DATA ENGINEERING

Data Quality and Observability

Contracts, tests and lineage for pipelines you can trust

Houssam Kodad

PDF · DATAFORGE BOOKS

Questions about this sample or the full edition: **support@dataforgebooks.com**

CONTENTS

Table of Contents

CHAPTER 01

What Trustworthy Data Means

The fastest way to lose a stakeholder's trust is a number that is quietly wrong. Once a decision-maker catches one bad figure, they second-guess every figure after it, and the platform you worked hard to build becomes something people route around with their own spreadsheets. Data quality is not a nice-to-have bolted on at the end; it is the thing that makes everything else worth having.

This chapter defines what "trustworthy" actually means in measurable terms, argues that quality should be treated as a product feature rather than a background chore, and puts a real, defensible cost on the wrong number so you can justify the investment to people who control budgets. We finish by previewing the system — tests, contracts, lineage and alerting — that the rest of the book assembles.

The goal throughout is to make "is this number right?" a question you can answer with evidence in seconds, rather than a question that triggers an afternoon of nervous investigation. A platform that can answer it confidently is one people build on; a platform that cannot is one they slowly abandon.

The Six Dimensions of Quality

"Good data" is too vague to act on. It becomes actionable the moment you break it into dimensions you can actually test. Completeness asks whether anything is missing. Validity asks whether values conform to rules. Accuracy asks whether they match reality. Consistency asks whether related systems agree. Timeliness asks whether the data is fresh enough to be useful. Uniqueness asks whether duplicates have crept in.

Each dimension maps cleanly to a concrete check. Completeness becomes a not-null or row-count test; validity becomes a constraint or accepted-values test; timeliness becomes a freshness SLA on a source. Naming the dimension you care about for a given dataset is the first step to writing a check that catches its failure automatically — before a human notices and loses faith in the whole platform.

Quality as a Product Feature

Teams that treat quality as cleanup do it last, under deadline pressure, and inconsistently, so it never quite happens. Teams that treat it as a feature build it in from the start: every dataset ships with tests, a named owner, and a documented freshness guarantee, exactly the way a software feature ships with tests and an SLA. The difference in outcomes is dramatic and compounding.

This shift changes the definition of done. A dataset is not finished when the pipeline runs without error; it is finished when it is observable and trustworthy — when a failure surfaces loudly and a stakeholder can see for themselves that it is fresh and correct. The rest of this book gives you the tooling to make that the default state of every table you ship, rather than an aspiration.

Tests That Catch Issues Early

The cheapest place to catch a data problem is at the boundary where bad data enters or is produced, long before it reaches a dashboard. A good testing strategy layers several kinds of check: schema tests that a column exists and has the right type, not-null and uniqueness tests on keys, accepted-values tests on categorical fields, and referential tests that every foreign key has a matching parent.

The crucial design choice is that a failing test should stop bad data from propagating, not merely log a warning that nobody reads. When a quality gate fails the build or quarantines the offending rows, the problem is contained to one place and one moment. When it only warns, the bad data flows downstream and you spend the afternoon tracing a wrong dashboard number back to its source — the exact situation tests exist to prevent.

```
# Distribution test: yesterday's row count should not deviate wildly
# from the trailing average — a cheap, powerful anomaly check.
with daily as (
    select date(created_at) as d, count(*) as n
    from orders group by 1
)
select d, n
from   daily
where  n &lt; 0.5 * avg(n) over (order by d rows between 7 preceding and 1 preceding)
   or  n > 1.5 * avg(n) over (order by d rows between 7 preceding and 1 preceding)
```

Freshness and the Silent Failure

The most dangerous data failure is the silent one: a feed that stops arriving without throwing an error, so yesterday's data simply never updates and every downstream number is quietly stale. No test on the data itself catches this, because the data that is there is perfectly valid — there is just not enough of it, and it is too old.

Freshness checks are the answer. They assert that the newest record in a table is no older than some threshold, turning "the feed silently stopped" into a loud, specific alert. Pairing a warning threshold with a stricter error threshold lets you distinguish "slightly late, keep an eye on it" from "broken, page

someone." Freshness is the single highest-value check most teams are missing when they start.

Data Contracts at the Source

Tests catch bad data after it has been produced. Contracts try to prevent it from being produced at all. A data contract is an explicit agreement about the schema, types and semantics of the data a producer emits — and, critically, it is enforced, so a producer that tries to rename a column or change a type is stopped in their own pipeline rather than breaking yours hours later.

Contracts move the cost of a breaking change onto the team best placed to handle it: the one making the change. They turn the most common and most maddening data incident — "someone upstream changed something and didn't tell us" — from a downstream fire-drill into a caught, reviewable event. We will look at how to introduce them without slowing producers to a crawl, because a contract nobody can satisfy is just friction.

Lineage and Blast Radius

When something does go wrong, the first question is always the same: what does this affect? Lineage — the map of which tables and dashboards derive from which — turns that question from an hour of tribal-knowledge archaeology into a glance at a graph. Column-level lineage goes further, showing exactly which downstream fields depend on the one that broke.

Lineage pays off in both directions. Downstream, it tells you the blast radius of an incident so you can warn exactly the right people. Upstream, it lets you trace a suspicious number back to its origin without guesswork. And before you make a change, it shows you what you might break — turning a risky edit into an informed one. Observability without lineage is a smoke alarm with no map of the building.

Alerting Without Crying Wolf

The failure mode of a maturing quality system is not too few alerts but too many. A check that fires on every minor blip trains the team to ignore it, and an ignored alert is worse than no alert because it provides false comfort. Good alerting is therefore as much about tuning and routing as about detection — every alert needs a severity, an owner, and a clear idea of what the recipient should do about it.

The principle is that an alert should represent a decision, not just a fluctuation. Pair severities so that warnings inform and errors page. Route each alert to the team that owns the data, not a shared channel where it dies. Review noisy checks and tighten them. The rest of this book builds toward a

quality system that is trusted precisely because, when it speaks, people know it is worth listening.

This is a free sample

You've reached the end of the sample chapter.

Get the complete book — every chapter, fully worked — at **dataforgebooks.com**.

FULL EDITION · 152 PAGES · PDF

Read the full title at **dataforgebooks.com**

Questions? support@dataforgebooks.com