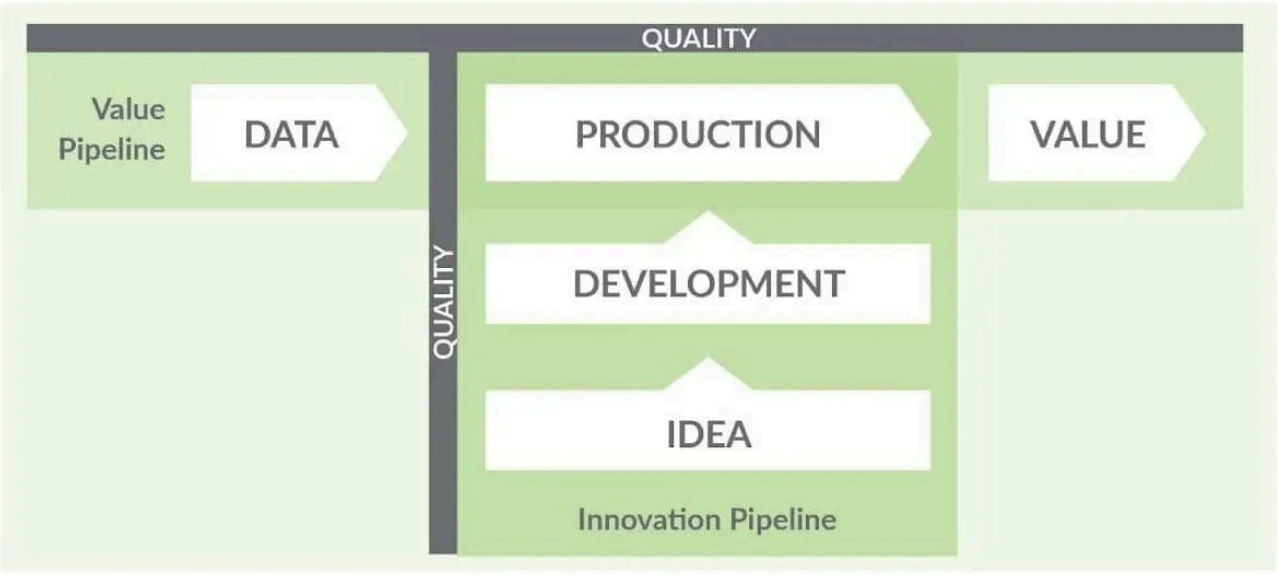
The promise of DataOps offers a way forward. By applying proven principles from agile development, lean manufacturing, and DevOps automation, we can restore sanity to our data stacks and dramatically improve the quality of both our traditional analytics and our emerging LLM-powered insights. This isn't just about adopting new tools; it's about fundamentally rethinking how we approach data quality and pipeline reliability in an age where AI is becoming ubiquitous.

The LLM Revolution: A Double-Edged Sword

Large Language Models have burst onto the scene with incredible capabilities, but they're also making the existing mess in data and analytics teams much worse. We're witnessing two simultaneous phenomena that are reshaping how organizations interact with data.

First, there's what we might call "**Analysis-A-Palooza.**" More people in more roles are now taking data directly to LLMs for analysis. Marketing managers who previously relied on pre-built dashboards are now uploading CSV files to ChatGPT or Claude. Sales representatives are asking AI to analyze their pipeline data. Executives are querying financial data using natural language. This democratization of data analysis sounds wonderful in theory, but it dramatically increases the need for extensive data quality controls. Every table, every column, every data point could be fed into an LLM, exponentially expanding the surface area for possible data quality issues.

Second, we're experiencing a "**Vibe Coding Extravaganza.**" The barrier to writing code has effectively broken down. Business analysts are using AI assistants to craft SQL queries. Product managers are generating Python scripts. Even executives are creating data transformations using LLMs. While this democratization of coding skills seems empowering, it results in more code being developed and deployed by people who might not fully grasp the implications of what they're building. This amplifies the critical need for end-to-end regression testing and thorough data quality coverage.

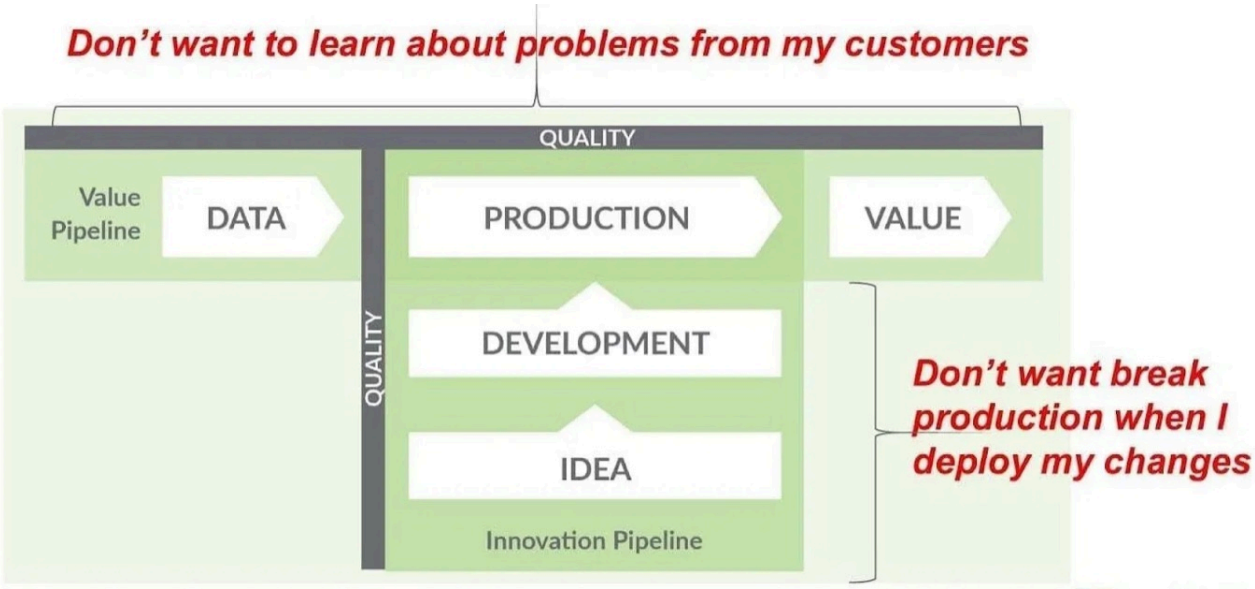


Ye Olde DataOps

Before we dive deeper into how LLMs are changing the game, it's essential to understand the DataOps framework that provides our foundation for addressing these challenges. DataOps recognizes two key pipelines in any data organization: the Value Pipeline and the Innovation Pipeline.

The **Value Pipeline** represents the flow from raw data through production systems to delivered value for customers and stakeholders. This is where your ETL processes, data warehouses, and reporting systems live. It's the backbone of your data operations, and any failure here directly impacts business operations.

The **Innovation Pipeline** represents the journey from ideas through development to production deployment. This pipeline is where new analytics capabilities are born, tested, and refined before being promoted to production. The key insight of DataOps is that these two pipelines must be managed with equal rigor, with quality gates at every stage.



What DataOps helps you avoid is learning about problems from your customers, which is both embarrassing and expensive. It also prevents the all-too-common scenario of breaking production during deployments. By implementing proper testing and monitoring throughout both pipelines, you can catch issues early when they're cheap and easy to fix.

Welcome To The 'Analysis-A-Palooza:' LLMs Directly Analyzing Data

The proliferation of AI and LLM usage across organizations has fundamentally changed the data quality equation. Previously, data teams could focus their quality efforts on critical datasets and key metrics that powered executive dashboards and operational reports. They knew which tables mattered most and which columns were actually being used. This allowed for targeted quality efforts on high-impact data.

That comfortable situation is now history. With LLMs, users are analyzing data from every corner of the data warehouse. That obscure table from a system sunset three years ago? Someone just fed it to ChatGPT for historical analysis. Those columns that were supposedly "just for reference"? They're now

being used to train custom models. Every single piece of data in your organization has become potentially critical.

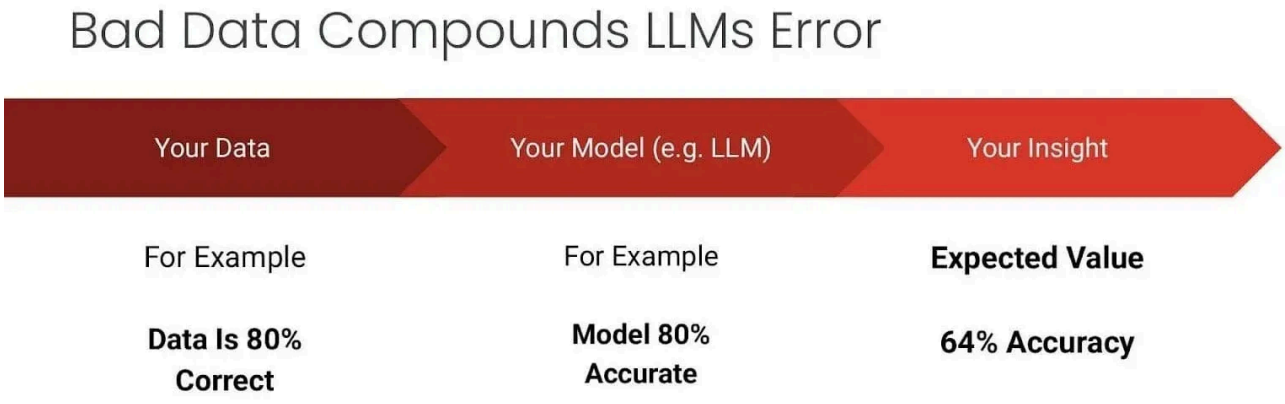
This dramatic expansion of the data usage surface area means we need a fundamentally different approach to test coverage. Manual testing strategies that might have worked when we had a dozen critical tables simply don't scale when an AI model can access every table and column at any time. Organizations need comprehensive test coverage across all levels and zones in their databases and data lakes. Every table should have tests for consistency, volume, freshness, and schema. Every column should be monitored for quality drift. Every significant business metric needs domain-specific validation.

The challenge becomes even more complex when you consider that different LLM use cases may interpret data quality issues differently. A null value that's acceptable for a summary report might cause an LLM to hallucinate wildly. A slight schema change that goes unnoticed in traditional analytics could

completely break an AI model's ability to process information correctly.

How Using LLMs To Analyze Data Compounds Data Quality Challenges

Here's a sobering reality that many organizations are just beginning to grasp: LLMs don't magically fix insufficient data; they amplify its impact. Consider a typical scenario where your data is 80% correct and your LLM is 80% accurate. Simple multiplication tells us that your expected output accuracy drops to just 64%. This compounding effect means that data quality issues that might have been tolerable in traditional analytics become catastrophic when fed through AI systems.



The implications are significant. Is that customer segmentation with a few misclassified records? When an LLM uses it to create personalized marketing campaigns, those mistakes can multiply across every piece of generated content. Is that a financial dataset with some missing values? When an AI analyzes it for strategic insights, those gaps can lead to fundamentally flawed recommendations.

This multiplication effect makes the traditional "we'll fix it if someone complains" approach to data quality completely untenable in the age of AI. We need proactive, comprehensive data quality strategies that catch issues before they're amplified by machine learning models or gobbled up by an LLM.

Data Quality Test Coverage Reduces Problems Created By LLM-Wielding Analysts

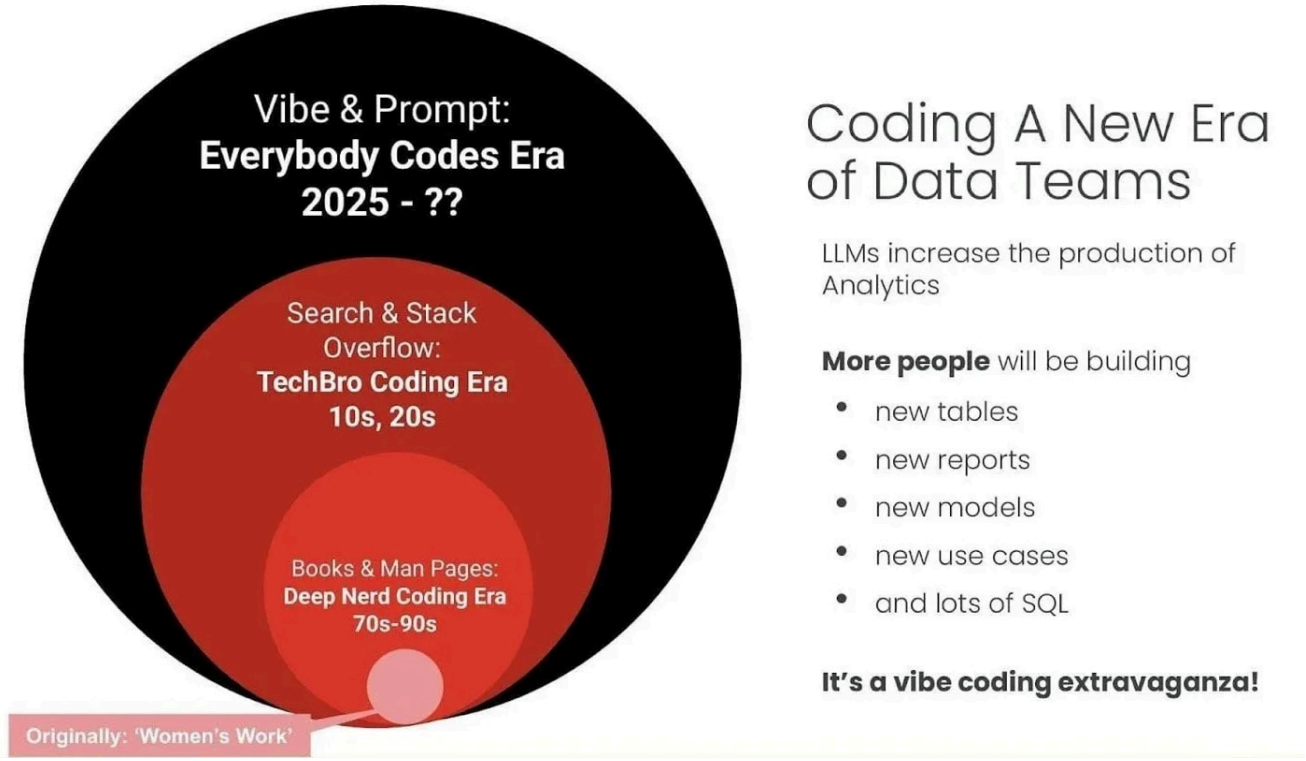
The explosion of LLM adoption has fundamentally changed the stakes for data quality test coverage. When organizations feed untested data directly into LLMs for analysis, they're essentially compounding two layers of uncertainty: the inherent probabilistic nature of LLM outputs and the unknown quality issues lurking in their data. LLMs can't distinguish between accurate and corrupted data - they'll confidently analyze incomplete datasets, outdated records, or incorrectly transformed fields as if they were pristine. This creates a hazardous scenario where sophisticated-sounding AI analysis masks underlying data problems, leading to decisions based on what appears to be cutting-edge intelligence but is actually sophisticated garbage. The hallucination problem that already plagues LLMs becomes exponentially worse when the training or analysis data itself contains errors, duplicates, or inconsistencies. The shift-left approach to data quality - testing data as early as possible in the pipeline rather than waiting until it reaches production - becomes critical when that production endpoint is an LLM that will amplify any data issues into convincing but flawed narratives.

Comprehensive test coverage acts as the critical safety net between raw data and LLM consumption. By implementing thorough validation checks across data pipelines - including schema validation, referential integrity tests, statistical distribution monitoring, and business rule verification - organizations can ensure that LLMs work with trustworthy inputs. This means testing not just the final data products, but every transformation step, every join operation, and every aggregation that shapes the data before it reaches the AI system. To achieve meaningful coverage, data teams typically need at least 2-3 quality tests per column and table. For a typical analytics database with hundreds of tables and thousands of columns, this translates to thousands of individual tests - a scale that's impossible to achieve through manual test creation. This is why **automated test generation tools** have become essential, using profiling and machine learning to automatically suggest and create appropriate tests based on the data's characteristics, historical patterns, and detected anomalies.

Inadequate test coverage for LLM-driven analysis risks catastrophic, often invisible, business impacts, such as AI making recommendations based on duplicate, stale, or incomplete data, presented with false authority. Adopting a shift-left approach catches issues cheaply at the source. Modern, automated data quality tools can rapidly profile data and generate tests, providing the necessary coverage density for LLM readiness. Conversely, robust, automated testing enables confident LLM scaling, with quality gates preventing bad data from propagating into AI outputs. Investing in comprehensive, automated testing not only prevents costly errors but also accelerates AI adoption by ensuring a solid data foundation.

Welcome To the Vibe Coding Extravaganza

We've entered what might be called the "Vibe Coding Era," a fundamental shift in who creates code and how it's created. The evolution is striking. We've moved from the "Deep Nerd Coding Era" of the 70s-90s, where programming required deep technical knowledge and manual consultation, through the "TechBro Coding Era" of the 2010s and 2020s, dominated by Stack Overflow and search-driven development, to today's "Everybody Codes Era" where natural language prompts generate functional code.



This democratization seems wonderful at first glance. Business users can now create their own data transformations. Analysts can write complex SQL without years of training. Managers can build Python scripts to automate their workflows. But this explosion of citizen developers brings serious challenges.

Consider how LLMs approach SQL generation. Recent benchmarks using Spider 2.0, a comprehensive test suite for SQL generation, reveal sobering limitations. Even the most advanced models struggle with real-world database queries. Spider 2.0-Snow peaks at just 59.05% accuracy, while Spider 2.0-DBT tops out at a mere 39.71%. These aren't edge cases or particularly complex queries; they're representative of actual business questions that organizations need to answer daily.

The fundamental issue is that LLMs can generate syntactically correct SQL that looks perfect but fails spectacularly when run against actual data. They don't understand your specific data models, business rules, or the inevitable quirks and exceptions that exist in every real-world database. They can't intuit that the "customer_type" field uses different codes in different tables, or that certain date ranges have known data quality issues.

This gap between apparent capability and actual performance creates a dangerous situation. Users gain false confidence from AI-generated code that works on sample data but breaks in production. They create tables and transformations that appear functional but introduce subtle errors that compound over time. They build analytics pipelines that seem sophisticated but lack proper error handling, testing, or documentation.

Metadata Context: Not A Miracle Cure For Vibe Coding

The key to improving LLM performance in data tasks lies in providing proper context. AI generates code that looks right but fails because it lacks an understanding of your actual data ecosystem. It doesn't know about your business logic, data relationships, or the countless unwritten rules that govern your data.

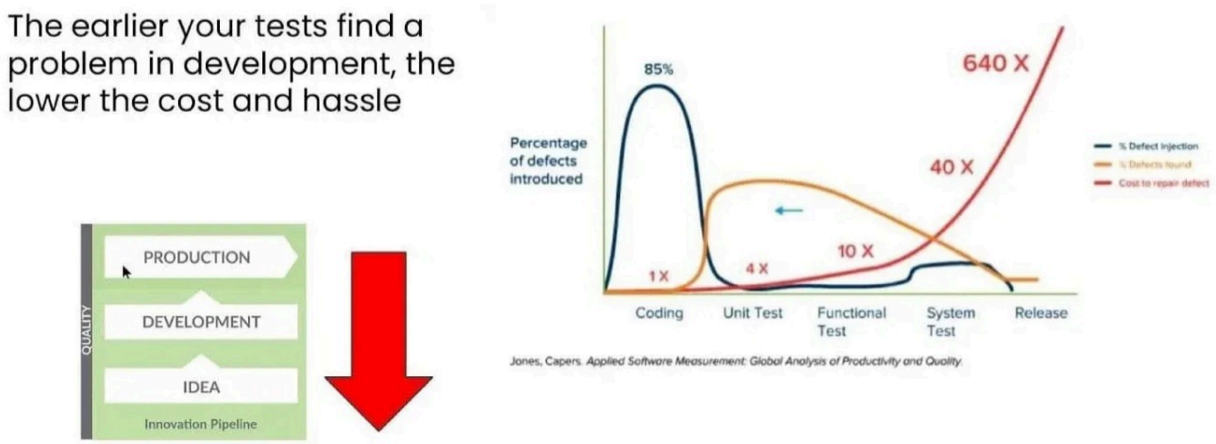
"Data Context" formalizes this hidden knowledge, making it accessible to AI systems. This includes data profiling information that reveals actual value distributions and patterns, data catalog entries that document business meanings and relationships, data quality test results that highlight known issues and constraints, and institutional knowledge about how data should and shouldn't be used.

When LLMs have access to this context, their accuracy improves dramatically. Instead of guessing that a field contains customer IDs, they know the exact format and valid ranges. Instead of assuming standard SQL will work, they understand which database-specific syntax to use. Instead of treating all data as equally reliable, they can account for known quality issues.

However, context isn't a miracle cure. Even with perfect context, we should expect more code to be created by more people, and much of it will be of lower quality than what experienced developers would produce. This reality makes comprehensive testing and monitoring even more critical.

Shift-Down Regression Testing Makes Vibe Coding Sustainable

Just as shift-left principles tell us to catch data quality issues early in the data pipeline, “shift-down” principles tell us to catch code and logic errors early in the development process. The economics are equally compelling. Issues caught during development might take minutes to fix. The same issues discovered in production could require hours of debugging, data cleanup, and stakeholder management.



This shift-down approach requires comprehensive end-to-end testing environments that allow developers to validate their changes before promotion to production. It means having test data that accurately represents production scenarios. It requires automated testing frameworks that can validate not just code syntax but actual business logic and data transformations.

In the age of vibe coding, where an AI might generate dozens of SQL queries in an afternoon, manual code review and testing simply don’t scale. Organizations need automated regression testing that can validate every change across the entire data pipeline. They need development environments that closely mirror production to catch integration issues. They need testing frameworks that can validate not just whether code runs, but whether it produces correct results.

Vibe Coding Demands Comprehensive Test Coverage

The path forward requires implementing comprehensive test coverage across both data and code. This isn’t just about adding a few unit tests or spot-checking critical tables. It requires a systematic approach that covers every layer of the data stack.

For data quality, organizations need automated test generation that can create thousands of tests across all tables and columns. Manual test writing doesn’t scale when dealing with modern data volumes. A junior operator using automated tools can generate 2,500 tests in two steps that would take a trained data engineer over seven months to create manually.

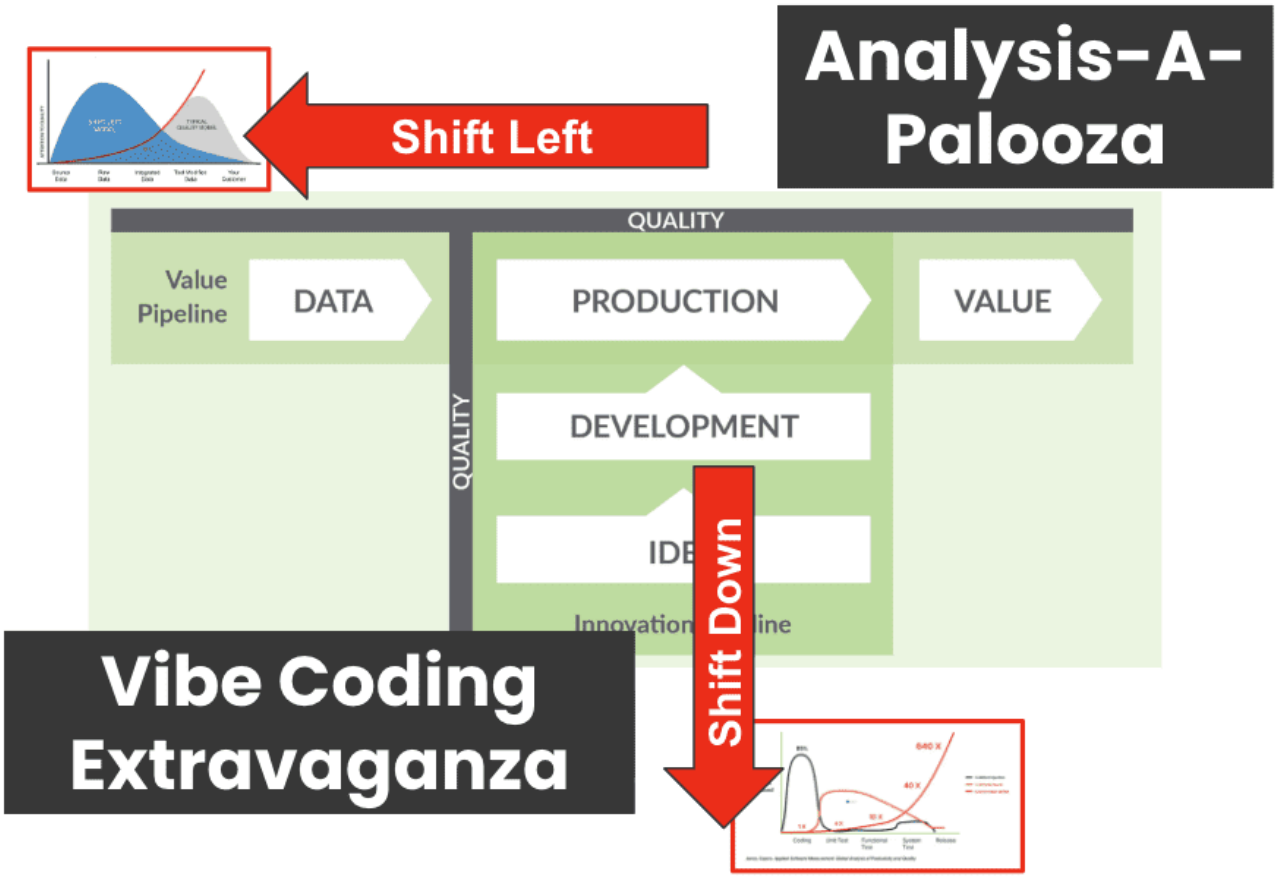
For code quality, especially in the era of AI-generated code, organizations need regression testing that validates the entire data pipeline from source to consumption. This includes testing data access layers, transformation logic, model code, visualization updates, and report generation. Every component that touches data must be validated.

The key insight is that testing can no longer be an afterthought or a “nice to have” when time permits. In a world where LLMs are increasing both data usage and code creation, comprehensive testing becomes the foundation of reliable data operations.

Conclusion: When Everyone Uses An LLM, Only DataOps Thinking Will Tame The Beast

The challenges we’ve discussed require new tools designed specifically for the AI age. Traditional data quality tools that spot-check a few tables won’t suffice when every piece of data might be fed into an LLM. Traditional code review processes break down when AI can generate more code in an hour than a human can review in a day.

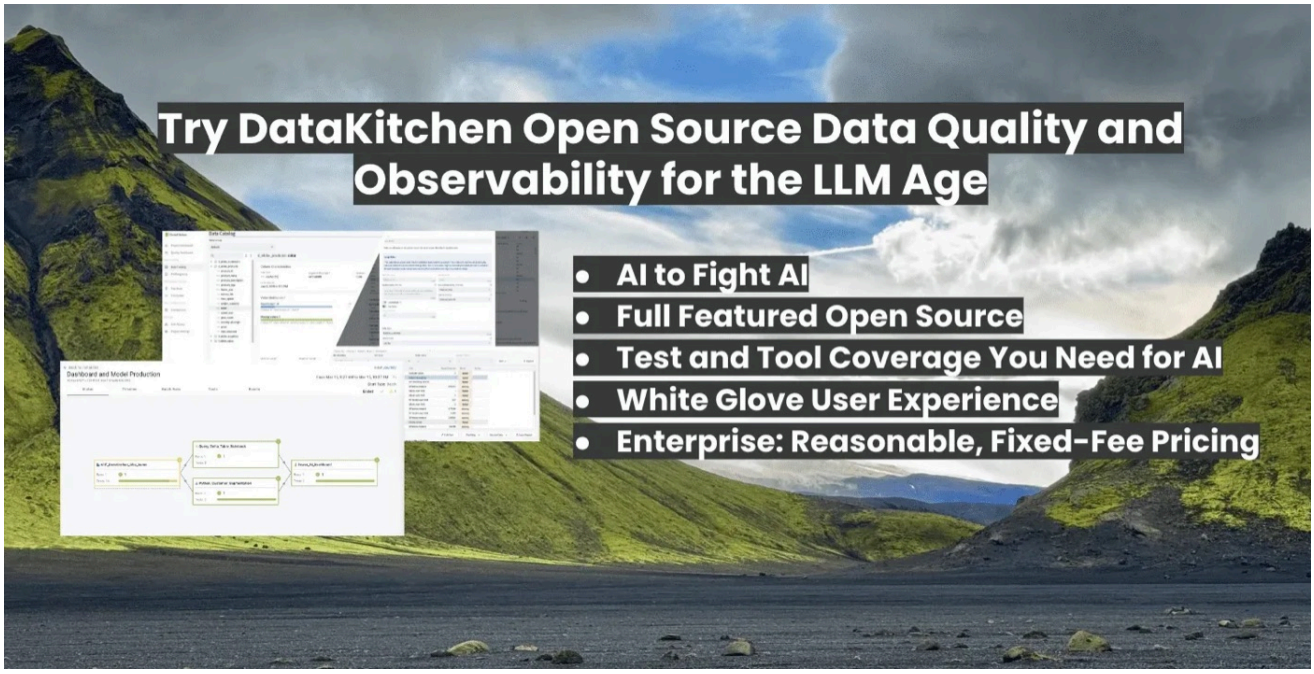
Modern data quality platforms need to provide AI-powered test generation that can automatically create comprehensive test suites across entire data estates. They need anomaly detection that can identify issues without explicit rule definition. They need data profiling that goes beyond simple statistics to understand the actual behavior and relationships.



Apply DataOps Principles for Success In The LLM Age

The solution to the Analysis-A-Palooza and Vibe Coding Extravaganza isn't to restrict LLM usage or return to centralized data control. The democratization of data analysis and code creation is here to stay, and organizations that embrace it while managing its challenges will have a significant competitive advantage.

The path forward requires embracing DataOps principles while adapting them for the AI age. This means implementing shift-left data quality practices that catch issues at the source before AI systems amplify them. It means adopting shift-down testing approaches that validate code and transformations in development before they impact production. It means comprehensive test coverage that accounts for the dramatically expanded surface area of data usage. It means providing rich context to AI systems while maintaining robust validation of their outputs.



Organizations that successfully navigate this transition will find themselves with data operations that are more robust, scalable, and valuable than ever before. They'll be able to leverage the full power of LLMs while maintaining data quality and reliability. They'll empower citizen developers while preventing chaos in production systems.

The integration of Large Language Models into data operations represents both an enormous opportunity and a significant challenge. LLMs are making our existing data quality and code quality issues worse by amplifying their impact and dramatically expanding who interacts with data and how they do it. But by applying DataOps principles, implementing comprehensive testing strategies, and adopting tools designed for the AI age, organizations can turn this challenge into a competitive advantage.

The key is to stop treating data quality and data observability as optional, nice-to-have add-ons and start treating them as fundamental requirements for AI success. Just as you wouldn't feed your body junk food and expect optimal performance, you can't feed your AI systems low-quality data and expect reliable

insights. By focusing on data quality early, reducing testing gaps, and ensuring comprehensive coverage across your entire data estate, you can ensure your AI systems receive the high-quality, nutritious data they need to thrive.

AI AND DATA QUALITY

The Equation For AI Success: DT + DX + CTX = 10x

Read the original article at: https://datakitchen.io/blog/the-equation-for-ai-success-dt-dx-ctx-10x/

Everyone wants AI to answer business questions with their data. The reality is most AI assistants built on enterprise data give mediocre, unreliable, or outright wrong answers , not because the AI is bad, but because the foundation underneath it is.

Your AI chatbot is lying to you. Not maliciously. Confidently. It queries your data warehouse, assembles an answer, and presents it with the same tone whether the underlying data is fresh, tested, and correctly joined, or three days stale, never validated, and pulled from a deprecated table nobody deleted in 2021.

This is the AI-on-data problem nobody wants to say out loud: the model is fine. The data underneath it is the disaster. Every hallucination, every wrong answer, every time an analyst says “I don’t trust what the chatbot told me” , that is a data infrastructure failure dressed up as an AI failure.

Everyone wants AI to answer business questions with their data.

Most AI assistants give mediocre or wrong answers. Not because the AI is bad, but because the foundation underneath it is.

The gap is a data problem. Not a model problem.
AI models are amplifiers—if you feed them untrustworthy data, they amplify bad answers at scale.

Everyone assumes AI fails at data analysis because the models are not smart enough. The MIT BEAVER study (September 2024) decisively disproves that assumption. When researchers tested leading LLMs against real enterprise data warehouses, models that had scored above 85% accuracy on standard benchmarks collapsed to nearly zero end-to-end execution accuracy on actual business data. The model did not get dumber. The context disappeared. Enterprise schemas were too complex, business questions required joins across many tables and nested aggregations, and the models had never encountered the private data they were suddenly being asked to reason about. When researchers simply handed the models the correct table and column context, performance recovered sharply. The intelligence was there the whole time. What was missing was the foundation underneath it.

OpenAI's look insideits internal data agent (January 2026) reveals something subtle but important: raw data alone is not enough to produce useful insights. Their teams do not just query tables; they ask questions that are grounded in context, such as product launches, feature changes, and user behavior patterns, and the system uses that surrounding knowledge to interpret results. The same metric can mean very different things depending on what was happening in the business at the time, which is why simple SQL outputs often fall short. By combining data with context, their approach turns analysis into something closer to reasoning rather than reporting. The result is that the same data that once produced ambiguity now produces insight, dramatically reducing errors across thousands of employees working on tens of thousands of datasets. It highlights a fundamental truth about data engineering: raw data alone is not enough. The intelligence was already there, but without context, it was incomplete. What was missing was the foundation that connects data to meaning.

A comprehensive survey of LLM-based text-to-SQL systems published on arXiv (updated through late 2025) finds that generating accurate SQL from natural language remains a long-standing challenge, specifically because of the complexities involved in database schema comprehension, and that the problem compounds as databases grow more complex. A separate ACM Computing Surveys paper on LLMs for text-to-SQL tasks found that domain-specific knowledge is critical, and that performance

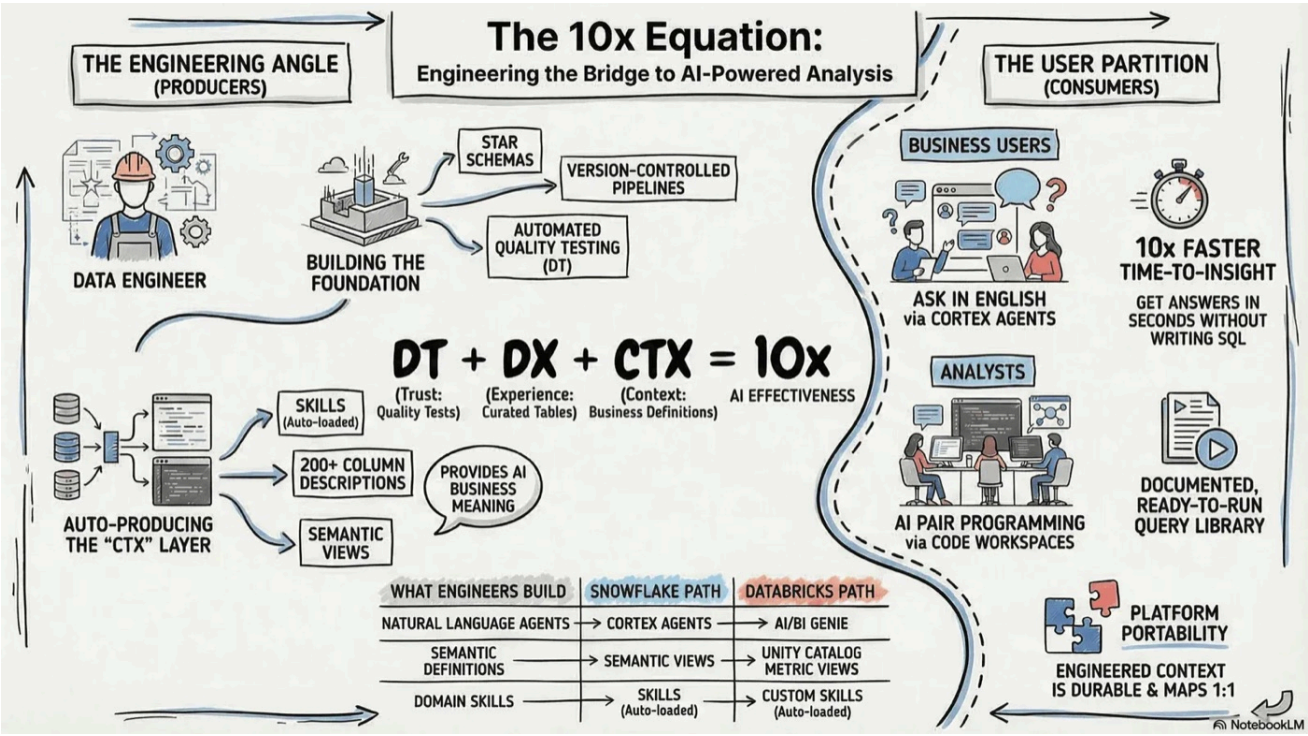
degrades significantly without additional SQL knowledge and external commonsense context. A 2025 arXiv paper on long-context LLMs for natural language to SQL demonstrated empirically that additional contextual information about column references, user-provided hints for clarification, and schema metadata all directly improve execution accuracy, with smaller models benefiting the most from richer context. A 2025 survey from the Journal of the American Statistical Association on LLM-based data agents echoes this theme, noting that LLM-powered data science agents have shown significant potential to transform data analysis, but their role depends on how well the system simplifies complex data tasks for users without deep expertise in the underlying schemas.

The gap comes down to three things. Your team does not fully trust the data the AI is querying. The schema the AI navigates is a graveyard of 150 tables, half of them wrong for the question being asked. And the AI has no idea what your business actually means by “equalized prescriptions” or “territory alignment.” It is a guess. It is always guessing.

Fix those three things, and the picture changes. Not incrementally. Fundamentally. We call it the 10x equation:

DT + DX + CTX = 10x Faster Data Analyis With AI

Data Trust. Data Experience. Context. This is not a theory. It is a checklist for the infrastructure work that makes AI analysis reliable, accurate, and genuinely useful. This is our experience with customers solving this challenge. Let’s break it down.



DT , Data Trust

The first term in the equation is the most foundational: your team must be able to trust the data before AI can use it reliably. Data Trust is not about having perfect data , it’s about having tested, monitored, and understood data, delivered to the right people at the right time.

What Data Trust Means in Practice

Data Trust has four components that work together:

- **Quality & Testing**: Data quality and testing , Every critical dataset has automated tests validating completeness, freshness, referential integrity, value ranges, and statistical distributions. If a column is supposed to have non-null values, there is a test that fails loudly when it doesn’t.
- **Timeliness**: Arrival time and delivery logistics , The right data arrives on time. Late data is wrong data. If the AI is answering questions about yesterday’s sales using 3-day-old data, every answer is wrong, regardless of the model’s quality.
- **Team Trust**: Team trust , Humans using AI outputs need confidence that the underlying data is reliable. This is cultural as much as technical. A team that has burned bad data will second-guess every AI answer , often correctly.
- **The Intersection**: Right data, right quality, right time , Data Trust is the intersection of all three: the right dataset, verified to an acceptable quality threshold, available when the business needs it.

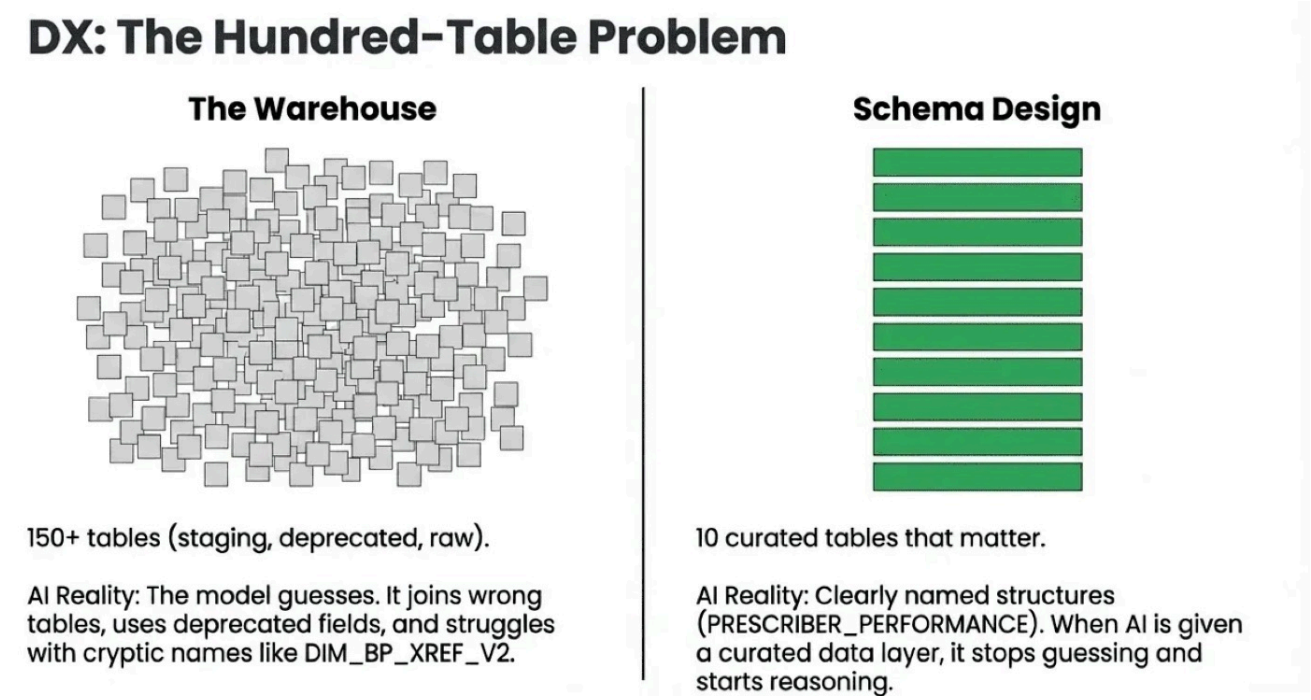
Why This Is the Foundation

AI models are amplifiers. They make whatever is underneath them faster and more accessible. If what is underneath them is untrustworthy data, AI amplifies bad answers at scale and speed. Data Trust is the prerequisite for everything else in the equation.

Tools like TestGen provide over 120 AI-driven data quality tests that can be applied to any dataset. They automatically profile schemas, flag anomalies, and surface data issues before they reach analysts or AI systems. This foundation is the infrastructure of Data Trust.

DX , Data Experience

Even if your data is trustworthy, AI and analysts alike will struggle if the experience of working with that data is confusing. Data Experience (DX) is about making data legible , a schema design that humans and AI can both read, understand, and use without becoming database archaeologists.



The Hundred-Table Problem

Most data warehouses and lakehouses have accumulated tables over years of work: staging tables, intermediate tables, deprecated views, experimental datasets, and duplicates from three different ETL eras. Nobody deletes anything.

When an AI assistant is given access to a data environment with 150 tables, several bad things happen: The model has to guess which tables are relevant to any given question. It will frequently choose the wrong option, joining tables that are technically related but semantically incorrect. Even when it chooses right, it may use deprecated fields, wrong grain, or mismatched time zones. Performance degrades because the AI explores a vast, confusing space rather than a focused, curated one.

The Solution: Ten Tables That Matter

A good data experience means curating the data landscape. Instead of exposing every table in your warehouse, you build a semantic layer , ten (or twenty, or thirty) clearly named, well-structured tables that represent the core business concepts an analyst or AI needs.

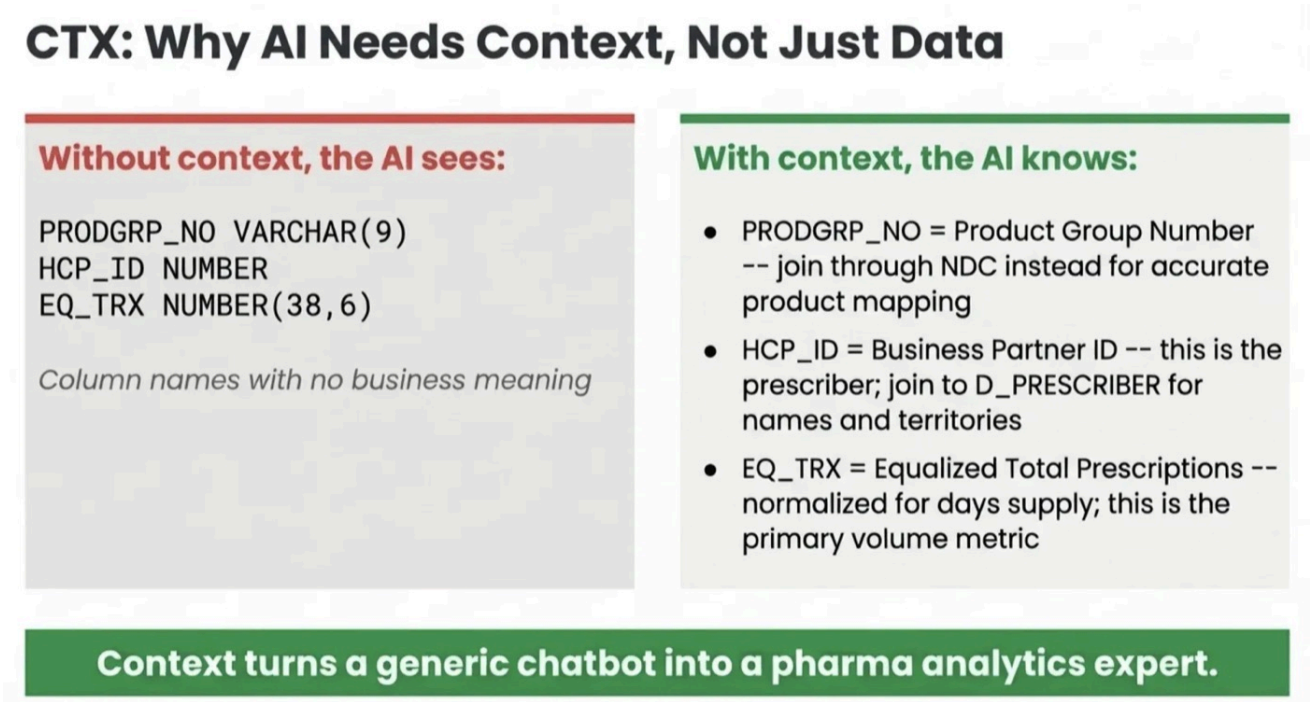
This is not about hiding data. It is about creating a front door. The warehouse can have 150 tables; the AI and the analyst should see the 10 that matter for their domain, with names and structures that make sense.

Good Data Experience looks like: Table names that say what they contain: PRESCRIBER_PERFORMANCE, not DIM_HCP_XREF_V2. Columns named for business concepts: EQUALIZED_PRESCRIPTIONS, not EQ_TRX. Consistent grain and join keys are documented explicitly. No ambiguous foreign keys , if there are multiple ways to join two tables, the right way is marked.

When AI is given a well-structured, curated data layer, it stops guessing and starts reasoning. That is a qualitative shift in output quality.

CTX , Context

Context is the multiplier. It is the difference between an AI assistant that can technically query your database and one that actually understands your business. Context (CTX) is data about data , the accumulated knowledge that turns a generic language model into a domain expert.



Why AI Needs Context, Not Just Data

Here is the core problem. When you connect an AI to a data warehouse, it sees this:

```
PRODGRP_NO    VARCHAR(9)

HCP_ID         NUMBER

EQ_TRX        NUMBER(38,6)
```

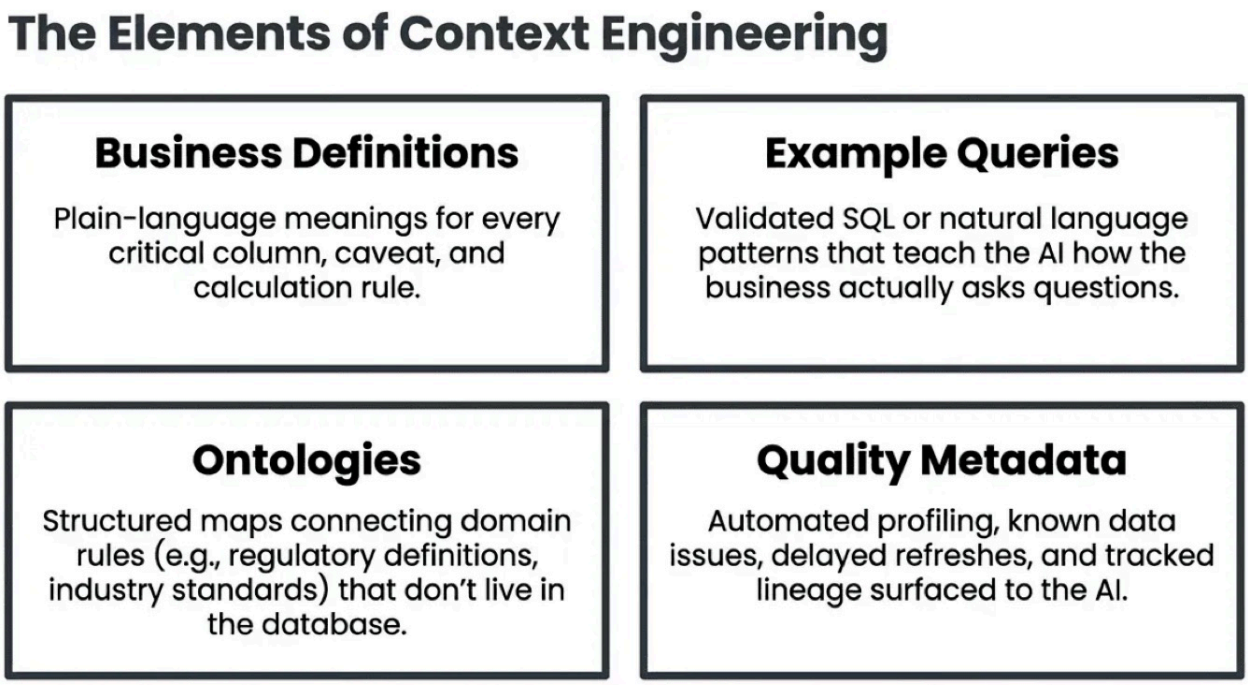
Three columns. Three cryptic names. No business meaning. The AI has no idea what these columns are, what they measure, how they relate to other tables, or what assumptions apply to their values.

Without context, the AI is making educated guesses dressed up as answers.

Apply Context to the same data. Completely different capability. The AI now knows what it's looking at, how to join it correctly, and what the values mean. Context turns a generic chatbot into a pharma analytics expert.

What Context Includes

Context is a broad category that covers several types of information:



Every column that matters should have a plain-language definition, the business question it answers, and any caveats about how it is calculated or when it is valid. This can live in a data catalog, a README, or a structured metadata layer.

One of the most effective ways to provide context is through example queries , SQL or natural-language queries that have been validated and annotated with the correct answer. These help teach the AI how your business asks questions and what proper answers look like.

“What were total equalized prescriptions for product group X in Q3?” → annotated SQL with correct join path.

“Which territories exceeded their prescription targets last month?” → annotated SQL with correct grain and filter logic.

In industries like pharma, finance, or healthcare, there are layers of domain knowledge that aren’t captured in the data itself: regulatory definitions, industry standards, and business rules that arise from how the market operates rather than how the database is structured.

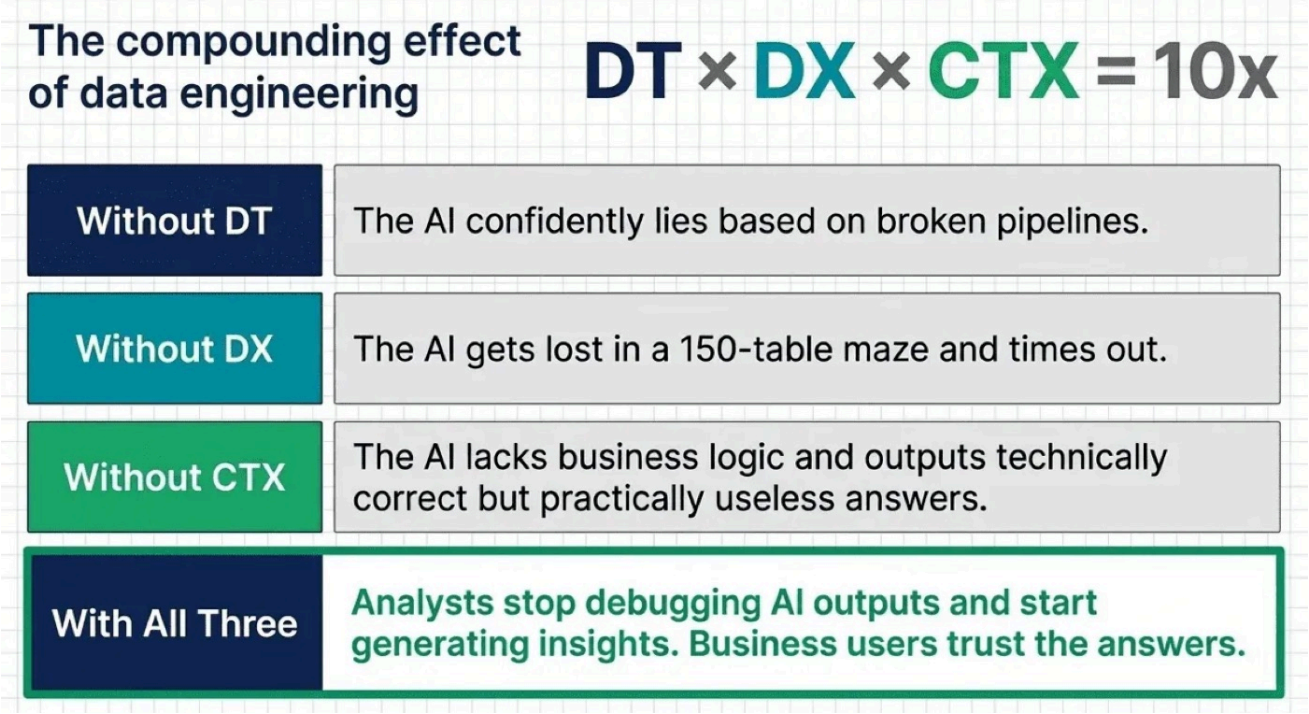
Providing the AI with an ontology , a structured map of how business concepts connect , greatly enhances the accuracy of answers to domain-specific questions.

Context also includes what you know about data quality. If a specific field has known data issues on certain dates, that counts as context. If a table’s refresh is delayed on Mondays, that is the context. If a column is deprecated and shouldn’t be used for new analysis, that is the context. DataOps Observability automatically surfaces this kind of metadata , tracking pipeline health, data freshness, test results, and lineage , and can feed it into AI context layers so the AI understands what it can and cannot rely on.

Automated schema profiling captures the statistical shape of your data: value distributions, null rates, cardinality, min/max ranges, and update frequency. Lineage traces the source of data and the transformations it has undergone. Both are essential contexts for an AI that is reasoning about data reliability and joins.

Putting It Together: The 10x Effect

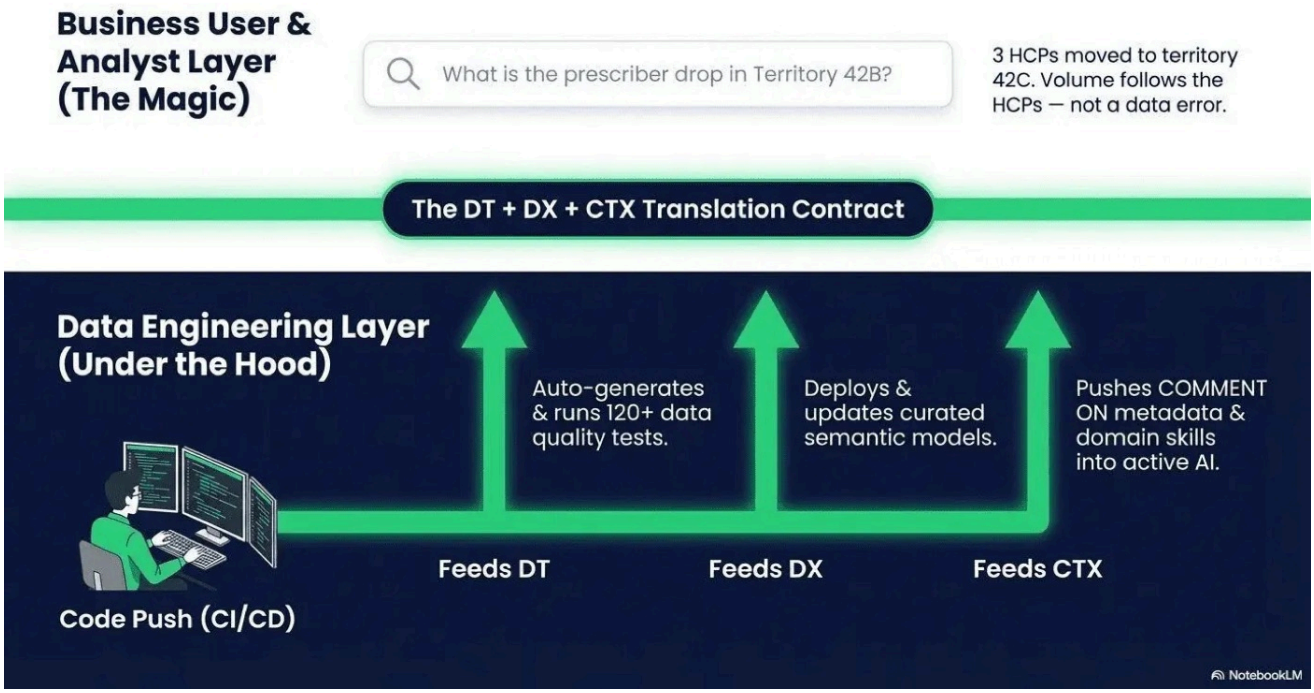
Individually, DT, DX, and CTX each enhance AI-powered data analysis. Together, they create a multiplying effect. A team with Data Trust knows its AI answers are based on trustworthy data. A team with strong Data Experience ensures the AI is working with the correct tables and columns. A team that has invested in Context understands the business well enough for the AI to ask the right questions. The result isn’t just slightly better , it’s fundamentally different. Analysts spend less time validating AI outputs and more time generating insights. Business users feel more confident trusting AI answers and start developing workflows that depend on them.



A Note for Data Engineering Teams

The work described here , building data quality tests, curating semantic layers, writing business definitions, maintaining ontologies , is data engineering work. It is not glamorous. It does not ship in a sprint. But it is the infrastructure on which AI-powered analytics is built.

Every hour invested in DT, DX, and CTX reduces the hours spent debugging AI outputs by an order of magnitude. The equation is as much about return on investment as it is about architecture.



Getting Started

You don't have to solve all three dimensions at once. Start where the pain is greatest:

Start with DT: If your team doesn't trust the data, start with DT. Run automated quality tests on your most critical tables. Use TestGen or a similar tool to profile your schemas and surface issues.

Start with DX: If analysts are wasting time navigating a confusing data landscape, begin there. Identify the ten tables that answer 80% of your business questions and build a clean semantic layer over them.

Start with CTX: If your AI gives technically correct but business-wise wrong answers, start with CTX. Write business definitions for your most important columns. Add five example queries. Build a data dictionary for the domain in which your AI operates.

The equation does not require perfection in any dimension. It requires meaningful progress in all three. A little more trust, a cleaner experience, and richer context compound into dramatically better AI-powered analysis.

How to diagnose and start fixing your data foundation

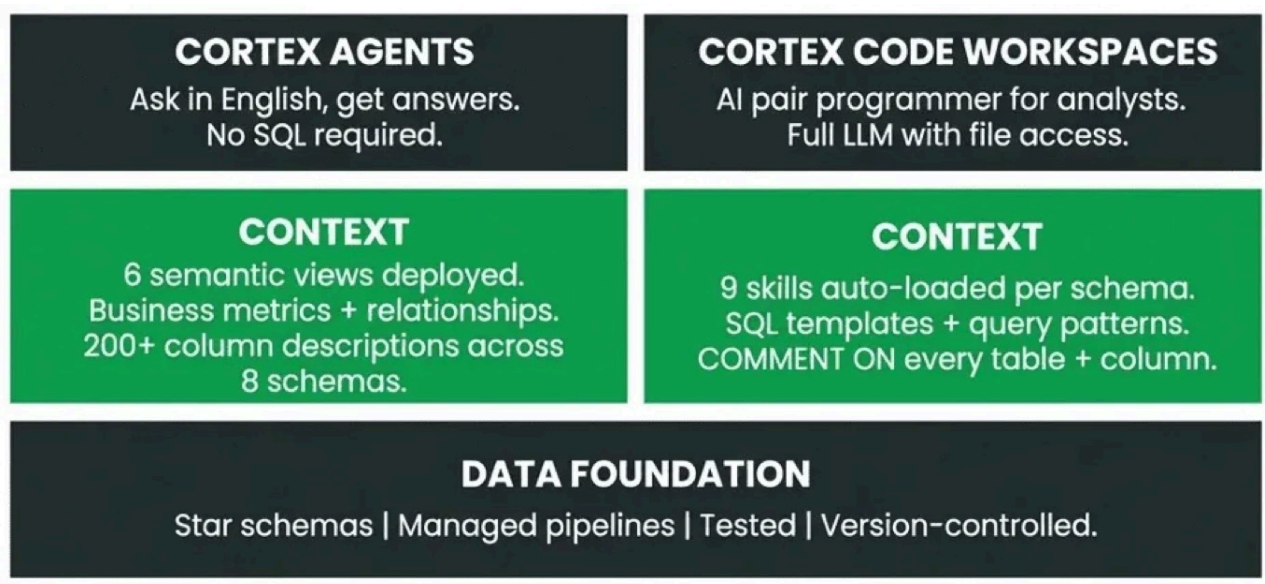
DT	**Symptom:** Your team doesn't trust the AI's answers.	**Diagnosis -> First Step:** Start with DT. Run automated quality tests on your top critical tables to establish a baseline.
DX	**Symptom:** Analysts waste hours finding the right data.	**Diagnosis -> First Step:** Start with DX. Identify the 10 tables that answer 80% of questions and build a clean semantic front door.
CTX	**Symptom:** AI gives technically correct but business-wrong answers.	**Diagnosis -> First Step:** Start with CTX. Write clear business definitions for top columns and add 5 verified example queries.

Real-World Example: Commercial Pharma Analytics on Snowflake & Databricks

The DT + DX + CTX framework is not abstract. Here is what it looks like deployed against a real pharma data environment running on Snowflake, with IQVIA data, field sales teams, and analysts who need answers without writing SQL.

The Architecture: Two Paths, One Foundation

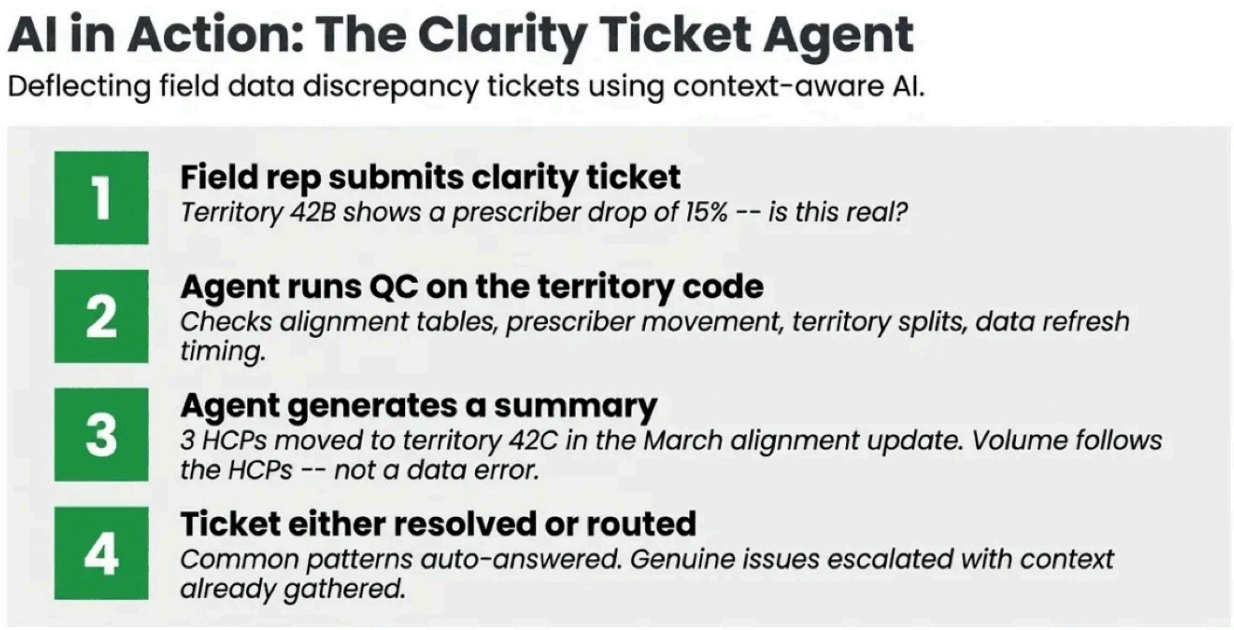
When context engineering is done properly, it enables two distinct AI-powered analytics modes to run simultaneously , both on the same underlying data foundation.



Both paths run on the same data foundation , star schemas, managed pipelines, tested and version-controlled. Context is what makes both accurate. Both paths are running simultaneously on production data.

Example: The Clarity Ticket Agent

With the right data foundation, context, and AI tooling in place, entirely new workflows become possible. Here is one concrete use case: an agent that automatically handles tickets for field data discrepancies.



Possible Phase 1 target: pilot focused on clarity tickets as first agent use case.

Field reps routinely submit tickets when the data does not match their expectations , a territory showing a sudden drop in prescribers, a product group with anomalous volume. This report does not reconcile with what they see in the field. Today, these tickets are routed to analysts. With a context-aware AI agent, many can be resolved instantly:

This agent is possible only because the data underneath it is trusted (DT), the schema is navigable (DX), and the context layer understands what territory alignment tables mean, how HCP movement is recorded, and what data refresh timing looks like (CTX). Without all three, the agent would hallucinate or confidently give wrong answers.

What the Team Actually Gets

When DT + DX + CTX is in place, the benefits are not uniform across the organization , they are specific to each type of user:

What Your Team Actually Gets

Business Users

Sales Ops, Market Access, Brand Analytics, Managed Care

- Ask questions in plain English
- Get answers in seconds
- No SQL required
- Self-serve via Snowsight browser

Analysts & Data Teams

- AI-powered workspaces with Cortex Code that knows your schemas
- Ready-to-run query library
- Documented business definitions in the platform
- Fewer "can you run this for me?" interruptions

Leadership

- Institutional knowledge captured in the platform
- Faster time-to-insight for market questions
- Foundation for advanced analytics
- AI adoption that meets users where they are

Context Engineering Is Platform-Portable

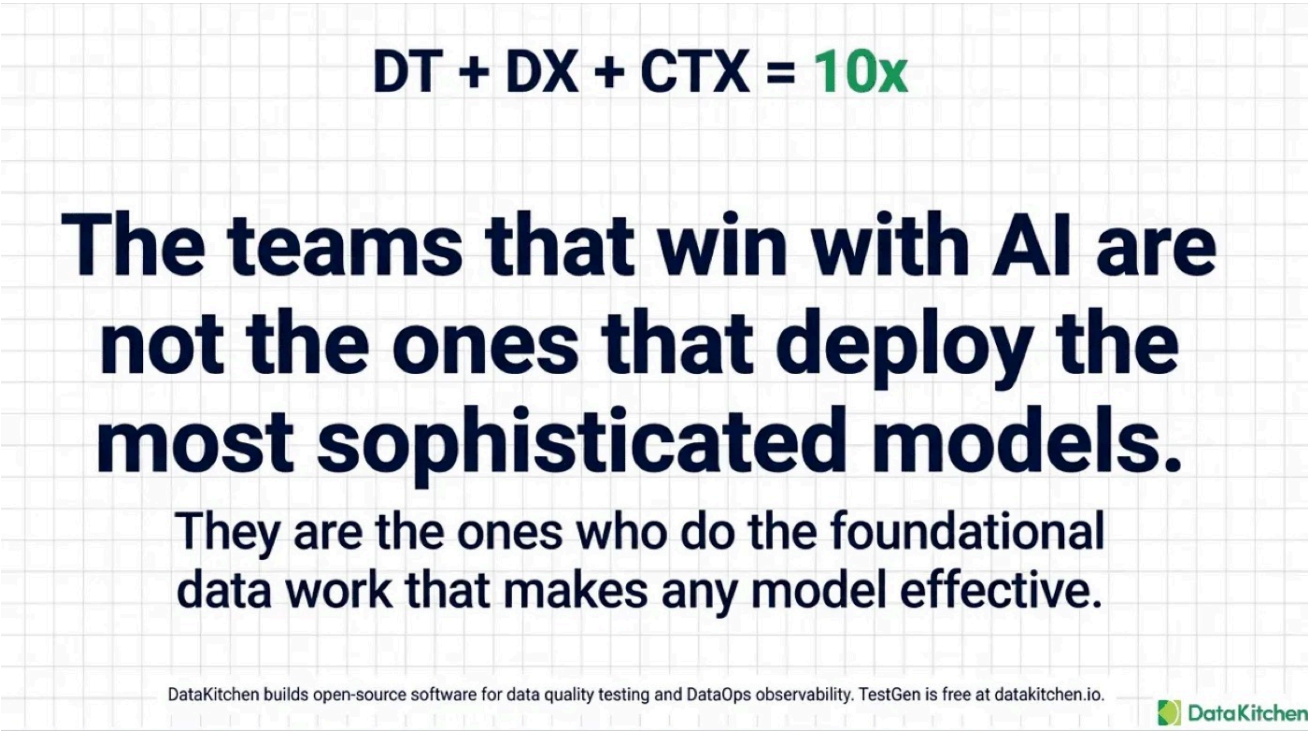
One important property of the CTX investment is its durability. The context you build , business definitions, semantic views, query patterns, domain skills , maps directly across platforms. If your organization moves from Snowflake to Databricks, or runs both, the content travels with you.

What We Build	Snowflake	Databricks
Natural language agents	Cortex Agents	AI/BI Genie
Semantic definitions	Semantic Views	Unity Catalog Metric Views
Verified/trusted queries	Verified Queries	Trusted Assets
AI coding assistant	Cortex Code	Databricks Assistant
Domain skills	Skills (auto-loaded)	Custom Skills (auto-loaded)
Feedback & observability	AI Observability Events	MLflow Tracing

Your investment in context engineering is durable. The platform is just where it runs.

Conclusion

The promise of AI-powered data analysis is real. Natural language queries, automated insight generation, and super-powered data analysis , these capabilities are available today. But they only deliver their full value when the data underneath them is trustworthy, legible, and enriched with context.



DT + DX + CTX = 10x is not a slogan. It is a checklist for the infrastructure work that makes AI analysis reliable, accurate, and genuinely useful to the people who depend on it.

The teams that will win with AI are not the ones that deploy the most sophisticated models. They are the ones who do the foundational data work that makes any model effective.

An Addendum for Data Engineering

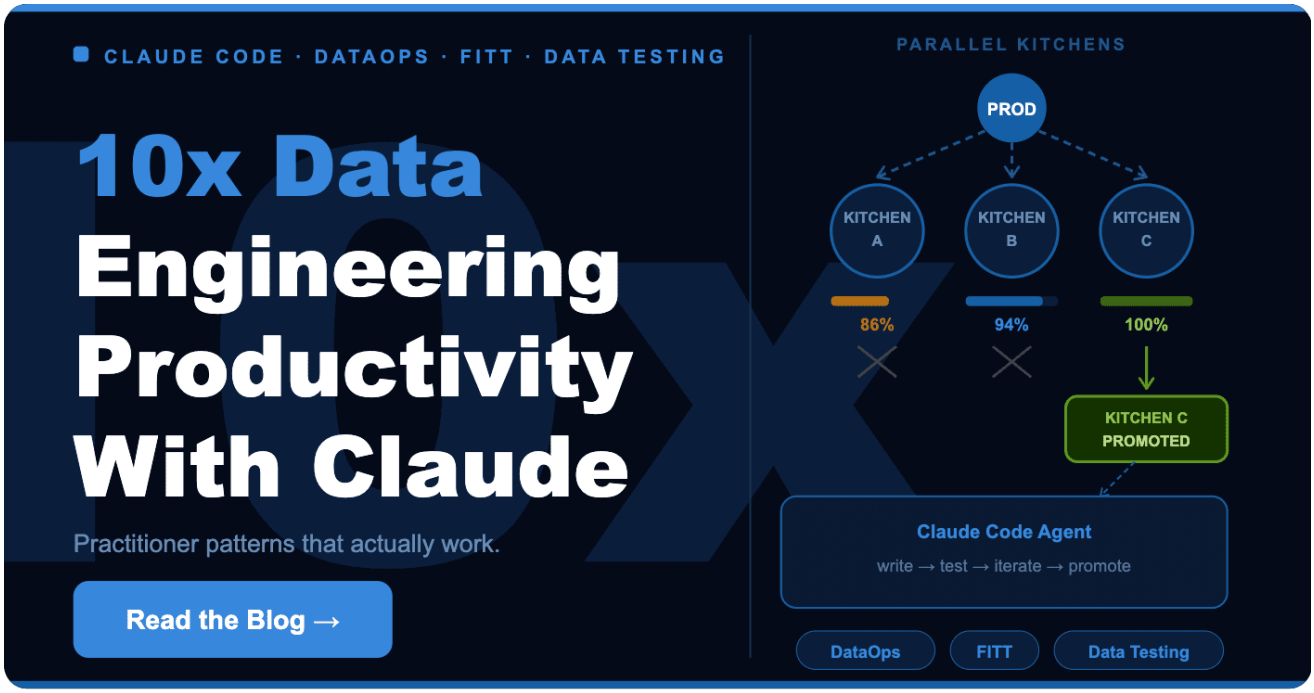
The 10x opportunity from LLMs is not limited to analysts. Data engineers building pipelines face the same foundational challenge: AI is an amplifier, and if the data flowing through those pipelines is untrustworthy, poorly tested, or missing operational context, it can amplify bad outputs at scale. We have a companion framework for that challenge in The Equation for AI Success: **Claude Code (DataOps + FITT + Data Testing) = 10x Data Engineering**. Where this post addresses how analysts make their data legible, trusted, and context-rich enough for AI to reason over it reliably, that one addresses how data engineers build the pipelines that make AI safe and productive. The two equations are meant to work together.

Read the companion piece: The Equation for AI Success: Claude Code + DataOps + FITT + Data Testing = 10x

AI AND DATA QUALITY

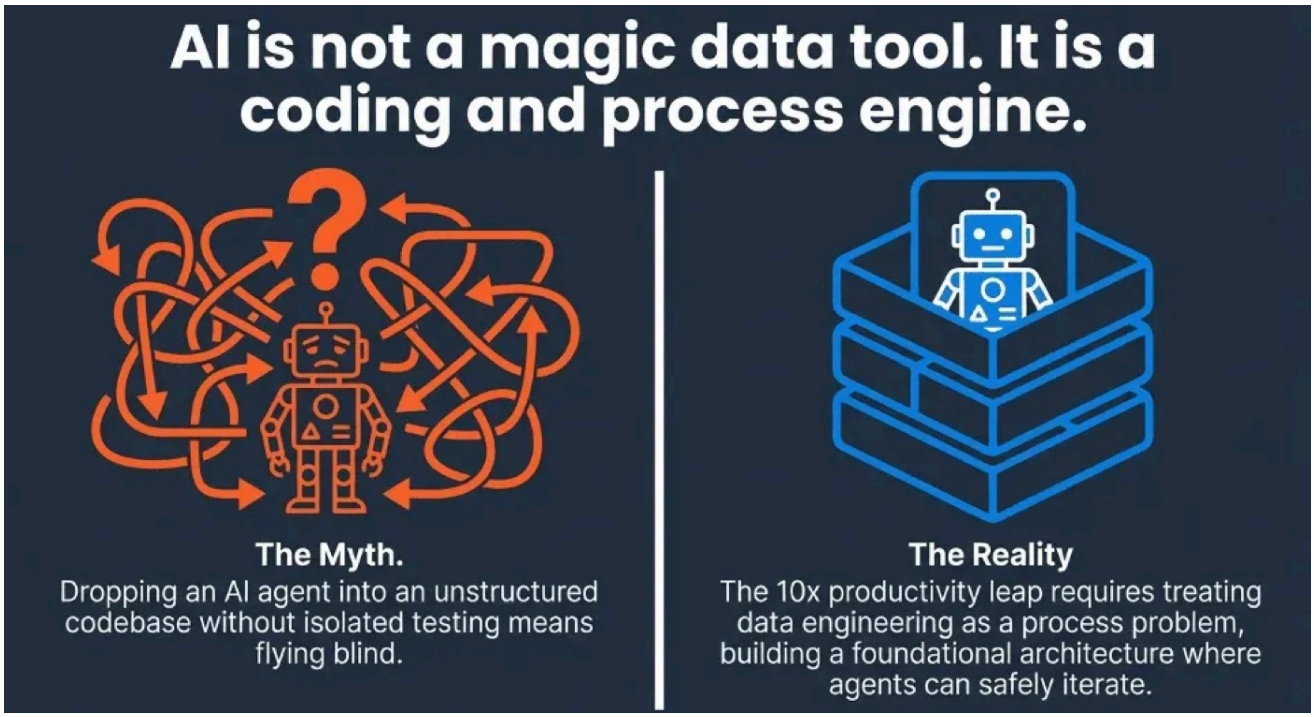
DataOps + FITT + Data Testing = 10x Data Engineering Productivity with AI

Read the original article at: https://datakitchen.io/blog/dataops-fitt-data-testing-10x-data-engineering-productivity-with-ai/



10x Your Data Engineering with AI: Practitioner Patterns That Actually Work

AI coding tools like Claude Code are generating significant excitement in software engineering. But for data engineers, getting 10 times the productivity isn't automatic. Just adding an AI agent to a messy pipeline and hoping it works usually leads to frustration.



At DataKitchen, we use Claude Code every day in our data engineering consulting work. We've found that the productivity boost is real, but it means changing how you approach your work. Most data teams focus on data itself: tables, metadata, schemas, and datasets. Claude Code isn't a data tool; it's about coding and processes acting upon data. To get the most out of it, you need to think like a process-centric data team, organizing work around pipelines, workflows, and transforms instead of just tables. Once you do that, the right foundation becomes clear: DataOps principles, FITT data architecture, and solid data testing all working together.

We summed up what makes this combination effective in a simple equation:

Claude Code(DataOps + FITT + Data Testing) = 10x Data Engineering Productivity

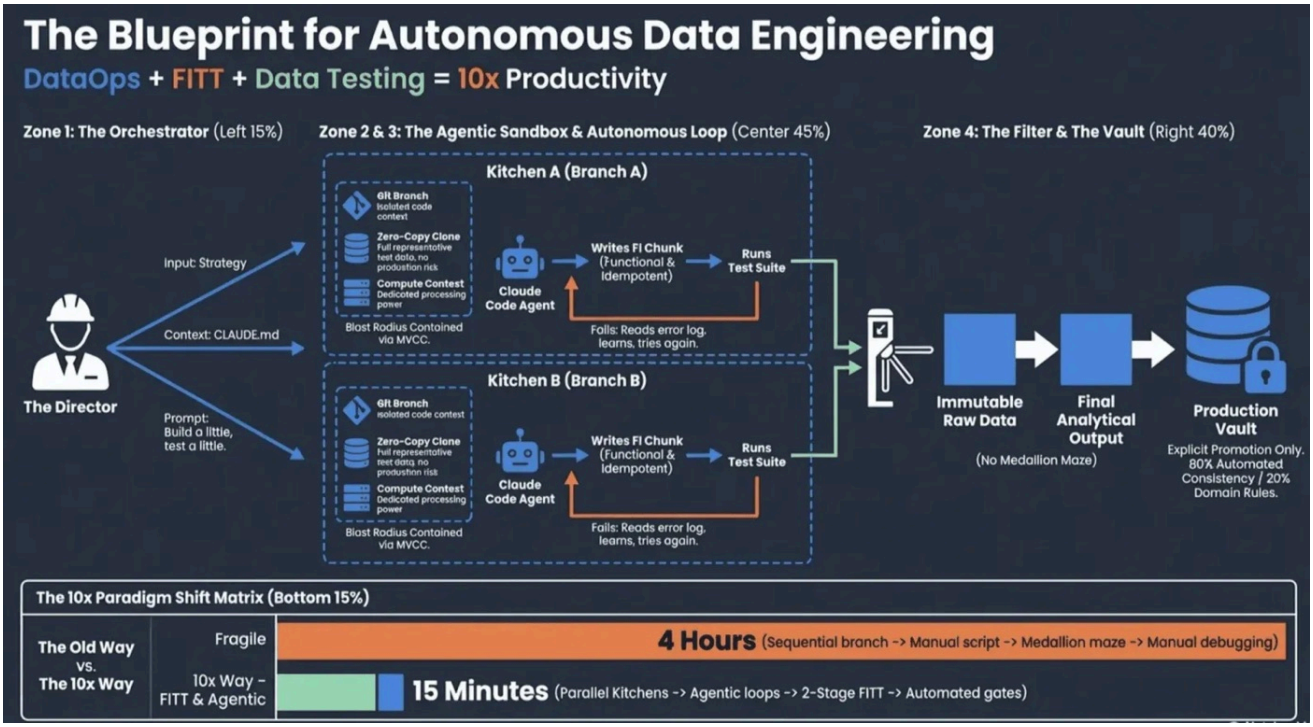
In this post, we'll break down each part and explain why all three are needed to unlock AI-assisted data engineering's full potential.

Why AI Alone Is Not Enough

Claude Code is genuinely impressive. Given a clear task, it can write SQL transforms, build ingestion logic, debug schema mismatches, and keep iterating on solutions for 20 minutes or more without human input. But capability alone is not enough. An AI agent working in a messy codebase, without isolated environments or tests to validate its output, has no reliable way to know whether what it produced is correct.

Understanding why requires understanding how Claude Code actually works. It does not write code once and hand it over. It operates in an agentic loop: writing a solution, running it, reading the output, diagnosing what broke, and trying again. That cycle repeats until the output satisfies the goal. The

iterations happen quickly, and the intermediate failures are invisible to the user, so the result can feel instantaneous. But what is actually happening is disciplined, self-directed trial and error, closer to how a skilled developer thinks than to how a code autocomplete tool operates.



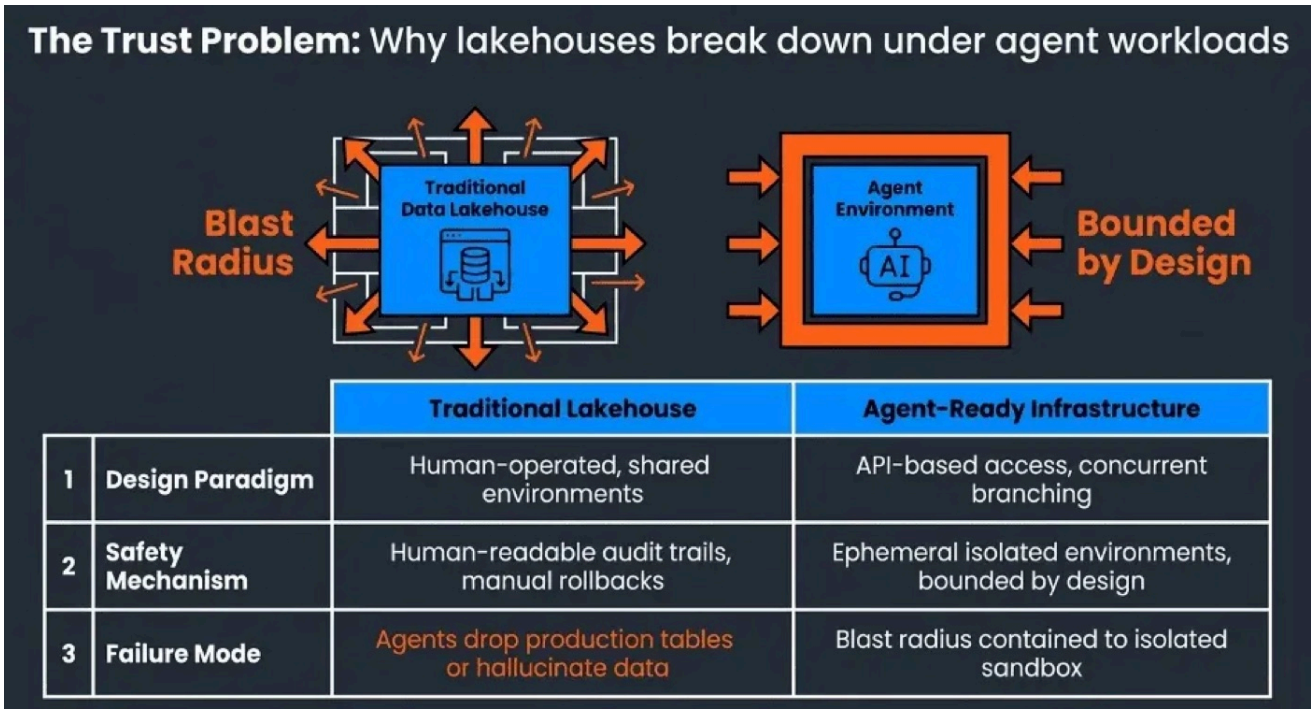
That loop only works when the environment supports it. If Claude is iterating against a shared production database, every failed attempt is a risk. If there are no tests, Claude has no signal for when it is done or whether it has regressed something upstream. The agentic loop and the infrastructure underneath it are not separate concerns. One depends entirely on the other.

The data engineers getting 10x results have invested in the scaffolding that makes the loop safe and productive. That scaffolding has three layers

Layer 1: DataOps

The Trust Problem

Even as AI capabilities improve rapidly, most enterprises are not ready to let agents loose on production data. The fears are concrete and legitimate: an agent accidentally dropping a table, polluting the data lake with hallucinated outputs, or silently corrupting a dimension that feeds downstream reporting. These are not hypothetical failure modes. They are the natural result of handing a powerful autonomous tool access to infrastructure that was never designed for agent access patterns.



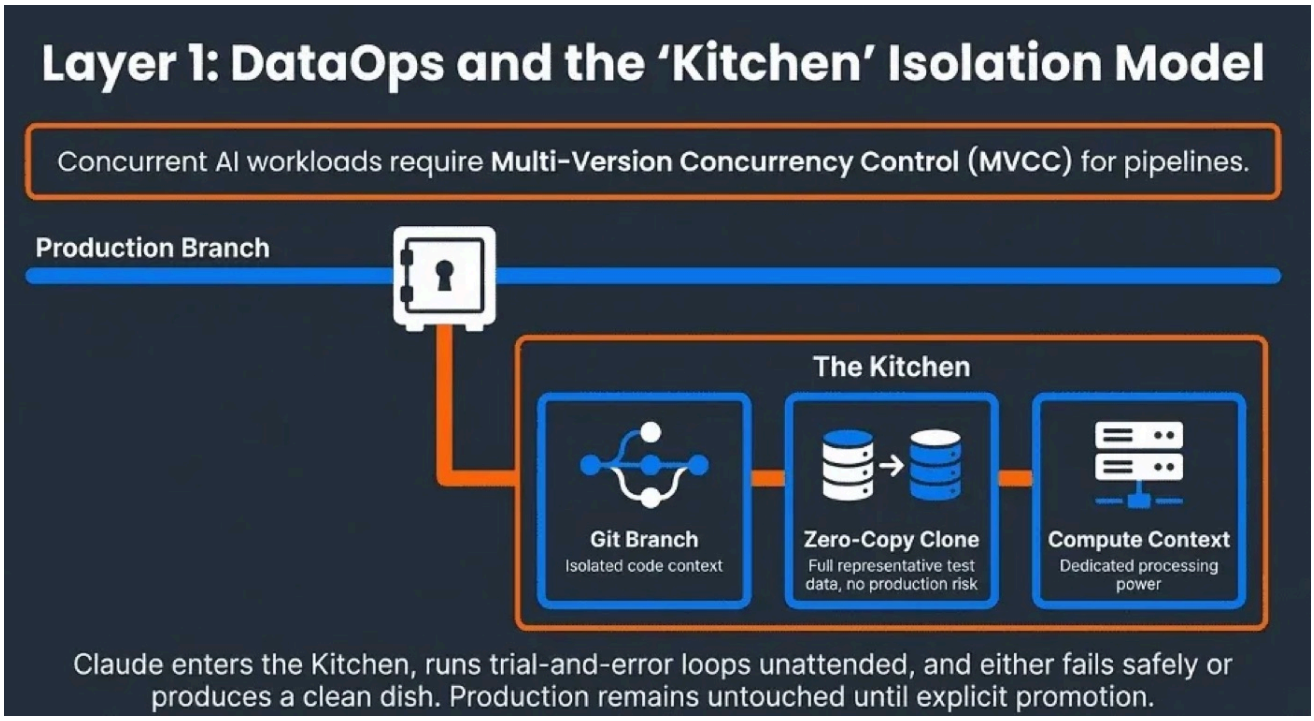
This trust gap is the real blocker for agentic data engineering adoption. It is not a capability problem. Claude Code can do the work. It is an infrastructure problem. Traditional lakehouses were built for human-operated workflows with human-readable audit trails and human-managed rollback procedures. AI Agents like Claude Code need something different.

The path to trustworthy agentic workflows starts with solving the infrastructure problem first. Design the data environment around agent access patterns, and governance follows naturally. Try to bolt governance onto an environment that was not designed for agents, and you are constantly fighting the architecture.

The Solution: Isolated Environments

DataOps is the application of Agile and DevOps principles to data work. The DataOps Cookbook is the definitive guide to these principles, and for AI-assisted data engineering, the most critical concept it surfaces is environment branching, not just code branching. Claude Code works through trial and error. Given a task, it will attempt a solution, run it, evaluate the result, and try again. This iteration loop can run for 30 minutes, an hour, or longer. During that time, you do not want it touching your production environment or your main branch.

The key insight is that correct concurrent AI workloads require solving the data, compute, and code branch isolation through a unified API. This is the same problem that database systems solved decades ago with MVCC (Multi-Version Concurrency Control). MVCC is the mechanism that gives each database user the illusion that they are the only user of the system. Every transaction sees a consistent snapshot of the data. Changes are isolated until committed. Other users are never exposed to work in progress.



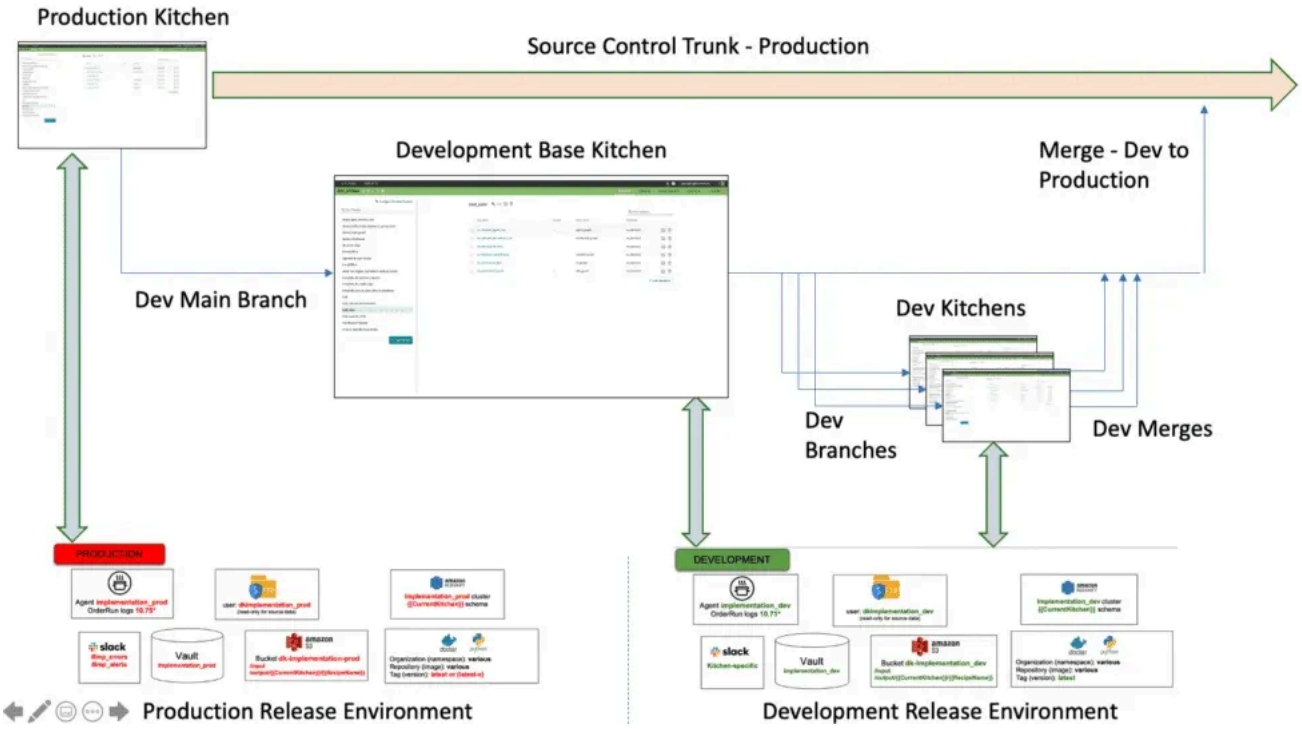
The same principle applies to AI agent environments. Each Claude session needs its own isolated view of the data and codebase, where it can read, write, experiment, and fail without any of that activity being visible to other sessions or to production. At DataKitchen, we call these isolated environments Kitchens. Each Kitchen bundles together three things Claude needs to work autonomously: a Git branch of the pipeline code, a pointer to the correct test data, and a separate compute environment. The test data itself is typically provisioned as a zero-copy clone of production, so Claude has a full representative dataset to run against without touching real data. Claude enters the Kitchen, writes or modifies a transform, runs it against the cloned dataset, evaluates whether the tests pass, and tries again if they do not. That loop repeats, unattended, until the work is done or Claude surfaces a genuine decision it cannot make alone. Production never sees any of it until the work is explicitly promoted.

Think of it as an ephemeral Kitchen for AI agents to work in. You can spin up many Kitchens for many agents simultaneously, each one cooking up a different approach to the same problem. Once their test cooking is complete, you evaluate the results, select the best dish, and promote it to production. The others get discarded. No mess, no risk, no cleanup.

Because Kitchens are fully isolated, there is nothing shared between them that could cause interference. A single engineer can run three or four Claude Code sessions in parallel, each on a different terminal, each working in a different folder, each mapped to a different Kitchen and Git branch, each taking a different approach to the same problem. They run unattended. Claude writes, tests, iterates, and either finishes or flags for human input. When you come back, you have multiple completed attempts to evaluate side by side rather than one sequential result. The orchestration is still yours: you set the brief for each Kitchen, they cook, you judge. But the execution is fully parallel and fully autonomous. The engineer's role shifts from writing pipeline code to directing a team of agents.

This isolation collapses the governance challenge down to a well-understood pattern: API-based access control. Instead of trying to audit every action an agent takes against a shared lakehouse, you control what the agent can do by controlling what the Kitchen API exposes. The agent can only see what it is given access to. It can only write to its own isolated context. The blast radius of any mistake is bounded by definition. DataKitchen's DataOps Automation platform provides this unified API, giving teams the infrastructure to run autonomous Claude sessions safely without rearchitecting their entire lakehouse from scratch. That way, Claude can try new SQL, a variation of Python Code, or a script tweak in isolation.

DataOps Automation gives you the process framework that sits on top of this. You branch your code and your environment simultaneously, so Claude works inside an isolated Kitchen with its own Git branch, test data pointer, and compute context while your main branch and production environment remain untouched. You treat data pipelines as processes rather than scripts, which makes it natural to hand off a bounded unit of work to an AI agent operating inside its own Kitchen. And you maintain full lineage of the action in an Order Run data structure, so every change Claude makes is traceable, reviewable, and revertable before anything touches production.



Governance of AI agents in data infrastructure is not mainly an AI issue. It is a concurrency issue. Address isolation first, and managing AI Agents becomes like overseeing multiple separate 'kitchens' of work. DataKitchen created its Kitchens so developers could work in isolated, aligned sandboxes and confidently combine their work. Fix the concurrency issue at the infrastructure level, and access control shifts from a complex organizational challenge to a simple API problem. This is crucial whether you have two developers or two hundred AI Agents.

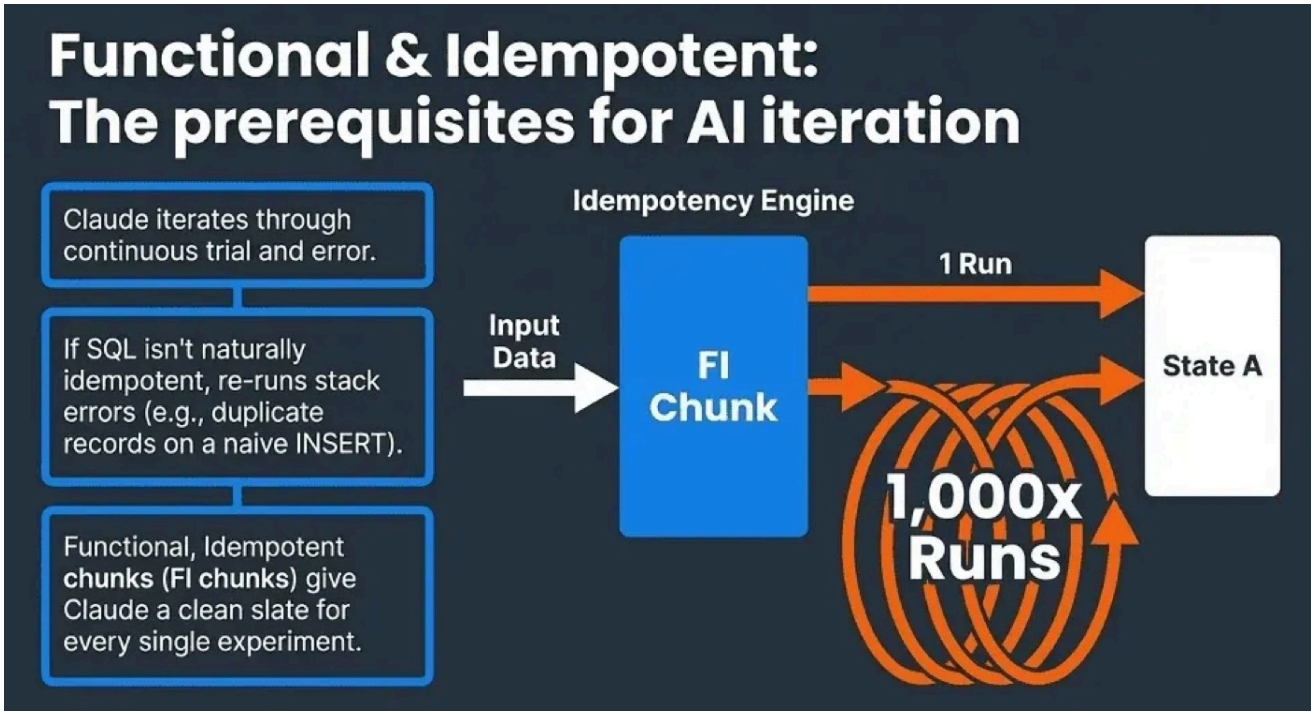
Layer 2: FITT Architecture

The second layer is architectural. We call it FITT: Functional, Idempotent, Tested, Two-stage. Most data architectures are designed around vendor capabilities or medallion-layer conventions. FITT is designed around data engineer productivity and, as it turns out, AI agent productivity too. The principles that make pipelines easier for humans to reason about and recover from are precisely the principles that make them safe and productive for an autonomous agent to work on.

Functional: same input, same output, every time

A functional transform has no mystery dependencies and no hidden state. For each code version, the transformation takes clearly defined inputs and produces predictable outputs. The same input produces the same output regardless of when or where it runs. Raw data is treated as immutable by default. Functional Data Engineering eliminates an entire class of bugs that plague traditional data systems: no more wondering whether yesterday's transformation affected today's results, no more state-dependent edge cases that only surface in production.

For an AI agent, this property is transformative. Claude can reason about a functional transform completely from its definition. It does not need to understand the history of prior runs or the current state of upstream tables. It sees inputs, it sees the expected output shape, and it can write, test, and iterate on the logic in total isolation. The transform is a mathematical function. Claude can treat it like one.



Idempotent: run it a thousand times, get the same result

Idempotency is the property that changes how the whole team thinks about failure and iteration. When a pipeline is idempotent, recovery is trivial. Did something break due to a schema change? Fix the transform and re-run it. Getting unexpected results? Re-run with debugging enabled on yesterday's data

and compare. Want to test a change safely? Re-run it on a historical snapshot. This eliminates the paralyzing fear of corrupting production data that keeps junior engineers from experimenting and keeps senior engineers up at night.

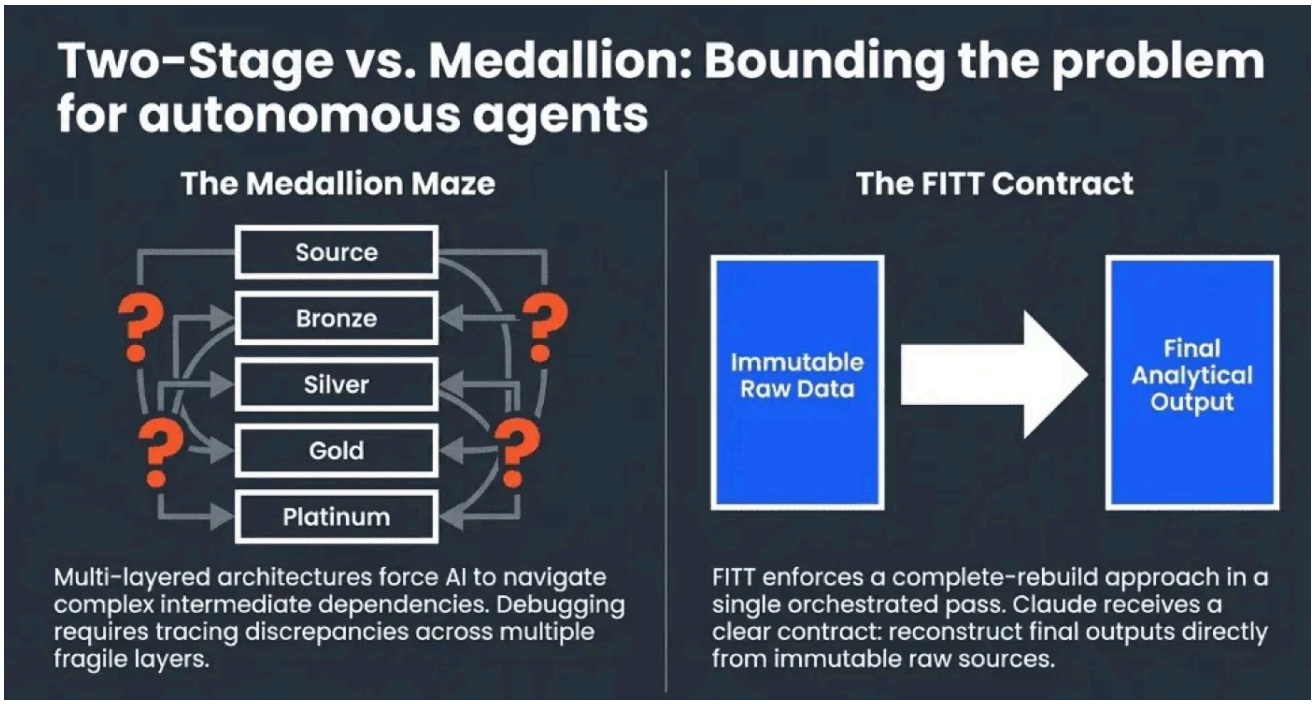
The practical implementation challenge is that SQL is not naturally idempotent. A naive INSERT run twice creates duplicate records. A CREATE TABLE AS SELECT fails on a second run if the table already exists. Making SQL truly idempotent requires deliberate design: explicitly managing state, using parameterized time windows so transforms address a specific data snapshot rather than assuming current state, and structuring each transform as what the FITT vs. Fragile SQL article calls an 'FI chunk', a Functional and Idempotent chunk that becomes the fundamental building block of a FITT-compliant pipeline. Whether you are using dbt, Databricks, Snowflake, or BigQuery, the goal is the same: each chunk manages its own state, declares its dependencies explicitly, and can be safely re-executed at any time.

For Claude, idempotency is not a nice-to-have. It is the property that makes autonomous iteration possible at all. Claude's approach to any task is trial and error: write a solution, run it, evaluate the result, adjust, and try again. That loop only works if every run starts from a known state and produces a deterministic result. Without idempotency, Claude's re-runs stack errors on top of each other. With it, each iteration is a clean experiment.

The FITT blog describes this as a 'build a little, test a little, learn a lot' development rhythm. That rhythm is exactly how Claude Code works. The architecture and the agent's working style are designed to work together.

Two-stage: raw to final, that is it.

Most teams inherit a multi-layered medallion architecture where bronze data does not match gold data, and debugging at 2 AM means tracing discrepancies across 5 intermediate layers. FITT replaces this with two stages: raw data that represents exactly what arrived from source systems , immutable and timestamped , and final data that is the complete analytical output rebuilt from that raw layer, no bronze, silver, gold, or platinum in between. The complete rebuild approach reconstructs the entire analytical dataset from immutable raw sources in a single orchestrated pass. If you need to verify last month's results or backfill three months of data after a bug fix, you adjust the time window parameters and re-run. The process is identical whether you are processing one day or one hundred.



This simplicity is a significant advantage for AI agents. A Claude session working inside a Kitchen does not need to understand which intermediate layer it is allowed to modify or which upstream tables are shared with other pipelines. It works with raw inputs and produces final outputs. The Two-stage contract is clear, bounded, and easy to test.

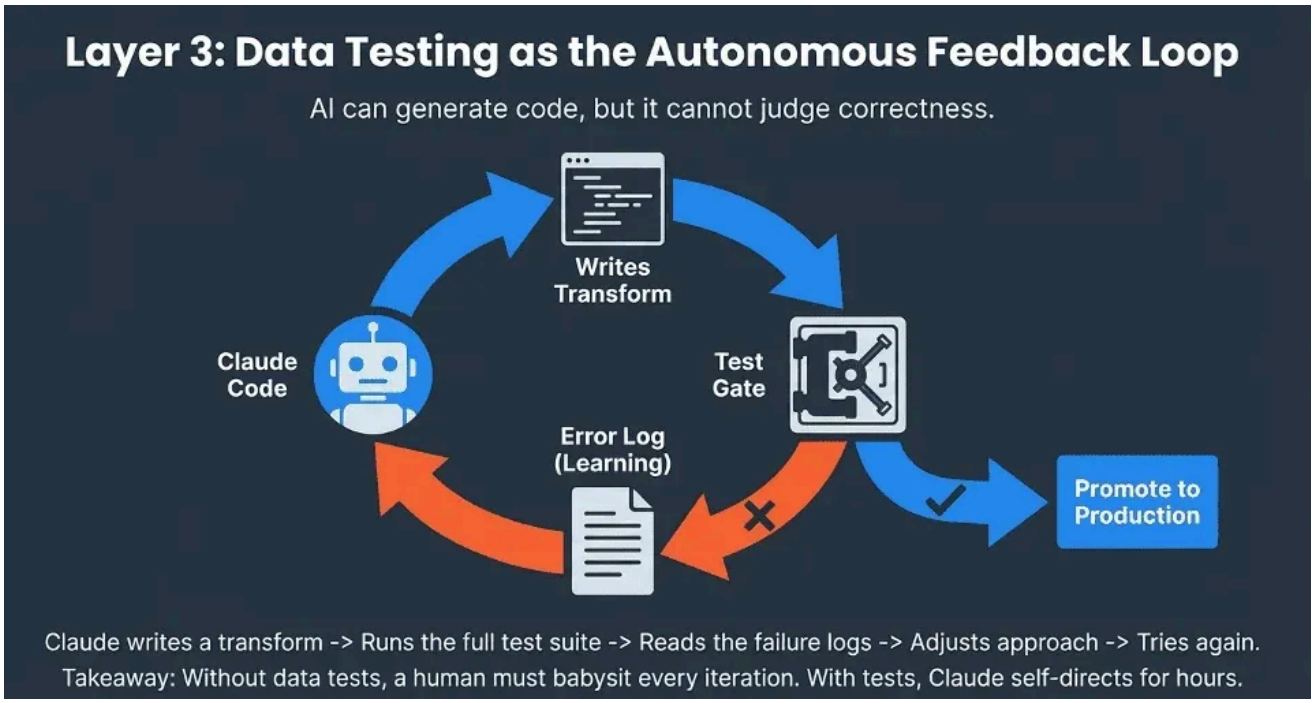
FITT is what makes Claude's problem-solving granular enough to succeed. An FI chunk has clear inputs, a clear expected output, and tests that define what 'done' means. Claude can pick one up, iterate on it in isolation without fear of breaking anything else, and know exactly when it has finished. That is the fundamental unit of autonomous AI data engineering.

The FITT architecture is similar to the write-audit-publish pattern that Netflix first talked about. Claude writes to its isolated Kitchen environment, audits the result against the test suite, and only promotes data forward when the checks pass. FITT provides autonomous quality control built into its architecture, not bolted on afterward.

Layer 3: Data Testing

This is arguably the most important layer, and the most underinvested in most data teams.

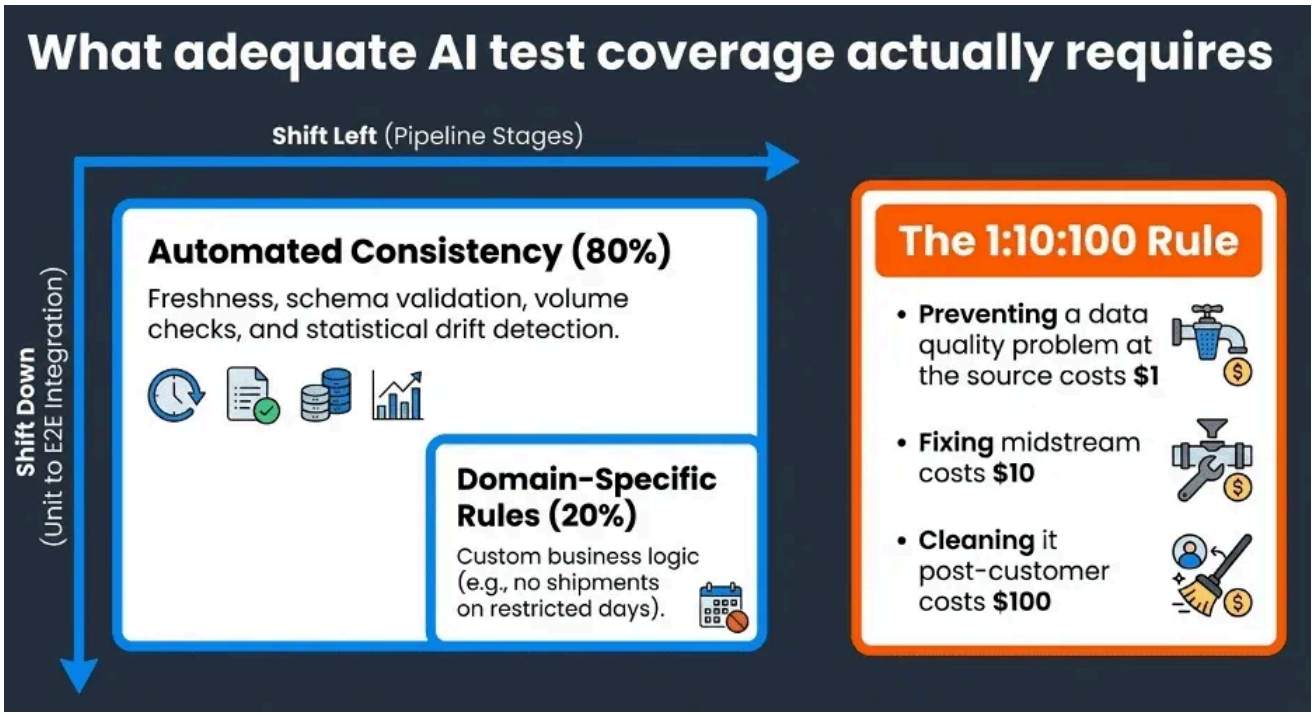
Claude Code can generate code. It can run code. What it cannot do on its own is judge whether the output is correct. That judgment has to come from data tests. Without adequate test coverage, an AI agent has no feedback loop. It cannot learn from its mistakes, cannot verify its progress, and cannot know when it is done. Our Definitive Guide to Test Coverage for Data Engineers goes deep on what a real coverage strategy looks like across every layer of your pipeline.



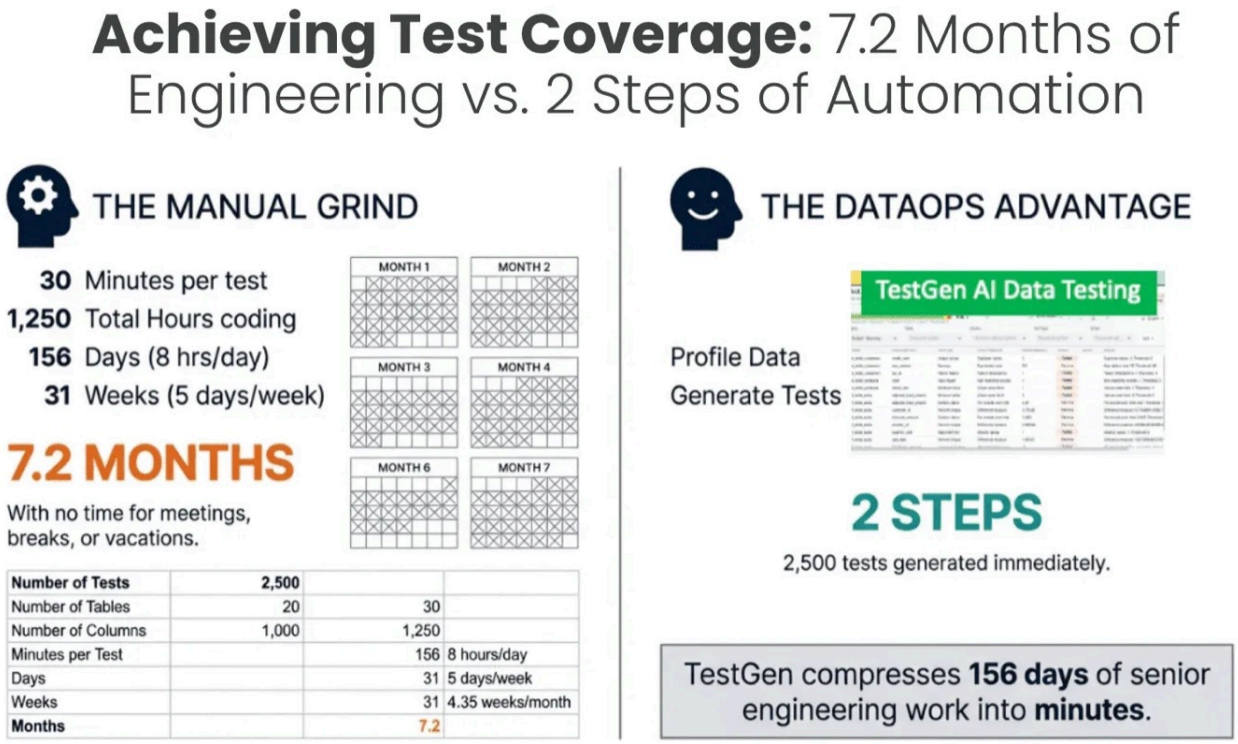
What adequate test coverage means in practice

The Definitive Guide frames test coverage around two complementary strategies: shift left and shift down. Shifting left means running data quality tests earlier in the pipeline lifecycle, catching problems before they propagate downstream and become exponentially more expensive to fix. The 1:10:100 rule captures the stakes: preventing a data quality problem at the source costs roughly $1 per record; fixing it midstream costs $10; cleaning it up after it has reached customers costs $100. Shifting down means going beyond unit-level validation to integration and end-to-end testing that validates how components interact across the full pipeline, not just in isolation.

Comprehensive coverage operates across two distinct dimensions simultaneously. Production test coverage ensures that data quality remains high throughout live operations, catching freshness failures, volume anomalies, schema drift, and distribution changes as they happen. Development test coverage validates that code changes do not introduce regressions or break existing functionality when promoted from a Kitchen to production. Many tests serve double duty across both dimensions, functioning as both production monitors and development validators.



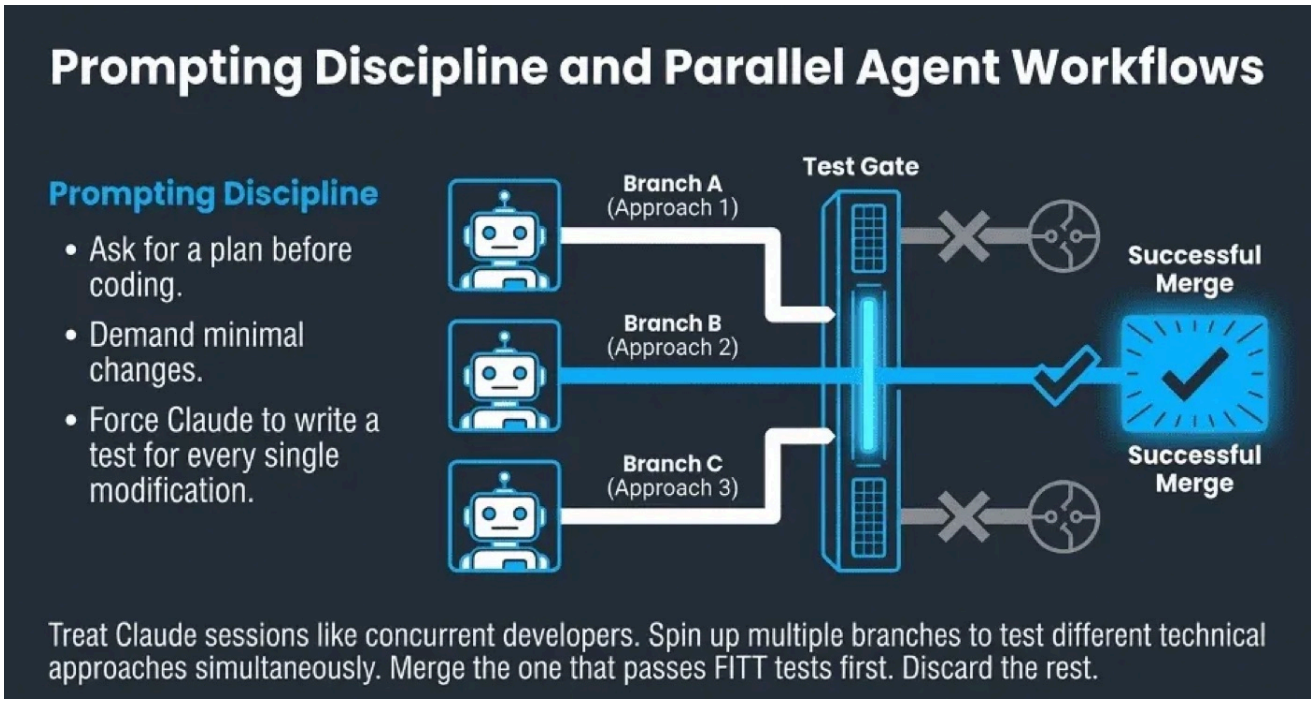
In terms of what the tests themselves cover, automated consistency tests handle the bulk of the work, roughly 80% of what a comprehensive suite needs. These include freshness and volume checks, schema validation, referential integrity, and data drift detection using statistical methods to catch gradual shifts in data characteristics that simpler threshold-based alerts miss. The remaining 20% is custom domain-specific tests that encode business rules no automated system can infer from data patterns alone: verifying that a medical practice has no more appointments than it has doctors, ensuring no shipments occur on restricted days, and confirming that revenue figures fall within historically expected ranges. Those business rule tests are where your domain knowledge becomes part of the quality gate that Claude must pass.



Representative test data is equally important. Claude needs to run against data that actually exercises the edge cases and failure modes of your pipeline. Minimal or synthetic test data can lead to false confidence. The zero-copy clone in the Kitchen exists precisely to give Claude a full, realistic dataset to work against without risking production. Development like this: yesterday's data run against today's code.

Tests as the AI feedback loop

Here is the workflow that makes long autonomous Claude sessions possible: Claude writes or modifies a transform, runs the full test suite against the branch, reads the failure output, adjusts its approach, and tries again.



This loop can run for 20, 30 minutes, or several hours when the problem is complex. You are not babysitting it. You check back in, and either the tests are green, or Claude has surfaced a real issue that needs guidance on.

High coverage data tests are the mechanism that turns Claude Code from a code generator into an autonomous engineering agent. Without them, you need a human in the loop for every iteration. With them, Claude can self-direct.

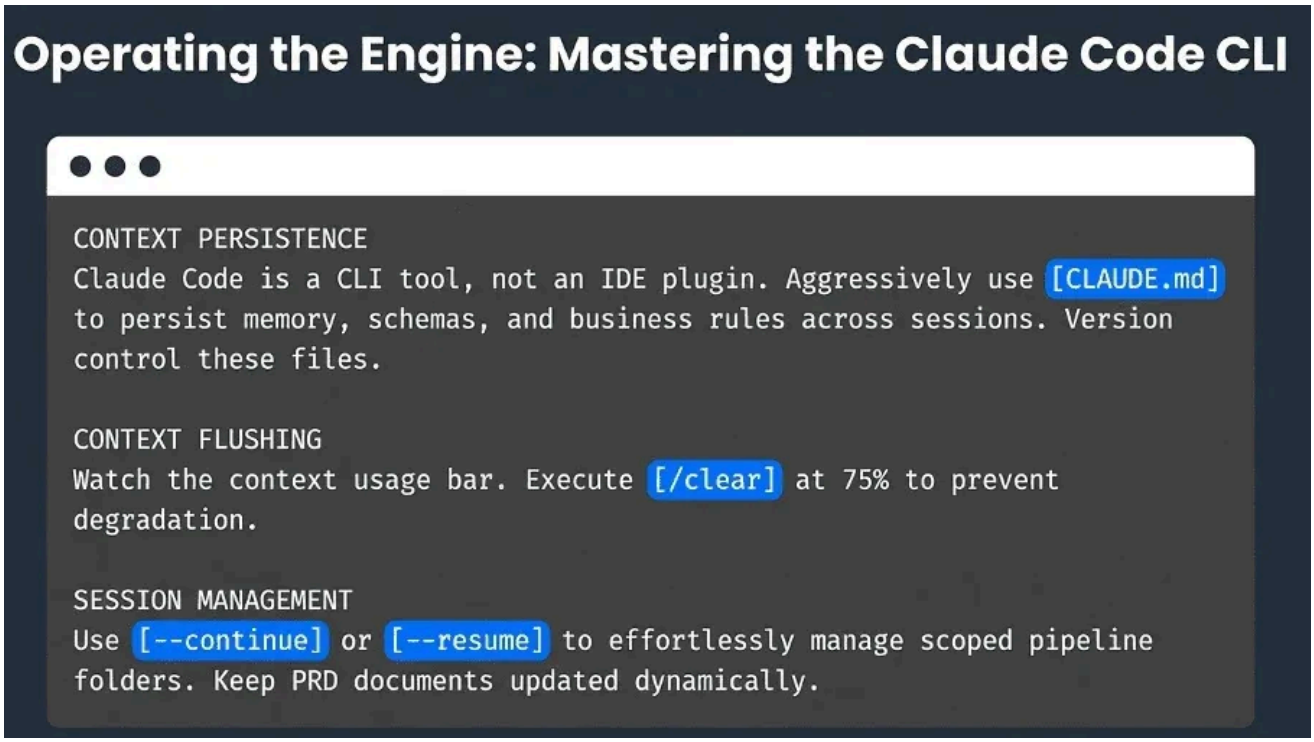
Data testing is how you manage AI risk. If Claude goes down the wrong path for 30 minutes, the tests catch it. Nothing ships until the tests pass. Your test suite is your safety net and your teacher simultaneously. DataKitchen's open-source TestGen is built exactly for this: it generates thousands of data quality tests automatically from your schema and data profile, giving Claude the feedback loop it needs to work autonomously without requiring your team to hand-write every test case.

For a comprehensive treatment of how data testing and observability fit together in a modern data engineering practice, see our new book The DataOps Way to Data Quality and Data Observability.

Tips: How Data Engineers Work with Claude Code

The first thing to understand about Claude Code is that it is a CLI tool, not an IDE plugin, and that distinction matters. Claude Code manages its own persistent memory through CLAUDE.MD files that are stored in your project folders. These files are loaded automatically at the start of every session, so Claude picks up exactly where you left off without you having to re-explain your schema, conventions, or the business rules you spent 20 minutes describing last Tuesday. Every AI chat session starts with a ticking clock as context builds and earlier information is pushed out. CLAUDE.md files solve this. Use them

aggressively. Ask Claude to document what it has built and what assumptions it is making so that knowledge survives session restarts, machine changes, and team handoffs. Version control those files in Git alongside your pipeline code.



Context management is a skill you will develop quickly. Watch the context usage bar in the CLI, and use /clear when it reaches 75%. Organize your work into one session per folder, grouped by topic or pipeline stage. When you need to resume, the fastest path is claude , continue to pick up the last session instantly, or claude , resume to browse and select from named sessions. For longer-running agent workflows, create a PRD (planning document) in the folder; ask Claude to update it as work progresses. A well-maintained PRD is structured knowledge, not just chat history, and it is what allows a teammate or a fresh Claude session to step in without losing ground.

Prompting discipline makes a measurable difference. Before Claude starts any task, ask it to make a plan and review it with you before touching anything. Ask what assumptions it is making. This two-minute check prevents Claude from spending 40 minutes on an approach that misses a constraint you forgot to mention. During work, push Claude toward minimal changes: the prompt that consistently produces tighter, more reviewable diffs is simply asking for the minimum change needed to achieve the goal. After Claude makes changes, ask it to write a test that proves the changes are correct. This directly feeds the test coverage that makes autonomous iteration possible and keeps the feedback loop tight. Explicitly save work to disk before clearing context.

The highest-leverage pattern in the slides is parallel development. Think of each Claude session as a separate developer. Different folders mean different branches, and you can run multiple terminals simultaneously, each with Claude grinding on a different pipeline unit in isolation. You do not need to make perfect upfront decisions about which approach is right. Spin up two branches with two different

approaches, let Claude iterate on both, and merge whichever one passes the tests first. The mental model shift is significant: you move from writing pipeline code to directing a team of autonomous agents, each working in its own safe sandbox, with data tests as the shared definition of done. Four-hour tasks become 15-minute tasks. The focus shifts entirely to what needs to be built, not how to build it.

A Walkthrough Example

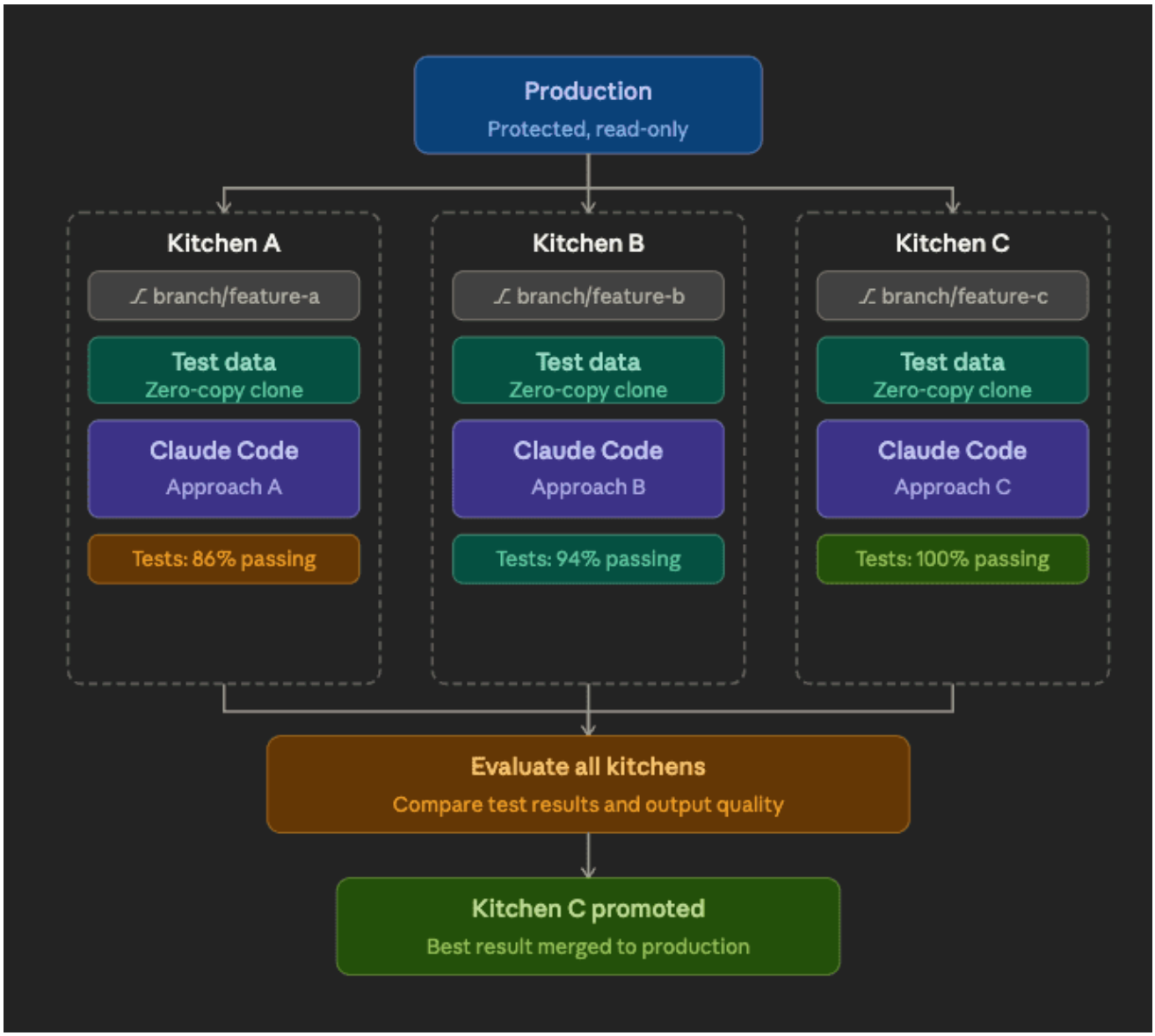
A data engineer has a task: rewrite a slow, brittle SQL transform that ingests raw sales data into a final reporting table. The existing version breaks on reruns, has hidden state dependencies, and takes 40 minutes to run on a bad day. Rather than working through one approach sequentially, the engineer spins up three approaches and lets Claude explore three strategies in parallel: parameterized time windows, explicit staging-table logic, and a full-step rebuild from raw.

Provisioning the Kitchens

Each Kitchen is a bundle of three things: a Git branch forked from main, a zero-copy clone of the production dataset, and a separate compute context.

The zero-copy clone provides each Claude session with a complete, realistic, production-scale dataset to work with without duplicating storage. Each Kitchen references the same underlying production data blocks, with a copy-on-write layer on top. When Claude generates output, those changes are made to the Kitchen's isolated layer. Production data blocks remain unchanged. The cost is nearly zero. The isolation is thorough.

The engineer opens three terminal sessions, points each at a different folder mapped to a different Kitchen, gives each a brief, and walks away.



Claude Iterates in Each Kitchen

Inside each Kitchen, Claude reads the CLAUDE.md context file, inspects the schema, and starts writing. It runs the transform against the cloned dataset, evaluates the output against the test suite, reviews any failures, revises, and reruns. It does not wait for input. It repeats this process until the tests pass or it encounters a decision it cannot make on its own. Kitchen A, using the parameterized time-window approach, encounters problems with late-arriving records. After four iterations, it reaches 86%, but notes that one cross-period consistency check is still failing and may require human judgment on an acceptable tolerance. Kitchen B, employing the staging table approach, proceeds smoothly through the standard automated tests and completes at 94%. It records that the remaining failures are mainly in a single edge case involving partially fulfilled cancelled orders. Kitchen C, running the full step rebuild, takes the most aggressive route. All tests pass: 100%. It logs an 8-minute rebuild time on the cloned dataset.

Evaluation and Promotion

With test results in hand across all three Kitchens, Claude compares the outcomes. Kitchen A is eliminated: below threshold and flagged for a structural issue that would require rearchitecting the windowing logic. Kitchen B passes at 94%, but it carries a known edge-case failure that would require additional work to resolve. Kitchen C wins: 100% test pass rate, clean output, and an 8-minute rebuild cycle, well within the production schedule for a transform running every 15 minutes.

Kitchen C's branch is promoted. The engineer reviews the diff, a clean, readable transformation built on a full rebuild from immutable raw sources following FITT conventions, and merges it into main in ten minutes. Kitchen A and Kitchen B are discarded. Their branches are deleted, their compute contexts shut down, and their zero-copy clones released. Nothing from those sessions ever made it into production.

What Just Happened

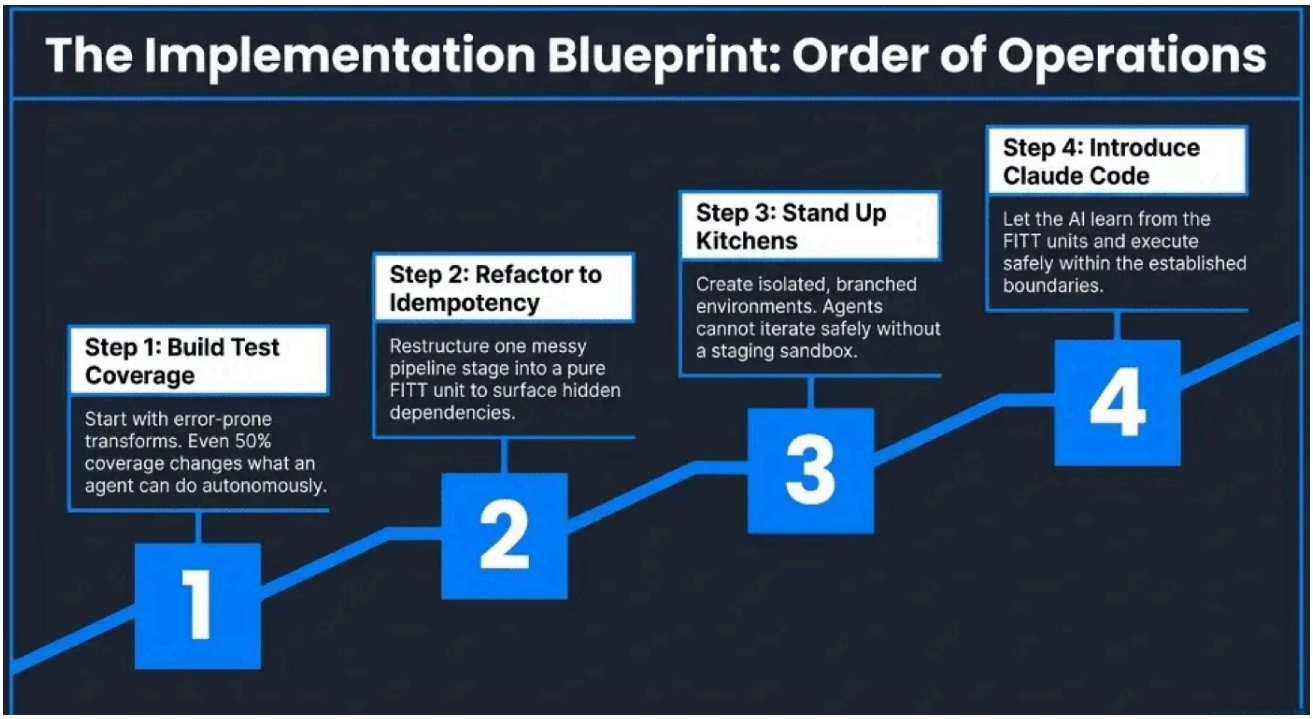
A task that would have taken a full day of sequential engineering was completed in under two hours, with three approaches explored simultaneously and the engineer's active involvement limited to writing the briefs and reviewing the winning diff. The test suite did the continuous validation. Claude did the iteration. The engineer made the final call. That division of labor is where the 10x comes from.

The 10x Is Real, But It Is Not Magic

We want to be specific about what 10x means here. It does not mean that Claude Code writes 10x as many lines. Tasks that used to take a day take two hours. A single engineer can run multiple parallel workstreams across isolated Kitchens simultaneously. Debugging time drops sharply because tests catch errors before they propagate. Onboarding new pipeline logic is accelerated because data tests are automatically generated and document the expected behavior. And code review is faster because every diff is scoped to a FITT unit with a clear, bounded surface area.

None of this happens without the foundation. DataOps provides the branching and process discipline. FITT provides the granularity and idempotency that make autonomous iteration safe. Data testing provides the feedback loop that enables autonomous iteration.

Remove any one of the three, and the productivity gains collapse. Together, they multiply each other.

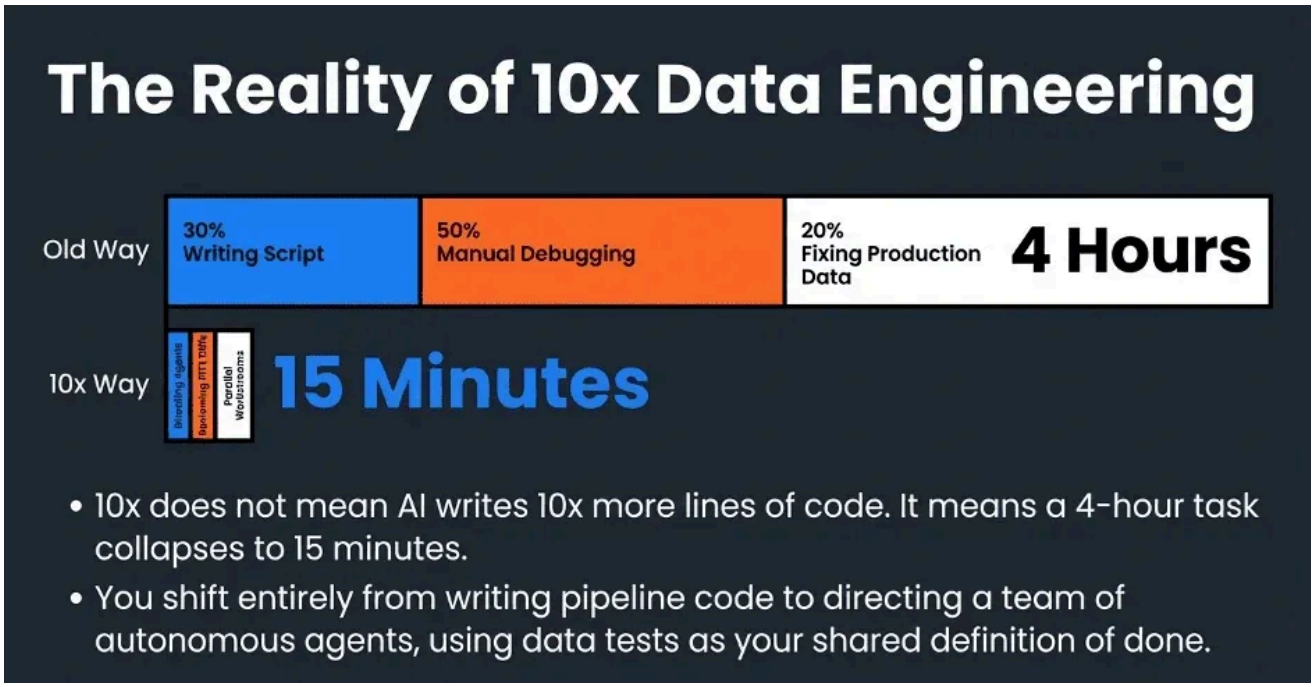


How To Get Started Today

How do I start moving toward this model that enables Claude Code's success? Build your data test coverage first. Start with the transforms that are most error-prone or most business-critical. Even 50% coverage on key transforms changes what Claude can do autonomously. Then refactor toward idempotency. Pick one messy pipeline stage and restructure it as a FITT unit. The refactor itself will surface hidden dependencies and assumptions that have been buried in the code. Next, stand up isolated branched environments. If your pipeline can only run in one place, you cannot safely let Claude iterate. Then, and only then, introduce Claude Code. With the foundation in place, the onboarding is fast. Claude learns the patterns from your existing FITT units and tests, and productive output starts immediately.

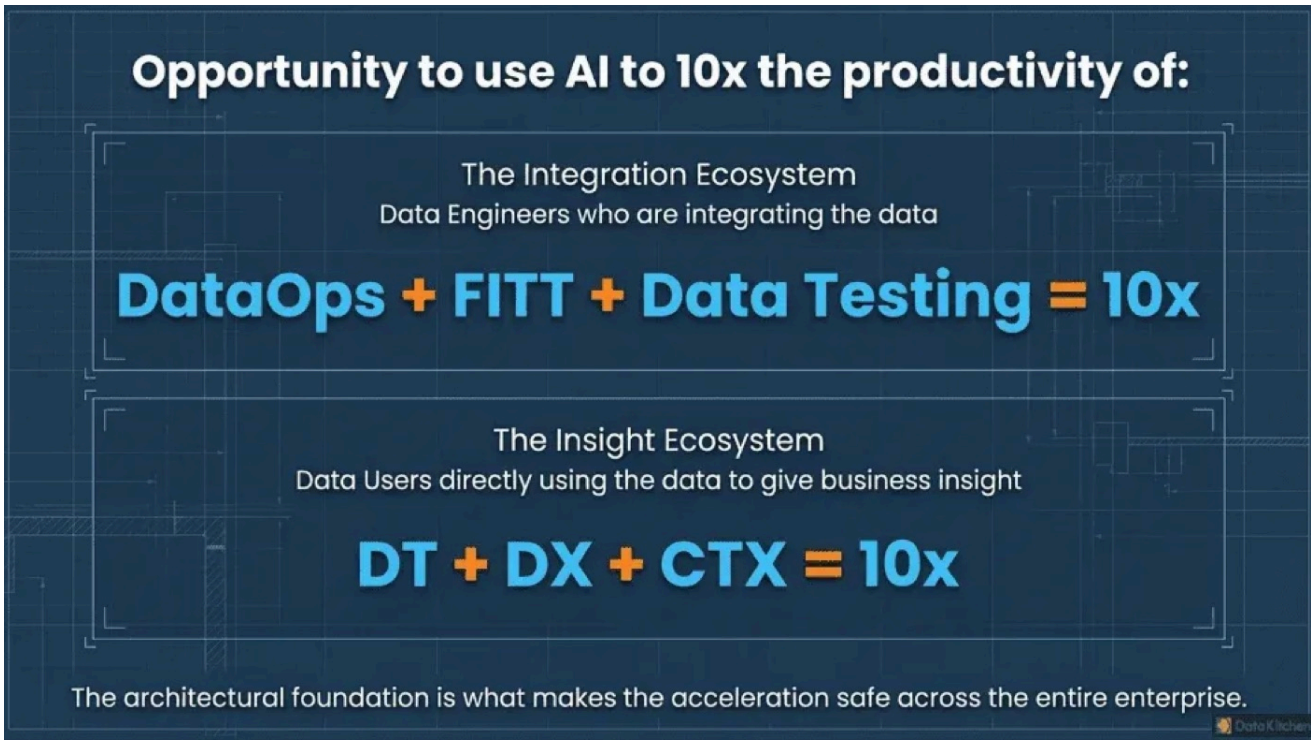
Conclusion

AI coding tools are going to change data engineering. That much is certain. The question is whether the change for your team is a genuine productivity multiplier or just a faster way to generate untested code that breaks in production.



The engineers getting 10x results are not the ones with the best prompts. They are the ones who invested in DataOps discipline, FITT architecture, and data test coverage before they picked up an AI tool. Claude Code is the accelerant. The foundation is what makes the acceleration safe.

The 10x opportunity from LLMs is not limited to data engineers. Analysts working directly with AI to answer business questions face the same foundational challenge: AI is an amplifier, and if what sits beneath it is untrustworthy, poorly documented, or missing business context, it can amplify bad answers at speed. We have a companion framework for that challenge in The Equation for AI Success: DT + DX + CTX = 10x. Where this post addresses how data engineers build the pipelines that make AI safe and productive, that one addresses how analysts make their data legible, trusted, and context-rich enough for AI to reason over it reliably. The two equations are meant to work together.


Opportunity to use AI to 10x the productivity of:
The Integration Ecosystem
Data Engineers who are integrating the data
DataOps + FITT + Data Testing = 10x
The Insight Ecosystem
Data Users directly using the data to give business insight
DT + DX + CTX = 10x
The architectural foundation is what makes the acceleration safe across the entire enterprise.

AI AND DATA QUALITY

Context As Infrastructure: The Six-Category Framework for AI-Ready Data Analysis

Read the original article at: https://datakitchen.io/blog/context-as-infrastructure-the-six-category-framework-for-ai-ready-data-analysis/

Context Is Infrastructure

Six Category Framework for AI-Ready Data Analysis

Playbook | DataOps | AI Strategy

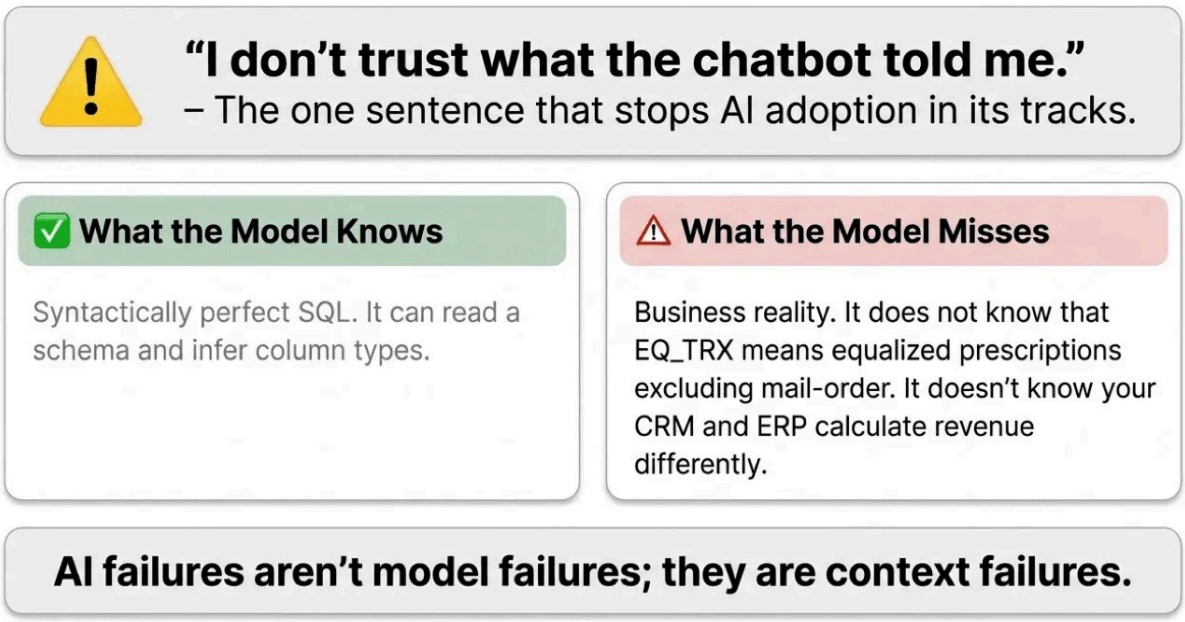
DataKitchen

This post is a follow-up to The Equation for AI Success: DT + DX + CTX = 10x. If you have not read that one yet, start there. It lays out the three-part foundation for reliable AI analysis: Data Trust, Data Experience, and Context. In April 2026, we ran a webinar that walked practitioners through the full framework live, with real examples from Snowflake and Databricks deployments. You can watch the recording here: 10x Your AI Data Analysis: The Context Engineering Formula That Works. This post goes deep into the third term in the equation. Context is the multiplier that turns a generic language model into something that actually understands your business. Here, we break it down into the six specific categories that make up a complete context layer, how you manufacture each one, and why keeping context current is now a parallel discipline to data engineering.

Anyone who has connected an AI assistant to their data warehouse knows this story. At first, the answers seem impressive. Then an analyst finds a number that does not make sense, traces it to the AI, and says, “I do not trust what the chatbot told me.” That one sentence can stop AI adoption in its tracks.

That reaction is not really about the model itself. It points to a problem with the context layer, or more often the lack of one.

The model is doing exactly what it was designed to do. It reads the schema in front of it, infers what the columns mean, generates syntactically plausible SQL, and returns an answer with confidence. What it cannot do is know that EQ_TRX means equalized prescriptions, normalized for days supply, excludes mail-order, and must be summed rather than averaged. It cannot know that TERR_CD changed its alignment definition in week 14. It cannot know whether the metric your finance team calls revenue and the metric your CRM calls revenue are computed differently, and the answer to any question depends on which one you mean.

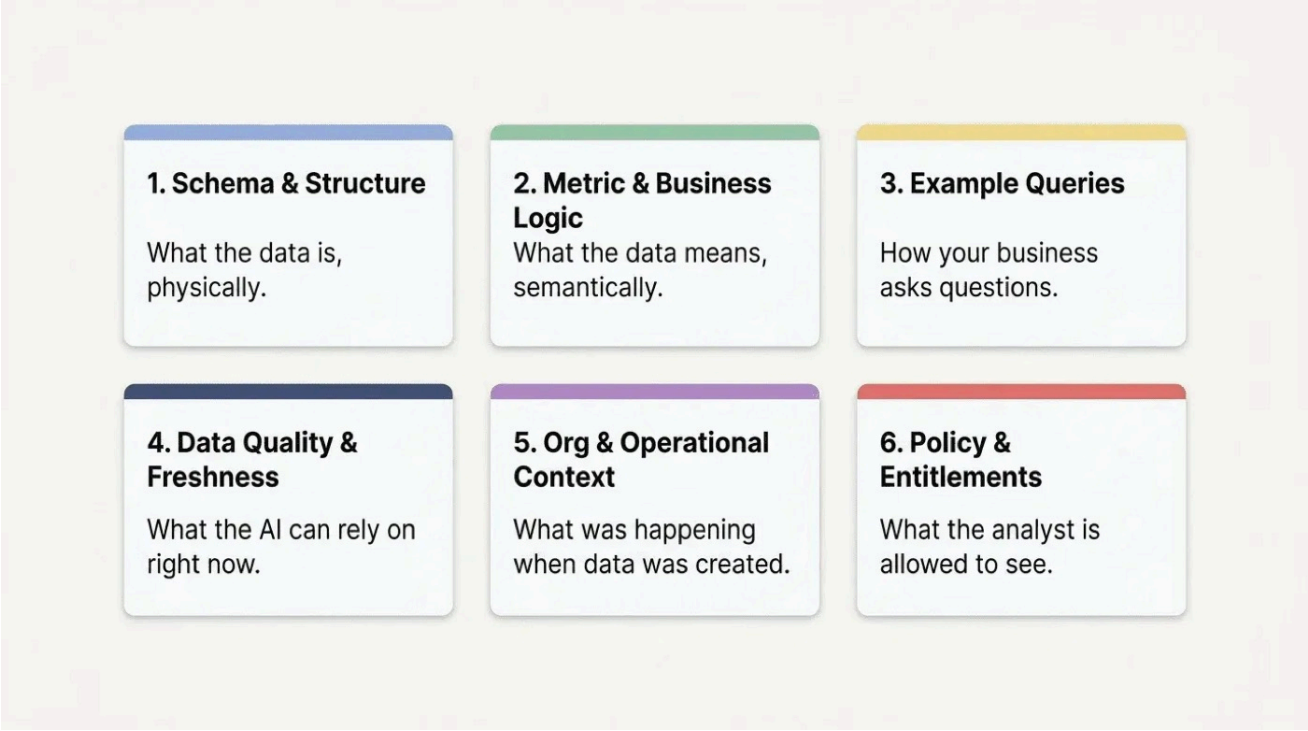


These are examples of context failures. They can be fixed. It takes intentional engineering effort, just as improving data quality or designing a schema does.

This blog explains what that work involves.

The Six Categories, and Why This Particular Taxonomy Matters

When people think about context for AI-on-data systems, they usually focus on column descriptions. Write some documentation, add it to the prompt, and call it done. But that is only Category 1 out of six. It is just the starting point, not the goal.



A complete context layer has to answer six fundamentally different questions:

1. What is the data, physically?
2. What does the data mean, semantically?
3. How does your business ask questions about this data?
4. What can the AI actually rely on right now?
5. What was happening in the business when this data was created?
6. What is this particular analyst allowed to see?

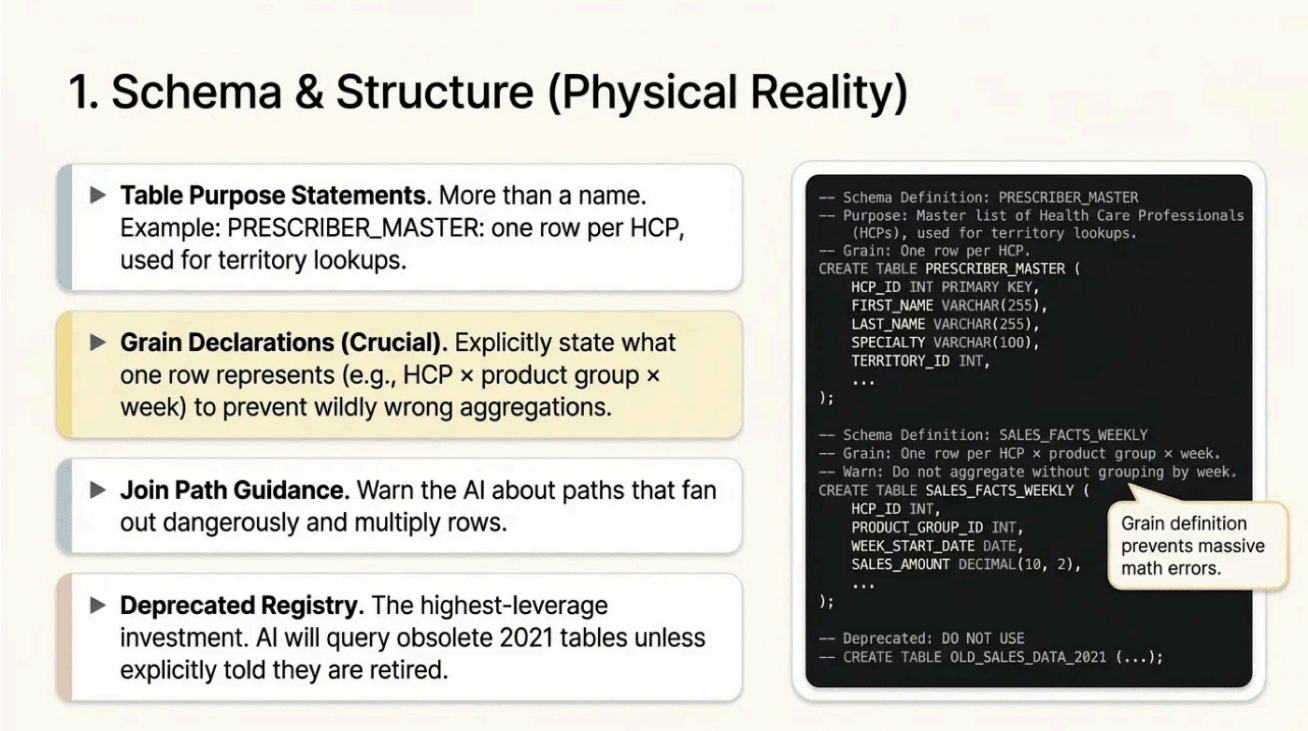
Each category requires a different manufacturing process, decays at a different rate, and needs a different maintenance discipline. You can have perfect Category 1 and still get confidently wrong answers because you are missing Category 2. You can have strong Categories 1 through 3 and watch the AI give answers based on stale data because you skipped Category 4. The taxonomy tells you what to build first, what to automate, and what requires new team habits.

Let’s go through each one.

Category 1: Schema and Structure

What it is

The schema and structure context are everything the AI needs to understand the physical shape of your data. This is the starting point. Without it, the AI is walking a warehouse it has never seen with no map.



The elements that belong here are more specific than most teams realize:

Table purpose statements. Not just the table name, but a sentence or two that says what the table is for and when to use it. “PRESCRIBER_MASTER: one row per healthcare provider. Use this table for territory, specialty, and NPI lookups. Do not use for prescription volume.” That one sentence eliminates an entire class of wrong joins.

Grain declarations. This is underinvested and critically important. The AI needs to know explicitly what one row represents. WEEKLY_PRESCRIPTIONS: one row per healthcare provider per product group per week. If the AI does not know the grain, it will aggregate at the wrong level. A question about total prescriptions by territory will return the correct SQL only if the AI knows the grain and can construct the correct GROUP BY. Get this wrong, and your numbers are off by factors, not percentages.

Column definitions with caveats. Not just what the column is called and what data type it holds, but what it measures, how it is calculated, and what the edge cases are. EQ_TRX: equalized total prescriptions, normalized for days supply to make 30-day and 90-day fills comparable, excludes mail-order channels, must be summed (never averaged without panel-size adjustment). That annotation is the difference between a number your analysts trust and a number they spend two hours investigating.

Join path guidance. When multiple join paths exist between tables, the AI needs to know which one is correct for which question type, and it needs warnings about join paths that fan out dangerously. A join between a fact table at weekly grain and a dimension table at daily grain can produce duplicate rows and wrong numbers. Document the right paths. Flag the dangerous ones.

Deprecated field registry. This is the highest-leverage investment in Category 1 for most real-world warehouses. Every data warehouse accumulates tables and columns that technically exist but should not be queried for new analysis. The AI has no way to know that FACT_TRX_PROD_BACKUP is from 2021 and was replaced by WEEKLY_PRESCRIPTIONS. Without an explicit registry of deprecated objects and pointers to their replacements, the AI will happily query them. It will find columns that match its expectations and return answers that are months out of date.

How do you manufacture it?

Category 1 is the most automatable of the six. Most of the raw material already exists in your metadata layer: information_schema, dbt documentation, data catalog entries, and existing README files. The manufacturing process for an initial pass looks like this:

Start by exporting your current metadata. Then use an AI coding assistant like Claude Code to analyze that metadata alongside your dbt models, transformation code, and pipeline definitions, and generate draft annotations for each table and column. The code-derived annotations are surprisingly good because pipeline code contains business logic that never surfaces in schema metadata. A dbt model that filters out mail-order channels before writing to WEEKLY_PRESCRIPTIONS will tell the AI something about EQ_TRX that no column description alone would convey.

From there, the process is human review and enrichment. Domain experts correct the drafts, add grain declarations, and document the deprecated registry. Plan for a few hours per domain area for the initial pass, not weeks.

How it decays and how you maintain it

Category 1 decays on schema change events: new tables, renamed columns, altered grain, deprecated objects. The maintenance discipline is to wire context updates into your CI/CD process. When a new table lands in production, the pipeline that creates it should trigger a context generation step. When a column is deprecated, the deprecated registry should be updated automatically as part of the deprecation workflow. This is not a manual quarterly review process. It is a hook in the deployment pipeline.

Category 2: Metric and Business Logic

What it is

Category 2 is what distinguishes an AI that can navigate your schema from one that actually understands your business. If Category 1 tells the AI what the data is, Category 2 tells it what the data means.

2. Metric & Business Logic (Semantic Meaning)

Canonical KPIs & Allowed Grains

Define filters, time windows, (e.g., NRR excludes churned accounts and requires a full 12-month observation window. Equalized prescriptions cannot be averaged by rep without panel-size adjustments.)

Calendars & Default Filters

Explicitly encode fiscal year offsets (e.g., Q3 means November) and default exclusions (test users, internal accounts).

Authoritative Sources

Resolve contested metrics. If CRM and Finance ERP both report revenue, define which system wins for which specific class of question.

Canonical KPI definitions. Every metric that matters needs a full specification: the name, the formula, the default filters, the time-window logic, and the edge cases. NRR (net revenue retention): computed on a cohort basis, includes expansion and contraction, excludes churned accounts, requires a full 12-month observation window to be valid, enterprise segment only unless segment filter is explicitly overridden. A one-line definition is not enough. Write it the way you would explain it to a new analyst on their first day.

Fiscal and operational calendar definitions. This category trips up more AI implementations than almost anything else because it looks trivial, but it isn't. If your fiscal year ends January 31, a question about Q3 means something different from what the AI expects. If your territory plan year runs from October to September, the AI needs to know that when someone asks about "last year's plan." If IQVIA prescription data arrives on Tuesday mornings, Monday afternoon answers to last week's questions are based on data that is a week old. All of this is business calendar context, and it is invisible to the model unless you document it.

Allowed aggregation grains per metric. Some metrics are only valid at certain aggregation levels. Equalized prescriptions can be summarized by territory, product group, and time period. You cannot average them across reps without adjusting for panel-size differences, or the metric will mean nothing. NPS scores can be averaged across respondents but not across time periods without careful weighting. Write these rules explicitly. The AI has no way to infer them.

Banned and deprecated metrics. This is the Category 2 equivalent of the Category 1 field registry. Metrics that look computable but should not be used anymore: old market share calculations that were replaced by a new methodology, metrics that two teams define differently and where one definition was

declared canonical, metrics that are technically correct but are not used for business decisions for reasons that live in people's heads. Name the banned metrics, say why they are banned, and point to the replacement. The AI will compute a banned metric unless you tell it not to.

Default filter logic. Every business has data that gets excluded from standard analysis: internal accounts, test users, pilot markets, specific geographies handled separately, and product lines in wind-down that skew trend lines. These exclusions live in analyst muscle memory. When someone runs a report, they automatically apply these filters. The AI will not unless you tell it to. Document the defaults and embed them in the context layer so the AI applies them unless the question explicitly overrides them.

Authoritative source rules for contested metrics. In almost every enterprise, some metrics exist across multiple systems, and those systems do not agree. Finance reports revenue from the ERP. CRM reports revenue from closed-won opportunities. They are different numbers because they measure different things at different points in the process. The AI needs to know which system wins for each question class, and under what conditions the other system is appropriate. This must be explicit. The AI cannot infer it from the data, and it will not flip a coin. It will pick whichever table seems most relevant to the question, but sometimes it picks the wrong one.

How do you manufacture it?

Category 2 cannot be fully automated because it encodes business decisions, not just technical facts. The manufacturing process involves pulling together the humans who own each metric: finance, analytics, product, sales ops, whoever maintains the definitions. Interview them. Ask them how they explain each metric to a new analyst. Ask them what questions trip up most people. Ask them what the common mistakes are when computing the metric.

Structured templates help. A KPI definition template with fields for formula, filters, time window, allowed grains, common mistakes, and authoritative source takes about 20 to 30 minutes per metric to fill out and produces something directly usable as context.

The highest-leverage investment in Category 2, and the one with the most immediate ROI, is getting your three to five most contested metrics fully specified. Revenue. Active users. Churn. Net retention. Whatever the metrics are that your leadership team asks about most and that cause the most "well, it depends on how you define it" conversations. Get those right, and you eliminate most of the confident-but-wrong answers the AI produces on high-visibility questions.

How it decays and how you maintain it

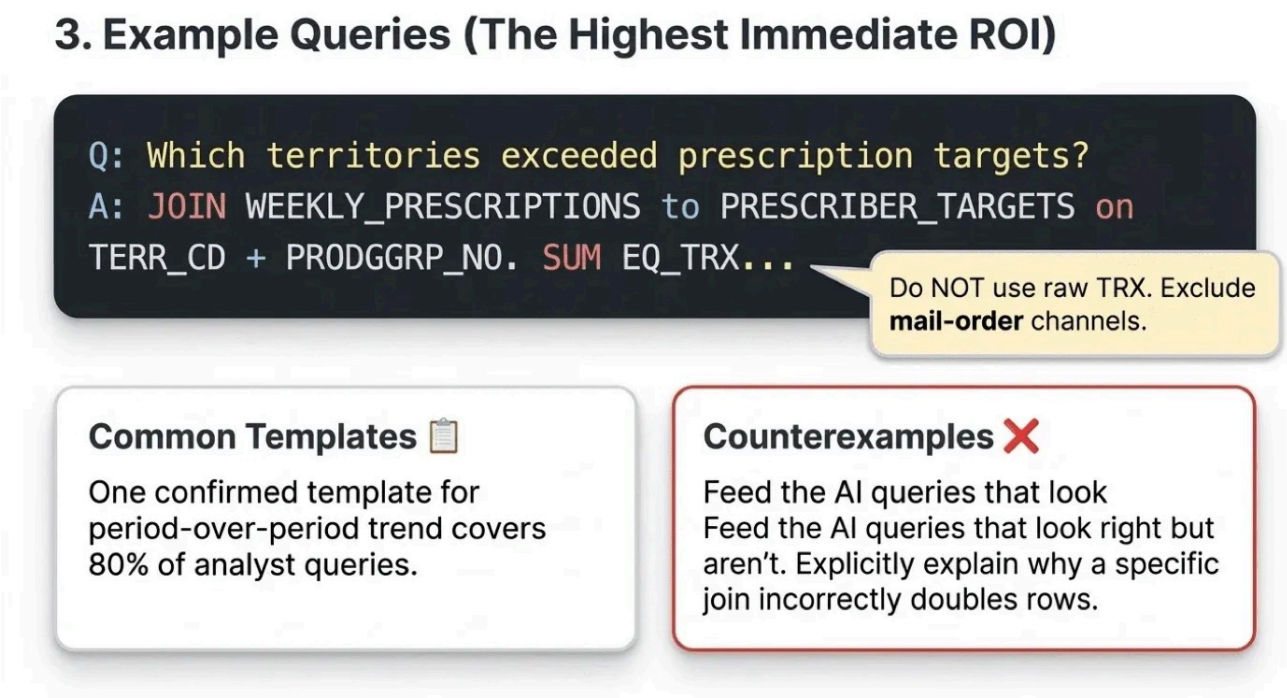
Category 2 decays on business logic change events: when a KPI definition is updated, when a new authoritative source is designated, or when a metric is deprecated or replaced. The maintenance discipline is version control with change history. When the definition of a KPI changes, you need to record not only what it is now, but also what it was before and when it changed. This is not just good practice for the context layer. It is critical to answer questions that span the change date, which is Category 5 territory, and we will get to it.

Category 3: Example Queries and Patterns

What it is

Category 3 is the most underrated and, in our experience, the one with the highest immediate return on investment per hour spent. If you do only one thing from this list, do this one.

An example query is a validated, confirmed-correct question-to-SQL mapping with annotations explaining the choices. Not just the SQL. The business question that motivated it, the SQL that answers it correctly, and notes on why it is correct: which tables were used and why, what the join path was and why that path rather than an alternative, what filters were applied by default, and what common mistakes this example is intended to prevent.



Here is what a useful example looks like:

Question: Which territories exceeded their equalized prescription targets last month?

SQL: JOIN WEEKLY_PRESCRIPTIONS to PRESCRIBER_TARGETS on TERR_CD, PRODGRP_NO, and PLAN_YR. SUM EQ_TRX WHERE WK_DT falls in the most recent complete fiscal month. Attainment equals the sum of EQ_TRX divided by ANNUAL_TGT times the fraction of the plan year elapsed. Use EQ_TRX, not TRX. Do not include mail-order channels.

That annotation prevents three common mistakes and teaches the AI the correct grain, metric, and join structure for this entire class of questions.

Common question templates. Beyond individual examples, you want parameterized templates for the major question types your analysts ask. Territory versus target. Period-over-period trend. Cohort comparison. Anomaly investigation (why did this metric change). KPI health check. If you have one confirmed, correct template for each of these patterns, you cover perhaps 80 percent of analyst questions. The long tail of edge cases can be handled by combining templates or extending them.

Counterexamples. These are underused but extremely effective. A counterexample is a query that appears correct but is not, accompanied by an explanation of why it fails. This join doubles rows because PRESCRIBER_MASTER has one row per NPI, but WEEKLY_PRESCRIPTIONS has one row per HCP_ID, and the relationship is not one-to-one. This calculation produces incorrect results for territories that underwent alignment changes mid-period because it uses point-in-time territory codes without accounting for the alignment history. Counterexamples teach the AI what to avoid. They are especially effective against the failure modes experienced analysts know about and the schema never shows.

Historical query patterns. If you have a query log, mine it. Which tables appear together in correct queries for which question types? When someone asks a territory performance question, what joins do your best analysts use? These patterns are a form of collective knowledge about what correct analysis looks like and can be used directly in context.

How do you manufacture it?

The manufacturing process for Category 3 is collaborative and iterative. Start by identifying your top 10 to 15 question types. For each one, have an analyst who owns that domain write the correct SQL and annotate it. Then review the annotation with someone who was not involved in writing it. The review catches assumptions that the author made implicitly but did not document.

Claude Code and similar AI coding assistants can help here. You can provide the schema, ask the assistant to generate candidate SQL for each question type, and then have domain experts validate and annotate the results. The validation step is essential. Treat the AI-generated SQL as a starting point, not a finished product.

As with Category 1, the initial investment is a few hours per question type. The ongoing investment adds examples as new question patterns emerge and updates existing examples when the schema or business logic changes.

How it decays and how you maintain it

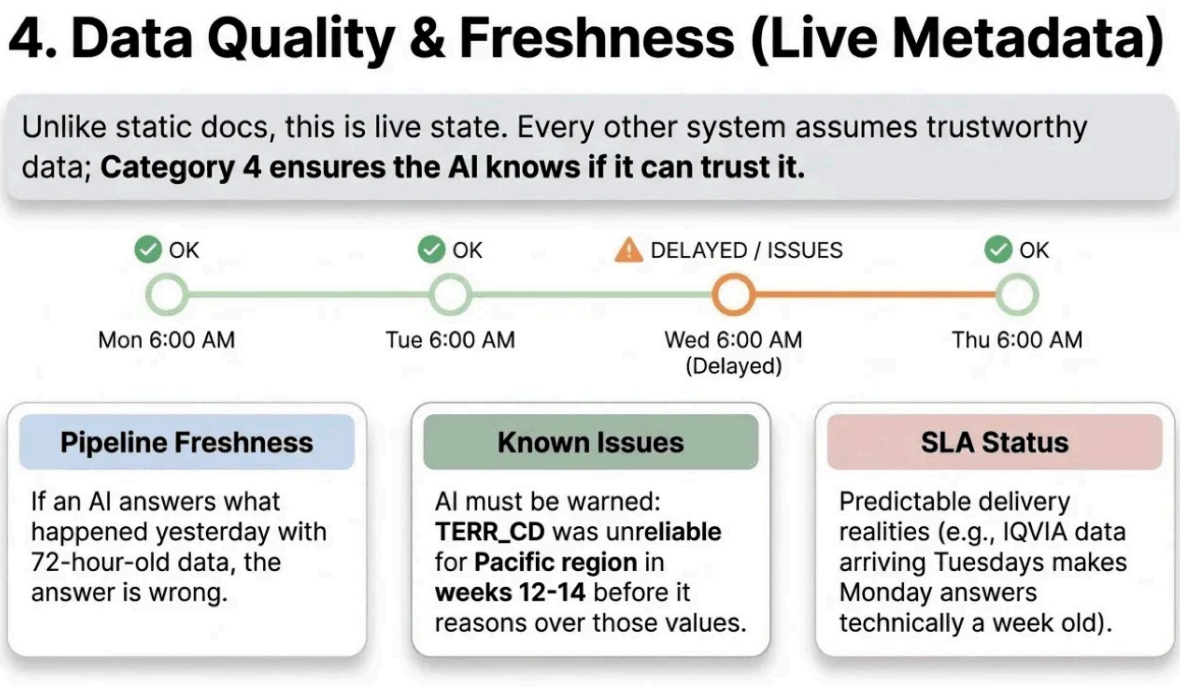
Category 3 decays more slowly than Categories 4 and 5. The main triggers for updates are schema changes that affect join paths or table names, and business logic changes that affect which metric or filter to use. Wire example query updates into the same CI/CD hooks you use for Category 1.

Category 4: Data Quality and Freshness

What it is

Category 4 is where DataKitchen's contribution to the context engineering conversation is most distinctive, because it is the category that almost everyone else is not building. Every other framework for AI-on-data context assumes that the data being queried is trustworthy. Category 4 is how the AI knows whether to trust it.

Category 4 context is not static documentation. It is live metadata about the current state of your data, surfaced into the AI context layer at query time.



Pipeline freshness timestamps. Every table in your AI-accessible schema should have a recorded last-refresh time and an expected refresh cadence. When an analyst asks a question and the AI queries WEEKLY_PRESCRIPTIONS, the context layer should tell the AI when that table was last refreshed. If the AI is answering a question about last week with data that is 72 hours old, it should say so. Without this context, the AI gives the same confident answer regardless of whether the data is current or stale.

Known data quality issues. If a specific column has a known problem within a specific date range or segment, the AI needs to know before it reasons over those values. TERR_CD was unreliable for the Pacific region in weeks 12 through 14 of 2026 due to an alignment migration. If that context is not present, the AI will reason over those values with the same confidence it brings to clean data, and it will produce wrong answers that it presents as correct.

Automated test results and quality scores. This is where data quality testing connects directly to context engineering. TestGen and similar tools run automated checks on your tables: completeness, freshness, referential integrity, value-range validation, and statistical distribution checks. Those test results are not just for the data engineering team. They belong in the AI context layer. If a table has a quality score below your threshold, or if a specific column has failed recent checks, the AI should caveat any answer derived from that data. A quality score per table, surfaced at query time, gives the AI a basis for expressing appropriate uncertainty rather than projecting false confidence.

SLA status and known delays. Most data environments have predictable delivery patterns. They are not failures, but they still affect how current the data is. IQVIA prescription data typically arrives on Tuesday mornings. On Monday afternoon, last week's data is the most recent available, but Monday morning data is actually the prior week. If the AI does not know this, it will describe Monday afternoon data as current when it is actually five days behind real-world events. That is not a data failure. It is a timing reality that the context layer needs to encode.

Data lineage and transformation history. Where did this data come from, and what happened to it on the way to the table the AI is querying? For derived metrics, especially, the AI should be able to explain the lineage of an answer. This metric is computed from IQVIA TRx adjusted for territory alignment as of last Tuesday's load. That provenance statement turns an answer into a verifiable claim.

How do you manufacture it?

Category 4 is fundamentally different from Categories 1 through 3 because it cannot be manufactured once and then updated periodically. It must be live. The manufacturing process is really a tooling and automation problem, not a documentation problem.

The right architecture continuously feeds pipeline observability data into the context layer. Every time a pipeline runs, the completion timestamp, quality check results, and any anomalies detected are written to a context metadata store that the AI can query at analysis time. DataOps Observability and similar tools generate exactly this data as a byproduct of normal pipeline operation. The engineering work is building the connector that makes it available to the AI context layer.

For teams that do not yet have pipeline observability instrumented, the minimum viable version of Category 4 is a freshness table: a simple registry that records the last-updated time for each AI-accessible table, maintained by the pipelines themselves as a final step in each load. That is not the full picture, but it is better than silence.

How it decays and how you maintain it

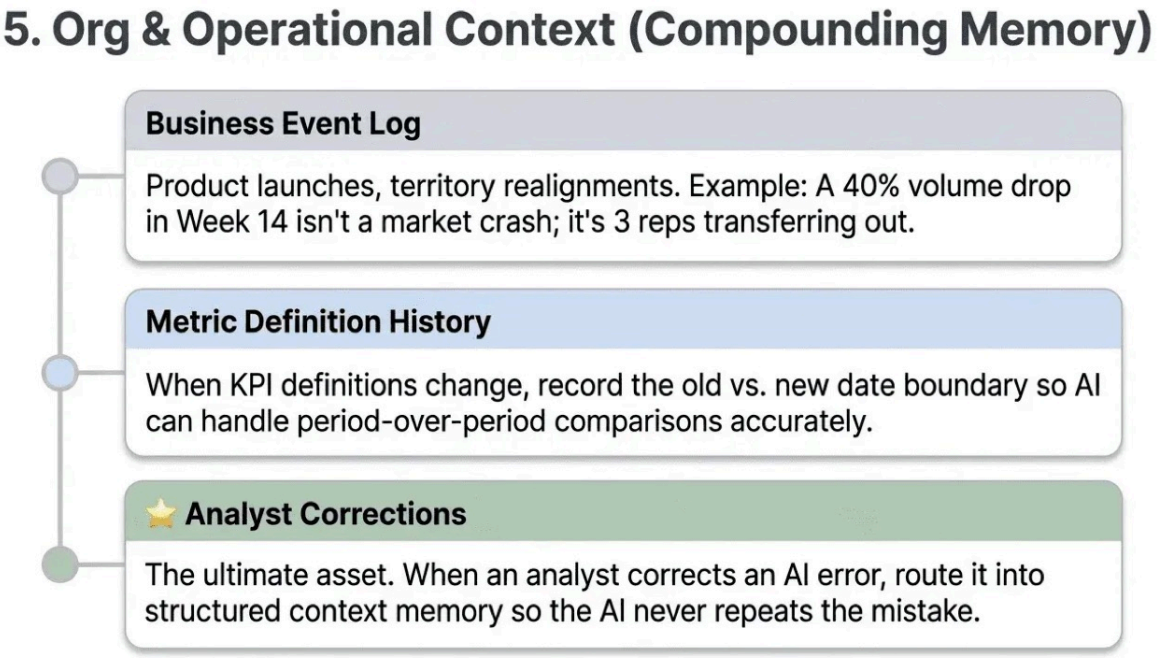
Category 4 does not decay in the same way documentation does. It is a live state. The maintenance challenge is not updating it; it is ensuring the automation that populates it continues to run correctly as pipelines evolve. The risk is that a pipeline is refactored, the freshness write step is inadvertently dropped, and the context layer starts serving stale quality metadata. Treat the Category 4 pipeline with the same monitoring discipline you apply to your data pipelines.

Category 5: Organizational and Operational Context

What it is

This is the category that distinguishes an AI that reports from one that reasons. And it is the hardest one to build, because the context it requires cannot be retrieved from databases. It can only be captured prospectively, at the moment business events occur.

A dip in prescription volume during a territory realignment is not the same as a dip caused by competitive pressure. A spike in ticket volume the week after a product launch is not the same as an organic increase in support demand. Without organizational and operational context, the AI cannot tell the difference. It will see the number and report it accurately. It will not understand what caused it.



Business event timeline. Product launches, pricing changes, feature rollouts, organizational restructurings, territory realignments, promotional windows, competitive events, and regulatory approvals. Any event that changes the relationship between business activity and the data that records it. This timeline should be stored as a structured log, queryable by date range, product, territory, and event type. When the AI analyzes a time series and detects an anomaly, it should be able to check the event log and surface relevant context. The 40 percent drop in Cobenfy territory scripts in week 14 is explained by the three HCPs who transferred out of the territory in the realignment that week. Without the event log, the AI can only report the drop. With it, it can explain it.

Metric definition history. When a KPI definition changes, the change needs to be recorded: when the new definition replaces the old definition, and when the transition date occurs. This is not the same as updating the current definition in Category 2. It is a versioned history that allows the AI to correctly answer questions that span the change date. If an analyst asks how Q1 2026 revenue compares to Q1 2025, and

the revenue definition changed in March 2026, the AI needs to know that it is comparing numbers computed on different bases. Without that context, the comparison will lack caveats and produce a number that is technically accurate but misleading.

Canonical source rules and change history. Which system is authoritative for which metric, and when did that designation change? If you migrated your revenue source from the ERP to a new data product in Q3 2025, questions spanning that migration require knowing which source to use for each period. This operational context is rarely documented and frequently confuses both human and AI analysis.

Analyst corrections and annotations. This is the most compounding asset in the entire context stack, and it is the one most teams fail to capture. Every time an AI gives a wrong answer and an analyst corrects it, that correction provides information about the right approach. If those corrections are captured, stored, and fed back into the context layer, the AI stops making the same mistake. If they are not captured, every user encounters the same error independently and fixes it in a conversation that disappears.

OpenAI's internal data agent demonstrates why this matters at scale. Their learning memory layer, which stores corrections from past queries and applies them to future queries, reduced query time on certain tasks from 22 minutes to 90 seconds. The memory is not making the model smarter. It is preventing the model from redoing work that has already been done correctly.

How do you manufacture it?

The event timeline can be bootstrapped from existing records: git commit history, product launch announcements, CRM notes, HR systems for org changes, and territory management systems for alignment changes. The initial population is a research and curation exercise, typically a few days' work for a single domain.

The harder problem is prospective capture. Once you have an event log, someone needs to maintain the habit of writing to it when business events occur. This is not a technical problem. It is an organizational process problem. The event log needs to be easy to write to (a simple form, an integration with your ticketing or project management system, a Slack bot that asks for structured input when certain kinds of changes are announced), and someone needs to own the discipline of keeping it current.

Analyst corrections require an explicit feedback mechanism in whatever interface analysts use to interact with the AI. When an analyst says, "This answer is wrong, the correct approach is X," that correction needs to be captured in a structured form and routed to the context layer rather than disappearing into a chat history.

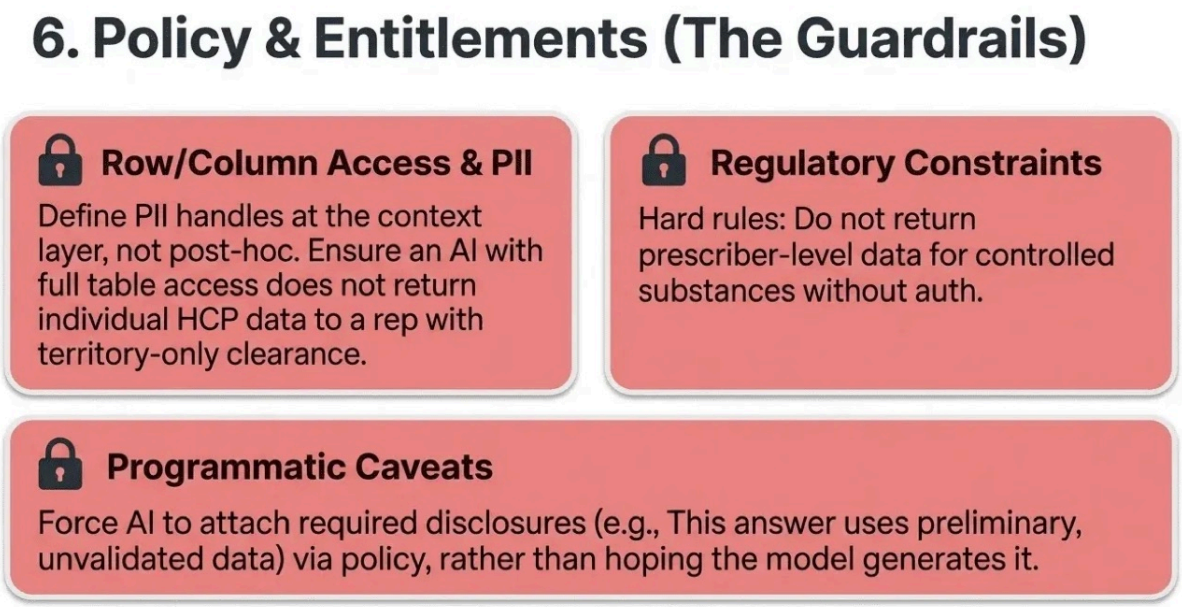
How it decays and how you maintain it

Category 5 has a different decay dynamic than the others. The business event timeline and metric definition history do not get stale in the same way that pipeline freshness data does. Old events remain relevant for answering historical questions. The maintenance challenge is gaps: events that happened but weren't recorded, corrections made conversationally but not captured. Audit the event log against known business history to find the gaps. Making it easy to add entries is the best prevention.

Category 6: Policy and Entitlements

What it is

Category 6 is the context that specifies what a particular analyst is allowed to see, compute, and disclose. In most implementations, this is treated as a security layer separate from context, enforced at the database level through row-level security and column masking. That enforcement layer is necessary but not sufficient. The AI also needs to understand the policy context that governs what it should do, not just what it is technically permitted to access.



Row- and column-level access rules. Which data is this analyst or agent allowed to retrieve? An AI that can access all rows in a prescriber table should not return individual-level HCP prescribing data to a sales rep who only has territory-level access. Row-level security enforces this at the database level. Context-level policy documentation tells the AI how to behave when the enforcement layer returns partial results: do not attempt to reconstruct suppressed rows; do not infer individual values from aggregates; and flag when results are partial due to access controls.

PII handling requirements. Which columns contain personally identifiable information, and how must they be treated in query results? This needs to be defined at the column level in the context layer, not discovered post-hoc when a report accidentally exposes individual-level data. The AI should know before it constructs a query whether the tables involved contain PII and what the handling requirements are.

Regulatory constraints. In pharma, finance, healthcare, and other regulated industries, certain computations or disclosures have compliance implications. “Do not return prescriber-level data for controlled substances without verifying rep-level authorization.” “This market research data is for internal

analysis only and must not be shared externally." "All preliminary data answers must be accompanied by a disclosure that the data has not been validated." These are constraints that need to be enforced programmatically, not left to the AI's judgment or the analyst's memory.

Required disclosure caveats. When an answer must be accompanied by specific language, that language should be attached programmatically, not generated by the model. "This answer uses preliminary data that has not been validated for the period ending March 31, 2026." "This metric is currently under review and may be revised." The AI should not be inventing these caveats; it should be surfacing them from the policy layer when they apply.

How do you manufacture it?

Category 6 manufacturing typically involves three stakeholders: the data governance team (which owns the policy definitions), the security team (which owns the enforcement layer), and the analytics team (which owns the user experience). The policy documentation itself is usually an extension of existing governance artifacts: data dictionaries that already identify PII columns, security policies that already define row-level access rules, and compliance documentation that already describes regulatory constraints. The engineering work is translating these existing artifacts into a form that the AI context layer can consume.

For regulated industries, this category should be designed before you start building Categories 1 through 5. It is much harder to retrofit access controls and disclosure requirements than to design them in from the beginning.

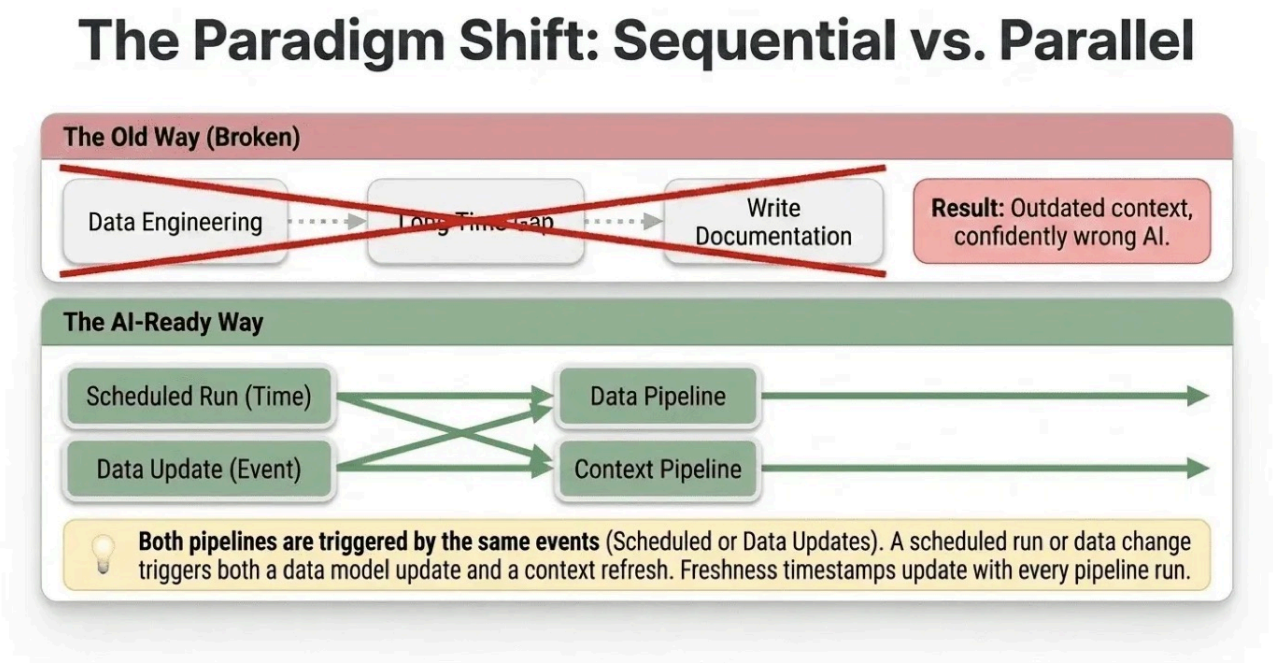
How it decays and how you maintain it

Category 6 decays on policy change events, such as when RBAC roles change, regulatory requirements are updated, or new data classifications are introduced. It tracks your existing governance and compliance change management processes. The maintenance discipline is making context-layer updates part of any governance policy change workflow, just as documentation updates are required for any data model change.

Context Engineering Runs Parallel to Data Engineering

Here is the shift in thinking that brings all six categories together.

For as long as data warehousing has existed, data engineering and context have been sequential rather than parallel. Engineers build pipelines, land data in tables, and then, eventually, someone writes documentation. Usually later than planned, usually less complete than intended, and usually not updated when the pipeline changes.



That sequence does not work for AI systems. An AI assistant that queries a table without current context is not usable. It is worse than useless, because it returns incorrect answers with confidence rather than no answers at all. The context is not optional documentation. It is a required input to a production system, just as data is.

This means context engineering needs to run in parallel with data engineering, not after it.

Every time a new table is created, a context artifact is generated alongside it: a table purpose statement, a grain declaration, column annotations, and join path documentation. Not weeks later. At deployment time. The same CI/CD process that creates the table creates the context.

Every time a pipeline is updated, the context artifacts are updated as well. Column definition changed? Context updated. Grain changed? Context updated with the new grain and a note about what changed and when. Table deprecated? Context updated with the deprecation date and the pointer to the replacement.

Every time a pipeline runs, the Category 4 context is refreshed automatically. Freshness timestamps are written. Quality check results are recorded. SLA status is updated. This happens as part of the pipeline execution, not as a separate manual process.

When business events occur, the Category 5 context is written. The product launch is recorded. The territory realignment is logged. The metric definition change is versioned. This requires organizational habits more than technical infrastructure, but the technical infrastructure needs to be ready to receive the input.

The Context Manufacturing Matrix

Category	Manufacturing Method	Decay Trigger	Maintenance Phase
1. Schema & Structure	AI Assist + Human Review	Schema Changes	Automated CI/CD Hook
2. Metric & Business Logic	Domain Expert Interviews	Business Logic Updates	Version Controlled History
3. Example Queries	Log Mining + Validation	Schema/Metric Updates	CI/CD Hooks
4. Data Quality & Freshness	Pipeline Observability	Continuous/Live	Live API Feed
5. Org & Operational Context	Prospective Logging	Real-World Events	Slack Bots / Ticketing
6. Policy & Entitlements	Governance Artifact Auth	Policy/RBAC Changes	Governance Workflows

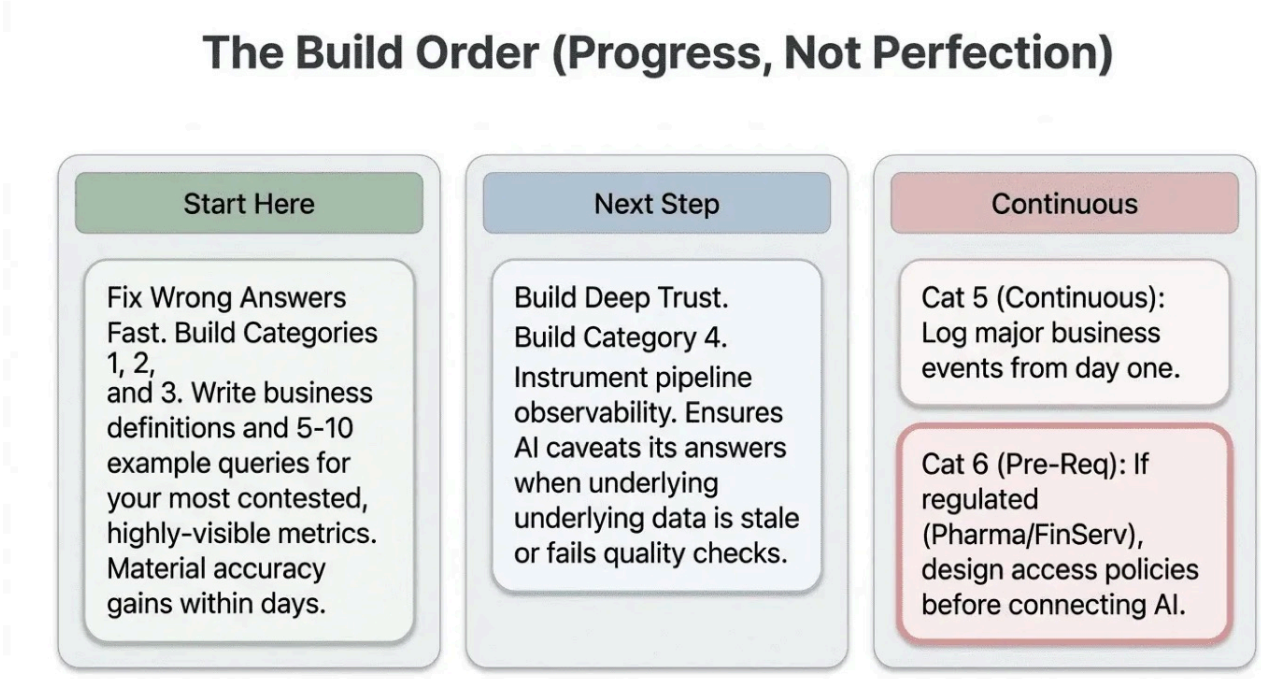
When an analyst corrects an AI answer, the correction is captured, structured, and routed into the context layer instead of dying in a chat history.

In practice, this looks like a context pipeline that runs alongside your data pipeline. Your data pipeline extracts, transforms, and loads data. Your context pipeline extracts, generates, validates, and loads context. They share triggers. A data pipeline completion event triggers both a data quality check and a context freshness update. A schema migration triggers both a table update and a context artifact update. A deployment event triggers both a data model change and a context documentation change.

The teams that build this infrastructure end up with something their competitors cannot easily replicate. Every business event captured in the context layer makes the AI more accurate when answering questions about that period. Every captured analyst correction reduces the AI's likelihood of repeating the same mistake. Every validated, stored example query improves the AI's performance on the question types your analysts care about most. The context layer accumulates value over time and cannot be replicated retroactively. You can copy a schema. You cannot reconstruct the operational history and institutional knowledge that has been prospectively captured in a context layer that has been running for two years.

The Build Order

The practical question is where to start. The answer depends on where your pain is greatest, but the general priority sequence holds across most implementations.



Start with Categories 1 through 3 if your AI is giving technically plausible but incorrect answers. These three categories resolve most wrong-table, wrong-join, and wrong-definition failures. They are the most automatable, produce the fastest results, and offer the clearest return on investment. Table purpose statements, grain declarations, and five to ten validated example queries for your most common question types will materially improve AI accuracy within days of deployment, not months.

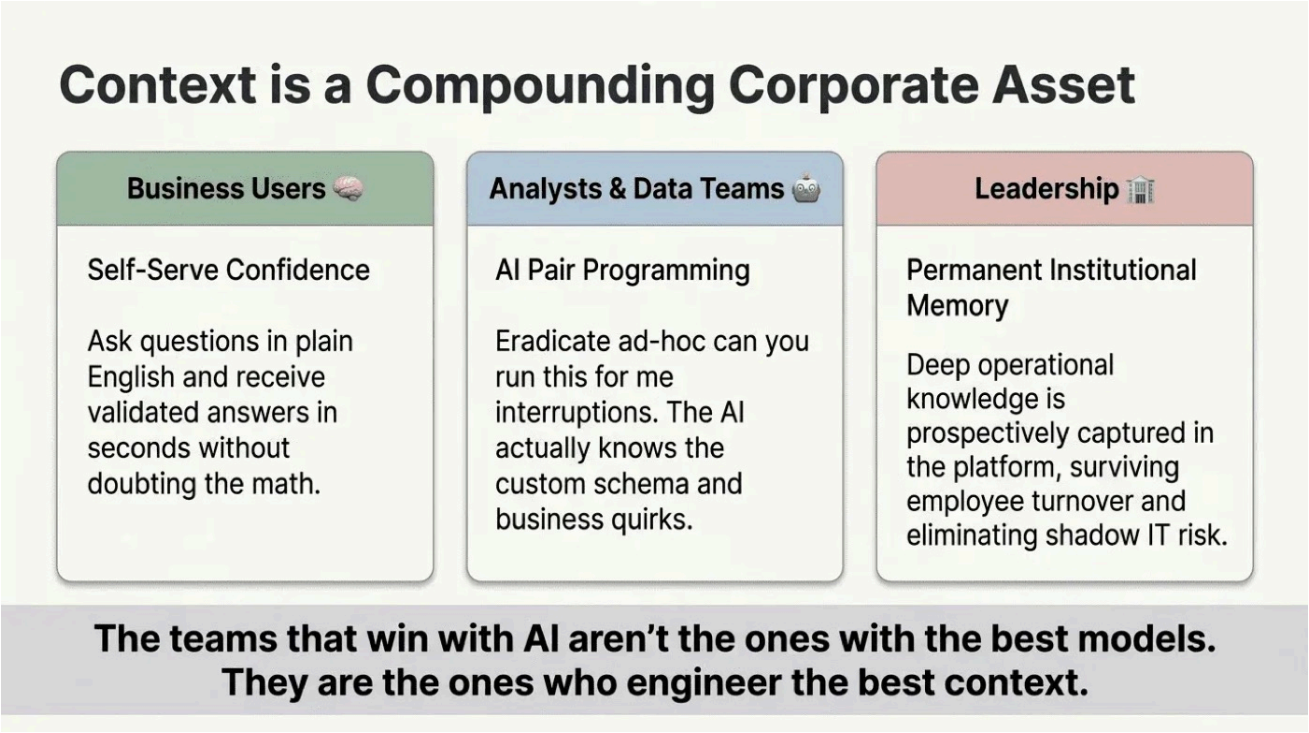
Add Category 4 once the basics are working, if your AI gives correct answers on clean data but fails when data is stale or has quality issues. This requires pipeline observability tooling, but the payoff is significant: the AI stops presenting stale answers with the same confidence as fresh ones, and it can caveat results when quality checks have recently failed on the underlying data. Without this, your analysts eventually stop trusting the AI, not because the model is wrong, but because they cannot tell when the data it is querying is reliable.

Build Category 5 continuously, starting from day one. The business event timeline and analyst correction capture need to start immediately, even if they start simply. Every week of operational history that goes unrecorded is context permanently lost. A simple log that records major business events, with date and brief description, is infinitely better than no log. You can always enrich it later. You cannot reconstruct it.

Design Category 6 before you start, if you are in a regulated industry. Access controls and disclosure requirements are much easier to design in than to retrofit. In pharma, finance, and healthcare, especially, the time to think about PII handling and regulatory constraints is before you connect the AI to production data, not after the first compliance incident.

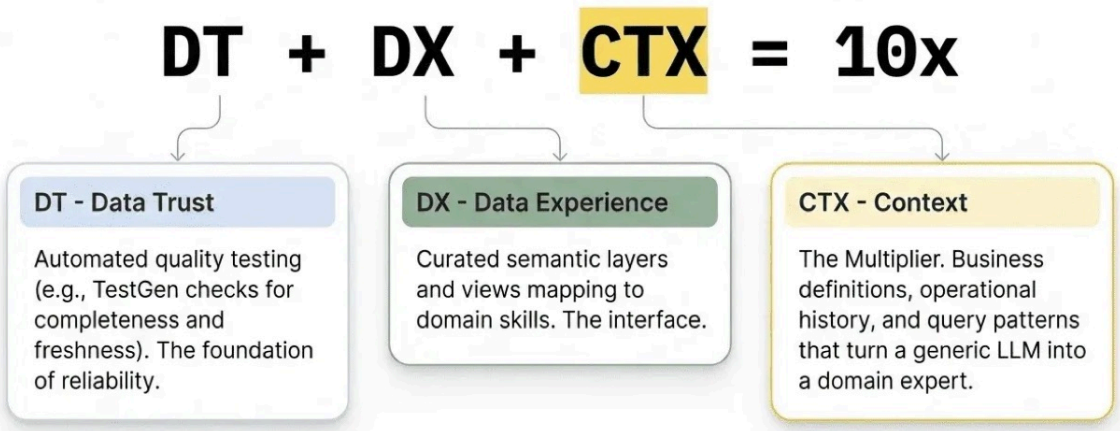
Closing: Context Is Infrastructure

The six categories of context in this post are infrastructure, not documentation. This layer sits between your data and your AI. It turns schemas into meaning, tables into business concepts, and raw query results into answers your analysts can trust.



Like any infrastructure, context requires engineering effort, regular maintenance, and discipline. But unlike most infrastructure, its value grows over time. A well-built, well-maintained context layer becomes a lasting asset. It holds years of knowledge in a form AI can use, even after the people who knew it have left.

The teams that will win with AI are not the ones with the best models. Models are becoming commodities. The teams that will win are the ones that do the foundational data work. Data Trust. Data Experience. Context. When built and maintained in parallel, those three things turn AI on your data from a demo that looks impressive but disappoints in production into a system your analysts actually rely on.



If you want to see these ideas worked through in more depth, with live schema examples and before-and-after walkthroughs from real Snowflake and Databricks deployments, watch the recording of the April 2026 webinar: 10x Your AI Data Analysis: The Context Engineering Formula That Works.

AI AND DATA QUALITY

Stop Burning Tokens: Data Observability and Quality as Your AI's System of Record

Read the original article at: https://datakitchen.io/blog/stop-burning-tokens-data-quality-observability-ai-system-of-record/

Everyone Is Adding AI. That Is the Problem.

Atlassian has an AI assistant. Docker has one. Monte Carlo has AI, root cause analysis, lineage/impact analysis, and cost optimization. Every tool wants to be your AI interface and include every feature.

Here is what I notice when I use those tools: I ignore the embedded AI and just ask Claude. The tool's AI does not have my context. It does not know my terminology, my priorities, or what happened last week. And now companies are starting to realize that convenience carries a bill. A DoiT survey (Sapio Research, Feb 2026, 500 finance leaders at orgs with 1,000+ employees) found that 79% of enterprises overspent on AI in 2026, even as per-token prices kept falling. Volume is the culprit, and it only gets worse when every tool in your stack burns tokens independently.

Context is the new oil. The question is: who does the refining?

The Factory Analogy

In a factory, you do not have nuts and bolts in random sizes. You standardize them, and that makes everything downstream faster and more reliable. You also do not manufacture the nuts and bolts yourself. You buy commodity parts and save your expertise for the next level of value.

I ran a demo recently where I typed one prompt to Claude: “Look at all the hygiene issues and data test failures in TestGen and list the first three I should investigate.” Claude used the MCP interface (the Model Context Protocol, which lets AI tools call external services directly) and came back with three findings: duplicate customer IDs, two completely empty columns, and a product category field with inconsistent values. I asked Claude to fix the category field. It wrote SQL to normalize all variants to the dominant form, which is exactly the kind of query I write myself. Then I asked what custom test I should add. Claude scanned 214 existing single-column tests and suggested a cross-table check: discount amount should not exceed the maximum allowed discount for that product. When I asked how it knew to recommend that, it said the field names were descriptive enough to make the relationship obvious.

That is the right division of labor. TestGen (DataKitchen’s data quality and observability platform) held the test (rules) and test results. The AI read that context and reasoned from there. Each did its own job and nothing more.

Two Jobs That Should Stay Separate

Most vendors blur a boundary that should stay clear. On one side sits determinism, auditability, and a system of record. On the other sits reasoning, synthesis, and discovery.

TestGen lives on the first side. When you add a test, it runs. Every time. At scale. It never forgets, never drifts, and keeps a full history: when a test ran, when it failed, what the data looked like at that moment. For regulated industries and compliance-conscious teams, that auditability is required, not optional.

AI lives on the second side. It is good at reasoning over context and surfacing what you did not think to look for. Ask an AI to maintain a test suite indefinitely and you will eventually be disappointed. Ask it to reason over profiled, tested data exposed via MCP, and you will be impressed.

Embedding AI inside every tool collapses this boundary. You get a reasoning engine trying to act as a system of record, burning tokens every time, and producing slightly different answers each run. That is the wrong architecture.

MCP Changes the Equation

Connect TestGen as an MCP service and whatever AI you already use, Claude, Cursor, Databricks Genie, Snowflake Cortex, can talk to it directly. Your AI does not change. Your workflow does not change. TestGen becomes the backend service your AI calls when it needs to know what the data looks like, what tests exist, and what has failed.

This is also why we decided not to build lineage into TestGen. The answer to “what will this error affect” lives in the code, and any developer already has an AI connected to their repository. No reason to duplicate that inside a data quality tool. Three pieces of context, each living where it belongs, each accessible via the right tool. The architecture does not have to be complicated; it just has to respect the boundary.

Tests the agent can call for itself are also the only way it stops grading its own homework.

Save Tokens. Get Consistency.

I asked Claude what tests I should add. Claude recommended the cross-table discount check and added it to TestGen. That test now runs on every cycle without burning a single token. I paid for the reasoning once and the test runs forever.

That is the compounding advantage. Every test you encode is a question you never pay to answer again. TestGen always returns the same answer, auditable and traceable to the specific run.

All the features. A tenth the price. And we save you the tokens.

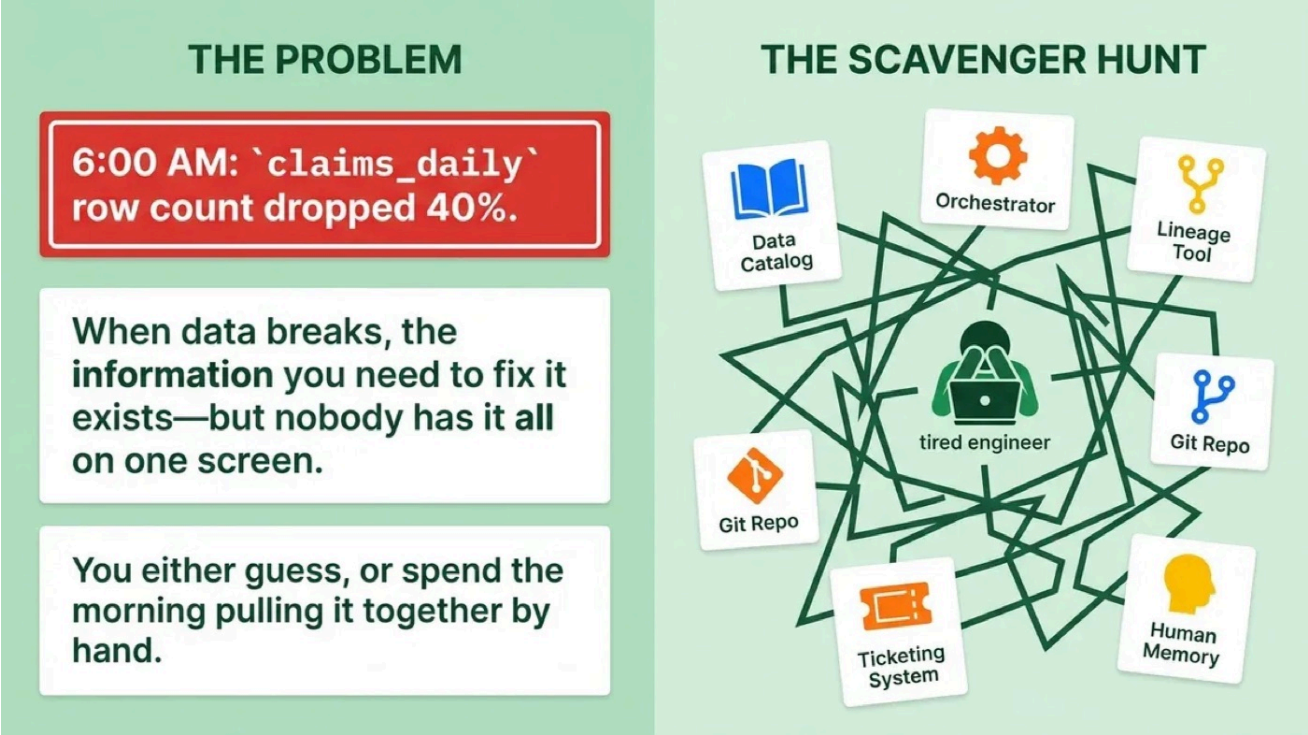
AI AND DATA QUALITY

Put Your AI At The Center Of Your Data Quality And Observability Decision-Making Process

Read the original article at: https://datakitchen.io/blog/put-your-ai-at-the-center-of-your-data-quality-decisions/

You need to know which job loaded that table, whether it finished or died halfway through, what sits downstream of it, who looks at those reports, and whether anyone opened a ticket about this last quarter. That information exists. It lives in your catalog, your orchestrator, your lineage tool, your git repo, your ticketing system, and in the heads of two people on your team. Nobody has all of it on one screen. So you either guess or spend the morning pulling it together by hand.

The quality of the decision tracks the amount of context behind it. That's the whole argument of this post. Your LLM is the one thing that can hold all that context at once, so put it at the center of the process.



Find problems before they reach production

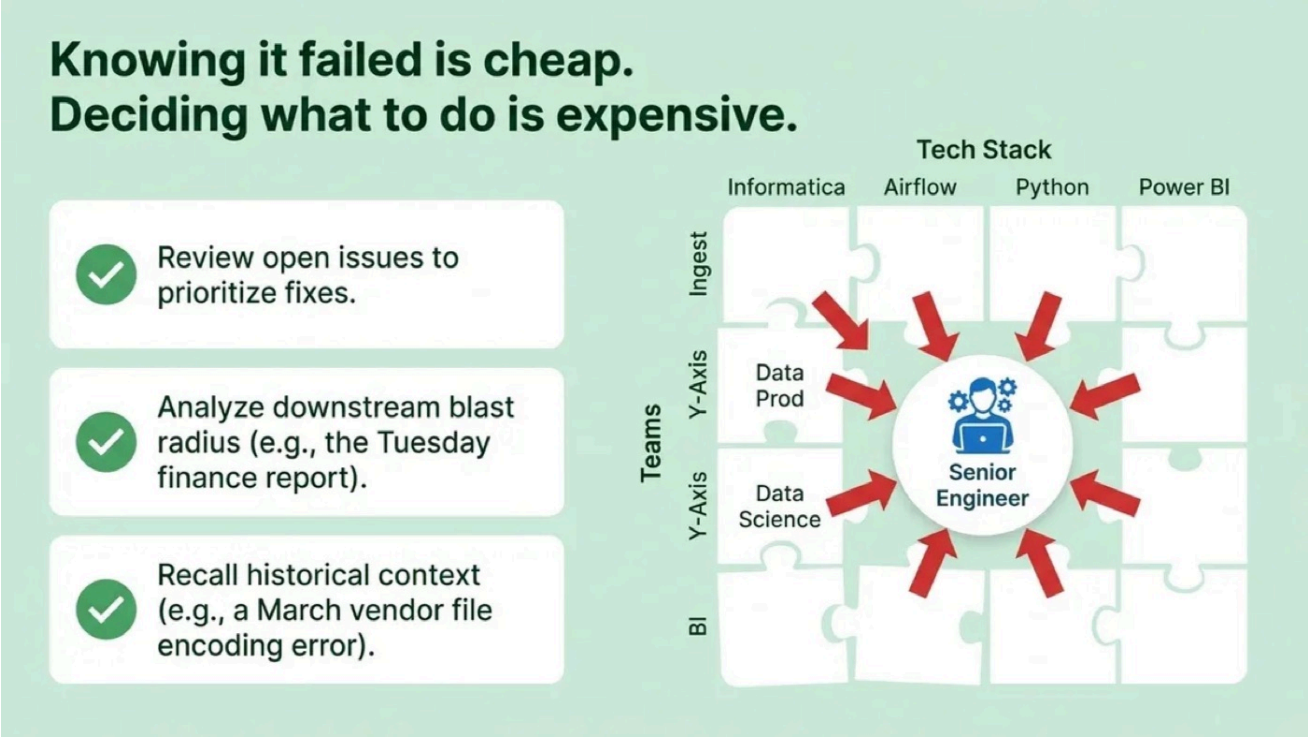
That is the job. We've been doing data engineering consulting since before DataKitchen existed, and the lesson hasn't changed. Monitor the data and the tools acting on it. Set tripwires that stop the run when something breaks. A bad file that never lands in the warehouse costs you nothing. The same file, discovered by the CFO on a dashboard, costs you a week.

When data breaks, the useful question is which process ran wrong. Column-to-column lineage tells you what feeds what. It doesn't tell you that the 3am Airflow task retried twice and then wrote half a table. A test-informed orchestrator gets you part of the way there. We've written before about why lineage alone isn't enough and what a data journey adds.

Even with that in place, the decision is still yours.

The decision is the hard part

Knowing a test failed is cheap. Deciding what to do about it is expensive. Someone has to review the open issues and pick which one to fix first. Someone has to review the existing tests and decide whether they're the right ones or write new ones. Someone has to know that `claims_daily` feeds the Tuesday reimbursement report and that finance reads it at 8am. And someone has to remember that this same table did this same thing in March, and the fix was a vendor file that arrived in the wrong encoding.



Part of the reason is how the system is built. Most data architectures are loosely coupled, and the seams follow the org chart. An ingest team, a data production team, a data science team, and a BI team, some reporting to IT and some to the business. Each one picked its own tools in its own decade. Airflow in one department, Informatica in another, Power BI in a third, and Python scripts everywhere in between. Nothing in that stack knows about the rest of it. When something breaks, the failure crosses four teams and three technologies, and the person diagnosing it has credentials for maybe two.

That person is usually your most senior engineer. They're the bottleneck because they're the only one with the full picture. The picture isn't written down anywhere.

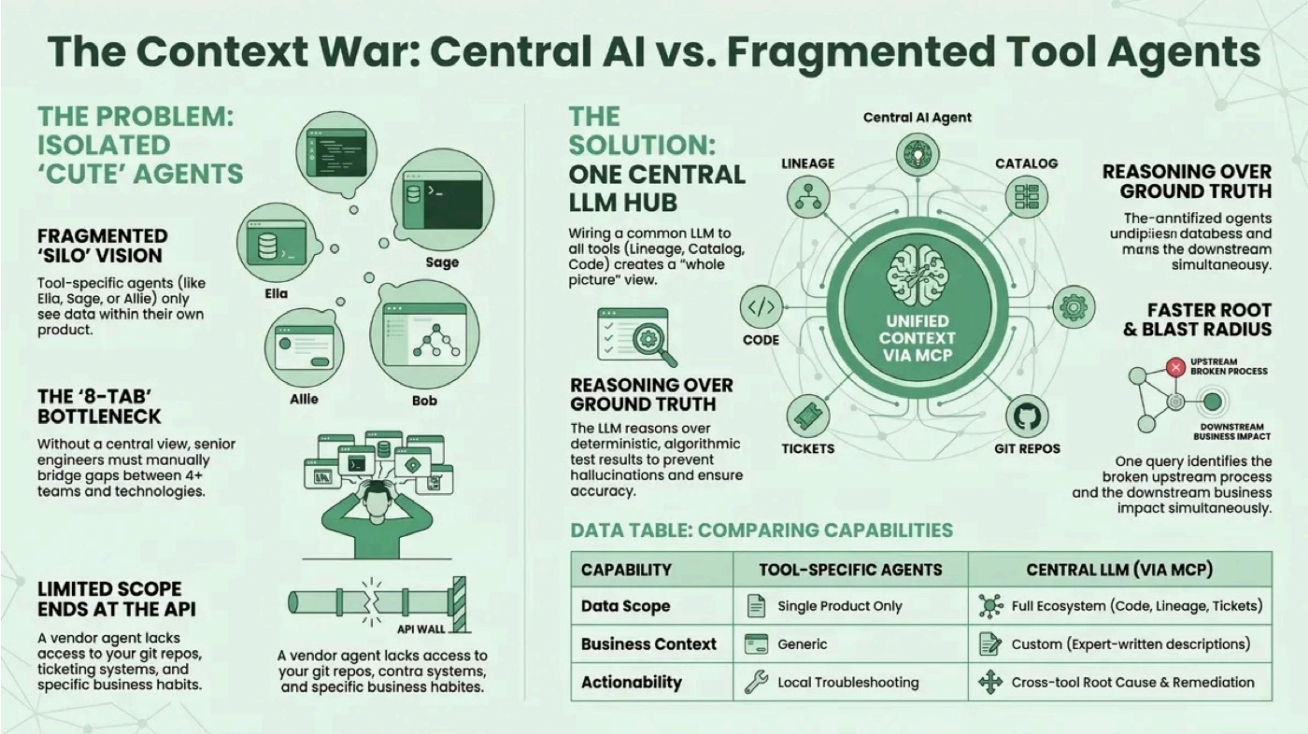
Only a central LLM holds all the context

Look at what the decision draws on. Data catalog entries and owners. Profiling results. Test definitions and their history of passes and failures. Data lineage: what connects to what. Process lineage: what just ran and what failed. The SQL and Python that run in the pipeline. Your ticketing system, both the past tickets and the one you're about to open. And the context nobody can buy: your customers, your domain, your department's habits and workarounds.

No data quality tool has that. No catalog has that. No orchestrator has that.

Every vendor now ships a cute named agent. Elementary has Ella. Sifflet has Sage. Alation has Allie. Soda has Ask AI and Contract Copilot. Collibra has AI Copilot, Atlan has Atlan AI, and Informatica has CLAIRE Copilot. Ask any of them a question, and it answers well inside its own product. Its scope ends

where the product ends. Maybe it has lineage, maybe it doesn't. Maybe it can read the process code, maybe it can't. Maybe there's a catalog behind it, maybe there isn't. Each one meters its own usage and sees its own slice.



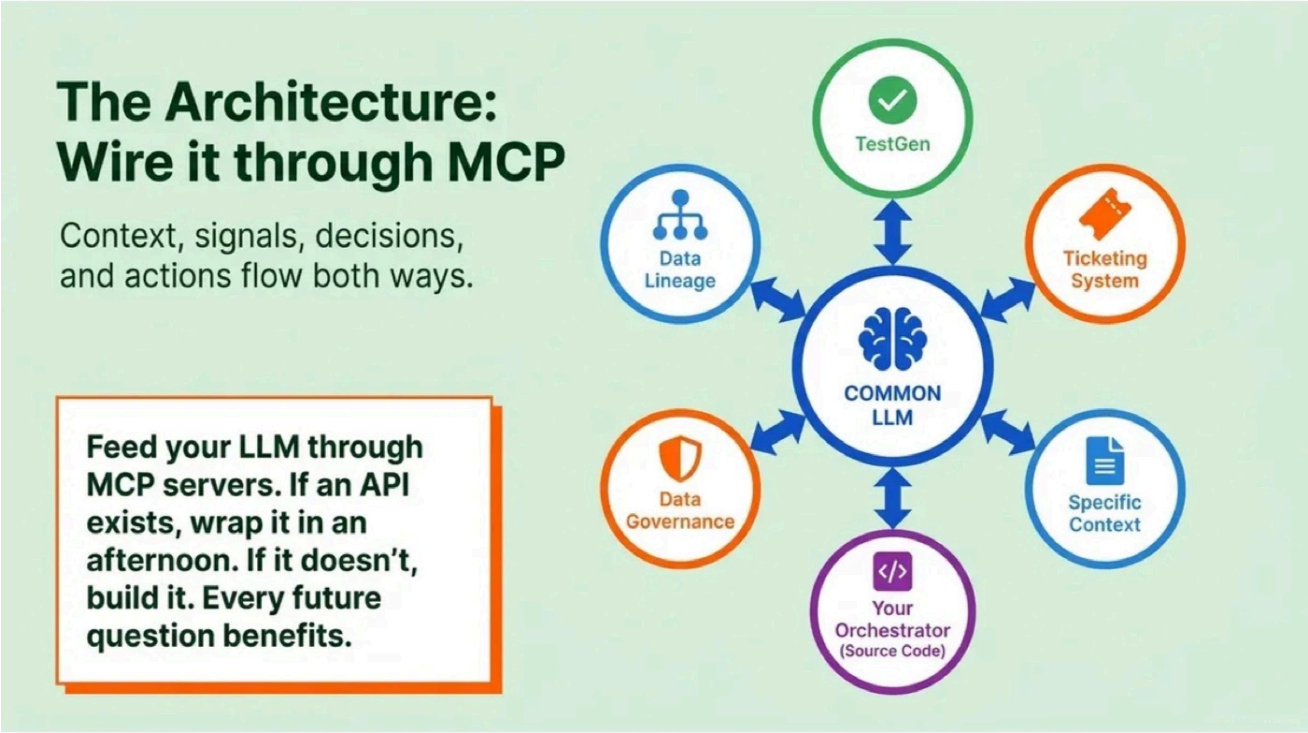
This isn't a theory. OpenAI wrote up the data agent it runs internally for 3,500 people across 70,000 datasets. Their finding was that even a strong model gave badly wrong answers, like misestimating user counts, until they surrounded it with layers of context: schema metadata and lineage, historical queries, expert-written table descriptions, and a nightly job where Codex reads the pipeline code that produces each table and records what it contains and how fresh it is. The model didn't change. The context did, and the answers followed.

The LLM your company already standardized on can do the same thing. Wire it to everything, and it sees the whole picture. Claude or ChatGPT, connected to all your tools, has more context than any single vendor's agent.

Wire it through MCP

Feed your LLM through MCP servers. Your data quality tool's profiling and test results. Your governance tool. Data lineage. Process lineage from your observability tool. Orchestration steps and the source code behind them. Your ticketing system. And a document, or a set of documents, that describes your business the way you'd explain it to a new hire.

Be honest with yourself about which of those have MCP servers today. Some do. Some have an API you'll wrap in an afternoon. Some you'll have to build. The point is that each one is a bounded piece of work, and once it's done, every future question benefits.

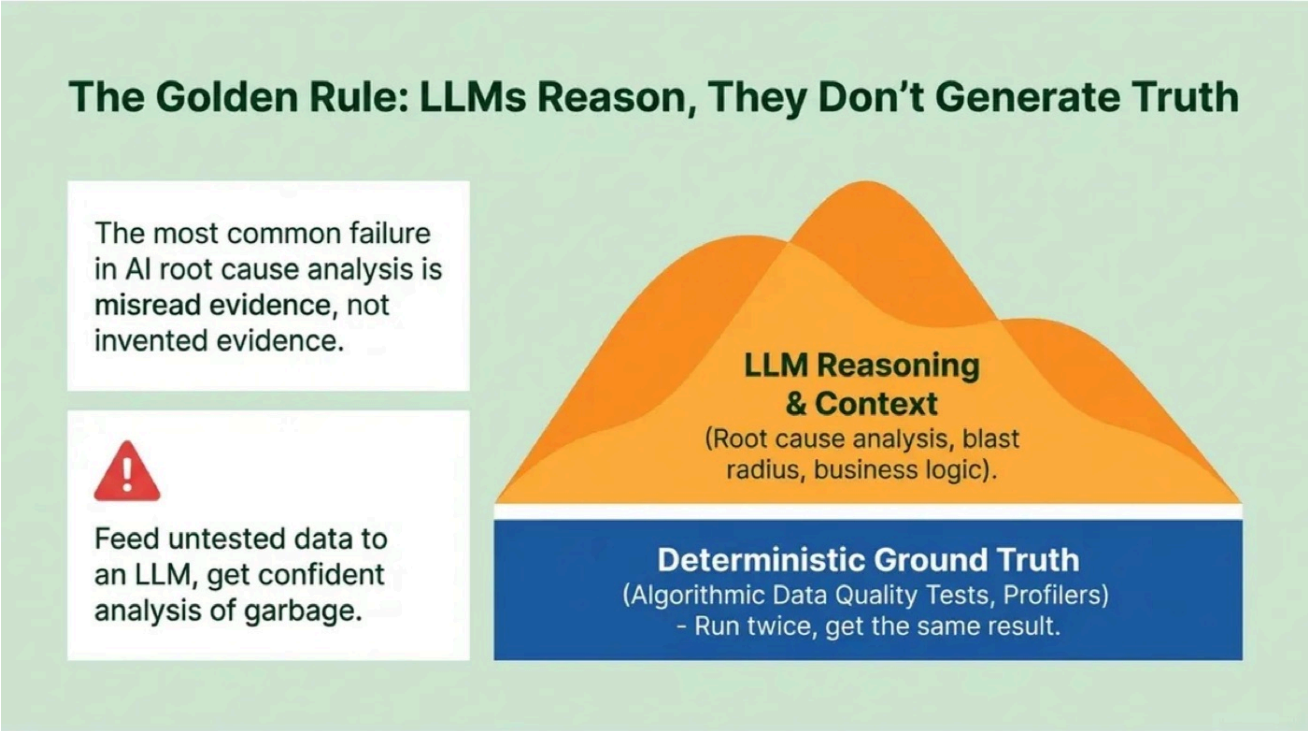


Now when the `claims_daily` test fails, you ask one question. The LLM pulls the failed run, reads the task that wrote the table, checks the diff in git from last night's deploy, walks the lineage to the reimbursement report, finds the March ticket, and tells you: this is the same encoding problem, here's who owns the vendor feed, and the report goes out in two hours. Upstream cause and downstream blast radius, prioritized against what your business actually cares about.

Keep deterministic tests underneath

The obvious objection: LLMs hallucinate, and you want one at the center of your incident response? Yes, and the objection is why the architecture works the way it does.

The most common failure in LLM-based root cause analysis isn't invented evidence. It's misread evidence: the model gets real logs and real metrics and draws the wrong conclusion from them. Picking a better model narrows that gap but doesn't close it. What closes it is giving the model ground truth it didn't produce. Your data quality tests should be algorithmic and deterministic. Run them twice on the same data, and get the same result. The LLM reasons over those results. It doesn't generate them. We made the same argument from the other direction last year: an LLM fed untested data produces confident analysis of garbage.

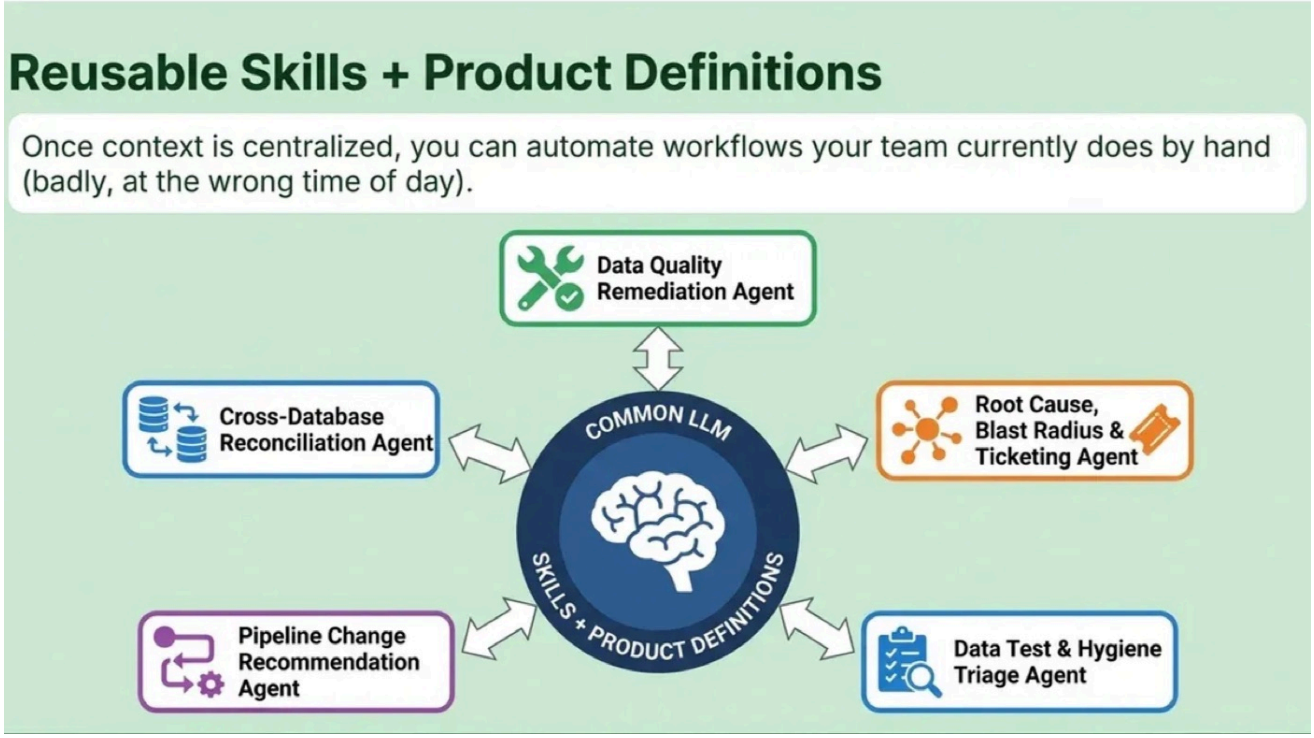


The same split applies to writing tests. A profiler can see that `discount_pct` ranges from 0 to 100. It can't know that anything above 40 needs a VP signature. Your LLM can, because you told it. Point it at the profile, describe what correct means for your data in plain English, and let it write the test definitions. Then check them against the raw table. The LLM contributes the business context. The test still runs deterministically.

Automate the repeatable parts with agents

Once the LLM has the context, most of these workflows repeat. Agents and skills can run them. That saves time, and it encodes company knowledge that currently lives in one engineer's head.

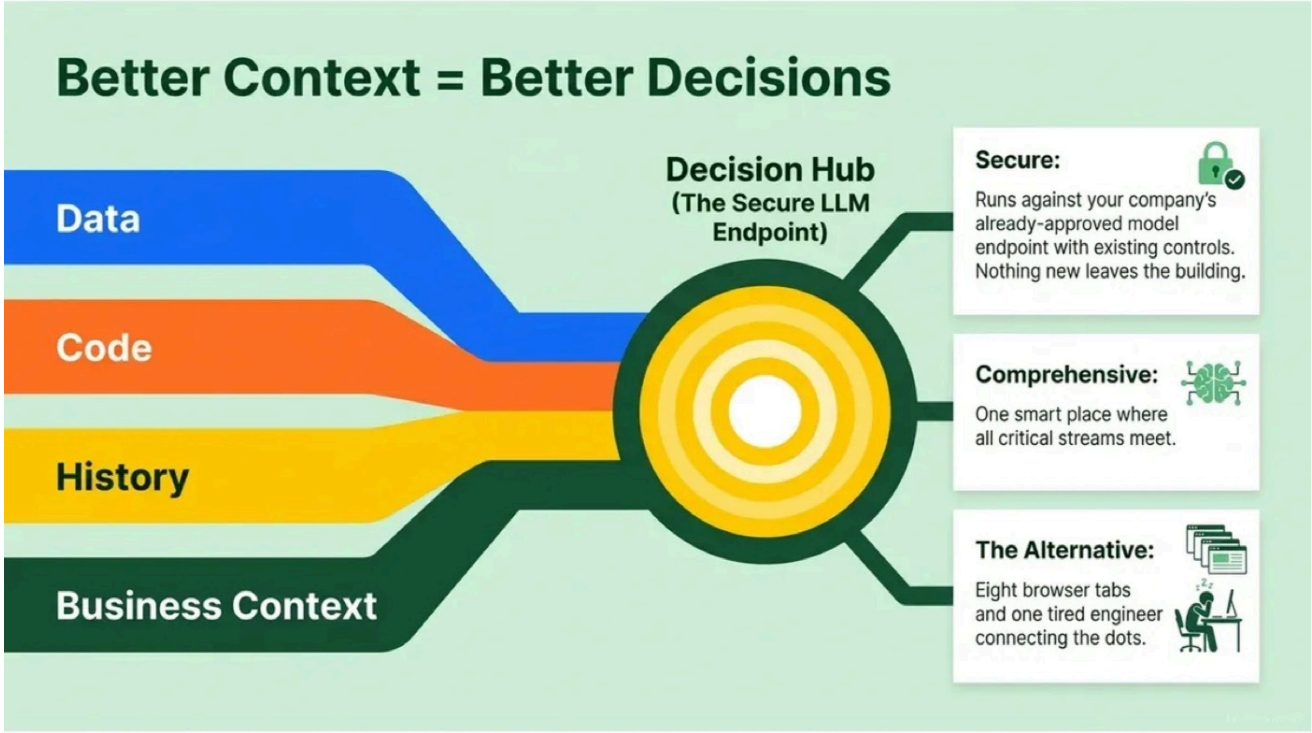
The market hasn't settled on an agent architecture. Most companies build agents their own way, and that's fine. That matches the data architecture you already have, which was also built department by department. The cost of building an agent has never been lower. With Claude Code, you can stand up a working one in an afternoon.



Five to start with: data quality remediation, cross-database reconciliation, root cause and blast radius analysis that opens the ticket for you, pipeline change recommendation, and test and hygiene triage. Each one is a workflow your team already does by hand, badly, at the wrong time of day.

Why the center matters

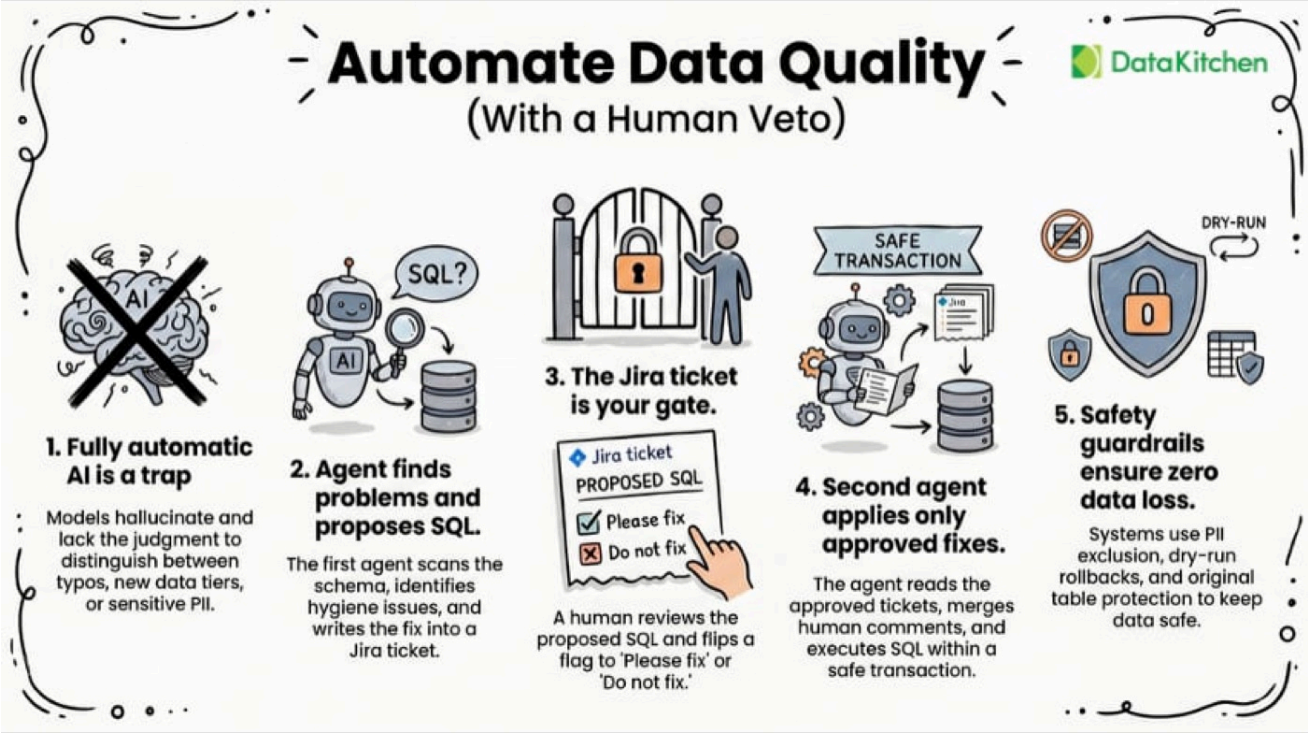
More information in the decision means a better decision. Your security team will ask what leaves the building, and the answer is nothing new: you run this against the model endpoint your company already approved, with the same controls. You want one smart place where the data, code, history, and business context meet. The alternative is what you have now: eight tabs and one tired engineer connecting the dots.


Better Context = Better Decisions
Data
Code
History
Business Context
Decision Hub
(The Secure LLM Endpoint)
Secure:
Runs against your company's already-approved model endpoint with existing controls. Nothing new leaves the building.
Comprehensive:
One smart place where all critical streams meet.
The Alternative:
Eight browser tabs and one tired engineer connecting the dots.

AI AND DATA QUALITY

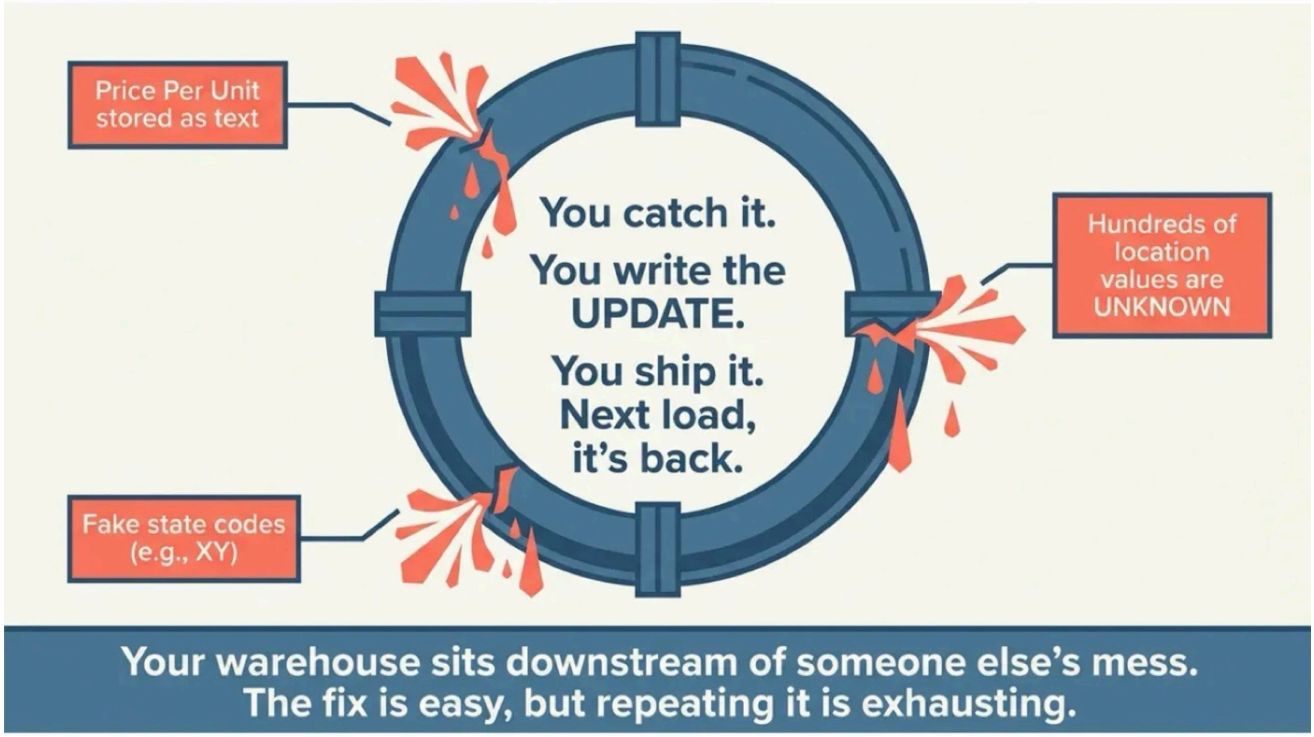
Agents to Fix Data Quality Automatically With Control: A Design Pattern

Read the original article at: https://datakitchen.io/blog/agents-fix-data-quality-automatically-with-control/



A data quality agent that fixes bad source data and asks first

Your warehouse sits downstream of someone else's mess. A partner sends a file in which "Price Per Unit" is stored as text, hundreds of "location" values are UNKNOWN, and three rows claim that a customer lives in state XY. You catch it. You write the same UPDATE you wrote last month. You ship it. Next load, it's back.



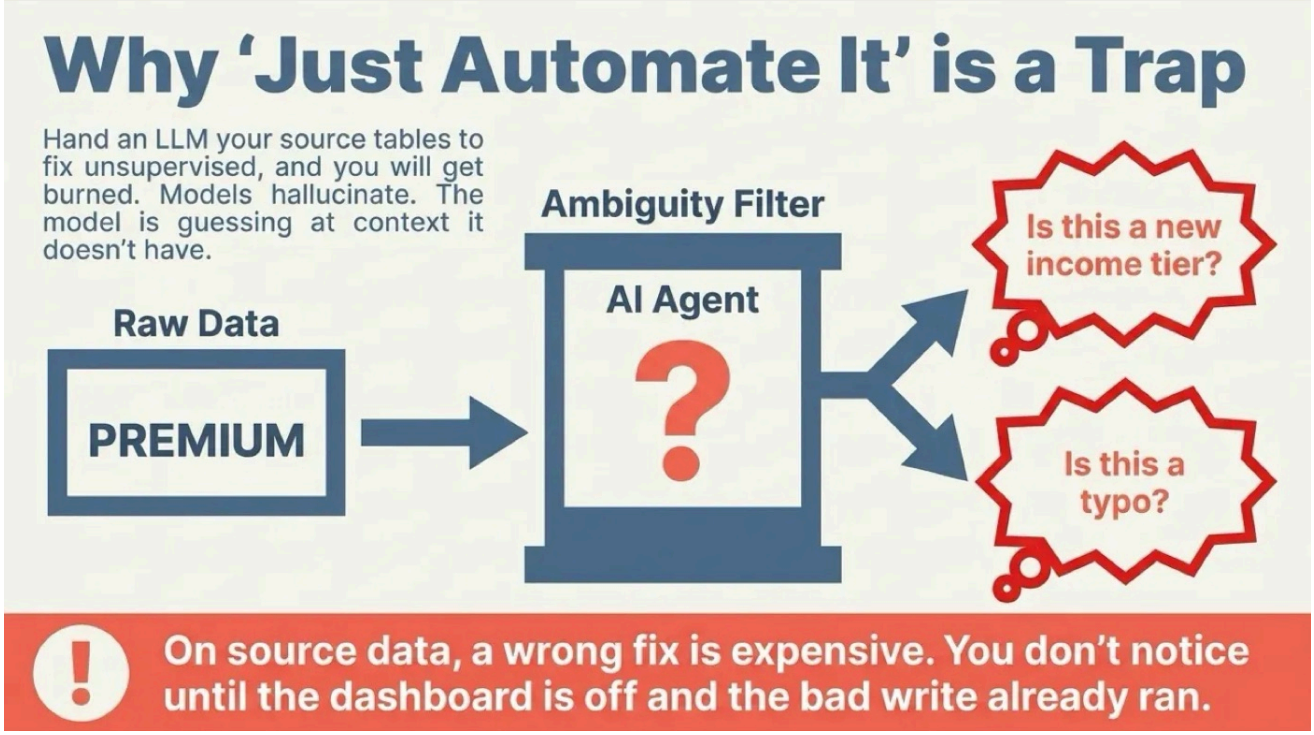
The fix is easy. The control is the hard part. You don't want an agent rewriting your source tables on its own at 2 am. You want it to find the problem, propose the exact SQL, and wait for a human to say "yes, fix that one" before anything runs.

So I built that. Two small Python programs: TestGen and Claude. One finds problems and proposes fixes. The other applies them, but only the ones you approved.

Why "just automate it" is a trap

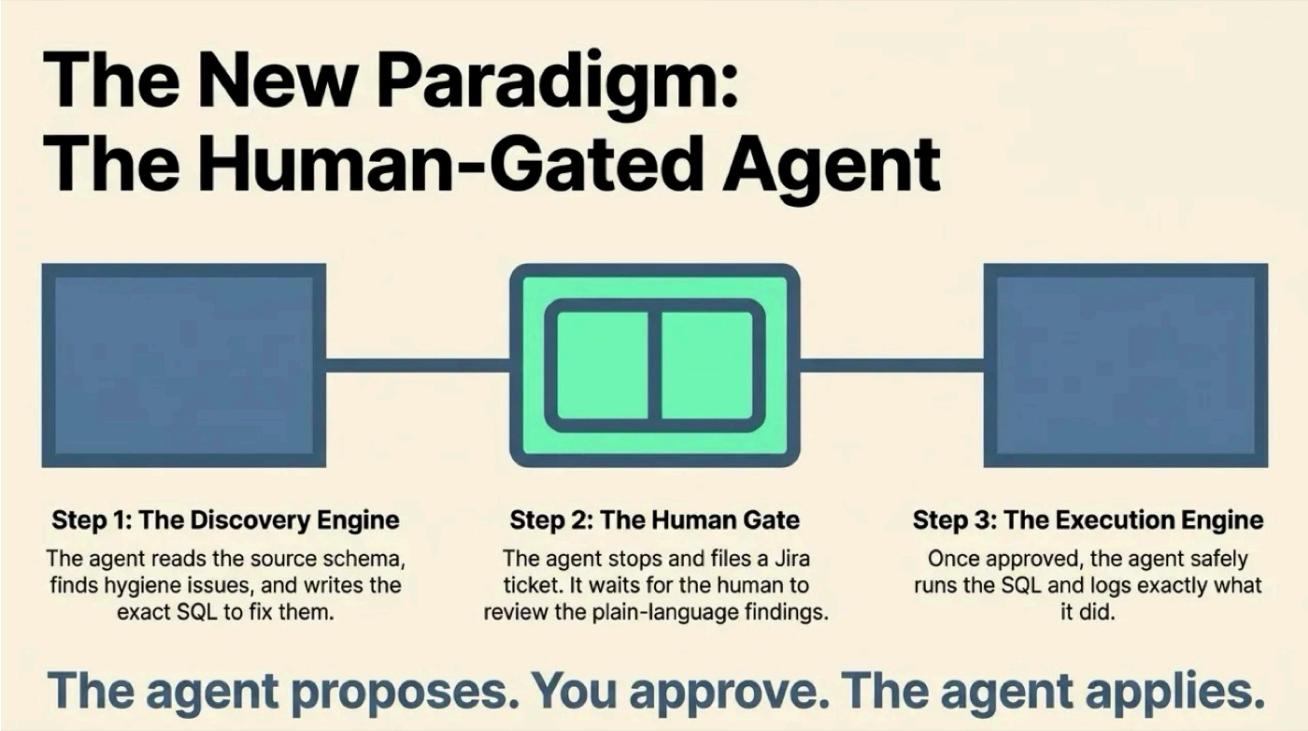
Bad source data is a judgment call. Is "PREMIUM" a new income tier or a typo? Should those 222 fake state codes become NULL, or map to real states? Is that column PII you should never touch? Hand an LLM your source tables and let it decide all of that unsupervised, and it will eventually null a field it shouldn't or reference a column that doesn't exist. On source data, a wrong fix is expensive. You don't catch it until the dashboard number is off, and by then the bad write has already run.

The SQL was never the hard part. The judgment is. You know whether "PREMIUM" is a typo. The model is guessing.



So you don't save time by taking yourself out of the loop. You save it by handing off the boring work and keeping the one decision that matters. The agent does the boring 90 percent: scans the data and writes the SQL. You make the call. Then the agent handles the other boring part: running the approved SQL and recording what it did.

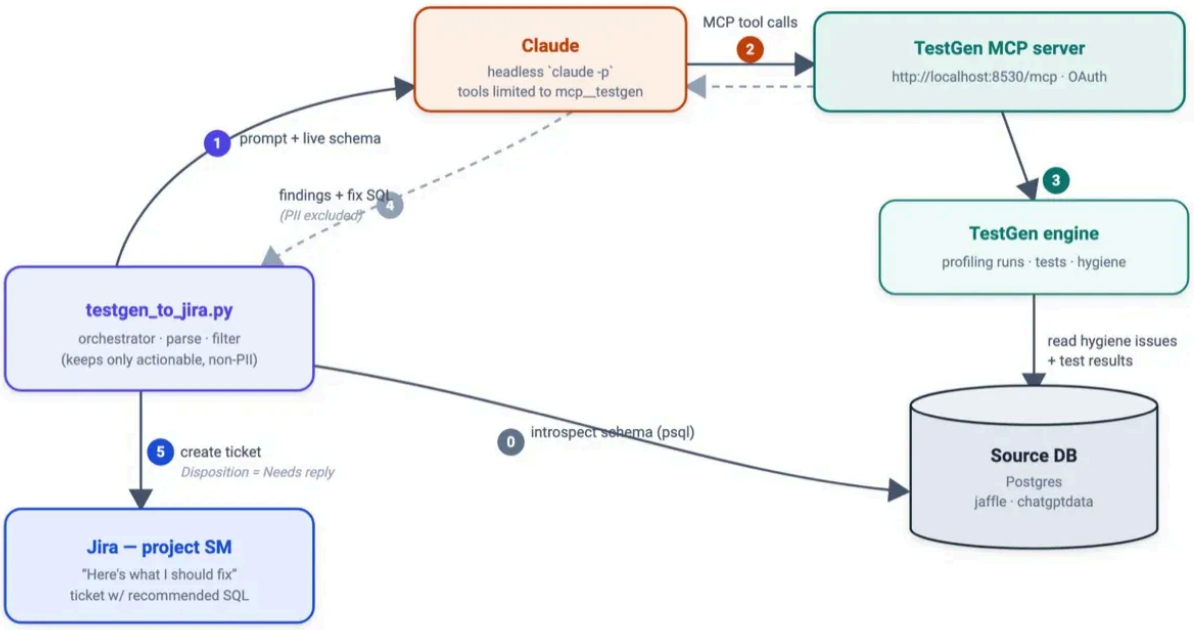
Put Jira in the middle. The agent writes its proposed fixes into a ticket and stops. You review each one and set a flag, with a comment if you want. Only then does the agent apply, and only what you approved. The ticket is the control plane. The agent proposes, you disposition, and the program logs what ran.



Program one: find the problems, file a ticket

The first program is testgen_to_jira.py. It asks TestGen what's wrong and turns the answer into a Jira ticket full of recommended fixes.

How testgen_to_jira.py discovers problems and files a ticket:



It runs in four moves. First, it reads your source database schema directly, so it knows a column is text before suggesting you retype it. Then it hands Claude a tight prompt and access to the TestGen MCP server. Claude calls TestGen to pull every hygiene issue and failing test, looks up what each test actually checks, and inspects the real failing rows. Claude sends back a list of findings. Each one has a plain-language description and the SQL to fix it.

The program then filters out anything that contains PII and keeps only the fixes it can apply to your tables. It opens a Jira ticket titled “Here’s what I should fix” and sets a disposition flag to Needs reply.

Nothing changed in your database. You have a ticket with concrete SQL and a flag waiting on you.

The human gate

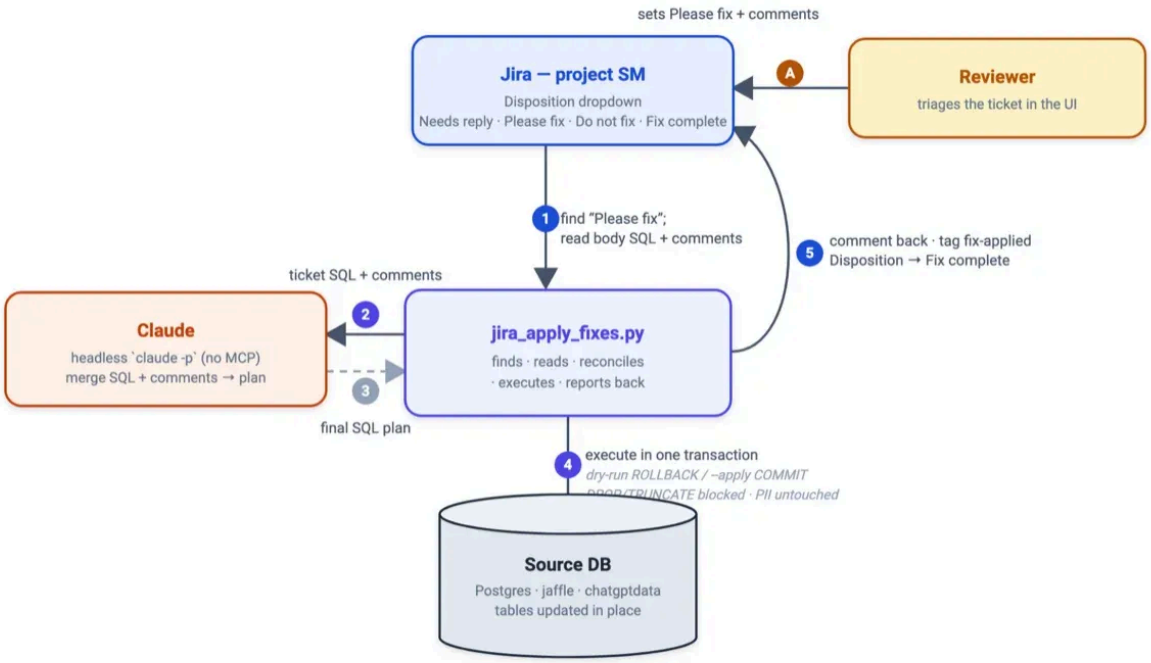
You open the ticket. You read the fixes. You leave a comment if you want one: “only null the placeholder tokens for now, hold the type changes for next release.” You flip the disposition flag to Please fix. Or do not fix. Your call, in the Jira UI, in ten seconds.

That flag is the whole point. The agent never crosses it on its own.

Program two: apply what was approved

The second program is jira_apply_fixes.py. It picks up where you left off.

How jira_apply_fixes.py reads approved tickets and applies the SQL:



It searches Jira for tickets flagged as “Please fix”. It reads the recommended SQL from the ticket body and every comment you left. It hands both to Claude with one job: merge them into a final, ordered set of SQL statements that honors your comments. If your comment said hold the type changes, Claude drops those statements. If two fixes overlap, they collapse into one.

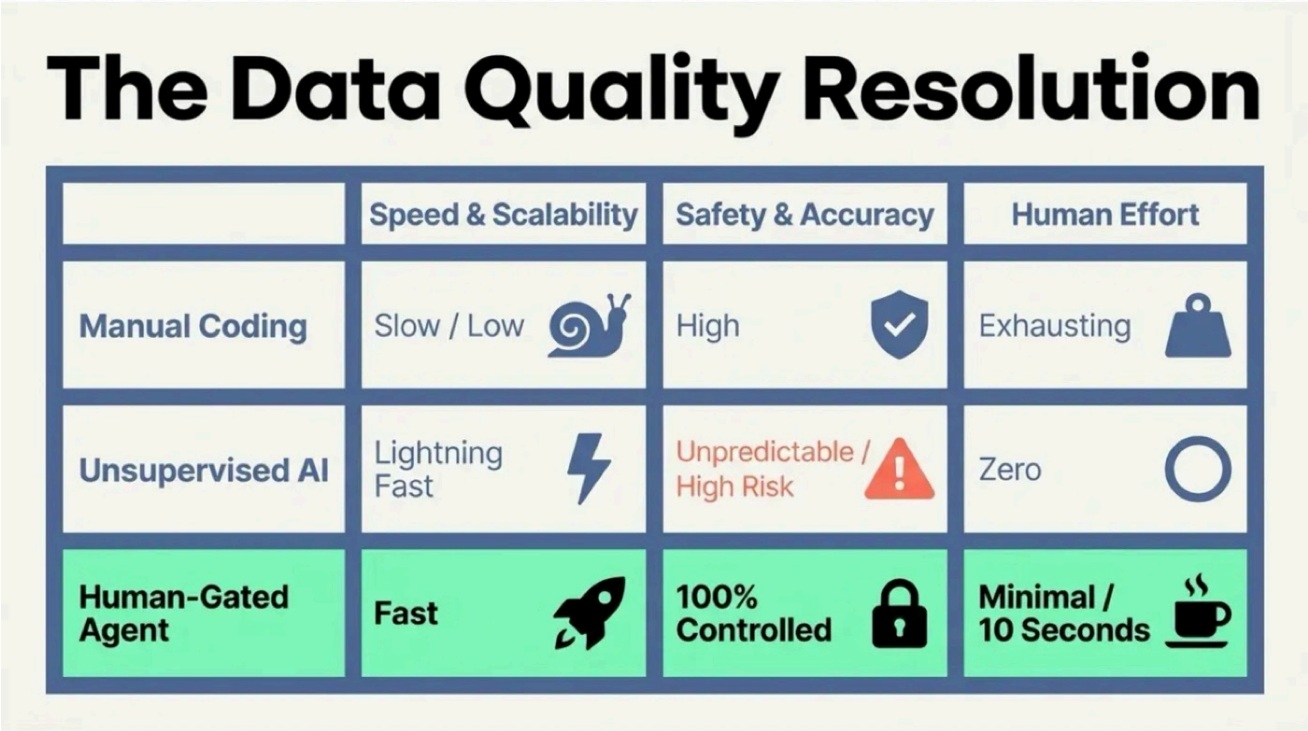
Then it runs the plan with the guardrails that make this safe to leave running. Everything executes in a single transaction, so any failure rolls back the whole thing. The program blocks DROP and TRUNCATE unless you explicitly allow them. By default, it never writes to your real tables. It rebuilds a fresh copy, table_applied, from the original and applies the fixes there, so the source stays exactly as it was. A dry run executes the SQL and rolls it back, so you can watch it work without committing anything.

When it commits, it comments back on the ticket with what ran, tags the ticket fix-applied, and flips the disposition to Fix complete. The loop closes itself.

Why this works

You get the speed of automation, and you keep the veto. The agent reads your schema, writes the SQL, runs it, and reports back. You decide which fixes happen, and you see exactly what ran. Bad data from the same source stops being a recurring chore and becomes a ticket you approve or reject over coffee.

The control lives in four places: the disposition flag a human sets, the PII exclusion the agent enforces, the copy tables that protect your originals, and the transaction that makes every apply all-or-nothing.



TestGen’s MCP server makes the first half possible. Claude reads your data quality results through it directly, with no glue code and no scraping a UI. If you want to build something like this, start here:

- TestGen MCP server documentation: https://docs.datakitchen.io/testgen/mcp/
- TestGen API reference: https://docs.datakitchen.io/testgen/api/

Your sources will keep sending bad data. Now you fix it once, on your terms, and the agent handles the rest.

Build it yourself

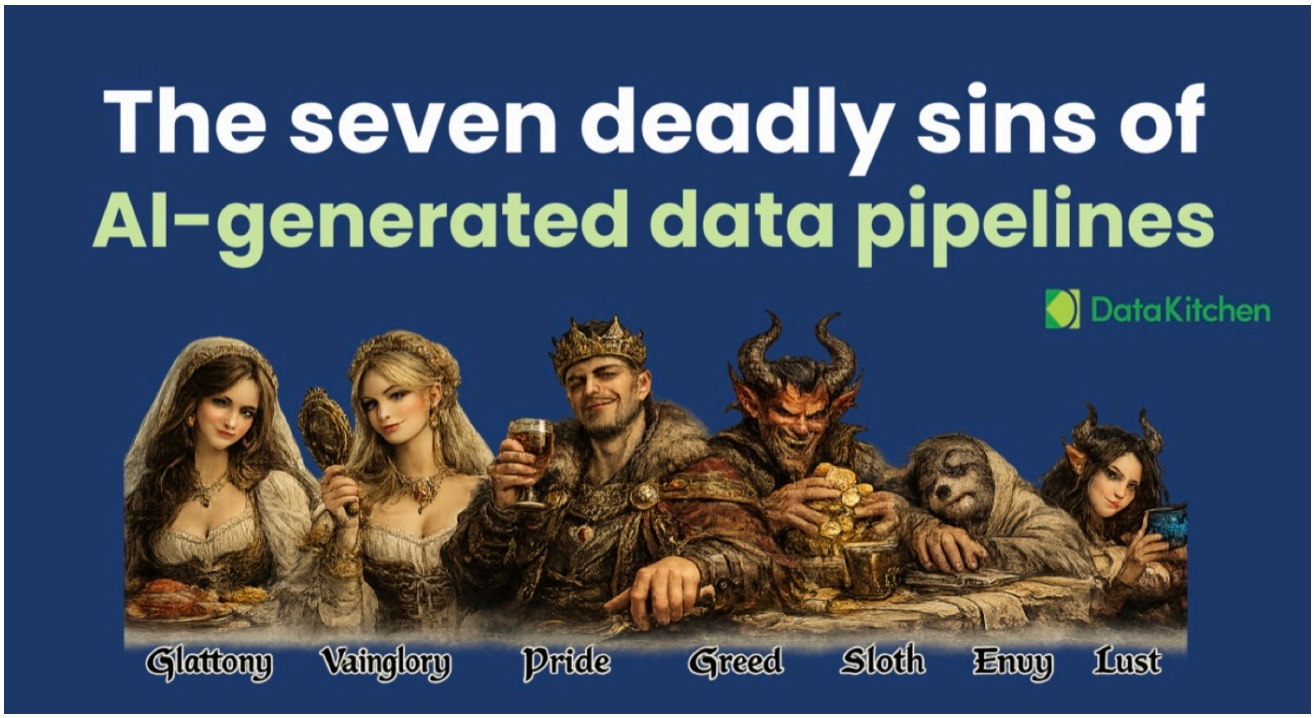
If you want to recreate this in your own environment, I wrote a PRD you can load straight into Claude Code. It names the two programs, the contract between them, the safety gates, and the config, enough for Claude to rebuild the pattern against your stack.

Download the PRD and load it into Claude Code to build the human-gated data quality fix agent in your own environment.

AI AND DATA QUALITY

The Seven Deadly Sins of AI-Generated Data Pipelines

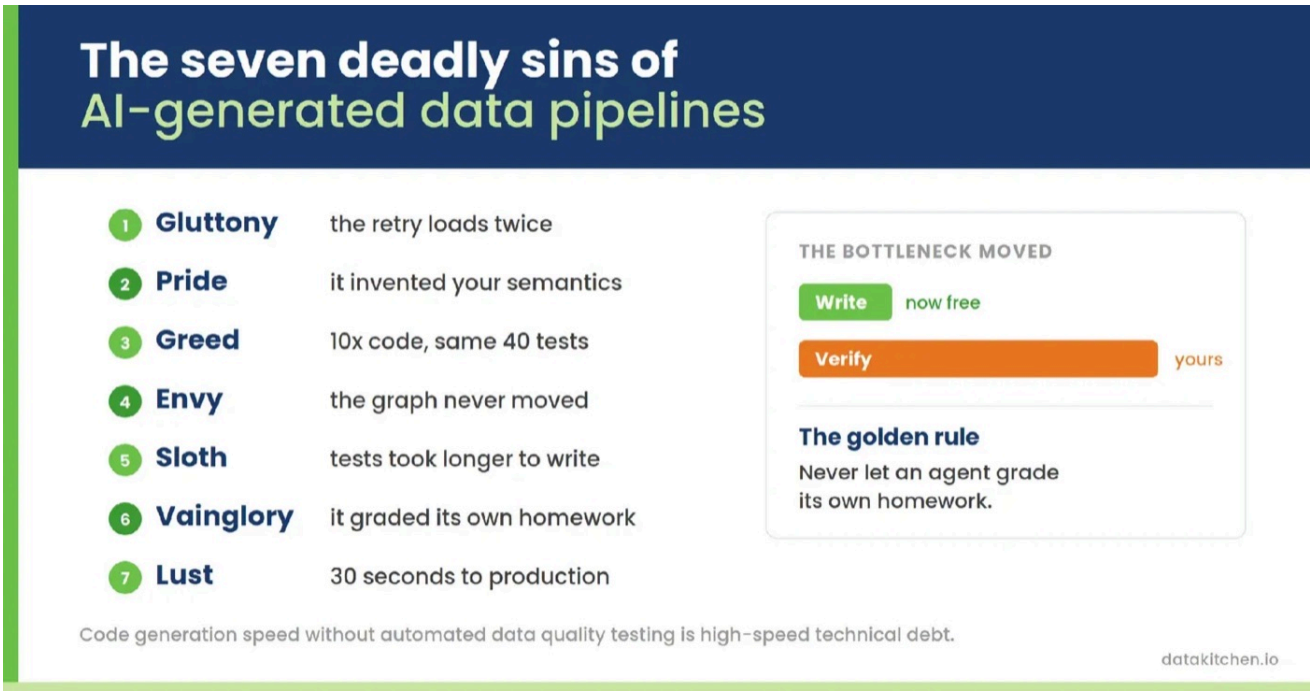
Read the original article at: https://datakitchen.io/blog/seven-deadly-sins-of-ai-generated-data-pipelines/



You shipped four dbt models before lunch. Three of them came out of Claude Code. You read them back, checked the grain, squinted at a join, and merged.

Now count the hours going the other direction. Reading. Checking. Working out whether the load ran twice. Producing code got cheap. Verifying it did not.

The output that hurts you was never the code that threw an exception. It’s the code that ran clean and quietly moved a number.

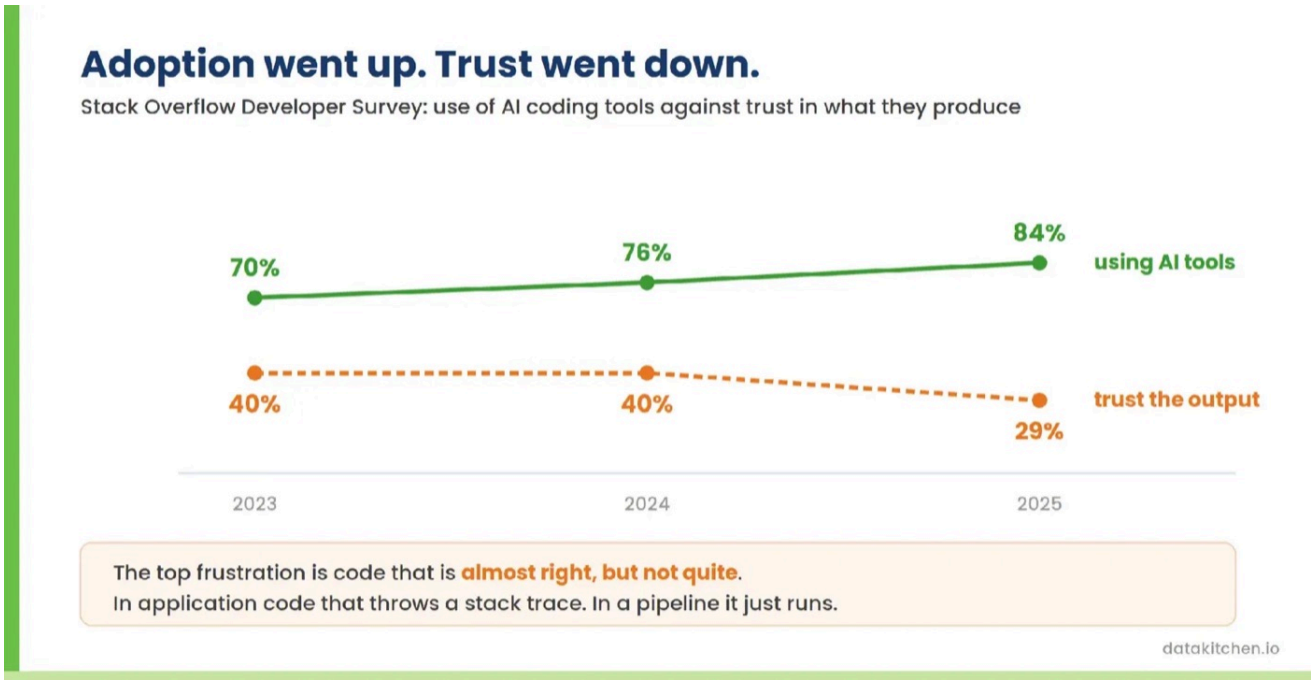


Seven failure modes, one rule holding them together.

Everybody is using it. Almost nobody trusts it

The 2025 Stack Overflow Developer Survey put 84% of developers on AI tools. Half of professional developers use them daily.

Trust went the other way. Only 29% of respondents say they trust AI output to be accurate, down from 40% the year before. Stack Overflow calls this the AI trust gap, and the number-one frustration behind it is code that is almost right but not quite.



Adoption rose every year. Trust did the opposite.

Almost right is the whole problem in data. Almost right in application code throws a stack trace. Almost right in a pipeline produces a revenue number that is 4% high because the join fanned out on a customer table with duplicate keys. Nothing fails. The dashboard renders. Someone makes a decision.

You already know what you use LLMs for. Window functions you don't want to write from memory. Legacy stored procedures nobody has touched since 2019. Airflow and Dagster DAG scaffolding. Parsing nested API JSON into something DuckDB will accept. Turning messy OCR text into a Pydantic schema. It works, and it's fast, and that's exactly why the failure modes are worth naming.

An engineer on r/dataengineering put it plainly: producing code is now "a very cheap commodity," and the expensive part is review and testing. GitClear's code quality research found AI power users hit up to 9x higher code churn, meaning code committed and then reverted or rewritten within two weeks, plus an 8x jump in duplicated blocks. When the pipeline breaks at 3 AM, nobody on call wrote the logic. That failure lands in the same place it always did, inside the four processes every data team runs, except now the code arrived faster than anyone could read it.

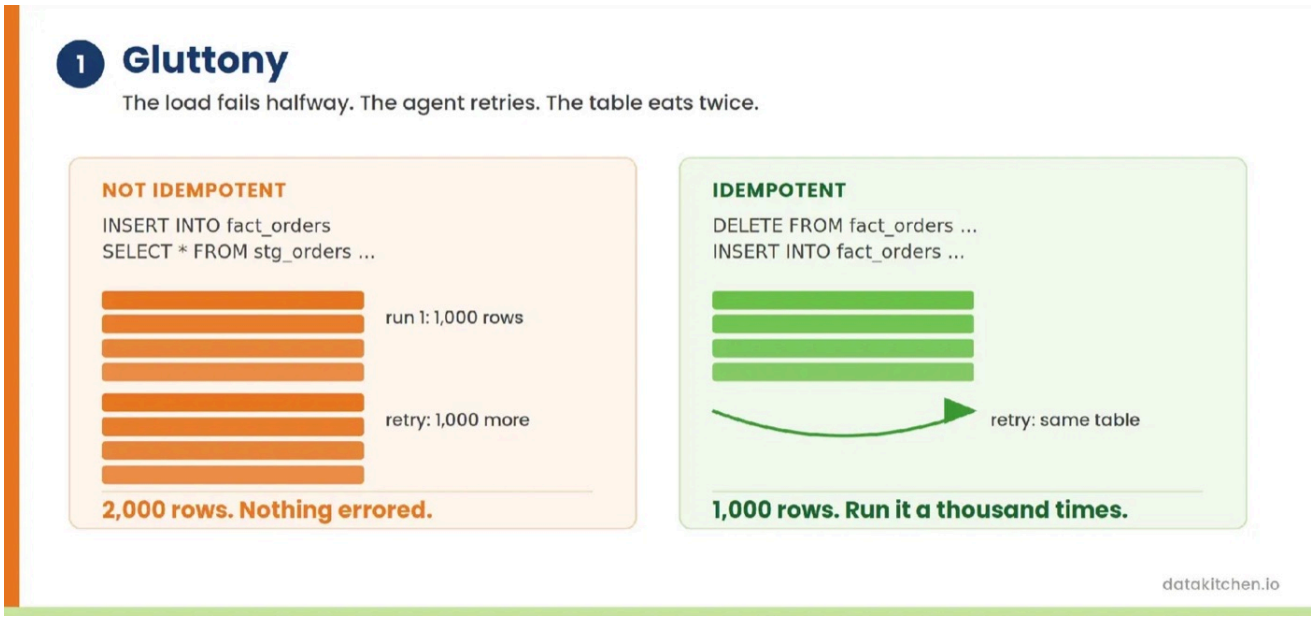
Seven ways AI-generated pipelines break. Each one has a mechanism and a fix.

These are the sins of the code. For the sins of the culture around it, read the companion post: The Seven Deadly Sins of Data Quality

Sins of construction

1. Gluttony, or the pipeline that eats twice

An agent writes an append-heavy load script. The API times out mid-run. The agent retries, because retrying is what agents do, and the target table gets the same rows again.



Four extra words of SQL is the whole difference.

This is what the model hands you, and it looks fine:

```
INSERT INTO fact_orders
SELECT * FROM stg_orders
WHERE order_date >= CURRENT_DATE - 1;
```

Run it twice and you have every row twice. Gluttony is consuming the same thing repeatedly, which is also the textbook definition of non-idempotent code. What you want is a load that produces the same table no matter how many times it runs:

```
DELETE FROM fact_orders
WHERE order_date >= CURRENT_DATE - 1;

INSERT INTO fact_orders
SELECT * FROM stg_orders
WHERE order_date >= CURRENT_DATE - 1;
```

Idempotency was always good practice. Agents make it mandatory. A human who watches a load fail halfway stops, looks at the target table, and decides what to do. An agent sees a non-zero exit code and runs the command again. Then again. Nobody is watching the row count, and the retry is the whole point of handing the job to an agent in the first place.

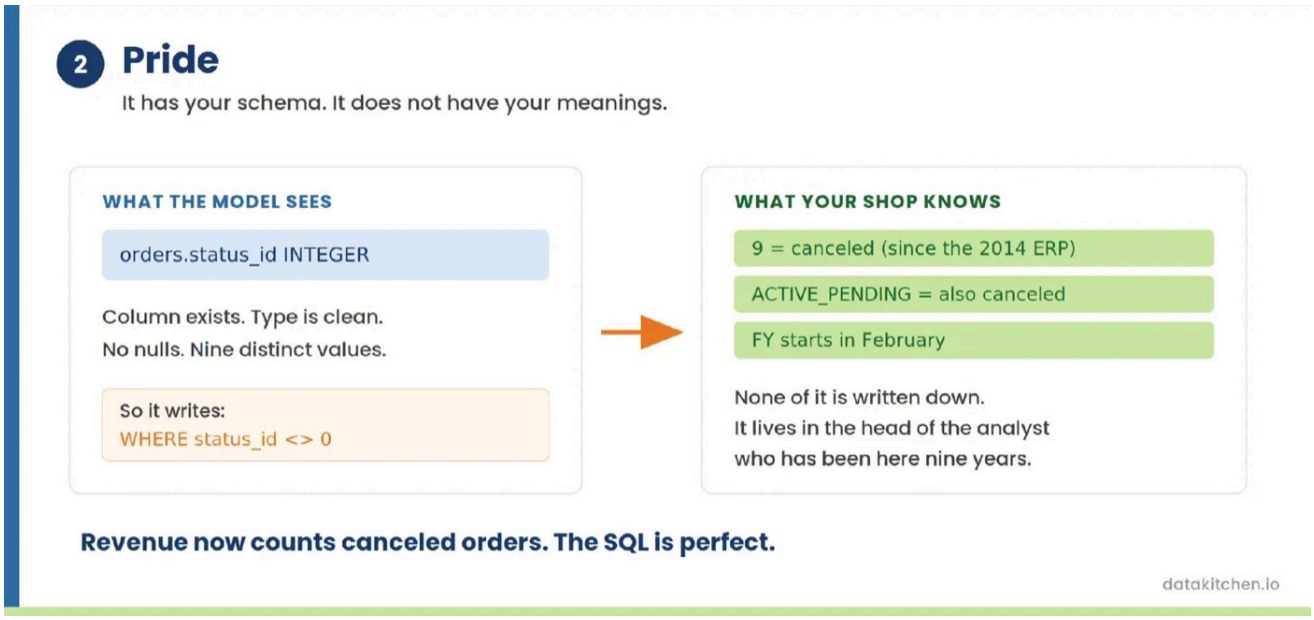
The test for whether you have it is one question. Can you run this job a thousand times and get the same table? If the answer is no, or if the answer is “yes as long as it doesn’t fail partway,” you don’t have it.

The pattern that gets you there is FITT: functional, idempotent, tested, two-stage. The I is the letter that matters here. Raw data stays immutable, transformations behave like pure functions, and any job can be rerun without reconstructing what happened last time. Recovery stops being an investigation and becomes a rerun. We wrote up why we won’t build pipelines any other way, and the SQL and orchestration mechanics are in FITT vs. fragile.

2. Pride, or assuming a frontier model knows your schema

The model has your table names. It does not know that status_id = 9 means canceled in the ERP your company bought in 2014. It does not know that ACTIVE_PENDING is what the migration script left behind for records canceled before the migration. It does not know your fiscal year starts in February, or that the natural key on the customer table stopped being unique after the acquisition.

An LLM with database access and no business context is a fast way to generate wrong data. It writes syntactically perfect SQL against semantics it invented.



Every signal the model can see says this column is clean.

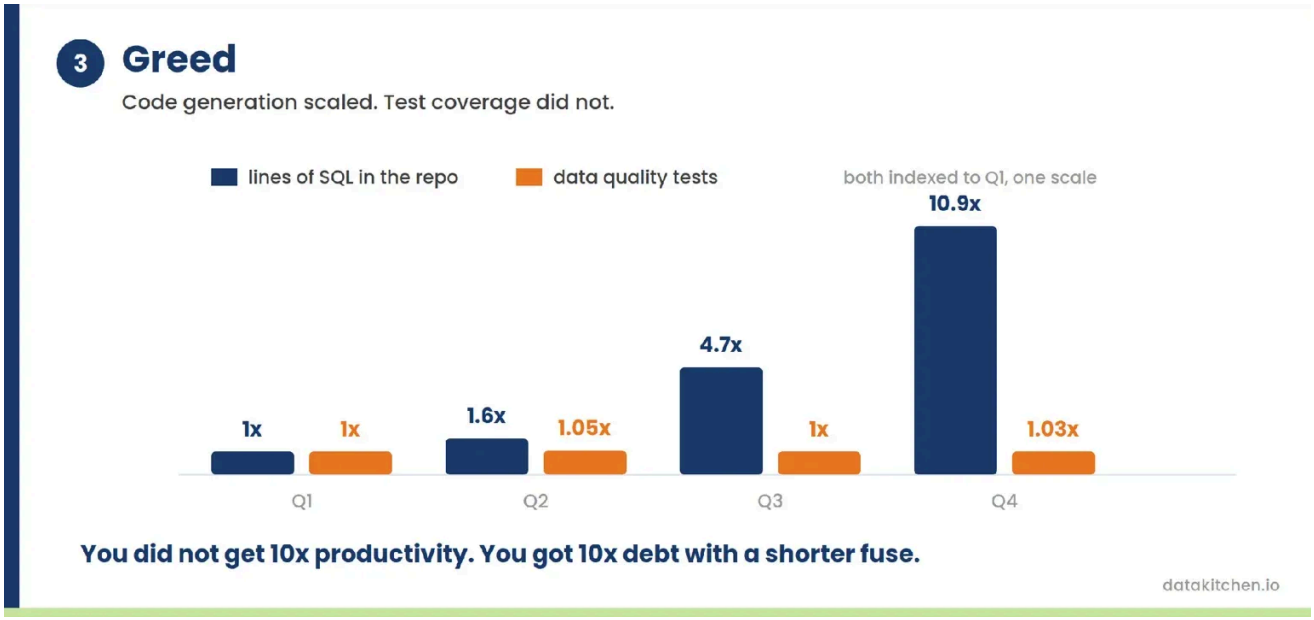
The reason this bites so hard is that the semantics were never written down. Your best data quality rules live in someone else’s head, usually the analyst who has been there nine years. A model cannot read that head. It can read column profiles, null rates, cardinality, value distributions, business definitions, and current test results, which is the argument in the equation for AI success. Feed it what correct looks like in your data and it stops guessing.

Sins of testing

3. Greed, or 4,000 new lines behind 40 old tests

You generated a quarter's worth of SQL in an afternoon. Your test suite is the same size it was Monday.

Greed is hoarding generated code without paying the coverage tax. Ship 10x the SQL against a fixed test suite and you didn't get 10x productivity. You got 10x technical debt with a shorter fuse.



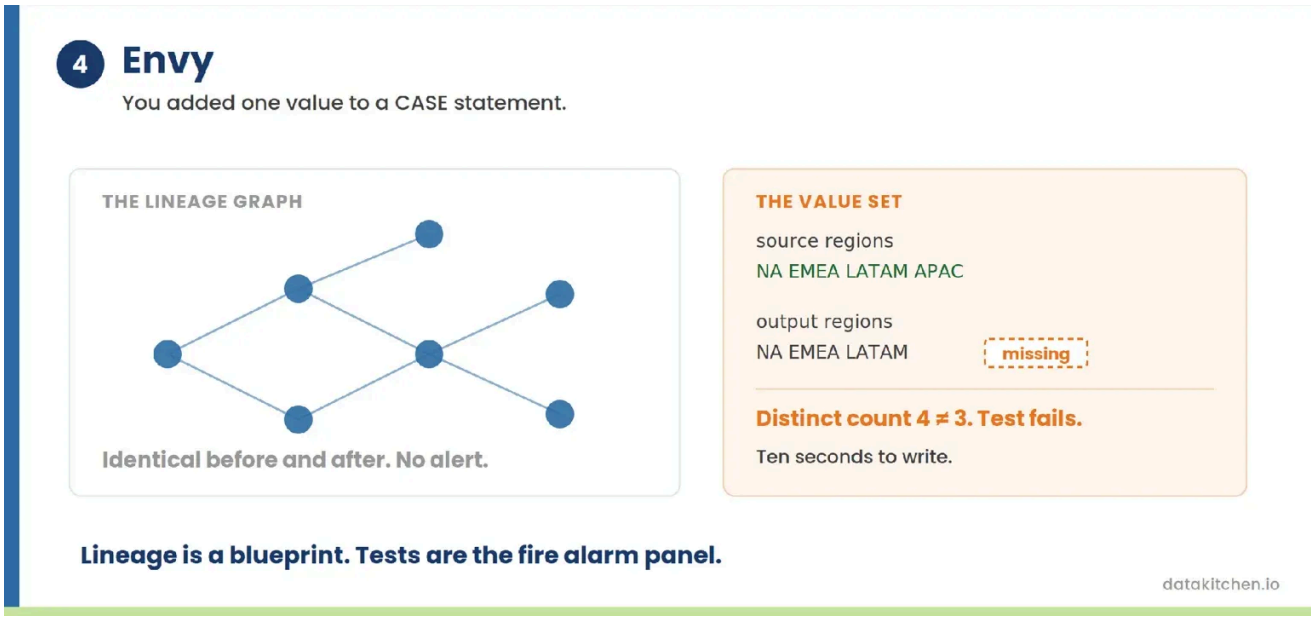
Both series indexed to Q1, on one scale.

Match generation velocity with test generation velocity. Every model an agent touches gets companion tests, written at the same time, derived from the profile of the actual data rather than from what the transform claims to do. Our definitive guide to test coverage walks through what full coverage means on a real warehouse, and why the count you need runs into the thousands rather than the dozens.

4. Envy, or you bought the graph because everyone had one

Somebody demoed a lineage graph to your steering committee. It glowed. It had 4,000 nodes. You bought it.

Then you added 'APAC' to a CASE statement. The graph did not move, because the dependencies did not change. The commission report dropped 30% and you heard about it from sales.



The graph cannot see a change that lives in the values.

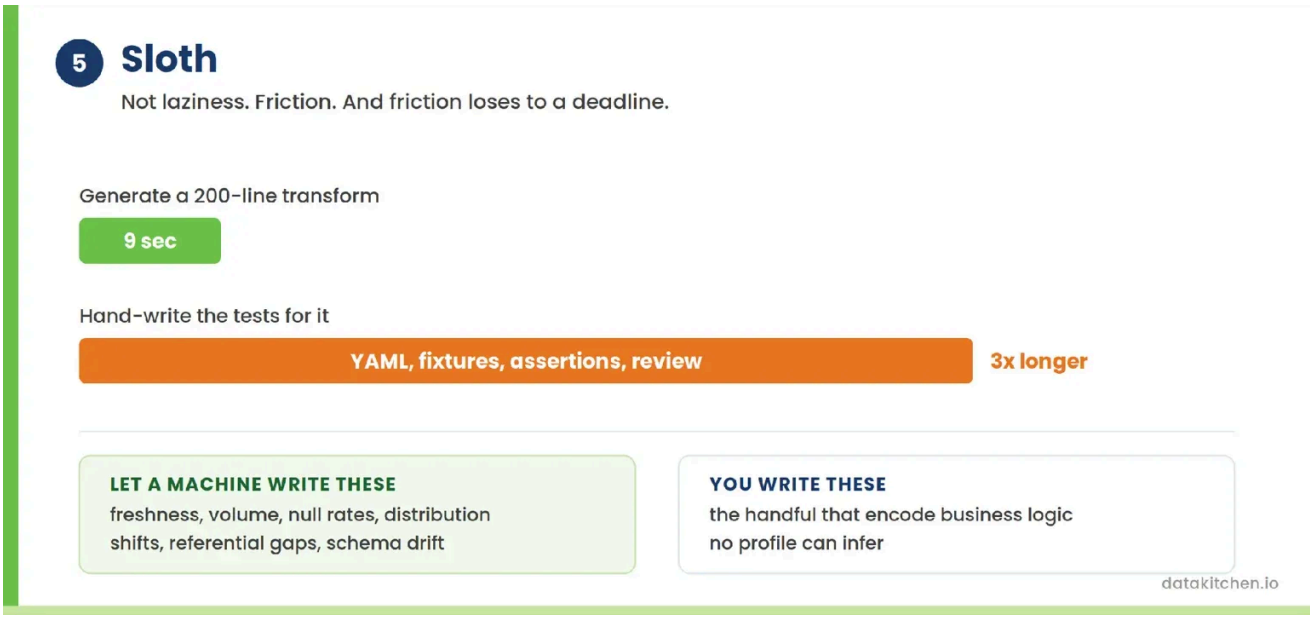
Lineage is a blueprint. Tests are the fire alarm panel. Lineage tells you where a fire could spread. Tests tell you which table is burning. Both have a job, and we've written about how lineage and data journeys complete each other. But lineage is static analysis, and no engineer ships on static analysis alone.

The fix for the CASE statement is not a bigger graph. It's a test on the value set: every region in the source appears in the output, and the distinct count of regions matches. Ten seconds to write. It catches the failure the graph cannot see, because the failure lives in the values, not the dependencies.

5. Sloth, or testing is hard so you skipped it

Writing YAML by hand. Mocking test data. Authoring assertions one at a time. Compare that to watching a model produce a 200-line transform in nine seconds.

Sloth here isn't laziness. It's friction. Manual test authoring takes three times longer than generating the pipeline did, so it loses every time there's a deadline.



Under a deadline, that contest has one outcome.

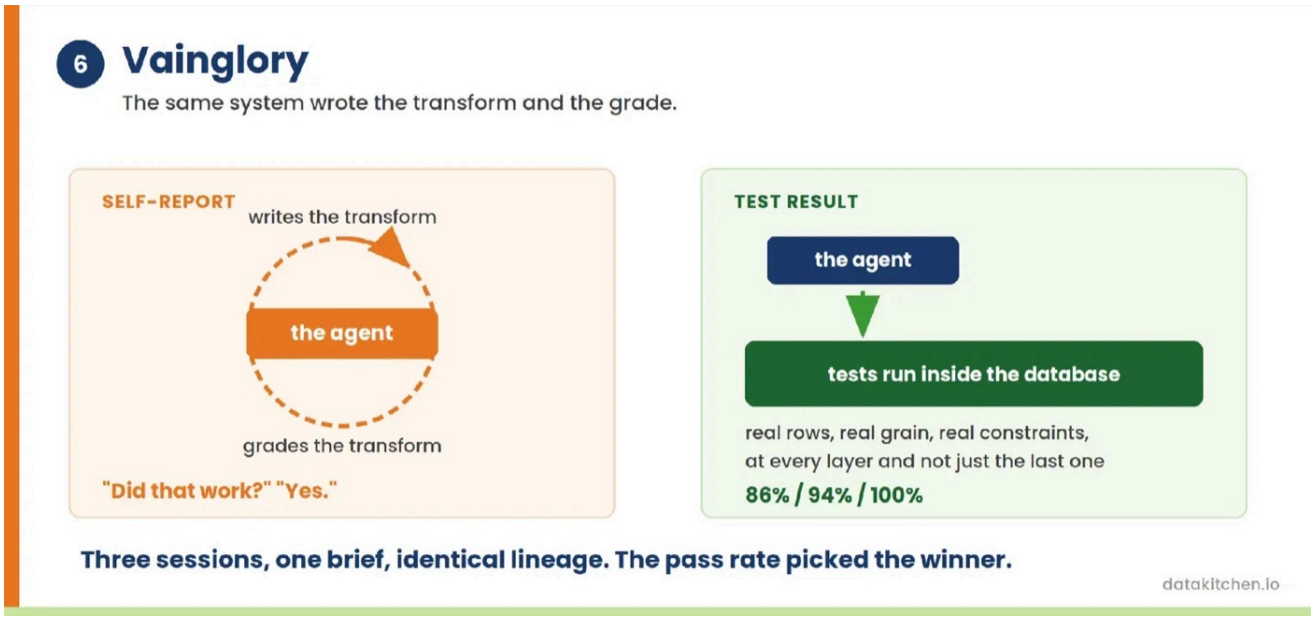
Remove the friction and sloth stops being an option. Profile the database, generate the baseline tests from what the data actually looks like, and spend your time editing the ones you disagree with instead of typing the other 200. Freshness, volume, null rates, distribution shifts, referential gaps, and schema drift are all mechanical. A machine should write them. You should be writing the eight tests that encode business logic nobody can infer from a profile. This is what shifting left and shifting down looks like in practice: catch it where it's cheap, and automate the part that's boring.

Sins of the agent loop

6. Vainglory, or letting the agent grade its own homework

You ask Cursor whether the transformation worked. It says yes. You merge.

The same system wrote the transform and the assessment of the transform. That's a self-report, not a test result. It's confirmation bias with a token budget.



A self-report is not a test result.

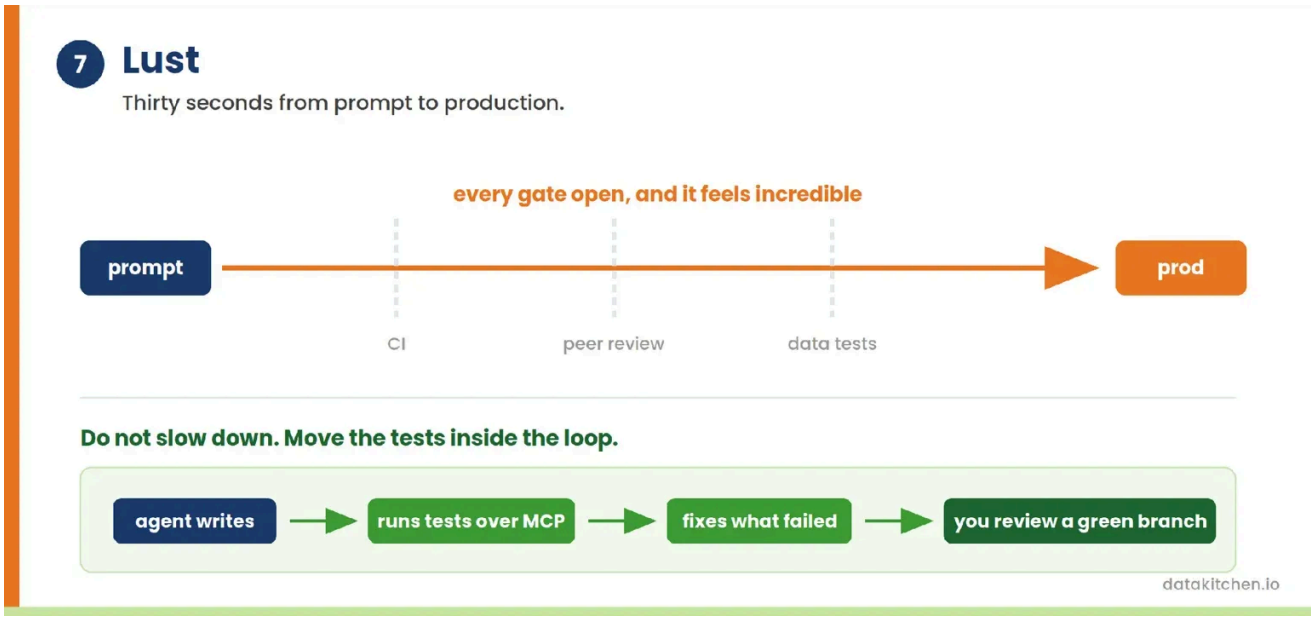
An agent cannot independently evaluate its own logic. It needs an external harness that runs against real rows, in a real database, checking real grain and real constraints. Put that harness at every layer, not just the last one. Test coverage in a medallion architecture covers where the tripwires go across bronze, silver, and gold, and why catching a problem in bronze costs a fraction of catching it in a dashboard.

Independent evaluation also settles arguments. Run one brief through three agent sessions and you get three implementations. Their lineage graphs come back identical. Their test pass rates come back 86%, 94%, and 100%. That number picks the winner and deletes the other two in about four seconds. Nothing else in your stack can do that.

7. Lust, or the lust for speed

Thirty seconds from prompt to production. Past CI, past peer review, past whoever was supposed to check. It feels incredible.

Moving fast without guardrails means you reach the outage sooner. This is the old hero pattern wearing a new hoodie. We've argued before that heroic data work is a symptom, not an achievement, and an agent that ships in 30 seconds makes the hero cycle run faster rather than ending it.



Do not slow down. Move the tests inside the loop.

The fix is not slowing down. The fix is moving the tests inside the agent's loop. An agent that can run real quality tests against real data as part of its own work finds its own failures and fixes them before it opens a pull request. You review a branch that already passed. The Model Context Protocol is what makes this practical: your testing tool becomes something the agent can call, the same way it calls the file system.

Velocity is free. Trust is earned

None of this is an argument for turning off Claude Code. AI is the biggest productivity shift data engineering has seen in a decade, and you'd be foolish to give it up.

The argument is narrower. Code generation speed without automated data quality testing is high-speed technical debt.

When generating code costs nothing, writing code stops being the bottleneck. Verification becomes the bottleneck. If a model writes 4,000 lines of transformation logic in 30 seconds, your testing infrastructure has to profile, test, and score those 4,000 lines in about 15. Otherwise you built a faster path to a production outage.

The golden rule and the four pieces

IMPORTANT

Never let an agent grade its own homework. That is the rule, and the four pieces below are how you hold yourself to it.

Four pieces make that work. Architecture first: functional, idempotent designs, so agent retries never duplicate rows or burn compute. Context second: real column profiles, business definitions, and quality metrics fed to the model so it knows what correct means in your data. Coverage third: generated tests

that grow at the same rate as your generated code, because hand-written YAML will never keep up. Loops fourth: your testing tools wired into the agent's workspace over MCP, so the agent catches its own errors before you ever see the branch.

None of the four requires buying anything. They require deciding that verification is engineering work and staffing it that way.

SECTION 7

Reference and Tooling

1. A Practical Guide to Selecting Data Quality Tools
2. The Data Quality and Observability Tools Landscape
3. The 2026 Data Quality and Data Observability Commercial Software Landscape
4. The 2026 Open Source Data Profiling Software Landscape
5. The Consultant's Growth Playbook: Accelerating Client Acquisition with DataKitchen Open Source TestGen
6. $1 Billion in Data Observability VC Investment: This Is Not Going to End Well
7. There are a lot of freaking data quality and data observability vendors
8. There Is No Microsoft Data Quality Tool in 2026. There Are Nineteen.
9. Where to Actually Start with DataOps in 2026

REFERENCE AND TOOLING

A Practical Guide to Selecting Data Quality Tools

REFERENCE AND TOOLING

A Practical Guide to Selecting Data Quality Tools

The DataOps Way to Data Quality and Data Observability

DataKitchen

The data quality tooling market has expanded significantly in the past five years. There are now more options than most teams can evaluate carefully, and the marketing language across vendors is nearly identical: "end-to-end observability," "trust your data," "catch issues before they reach production." Choosing the wrong tool wastes budget and engineering time. Choosing no tool at all leaves you blind.

This chapter is structured by use case, not by vendor. For each major use case, the options are laid out alongside their real tradeoffs. The goal is to help you match your actual situation to the right level of tooling, rather than to sell you on any particular product.

Use Case 1: Testing Data in a SQL Warehouse

If you work with Snowflake, BigQuery, or Redshift, you have several solid options for data quality testing. The right choice depends on your existing stack, your team's technical capacity, and how custom your testing requirements are.

dbt Tests (Built-in)

If you are already using dbt for transformation, the built-in test framework is the obvious starting point. dbt offers four generic tests out of the box: `unique`, `not_null`, `accepted_values`, and `relationships`. These cover a surprising percentage of the testing needs most teams have. You can

also write custom singular tests (arbitrary SQL queries that should return zero rows) and install packages like `dbt-expectations` or `dbt-utils` to extend coverage.

Tradeoffs: dbt tests are free, integrated directly into your development workflow, and require no additional infrastructure. The limitations are real though. Tests run at model build time, not continuously. You cannot test freshness on tables you did not build. The test DSL is YAML-based and can become verbose for complex business rules. For teams whose entire transformation layer lives in dbt, these limitations are often acceptable. For teams with data that lives outside dbt models, they are not.

Great Expectations

Great Expectations (GX) is a Python-based framework for defining, running, and documenting data quality tests called "expectations." It supports a wide range of built-in expectation types -- column value ranges, regex patterns, statistical distributions, cross-table relationships -- and can be run against SQL warehouses, dataframes, and files.

Tradeoffs: GX is powerful and flexible. It also has a steeper learning curve than dbt tests. The configuration model (Data Sources, Data Assets, Expectation Suites, Checkpoints) requires upfront investment to understand. The documentation has improved significantly but is still denser than most teams expect. GX makes sense for teams with complex testing requirements that go beyond what dbt's built-in tests can express, or for teams that need to test data outside of a dbt context.

Soda Core

Soda Core is an open-source framework that lets you define data quality checks in a YAML-based syntax called SodaCL (Soda Checks Language). Checks run against your warehouse using SQL under the hood. Soda also offers a commercial platform (Soda Cloud) for alerting, history, and collaboration, but the core testing framework is free.

Tradeoffs: Soda's check syntax is readable and approachable, often easier to pick up than GX for analysts and analytics engineers who are not primarily Python developers. The open-source core is capable. The commercial platform adds significant value for larger teams but is a meaningful cost. Soda is a reasonable choice for teams that want something between dbt's simplicity and GX's power, particularly if non-engineers will be writing or reviewing tests.

Custom SQL Checks

For teams with simple, stable testing requirements and the engineering capacity to maintain them, custom SQL checks -- queries that return rows when a condition is violated, run via Airflow or another orchestrator -- are a valid option. They give you complete control over logic, scheduling, and alerting.

Tradeoffs: Custom SQL checks are flexible and have no external dependencies. They are also high-maintenance. As your data environment grows, the collection of ad hoc SQL checks becomes difficult to organize, version, and document. This approach works well for small teams with a handful of critical tables. It does not scale cleanly beyond that.

Use Case 2: Testing Data in Streaming Pipelines

Streaming data introduces a different set of challenges. The data is moving, the volume is high, and the window for catching an error before it influences a downstream system is narrow. Your options here are structurally different from warehouse testing.

Schema Registries

A schema registry (Confluent Schema Registry is the most widely used) enforces a defined schema on every message produced to a Kafka topic. Producers that attempt to publish messages that violate the schema are rejected at the point of write. This is the most reliable way to prevent structural data errors from entering a streaming pipeline.

Tradeoffs: Schema registries catch structural errors: missing fields, wrong data types, fields with unexpected nullability. They do not catch business logic errors: a field with the right type but a semantically wrong value (like a currency field in the wrong currency). They also require schema-aware producers and consumers, which means changes to your messaging infrastructure. For teams on Kafka or other schema-registry-compatible systems, this is a foundational control.

In-Stream Validation with Faust or Flink

You can build validation logic directly into your stream processing application. A Flink or Faust job can inspect every record, apply business rule checks, and route invalid records to a dead-letter queue while allowing valid records to continue.

Tradeoffs: In-stream validation is fast and catches errors at the earliest possible point after ingestion. It adds latency to your pipeline -- small but nonzero. It also adds complexity: the validation logic needs to be maintained alongside the processing logic, and changes to business rules require deploying new versions of the processing job. This approach is worth the investment for high-stakes real-time pipelines. It is overkill for pipelines where a post-stream check is fast enough.

Post-Stream Checks

For pipelines where real-time error detection is not strictly required, you can run quality checks against the materialized output of the stream -- a database table or a file written by the stream processor -- using the same tools you would use for warehouse testing.

Tradeoffs: Post-stream checks are simpler to implement and maintain than in-stream validation. The cost is latency: an error that enters the stream at 2 PM may not be caught until the next hourly check runs at 3 PM. For operational pipelines where a one-hour detection window is acceptable, this is often the right tradeoff. For pipelines driving real-time decisions (live promotions, fraud detection), it is not.

Use Case 3: Monitoring Pipelines for Downtime and Freshness

Data quality includes freshness. A pipeline that runs but produces stale data is as much a problem as one that runs but produces incorrect data. Your monitoring strategy needs to cover both.

dbt Freshness Checks

dbt has a built-in source freshness feature that checks the timestamp of the most recently loaded record in a source table and fails if it exceeds a defined threshold. You can set `warn_after` and `error_after` thresholds per source.

Tradeoffs: dbt freshness checks are simple to configure and free. They only cover dbt sources -- tables that are explicitly defined in your dbt project. If you have downstream consumers that are not in dbt, or if your freshness monitoring needs to cover more than source tables, you will need something else.

Airflow SLA Miss Alerting

If you orchestrate pipelines with Airflow, SLA miss alerting lets you define expected completion times for DAGs and tasks. When a pipeline does not finish by its SLA, Airflow sends an alert.

Tradeoffs: Airflow SLA alerting is good for detecting pipeline downtime and delays. It does not tell you whether the pipeline produced correct output, only that it completed (or did not). Combine it with output shape checks for complete coverage. The configuration is in the DAG definition, which makes it straightforward for teams already using Airflow.

Monte Carlo and Metaplane

Monte Carlo and Metaplane are commercial data observability platforms that monitor your warehouse for anomalies -- unusual row count changes, distribution shifts, schema changes, freshness violations -- using automated machine learning-based detection rather than hand-written thresholds.

Tradeoffs: These platforms are expensive, typically priced at $40,000 to $150,000 per year depending on scale. They provide genuine value in cross-system observability: they can detect anomalies you would not have thought to write a test for. For teams at sufficient scale with budget to match, the auto-detection capability is a meaningful advantage. For smaller teams, the cost is hard to justify given what open-source tools can cover.

Custom Cron Checks

A scheduled script (via cron, Airflow, or Lambda) that queries a table, checks row counts or max timestamps, and fires an alert if thresholds are breached is a legitimate monitoring approach for teams with small, well-understood pipelines.

Tradeoffs: Custom cron checks work well at small scale. They require ongoing maintenance and offer no auto-detection capability. Teams with fewer than 20 pipelines and clear expectations about what normal looks like will find them sufficient.

Use Case 4: Testing Data Quality in Python

If your pipeline processes data in Python, either with pandas for smaller datasets or Spark for large-scale distributed processing, you have a separate set of testing options.

Great Expectations

GX supports pandas DataFrames natively and has Spark support. You can run the same expectation suites against DataFrames as you run against SQL tables, which is valuable if you want a consistent testing framework across your batch and Python workloads.

Tradeoffs: GX adds overhead to Python pipelines -- the framework introduces latency and memory cost when running against large DataFrames. It is comprehensive but heavy. For data that goes through a SQL warehouse anyway, running GX tests after the data lands in the warehouse is often more efficient than running them inline in the Python job.

Pandera

Pandera is a lightweight Python library for defining schemas and validating DataFrames. You define a schema using Python type annotations or a schema class, and Pandera validates the DataFrame against it. It integrates cleanly with pandas and supports pydantic-style schema definitions.

Tradeoffs: Pandera is much lighter than GX and much easier to add to an existing Python pipeline. It is focused on schema validation: column types, allowed values, value ranges, and basic statistical properties. It is not a full observability framework. For teams that need simple, inline validation of DataFrames without the overhead of GX, Pandera is an excellent choice.

Custom Assertions

For simple cases, Python assert statements or explicit exception raising on validation failures are fast to write and easy to understand.

Tradeoffs: Custom assertions are fine for small, well-understood pipelines owned by a single engineer. They become difficult to maintain at scale: there is no standardized reporting, no expectation library to draw from, and the assertions tend to be inconsistent across different parts of the codebase. Use them for prototypes and simple pipelines. Replace them with Pandera or GX when the pipeline becomes production-critical.

Use Case 5: Data Contracts and Schema Enforcement

A data contract is an explicit, versioned agreement about the structure and semantics of a dataset: field names, types, units, valid ranges, and a process for managing changes. Contracts are most valuable at the boundaries between systems -- where one team's output becomes another team's input.

dbt Contracts

Starting with dbt 1.5, dbt supports model contracts: you can declare the expected column names and data types for a model, and dbt will enforce them at build time. If a model produces columns that do not match the contract, the build fails.

Tradeoffs: dbt contracts are integrated and free. They enforce structure at the model level. They do not enforce semantics (the right type but the wrong meaning) and do not provide a change notification process for downstream consumers. For teams that use dbt and want a lightweight way to prevent

accidental schema changes from silently breaking downstream consumers, dbt contracts are a reasonable starting point.

Confluent Schema Registry

For event-driven architectures on Kafka, Confluent Schema Registry enforces Avro, JSON Schema, or Protobuf schemas on producers and consumers. It supports compatibility modes that control what kinds of changes are allowed (backward compatible, forward compatible, full compatibility).

Tradeoffs: Schema Registry is the right tool for streaming contracts. It enforces schema at the point of production, prevents breaking changes from reaching consumers without an explicit compatibility check, and provides a central registry of all schemas and their versions. It requires your producers and consumers to use schema-aware serialization, which is a nontrivial infrastructure requirement if you are not already there.

Custom Contract Libraries

Some teams build internal contract systems: a YAML or JSON definition of a table's expected structure and semantics, a validation step in the pipeline that checks the actual data against the contract, and a process for proposing and approving contract changes.

Tradeoffs: Custom contracts offer maximum flexibility and no vendor dependency. They require significant engineering investment to build and maintain. For teams with unusual requirements or strong preferences against external dependencies, this can be the right choice. For most teams, starting with dbt contracts or Schema Registry and adding custom semantics as needed is more practical.

Decision Matrix

Use this matrix as a starting point, not a final answer. Your specific stack, existing tooling, and team capabilities will all influence the right choice.

Team size	Stack	Risk tolerance	Budget	Recommended approach
1-3 engineers	dbt plus Snowflake/BigQuery/Redshift	Medium	Low	dbt built-in tests, dbt freshness, Airflow SLA alerts
1-3 engineers	Python pipelines	Medium	Low	Pandera for inline validation, custom SQL checks post-load
4-10 engineers	dbt plus warehouse	Low (compliance, finance)	Medium	dbt tests plus Soda Core for richer checks, Airflow SLA alerts
4-10 engineers	Mixed (warehouse plus streaming)	Low	Medium	dbt tests, Schema Registry for Kafka, Soda or GX for warehouse
10+ engineers	Multi-system, mixed stack	Low	High	Monte Carlo or Metaplane for observability, dbt contracts, Schema Registry
Any	Streaming with real-time requirements	Very low	Any	Schema Registry plus in-stream validation (Flink/Faust)

Where Commercial Tools Win

Commercial observability platforms like Monte Carlo and Metaplane provide genuine advantages that are difficult to replicate with open-source tooling:

Automated anomaly detection. These platforms learn what normal looks like for your data and alert you when something deviates, without requiring you to define thresholds in advance. This catches categories of errors you did not know to write tests for.

Cross-system lineage. When an error occurs, commercial platforms can show you which upstream sources contributed to the problem and which downstream tables are affected. Replicating this with open-source tools requires significant custom instrumentation.

Unified alerting and history. A single platform covering your entire data environment with a shared alert history and team collaboration features is easier to operate than a collection of different tools with separate notification channels.

The cost is real. At $60,000 or more per year for mid-market pricing, these platforms require a business case. For organizations where data errors carry significant financial or regulatory consequences, that case is often easy to make. For teams that are earlier in their data quality journey, the open-source path is a reasonable starting point that can scale further than most teams expect.

Where Open Source Wins

Open-source tools win on cost and flexibility. dbt tests, Soda Core, Great Expectations, Pandera, and schema registries cover the majority of testing use cases for most teams. They are well-maintained, widely used, and actively developed. They integrate with standard orchestration and alerting infrastructure without licensing constraints.

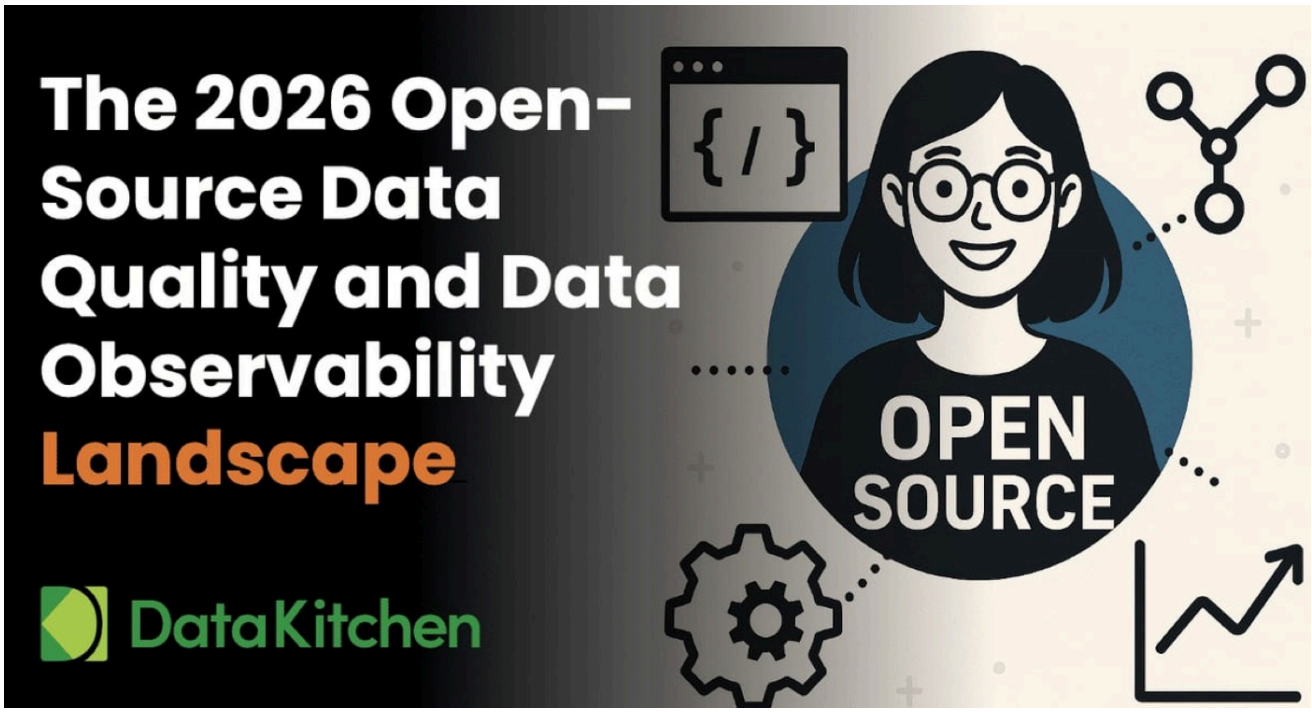
The tradeoff is that open-source requires you to define what normal looks like. Automated anomaly detection, cross-system lineage, and unified observability across an entire data environment require either commercial tooling or significant custom engineering. For teams with clear, bounded testing requirements, open-source is often the better choice. For teams with broad, complex data environments and a preference for less custom engineering, commercial platforms are worth evaluating seriously.

Start with the use case. Match the tool to the problem. Revisit the decision as your environment grows.

REFERENCE AND TOOLING

The Data Quality and Observability Tools Landscape

Read the original article at: https://datakitchen.io/blog/the-2026-open-source-data-quality-and-data-observability-landscape/



The data quality tooling market has grown quickly enough that it is genuinely hard to know what to use. There are dozens of tools, each with overlapping feature sets and marketing that sounds identical. This chapter cuts through that by organizing the landscape around what you are trying to do, not around which vendor makes what. For each use case, the options are presented honestly: what each tool is good at, where it falls short, and how to choose.

Use Case 1: Testing Data Quality in a SQL Warehouse

What you are trying to do: Write assertions against your warehouse tables that tell you when data violates expected conditions. Run those assertions in CI and on a schedule.

Open Source Options

dbt tests are the natural starting point for any team already using dbt. The built-in generic tests (unique, not_null, accepted_values, relationships) cover the majority of structural quality checks. Custom generic tests and singular tests handle the rest. The integration with dbt's compilation and execution model

means tests are first-class citizens alongside your SQL transformations. The cost is zero beyond the compute to run them. The limitation is that dbt tests run on demand, not continuously, and they have no anomaly detection.

Great Expectations is the most featureful open-source library for SQL warehouse testing. Its "Expectations" library covers hundreds of assertion types, including statistical checks (column mean within a range, value distribution matching a reference), cardinality checks, and cross-table comparisons. The tradeoff is complexity: the configuration model is verbose, the CLI has a learning curve, and the Data Docs output requires setup to be useful. Great Expectations is the right choice when you need assertion types that dbt does not support or when you are testing outside a dbt project.

Soda Core occupies a middle ground. Its YAML-based check syntax is cleaner than Great Expectations and easier to read:

```
# checks for orders table (Soda Core syntax)
checks for orders:
  - row_count > 0
  - missing_count(order_id) = 0
  - duplicate_count(order_id) = 0
  - invalid_count(status) = 0:
      valid values: [pending, processing, shipped, delivered, cancelled]
  - avg(order_amount_usd) between 15 and 500
  - freshness(created_at) < 2h
```

Soda Core connects directly to Snowflake, BigQuery, Redshift, Databricks, and others without requiring dbt. It is a good fit for teams that want lightweight, readable data quality checks outside the dbt ecosystem.

DataKitchen TestGen is an open-source SQL warehouse testing tool that focuses on automated test generation from profiled data. It generates suggested tests by analyzing the statistical distribution of existing columns, which reduces the effort of writing tests from scratch. It is worth evaluating if you are testing a large number of tables and want to bootstrap coverage quickly.

Commercial Options

Monte Carlo is a data observability platform that includes automated anomaly detection, table-level freshness monitoring, lineage, and incident management. The anomaly detection is ML-based and does not require you to manually specify thresholds. Monte Carlo monitors your warehouse continuously, not just on demand. The tradeoff is cost (it is priced per monitored asset) and the fact that you are delegating anomaly definition to a model you do not control.

Anomalo specializes in ML-based anomaly detection for warehouse data. It learns baseline distributions for your tables and alerts when values drift from expected patterns, accounting for seasonality and day-of-week effects. It is a good fit for teams with large tables where manually specifying every quality check is impractical.

Atlan is a data catalog and governance platform that includes data quality scoring, lineage, and collaboration workflows. It is less focused on raw test execution and more focused on asset discovery, business context, and quality metrics surfaced in a UI. Relevant when you need a cross-system quality view beyond the warehouse.

How to Choose

Start with dbt tests if your team is already using dbt. Add Soda Core if you need checks outside the dbt model graph (e.g., on source tables before they enter dbt). Evaluate Great Expectations if you need assertion types that dbt and Soda do not support. Move to commercial tools when: the volume of tables to monitor makes manual threshold management unsustainable, you need continuous monitoring rather than CI-only testing, or you need cross-system lineage as part of quality alerting.

Use Case 2: Pipeline Monitoring and Data Downtime Detection

What you are trying to do: Know when a pipeline is late, failing silently, or producing data that looks different from normal, without waiting for a downstream consumer to report it.

Open Source Options

Airflow SLA alerts are built into the orchestrator most teams are already running. An SLA miss alert fires when a task or DAG does not complete within a configured window:

```
# Airflow SLA example
from datetime import timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator

default_args = {
    "sla": timedelta(hours=2),  # Alert if this DAG takes more than 2 hours
}

with DAG(
    dag_id="orders_pipeline",
    default_args=default_args,
    sla_miss_callback=notify_slack_on_sla_miss,
    schedule_interval="0 * * * *",
) as dag:
    transform = PythonOperator(
        task_id="transform_orders",
        python_callable=run_dbt_orders,
        sla=timedelta(hours=1),  # Per-task SLA
    )
```

This is functional for detecting late pipelines but has no awareness of data content. A pipeline that finishes on time but produces an empty table will not trigger an SLA alert.

Prefect provides similar notification hooks with a cleaner Python API and a better UI for tracking run history. Prefect's alerting is still execution-focused: it tells you a run failed or was late, not that the data was wrong.

OpenLineage is an open standard for pipeline lineage metadata. It emits lineage events from Airflow, Spark, dbt, and other tools into a compatible backend. Combined with Marquez (the open-source reference implementation), it gives you a lineage graph you can query. It does not do anomaly detection or quality testing; it provides the lineage layer that other tools build on.

Commercial Options

Monte Carlo monitors both pipeline execution and data content. It detects freshness anomalies (a table that normally updates hourly has not updated in three hours), volume anomalies (a table that normally has 100k rows has 200 rows), and schema changes. It integrates with Airflow, dbt, and most warehouses. This is the most complete out-of-the-box pipeline monitoring solution in the market.

Metaplane (now part of a broader observability stack) offered automated anomaly detection and lineage for data pipelines. The category is competitive; evaluate current offerings at time of purchase.

DataKitchen Observability (open source) focuses on pipeline-level monitoring: tracking which pipelines ran, when they ran, whether they produced data, and surfacing journey-level health across a multi-step pipeline. It is a good fit for teams that want observability across a DataOps workflow without a commercial tool's cost.

How to Choose

Start with your orchestrator's built-in SLA alerts. They are free and already configured. Add content-level monitoring (volume, freshness, distribution) when SLA alerts alone generate too many false negatives: pipelines completing successfully but producing wrong data. For cross-pipeline incident management and automated anomaly detection, evaluate commercial options based on your stack and budget.

Use Case 3: Data Lineage

What you are trying to do: Know what upstream sources feed into a given table or dashboard, and what downstream consumers would be affected if that table changed or broke.

Open Source Options

OpenLineage is the standard. It is an open specification backed by the Linux Foundation with implementations for Airflow, Spark, dbt, Flink, and others. When you emit OpenLineage events from your pipelines, you get a consistent lineage graph across tools. It is not a UI; it is an API and a format.

Marquez is the open-source reference implementation of the OpenLineage spec. It stores lineage events in a Postgres-backed service and provides a REST API and basic UI. Running Marquez requires operating a service, but the setup is straightforward with Docker:

```
# Start Marquez with Docker Compose
git clone https://github.com/MarquezProject/marquez.git
cd marquez
docker-compose up
```

Configure the OpenLineage Airflow integration to emit events to Marquez:

```
pip install openlineage-airflow

# In airflow.cfg or environment variables:
# OPENLINEAGE_URL=http://marquez:5000
# OPENLINEAGE_NAMESPACE=my_company
```

dbt's built-in lineage (available in dbt Cloud's Explorer and locally via `dbt docs generate`) gives you model-level lineage within the dbt project. It shows which models depend on which other models and which sources feed into which models. It does not extend lineage into Airflow, Spark, or BI tools.

Commercial Options

Atlan provides cross-system lineage visualization with AI-assisted impact analysis. When you search for a table in Atlan, it shows you the lineage graph from source to dashboard, across dbt, Airflow, Fivetran, Looker, Tableau, and others. The AI-assisted impact analysis can summarize which dashboards and reports are affected by a proposed schema change. This is genuinely useful for large organizations with complex pipelines.

Monte Carlo includes lineage as part of its observability platform, with automatic column-level lineage derived from query parsing. This is more granular than table-level lineage and more useful for root cause analysis.

How to Choose

For teams operating within a single dbt project, dbt's built-in lineage is sufficient. For teams operating across Airflow, dbt, Spark, and BI tools and needing cross-system lineage, deploy OpenLineage plus Marquez as a free baseline, and evaluate Atlan or Monte Carlo when the maintenance cost of the open-source stack outweighs the benefit.

Use Case 4: Schema Enforcement and Contracts

What you are trying to do: Prevent schema-breaking changes from reaching consumers without explicit agreement. Enforce that producers publish data in the shape consumers expect.

Open Source Options

Confluent Schema Registry is the standard for Kafka-based event streams. It stores Avro, JSON Schema, and Protobuf schemas and enforces compatibility rules on producer registration. Producers that try to publish a schema change that violates the configured compatibility level (BACKWARD, FORWARD, FULL) are rejected at registration time, before any bad messages reach consumers.

Compatibility level settings:

```
# Set FULL_TRANSITIVE compatibility for maximum safety
curl -X PUT http://schema-registry:8081/config/analytics.web.page_viewed-value \
  -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  -d '{"compatibility": "FULL_TRANSITIVE"}'
```

`FULL_TRANSITIVE` means a new schema must be compatible with all previous versions, not just the most recent one. This is the strictest setting and the right choice for event topics with multiple consumers.

dbt contracts (dbt 1.5+) enforce schema contracts at the warehouse model level. When `contract: enforced: true` is set on a model, dbt validates the output schema at compile time. This catches column renames and type changes before they reach production.

Soda Core can enforce schema checks as part of a quality check run:

```
checks for transactions:
  - schema:
      fail:
        when required column missing: [transaction_id, account_id, amount_cents,
currency_code, status, created_at]
        when wrong column type:
          amount_cents: integer
          created_at: timestamp with time zone
```

Commercial Options

Atlan and **Collibra** add governance workflows on top of schema enforcement: change request processes, approval workflows, business glossary integration, and audit trails. These capabilities matter in regulated industries or large organizations where schema changes require cross-team approval. They are not relevant for most engineering teams that need basic contract enforcement.

How to Choose

Use schema registry for any Kafka-based event stream. It is the only tool that enforces contracts at the transport layer, preventing bad messages from reaching consumers at all. Use dbt contracts for warehouse models. Use schema-level Soda checks for warehouse tables outside the dbt project. Evaluate Atlan or Collibra only if you need governance workflow management on top of technical enforcement.

Use Case 5: Data Quality in Python and Spark Pipelines

What you are trying to do: Validate DataFrames in a Python or Spark pipeline before writing them to a destination. Catch type mismatches, null violations, and business rule violations in code.

Open Source Options

Pandera is the cleanest library for validating Pandas DataFrames. Its schema definition is Pythonic and readable. It supports type coercion, custom checks, and statistical assertions. It works with Pandas, Polars, and Dask. The `lazy=True` validation option collects all failures before raising, which is important in production pipelines where you need the full error picture.

```
import pandera as pa
from pandera import Column, DataFrameSchema, Check

schema = DataFrameSchema(
    {
        "order_id": Column(str, nullable=False, unique=True),
        "customer_id": Column(str, nullable=False),
        "amount_usd": Column(
            float,
            nullable=False,
            checks=[
                Check.greater_than(0),
                Check.less_than_or_equal_to(10_000),
            ],
        ),
        "status": Column(
            str,
            checks=Check.isin(["pending", "shipped", "delivered", "cancelled"]),
        ),
        "created_at": Column("datetime64[ns]", nullable=False),
    },
    coerce=False,
)

# Validate and get detailed error report if it fails
validated_df = schema.validate(df, lazy=True)
```

Deequ is Amazon's open-source data quality library for Apache Spark. It is built for large-scale validation: it runs constraint checks as Spark jobs, profiles data distributions, and can persist quality metrics to a time series store for trending.

```
from pydeequ.checks import Check, CheckLevel
from pydeequ.verification import VerificationSuite, VerificationResult

check = Check(spark, CheckLevel.Error, "Orders Quality Check") \
    .isComplete("order_id") \
    .isUnique("order_id") \
    .isComplete("customer_id") \
    .isNonNegative("amount_usd") \
    .isContainedIn("status", ["pending", "shipped", "delivered", "cancelled"])

result = VerificationSuite(spark) \
    .onData(orders_df) \
    .addCheck(check) \
    .run()

verification_result_df = VerificationResult.checkResultsAsDataFrame(spark, result)
```

Deequ's anomaly detection module can track metrics over time and alert when a metric deviates from its historical baseline, which is closer to what commercial tools provide.

Great Expectations supports both Pandas and Spark backends. If your team is already using Great Expectations for SQL warehouse testing, extending it to Python pipelines is natural. The same Expectation library works across both.

Commercial Options

Monte Carlo and **Collibra** can integrate with Spark pipelines, but the integration typically requires writing custom instrumentation to emit quality events to their platforms. The out-of-the-box experience is significantly better for warehouse tables than for Spark jobs. Evaluate commercial Spark integration only after the open-source options have proven insufficient.

How to Choose

Use Pandera for Pandas-based pipelines. Use Deequ for PySpark pipelines, especially at scale. Use Great Expectations if you want a unified validation layer across SQL, Pandas, and Spark. Commercial tools add little value at the Python/Spark layer that open-source tools do not already cover.

Honest Comparison: Open Source vs. Commercial

The choice between open-source and commercial data quality tools is often framed as a cost question. It is actually a maintenance question.

What open source does better:

Open-source tools are flexible. You can customize them to fit your exact workflow, integrate them with your existing alerting infrastructure, and run them in any environment. You are not locked into a vendor's data model or pricing structure. The community is active: dbt, Great Expectations, Soda Core, and OpenLineage all have large user communities and regular releases. And the cost of open-source tooling is zero for licenses (though not for engineering time to configure, maintain, and extend them).

What commercial does better:

Commercial data quality and observability platforms are built to minimize the time from "something is wrong" to "we know what is wrong and who should fix it." They provide automated anomaly detection that works without you specifying every threshold. They provide cross-system lineage that spans your warehouse, orchestrator, and BI tools without you instrumenting each one separately. They provide incident management workflows: alerting, assignment, escalation, and resolution tracking. They provide dashboards that non-engineers can use to understand pipeline health. And they come with support contracts.

The honest answer is this: most teams with fewer than 200 tables and a single warehouse should start with open-source tools. dbt tests, Soda Core, and OpenLineage with Marquez cover the majority of data quality use cases at zero license cost. The engineering time to configure and maintain these tools is real, but it is well-understood work.

Teams should evaluate commercial tools when one or more of the following is true: you are monitoring hundreds of tables across multiple systems; you need automated anomaly detection without manual threshold management; your data quality incidents are consuming significant on-call engineering time that a better alerting system would reduce; or you need cross-system lineage that open-source tools cannot provide for your specific stack.

A Word on DataKitchen

DataKitchen publishes two open-source tools relevant to this landscape. **TestGen** profiles warehouse tables and generates suggested data quality tests based on observed distributions. It is a practical tool for bootstrapping test coverage on a large number of tables where writing tests manually would take too long. It fits in the Use Case 1 section above, alongside dbt tests, Great Expectations, and Soda Core.

DataKitchen Observability monitors pipeline runs and data journeys. It tracks whether pipelines completed, whether they produced data, and whether that data meets defined expectations. It fits in the Use Case 2 section, alongside orchestrator-native SLA alerts and commercial platforms like Monte Carlo. It is open source and free to run.

Both tools are worth evaluating in the context of their use cases. They are not the only option in either category, but they are legitimate choices with active development and a real user community.

REFERENCE AND TOOLING

The 2026 Data Quality and Data Observability Commercial Software Landscape

Read the original article at: https://datakitchen.io/blog/the-2026-data-quality-and-data-observability-commercial-software-landscape/

More than 50 vendors sell data quality or data observability software. The market has never been more capable, more crowded, or more confusing. First published December 2025, updated September 2026 for the GX Cloud shutdown.

It is also consolidating. Datadog bought Metaplane. Snowflake bought Select Star. And in 2026 the category had its first outright shutdown. Great Expectations was acquired and its assets went to FICO. GX Cloud, the commercial product, closed on June 1, 2026 with about 30 days' notice to customers. The message from the big platforms is that data quality is too strategic to leave to a point vendor. The message from GX is what happens to a point vendor that no platform buys.

Between legacy suites bolting on monitoring, venture-backed startups promising AI-powered everything, and open-source projects with real traction, picking a tool feels less like software selection and more like a survival exercise. Whether you're a data engineer buried in pipeline failures, a governance lead trying to show a return on the program, or a CDO wondering why nobody trusts the dashboards, this is the map. We keep it so you don't have to.

Modern Data Observability & Data Quality Commercial Software

These focus on automated monitoring, anomalies, lineage, and data reliability for modern stacks.

- Monte Carlo: The best-known data observability platform. Monitors freshness, volume, schema, and quality incidents across warehouses and BI tools with ML-based anomaly detection.
- Bigeye: Column-level metrics and anomaly detection with SLA-style monitoring.
- Anomalo: AI-driven platform that detects data quality issues across structured and unstructured data with little manual rule configuration.
- Metaplane (now part of Datadog): ML-based anomaly detection and lineage for modern warehouses and BI tools, now inside Datadog's observability suite.
- Validio: ML-based monitoring of data quality and business metrics with segmented anomaly detection.
- Soda (Soda Cloud + Soda Core): An open-source checks framework plus a SaaS control plane for rules, metrics, and monitoring.
- Datafold: Data diffs for CI/CD workflows, plus observability for transformations and pipelines.
- Acceldata: Enterprise data observability that monitors performance, cost, and quality across large data platforms.
- Lightup: SaaS observability with automated anomaly detection for key tables and metrics.
- Synq: Data quality and observability for analytics pipelines: freshness, schema change, anomalies.
- Pantomath: Cloud-native data observability and monitoring for modern data infrastructure.
- ICEDQ: Data testing and monitoring platform for ETL validation and migration projects.
- RightData: Self-service data quality platform that lets business users profile, test, and monitor data.
- FirstEigen (DataBuck): Automated data quality validation that uses ML to generate and run rules.
- Databand (IBM): Pipeline monitoring and data quality with end-to-end visibility into data workflows, now part of IBM.
- Kensu: Data observability built for DataOps teams, tracking lineage and quality across pipelines.
- Telmai: AI-based data observability that detects and alerts on quality issues without manual configuration.
- DataKitchen (Enterprise DataOps Data Quality TestGen + Observability): Open-source and enterprise tools for automated test generation, profiling, anomaly detection, and end-to-end data journey observability.
- DQLabs: Cloud-native data quality and observability with AI-based profiling, anomaly detection, and quality scorecards.
- Rakuten SixthSense: Enterprise data quality and observability with automated monitoring and alerting.
- Sifflet: AI-augmented data observability for technical and business teams: automated monitoring, lineage, alerting.
- Elementary: dbt-native data observability: anomaly detection, quality monitoring, reporting.
- Evidently AI: Open-source ML monitoring for data drift, model performance, and data quality.
- Piperider: Open-source data reliability toolkit that profiles data and tracks change over time.

- Qualytics: Automated data quality and observability with AI-driven anomaly detection and profiling.
- AWS Glue Data Quality (Amazon): Managed DQ service built on Deequ with rule recommendation, scores, and anomaly detection.
- Great Expectations / GX: Open-source expectations framework. Its commercial product, GX Cloud, shut down on June 1, 2026 after the company was acquired; the GX Core project continues under Fivetran stewardship, so there is no longer a commercial GX to buy.

GX Cloud customers got about 30 days to migrate. Read When the cloud goes dark: a note to Great Expectations customers for what happened and how to move your tests.

Traditional Commercial / "Augmented" Data Quality Platforms

These are the big enterprise suites you'll see in Gartner-style evaluations.

- Informatica Data Quality / Cloud Data Quality: Broad DQ plus profiling, cleansing, matching, and monitoring, often bundled with the integration stack.
- SAP Data Services: ETL plus data quality and cleansing, often paired with SAP Information Steward for governance.
- SAP Information Steward: Metadata, profiling, and rule-based DQ on SAP and non-SAP sources.
- Oracle Enterprise Data Quality (EDQ): Profiling, matching, address verification, and monitoring for Oracle and non-Oracle data.
- Ataccama ONE: Data quality, MDM, governance, and catalog in one platform.
- Qlik Talend (Talend Data Fabric / Cloud): Integration plus DQ (profiling, cleansing, matching) across cloud and on-prem.
- IBM InfoSphere QualityStage: Enterprise DQ focused on matching, standardization, survivorship, and MDM scenarios.
- SAS Data Quality: Profiling, cleansing, and matching, tightly integrated with SAS analytics.
- Precisely Data Integrity Suite / Trillium heritage: DQ, enrichment, and address validation with strong location and postal capabilities.

Looking for open source? The 2026 Open-Source Data Quality and Data Observability Landscape maps the projects, including which ones ship the whole product in open source and which ship a demo.

Catalog / Governance Platforms With Data Quality Features

Some catalog and governance tools have embedded or tightly integrated DQ engines:

- Collibra Data Intelligence Platform + Collibra DQ & Observability: Catalog and governance with embedded DQ and observability.
- Informatica Intelligent Data Management Cloud (IDMC): Cataloging, governance, and DQ in one cloud platform.

- Experian Data Quality Platform: Customer and contact data quality and governance, with profiling, monitoring, and enrichment.
- Atlan: Active metadata platform with quality monitoring.
- Alation: Data catalog with quality indicators.
- data.world: Collaborative data catalog with quality features.
- Stemma (acquired by Teradata): Data discovery and quality.
- Select Star: Automated data discovery with lineage. Now part of Snowflake.
- Castor: Data catalog with automated documentation.
- DataHub (LinkedIn): Open-source metadata platform.
- Acryl Data: The commercial company behind DataHub, offering a managed catalog and observability.
- OpenMetadata: Open-source metadata and data quality platform with data discovery.
- Apache Atlas: Open-source data governance and metadata framework with data quality capabilities.
- Amundsen: Lyft's open-source data discovery and metadata platform.
- Microsoft Purview: Governance and catalog first, with data quality, classification, and policy features across Azure and hybrid.

Specialist Contact / Reference Data Quality Vendors

Focused on customer and contact data, addresses, and identity, but still very much "data quality software":

- Experian Data Quality: Address, email, and phone verification, enrichment, matching, profiling, and ongoing monitoring.
- Melissa Data Quality: Address, email, and phone verification, dedupe, profiling, and enrichment; components for SQL Server, ETL tools, and SaaS.

What the GX Cloud Shutdown Means for a Buyer

Every vendor on this page has an owner, and the owner has a plan for the product that is not your plan. For most of the modern list, the owner is a venture fund with an exit clock, usually seven to ten years from the first check. The category has taken on more than a billion dollars of that money, and enterprise list prices still average about $172,000 a year for capabilities that have converged across vendors. Acquisitions, repricing, and sunsets are how that math resolves. GX Cloud is the first shutdown. It will not be the last.

A platform buys a point tool to grow its own bill, not to shrink yours.

So add two questions to the RFP that no feature matrix asks. Who owns this vendor, and what clock are they on? And if the product disappears in 30 days, what do I keep? A tool that runs in your environment, with an open-source edition that is the whole product rather than a trial, leaves you with your tests. A SaaS-only product with a proprietary rule format leaves you with an export.

The AI Has to Be Able to Call the Tool

One more question for the RFP, because it will matter more than most of the checklist: how does my assistant use this? Your team already has Claude, Copilot, or Cursor open all day. A data quality tool those assistants cannot call is a dashboard somebody has to remember to look at. The Model Context Protocol (MCP) is the open standard that fixes this. The bar is not “has an MCP server.” The bar is complete: can the assistant read a profile, generate a test, run it, record the disposition, and set a monitor, or can it only fetch a score? Ask each vendor for the MCP tool count and how many of those tools can write. TestGen’s answer is 96 tools across the whole loop, each running under the connected person’s own role, in the open-source edition.

Where This Leaves You

The market is not going to get simpler. Expect more acquisitions, more feature overlap, and more vendors claiming to do everything. The best tool is not the one with the most features or the best demo. It is the one your team will use on a Tuesday. Get clear on the real problem first: broken pipelines, stakeholders who don’t trust the numbers, a regulator. Evaluate against that pain, not a generic checklist. Decide whether you need enterprise hand-holding or whether your team can run open source. And think hard about what happens when the hot startup gets acquired or pivots, because in 2026 that stopped being hypothetical.

The fanciest observability platform will not save a data architecture that is a mess to begin with. Tools multiply good practice. They do not replace it.

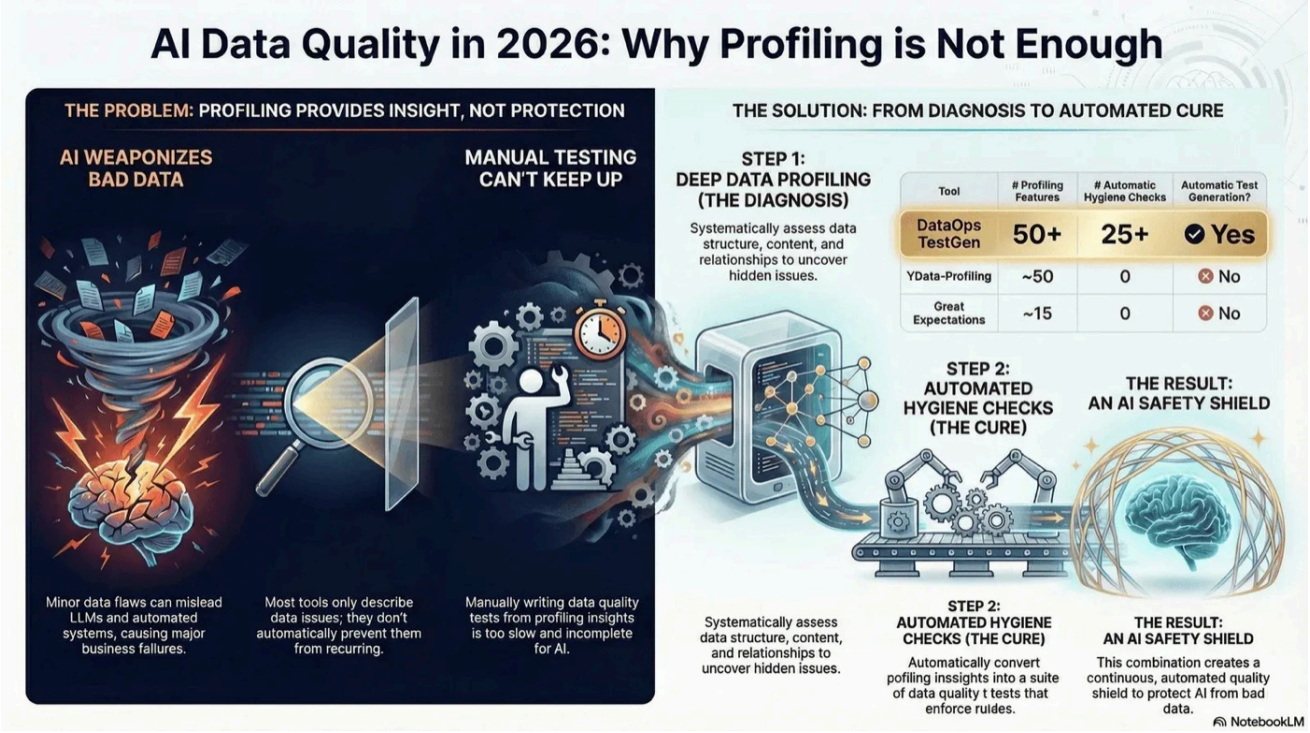
REFERENCE AND TOOLING

The 2026 Open Source Data Profiling Software Landscape

Read the original article at: https://datakitchen.io/blog/the-2026-open-source-data-profiling-software-landscape/

The 2026 Reality: When Bad Data Turns AI Against You

The world did not ease into AI. It sprinted past you. One month you were managing dashboards and debugging pipelines. The next, your LLM was telling the CEO that revenue was up 40 percent while sales were in freefall. Your analytics engineers were vibe-coding SQL transforms at midnight, hoping today wasn't the day they broke every downstream model. Your predictive systems, once the pride of the data science program, were degrading faster than you could retrain them. And the BI reports? Nobody trusts them. Worse, nobody trusts you.



Welcome to 2026, where bad data doesn't stop at a broken dashboard. It gets fed to a model that acts on it. The bind is that AI demands clean data at the exact moment when volume, velocity, and complexity have made hand-written testing impossible to keep up. Your pipelines feed systems that retrain every hour, decide in milliseconds, and ingest dozens of sources your team barely understands. Your quality process is still SQL tests written like it's 2015: slow, brittle, incomplete.

Data quality is no longer a governance checkbox. It is AI safety at enterprise scale, and it starts with knowing what is in your columns.

What Is Data Profiling? And Why AI Has Made It Urgent Again

Data profiling computes statistics that describe what is in your data: distributions, null rates, distinct counts, minimums and maximums, value patterns. It is the diagnostic step that tells you whether your AI systems are being fed truth or poison. It comes before modeling, visualization, or any decision, especially a decision a model makes on its own.

Gartner defines profiling as a technology used for "discovering and investigating data quality issues," which lets you trace errors to their origin and certify data as fit for use. In an AI pipeline the definition has teeth. A column that quietly went 40 percent null does not produce an error. It produces a model that is confidently wrong, with no step in between where a person would have noticed.

Core Techniques of Data Profiling

Profiling splits into three techniques.

Structure discovery checks format and consistency. It catches misformatted postal codes, phone numbers containing letters, and fields whose values break the schema's assumptions. The question it answers: *does this data look like what it is supposed to be?*

Content discovery looks inside the values: patterns, missingness, anomalies, inconsistencies. It finds nulls, misspellings, invalid values, and systemic errors. The question: *what is actually in here, and does it make sense?*

Relationship discovery analyzes connections between tables and fields. Dependencies, correlations, referential integrity. The question: *how do these datasets relate, and can those relationships be trusted?*

What Profiling Buys You

Done well, profiling exposes governance and compliance risk early, catches errors before they propagate downstream, and gives you a map of sources, relationships, and lineage. It is the first serious act of data hygiene. And in a world where AI consumes data continuously and silently, hygiene is everything.

But a profile on its own is a document. Nothing enforces it.

Why Traditional Profiling Tools Are Falling Short

The open-source projects in this space are good at what they do. YData-Profiling generates a rich report with one line of code. Deequ profiles at Spark scale. DataCleaner, OpenRefine, Aggregate Profiler, Metabase, and Apache Griffin each earn their place. Great Expectations ships a profiler that suggests

expectations, though its story changed this year: the company was acquired and its commercial product, GX Cloud, shut down on June 1, 2026. The open-source GX Core continues under Fivetran stewardship, so the profiler is still there. The paid product that used to sit on top of it is not.

All of them share one flaw. They stop at insight.

They describe the data and do not enforce its quality. They identify problems and do not prevent them. They produce visibility, not safety.

That gap used to be tolerable. A human read the report, wrote a rule, and moved on. In the AI era there is no human in that seat. When an LLM reads malformed data it does not ask for clarification. It hallucinates. When a model sees unexpected categories it does not ask for help. It degrades silently. What you need is not descriptive profiling but prescriptive, automated hygiene enforcement.

The Comparison: Profiling Depth vs. Hygiene Automation

Tool	URL	Profiling Descriptions	Profiling Checks	Automatic Data Hygiene Checks	Automatic Test Generation?	The 2026 Verdict
DataOps Data Quality TestGen (Open Source)	github.com/DataKitchen/dataops-testgen	55	50+	25+	Yes	The only open-source tool built for autonomous data hygiene.
YData-Profiling	github.com/ydataai/ydata-profiling	~35 to 45	~50	0	No	Rich profiling, zero protection.
Great Expectations (Profiler)	github.com/great-expectations/great_expectations	~10 to 15	~15	0	No	Manual expectation writing is still required. GX Core continues under Fivetran; GX Cloud shut down June 1, 2026.
Deequ	github.com/awslabs/deequ	~10 to 15	~20	0	No	Strong at scale, weak at hygiene.
DataCleaner	sourceforge.net/projects/dataquality	~20+	~25	0	No	A strong profiler that stops short of enforcement.
OpenRefine	github.com/OpenRefine/OpenRefine	~15	~15	0	No	Great for cleanup, not for automated quality.
Aggregate Profiler	sourceforge.net/projects/dataquality	~10	~10	0	No	Lightweight; not enough for an AI-driven organization.
Metabase (OSS)	github.com/metabase/metabase	~5 to 10	~10	0	No	Useful for BI; irrelevant for AI safety.

IMPORTANT

Profiling shows you what’s wrong.

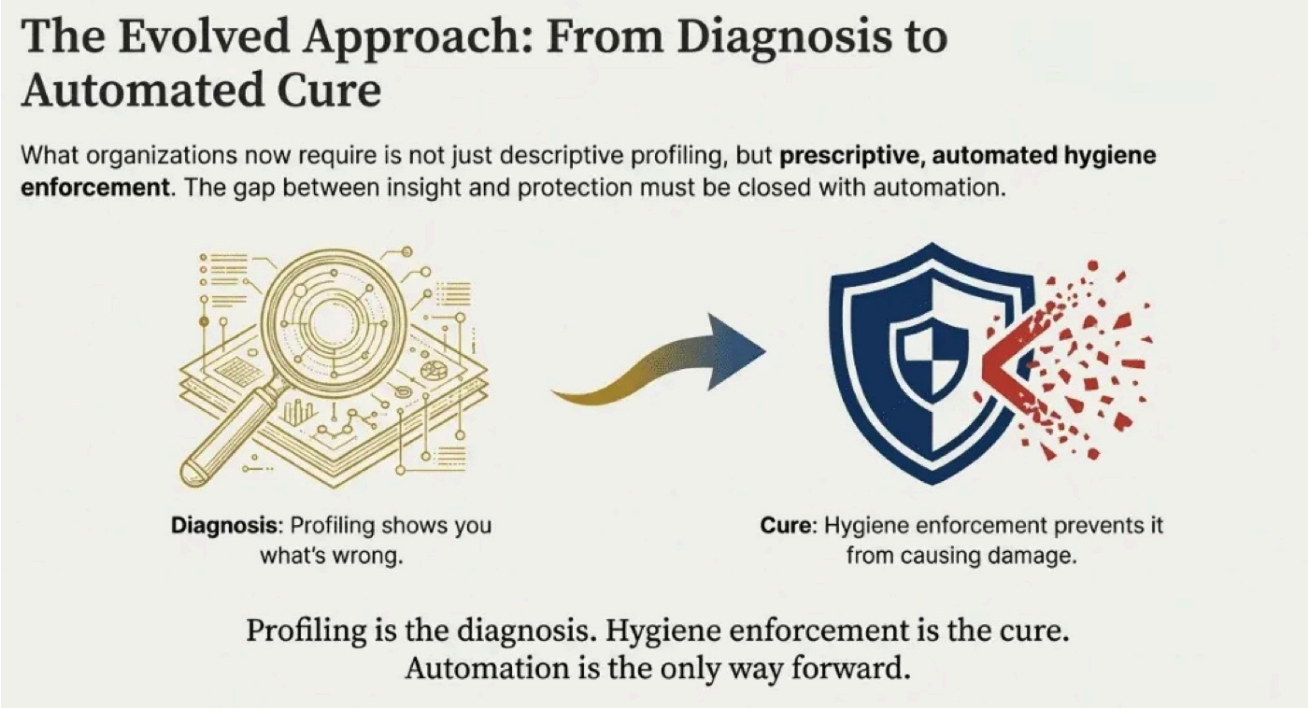
Hygiene automation prevents it from causing damage.

AI needs both.

The Breakthrough: Automated Data Hygiene Testing From Profiling Signals

DataOps Data Quality TestGen is the only open-source tool on that table that links profiling to action. It profiles 55 characteristics per column, then uses the results to generate more than 120 data quality tests and apply 25-plus hygiene checks: missingness, null anomalies, duplicates, type mismatches, pattern violations, value-range issues, cardinality surprises, referential failures, and distribution shifts.

The profile becomes enforcement instead of a report you skim once.



Nobody writes rules. Nobody maintains YAML. Nobody has to anticipate every way a column can fail, because the profiler already measured how it behaves. That is the shift from passive understanding to running AI safety: deep profiling across the whole estate, prescriptive hygiene checks instead of descriptive statistics, and no manual quality process anywhere it can be avoided.

IMPORTANT

Profiling is the diagnosis. Hygiene enforcement is the cure.

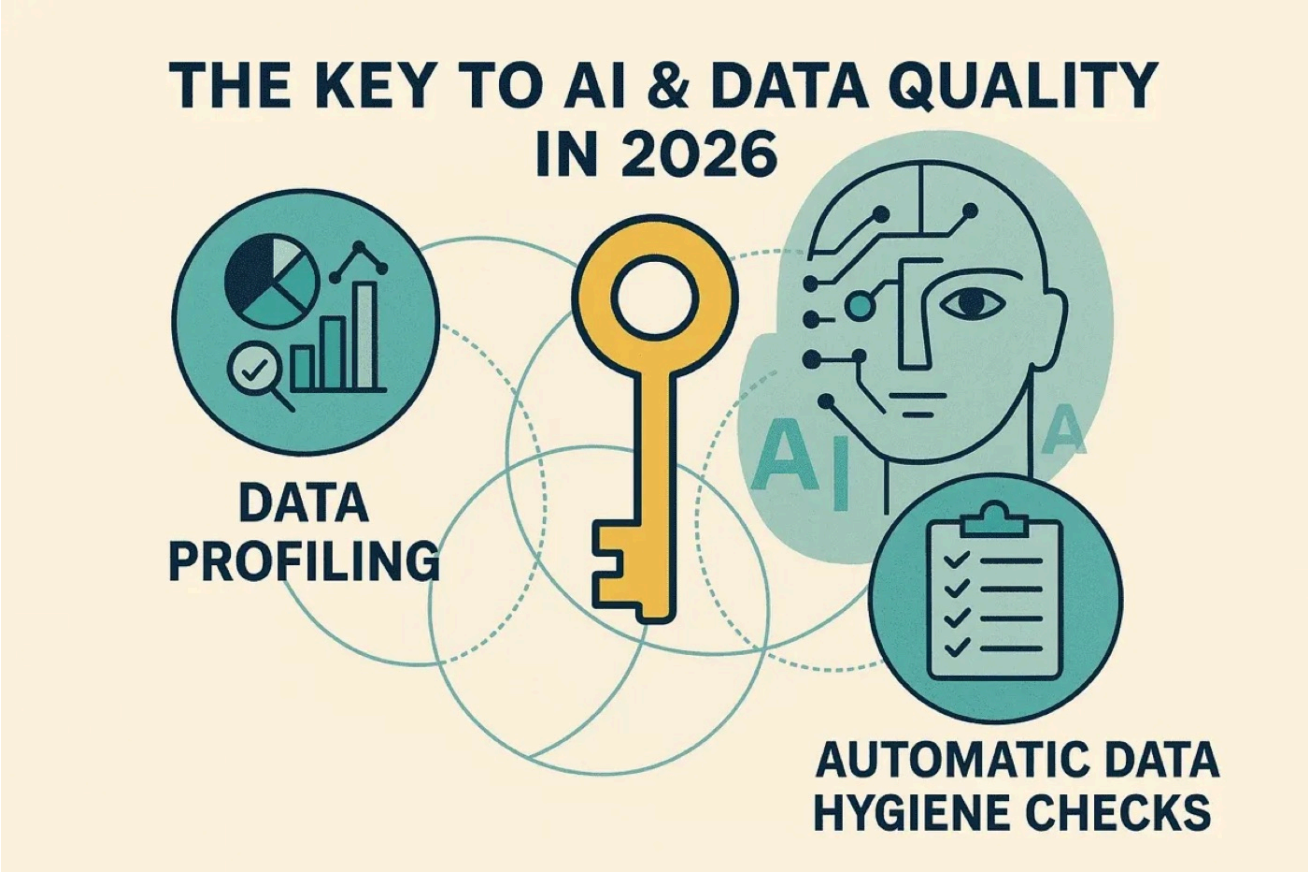
The Profile Has to Reach the AI, Not Just the Engineer

There is a second gap in the table above, and it is newer. Every one of those tools writes its profile for a human to read: an HTML report, a notebook cell, a dashboard. Your assistant cannot read any of it. So when you ask Claude or Cursor whether a column is safe to train on, it guesses from whatever you pasted into the chat.

The Model Context Protocol (MCP) is the open standard that closes this. A profiler with a complete MCP interface hands the assistant the actual statistics, the hygiene findings, and the test history, and lets it act on them under your permissions. TestGen's MCP server exposes 96 tools across the whole data quality loop, profiling included, in the open-source edition. In one session, a single question about a database's inventory and health came back with a quality score of 78.9 and 30 hygiene issues nobody had tested for. One supposedly clean table was still carrying names, emails, zip codes, and income flagged as PII. No report was opened. The assistant read the profile directly.

A profiling tool your AI cannot call is a PDF.

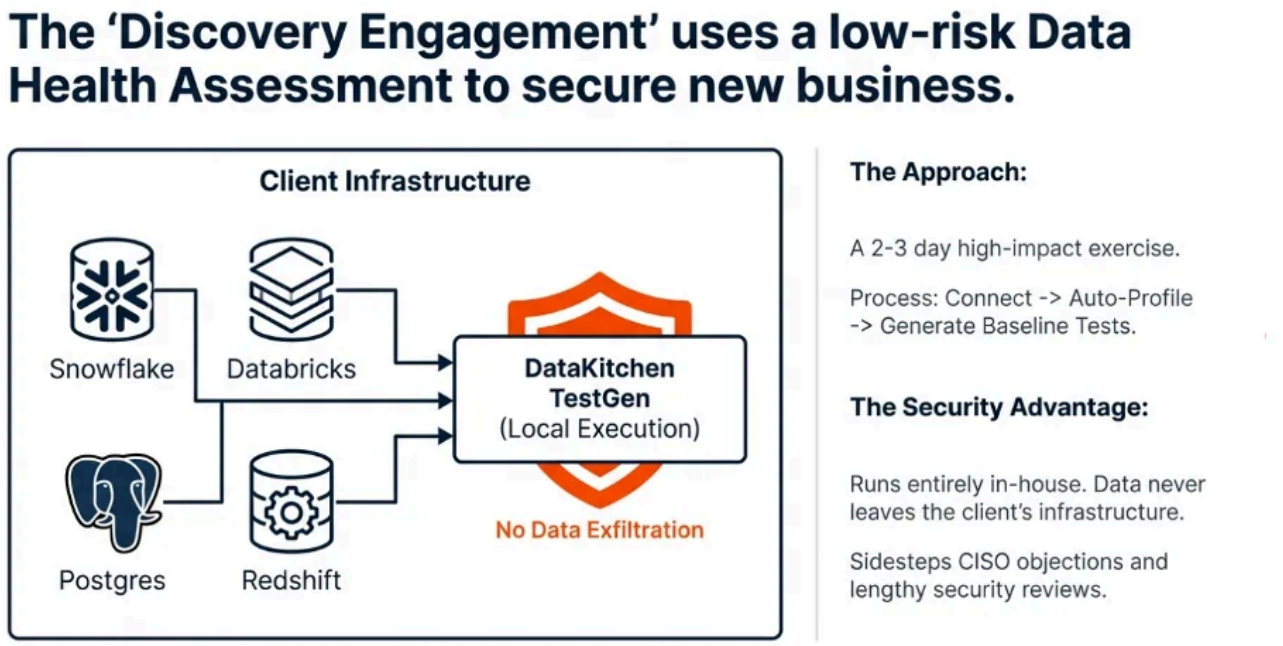
Final Word: Data Must Be Understood & Clean Before It Is Powerful



REFERENCE AND TOOLING

The Consultant's Growth Playbook: Accelerating Client Acquisition with DataKitchen Open Source TestGen

Read the original article at: https://datakitchen.io/blog/the-consultants-growth-playbook-accelerating-client-acquisition-and-retention-with-datakitchen-open-source-testgen/



If you've worked in data and analytics consulting, you know the hardest part isn't always the technical side. The real challenge is finding new clients, growing existing accounts, and proving your value to justify your fees.

The tools you choose make a difference. They impact the quality of your work and your ability to win and grow business. Expensive enterprise data quality platforms can be a barrier. Clients often hesitate at high licensing costs before seeing any value, and recommending pricey tools to budget-conscious organizations can stall deals for months.

DataKitchen's open-source data quality and observability tools, especially DataOps Data Quality TestGen, give you a strong advantage. You can show your team's value and highlight the need for your services without asking clients to commit to a budget right away. By connecting to their data and uncovering hidden issues, you can start conversations that lead to real projects.

Goal 1: Acquire New Customers

Every consultant faces the same question: how do you get started with a prospect who doesn't know you? Cold emails and presentations only go so far. What really helps is showing you understand their specific problems, sometimes even better than they do. TestGen gives you a clear way to do this.

The Discovery Engagement Approach

One approach that works is to offer prospects a simple "data health assessment" to start. Present it as a quick exercise where you connect TestGen to a sample of their data and create a detailed data quality report.

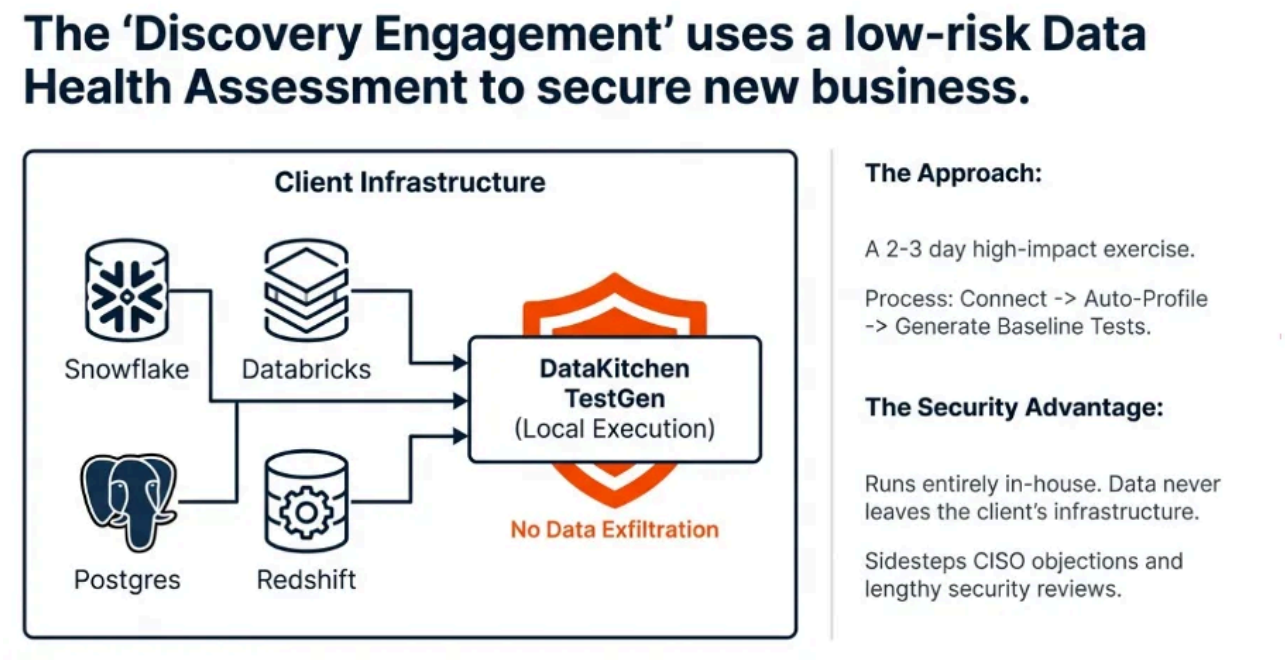
The mechanics are straightforward. TestGen connects to major database platforms (including Snowflake, Databricks, PostgreSQL, SQL Server, and Redshift). Once connected, it automatically profiles the data and generates a baseline set of data hygiene metrics and a data quality score without requiring custom logic. Within hours, you have concrete findings.

TestGen is especially good for client assessments because it runs entirely in-house. You set it up in the client's environment, so their data never leaves their systems. This removes security worries that can slow down discussions about data tools. If a CISO asks where the data goes, you can say it stays put. TestGen checks the data in place and saves results locally. For consultants, this means you can suggest running TestGen on production data without long security reviews or procurement delays.

Since TestGen works on unlimited tables, you aren't limited in what you can check. Unlike commercial tools that charge per table or data source, you can profile and score all of a client's data if needed. This allows you to give a complete assessment and show the full range of data quality issues across the organization.

Creating the Data Quality Score

You can turn TestGen's results into a data quality score, a single number that gives executives something clear to consider. Most organizations don't know how to measure their data quality. They might say it's "messy" or that some reports "can't be trusted," but they don't have a consistent way to track it.



If you tell a VP of Analytics that their customer data scores 67 out of 100, and show which tables and columns are lowering that score, you change the conversation. You're not just selling consulting services, you're showing real evidence of a problem and positioning yourself as the one who can solve it.

Converting Assessments into Engagements

The assessment itself is a small task, but it reliably leads to larger conversations. Once you've shown a client their data quality issues, the natural next question is: what do we do about it? This opens doors to remediation work, data pipeline improvements, governance implementations, custom test development, and ongoing data quality monitoring arrangements.

Since TestGen is open source under the Apache 2.0 license, you don't have to ask clients to buy expensive software before they see value. The only upfront cost is your consulting time, which you can price competitively for the first assessment, knowing more work may follow. The Apache 2.0 license also allows clients to use, modify, and deploy the tool however they want. With TestGen, there are no usage limits, no tracking, and no surprise licensing fees later.

Goal 2: Deepen an Existing Customer Relationship

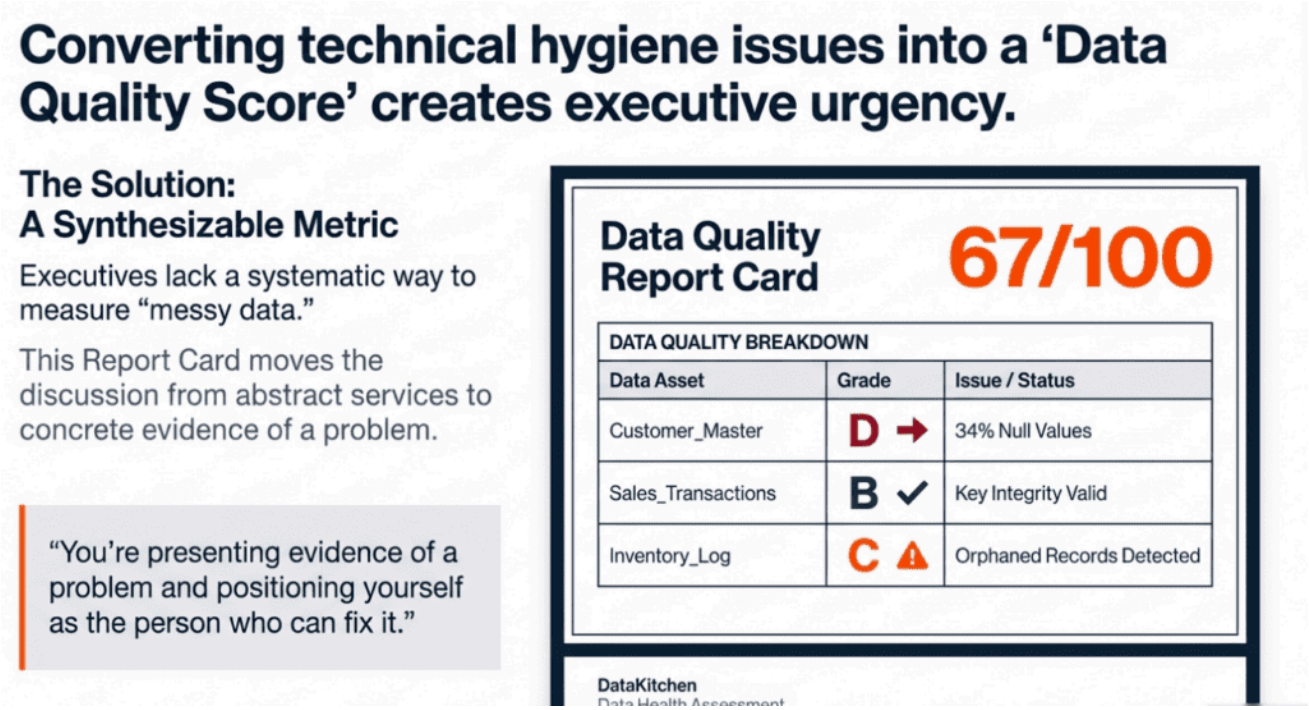
It's usually easier to grow a business with current clients rather than to find new ones. They already trust you, know how you work, and often have a budget set aside for your services. The key is to find new problems to solve and show value beyond what you're already doing.

Data quality work is often a missed opportunity in accounts where you're already handling analytics, pipeline development, or data science projects.

Surfacing Problems They Didn't Know They Had

Even clients you know well may have data quality issues you haven't found yet. This happens because data problems often stay hidden until something breaks later on. Teams usually focus on building new features instead of checking existing data, and no one wants to point out problems without offering solutions.

TestGen helps you spot issues early and come prepared with both the problem and a solution. Try running it on data sources you haven't checked before. You can profile, test, and score datasets from other teams or systems that feed into your pipelines. You'll likely find data hygiene issues worth talking about.



TestGen is built to run safely on production data, so you can use it on live systems. There's no need to make sample datasets or use old copies. The tool runs read-only queries on the real data your client relies on, giving you results that reflect the current situation, not just a cleaned-up sample.

TestGen can also monitor data for unusual changes over time. It automatically spots shifts in frequency, volume, schema, and other patterns. You can set up TestGen on a database, let it generate alerts for a week or a month, and then share those alerts with your client. For example, you might ask, "Did you know this data was late? Did you notice the schema changed? Are you missing data here?"

First, there's fixing the problems. When you find data quality issues, someone has to address them. This could mean updating source system checks, adding logic to pipelines, or working with other teams to solve the root causes. Since you found the issues, you're in a good spot to lead this work.

Third, there's custom testing. While TestGen creates automated tests, clients often need custom tests for their specific business rules. For example, a retail client might want to check that inventory counts match physical audits, while a financial services client might need tests for regulatory metrics. Building these

custom tests shows your deep understanding of their business. TestGen aims to cover about 80% of data quality tests automatically, but the remaining 20%, the unique ones, are often the most valuable and where your team's expertise really matters.

Fourth, there's the setup and making it part of daily operations. If a client finds TestGen useful during an assessment, they may want to add it to their regular systems. This means setting up the tool, connecting it to their workflows, training their team, and building processes for ongoing data quality management. With the Apache 2.0 license, there's no need for commercial negotiations, you're just helping them use open source software they can keep using as long as they want. And TestGen's Enterprise license is a reasonable cost based on the number of users and connections - not data size or usage.

The Land and Expand Model

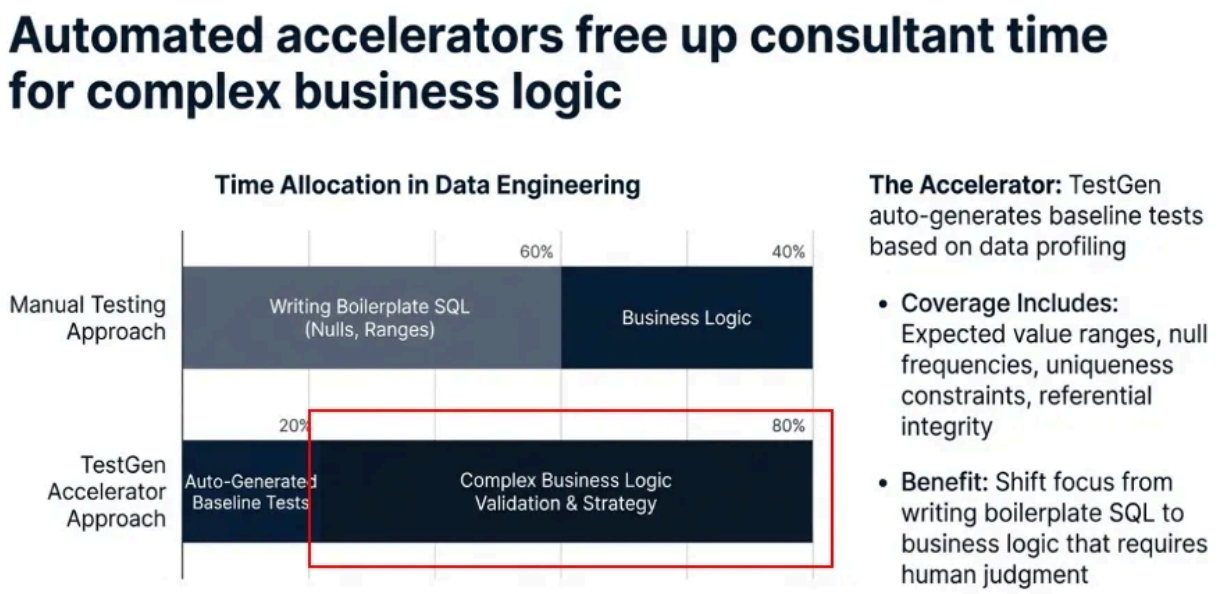
This approach is powerful because it adds value without taking away from existing work. You're not fighting for the same budget as other projects. Instead, you're finding new problems and creating new value. A client who hired you for dashboards might now want data quality help. A client who uses you for pipelines might add observability services. Each new skill opens up more opportunities.

Goal 3: Improve Project Deliverables

Beyond helping you win business, TestGen improves the quality of your technical work. In a competitive market, the quality of your deliverables sets you apart.

Automated Test Generation as an Accelerator

Writing tests is one of the most time-consuming parts of building production data pipelines. Sound engineers know that untested code is risky, but the push to deliver features often delays testing. This can lead to systems that work, until they suddenly don't, often at the worst times. TestGen automatically generates a basic set of data tests based on profiling. When you connect it to a data source, it analyzes the actual data and generates tests for expected value ranges, missing-value frequency, uniqueness, referential integrity, and more.



Automation test creation doesn't replace careful, domain-specific testing. Instead, it takes care of the basic tests, so you can focus on checking business logic that needs your expertise. The result is better test coverage, delivered more quickly.

Since TestGen can run on unlimited tables with no licensing limits, you can create complete test suites across all your data platforms. Commercial tools that charge per table often limit test coverage. With TestGen, you can test everything, staging tables, transformations, final outputs, without worrying about extra costs for your client.

Embedding Quality in Your Deliverables

When you deliver a data pipeline or analytics solution, what sets professional work apart is how robust it is. Amateur work fixes the immediate problem. Professional work solves the problem and also plans for possible future issues.

Adding automated data quality tests and observability tools to your deliverables shows this level of professionalism. When you hand over a pipeline that checks its own health, alerts on problems, and shows data quality metrics, you give your client something they can trust to run smoothly.

Move clients from vague expectations to concrete Service Level Agreements (SLAs).

Professional Delivery = Self-Monitoring Pipelines.

The SLA Definition: What does "Working" mean?

- ✔ Data arrives within the time window.
- ✔ Row counts fall within expected ranges.
- ✔ Quality scores remain > 95.
- ✔ Key metrics do not deviate beyond thresholds.

The Shift:

From Amateur Delivery ("It works until it breaks") to Professional Delivery (A pipeline that monitors its own health).

Outcome:

Shared expectations protect both the consultant and the client.

Differentiation in Competitive Situations

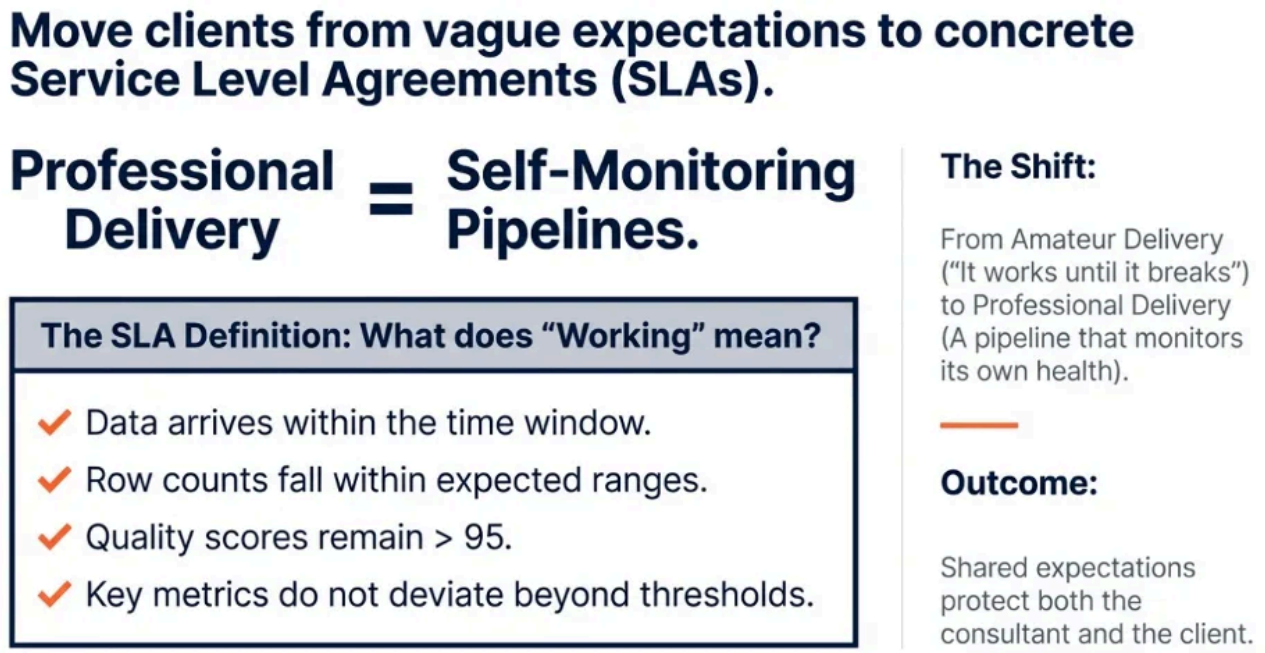
When clients are choosing between consultants, showing advanced testing and observability can set you apart. Many consultants only focus on features. By highlighting quality, reliability, and long-term success, you show you're a partner who thinks beyond just the first delivery.

Goal 4: Improve Staff Skills

If you run a consulting practice with multiple team members, the skills of your staff directly impact your capacity to win and deliver work. Investing in training pays dividends through better delivery, higher client satisfaction, and the ability to command premium rates.

Free Educational Resources And Certification Programs

DataKitchen has published detailed, free books on DataOps and DataOps change management. These resources give your team a solid foundation, not just in using the tools, but in understanding the bigger ideas behind good data operations. For consultants, this education helps in several ways. It builds technical skills, gives you a common language for talking about data challenges with clients, and shows that your team understands industry best practices.



In addition to books, DataKitchen also offers free online certification programs in DataOps and data quality. These certifications give your team a clear learning path and help them fully understand key concepts. Certifications are essential for consulting teams. They show outside proof of your team's skills, give clients confidence in your expertise, and set a clear standard for what "competent" means in your organization. This makes it easier to assess and grow your talent.

Using these free resources can become part of your company culture. New hires can complete certifications during onboarding. Teams can discuss book concepts in meetings, and delivery teams can apply best practices from these resources to client projects. Investing in skills pays off over time. Teams that keep learning and improving deliver better results, make clients happier, and earn more referrals and repeat business.

The Favorable Economics of Open Source for Consultants

Eliminating Procurement Friction

Enterprise software sales cycles are often long and unpredictable. If you recommend a tool that is a big investment, you add extra steps that you can't control. Budget approvals can take months, procurement adds complexity, and other priorities can stop deals altogether.

Open source removes these obstacles. You can start using TestGen right away, with no software costs for the client. The only investment is your time, which you're already billing. This makes it much faster to go from the first conversation to an active project.

The Apache 2.0 Advantage

TestGen's Apache-2.0 license is ideal for consulting projects. It allows commercial use with no restrictions, lets you modify it without sharing changes, and you can distribute it without paying royalties. For clients, this means they can use TestGen across their whole organization, add it to their products, and build on it without legal headaches.

For consultants, this means you can recommend and use TestGen without worrying about license compliance. There's no user counting, no usage tracking, and no risk of audits. You can build your own features on top, create custom add-ons for clients, and make it part of your regular toolkit.

Unlimited Data Test Coverage Without Unlimited Cost

A major economic benefit of TestGen is that it supports an unlimited number of tables. Most commercial data quality tools charge by table, data source, or data volume. These pricing models force you to choose between full coverage and saving money. They are too expensive.

With TestGen, you don't have to make that tradeoff. You can profile and test every table in a client's data warehouse at no extra cost. This lets you build truly comprehensive data quality programs rather than cutting corners due to budget constraints.

Security Without Compromise

Because TestGen runs entirely in-house, it solves a common concern with data tools. Many organizations, especially in regulated industries, can't send data to outside services. Even those without strict rules are often careful about letting data leave their control.

TestGen avoids this issue altogether. You install it in the client's environment, it checks the data in place, and the results stay local. There's no SaaS, no cloud dependency, and no risk of data leaving. This makes it a good fit for clients who wouldn't approve a cloud-based data quality tool.

Demonstrating Value Before Asking for Commitment

Traditional enterprise software asks clients to commit budget based on promises. They're buying features they haven't tried yet, often from vendors they don't fully trust. The client takes all the risk. Open source flips this model. You show value first. The client sees real results from TestGen on their own data before making a decision. The risk moves to you as the consultant, but you can handle it with efficient work and good client relationships.

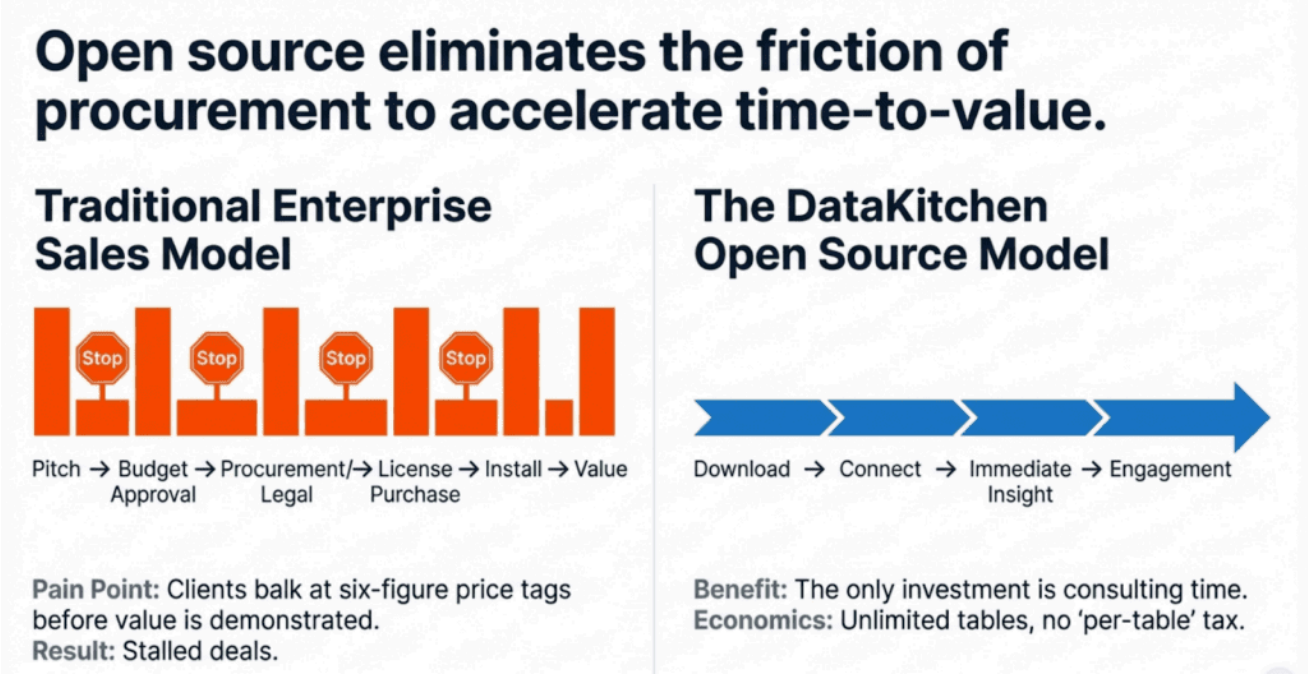
Most importantly, recommending open-source tools aligns your interests with your clients'. When they are ready to run production testing with multiple users, DataKitchen offers a low-cost enterprise version. DataKitchen has excellent support from experienced data people like you.

Clients notice and value this alignment. It builds trust, which is key to long-term consulting relationships.

Your Next Step

Begin by learning the tools yourself. Install TestGen, connect it to some data you can access, and see what it finds. Get to know its profiling, test generation, and reporting features. The better you know the tools, the more confidently you can show them to clients.

Next, look for opportunities with your current clients. Are there accounts with data quality issues you haven't tackled? Which clients could use a data health assessment? Which projects could benefit from better testing and observability? You probably have chances you haven't noticed yet.



Then, define your service. How will you present data quality assessments? What's the scope, timeline, and price? What will you deliver? A clear offering makes it easier to talk with prospects and clients.

Finally, invest in your team's skills. Share the free DataKitchen resources with them. Encourage them to complete certifications. Make data quality and observability a core strength of your practice.

Conclusion Consulting is all about creating value for clients and earning a share of that value as fees. The better you are at spotting client problems, showing solutions, and delivering results, the more successful your business will be. DataKitchen's open-source data quality and observability tools give consultants strong capabilities at every stage of the client relationship. They help you win new clients by showing value right away, deepen relationships by uncovering new problems, and improve your deliverables with automated testing and advanced monitoring. The free educational resources also help your team grow over time. The Apache 2.0 license, unlimited table coverage, and in-house setup remove the usual barriers to using powerful data quality tools in client projects. You can run TestGen on production data confidently, knowing security is built in and clients won't face surprise costs as they use it more. In a market where data quality problems are everywhere but good solutions are rare, consultants who can find and fix these issues have a real edge. Open source tools make this advantage possible without the usual barriers of enterprise software. The opportunity is there. The tools are free.

Conclusion

Consulting is all about creating value for clients and earning a share of that value as fees. The better you are at spotting client problems, showing solutions, and delivering results, the more successful your business will be.

DataKitchen's open-source data quality and observability tools give consultants strong capabilities at every stage of the client relationship. They help you win new clients by showing value right away, deepen relationships by uncovering new problems, and improve your deliverables with automated testing and advanced monitoring. The free educational resources also help your team grow over time.

The Apache 2.0 license, unlimited table coverage, and in-house setup remove the usual barriers to using powerful data quality tools in client projects. You can run TestGen on production data confidently, knowing security is built in and clients won't face surprise costs as they use it more. In a market where data quality

problems are everywhere but good solutions are rare, consultants who can find and fix these issues have a real edge. Open source tools make this advantage possible without the usual barriers of enterprise software.

A Summary Of The Consultant's Guide To Using DataKitchen's Open Source Data Quality And Data Observability

1. Acquire New Customers through Low-Commitment Assessments: Consultants can use TestGen to offer "data health assessments," which connect the tool to a prospect's data to generate a comprehensive data quality report and a concrete "data quality score." This shifts the conversation from abstract services to evidence-based problem-solving.
2. Deepen Existing Relationships by Surfacing Hidden Issues: Even in established accounts, TestGen can proactively identify hygiene issues in datasets that were previously unexamined. This naturally leads to expanded work in remediation, ongoing monitoring, custom testing, and operationalizing the tools within the client's permanent infrastructure.
3. Improve Project Deliverables via Automation: TestGen accelerates by automatically generating baseline tests from data profiling. This allows consultants to deliver more robust, professional-grade pipelines that include observability instrumentation and measurable Service Level Agreements (SLAs), serving as a major differentiator in a competitive market.
4. Enhance Staff Skills with Free Resources: DataKitchen provides free books and certification programs for DataOps and data quality. Utilizing these resources helps consulting firms build a learning culture and provides clients with external validation of their team's expertise.

The underlying advantage of this approach is the economics of open source. Because TestGen is licensed under Apache 2.0, it can be deployed in-house (addressing security concerns), run against unlimited tables without incremental costs, and used without royalty payments or licensing surprises. This aligns the consultant's interests with the client's, building long-term trust through a "value-first" model.

REFERENCE AND TOOLING

$1 Billion in Data Observability VC Investment: This Is Not Going to End Well

Read the original article at: https://datakitchen.io/blog/1-billion-ivc-investment-this-is-not-going-to-end-well/

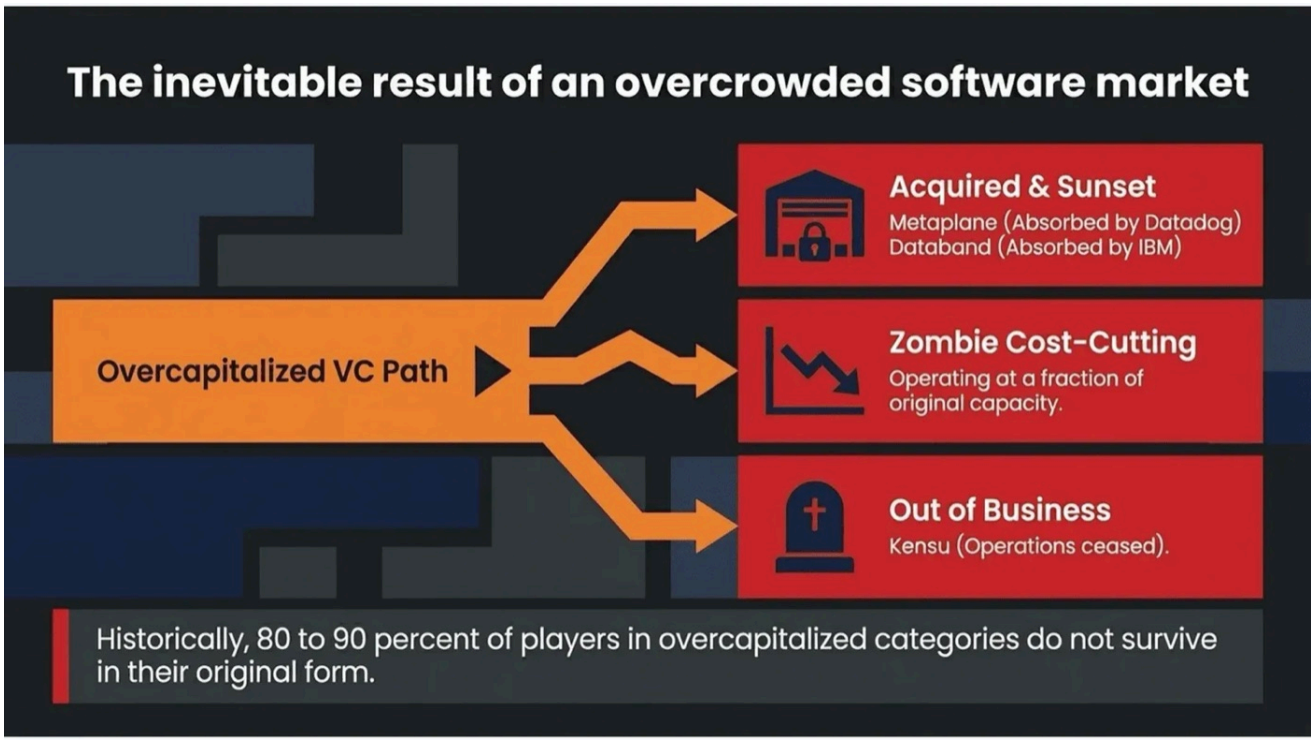
Too Much Money, Too Many Vendors, Your Problem

You have probably noticed that your inbox is full of data observability or data quality vendor pitches, your conference has a dozen booths for tools that all look roughly the same, and somehow every budget conversation for any of them starts at **six figures**. There is a reason for that, and it is not the technology's complexity. Gartner estimates that poor data quality costs organizations an average of $12.9 million per year. The tools designed to prevent that loss are now priced so that most teams can afford to monitor only a fraction of their data.

Last week, the data observability and data quality industry crossed a remarkable threshold: over one billion dollars in cumulative venture capital funding, most of it raised in the last five years. The milestone was marked by Validio's $30 million Series A, a raise framed around strengthening enterprise data quality for AI adoption. If you are a data engineer, a data quality manager, or someone responsible for choosing tools to monitor your organization's data pipelines, that number matters to you far more than you might realize. It is not a sign of a healthy, maturing market. **It is a warning.**

Venture capital investing runs on narrative momentum. When a category starts generating headlines, investors pile in, often regardless of whether the underlying economics justify the capital being deployed. The VC pile-on in data observability and data quality began around 2022 and accelerated through 2023. Investors did not carefully evaluate whether the market could absorb twenty-plus well-funded competitors. They saw a category getting traction and moved fast.

The predictable result is what always follows an overcrowded VC-funded market: the **vast majority of companies will either fail outright or be absorbed** through distressed acquisitions. Historically, in overcapitalized software categories, roughly 80 to 90 percent of the players do not survive in their original form. Some shut down. Some are acquired for their customer lists, and the product is quietly sunset. Some raise no further capital, survive on whatever revenue they have, and slowly reduce headcount until they become what VC's call 'zombie companies'.



For your team, this is entirely bad news. The companies that do survive face pressure to increase revenue because their investors need a return on investment. That pressure flows directly to you in the form of **price increases, contract restructuring, and feature gating** that did not exist when you signed your original agreement.

26 Vendors, $1 Billion, and a Price Increase Clock That Is Already Ticking

Here is the funding status across the major players in the space. These are not abstract numbers. They represent the financial pressure sitting behind every enterprise sales conversation you will have.

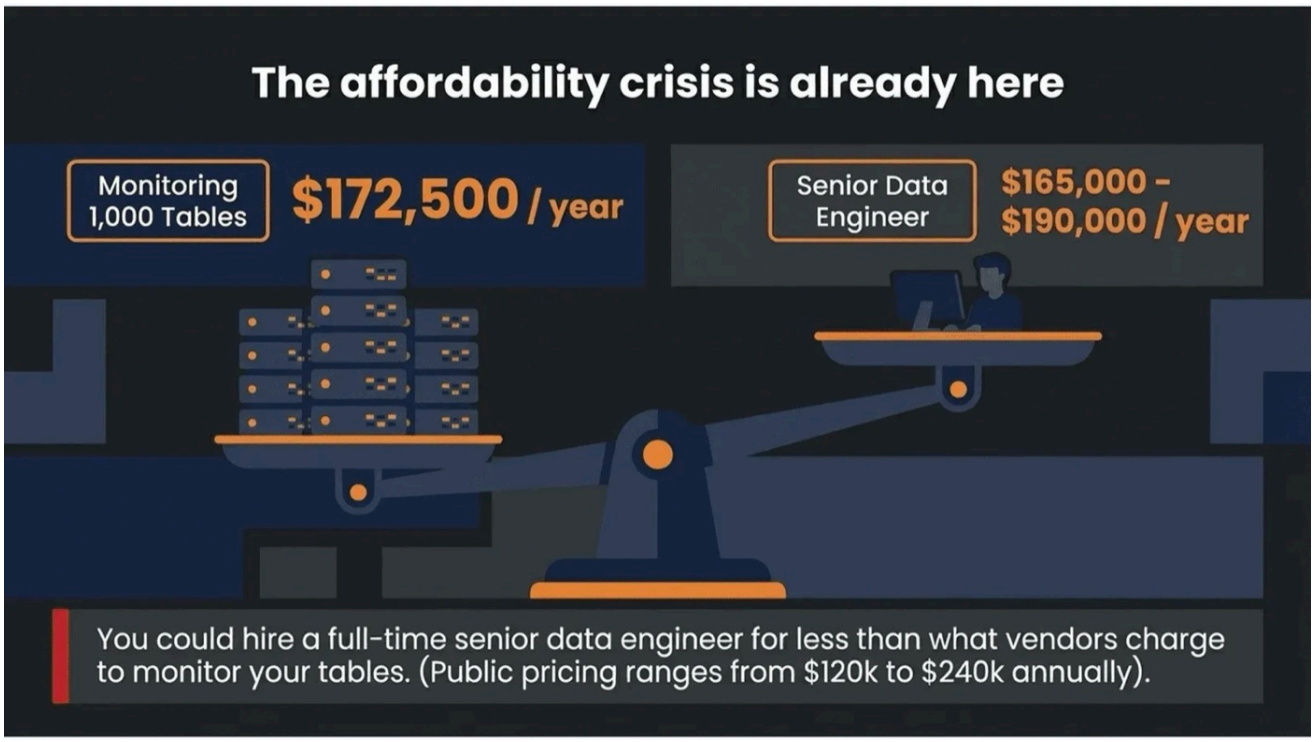
Vendor	Category	Funding	Stage
Monte Carlo	Data Observability	$371M	Series E
Acceldata	Enterprise Data Observability	$106M	Series C
Anomalo	AI-Driven Data Quality	$82M	Series B
Bigeye	Data Observability	$74M	Series B+
Great Expectations / GX	Open-Source DQ + Cloud	$65M	Series C
Sifflet	AI-Augmented Observability	$30M	Series B
Gable	Data Contracts / DevOps	$27M	Series A
Datafold	Data Reliability / CI-CD	$26M	Series A
Validio	Data Quality and Observability	$48M	Series A

Vendor	Category	Funding	Stage
Pantomath	Cloud-Native Observability	$48M	Seed
Soda	DQ Testing + Observability	$28M	Series B
DQLabs	Augmented Data Quality	$20M	Series A
Lightup	DQ Monitoring (No-Code)	$21M	Series A
Databand	Pipeline Monitoring	$15M	Acquired (IBM)
Metaplane	Data Observability	$10M	Acquired (Datadog)
Telmai	AI Data Observability	$9M	Seed
Elementary Data	dbt-Native Observability	$8M	Seed/Series A
Qualytics	AI Data Quality	$8M	Seed/Series A
Synq	Pipeline Observability	$8M	Acquired (Coalesce)
DataRecce (Recce)	Data Testing / CI-CD	$4M	Pre-Seed
RightData	Self-Service DQ	$5M	Seed/Series A
ICEDQ	ETL / Data Migration Testing	$0	Bootstrapped
FirstEigen (DataBuck)	Automated DQ Validation	$0	Bootstrapped
Kensu	Data Lineage & Observability	$7M	Closed CEO, Founder, looking for a job
Rakuten SixthSense	Enterprise DQ (Corporate)	N/A	Corporate
DataKitchen	DataOps + DQ + Pipeline Obs.	$0	Profitable, Independent
TOTAL (tracked vendors)	**26 vendors**	**$1,021M+**	

The acquisitions of Metaplane, Databand, and, just this week, Synq, warrant careful analysis. When a data infrastructure company acquires a point solution, it rarely invests heavily in that product. Founders leave, moving on to their next startup with a feather in their cap. Usually, the acquiring company aims to eliminate a competitive overlap, absorb a customer base, or add a feature to an existing product bundle. In both cases, the independent roadmap for the acquired product typically comes to an end. If you were a Metaplane customer, you are now a Datadog customer, whether you wanted that or not.

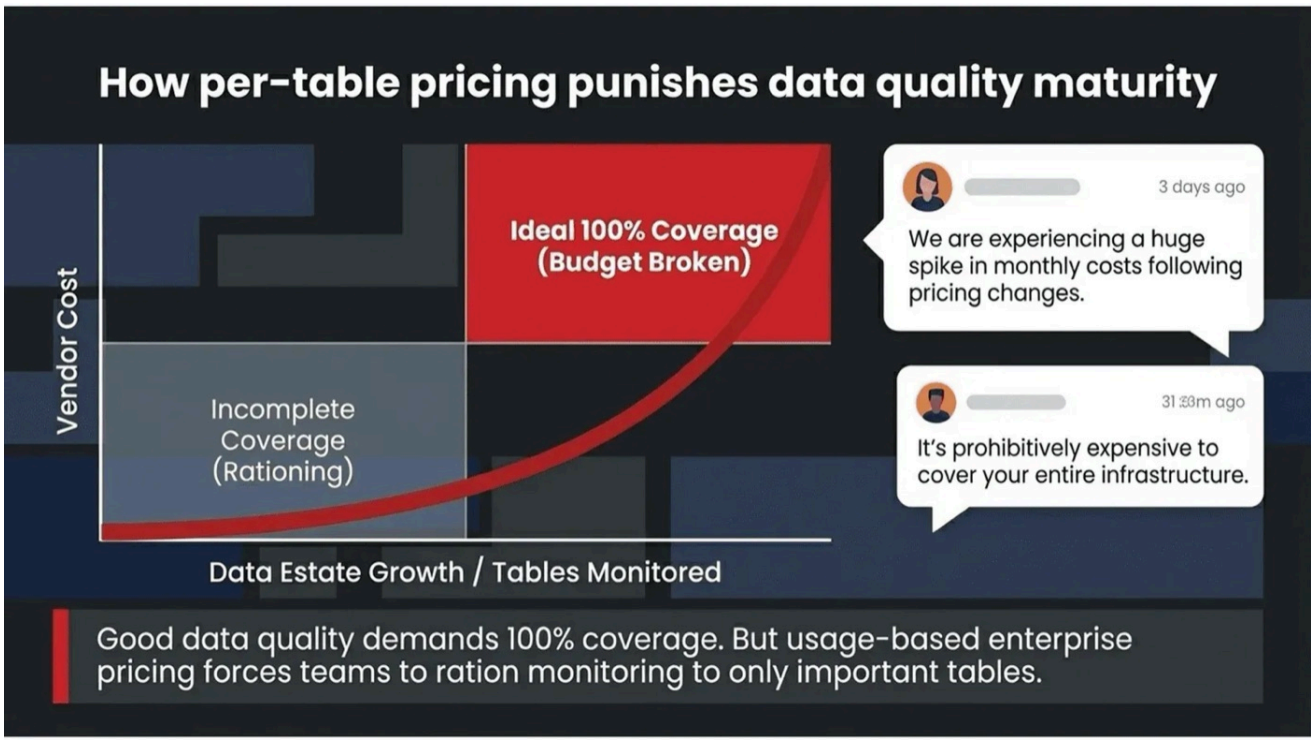
Monitoring 1,000 Tables Costs More Than Hiring an Engineer.

The **affordability crisis** in data observability is here. The funding situation has produced a concrete, measurable problem for buyers today. The industry average cost to monitor 1,000 tables is approximately $172,500 per year.



For context, the median annual salary of a senior data engineer in the United States is roughly $165,000 to $190,000. You could hire a full-time engineer for less than what some vendors charge to monitor your tables. When surveying publicly available pricing from major data observability vendors for a 1,000-table monitoring scenario, the results ranged from approximately $120,000 to $240,000 annually. When the floor is $120,000, teams stop asking whether to monitor everything and start asking which tables they can afford to leave dark.

Usage-based pricing creates a perverse outcome for teams trying to do the right thing. Instead of monitoring everything, they are forced to pick and choose. They decide which tables are “important enough” to monitor and leave the rest uncovered. Testing limits is the opposite of good data quality practice. In any serious quality assurance framework, coverage is the point. You do not get to choose which parts of your factory floor to inspect and which to ignore.



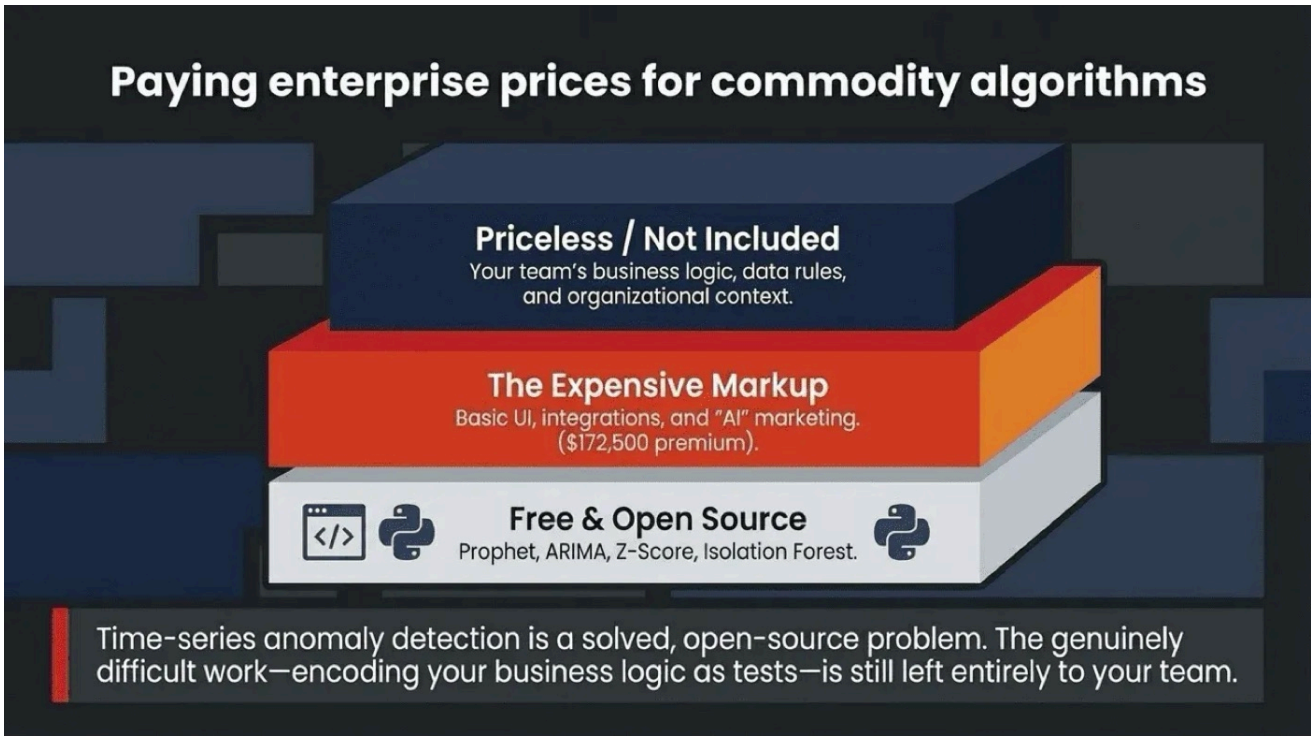
To be fair, Monte Carlo and Acceldata primarily sell to Fortune 500 data organizations with hundreds of engineers, multi-cloud environments, and dedicated data infrastructure budgets. But the pricing model does not stay in that lane. The same per-table pricing that lands at $500,000 for a large enterprise lands at $100,000 for a mid-market team with 600 tables, which is most of the teams reading this.

In today's AI-driven world, data quality is no longer a luxury. But most data teams are caught in a vise: the cost of data quality is unrealistic, and the price of failure is unacceptable. It's not just that enterprise data quality software is outrageously priced. It's that data quality needs are too often underestimated and underfunded, until it's too late. The fact is that data quality in the real world has to be achievable, affordable, and efficient.

You Are Paying Enterprise Prices for Commodity Algorithms

Every data observability vendor on that list runs on the same open-source algorithms: Facebook's Prophet, ARIMA, Z-score, and Isolation Forest. Their Python implementations are on GitHub. A working implementation is half a page of code. The $1 billion in VC funding did not yield better algorithms. It produced better sales teams and higher price targets.

Most vendors wrap these free components in a UI, add data connectors, and go-to-market as "AI-powered enterprise software." The mere mention of AI in a vendor's pitch is not evidence of proprietary technology. It is evidence of a higher target price point, set by investors who need a return on $1 Billion. That is what produces high-cost, extractive, usage-based pricing.



The anomaly detection is the easy part. The hard part is knowing that your revenue table should never have a negative value, or that a specific column cannot be null after a certain date. None of that knowledge lives in an algorithm. It lives in your team. The expensive vendors are not selling you that knowledge. They are selling you the easy part at a premium price.

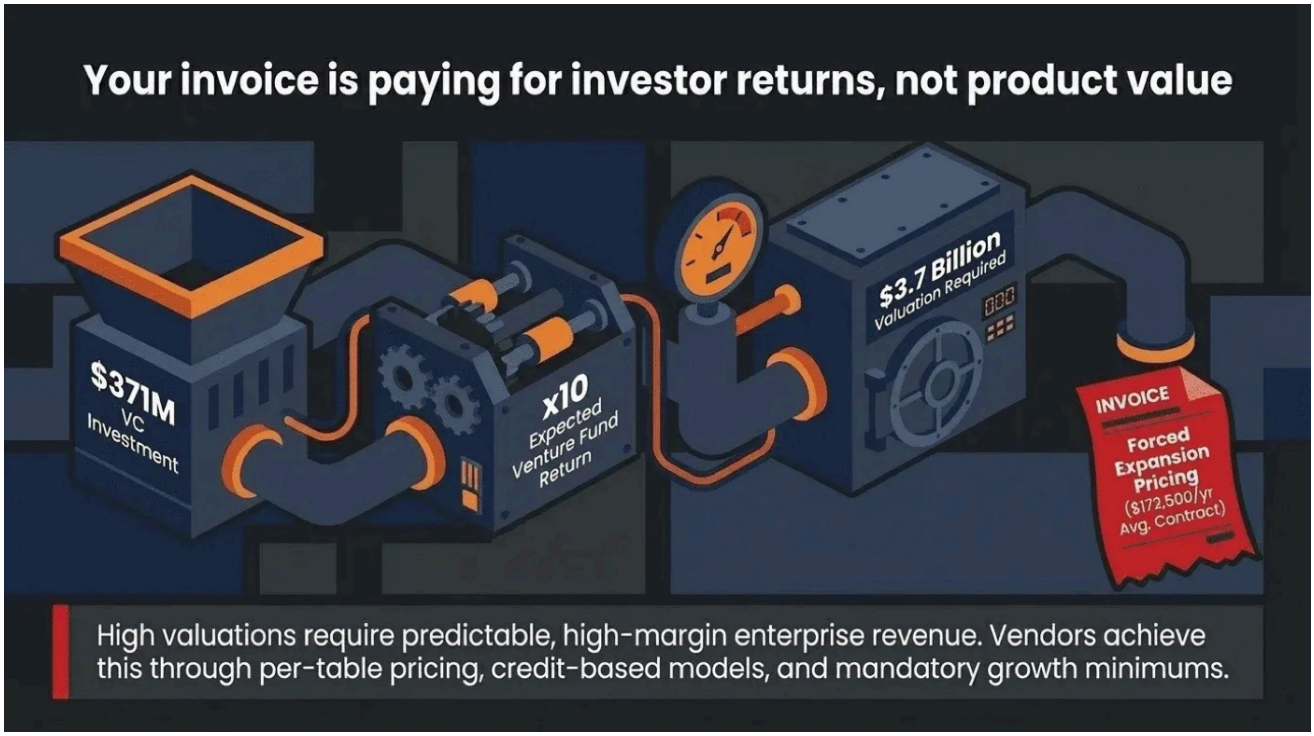
Building it yourself is an option (until it becomes a second job). dbt tests and a few Python scripts can get a small team meaningful coverage for close to zero dollars. Maybe you will just ask Claude. But the maintenance compounds quietly. Alerting breaks when pipeline topology changes. The profiling logic that no one documented becomes a Friday night incident. The business analyst who needs to understand why last Tuesday's report was wrong has no UI to adjust the test parameters. And every new engineer who joins inherits a bespoke monitoring stack with no owner, no roadmap, and no community.

Your Invoice Is Paying for Investor Returns, Not Product Value

Understanding why these prices exist requires understanding the financial math behind the companies charging them. Monte Carlo has raised $371 million. A typical venture fund models for a 10x return on successful investments. That means investors in Monte Carlo's later rounds need the company to reach a valuation of several billion dollars to produce the returns they modeled. That valuation requires predictable, high-margin, enterprise-scale revenue.

The way you get to that revenue is not by charging $10,000 a year. It is by charging $120,000 to $200,000 or more, anchored on a pricing model that grows with usage. Per-table pricing, credit-based consumption models, and annual contracts with mandatory growth minimums are not coincidental choices. They are deliberate mechanisms to extract increasing revenue from existing customers year over year.

In 2022, DataDog charged one of its customers (Coinbase) $65 million for one year of enterprise IT Observability. This is not a misprint. The next year, the crypto industry hit a sudden bump, hurting Coinbase's business. As revenue dried up, the company turned its attention to reducing its overly high costs. For IT observability, Coinbase moved away from DataDog and onto an open-source Grafana/Prometheus/ClickHouse stack.



In their VC pitches, Monte Carlo called themselves 'DataDog for data engineers.' This was not a functional definition. It's a pricing and valuation vision for their investors. In its latest investor deck, Datadog discussed its land-and-expand pricing model. They start at $50K and expand to over $500K per year on average, with over 600 $1M+ and 34 $10M+ customers. Data is the lifeblood of many companies. And those six, seven, eight-figure numbers are what's in your future with the current Data Observability market.

The boards of data observability are looking at the parallels with IT observability. "One customer went from $50K to $24 million in 15 years, growing 48 percent annually." A Redmonk analyst cited the statistic: "Most companies now spend somewhere between 20 and 25% of their entire infrastructure budget on their IT observability tools." The boards of your data observability vendor see this as a preview of what data quality pricing looks like if this market matures the same way. 25% of your budget is genuinely alarming.

As one data engineer noted in a public thread on r/dataengineeringafter a vendor repricing event: "We are experiencing a huge spike in monthly costs following pricing changes." Another in the same community warned: "It's prohibitively expensive to cover your entire infrastructure." These are not isolated

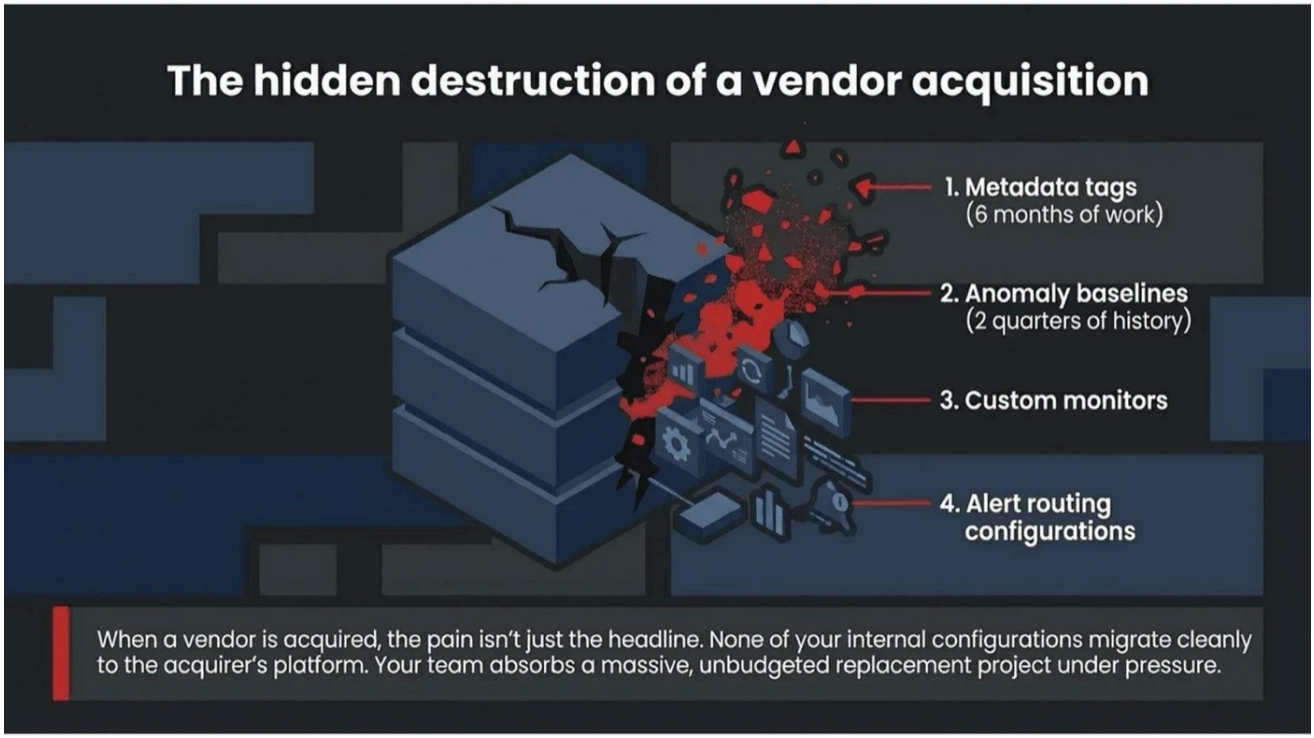
complaints. Search any data engineering forum, and you will find the same pattern repeating: a team signs a contract at one price, grows their data estate as good data teams do, and discovers the bill has compounded in ways that were not obvious at signing.

These are not edge cases. They are the predictable outcome of investors who need their portfolio companies to demonstrate rapidly expanding revenue, channeled through pricing models that make expansion the default behavior. The cost of the technology does not set the pricing. It is not set by the value delivered to you. It is set by what a spreadsheet in an investment committee meeting says the company needs to justify its valuation.

When the Funding Stops, So Does Your Vendor

A significant number of companies in the table above will not be able to raise their next funding round. The venture market for data infrastructure has cooled substantially since 2022. Investors who were willing to fund a Series B at a 20x revenue multiple in 2022 are looking for actual profits, or at a minimum, a credible path to them, in 2026. There is another VC bubble where everyone is putting their funding in 2026: AI.

Companies that raised large funding rounds but haven't reached the revenue level to justify their next valuation will face tough choices. They can attempt to raise money in a down round, which hurts morale, resets option strike prices, and signals trouble. They might try to be acquired, often at a price that eliminates the value of common stockholders, including employees. They could also try to cut costs enough to become profitable, which usually involves significant layoffs. Or they might shut down. None of these options is ideal for the teams that built their data quality programs on top of their products.



If you are currently evaluating a vendor that has raised $50 million or more and has not disclosed anything resembling profitability, you are taking on real business continuity risk. The product you buy today may not exist in its current form in eighteen months. This is not speculation. Look at the list. Kensu is gone. Databand is part of IBM, which has shown little enthusiasm for developing it. Metaplane is inside Datadog. Two acquisitions and one apparent shutdown out of roughly twenty-five significant players in a space that is only a few years old.

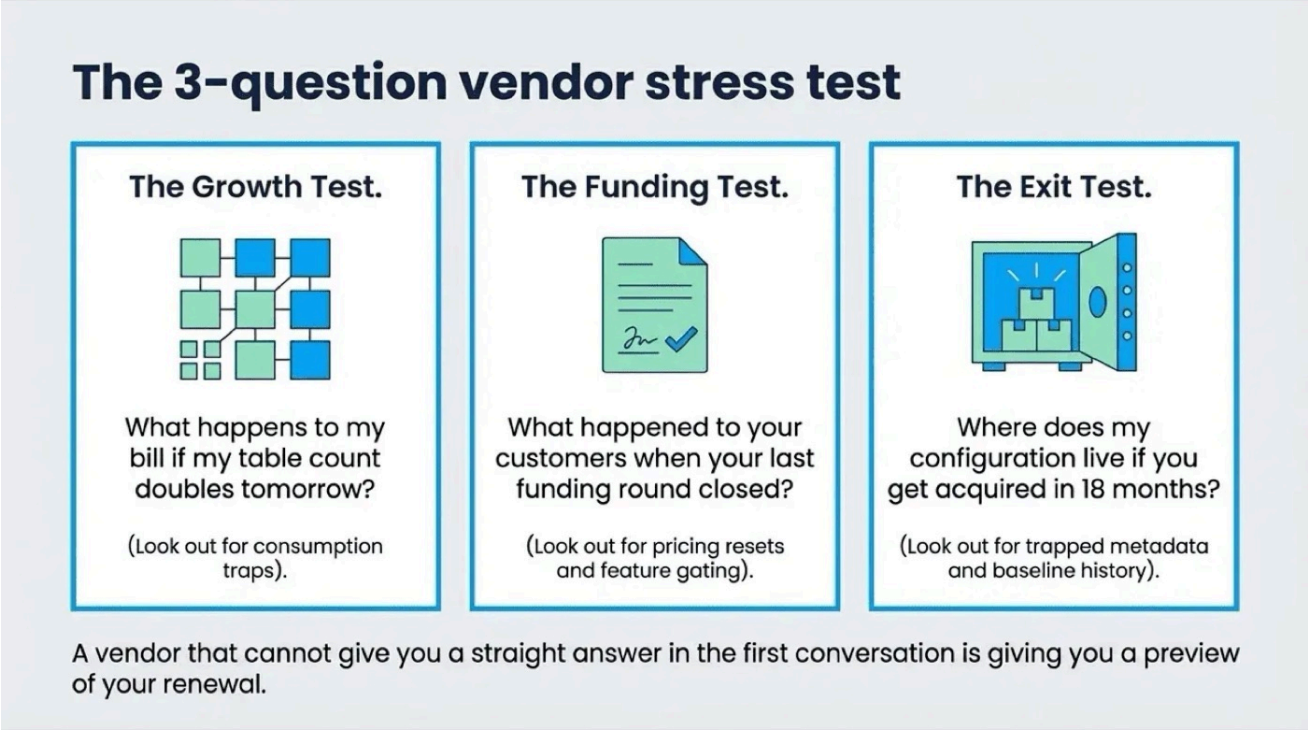
And when an acquisition happens, the pain is not just the headline. It is everything your team built inside that tool. The metadata your engineers tagged over six months. The anomaly baselines took two quarters of ingestion history to become meaningful. The custom monitors are tuned to your specific data. The alert routing for your on-call rotation depends on. None of that migrates cleanly to the acquirer's platform. In most cases, it does not migrate at all. Your team absorbs a replacement project under pressure, on top of everything else they were already doing. That cost never appears in the original contract. It should.

A reasonable counterargument is that the market consolidating to three or four well-capitalized survivors with stable roadmaps and deep integrations could be better for enterprise buyers than twenty fragmented point solutions competing for the same budget. History suggests otherwise. The IT observability market consolidated around Datadog, New Relic, and Dynatrace, and customers are paying more. Consolidation does not produce lower prices. It produces pricing power.

Three Questions to Ask Before You Sign Anything

Given everything above, here is a practical framework for evaluating data observability and data quality tools. Three questions cut through most of the noise.

Ask what happens to your bill if you double your table count tomorrow. Per-table and consumption-based pricing models are designed to grow your costs every time you do the right thing. Add more data? Pay more. Run more tests? Pay more. A vendor that cannot give you a straight answer on this in the first sales conversation is telling you something important about how the relationship will go at renewal.



Ask what happened to their customers when the last funding round closed. A new investment round almost always follows pricing resets, contract restructuring, and feature gating. If the answer is nothing changed, ask for that in writing. If the answer is evasive, treat it as a preview.

Ask what your exit path looks like if this vendor is acquired eighteen months from now. What do you own? Where do your configuration, your test history, and your anomaly baselines live? Can you export it? If the honest answer is that a migration would cost your team three months of work, that cost belongs in your total cost of ownership calculation from day one.

If you are already in a contract, this section matters more to you than it does to someone evaluating for the first time. Renewal is where you have the most information and the most leverage. You know what the tool actually costs your team in time and frustration, not just in dollars. You know whether the anomaly detection produces alerts that your engineers act on or noise they have learned to ignore. You know what the vendor said during the sales process and whether it matched reality. And you have a credible exit threat, which is the only negotiating position that produces a better price. The three questions below are not just for new evaluations. They are the questions to bring into your renewal conversation.

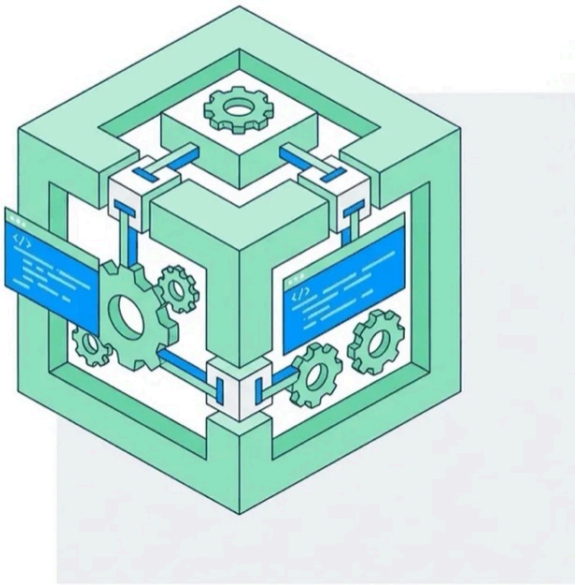
What a Data Observability Tool Looks Like When No One Needs a 10x Return

We are obviously not a neutral party here. We built TestGen because we could not justify the cost of the tools that existed, and because we kept running into data teams who were rationing their monitoring coverage for the same reason. What follows is our answer to the problem we just described. You should evaluate it with the same skepticism you would apply to any vendor.

There is a profitable, transparent alternative

DataKitchen built TestGen and DataOps Observability for data teams who refuse to ration their monitoring coverage.

- Zero Venture Capital Raised
- Profitable & Independent
- No Series C investors demanding 3x YoY growth
- Open-Source Foundation



Feature	TestGen Open Source	TestGen Enterprise	Typical Observability Vendor
Price	$0 (free forever)	$100/user or DB/connection	$50,000 to $200,000+ per year
Complete User Interface	Yes	Yes	Yes
Data Profiling (51 characteristics)	Yes	Yes	Partial
Auto Test Generation	Yes	Yes	No
Hygiene Issue Detection (27 types)	Yes	Yes	No
Quality Scoring and Dashboards	Yes	Yes	Partial
Table Monitors (ML anomaly detection)	Yes	Yes	Yes
SSO / Multi-project / RBAC	No	Yes	Yes
Pricing Transparency	Yes	Yes	No
VC-Backed Pricing Risk	None	None	High

The auto-generated tests deserve particular emphasis. One of the primary reasons data engineers do not test more is the overhead of writing tests manually. Writing a test for one table takes time. Writing tests for 500 tables is a multi-week project. TestGen profiles your tables, identifies the meaningful characteristics of your data, and automatically generates a complete test suite. This is not a feature the $150,000-per-year vendors offer because their revenue model depends on you needing to do more, not less.

A fair question is whether a small profitable company is any safer than a 300-person VC-backed one. The honest answer is that the size of our team is irrelevant, since TestGen is open source. If DataKitchen ceased to exist tomorrow, the code would not disappear. Your tests, your configuration, and your baselines live in your environment, not ours. That is the actual answer to the sustainability question, and it is one no VC-backed vendor can give you.

The Market Is Telling You Something. Listen

One billion dollars in venture capital invested in a software category does not make that category more trustworthy. It makes it **more expensive and more fragile**. The companies that raised that capital are obligated to produce returns for their investors. Those returns will come from customers. The data engineering community deserves tools that are priced based on the cost of the technology and the value it delivers, not on what a venture fund spreadsheet says a company needs to be worth.

> **"The conference booth size isn't the signal; the number on the pricing page is."**
>
> One billion dollars in venture capital doesn't make a software category more trustworthy—it makes it more expensive and **fragile.** The data engineering community deserves tools priced on the value they deliver, not on what a VC spreadsheet demands.

None of this is a criticism of the founders building these companies. Starting a company is genuinely hard. We have been at this for 12 years, and we know firsthand that the late nights, stress, and struggles are real for anyone trying to build something from nothing. I hope all these founders succeed. But the nature of VC fund returns and technology markets is what it is. You did not cause them, and you cannot change them. What you can do is make a buying decision that prevents your team from being exposed when those dynamics unfold.

The conference booth size isn't the signal; the price number on the page is.

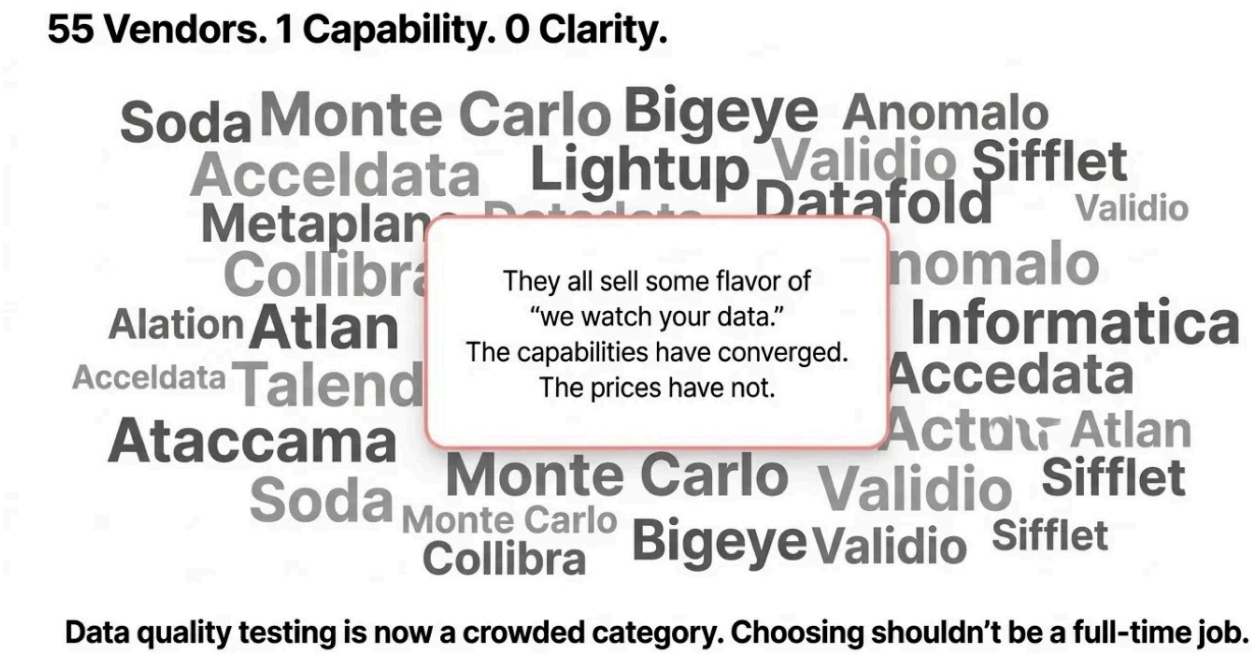
REFERENCE AND TOOLING

There are a lot of freaking data quality and data observability vendors

Read the original article at: https://datakitchen.io/blog/there-are-a-lot-of-freaking-data-quality-and-data-observability-vendors/

We just published a comparison page that lays out 55 vendors in the data quality, data observability, and data governance (with data quality testing) space side by side. License model, deployment, capabilities, and price. One table. No “request a demo to see pricing.”

The list is long because the category is crowded. Soda, Monte Carlo, Bigeye, Anomalo, Acceldata, Lightup, Validio, Sifflet, Metaplane, Datafold, Collibra, Alation, Atlan, Informatica, Talend, Ataccama, and another three dozen vendors all sell some flavor of “we watch your data.” The capabilities have converged. The prices have not. We’ve written three deep-dive landscape posts if you want the full map: the 2026 commercial software landscape, the 2026 open-source data quality and observability landscape, and the 2026 open-source data profiling landscape.



Psychologists call this choice overload. Sheena Iyengar's jam study showed that shoppers given 24 jams to taste bought ten times less jam than shoppers given 6. More options, fewer decisions. The data quality category has 55 jams.

Here's where TestGen stands out.

What you get in 15 minutes

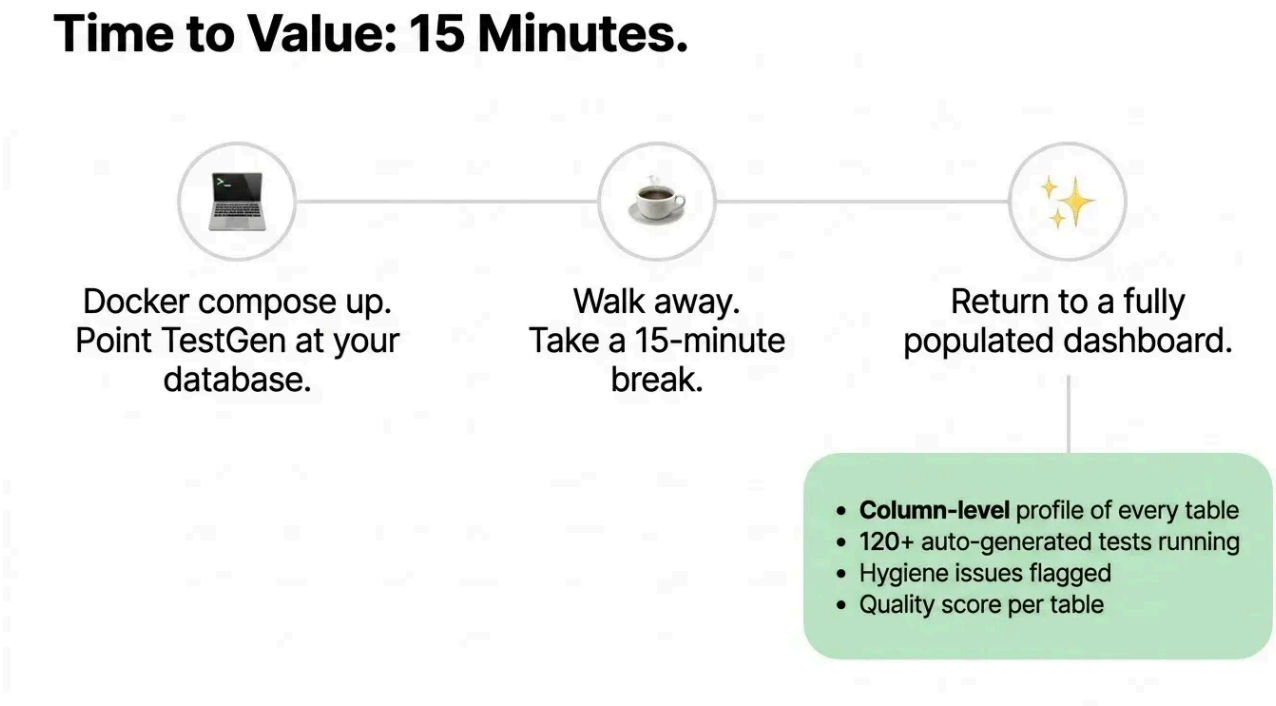
Docker compose up. Point TestGen at your database. Walk away for a coffee. When you come back, you have a column-level profile of every table, 120-plus auto-generated tests running against your data, hygiene issues flagged, and a quality score per table. That's the whole onboarding. No checks.yml to write. No schema.yml to maintain. No vendor solutions architect on the calendar.

TestGen runs on Snowflake, Databricks, Azure Synapse, Azure SQL, SQL Server, Microsoft OneLake, BigQuery, Redshift, Aurora PostgreSQL, Oracle, SAP HANA, PostgreSQL, and Salesforce Data 360. If your warehouse is on that list, you're done with the connector question. Install instructions are at docs.datakitchen.io.

Open source, you can actually run

TestGen is Apache 2.0. The full testing engine, profiling, auto-generated tests, hygiene detection, and anomaly monitoring run on your laptop or your cluster. No commercial license needed. Same code in both the open-source and enterprise editions.

The 120-plus auto-generated tests aren't 120 null checks dressed up as tests. They cover functional dependency violations, referential integrity gaps, distribution shifts, cardinality surprises, pattern violations, value-range issues, type mismatches, and freshness and volume drift. Soda Core makes you write checks.yml. dbt tests make you write schema.yml. TestGen writes the tests for you. You edit the ones you don't like.



In-database execution, your data never leaves the perimeter

Point TestGen at a database. It profiles every column, writes the tests, and pushes the queries into your database for speed and security. Your data never leaves the perimeter. Security review goes faster when you can tell the team there's no data egress.

TestGen also runs as a quality gate inside your pipeline. Drop it into an Airflow task, a Databricks job DAG, or a CI step. Bad data fails the build the same way bad code does.

Real support from the people who built it

DataKitchen is a profitable, bootstrapped company that has shipped DataOps software since 2013. Same team, same product line, same pricing. The engineers who wrote TestGen answer support tickets. No chatbot wall. No offshore tier-one triage. No acquisition shutdown email landing in your inbox.

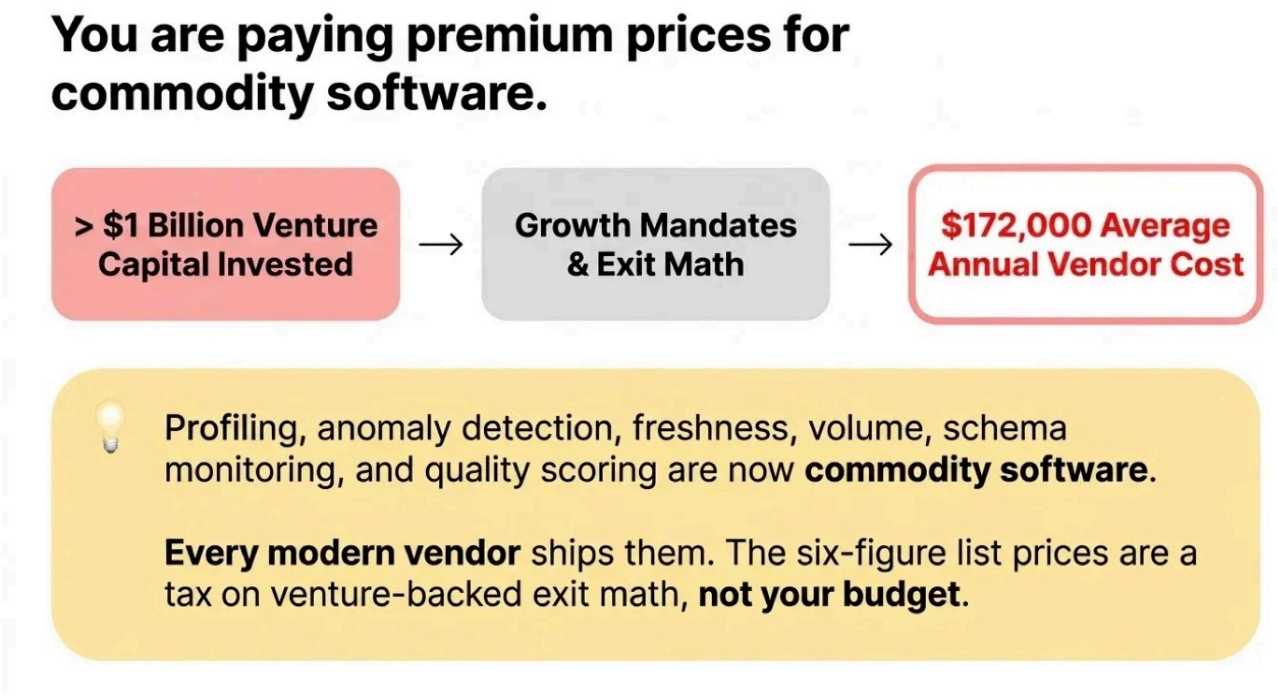
Bristol Myers Squibb runs TestGen on the commercial datasets behind its specialty drug reporting. Progeny Health runs it on neonatal claims data. Pharma and healthcare don't tolerate flaky tooling.

Fair, predictable pricing

TestGen Enterprise is a flat $100 per month per user and database connection. No per-table tax. No credit metering. No negotiated mystery quote. Enterprise vendors on the comparison page average $172,000 a year. TestGen Enterprise is $12,000 a year for 10 users on 1,000 tables.

Data quality and data observability are now commodity software

You should not be paying premium prices for any of it: profiling, anomaly detection, freshness, volume and schema monitoring, quality scoring. Every modern vendor on the list ships them. The six-figure list prices in this category are premium charges for what has become commodity software. We laid out the math in You're Massively Overpaying for Data Observability.



Anomaly detection and a catalog you'll actually use

TestGen runs out-of-the-box freshness, volume, schema, and data drift monitors. The data catalog gives you metadata, hygiene issues, PII risk, test results, and Critical Data Elements in a single view. Shareable issue reports route the right ticket to the right Slack channel with one click.

Venture-backed, growth-mandated

Most vendors on the comparison page are backed by venture capital. The pricing, the roadmap, and the support model are built around exit math, not your budget. The data observability category has attracted more than a billion dollars in venture investment, and we wrote about why that's not going to end well for buyers.

Predictable Pricing. Zero Gotchas.

Open Source	TestGen Enterprise
$0	$100 / user / month
• Apache 2.0 license. • Run locally or on your cluster. • Full testing engine.	• No per-table tax. No mystery quotes. • ($12,000/yr for 10 users on 1,000 tables).

Break your choice overload. Run Docker Compose today and get your first quality score in 15 minutes. Docs at docs.datakitchen.io.

Break your Choice Overload

Install the open-source TestGen and try it on your data. Fifteen minutes from Docker Compose up to your first quality score. The auto-generated tests, profiling, and anomaly monitors cover what most teams need without writing a line of YAML.

We'll become your favorite jam.

REFERENCE AND TOOLING

There Is No Microsoft Data Quality Tool in 2026. There Are Nineteen.

Read the original article at: https://datakitchen.io/blog/no-microsoft-data-quality-tool-there-are-nineteen/

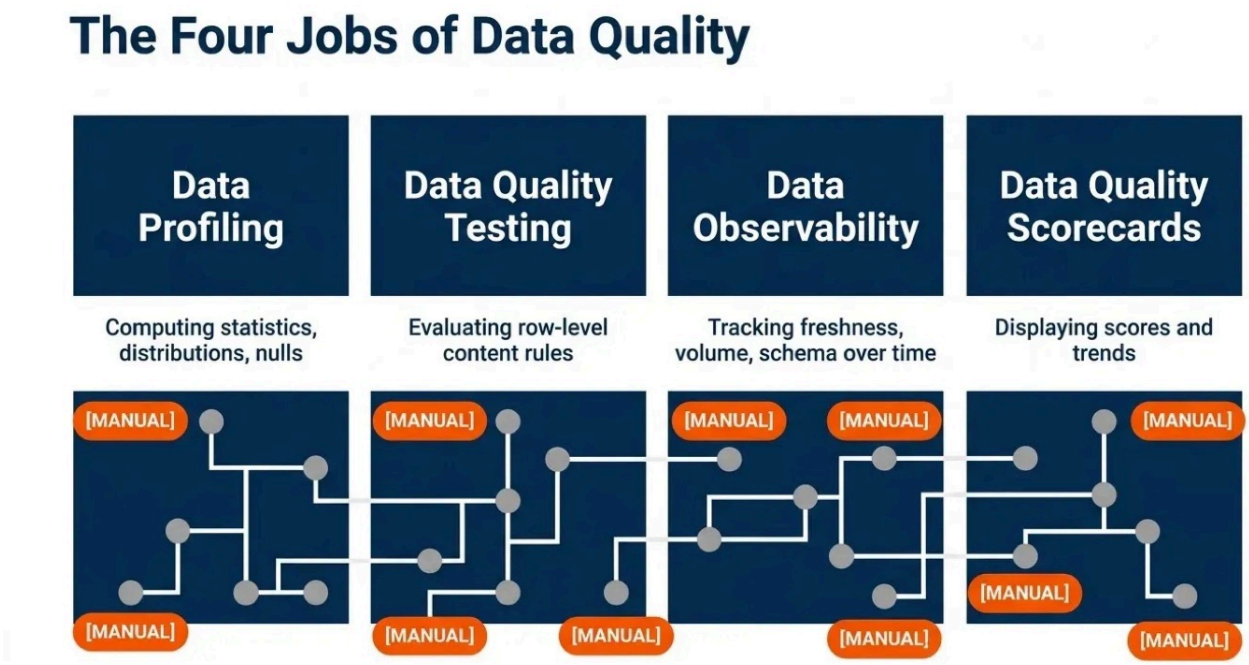
You run a Microsoft shop. Someone in compliance asked you to verify that the numbers in the report are correct. So you went looking for the data quality tool.

There isn't one. There are nineteen, spread across five products, and they don't talk to each other.

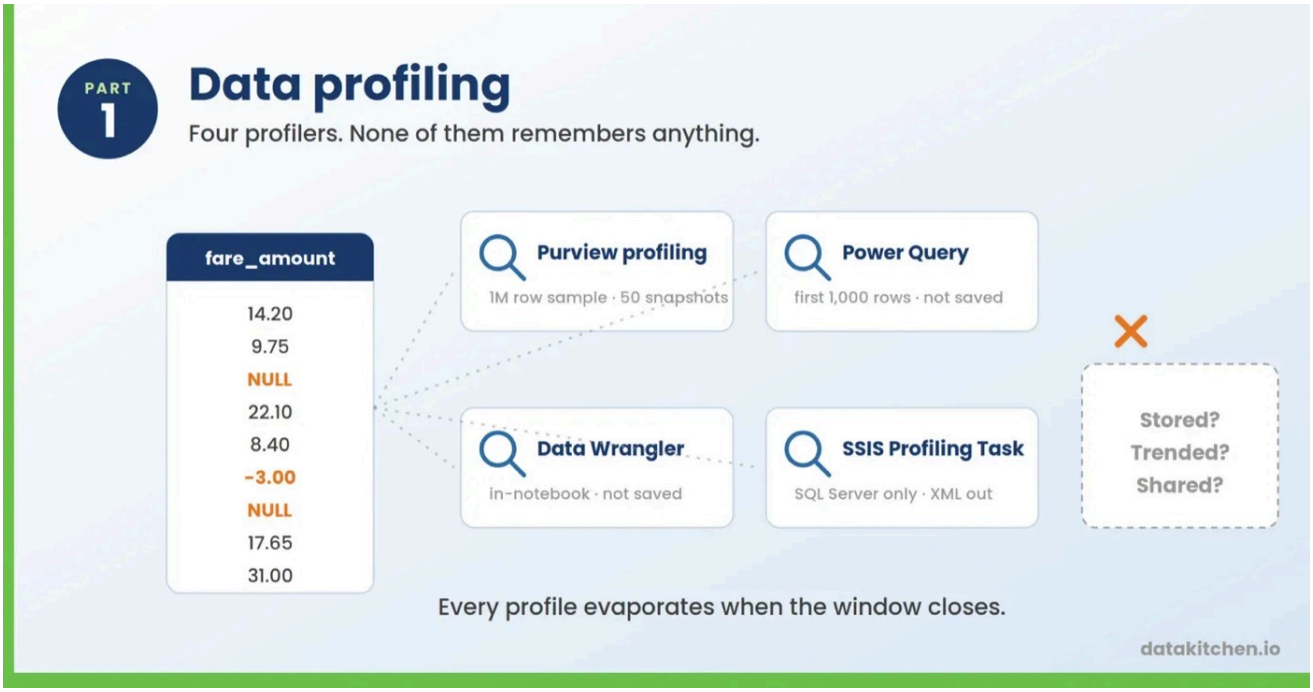
That's not a knock on Microsoft. It's what happens when a data quality story gets assembled from a governance catalog, a Spark feature, a dataflow transform, a Power Query panel, a T-SQL keyword, a streaming engine built for IoT, and two products that were deleted last year. Each piece is real. Each piece does something.

The trouble starts when you try to draw one line from "here are my 4,000 tables" to "here is a score I trust."

Here's what that line costs. You stitch six tools together, and you write every rule, every threshold, and every constraint by hand, in five different syntaxes, in portals without git. You build the scorecard yourself, on a semantic model you maintain, fed by a preview feature. Then you keep all of it alive: the thresholds nobody has revisited page you every Sunday, the person who built the dashboard changes jobs, and Microsoft offers no guidance on which of the overlapping options to pick or which one gets deleted next. Ask anyone who built on Data Quality Services.



That's the pain. This post is the map. The scope is the four jobs a data team actually has to do: data profiling, data quality testing, data observability (monitoring freshness, volume, schema, and metrics over time), and data quality scorecards. For each job, what Microsoft recommends, what the alternatives are, and the exact point where each one stops helping.



Part 1: How to do data profiling

Profiling means computing statistics about what's in your columns. Distributions, nulls, distinct counts, min and max, patterns.

It goes first because everything after it depends on it. You can't write a sensible test until you know what's in the column, and you can't set a threshold until you know what normal looks like. It's also the step most teams skip.

What Microsoft recommends: Purview data profiling, which is step five of the Purview data quality lifecycle and the basis for your rules. Power Query profiling if you're a Power BI author. Data Wrangler if you live in a notebook. The SSIS Data Profiling Task if you're on-prem.

Tool	Manual or automatic	Overview
Purview data profiling	Manual trigger, AI column selection	Per-asset statistical snapshot on a random 1-million-row sample; 50 snapshots retained. Not supported for on-premises sources.
Power Query and Data Wrangler	Automatic in the editor	Live per-column statistics while a person has the tool open. First 1,000 rows by default in Power Query, and nothing is saved anywhere.
SSIS Data Profiling Task	Manual to configure, automatic to run	The only schedulable profiler here includes candidate keys and functional dependencies. SQL Server sources only, XML output, can't trigger anything.

1. Purview data profiling

In Unified Catalog, you pick a data asset and hit Profile. An AI recommendation engine preselects the columns it thinks matter, and you add or remove from there. When the job finishes, you browse a statistical snapshot per column: distribution, min, max, standard deviation, uniqueness, completeness, duplicates, and more.

This is the intended starting point. Profile first, read the results, then write rules that match what you found.

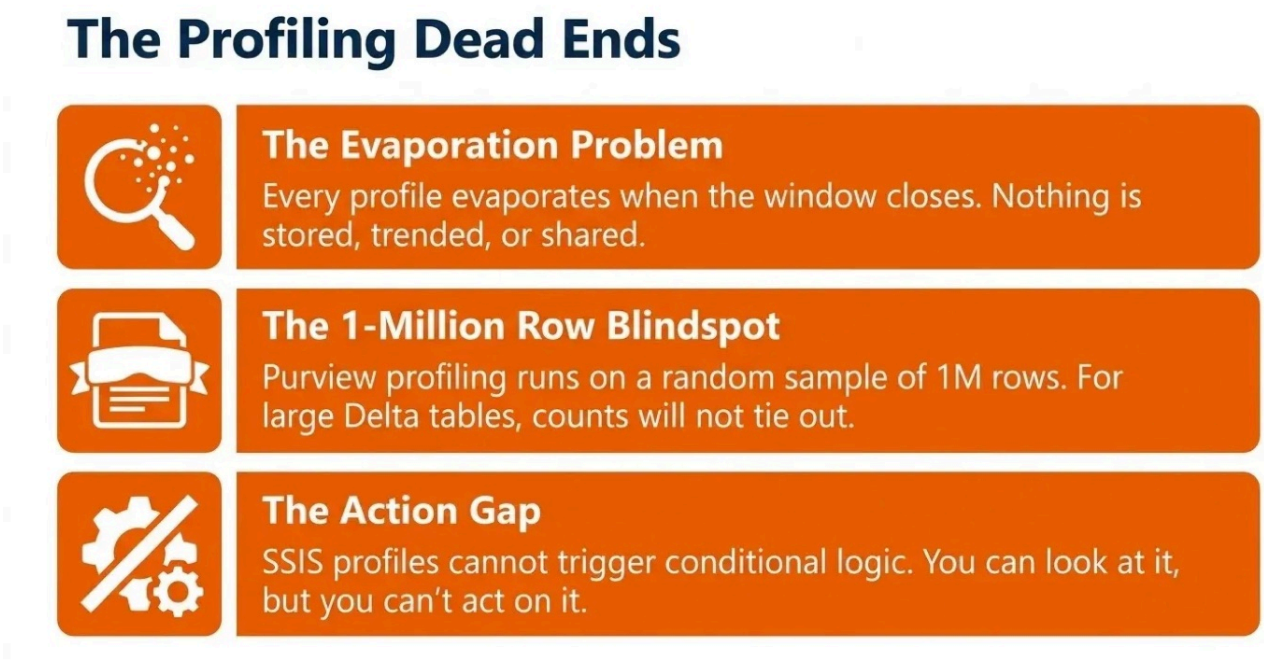
Where it stops. Profiling runs on a random sample of 1 million rows. If your Delta table is bigger than that, your counts won't tie out to the table, and the docs say so. Purview keeps 50 snapshots of profiling and assessment history, so your trend line has a horizon. Profiling isn't supported for on-premises sources at all. And it's a manual button in a portal, per asset, which is fine for 20 tables and absurd for 4,000.

Worth separating from the Data Map scan, which people confuse with profiling. That scan samples the first 128 rows to classify columns and pull the schema. It tells you a column looks like a Social Security number. It tells you nothing about how many of them are blank.

2. The interactive profilers: Power Query and Data Wrangler

Every Power BI author already has profiling and half of them don't know it. In the Power Query editor, the View ribbon has three data profiling tools. Column quality labels values valid, error, or empty. Column distribution shows a histogram plus distinct and unique counts. Column profile adds min, max, average, and standard deviation. Free, instant, and available in Power BI Desktop, Dataflow Gen2, and Excel. If you work in a Fabric notebook instead, Data Wrangler does the same job on a pandas or Spark DataFrame, and `display(df, summary = true)` gets you column statistics in one line.

Where they stop. Power Query profiles the first 1,000 rows by default. Switch to the full dataset and the editor rescans everything on every change, which on a few million rows is unusable, so most people leave it at 1,000 and quietly draw conclusions about a table they've seen 0.02 percent of. And for all of these, close the window and the profile is gone. Nothing is stored, trended, or shared.

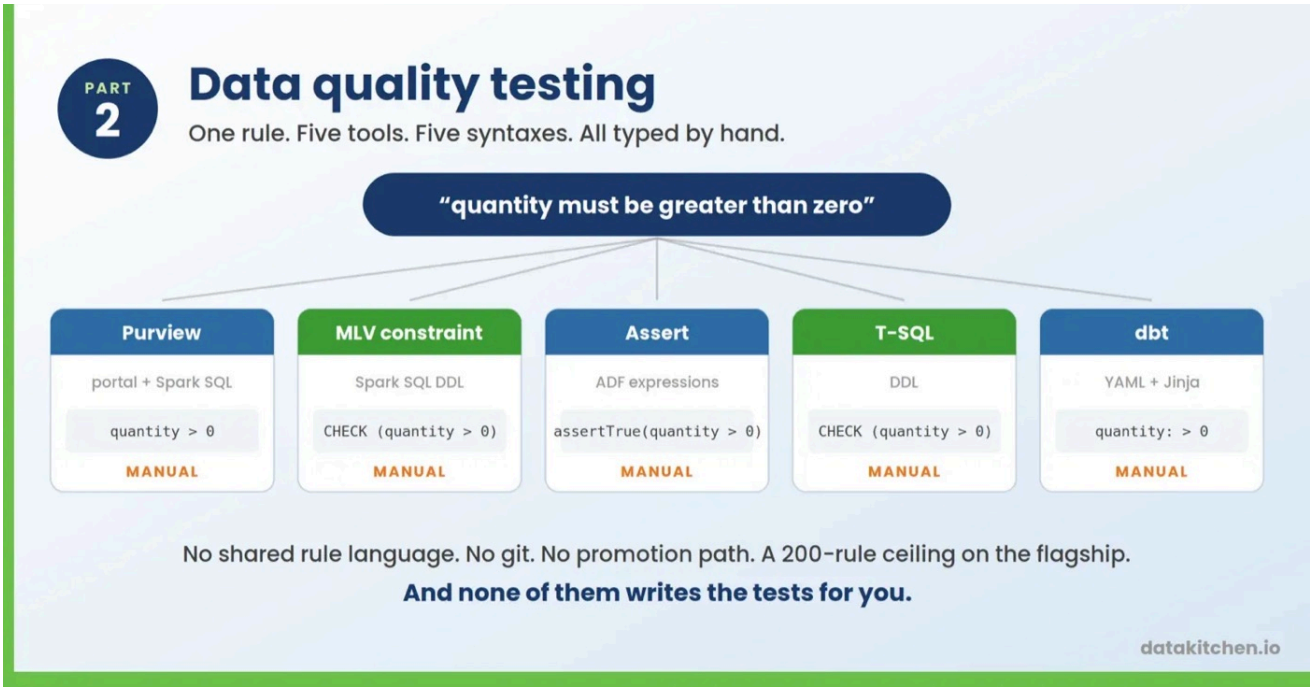


3. The SSIS Data Profiling Task

If you're on-prem and you've had SSIS since 2008, you already own this. Drop the Data Profiling Task into a package and it computes column length distributions, value distributions, null ratios, and pattern profiles. It also does two things most modern profilers skip: candidate key detection and functional dependency analysis, which tell you whether a column really is unique and whether one column actually determines another. It's the only scheduled, repeatable profiler on this list.

Where it stops. It only works with data stored in SQL Server. Not files, not third-party sources. The output is XML that you open in a separate standalone viewer. And here's the line from the docs that ends the conversation: the Data Profiling Task has no built-in features that let you use conditional logic to connect it to downstream tasks based on the profile output.

You can look at the profile. You can't act on it.



Part 2: How to do data quality testing

A test evaluates a rule against your rows and tells you which ones failed. Everything in this part does that. Watching the data change over time is the next part, and cleansing tools and catalogs are somebody else's post.

One distinction runs through all of it. A row-level test examines a single row and determines whether it's valid. An aggregate test looks at the whole set and asks a question about the set: is the null rate under 20 percent, are there at least two distinct payment types, is the average fare inside a sane range?

Most of what Microsoft ships is row-level only. That isn't a footnote. It's the reason most of the options below physically cannot express the checks you want most.

What Microsoft recommends: Microsoft Purview Unified Catalog data quality if you want governance and scores. Materialized lake views if you're in Fabric and want the pipeline to stop. A dbt job if your team already writes SQL and wants tests in git. T-SQL constraints if your warehouse is still on SQL Server. You'll probably need more than one.

Tool	Manual or automatic	Overview
Purview Unified Catalog data quality	Manual rules, automatic scans	Portal-authored rules across six quality dimensions, scored and rolled up to the governance domain. Suggested rules propose some for you.

Tool	Manual or automatic	Overview
Fabric materialized lake view constraints	Manual rules, automatic enforcement	CHECK constraints in the view definition that drop rows or fail the refresh. Fabric only.
ADF and Synapse Assert	Manual	Row assertions built by hand in the mapping data flow canvas. Now in public preview inside Fabric Dataflow Gen2.
T-SQL constraints	Manual to declare, automatic to enforce	NOT NULL, CHECK, UNIQUE, and foreign keys rejecting bad rows at write time. Free and on-prem, and not enforced in Fabric.
dbt job in Fabric	Manual	dbt Core running natively in a Fabric workspace. Tests live in git next to the models. Still preview.

1. Microsoft Purview Unified Catalog data quality

This is the flagship, and it's the one most people mean when they say "we use Purview for data quality."

Getting to your first test takes eight steps. You assign steward permissions, register and scan the source in Data Map, add the asset to a data product, set up a separate data source connection, run profiling, create rules, run a data quality scan, then review the results. The connection for data quality is not the same connection you used for metadata scanning. You set both up.

Once you're in, the out-of-the-box rules cover freshness, unique values, string format match, data type match, duplicate rows, empty and blank fields, table lookup, and custom. Format match handles enumerations, LIKE patterns, and regex.

Rules map to six quality dimensions: accuracy, completeness, conformity, consistency, freshness, and uniqueness. That's the vocabulary your governance team already speaks, and it's a big part of why Purview wins the room before anybody looks at the SQL.

Custom rules are where the real work happens. Every custom rule has three parts. A row expression that returns true when the row passes. An optional filter expression that narrows which rows get evaluated. A null expression that decides what happens when the value is missing. You write those in one of three languages: regular expressions, the Azure Data Factory data flow expression language, or Spark SQL.

The Spark SQL option is the good one. You can write real predicates, including subqueries and window functions. A null-rate ceiling looks like `(SELECT avg(CASE WHEN fareAmount IS NULL THEN 1 ELSE 0 END) FROM yourtable) < 0.20`. A uniqueness check looks like `COUNT(1) OVER (PARTITION BY vendorID) = 1`. If you write SQL for a living, this feels normal.

One footgun from Microsoft's own docs, and it's a nasty one. Write a correlated subquery with an unqualified column and Spark resolves the outer reference to the inner scope, so `WHERE t.payment_type = payment_type` becomes `t.payment_type = t.payment_type`, always true, and

your per-group minimum silently becomes a global minimum. The rule runs green and the answer is wrong. Alias the inner columns.

Scoring divides passed records by passed plus failed plus miscast plus empty, where miscast means the value can't be converted to the declared type. Users can also mark rows as ignored, and ignored rows drop out of the denominator entirely. Sit with that for a second: exclude the rows that fail and the score goes up, with nothing on the scorecard saying so. Column scores roll up to the asset, then the data product, then the governance domain. Since spring you can set score thresholds at the rule level and the asset level instead of holding every column to the same bar. Failed rows can be published to a Fabric Lakehouse or ADLS Gen2 so somebody can actually go fix them.

Pricing runs on two meters. A governed asset is a table, file, or report you've linked to a data product or tagged as a critical data element, which is Purview's term for a field the business has actually agreed matters. Scanned assets sitting in the Data Map are free. Governed assets are billed per asset per day, and scans consume Data Governance Processing Units billed by the compute hour, so the bill scales with how many tables you govern and how often you check them. Microsoft is upfront that the metadata scan and the quality scan are separate engines, and you pay for both, because the metadata engine samples the first 128 rows, and the quality scan reads every row in the column.

Where it stops. Custom SQL rules can't do joins. The docs say it plainly: rules operate on a single dataset, and you can't join multiple tables. So no source-to-target reconciliation. No cross-system referential integrity. No "did the 4.2 million rows that left SAP arrive in the warehouse?" The one cross-table escape hatch is the table lookup rule, and the reference table must reside in the same governance domain.

You also hit a hard ceiling of 200 active rules per data asset. Go past it and the scan fails. The documented workaround is to add the same table to multiple data products or toggle rules on and off, which is a scheduling problem dressed up as a feature.

Do the arithmetic. Two hundred rules on a 300-column table is less than one rule per column. That's not a coverage strategy, and we've written about what real coverage actually takes.

File support is narrower than you'd expect. Data quality supports Delta, Parquet, Iceberg ORC, and Iceberg Avro. It does not support CSV, TSV, or text files. Column names with spaces aren't supported either.

Rules live in a UI. The data quality REST API is still preview. There's no git, no diff, no promotion from dev to prod, no code review.

And the scans run after your pipeline finishes. Purview tells you the gold table was wrong. It doesn't stop the gold table from being wrong.

2. Fabric materialized lake view constraints

Materialized lake views went generally available at FabCon in March 2026. You define a view in Spark SQL or PySpark, Fabric materializes it as a Delta table, tracks dependencies between views, and orders the refreshes for you.

You attach quality constraints to the view definition:

```
CREATE OR REPLACE MATERIALIZED LAKE VIEW silver.valid_orders (
  CONSTRAINT positive_quantity CHECK (quantity > 0) ON MISMATCH DROP,
  CONSTRAINT valid_date CHECK (orderDate >= '2020-01-01') ON MISMATCH FAIL
) AS SELECT * FROM bronze.orders
```

DROP removes the bad rows, logs the count, and keeps going. FAIL stops the refresh at the first violation, and FAIL is the default if you leave the clause off. If you mix them, FAIL wins. Violation counts land in the lineage view and in a data quality report, which Part 4 covers.

This is the closest thing Microsoft has to a real test that runs inside the pipeline.

Where it stops. You write every constraint by hand. There's nothing that looks at your data and suggests what to check. Constraints only fire during refresh, and only inside a materialized lake view, so the rest of your lakehouse is uncovered. It's Fabric only.

The bigger issue is ON MISMATCH DROP. It doesn't quarantine the row. It deletes it from the output and writes a number to a log. Your gold table now looks clean because the evidence left the building. Somebody has to be watching those counts, and nobody is watching those counts.

The harder question underneath this is how many tests each layer needs and where they belong. We worked through that in data quality test coverage in a medallion architecture.

3. The Assert transformation in ADF and Synapse

The old one, still shipping. Inside a mapping data flow you drop in an Assert transformation, pick expect true, expect unique, or expect exists, write an expression, and set a custom error message. Downstream you call `hasError()` and route the bad rows somewhere.

It works. People have run it for years.

Where it stops. It only exists inside a mapping data flow. If your transformation is a notebook or a stored procedure, there's nothing to attach it to. There's no results history, no trend, no score, no catalog. You get a run that failed and a log to read.

It does have a path into Fabric, which is newer than most write-ups reflect. Assert never got a native Dataflow Gen2 equivalent, but mapping data flow transforms hit public preview in June 2026 and bring the whole ADF canvas inside Dataflow Gen2. Assert is on the supported transformation list, described as a row modifier that defines assert rules for rows in the data stream. There's a migration experience that converts existing ADF and Synapse pipelines into MDF transforms.

Read the preview limits before you plan around it. MDF transforms only execute through a pipeline Dataflow activity, so you can't run one from Dataflow Gen2 directly and Save is the only option on the Save and run menu. Flowlets, the Data Flow Library, and user-defined functions aren't supported. Execution still uses the underlying Synapse Spark runtime. And Microsoft says plainly that not all Mapping Data Flow capabilities made the preview.

4. T-SQL constraints in SQL Server and Azure SQL

The oldest data quality test in the building, and the only one on this list that's free, on-prem, and enforced at write time. NOT NULL, CHECK, UNIQUE, and primary and foreign key constraints reject bad rows at the door. A foreign key is a referential integrity constraint that actually holds, which is the exact test Purview custom SQL rules can't write.

If your warehouse runs on SQL Server or Azure SQL, you already have this, and you should use more of it.

Where it stops. It rejects the row instead of recording the problem, so the failure shows up as a load error at 2am rather than a quality score. It only protects the table you defined it on. It won't tell you a column went from 2 percent null to 40 percent null, because both are legal. And check whether yours are actually on: a constraint added `WITH NOCHECK` never validates existing rows and the optimizer stops trusting it, and half the estates that say "we have foreign keys" have untrusted ones.

And it doesn't come with you to Fabric. In a Fabric Warehouse, table constraints are hints, not rules. PRIMARY KEY and UNIQUE are only supported with NONCLUSTERED and NOT ENFORCED, and FOREIGN KEY is always NOT ENFORCED. Insert the same key four times and all four rows land. In a

Fabric Lakehouse you can't declare them at all. The error reads "Table constraints (Primary and Foreign key) are not supported in V2 Datasource tables."

So the one enforced test you've relied on for your whole career is the one you lose on the way to the lakehouse.

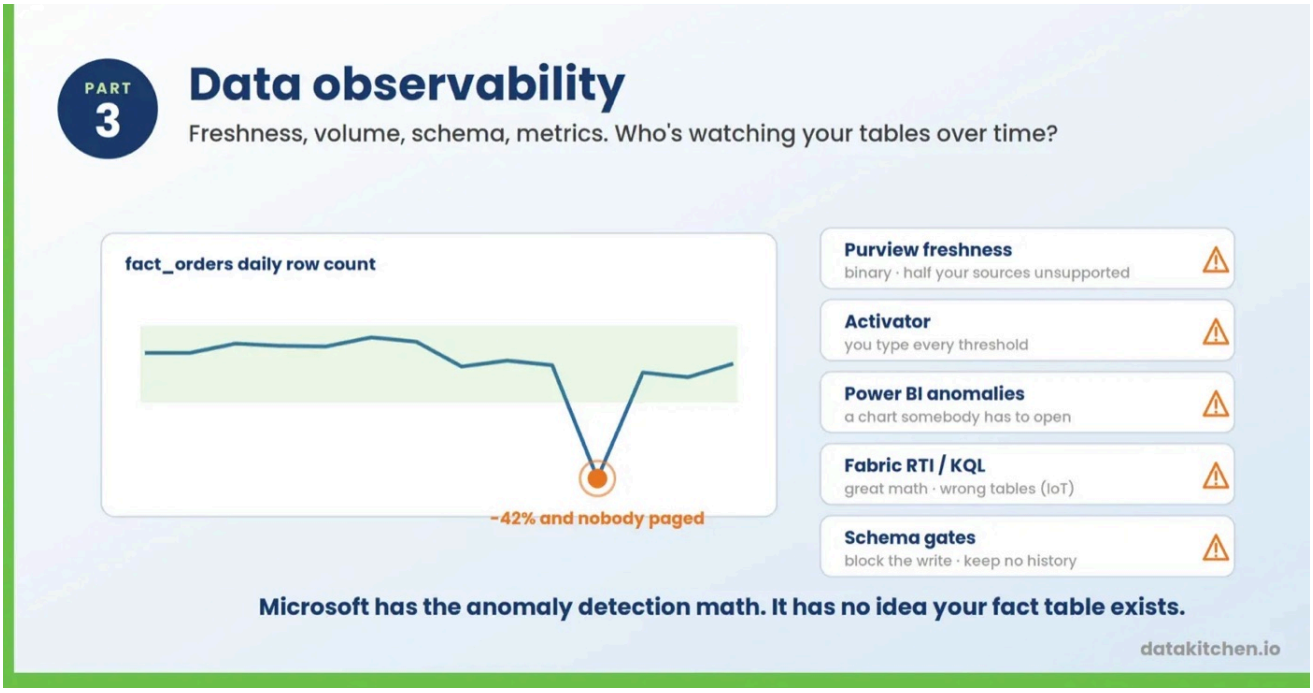
5. dbt job in Fabric

For a lot of Microsoft shops, this is already the real answer, and it's now a first-class Fabric item. Announced at Ignite 2025 and rolled out that December, dbt job runs dbt Core inside a Fabric workspace as a managed item type. No dbt Cloud account, no self-hosted Airflow, no local install. You author models, define dependencies, and run tests in one place, with Fabric handling scheduling and monitoring and run logs landing in OneLake.

The reason engineers reach for it over everything else in this part: tests live in git next to the models, and `dbt build` interleaves models and tests so a failed test stops the model downstream of it. That's version control, code review, and a pipeline gate in one tool, which is three things Purview doesn't have.

Where it stops. Still preview. Runtime v1.0 covers dbt Core 1.9 across four adapters: Fabric Warehouse, Azure SQL Database, PostgreSQL, and Snowflake. Some partner adapters aren't supported. There's no build caching, so every run compiles fresh from source.

The coverage problem is the same as everywhere else, just in YAML instead of a portal. dbt ships four built-in generic tests: unique, not_null, accepted_values, and relationships. Everything past that you write yourself, for every column, forever. Point it at 4,000 tables and you have 4,000 tables and no tests.



Part 3: How to do data observability

Different job from testing, and the line is worth stating precisely. Tests verify data content rules: column formats, value ranges, referential integrity. Monitors track table-level patterns over time: whether the table is updating on schedule, whether row counts are moving unexpectedly, whether the schema shifted, whether a metric is still in bounds.

Four categories. Freshness, volume, schema, and metric.

The distinction that matters is where the expected range comes from. A test compares your data to a number you typed. A monitor learns from historical patterns and flags the deviation on its own. Nobody has to know in advance that Tuesday's load is normally 4.2 million rows.

These are the failures that catch you sideways. A pipeline that quietly stopped running. A table that doubled overnight. A column that got dropped without notice. None of them fail a test. All of them break a dashboard.

What Microsoft recommends: nothing, because there is no Microsoft data observability product. There's one freshness rule, a rules engine where you set the thresholds yourself, a chart feature, a streaming stack built for IoT telemetry, and two schema gates that block bad writes without recording what changed. You assemble it.

Tool	Manual or automatic	Overview
Purview freshness rule	Manual	The only native freshness check. Binary pass or fail at the asset level, and unsupported on several major sources.
Fabric Activator	Manual thresholds, automatic firing	Stateful rules including heartbeat detection for data that stopped arriving. You supply every threshold.
Power BI anomaly detection	Automatic	Learned expected range on a line chart, with ranked explanations. A visual, not a monitor.
Fabric Real-Time Intelligence anomaly detection	Automatic	The real thing: learned baselines, continuous monitoring, notifications. Requires an Eventhouse and a KQL database.
ADF schema drift and Delta enforcement	Automatic gates	One tolerates schema changes silently, one rejects the mismatched write. Neither records what changed.

1. The Purview freshness rule

Freshness is one of Purview's six dimensions and it gets one rule. You pick a last modified date column and set the expected window.

Where it stops. The score is 100 or 0. There's no "arrived four hours late," just pass or fail. It applies at the asset level rather than the column level. And it isn't supported for Snowflake, Azure Databricks Unity Catalog, Google BigQuery, Synapse, or Azure SQL, which covers most of what a large shop actually runs.

Two other Purview features get pointed at this problem and neither one qualifies. The 50 retained profiling snapshots are stored results, not a monitor, and nothing compares them for you. Data quality actions fire on outliers within a profiled column and on high null counts, which is a statistic about the values in a single scan, not a comparison against last week.

So Purview covers one of the four categories, partially. Nothing in it watches row counts. Nothing in it tells you a column appeared or disappeared.

2. Fabric Activator

Activator is the closest thing Microsoft has to a monitoring layer you'd point at operational data. Rules are stateful, which is what makes them interesting here. BECOMES, INCREASES, DECREASES, and EXIT RANGE all compare a value to its previous state rather than evaluating one row in isolation.

The heartbeat rule is a genuine freshness monitor. It fires on the absence of data, so a stream that stops sends you a message. SQL query rules, in preview since March 2026, run a query against Fabric Data Warehouse on a schedule, which is how you'd build a row count monitor if you were going to build one.

Where it stops. You supply every threshold. Activator will tell you when row count drops below 4 million because you told it 4 million. It doesn't know that 4.2 million is normal for a Tuesday, and it will page you every Sunday when the number legitimately halves. That's the difference between alerting and observability, and it's the whole difference.

3. Power BI anomaly detection

Add Find Anomalies from the Analytics pane on a line chart and Power BI marks the points that fall outside an expected range it works out from the series itself. Sensitivity is adjustable. It also explains the anomalies, running an analysis across fields in your model and ranking the contributing dimensions by explanatory strength.

That explanation feature is better than it sounds. Pointing at the seller and city responsible for a spike is real root cause work, and no other Microsoft tool on this list does it.

Where it stops. It's a chart feature. Line charts only, time series on the axis, minimum four data points, no legends or multiple values, no drill-down. It doesn't run on a schedule, doesn't store results, and doesn't page anybody. Somebody has to open the report and look at it, which means it catches the anomaly right after the meeting where the wrong number was presented.

4. Fabric Real-Time Intelligence

Fabric Real-Time Intelligence is where Microsoft's actual anomaly detection lives, and almost nobody evaluating data quality tooling finds it, because it sits in the streaming stack.

KQL ships `series_decompose_anomalies()` natively, which runs univariate anomaly detection across thousands of time series in seconds. Pair it with `make-series` and you have volume monitoring in one query. There's multivariate anomaly detection too, for when the combination of values is wrong even though each one looks fine on its own.

On top of that sits a newer anomaly detection feature in preview that runs natively on Eventhouse tables without copying data. You get interactive exploration with adjustable model sensitivity, continuous monitoring with automated notifications, and reanalysis as new data arrives. Results are published to the Real-Time Hub, where you can alert on them or hand them to a Fabric data agent.

Where it stops. Your data has to be in an Eventhouse with a KQL database and the Python plugin enabled. This was built for IoT telemetry and fraud detection, not for the 4,000 tables in your warehouse. Getting a row count history for a Delta table into an Eventhouse so you can run anomaly detection on it is a pipeline you write and maintain.

Worth knowing if you built on the old Azure AI Anomaly Detector: it retires on 1 October 2026, and you haven't been able to create new resources since September 2023. Microsoft's own migration guidance points to Fabric, which integrates the same open-source anomaly detector.

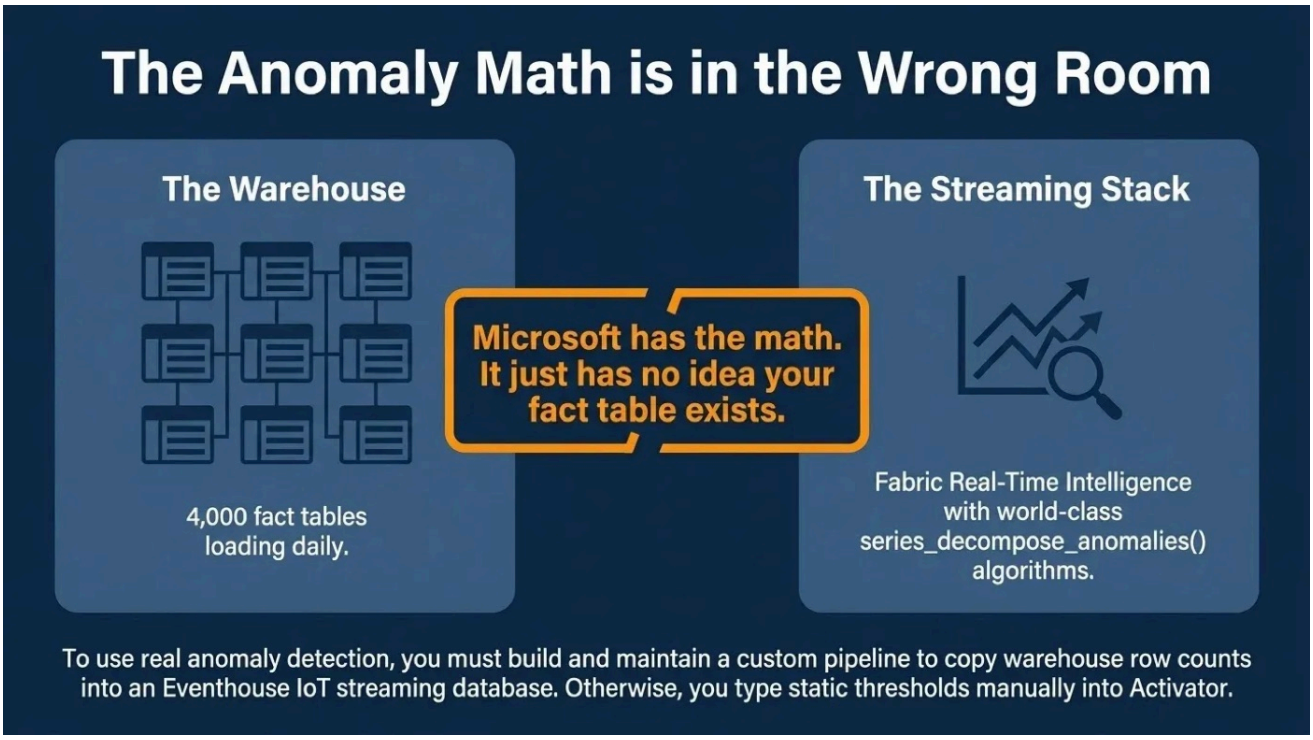
5. Schema: two gates, no history

Microsoft gives you two ways to react to a schema change and no way to see one.

In ADF and Synapse mapping data flows, the schema drift feature exists to make schema changes not break your pipeline. Allow schema drift and new columns flow through to the sink silently. The one setting that tests schema, Validate schema on the source, fails the run rather than recording what changed.

In a lakehouse, Delta enforces schema on write by default, rejecting a write that doesn't match the table. Real protection, free, zero configuration. Then most ETL pipelines opt into `mergeSchema` for additive changes, and from that moment new columns arrive silently, which is the exact event you wanted to be told about. Renamed and dropped columns aren't covered by auto-evolution at all, so they break downstream consumers with no warning.

Where both stop. One tolerates the change, one blocks the write, and neither keeps a history. The Delta transaction log records every schema change, but nothing reads it for you and turns it into an alert. Nothing anywhere in the stack tells you what changed between Monday and Tuesday.

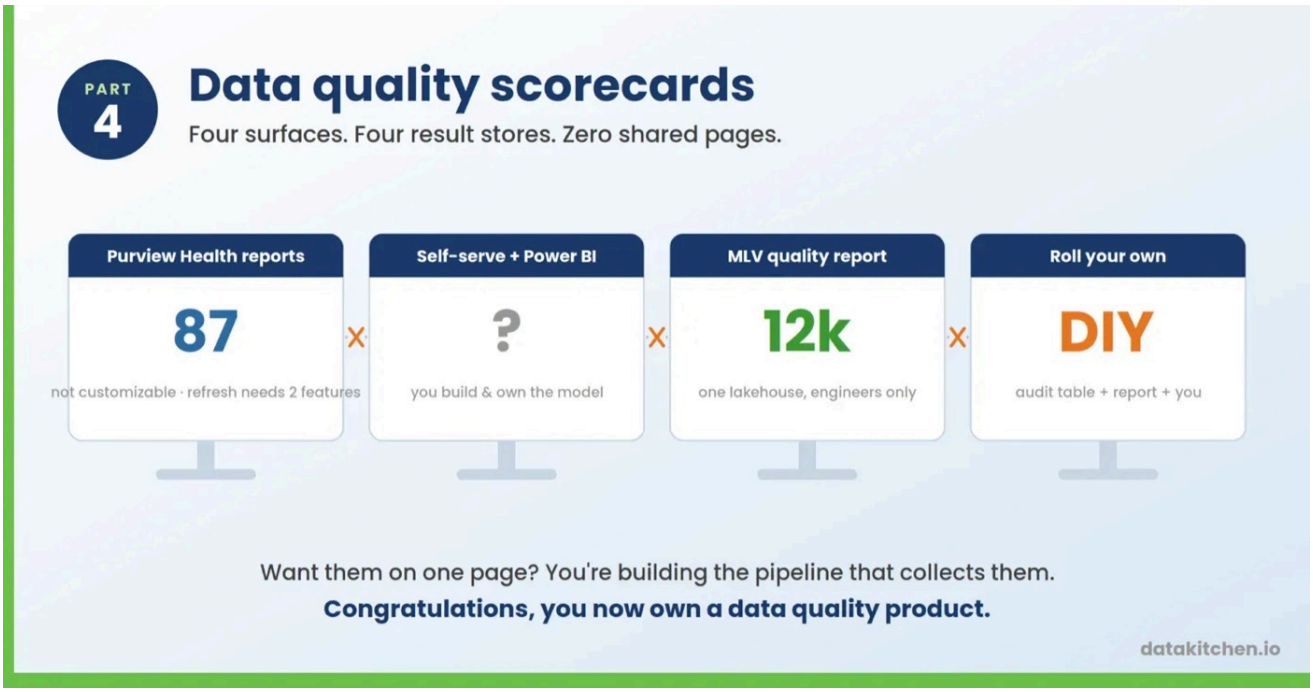


Where the whole observability story stops

Line up the four things this part is supposed to cover.

Freshness: one binary rule in Purview that doesn't work on half your sources, plus a heartbeat in Activator if your data is streaming. Volume: nothing native, unless you write a SQL query rule with a threshold you picked or move row counts into an Eventhouse. Schema: two gates that block a bad write and no way to see the history of changes. Metric anomalies: genuinely good algorithms, in a chart nobody watches and a streaming engine your warehouse tables aren't in.

The through line is that Microsoft has the math and hasn't pointed it at your tables. `series_decompose_anomalies()` would tell you your fact table loaded 40 percent lighter. It just has no idea your fact table exists.



Part 4: How to build data quality scorecards

A scorecard shows scores and trends to somebody who didn't write the tests or run the monitors. That's the bar for this part. The remediation queue and the alerting layer sit at the end, labeled as what they are.

What Microsoft recommends: the built-in Health management reports in Purview for the standard view, and self-serve analytics in OneLake plus Power BI when you want your own.

Tool	Manual or automatic	Overview
Purview Health Management reports	Automatic	Built-in scorecard with dimension scores and per-rule trend history. Not customizable, and it won't refresh without health controls plus self-serve analytics.
Self-serve analytics into OneLake, then Power BI	Manual	Purview publishes governance metadata to a lakehouse and you build the semantic model and report. Still in preview.
Materialized lake view data quality report	Automatic	Fabric charts violation counts by constraint and by view over time. One lakehouse, engineering audience.
Roll your own on audit tables	Manual	Write results to your own table and point Power BI, Power BI Metrics, or SSRS at it. The only way to get more than one engine on a page.

Tool	Manual or automatic	Overview
Purview data quality actions	Automatic	A severity-ranked work queue generated from outliers and null counts. Not a dashboard, but the closest thing to somebody fixing something.
Fabric Activator	Manual rules, automatic firing	Alerting, not a dashboard. Rules fire emails, Teams messages, or Fabric jobs when a condition hits.

1. Purview Health Management reports

Purview ships built-in reports under Health management. The data quality health report shows dimension scores and rule results across the estate. Scores roll up from column to asset to data product to governance domain, and every rule has a History tab with the trend across scan runs.

One detail that tells you something about how integrated this all is: Microsoft's own page for the report says it helps you understand accuracy, completeness, consistency, reliability, and timeliness. The rules feeding it use accuracy, completeness, conformity, consistency, freshness, and uniqueness. Reliability and timeliness aren't rule dimensions, and conformity and uniqueness aren't in the report's description. The scorecard and the rules that populate it don't use the same vocabulary.

This is the real scorecard, and it's good.

Where it stops. The out-of-the-box reports aren't customizable. And the data quality health report won't refresh unless you're running data health controls and you've subscribed your Unified Catalog metadata to self-serve analytics.

Data health controls are a separate feature from data quality rules, and people conflate them constantly. Controls score your governance posture, things like whether assets have owners, descriptions, and classifications. They run on their own schedule and consume their own DGPUs. Your data quality dashboard depends on a governance feature you might not have turned on, and if you skip it you get a stale report that doesn't tell you it's stale.

2. Self-serve analytics into OneLake, then Power BI

For anything custom, you use self-serve analytics, still in preview. Purview publishes a third normal form model of your governance metadata into a Fabric Lakehouse or ADLS Gen2. You get governance domains, data products, data assets, glossary terms, data quality rules, dimensions, the pass and fail counts behind every score, and the health actions somebody opened to fix them.

From there, it's a Fabric semantic model and a Power BI report you design and maintain yourself. Point the error records feed at the same lakehouse, and your dashboard can drill from a score down to the rows that failed.

That's a real dashboard, and it's a project. Budget for a data model, a semantic layer, and somebody who owns the refresh.

3. The materialized lake view data quality report

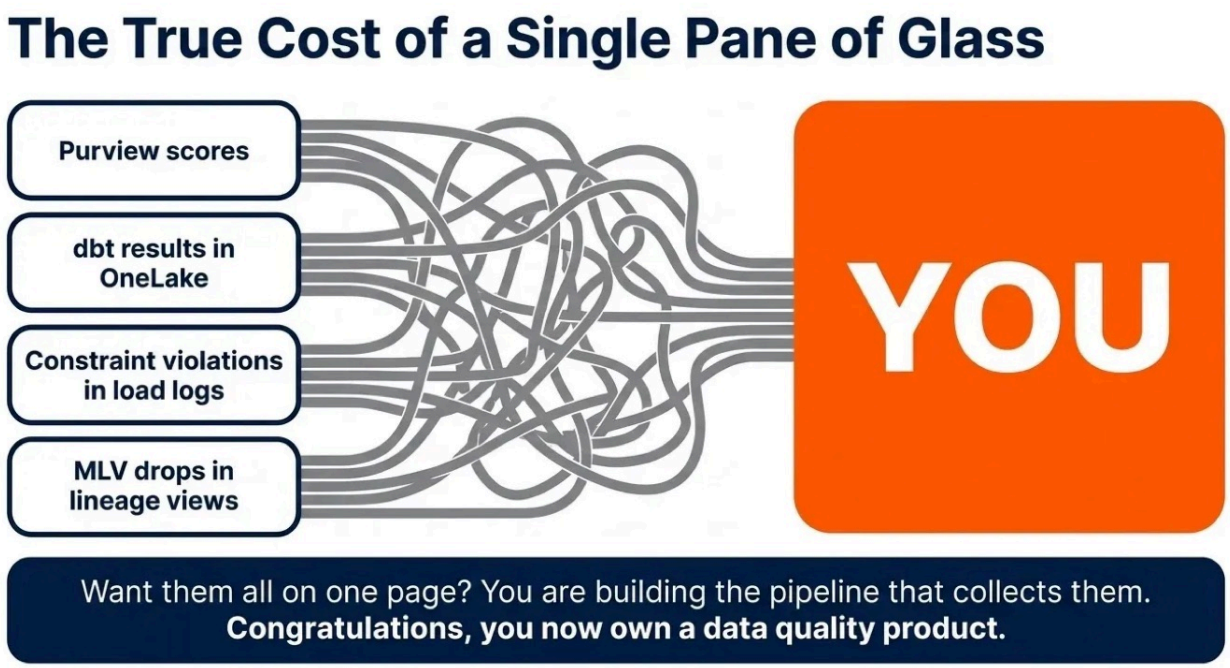
Fabric keeps its own quality reporting for materialized lake views, separate from Purview. The lineage view shows dropped row counts per constraint, and the data quality report shows counts by constraint, by view, over time. A rule that normally drops a tenth of a percent and suddenly drops fifteen shows up as a spike you can see.

Where it stops. It covers materialized lake views in that lakehouse and nothing else. It doesn't roll up to a data product, a domain, or a business owner. Nobody outside the engineering team opens it.

4. Roll your own on audit tables

Plenty of teams skip all of the above, write test results to their own audit table, and point Power BI at it. It's the only way to get results from more than one engine onto the same page. Power BI Metrics fits here for SLA-shaped tracking, and SSRS does the job on-prem.

Where it stops. You designed it, you built it, you own it, and the person who wrote it is going to change jobs.



And the alerting layer, which isn't a dashboard

Two more things get pointed at this job. Activator fires emails, Teams messages, or Fabric jobs when a condition you defined is met, and Microsoft's own docs frame its data quality role as reacting to problems found elsewhere. Purview data quality actions auto-generate a severity-ranked work queue from outliers and null counts and roll it up to the governance domain. The first is a pager; the second is a to-do list. Both are useful. Neither shows a score to a stakeholder.

Where the whole scorecard story stops. Four surfaces produce data quality results, and each one keeps its own. Purview scores live in Purview. Materialized lake view violations live in the Fabric lineage view. dbt test results live in run artifacts in OneLake. Constraint violations are a load error in a job log. Want them on one page? You're building the pipeline that collects them, and now you own a data quality product.

The category Microsoft exited: cleansing, matching, and mastering

Everything above measures. Nothing above fixes.

For fifteen years, the fixing was Microsoft's job too. Data Quality Services did cleansing, matching, standardization, and de-duplication against a knowledge base, with reference data from external providers and profiling built into the tasks. Master Data Services did mastering, with business rules that validated members before a version could be committed.

Between them, they covered a whole category: cleansing, fuzzy matching, survivorship, golden records, reference data. Nothing in the current Microsoft stack does any of it.

Both were removed in SQL Server 2025. Not deprecated with a runway. Removed. They're still supported in SQL Server 2022 and earlier, and if you're running DQS today, you have to uninstall it before you can even upgrade.

If you built on DQS, the migration guidance points to Purview, which measures quality and cannot cleanse, match, or master anything. That isn't a like-for-like replacement. It's a different product for a different job.

Worth being straight about this: TestGen doesn't remediate either. Measuring and fixing are separate problems and require separate tools. The point isn't that Microsoft lacks a remediation story. It's that Microsoft had one, deleted it, and is now pointing displaced customers at a measurement tool.

The on-prem problem nobody puts on the slide

Most of this post assumes you're in Fabric or Azure. A lot of Microsoft data isn't.

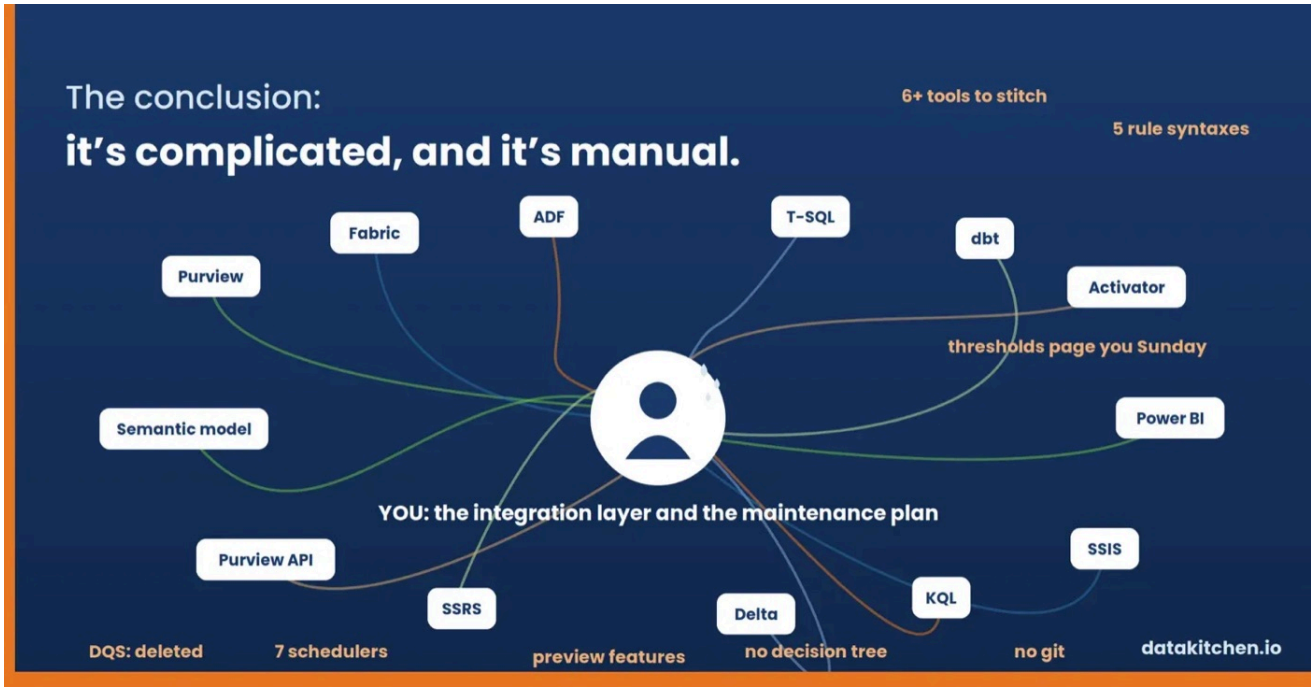
You've got Azure SQL Database and Azure Synapse Analytics in the cloud. You've also got Microsoft SQL Server and a stack of SQL Server 2019 boxes in a rack, holding the data the business actually runs on. SQL Server 2025 adds EXPOSE TO FABRIC and OneLake is coming for more of this, but "coming" isn't a test plan for the quarter you're in.

Here's the list from the rack. Materialized lake views need Fabric. Activator needs Fabric. Real-Time Intelligence anomaly detection needs an Eventhouse. Delta schema enforcement needs Delta tables. The dbt job needs a Fabric workspace, though dbt Core on your own machine has always worked against SQL Server. Data Wrangler needs a Fabric notebook. DQS and MDS are gone. T-SQL constraints work and always have, but they reject rows rather than score them. The SSIS Data Profiling Task profiles but can't trigger anything. Nothing on the list watches whether last night's load arrived.

That leaves Purview. Purview data quality does support on-premises SQL Server and Oracle, and here's what that takes: you stand up a Kubernetes cluster to host the scanning runtime, because the data stays on your premises. Profiling isn't supported for on-prem sources at all. The ADF-style custom expressions don't work there either, only custom SQL. The freshness rule isn't supported for SQL Server.

So the answer to "how do I test the data in my on-prem warehouse" is: run a Kubernetes cluster, skip profiling, and don't ask about freshness.

Meanwhile, the Purview scan engine reads your rows into its own managed Spark service. For many pharma and financial services teams, that conversation with security ends the evaluation.



The conclusion: it's complicated, and it's manual

There is no Microsoft data quality product. There are nineteen rows across the four tables in this post, and Activator appears in two of them because it does two different jobs. Doing all four jobs means stitching six or seven of those rows together.

Here's the realistic stack. Purview for profiling and scores. Materialized lake view constraints if you want the pipeline to stop. The Assert transformation for anything running through a mapping data flow. T-SQL constraints on whatever is still in SQL Server. Activator for alerts. A Power BI report you design yourself, on a semantic model you build, over metadata from a preview feature you subscribed to. Six tools, four jobs, no shared rule language, no shared scheduler, no shared results store.

Then look at the middle column of those tables. The word manual appears in twelve of the nineteen rows. Every quality rule, every threshold, every constraint, every semantic model is something a person writes and then owns forever. The person who built the scorecard changed jobs. The thresholds nobody has revisited since the pipeline was rewritten keep firing every Sunday.

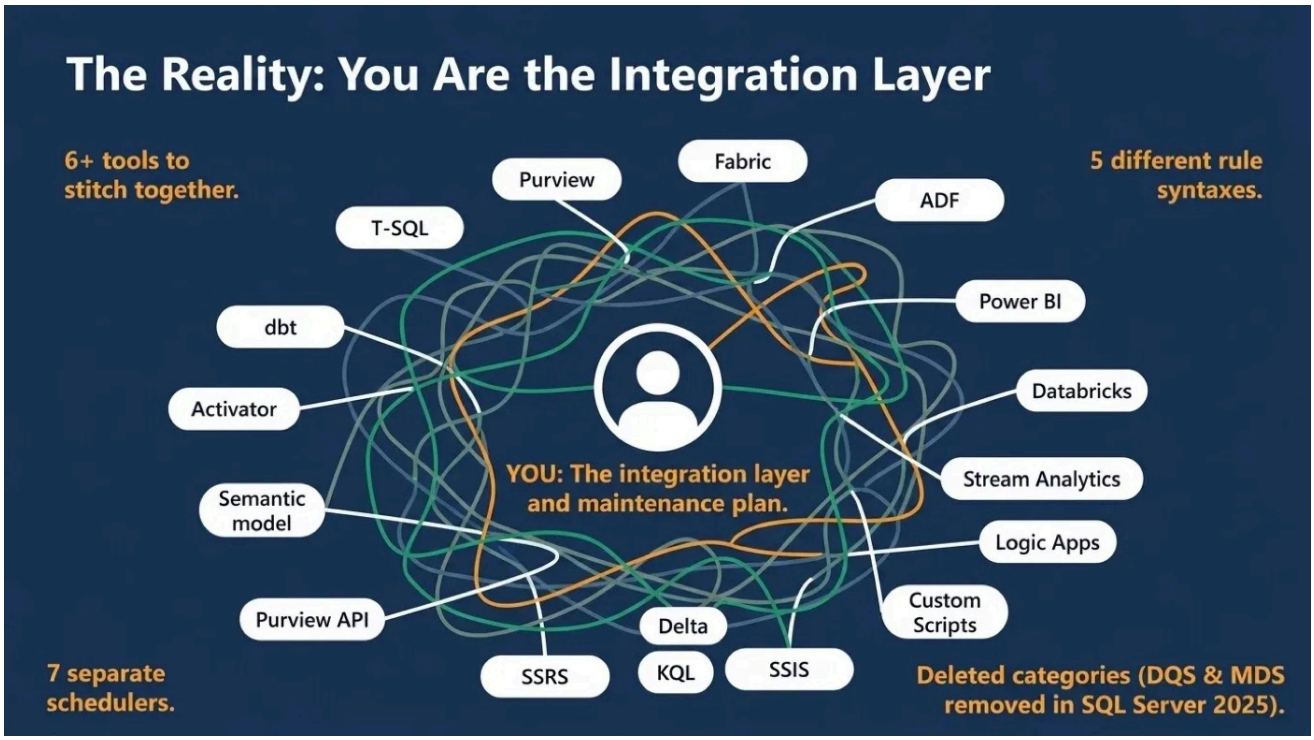
That’s before the overlap. There are five different ways to assert that a row is valid, and they use five different syntaxes. Four different ways to profile a column, and none of them store a result that the others can read. Two gates that block a bad schema and none to show you its history. Two genuinely good anomaly detection engines, neither of which knows your warehouse tables exist. Four paths to a scorecard, one of which is “build it yourself.”

Then there’s the question nobody answers on a slide, which is what runs all this and when. Purview quality scans are scheduled inside Purview, bill their own compute, and reread every row in the column each time, while constraints evaluate only what’s being written. Materialized lake views run on their own schedules. The dbt job is scheduled in Data Factory. Assert runs whenever its data flow runs. Activator evaluates continuously. Power Query profiling runs when a person has the editor open. Six or seven schedulers, no shared calendar, and no single place to see whether last night’s quality checks actually ran.

Microsoft ships all of it and tells you nothing about which one to pick. There is no decision tree. There is no page that says use this for a lakehouse, use that for on-prem, and here’s what to do when your estate is both.

And nothing tells you what’s about to disappear. Ask the teams who built on Data Quality Services and Master Data Services, both removed in SQL Server 2025 with no deprecation runway. Or the ones on Azure AI Anomaly Detector, which retires on 1 October 2026 and stopped accepting new resources in September 2023. Synapse Link is discontinued in SQL Server 2025 and you’re pointed at Fabric mirroring. Purview access policies are discontinued there too. The Assert transformation never got a native Dataflow Gen2 equivalent and instead got a preview migration path six years after it shipped.

The pattern isn’t that Microsoft ships bad tools. Purview’s scoring model is good. `series_decompose_anomalies()` is good. Materialized lake view constraints are the right idea. The pattern is that you’re the integration layer, and you’re also the maintenance plan.



There's also a dimension nobody can cover anymore. Purview scores across accuracy, completeness, conformity, consistency, freshness, and uniqueness. Uniqueness is well served. Completeness and conformity are fine. Freshness is one binary rule that doesn't work on half the sources you own.

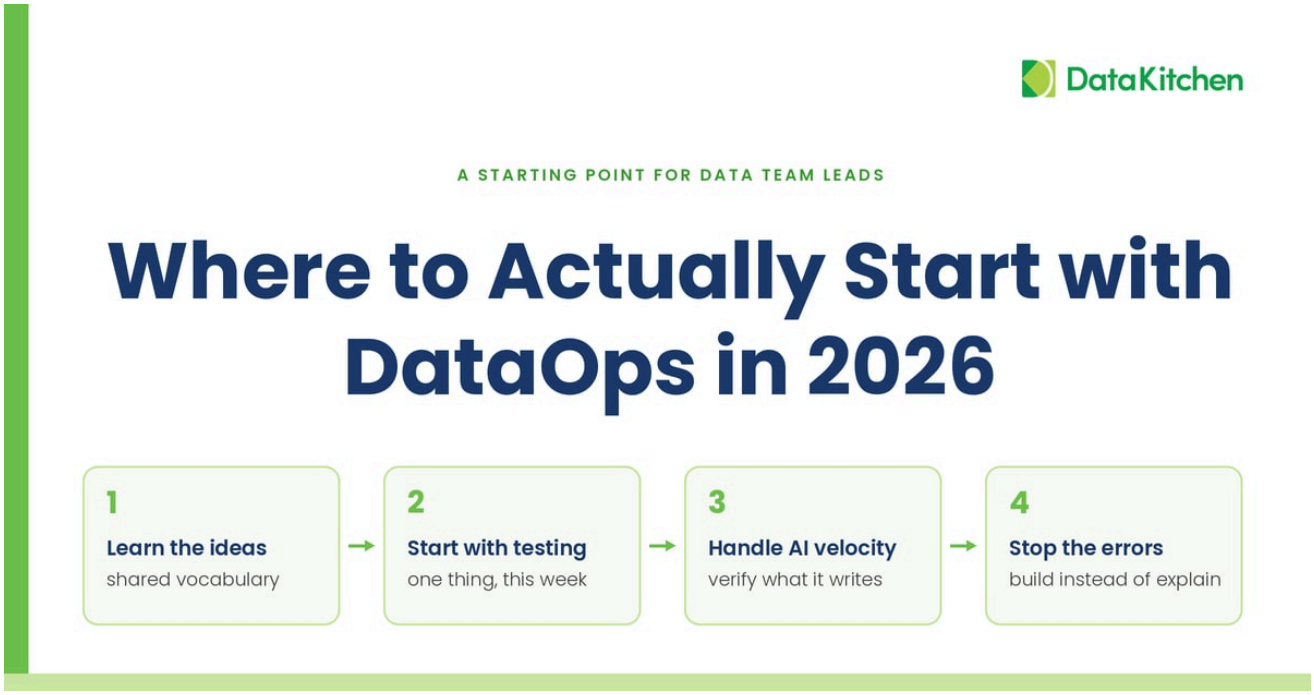
Accuracy is the interesting one, because accuracy means comparing your data against a source of truth. That was the job of DQS knowledge bases and reference data. With those deleted, Microsoft ships a scorecard with an accuracy dimension on it and no tool that can honestly populate it.

One more thing, and it's the one that matters most. A score is only worth as much as the tests underneath it. If a fraction of your tables have rules, your domain score measures the tables somebody had time to configure. It's precise and it's meaningless. Nothing on this list writes tests at scale or connects profiling to testing automatically. Scoring, monitoring, production tripwires, and deployment regression testing are four separate jobs. Each Microsoft option serves one or two.

REFERENCE AND TOOLING

Where to Actually Start with DataOps in 2026

Read the original article at: https://datakitchen.io/blog/where-to-actually-start-with-dataops-in-2026/



You still write code, and now you're also answerable for what three other people ship. Monday opens with a Slack thread about a number that moved. You put two of your reports on it and lose three days. The sprint work slips, and the engineer you brought on to build models spends her week doing forensics on a schema change nobody announced. And your customers want more, and more, and more.

You didn't take the job to run an incident desk. Neither did they. So the question isn't whether your team needs DataOps. The question is narrower and more annoying: what do you do first, what do you hand your reports to read, and in what order?

We've been pushing DataOps for over a dozen years. We started DataKitchen in 2013, wrote the DataOps Manifesto, and led this movement from the beginning, back when explaining the idea took 20 minutes and half the room thought you meant DevOps with a typo. Everything below is what we've learned and published since.

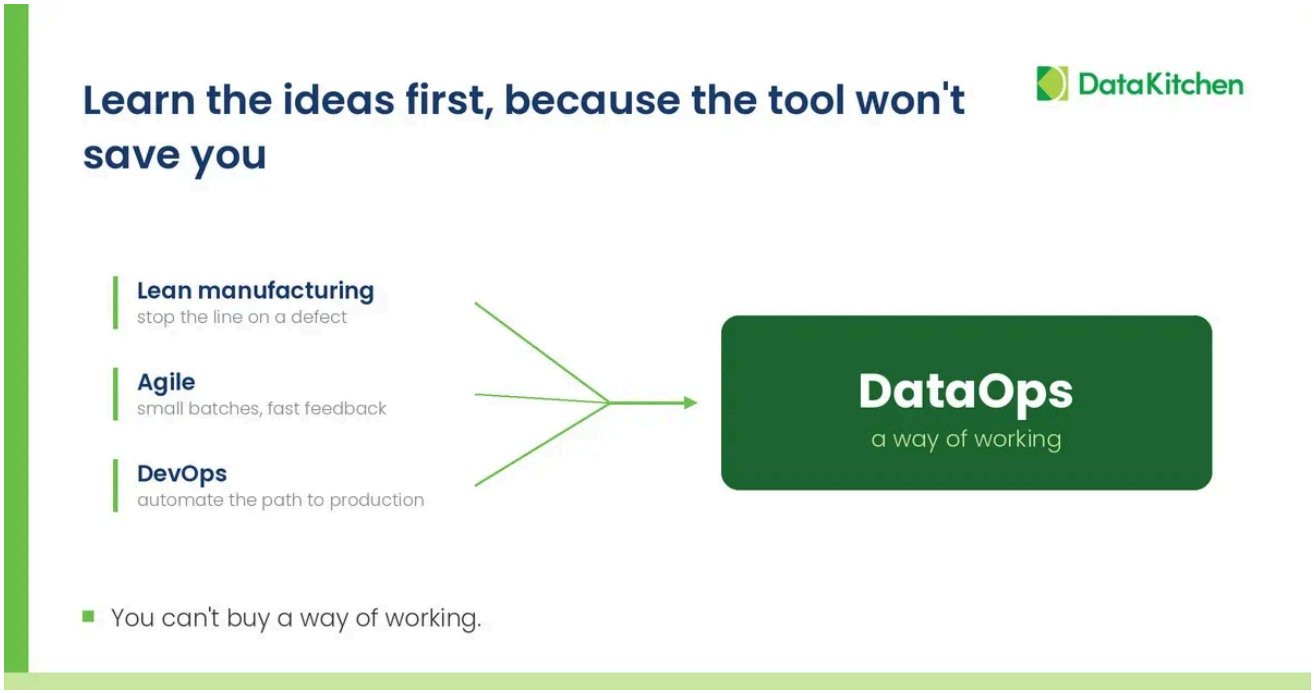
This is that answer. Four sections. The first one gives you and your reports a shared set of ideas. The second gives you one thing your team can do this week without asking anybody for budget. The third deals with AI, because your reports now generate pipeline code faster than you can review it, and you are the reviewer. The last one is for whoever on your team came from software and can't figure out why everything is so weird here.

Learn the ideas first, because tools won’t save you

DataOps is a way of working, and you can't buy a way of working.

People buy a data quality tool, point it at the warehouse, and wonder why nothing is better six months later. The tool was fine. The problem was the process around it, and no vendor ships that.

Start with the manifesto. It’s 18 points and takes 10 minutes. Read it yourself, send it to your reports, and see which points start an argument. That argument is useful. It usually lands on the part of your process that’s broken and that everyone has agreed not to talk about. From there, the order that works is the seven steps for what to actually change, the fundamentals course so your whole team means the same thing by the word “test,” and the maturity model to find out which weakness to attack first. A score beats your opinion when you’re asking your manager for time to fix something.



The “why” is worth more than it sounds. DataOps borrows from Lean manufacturing, Agile, and DevOps because those disciplines have already solved your problem: shipping reliable work with a handful of engineers and constant interruptions. Sprints look different when half the week is someone walking up to a desk with a broken dashboard, and the Agile webinar is honest about which parts don’t transfer cleanly.

One of these is for later. Recipes for DataOps Success is about running this across an organization, and you don’t own the org chart. What you own is a team that can produce a result other groups want, which is how this actually spreads. Read it when people outside your team start asking how you did it.

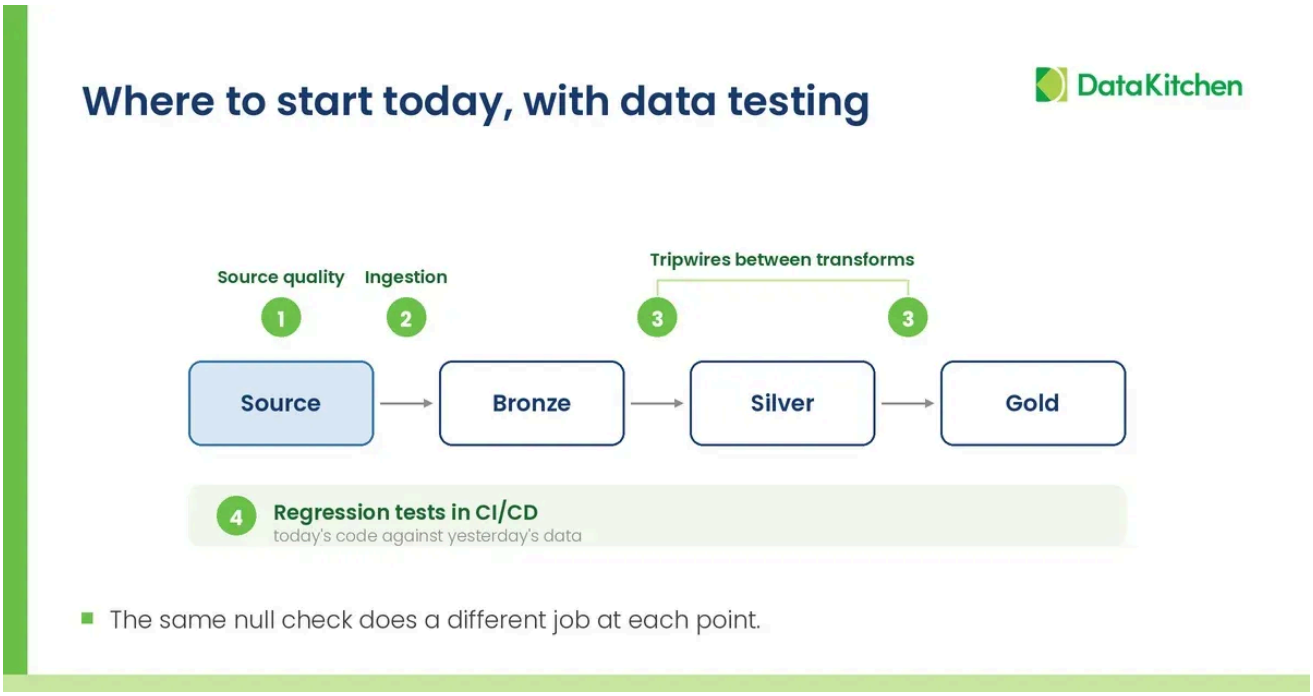
Where to start today, with data testing

You don't get trust back until the errors stop reaching your customers.

Pick one place in your current setup and improve it. Not a platform migration, which you couldn't get approved anyway. One place.

That place is testing. Every other DataOps improvement compounds on top of it. Deployment automation is great, but it can make bad data arrive faster if you skip this step. So you check the data in development, before your team merges, and in production, so you catch the vendor who changed their file format over the weekend.

The question your reports will hit in the first hour is where to put the tests, and the four points post is the most useful thing in this section. There are four places. Source data, so you can push problems back to whoever owns them. Ingestion, so you catch what arrives late, short, or reshaped. Between transforms, where a tripwire stops bad data before it reaches gold. And CI/CD, so a renamed column doesn't break a report three layers down. The same null check does a different job at each one. Run it in a single place, and you're covered for one failure mode and blind to the other three.



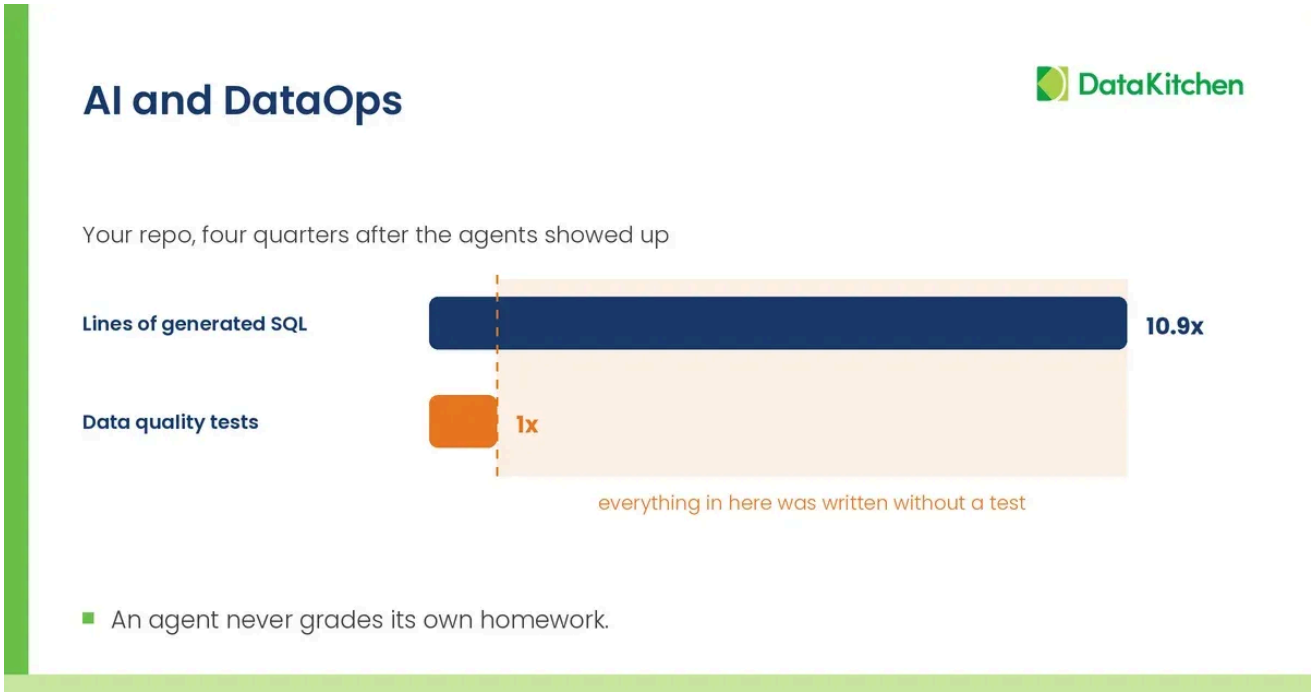
You'll also get an objection from whoever already wrote a few dbt tests. Do the math with them. A bronze layer of 100 tables averaging 10 columns needs roughly 200 table tests and 2,000 column tests, and silver and gold each want their own. At 30 minutes per test, that's months of typing, which is exactly why nobody does it. TestGen profiles the data and generates about 80 percent of them. Your team spends its time on the 20 percent that encodes business logic no profiler can infer. Docker compose up, point it at a schema, and the queries run inside your database, so no data leaves your perimeter and your security review gets shorter.

One distinction worth making before your team starts. Data quality tells you the values are wrong. Process observability tells you the job that produces those values ran late, ran twice, or didn't run at all. Most teams instrument the first and skip the second, which is how you end up with data that is perfectly valid and eight hours stale.

AI and DataOps

AI didn't change what good data engineering means. It changed how fast bad data engineering happens.

Your reports can generate a new transformation in 90 seconds, and it lands in review faster than you can read it. The bottleneck moved, and it moved onto your desk. Review catches the code that throws a stack trace. It doesn’t catch the join that fanned out on duplicate keys and made the revenue number 4 percent higher. Nothing failed. The dashboard rendered. Somebody made a decision.



Three things to take from the reading below. Idempotency stopped being good practice and became mandatory: a human who watches a load fail stops halfway and looks at the target table, while an agent sees a non-zero exit code and runs the command again, turning 1,000 rows into 2,000 with nobody watching the row count. Test coverage has to grow at the same rate as generated code, or you get 4,000 new lines of SQL sitting behind 40 old tests. And an agent never grades its own homework, which is the rule holding all seven sins together. The piece most teams miss is context. Your AI is only as good as what it knows about your schemas, your conventions, and the reason that one table has a column called `flag_2_final`.

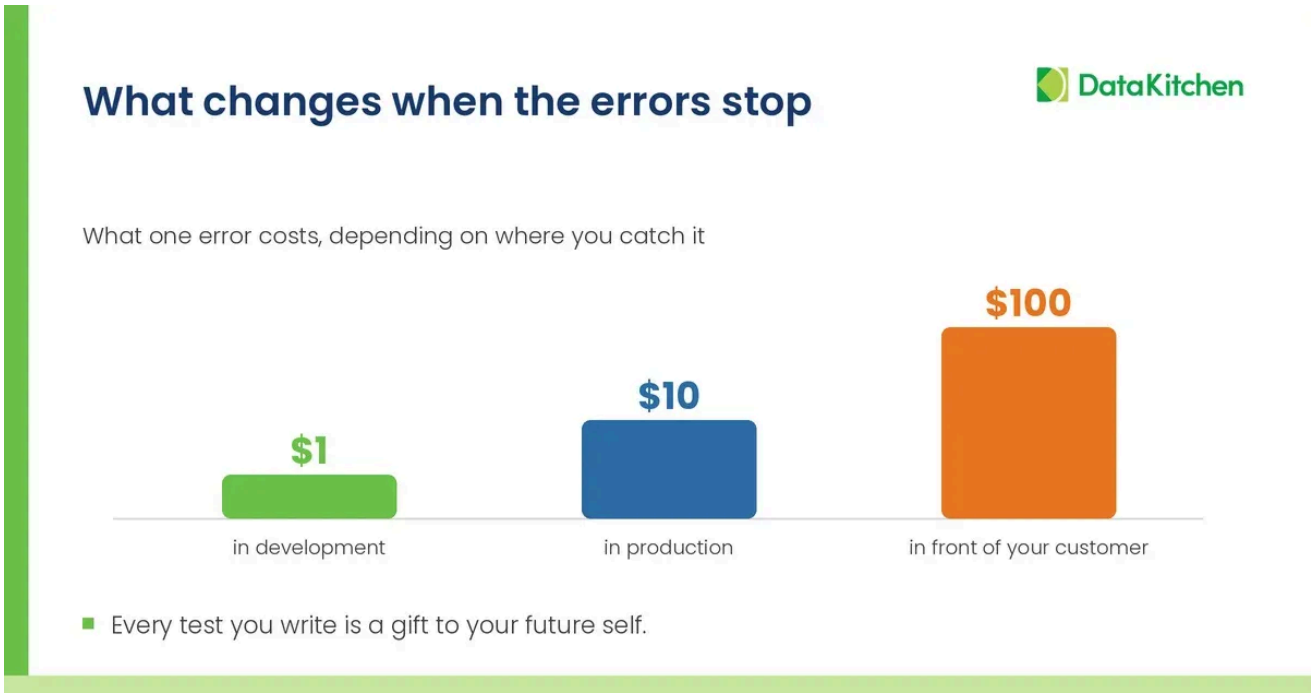
What changes when the errors stop

Reliability that depends on who is paying attention isn't reliability. It's luck with a rotation.

Right now your pipelines work partly because certain people are watching them. You know which tables are fragile. Your senior engineer knows which vendor file arrives malformed twice a quarter. That knowledge lives in heads, which means your error rate is a function of who is awake and who is on vacation. Heroics feel like competence from the inside. From the outside, they look like a system nobody can predict.

Everything in this post is one idea applied repeatedly: move that knowledge out of heads and into the system. It's borrowed from manufacturing and from software, and it holds in data too. Quality is a property of the process, not the vigilance of whoever is standing at the end of the line.

The economics are the 1:10:100 rule, which George Labovitz and Yu Sang Chang put a name to in 1992. A problem caught in development costs about a dollar. The same problem caught in production costs ten. Caught by your customer, in a meeting, on a slide with your team's name on it, it costs a hundred, and most of that hundred is paid in trust you spend the next two quarters earning back.



The return shows up on your team's calendar. Fewer days lost to forensics. Releases nobody has to stay late and babysit. Here's the marker of a high-performing data team that I like best: you can hand a 22-year-old the keys, let them rename a column, and a red light goes off telling them they broke a downstream report before it ever reaches production. That engineer learns in an hour what used to take an outage to teach. You review with less dread. Nobody apologizes to a VP.

Your team stops explaining and starts building. Every test you write is a gift to your future self, and when you have reports, to theirs.

Appendix: if somebody on your team came from software

In software, you control your inputs. In data, you don't.

Somebody on your team came from DevOps or application development, and this world is strange to them. The deploys are weird. The tests are weird. Production breaks because of data nobody on the team wrote, from a vendor none of you have met.

All of us who founded DataKitchen made this same move about 20 years ago, so they have our sympathy, and we've collected the shortest path we know.

The thing that has to land before anything else makes sense: your code can be perfect and your pipeline still fails, because the data changed underneath it. Every practice in this post comes from that one fact.

Read More

The following articles were not included in this edition of the book but cover related topics in depth. All are available free at datakitchen.io.

- Why Data Quality Isn't Worth the Effort: Data Quality Coffee With Uncle Chip
- Data Quality Gluttony: When More Data Makes Things Worse
- Data Quality When You Don't Understand the Data: Data Quality Coffee With Uncle Chip
- Warring Tribes into Winning Teams: Improving Teamwork in Your Data Organization
- The Art of Data Buck-Passing 101: Mastering the Blame Game in Data and Analytics
- Unlocking Data Team Success: Are You Process-Centric or Data-Centric?
- The Syntax, Semantics, and Pragmatics Gap in Data Quality Validation Testing
- Data Quality Testing: A Shared Resource for Modern Data Teams
- Peter Piper on the Four Ps of AI Data Quality: Purge, Patch, Push Back, or Pass
- Critical Data Elements: Your Shortcut to Data Governance That Actually Works
- Data Quality Circles: The Key to Elevating Data and Analytics Team Performance
- War Rooms Suck
- Navigating the Storm: How Data Engineering Teams Can Overcome a Data Quality Crisis
- Why Not Hearing About Data Errors Should Worry Your Data Team
- Data Quality Assessments: A Practical Guide
- No Python, No SQL Templates, No YAML: Why Your Open Source Data Quality Tool Should Not Require Coding
- The Consultant's Growth Playbook: Accelerating Client Acquisition with DataKitchen Open Source TestGen
- What Does Flossing One Tooth Have to Do with Data Quality?
- DataOps Observability: Taming the Chaos of Data Journeys
- The Five Use Cases in Data Observability: Data Quality in New Data Sources
- The Five Use Cases in Data Observability: Effective Data Anomaly Monitoring
- The Five Use Cases in Data Observability: Mastering Data Production
- The Five Use Cases in Data Observability: Fast, Safe Development and Deployment
- The Five Use Cases in Data Observability: Ensuring Accuracy in Data Migration
- Be the First to Know: Smart, Continuous Table Monitoring Has Arrived in TestGen
- Shopping for Tests: A New Faceted Picker for TestGen's Test Catalog
- TestGen Now Supports Oracle and SAP HANA, with a New Setup Wizard to Get You Running Fast
- DataKitchen Enhances Its Support for Data Stewards to Manage Data Quality Tests in TestGen
- Webinar: The Seven Deadly Sins of AI-Generated Data Pipelines
- Webinar: Stop Clicking, Start Asking , The AI Playbook For Data Quality
- Webinar: 10x Your Data Engineering With AI
- Webinar: 10x Your AI Data Analysis , The Context Engineering Formula That Works

- Webinar: The Four Points In Your Medallion Architecture Where Data Testing Really Matters
- Webinar: You’re Massively Overpaying For Data Observability
- On-Demand Webinar: Seven Sins Of Data Quality
- Webinar: A Masterclass In The Six Types Of Data Quality Dashboards
- TestGen MCP Cheat Sheet: 96 Tools, One Page, Zero Excuses
- Why Do I Need a Star Schema?
- The Age Of Outsourcing Your Commercial Pharma Data Team To India Is Over
- You Built the Thing. Now You're Stuck Running It.
- How TestGen Complements Microsoft Purview for Enterprise Data Quality
- The First Move When You Inherit Someone Else's Data