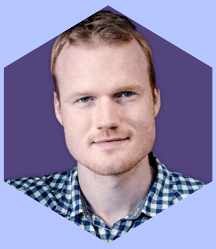
SYNQ

The Definitive Guide to Building Data Products

SYNQ

Foreword



Petr Janda
CEO & founder

Mikkel Dengsøe
Co founder

Building data products is complex. Scaling them to meet business needs is even more challenging. The Definitive Guide to Data Products is a practical guide designed to help teams design, build, deploy, and operate products that power critical workflows.

Whether you're a data engineer, analytics manager, or part of a broader data team, Building Reliable Data Products offers the insights and clear framework to build data systems that reliably drive meaningful business value.

You can always find the latest version of the guide on synq.io/guide

UNLOCK INSIGHTS FROM TEAMS LIKE THESE

IAG INTERNATIONAL AIRLINES GROUP · Typeform · bp · Shalion Actionable eRetail Data · Ebury · aiven · BETTER COLLECTIVE · reMarkable · SPENDESK · LUNAR · Dext · clair · voi.

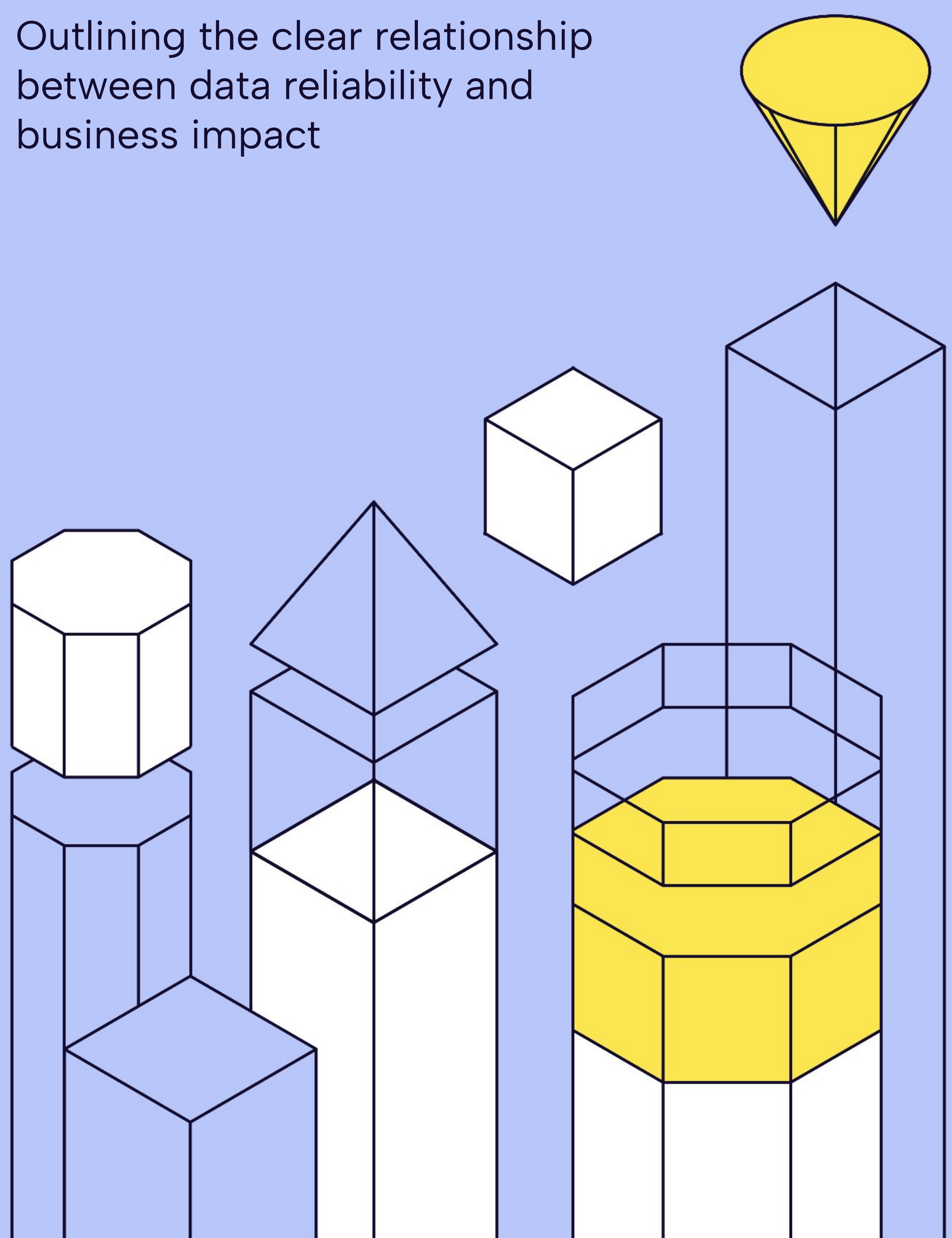
Table of contents

SYNQ

CHAPTER 1

Introduction

Outlining the clear relationship between data reliability and business impact



We've led data and software teams at Google, GWI, Pleo, and Monzo and spoken with thousands of data teams over the past years. Over again we see the same struggles from data teams – they lack a framework to support the growing expectations of use cases from data in their companies.

More is expected from data teams

Data is increasingly used for business-critical use cases

Data has evolved from being a nice-to-have to becoming a critical component of core business processes.

Data used for decision-making is important and if data is incorrect it may lead to wrong decisions and over time a loss of trust in data. But data-forward businesses have data that is truly business-critical. If this data is wrong or stale you have a hair-on-fire moment and there is an immediate business impact: Tens of thousands of customers may get the wrong email as the reverse ETL tool is reading from a stale data model. You're reporting incorrect data to regulators and your C-suite can be held personally liable. Or your forecasting model is not running and hundreds of employees in customer support can't get their next shift schedules before the holidays

Data stacks are getting more complex

15% of dbt projects have more than 1,000 models, and 5% have more than 5,000 models. These aren't just large numbers—they represent growing data ecosystems connected to hundreds of upstream data sources and feeding hundreds of downstream destinations. Data Mesh has gained traction as a solution for scaling data work but teams struggle translating the principles into actions.

This scale comes at a cost. Velocity and agility slows, creating frustration both inside and outside the data team. Collaboration becomes harder as no one is familiar with the entire code base. Time spent in meetings goes up relative to time spent getting things done. Quality becomes harder to enforce over a growing surface area and user-reported errors increase. And SLA achievements decline as more jobs fail, but no amount of retros seems to reverse this trend.

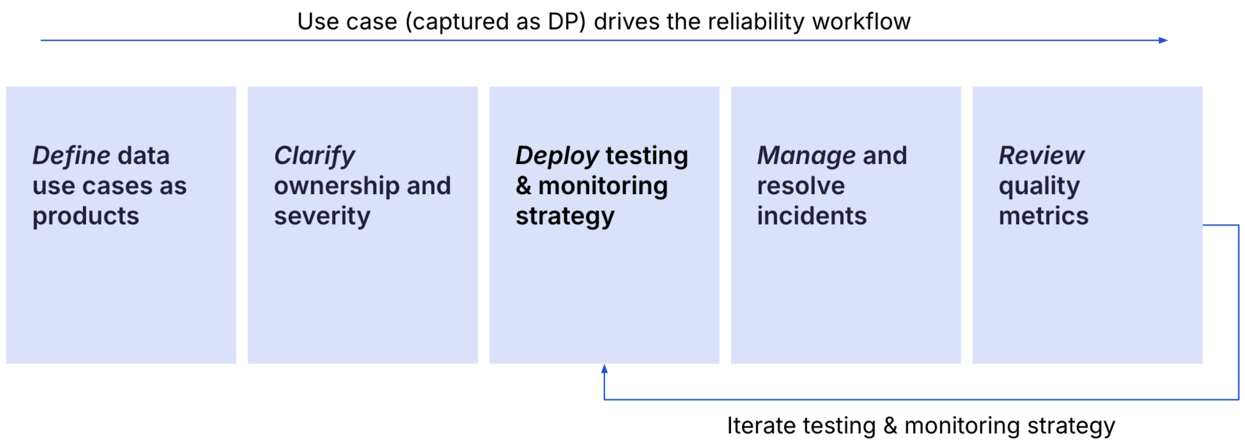
Lack of testing framework

The root cause of this is that data teams are not well-equipped to take a strategic approach to building reliable data. This means that tests are placed sporadically without much consideration of the use case of the data and that ownership is unclear creating 'broken windows' of tests that are left failing too long.

As a consequence, data teams drown in alert, stakeholders are the first to discover issues, and the belief in the impact of testing starts to lose its impact.

The Data Product Reliability Workflow

We recommend you think systemically about the reliability of your data stack through a 5-step framework – from defining data use cases as products, setting ownership & severity, deploying strategic tests & monitors, and establishing quality metrics. We call it *The Data Product Reliability Workflow*


Use case (captured as DP) drives the reliability workflow
Define data use cases as products
Clarify ownership and severity
Deploy testing & monitoring strategy
Manage and resolve incidents
Review quality metrics
Iterate testing & monitoring strategy

In this guide, we'll work you through actionable steps you can take to adopt the Data Reliability Workflow at your company, including specific examples, tips & tricks, and considerations.

- **Chapter 2 – Setting expectations**: A framework to define data products to manage tiers of data assets and SLAs for maintenance.
- **Chapter 3 – Proactive testing & monitoring**: We build a testing and monitoring framework that maximizes errors caught and minimizes alerts.
- **Chapter 4 – Ownership with rapid response**: Developing scalable ownership and efficient incident management processes to quickly resolve issues.
- **Chapter 5 – Continuous improvement**: Establishing feedback loops and learning processes to continuously enhance data reliability practices.

Reliable data isn't just an operational requirement, it's a competitive advantage. By prioritizing trust and quality, your team will be trusted by owning critical processes and moving faster.

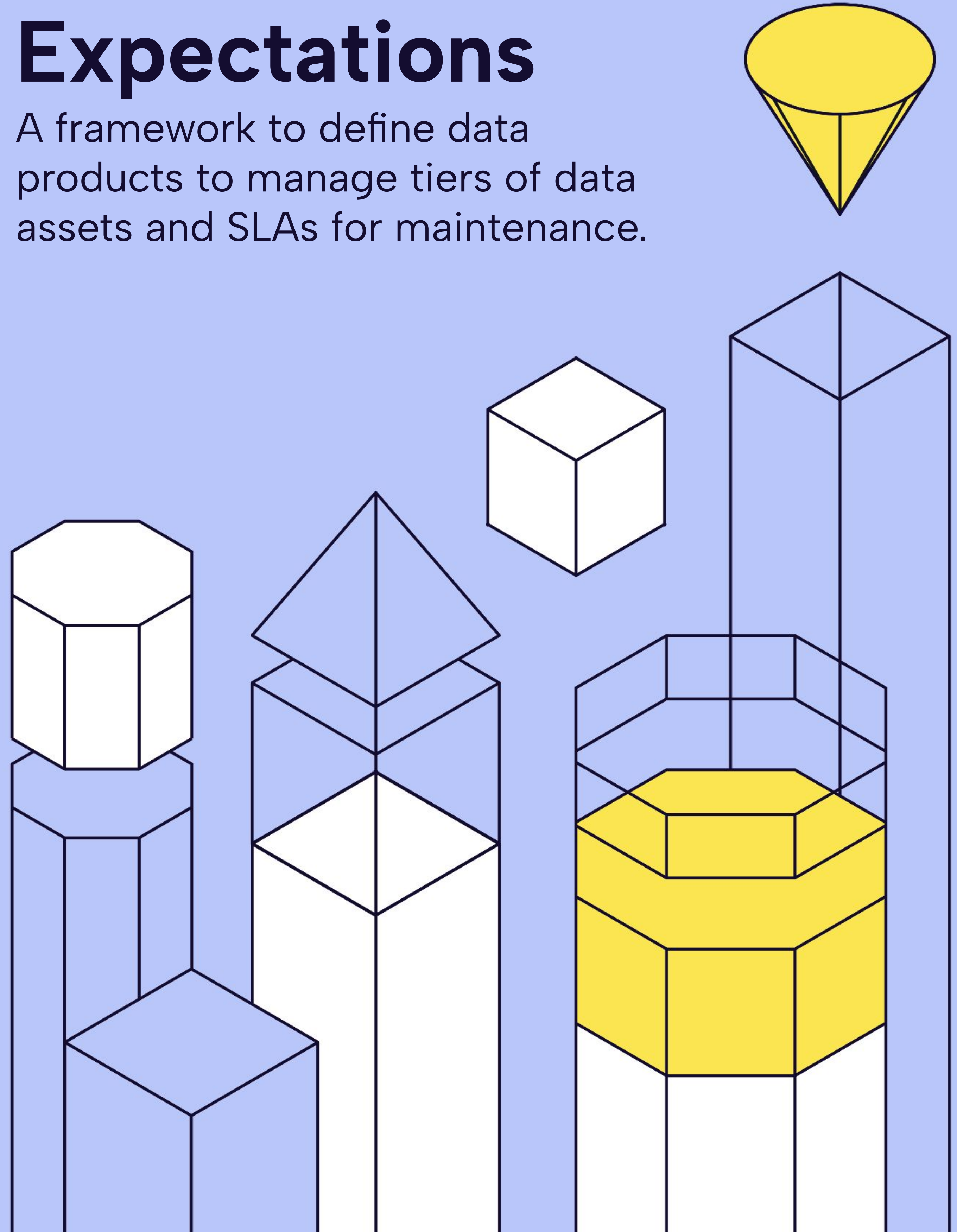
Happy reading!

SYNQ

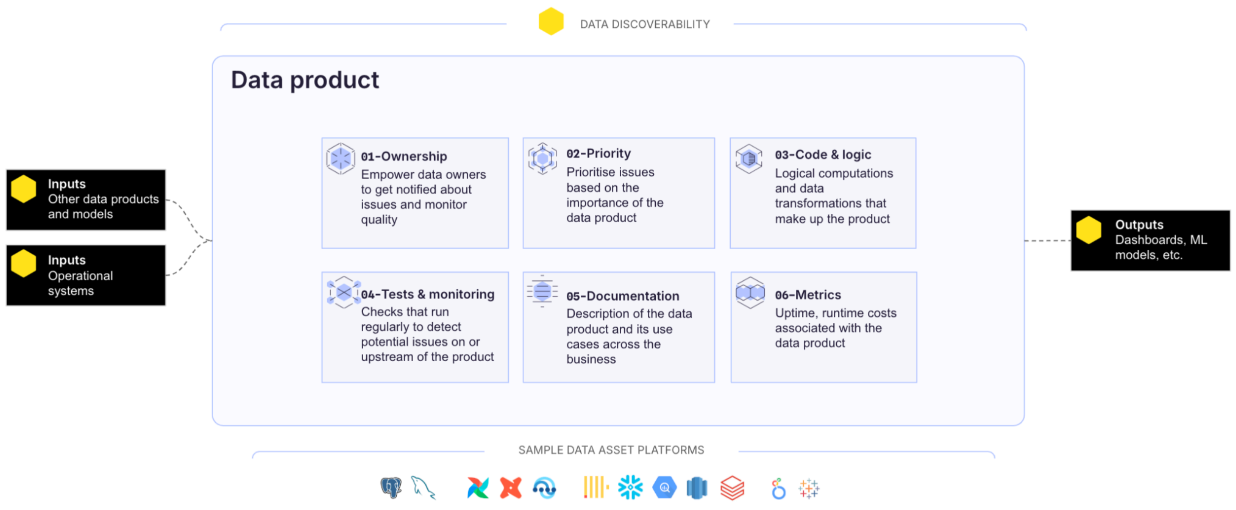
CHAPTER 2

Setting Expectations

A framework to define data products to manage tiers of data assets and SLAs for maintenance.



A data product is a group of data assets structured based on their use case. If you define your data products well, everything follows. Ownership becomes clear, it guides your testing and monitoring strategy, and you can manage and maintain SLAs in a way that's consistent with data consumers' expectations.


DATA DISCOVERABILITY
Data product
01-Ownership
Empower data owners to get notified about issues and monitor quality
02-Priority
Prioritise issues based on the importance of the data product
03-Code & logic
Logical computations and data transformations that make up the product
04-Tests & monitoring
Checks that run regularly to detect potential issues on or upstream of the product
05-Documentation
Description of the data product and its use cases across the business
06-Metrics
Uptime, runtime costs associated with the data product
Inputs
Other data products and models
Inputs
Operational systems
Outputs
Dashboards, ML models, etc.
SAMPLE DATA ASSET PLATFORMS

The building blocks of a data product – ownership, prioritization, code & logic, tests, documentation, and metrics.

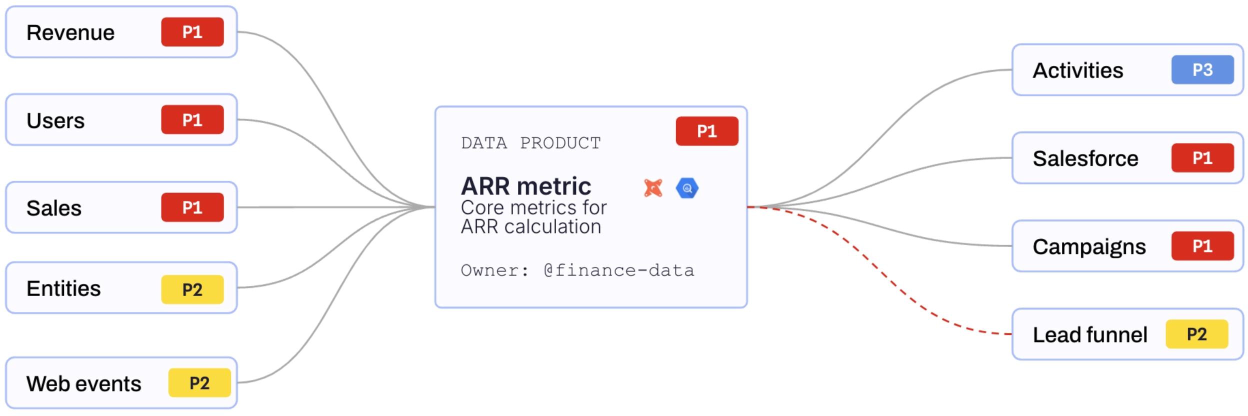
A data product rarely works in isolation but most often relies on input, either other data products or data directly from operational systems. Therefore, expectations for the data product should be set depending on its entire lineage.

In this chapter, we'll look at everything you need to know to get started with data products – from identifying and defining them to determining their priority and setting SLAs.

Case Study: How data products helped Aiven untangle their spaghetti lineage

The data team at Aiven, the open-source AI & data platform, struggled with untangling dependency across their data stack with more than 900 dbt models. Circular dependencies meant that it was difficult to make sense of the lineage which slowed down root cause analysis and made it difficult to make system-design decisions. It was particularly difficult for their core data products with hundreds of dependencies such as the ARR calculation, which is one of the most important metrics provided by the data team.

By encapsulating data products across data-producing and consuming domains, they could clearly understand the lineage, and instead of having to understand hundreds of upstream tables, they could visualize and understand the line through the lens of just a handful of data products.



With this at hand, they can trace back an issue to a faulty upstream data product and easily identify its owner for escalation, or see which data products are impacted downstream from an upstream failure–all without everyone needing to understand all the internals of each data product.

Another benefit is that system design issues stand out, so for example, if a circular dependency is introduced, it's much easier to understand it across a few dozen data products instead of a spaghetti lineage of hundreds of data models.

Identifying your data products

If you can identify the most critical business processes your data team supports, those are most often the data products you should identify. If you're unsure which ones they are, look for these signals such as what was impacted in your most recent critical data failure, or which data assets have the highest usage across your company.

Here are some examples of what can make up a data product:

- A set of dbt models and metrics within a specific dbt folder, like a finance mart
- A group of dbt models linked by an exposure, for instance, models used by a customer lifetime value (CLTV) model that powers marketing automation
- A selected collection of dashboards in a BI tool, such as core KPI reporting

At SYNQ, we think about our data products as either ***producer*** or ***consumer*** products. Producer data products are read from operational systems (e.g., API or SalesForce data) and owned by data or platform engineers. Encapsulating them in data products provides an easy-to-understand getaway for downstream teams to escalate issues upstream without grasping the full complexity of the internals of the data products. Consumer data products directly expose data to consumers and read from other data products or assets (e.g., CLTV or Marketing Attribution Model), often owned by data analysts or scientists. Encapsulating these in data products gives everyone a good understanding of the direct impact of issues and which stakeholders should be notified.

The number of data products is highly dependent on your company's size and complexity. Some companies have dozens or hundreds of data products while others have just a handful. Start small, by identifying a handful of your most important data products, and then build out from there.

Case study: Establish the right granularity when identifying data products

The data team at Aiven started with high-level products such as Sales and Marketing but realized they needed to go a step deeper to have the most impact. "If the Marketing data product has an issue, that may be fine. However, if the Attribution data product within Marketing has an issue, we must immediately jump on it. This is the level of detail our data products need to be able to capture." - Stijn, Data Engineering Manager

Defining your data products

Once you've identified your data products, the next step is to define them. When defining your data products, we recommend following these five steps:

1. Data products should be defined as close to where they're used as possible to reflect the experience of the data consumer
2. Data products should take into account upstream dependencies as far upstream as possible to give a complete overview
3. Data products should have an owner responsible for the operations during their entire lifecycle such as continuous monitoring to ensure quality and availability
4. Data products should have a priority assigned indicating their importance (P1, P2, ...)
5. Data products should have a description so that people with less context can understand it's use case

As you scale your data products, you can group them into domains to make it easier to manage them. We recommend you keep data products as focused as possible – for example, it's never a good idea having a data product consist of many hundreds of assets as it will create a web of interdependencies.

Case study: Group data products into related domains

The data team at Aiven groups data products into business groups such as Sales, Finance, and Marketing as well as Core and API for more technical data products. "With this at hand, we can monitor the health across different groups to get a high-level overview, while also zooming into specific data products. This is also helpful for us when mapping priority–for example, our Attribution Marketing product is P1 while our Market Research product is P3." – Stijn, Data Engineering Manager

The definitions of data products should follow existing workflows. For example, if you already have a process for defining asset ownership and priority, this will also fit data product definitions.

At SYNQ, we take a pragmatic approach. *Producer* data products are defined based on specific assets such as Postgres tables and dbt sources. *Intermediate* data products are defined in code through dbt metadata tags and groups. *Consumer* data products are defined through their folder structure in dbt and Looker which resembles their use cases.

```
version: 2

data_products:
  - name: marketing_attribution
    description: >
      This data product includes models for tracking marketing attribution
      across various channels. It powers the Marketing Attribution Dashboard
      and is critical for assessing campaign performance and optimizing
      marketing spend.
    owner:
      name: marketing_data_team
      slack_channel: "#marketing-data-team"
    priority: P1
    assets:
      - name: dim_marketing_campaigns
      - name: fct_channel_performance
      - name: fct_attribution_model
```

While you'll want visibility into the data product's dependencies as far upstream as possible, we don't recommend managing this manually. It's not uncommon to have hundreds of dependencies upstream of a data product, and new dependencies can be added without the data product owner being aware of it. Instead, you should rely on automated lineage toolings such as those from a data catalog, dbt, or a data reliability platform.

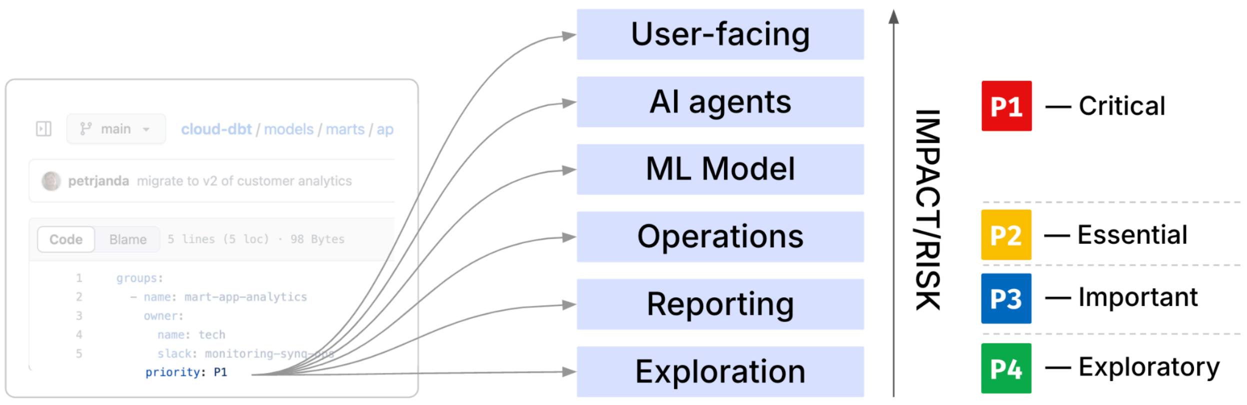
Determining the priority of data products

As you identify your data products, you should carefully consider their priority. This serves as a guiding principle for key data product workflows: How soon should issues be addressed, do they require an on-call schedule and what's a reasonable SLA.

Determining the priority of your data products is significantly easier if you've defined your data products at the right level of granularity. For example, if you've defined an entire product as Marketing, many people will have a different take on its priority. But if you've split Marketing into Market Research, Attribution Reporting, and CLTV calculations, some will naturally be more important than others.

At SYNQ, we use the following priority levels:

- **Product operations (P1)** when a user-facing operational system runs queries against the dataset. The data is either directly displayed to the customer or powers product internals (such as our anomaly engine), with a direct impact on the functionality of our product.
- **Client exports (P2)** when a given dataset is shared with any customer in an "offline" or ad-hoc manner, which isn't operational but could still impact the customer directly.
- **Business critical workflows (P2)** when analytical data feeds other operational systems with indirect impact on customers such as marketing automation or critical business decisions such as health scoring.
- **Business intelligence (P3)** which includes standardized reports and datasets that we use for company strategic and tactical decisions.
- Every other use case is classified as **Others (P4)**.



```yaml
groups:
  - name: mart-app-analytics
    owner:
      name: tech
      slack: monitoring-synq-ops
    priority: P1
```

Defining severity based on the use case. P1 is critical. P4 is exploratory

The priority of your data products must be closely linked to expectations for how to deal with data incidents.

Case study: Get exec buy-in for setting data product priority at fintech scaleup

One pitfall when defining data product priority – especially in larger data teams with many stakeholders – is that they'll have widely different opinions on what's important. You should establish a set of boundaries for how these are defined. After all, not everything can be P1. We recommend working with senior stakeholders to agree on what the priority is and get their buy-in.

The Danish fintech Lunar established a data governance framework with exec buy-in to define critical data elements across the company. "Every three months, we meet with the chief risk officer, chief technology officer, and bank CEO to update them on the latest developments, risks, and opportunities. This helps everyone contribute to and have a stake in our data quality" - Bjørn, Data Manager

Establishing SLAs for your data products

An SLA (Service Level Agreement) is a contract that defines the expected service level between a provider and consumer, including remedies if expectations aren't met. For example, a P1 SLA for a critical customer-facing dashboard might guarantee 99.9% uptime. If the dashboard is stale or has other data quality issues, the response time is 30 minutes with a two-hour resolution. In contrast, a less critical P3 dashboard may have 95% uptime and slower response times, like 48 hours for resolution.

We don't recommend measuring the entire system with metrics such as model test coverage, as these can give a false sense of security and lead your team to optimize toward the wrong goals.

"Measuring data product SLAs directly ties to the experience your data consumers have and is therefore a much better metric for the quality of your team's output."

Measuring data product SLAs

We recommend that you consider two metrics to get the most complete picture.

- Coverage – what % of assets have the required data controls in place
- Quality score/SLA – what % of controls in place are passing successfully

Only by looking at both metrics can you confidently say whether the data product SLA is meeting the expectations you've set. Both metrics should be monitored across the lineage of the data product to include the status of any upstream dependencies.

Case Study: Data product SLA metrics should include all upstream dependencies

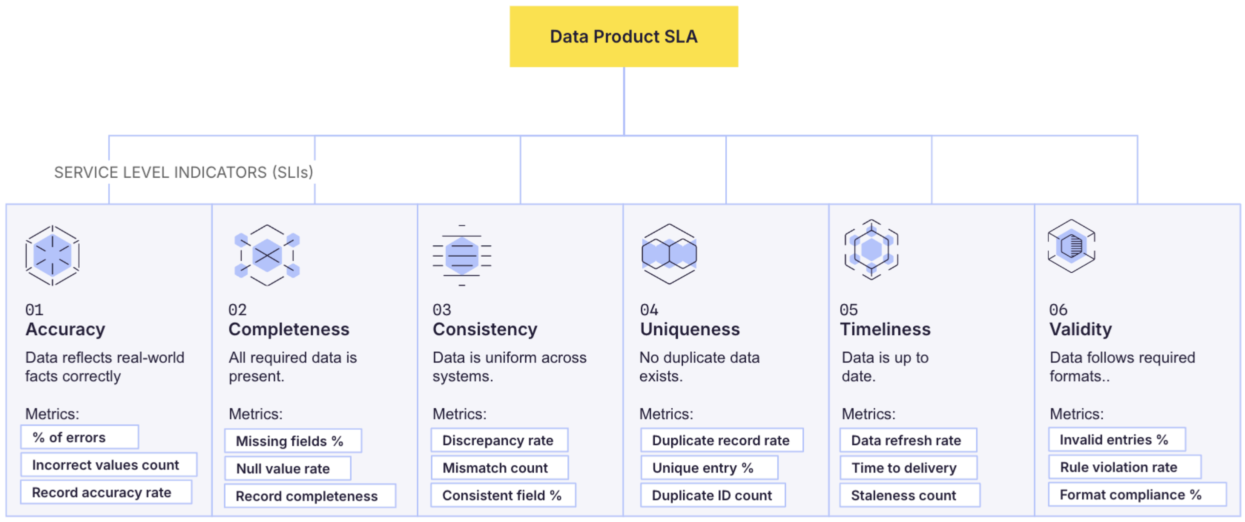
The data team at Lunar reports on data quality KPIs of their most important data products to the C-level every 3 months. "As a regulated company, we need to be able to demonstrate to regulators that we have sufficient data controls and an ability to trace these across all dependencies. This also gives us something demonstrable we can show to our regulators." – Bjørn, Data Manager

Coverage–define the data control expectations (coverage) for your data products based on their priority. This helps align everyone around expectations and the level of monitoring that's sufficient. For example, for P1 data products you may agree on expectations as:

- All sources must have freshness checks – either explicitly or implicitly through self-learning monitors
- Key metrics must have relevant accuracy checks in place
- Row count must be tested before and after joins are performed
- Key fields must have `not_null` and `uniqueness` tests

Quality score/SLA–the best way to measure the SLA of a data product is to look at all data controls through the lens of different KPIs, also known as Service Level Indicators (SLIs). Grouping your SLIs into meaningful areas helps isolate problematic areas and fits well into the type of tests you can perform in tools like dbt. (1) Accuracy—does the data reflect real-world facts (2) Completeness—is all required data present and available (3) Consistency—is the data uniform across systems or sources (4) Uniqueness—are there no duplicate records in the dataset (5) Timeliness—is the data updated and fresh (6) Validity—does the data conform to the required formats and business rules.

To measure the SLA of data products you can sum up the performance of the SLIs. SLA is then calculated as $sum(errored\ SLIs)\ /\ sum(SLIs)$.



Some data products may have different sensitivity to different SLIs. For example, for an ML model training set, accurate and complete data may be more important than timely data. In these cases, you should define separate Service Level Objectives (SLOs) for each SLI.

A benefit of the approach above is that once you've set up adequate data controls, monitoring, and measurement are fully automated and objective. We recommend that you also monitor the number of declared incidents by their priority as these give you a more subjective view into when a data product was impacted by an issue that was escalated to an incident.

Internally at SYNQ, we rely on automated grouping for all our SLIs. This means that new data controls are automatically grouped and counted towards the data product SLA and coverage as they're added.

Setting expectations for SLA levels and remediation

For a given month, to keep different SLA levels given 1,000 data checks on and upstream of the data product, this is the level of failures you can tolerate

- 99.9% reliability: 1 failure, equivalent to 43 minutes of downtime.
- 99.5% reliability: 5 failures, equal to 3.6 hours of downtime.
- 99% reliability: 10 failures, around 7.2 hours of downtime.
- 95% reliability: 50 failures, equivalent to 1.5 days of downtime.
- 90% reliability: 100 failures, up to 3 days of downtime.

Case Study: Shalion contractually commits data product SLAs to external customers

For Shalion – the eCommerce data solution platform – data is their product. If they provide unreliable insights to end-users through their customer-facing dashboards, this directly impacts customer trust and retention. Therefore, the data team's next step is to contractually commit to SLAs around the accuracy and correctness of their data products and thoroughly measure the time it takes to detect and notify impacted stakeholders of issues.

In the ideal world, you would be able to guarantee 99.9% SLA of all your P1 data products. In reality, this may be a steep goal to reach. In some cases, the SLA may be something you commit to contractually such as the uptime of a customer-facing dashboard. But in most cases, it will serve as a guideline, and help align data consumers and producers around a common goal.

You should align your incident management processes closely to the priority and SLA of your data products. A 24-hour response time on a P2 data product with a 90% SLA may be OK. But you may need to jump on issues on P1 data products much faster than that.

SYNQ

CHAPTER 3

Proactive Testing & Monitoring

We build a testing and monitoring framework that that maximises errors caught and minimises alerts.

Before we talk about building a robust testing strategy, it is worth discussing some common approaches to testing and monitoring that are widely adopted across the industry but not optimal.

Anti-patterns

Redundant testing

Data platforms are composed of data assets — tables, metrics, dashboards and more — created by transformation tools like dbt models, Spark jobs or ingest pipelines.

A simple mental model I frequently use is to see data platforms as a network of data assets, interconnected by transformations that build them on top of each other. Data flows through the network from source systems through layers of transformations into its final use case, whether a dashboard, data export, or machine learning model.

To ensure these complex chains of transformations work, data practitioners are increasingly applying various testing techniques such as data tests, anomaly monitors, data contracts or unit tests. Especially built-in tests in transformations frameworks such as `unique`, `not_null`, `accepted_values`, or `relationship` tests are the most popular.

To encourage adoption of testing many teams set test coverage targets and generally encourage teams to write tests for every model. But while the intention might be good, this 'test every model' approach to testing might not yield the best results.

The model-centric approach does not consider the wider context, which almost always leads to redundancies. Tests are applied in a sequence, following the DAG of transformations, often without any chance for this data to go wrong as it moves between the models. This doesn't just lead to unnecessary computing costs. More severely, it creates a false sense of safety.

Example: Redundant basic dbt tests

*We've built a model called **`stg_latest_issues`** that pulls data from a **`stg_issues`** model. It has relatively simple code:*

```
select
```

```
    id,
    workspace,
    ...
  from
    {{ ref('stg_issues') }}
  order by
    ingested_at
  limit
    1 by id
```

*The upstream table contains one record for every modification of a given issue, and the model pulls the latest record for each issue by **`id`**, which is the system's unique identifier for the problem.*

*One possible issue with such data is that an id will be empty, suggesting that we are working with a corrupted issue record. It is also why the upstream table **`stg_issues`** verifies that such a situation can't happen with the following test:*

```
select * from {{ ref('stg_issues') }} where id = ''
```

If this test returns any records, it will fail.

*But what happens when we pull data to the downstream model **`stg_latest_issues`**? Do we test for empty **`id`** again? How could this **`id`** possibly become empty? The short answer is that it can't.*

The above example is a very simplified case of a problem that can lead to significant redundancies in testing.

"Mechanically re-applied tests without awareness of the broader data pipeline do not add more safety to our data."

Disconnected tools

As data systems evolve, so does the data quality tooling. Today, it's common to see teams using combination of tools for data quality: dbt for testing, a separate platform to execute anomaly monitoring, and another system to monitor overall ETL pipelines that coordinate their execution.

Teams involved in analytics across Data Engineering, Analytics Engineering, Data Science, Data Analytics also have different needs. While data engineers are predominantly focused on executing pipelines and want to ensure that all necessary jobs work correctly, analytics engineers or data analysts are more concerned about the content of the data itself, deploying data tests and anomaly monitoring. On top of this, governance teams have their expectations to understand the strategic evolution of data quality, for which they frequently profile the data and measure its quality on several governance dimensions.

It leads to fragmentation.

Example: Freshness monitoring on tables goes wild

A practical example is a setup with a dbt project with hundreds of tables complemented by a sub-optimally configured anomaly monitoring tool. dbt executes in hourly jobs, so the entire DAG refreshes once per hour under normal system operations.

On the side, the team has deployed an anomaly monitoring system, and one of its rules is monitoring for freshness, e.g., detecting if the data gets abnormally delayed. These types of monitors are now table stakes in many monitoring tools, and therefore, they will quickly learn a correct pattern of data refreshing every hour.

One day, this pipeline breaks at 2 am. One dbt model didn't execute correctly due to a timeout issue with the underlying data platform. It causes the entire DAG to fail, with one failed model and hundreds of others skipped.

Thanks to a well-organized dbt alert, the right team gets notified about the failure with the proper context of what failed and the fact that the entire downstream pipeline is affected.

But what about the freshness monitors? Every monitor for every skipped model reports an issue just an hour later. The team is hit with unactionable alerts, which *add little value to their issue resolution. dbt job failure has already provided all the context of what models have been skipped.*

A more cohesive and strategic approach to testing could have avoided the unpleasant experience of hundreds of unnecessary alerts.

For example:

"If we correctly monitor our dbt jobs, why do we want to deploy any freshness anomaly monitors on any tables created by models in these dbt pipelines?"

There should be no scenario in which we wouldn't know about a failure deterministically by monitoring the jobs. In above scenario an anomal monitor that behaves probabilistically introduces a potential risk of false alerts and at best, it will learn to predict delays in our scheduler, which we already know and control.

'Test/monitor all'

Data ecosystems are becoming increasingly complicated, and testing approaches recommending testing or anomaly monitoring every table need more context of critical use cases. One example is hundreds of freshness tests deployed where they add no value.

One standard industry advice is monitoring a wide range of tables (or all) for table-level anomaly patterns. One possible reason for this advice is that such monitors have become relatively easy to execute at scale. But such a setup is often noisy. Data anomalies tend to cascade across data stacks. So, an anomaly detected in one table could trigger dozens of other tables to report abnormal data patterns without much additional value.

The same applies to testing. If we attempt to test everything, we will likely receive a proportionally higher number of alerts, but we might need more incentive or urgency to fix them.

Therefore, the desire to have a wholly tested data stack, whether with anomaly monitoring or data testing, is academic. It sounds like a good thing to do in theory. Still, today's data systems practically contain a mixture of use cases, and data flows with different risks and expectations of reliability.

Focus on models

Today's prevalent approach to testing in the industry is focused on models, which is why we see advice such as 'every model should be tested.' We also see teams setting targets, such as test coverage, to incentivize teams to make measurable progress towards such goals. While such advice takes some inspiration from software engineering, it lacks nuance. Most of the software engineering code in systems we take inspiration from is in production. If such systems fail, they cause an immediate and direct impact on customers, so their reliability is paramount. But that is not the only type of code software engineers write.

In some cases, engineers write one-off scripts to execute a finite task. Such code will not become part of the production system and, in some ways, becomes obsolete once the task is done. However, for posterity, scripts are frequently checked in a source code repository and live alongside the production codebase. Still, engineers typically apply much more rigorous testing on code that will have to run in production than ad-hoc code.

This distinction is even more critical in analytics data platforms.

Today's analytics systems don't clearly distinguish between production and scripting. In software engineering, the software is typically composed of reusable modules and libraries that can be freely reused in scripts, often executed on an engineer's local computer. Data teams have to deal with many more constraints. Their systems run on cloud data platforms and must process vast amounts of data to perform experiments. They often have no option but to include their work in the rest of the data in their production data platforms.

"This forces data teams to experiment in a way that is much more integrated with their other, often critical data. In other words, the experimental code is mixed in a DAG of models created on each other."

Case Study: Life-cycle of a reverse ETL

You are an analytics engineer at a mid-sized organization with 500 dbt models. Since your organization has invested in data for a while, many of these models feed into various dashboards and other data applications. Some of these data outputs are very important to your company. One of them is a reverse ETL that feeds data back into Salesforce, which helps your commercial team colleagues prioritize which accounts they reach out to tomorrow. Another essential use case is in *marketing. Your demand gen team uses customer CLTV data to feed look-alike models in advertising platforms. These use cases are critical.*

You've been asked to develop a new version of customer scoring. The analysts created a solid hypothesis on how new scoring can work and validated with stakeholders that new metrics are indeed meaningful and would be valuable to the commercial teams, but it needs some work; they only wrangled data in dozens of spreadsheets, but it's fragile and can't be used as is. That is why you come in to make it more robust. You will need to bring several new data sources into the data stack and write a series of models that process them. This will take at least a week and collaboration with data engineers.

In such a scenario, it's natural and expected that all this code starts incrementally landing into the dbt project, layer by layer. From this point, it may take weeks before this new pipeline matures enough to replace the current production system.

One day, you get a ping from your colleague on a weekly rota to triage incoming data errors, and some of your models are timing out. The other day, one of the data sources didn't refresh. The first problem was a mistake when excessive data arrived at the warehouse, which caused problems. In other words, the pipeline skipped one crucial task, and retry didn't work. You knew this could happen as data eng. The team is also working on the new pipelines. It's not a big deal; it's a work in progress. All this will be solved before the system gets to production, and the team is well aware.

And this is just your project. A dozen others are being developed in parallel.

It's essential to understand this dynamic.

"It's not uncommon for data platforms to feed into dozens or hundreds of final use cases that need to be operated with vastly different reliability guarantees."

In some cases, the specific use case is a work in progress. In others, it's an experimental dashboard one of the product analysts put together for their team. In another case, it's a daily pulse one of the teams in operations sets as a reminder of something nice to know. In contrast, you might have use cases such as an automated marketing pipeline deciding how to allocate hundreds of thousands of dollars monthly. All these use cases are intertwined.

The result: Alert overload

Combine all above and we've built a solid recipe for alert overload. Many teams are drowning in large volume of hard-to-action, unclear, or unimportant alerts.

This nicely highlights the challenge: We need a better, more strategic, approach. The best approach to testing will work with several key, to a degree contradicting forces:

- Minimize the number of alerts received to prevent alert fatigue.
- We must minimize the number of tests and monitors we create and execute to reduce overall complexity, cost, and alert fatigue.
- Maximize the chance of detecting issues.

Testing software well is an art and skill that software engineers spend years mastering. Data teams are no different. However, the benefits are clear: well-done testing means more reliable systems and more robust data.

There are several vital considerations to make.

Designing a Testing Strategy

Testing pipelines, not models

The above example could help us inspire an alternative way of looking at our data platform. Instead of seeing it as a network of interconnected models with dozens of outputs, we can start from the end use cases and data products and expand them into pipelines.

A data product pipeline is a direct acyclic graph of its upstream dependencies, including all models, tables, and pipelines that move data from its source into the product.

Consequently, the pipeline can be seen as an end-to-end horizontal slice through the entire data platform, traversing all system layers. This approach brings several benefits, especially if we've done diligent work on the definition of data products and their severity:

- **Severity can back-propagate**. By definition, the pipeline of models feeding into the P1 critical data product has to be treated to the same degree. Otherwise, we will create misaligned incentives. The downstream team is held up to a P1 priority while someone else upstream is not; that doesn't work. Severity back-propagates.
- **More apparent impact assessment**: Failures identified within the data product pipeline can be directly linked to the product. This can significantly improve

communication across teams, as the pipeline will be expected to run through data assets owned by different teams.

- **Feedback on architecture**. Since the content of the pipeline carries the severity of the end use case, we could use pipelines to understand how much use cases are tangled across different priorities (P1 to Pn). More intersections between them mean we increase the chance of work on the P4 pipeline, possibly unintentionally impacting the P1 system.

Another way to look at the centric approach is that besides various technical and workflow benefits, identifying the pipelines, especially the ones feeding into critical data use cases, gives us a lot of focus. As a result, we can evolve our approach to testing, focusing our energy and effort on pipelines that feed highly critical products.

—

However, despite the above benefits, the data pipeline-centric approach is rare in the analytics community. At SYNQ, we believe this is mainly because we don't have the proper tools to work that way. To establish this new approach, we need to solve the following:

1. **Anchor data product definition into observability** and catalogs and use underlying metadata, particularly lineage, to identify up-to-date dependency chains leading to the data products.
2. **Create pipeline automation workflows**. Once we identify the pipeline, we need better systems to govern it. We should be able to deploy monitoring and focus our testing on specific pipeline(s), as well as create workflows that help us improve the architecture of our system, making it easier to maintain large numbers of such pipelines in a single data ecosystem.
3. **Establish data product pipeline oversight**. Instead of looking at entire systems with metrics such as model test coverage, we will need to reason about pipelines by understanding failure rates, SLAs, issues, and incidents by pipelines.

Case Study: Establishing P1 Data Product pipeline for SYNQ user-facing analytics.

At SYNQ, we operate a data platform with hundreds of tables and models and face similar challenges as most data teams: our data platform, ClickHouse, feeds a broad mixture of data use cases, from P1 user-facing systems to exploratory analysis of feature adoption in our product (P4).

Our latest pipeline was for user-facing analytics that we expose to customers to help them understand quality metrics such as the percentage of failed tests, time to resolution, and the number of issues they can analyze by data products or teams.

As a consequence of being customer-facing, this pipeline is P1, e.g., its highest importance, and we treat it as any other production system that is powering our user-facing product.

The end product is a React.js application that interactively exposes data to our customers. Under the hood, it queries multiple ClickHouse Parameterised Views that act as an interface to the FE system. This view leverages data from 4 different source systems (SYNQ micro-services) to calculate the final metrics.

To create clarity, we have defined final views as a data product that anchored this use case into our observability approach.

With the following definition, we've used lineage to identify the pipeline.

Understanding our pipeline would allow us to start reasoning about our testing in a particular way—ensuring that data that arrives into final views is robust and reliable, with little room for error.

Testing Layers

With an approach to testing anchored around data pipelines, we've created a lot more clarity and focus on specific models that we want to test to an elevated degree since they feed the P1 product. This is beneficial for the particular pipeline, but without additional concepts, it could be dangerous in the long term.

As we discussed earlier, it's widespread and reasonable to expect data product pipelines to be interconnected. However, developing a unique and specific strategy for every pipeline could also be complex and infeasible.

This is why we think about testing in layers.

Data platform layers are a standard mechanism for tackling their complexity. We often hear about raw, staging, mart layers, or medallion architectures with bronze/silver/gold layers.

"For data architecture and modeling, the data platform layers help us decide what kind of transformations to apply where. Testing is the same; we want a testing strategy outlining what type of testing to use at every data platform layer."

We do so with several objectives:

- Alignment around what to expect. Layers literally' slice' the big testing problem into smaller parts, making it easier to reason about what kind of testing we can expect on data assets.
- We are setting a common standard. We classify the decisions by strategy, removing some from everyday work. By deciding what to test at each layer, we align the team.
- We are removing redundancies. Retesting for the same context, like retesting that customer_id is not_null in every single model across layers of the transformation, doesn't bring more reliability, generates more cost, and possibly contributes to alert fatigue. We want to eliminate redundancies.

The golden rule to strive for (the same as we do when testing in software engineering) is as follows:

"For any potential failure mode of our data, we want the minimum, ideally one, test or monitor to fail."

Let's say we have five customer records with customer_id NULL. This is unexpected. But if more than one test fails, it's a code smell. It contributes to clutter in alerting, adds more code to maintain, and reduces clarity. One test should have failed.

Defining the testing layers

Testing layers should be tightly coupled with architecture. In another way, design testing into layers that fit your architecture design.

At SYNQ, on the highest level, we slice our data platform into the following layers:

1. Pipelines start with the data source (operational system) and end when data enters the analytical system. This is where you transform the data, typically in the first layer of tables in the data warehouse. Pipelines often contain ETL pipelines, raw data lakes, or streaming systems.
2. Sources are the first layer of data that enters the data warehouse. The source is typically the first layer of data where we can apply transformations in SQL and implement tests from dbt or similar analytics engineering tooling.
3. Transformations are layers of data transformations where we clean up the data, model key business concepts, and create data assets ready for analytics use cases, typically our core data models.
4. Marts is the last layer of data, where data leaves the platform into a given use case(s).

The above model is generic and should apply well to most of today's typical data platforms.

The Strategic Approach

We've finally set enough foundations to move on to the meaty part of the testing and monitoring strategy design: Deciding what types of tests to apply and where.

We will work with the following fundamental principles:

1. Tests and monitors are designed to complement each other. We decide what failure modes we test within each layer of our data platform and pipelines.
2. We minimize redundancy. We avoid putting similar or identical tests into a sequence of models in the DAG to ensure that only one test fails on failure.
3. We combine data testing and anomaly monitors into a cohesive testing strategy that avoids duplication.

Testing Sources

Let's start with the definition.

What do we mean by a source?

The source layer of a data platform is a collection of data assets ingested from third-party systems like CRM, ERP, company products, or other non-data teams like engineers and business operations within or outside the company.

"In other words, it's the layer of data that is under the control of the data team for the first time."

In many organizations, this is directly linked to a concept of dbt sources, as we frequently load data into data platforms from ingest pipelines or cold storage systems, which are primarily not used for analysis to start the modeling activities inside of dbt.

While this definition is practical and applicable to most data teams, some teams have additional pre-processing steps before dbt, and therefore, their data sources are more upstream.

A practical guideline to decide what our sources are would be as follows:

"Find a layer of data as close as possible to the source, where you can control the data and add monitoring and tests without dependency on other teams."

dbt sources or the first layers of models in dbt perfectly fit this definition. But so would an upstream platform with well-established testing tooling such as great_expectations.

Why invest in testing sources?

The above definition of data sources as the upstream layer in the data team's control has a reason. A source layer of data that fits this definition essentially becomes an interface.

It is an interface with many exciting properties, which are very helpful for testing:

1. **It separates systems built predominantly by the data team from other systems**. Everything downstream of sources is data models and other analytical assets. Most of the business happens upstream.
2. **Detecting issues at source means detecting issues from upstream**. Since we test data that arrives at the data platform as upstream as possible, we are not testing any login within the models. We are testing assumptions about the input data to the analytics system, which means we are testing the consistency of data that arrives from the source.
3. **It aligns well with ownership**—downstream from the source, data analytics teams own it, while upstream is owned by data engineering and operations and engineering teams across the business. Well-tested sources could be a great mechanism to engage the upstream team in data quality.

Sources are input we must work with throughout the rest of the pipeline. They are the most upstream assets within data analytics modeling, and so failure to detect errors at this layer often has broad and severe consequences.

"Testing sources is a high-leverage activity. We are verifying the quality of data that feeds into every other model in our system, which has the most significant number of downstream dependencies and attributable usage."

This definition is also a reason for the following advice, which we apply internally at SYNQ but also advice to our customers:

"When in doubt about where to start with testing, start with sources."

We will combine this advice with our shifted focus on testing pipelines to create a practical prioritization framework for quality improvements.

What does a well-tested source look like?

Previously, we've demystified a common technique: put unique and not_null tests on the primary key and consider the source (or model) well-tested. Such a strategy promotes a setup where most data flows through the system untested, creating a false sense of security.

This is particularly important at the source.

One critical decision we will make is what kind of tests we want to apply at the source. To some degree, this is generic, but you should always consider your context critically, especially from the risk perspective: What is the risk of not detecting an issue at the source?

We follow with a list of typical data issues we want to prevent at the source, outline what types of tests are most applicable, and provide a rationale for why and how we should set them up.

Testing structure

data comes from the business in many shapes and forms, and given that the business tends to evolve, and so does the data structure, it is essential to verify that the critical expectations we rely on are met.

When reasoning about how deep to go, especially when the criticality of the use cases for the data being brought to the data platform is not fully known, it's worth being defensive and testing sources well so they can support use cases with the highest criticality.

"Another way to assess the depth of testing sources is to look at the data source from the business perspective"

How central is this data to our business?

I'll give a few examples that bring this advice to life:

- In fintech, data sources that represent transactions

- In e-commerce or logistics, data sources that represent orders or shipments
- In SaaS or infrastructure, the data that represents usage and engagement
- In Marketplaces, the data that represents demand and supply

These sources will likely be central to the respective data platforms. Of course, this is a small example to illustrate the principle. Think about how key every source is to your business domain, which will likely correlate with the criticality of use cases built on top of it.

With that in mind, let's dive into the testing itself. For well-tested sources, we, in general, want to have the following in place:

- **Data is present and complete**. We are mainly focused on detecting not_null and empty values. This can be well done with dbt and similar frameworks, as these tests are readily available and widely used.
- **Ensuring uniqueness**—duplicate data is one of the most common ways data can go wrong. Applying unique tests from dbt is a way to prevent issues. Besides the built-in unique dbt test, there are also several extensions, such as unique_combination_of_columns or even conditional unique tests, that could test for more nuanced business scenarios around unique data.
- **Testing values**—for low cardinality fields, it's worth verifying the expected (or blocked) values to catch deviations early. We tend to recommend an approach with explicit enumeration (list of accepted_values which comes with dbt Core) because the data team gets explicitly notified about new values, which often have to be handled in downstream models as otherwise they might get rolled up into 'other' logic branches or even completely unhandled, causing issues.
- **Testing format of values**—For fields with higher cardinality where it's infeasible to enumerate the values explicitly, it's worth thinking about testing the format of the values in another form. For numeric values, this could be done via min/max logic; for test values, this could be a regular expression for the expected format (like birthdate or email). These tests are beneficial in detecting corrupted records at scale early.
- **Monitoring ingests of data**—one of the most common sources of issues in analytics systems is the broken flow of data, typically as an unintentional side-effect of change upstream. Multiple layers of ensuring that consistent data flow happens are worth breaking down further:
- **Freshness testing** is an excellent tool for scenarios where we want to ensure complex SLAs regarding the timeliness of the data. For data sources that we now have to be refreshed at least every period, a deterministic freshness test (like the dbt source freshness test) is the right tool.

- **Freshness monitoring**—in comparison, many data sources are not as specific in their update frequencies. Further, setting up hard thresholds on hundreds or thousands of sources is impractical. Freshness anomaly monitoring is a viable alternative, as the threshold can be learned from historical data patterns. This approach is superior in case freshness has seasonality or pattern that is hard to express in simple freshness tests, like scenarios when the data pipeline gets quiet every night or every weekend. Freshness tests would either alert too frequently (and cause alert fatigue) or have to accept considerable delays to account for the most extended gaps of inactivity. These criteria could be used to decide if a test or monitor is a better solution.

- **Volume monitoring**—ensuring a solid volume of data is even more firmly in the anomaly monitoring domain. Maintaining expected thresholds of increments of data is very impractical and tedious. Especially given typical evolution of business, which grows and changes, data volumes tend to fluctuate much more than the frequency of loads (freshness). Therefore, anomaly monitoring that can further model seasonalities is a more feasible option.

- **Monitoring deeper**—the final consideration is whether tracking data on the table level is sufficient or if deeper monitoring is required. Consider a scenario where the data source in the platform is a fan-in pattern of many underlying data sources—like hundreds of events tracked from a website. A single event missing would most likely be unnoticed by all other solutions — tests, volume, or freshness monitors set up at table level will not be able to pick up nuanced changes where perhaps less than 0,1% of data stops flowing. Yet this could still be 100% of typical data for a specific website traffic event that broke due to a faulty website update. In critical data sources, deeper monitoring that works on segments of data (volume monitoring per event) is a great tool.

Does this sound like a lot? It's indeed more than 'one test per model,' but this is what it takes to test sources well. I've previously mentioned that mechanical testing is problematic as we are not intentionally designing test suites. But I'd like to nuance it from this point.

By defining a testing strategy for sources like the one above, we have applied strategic nuance—we've intentionally decided how we will test sources, deciding what tests we use (and what tests don't). But with a chosen strategy, we can roll out the above setup at scale, to a degree with automation, making the daunting task of writing all these tests much more approachable.

The role of data contracts

At this point, it's worth reflecting on the relationship of source testing with another concept: data contracts.

"A data contract is a document that defines the structure, format, semantics, quality, and terms of use for exchanging data between a data provider and its consumers. It is like an API but for data. —https://datacontract.com/"

This definition is essential as it clarifies that well-tested sources and data contract definitions are complementary techniques. Data sources are very reliant on source testing. This is because data contract specification and tooling focus on an explicit definition of data contracts, including the definition of quality—which might include a definition of data testing—but data contracts themselves are a description. They don't enforce the quality itself.

With that in mind, teams that already use data contracts should integrate testing data sources and implementing data contracts into a single workstream. A data contract can act as a prescription for what should be tested while tooling such as dbt or SYNQ can act as an execution environment where these tests get provisioned, executed, and tracked.

Testing Transformations

It's precisely because we are still applying a pipeline-centric approach to testing that the volume of tests we will be adding going forward will start to decrease. In other words, we've done a lot of testing at source, which we don't have to repeat.

"With a well-established testing strategy for sources, transformations can have much lighter testing suites, where we focus only on what has changed."

This fundamentally differs from model-centric testing, where we look at each model in isolation. Instead, going forward, we consider the tests we've already established upstream (at source) and fill the gaps.

Several key types of transformations are essential to test because they tend to be prone to errors:

1. Data cleaning and normalization—is often simple logic that is easy to test
2. New combined/derived columns—when new columns are created due to SQL logic that transforms or combines multiple other columns, it's a new business logic. This could be anything from simple transformations, such as recoding numerical codes into more user-friendly values, to complex CASE / WHEN statements that could introduce a big chunk of business logic.

3. Table joins—which are a persistent source of errors caused by duplicate joining variables.
4. Data aggregation—where we create new groups of data typically by using GROUP BY clauses or analytical functions
5. Data Structuring and Reshaping—where data structure in the table is fundamentally regrouped, typically using groups by clauses or, in some cases, more advanced concepts like pivots.

Given each model in our ecosystem could be a different combination of the above, it could be helpful to define testing techniques for each type of concept, which can then be combined into a final testing suite for the given model.

Data Cleaning and Normalization

Cleaning and normalization are often very lightweight data transformations; therefore, testing them could be equally lightweight.

Cleaning and normalization code is also very frequently a source of redundancy. It's because lightweight transformations are very unlikely (or in some cases impossibly) a source of data errors.

Think about simple changes such as normalizing column names:

```
select
  id as issue_id,
  workspace as tenant,
  toDateTime(created_at) as started_date
  ...
from
  {{ source('incidents', 'issues') }}
order by
```

We could retest `issue_id` and `tenant` or `started_date`, but such tests could be redundant to the testing we have already done at the source.

Such examples can be tests we've done to verify that the upstream id is not null or unique, that the workspace is always there and has correct reference to the workspaces table, and that created_at exists; all these tests significantly contribute to the reliability of this downstream model, too. For example, the upstream id uniqueness test guarantees that issue_id will be unique, too.

Eliminating such redundant tests has several key benefits:

- It is easier to reason about—we have a more unified place for each type of test
- It's less likely to generate redundant alerting—especially if you don't run a dbt build or these tests have WARN-level severity. Otherwise, you would get more failed test results than you need.
- It's less compute—each test is an additional query to your data, and therefore, eliminating redundant tests also eliminates redundant traffic.

Case Study: Removing redundant uniqueness tests to save 10% of warehouse cost

At the start of 2024, we engaged with a customer looking for ways to save costs on their data platform. They have done an excellent job with models, switching many to incremental materialization or optimizing their underlying SQL structure, but they forgot about testing.

After a quick analysis, we identified a sequence of 3 tables in their DAG, each testing for the uniqueness of the primary key (id) on a large table. The test was verifying the uniqueness of more than a billion records hourly and to make matters worse. These tests were not just slow on their own but also redundant; just one test would create enough safety. By removing redundant downstream tests and changing their frequency, we reduced data warehouse computing cost by 10% in a single commit, saving tens of thousands of dollars per year.

Newly derived columns

Newly derived columns could be looked at through a simple conceptual lens.

"A model that creates derived columns is their source; therefore, we can apply a playbook from testing data at the source to test derived columns."

In other words, derived columns are worth testing deeply, proportional to the complexity of the logic applied to create them. We use various typical data testing methods, such as uniqueness, non-null, accepted values, etc. We should always remember the logic and consider where it can break.

For example, consider the following logic:

```
CASE
  WHEN … THEN ‘active’
  WHEN … THEN ‘pending’
  ELSE 'inactive'
END AS status
```

Testing such a column for `not_null` values in many scenarios doesn't make sense as we implicitly ensure some by the `ELSE` clause.

Derived columns often contain much more complex logic, which could combine data from many columns that might come on top of join across multiple tables. In such scenarios, bringing a technique we haven't discussed might be beneficial: unit testing.

“Unit testing differs from data testing as it creates a complete testing cycle, from seeding the correct data to executing the logic and asserting the output.”

This makes it a great tool of choice when testing multiple scenarios where we seed/test/assert different setups. Doing such testing on transformations that power-derived columns could be an efficient and not-so-complex way to introduce unit tests to your quality strategy.

But keep one crucial thing in mind.

“The maintenance cost of unit tests is higher than the maintenance cost of data testing.”

When we decide to change the SQL logic, we must update unit test expectations for all scenarios. It's a natural trade-off: the more diligent reliability guarantees we get, the more work we must do to maintain the tests.

It's essential to balance robust test coverage and ease of maintenance. Every unit test should have its purpose and be typically introduced in scenarios when more straightforward data tests are insufficient.

Joins

Joins are notoriously error-prone, typically due to unexpected duplicates in any of the joining variables.

The scenario is simple: we join two tables on a variable that is — unexpectedly — not unique. As a result, we duplicate rows. This problem is called 'fanout'.

It's why testing models with joins is particularly important, especially with the right types of tests that specifically aim to detect the 'fanout' problem.

At SYNQ we frequently use one of the following strategies.

Row count validation

In cases where we're joining data without aggregating or filters, we can apply the row count check.

Consider the following scenario

```
select
  issues.id as issue_id,
  incidents.incident_id as incident_id,
  ...
from
  issues
    left join incidents on issues.incident_id = incidents.id
```

The **issues** table is a base table of the join. In our domain, one **issue** can be part of no or one **incident**. This means that under no circumstance we should join multiple items. But it would be naive to assume that this important business assumption is always met. This is why we need a test.

Given we've constructed our join mainly to enrich **issues** with additional data from **incidents** without additional logic we can write a simple check. Number of rows in our new table should match number of records in **issues** table. In other words we verify that no additional rows were created.

One of the benefits of the row count test is that we can execute it as comparison of `select count(*) from orders` and `select count(*) from orders_with_incidents`. Executing such query is typically cheap, as data platform can use underlying metadata potentially without scanning any data from the underlying storage.

Verify cardinality of join keys

Alternative approach is to verify cardinality of columns involved in the join. As we discussed, the assumption is that an **issue** can only be part of one **incident**. This could be direclty verified by one of the tests in **dbt_utils**:

```
# models/issues.yml
version: 2

models:
  - name: issues
    tests:
      - dbt_utils.cardinality_equality:
          compare_model: ref('incidents')
          compare_column: id
          column: incident_id
```

This test will fail if the number of distinct values in **incident_id** is different from the number of distinct values in id in **incidents** table.

The above test has a potential to be even more accurate way to prevent the fanout problem, but it has a caveat. Compared with row count test it has a lot more complex SQL logic. This might not be a problem for small models, but it might be a challenge if you're dealing with large number of rows.

Aggregations

Aggregations are another essetial building block of data products. Whether it's a simple count of rows or a complex aggregation, aggregations typically combine data from multiple rows into more meaningful measures.

Data models with aggregation logic have a number of interesting testing challenges:

- Aggregations change the number of rows in the table, typically aligned with granularity of grouping.
- Aggregations create new columns that represent aggregated measures.

- Advanced concepts such as window functions can introduce very complex and hard to test logic.

This means that correctly verifying aggregations might be a complex task, which we can approach with number of techniques.

Testing Products

The final layer is frequently called data mart which describes a set of datasets that are purposely assembled for a given set of use cases. The exact structure varies from company to company. In some cases marts align around functions such as marketing, sales, product. In other cases marts align around purpose of data like fraud analytics, customer health, etc.

Whatever organisation you choose the marts should have one thing in common — they should contain data products that are ready to be consumed from processes in the business. This also means that we ensure that they work reliably, with testing.

To test data products, we can shift the approach again. Since we've done the work of methodologically working through the layers of the system and tested reconciliation with operations, sources and transformations well we no longer need to test the basics.

"Instead of focusing on technical aspects of data, when testing data products, we verify the logic encoded in SQL transformations."

Beyond unit testing — Integration testing in Engineering

Testing software well is always a challenge. This is no different in data and other software systems. This is also why over time the software industry developed a whole range of testing techniques — unit testing, integration testing, contract testing, behavioral testing, end-to-end testing, smoke testing etc.

In software, unit testing is very valuable tool that complements the development process. We build functionality with unit tests in small increments and continue enhancing our tests as we build our software. The flow goes like this:

1. *Write a test that defines input and output of a new unit of software*
2. *Implement the functionality, ensuring that the tests pass*
3. *Repeat*

If you follow this process rigorously, its called TDD (test-driven development). I personally don't think its worth following religiously. Sometimes its better to code first, sometimes its better to start with a test. The important bit is that we consider writing tests an integral part of building software.

But unit testing has its limits.

Most software consists of large number of units (functions / modules / services). It has many inputs and outputs which interact in non trivial ways. To test all units in isolatation is only possible if we mock inputs of each tested unit. This is often impractical and potentially very fragile. It's because mocked interfaces that drive our tests tend to change and mock that mimics changed interface is a faulty interface. Our tests might continue to confirm our software units work as expected, but the system as a whole doesn't work correctly anymore. Once units are deployed to production together, the system fails, at the interface.

This is also why software engineers use whole range of other testing techniques beyond unit tests. One such testing technique is called integration testing.

In integration tests we don't focus on granular units of software, but as the name hints, we focus on testing the units together, integrated. The integration test is therefore intended to verify large pieces of software together. In some cases it could be large modules with many units, in other cases we can integration test functionality of the entire system altogether.

Integration tests of course also have their downsides. They are often slower and more complex to setup, as they test larger system components, which might involve setup of databases and other heavy software components.

The key is to combine unit and integration tests well. They both have their function in the testing strategy and work the best together.

While integration testing is not really formalised and widely adopted in data platforms and SQL systems, we can take the inspiration from this approach and apply it in our data product tests.

“Since final data products are directly linked to a specific use case, we can verify the accuracy of our data system by mimicking queries specific to the use case.”

In other words, we run real-world queries and assert that response we got is correct. This type of testing could be seen close to integration testing. We are not mimicking any interfaces, we are querying data from the final layer of transformation, which will be only correct if all upstream models function correctly too.

Regression and business logic tests

Beyond generic tests dbt ships with so-called singular tests. Singular test is simply a SQL query that should isolate faulty data.

In the context of our incident management system that produces information about mean time of resolution of issues we could write integration test that verifies that under no circumstances our system can return time to resolution of any issue that is negative.

Conceptually it could be as simple as:

```sql
select * from incidents where time_to_resolution < 0
```

Another key metric we produce in our final data product is measurement of percentage of failed tests. For example if for any reason our logic would count more failing tests than the tests that are actually running (which is data joined from multiple sources), we would want to know about it.

```sql
select
  workspace,
  toStartOfDay(created_at) as day,
  pct_test_failure
from incidents
where pct_test_failure < 0 or pct_test_failure > 100
group by workspace, day
```

We've created this test after identifying logical issue with some of our calculations, which under some circumstances would return invalid failure rate. We've for example counted that 10 out of 7 tests have failed, which is clearly not possible.

Tests like above are very useful to catch such logical issues that can appear anywhere in the data system. The test the system from outside–in. In some way, mimicking the queries that are actually run in the business.

As outlined another good use of these tests is to verify regressions.

"In software engineering, when issue is reported the engineers often spend considerable time to replicate the issue. This means thay can replay the faulty scenario and write a test that will fail. This could be done with these data product tests too. This approach would act as final confirmation that the issue is fixed and as a regression guard."

Testing historical consistency

Almost all analytical systems provide information about history. Whether it is historical revenue, product usage or customer acquisition, we often analyse data over periods of time.

This attribute of analytics system could be leveraged for testing.

Suppose our analytics system has measured 10,000 resolved incidents. The resolved status is actually an attribute of incident that logically combines multiple fects. We could test that the number of resolved incidents is consistent over time by comparing live data from our system querying historical period with a historical snapshot.

"We could verify our business logic by testing historical consistency by comparing live data from our system querying historical period with a historical snapshot."

The test would be as simple as:

```
select
```

```
    concat('Expected 10000 resolved incidents but found ',
toString(countIf(status = 'resolved'))) as error_message

from

    incidents

where

    year = 2024 and

    workspace = 'acme'

having

    countIf(status = 'resolved') != 10000
```

If the historical consistency of this data is not correct, we would want to know about it.

Note: Tests for historical consistency need to be used with caution. They are excellent for catching issues that would unintentionally change historical data, but are also expensive to maintain in case when calculations are actually changing. In such case its difficult to find and verify the new correct asserted value.

Summary

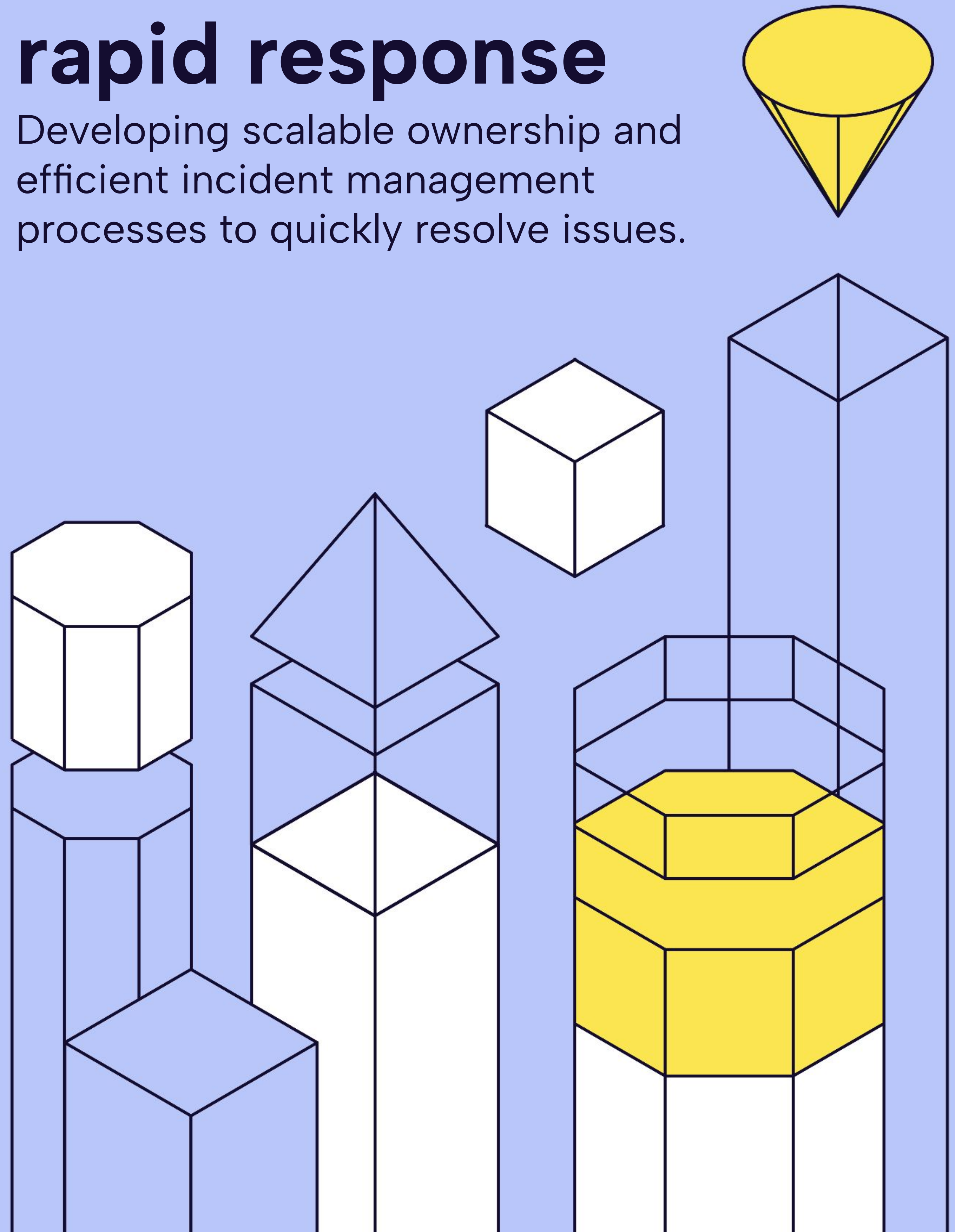
In this final section we've covered how to test data products. We've taken inspiration from software engineering and focused on end-to-end verification of our data products (and entire data system). This approach complements our previous work on testing sources and transformations and complementes them with final verification that our data systems work reliably.

SYNQ

CHAPTER 4

Ownership with rapid response

Developing scalable ownership and efficient incident management processes to quickly resolve issues.



As the data stack grows in complexity, it's no longer possible for one person to keep everything in their head, and more often than not, the person who notices a problem is not the right person to fix it. Simultaneously, the number of upstream and downstream dependencies has exploded, making it challenging to locate the right upstream owner or notify impacted stakeholders. Well-defined ownership and incident management processes can help with this by clarifying who's responsible, and how they're notified and streamlining the process around incident and issue response.

The ideal end state

If you're succeeding with your ownership initiatives you're likely to see one or more of the following impacts: (1) Time to resolution is reduced as issues are more often brought directly in front of the relevant owners, (2) time spent debugging and triaging issues are reduced as it's easier to locate the relevant owner, (3) upstream teams start taking ownership of data they produce improving the quality of data at the source and (4) teams start to systematically improve their domains as data quality KPIs are made more transparent

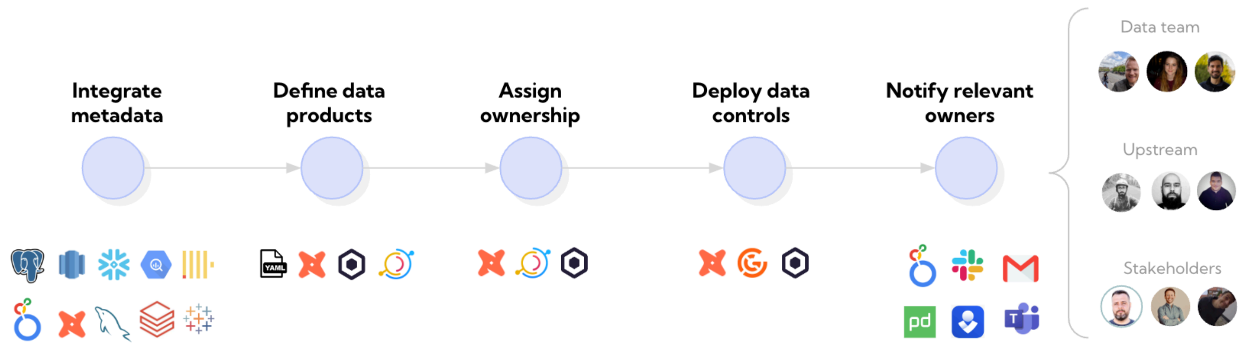
While there's no one-fit-answer to ownership and incident management, working through the four steps below will set you up well no matter if you're a 5-person data team or a data team in the Fortune 500.

1. Getting started with ownership
2. Defining ownership
3. Activating ownership by alerting the right people with the right context
4. Adopting Incident management in your data team for rapid response

Getting started with ownership

Ownership can be daunting, as it's both a technical and cultural challenge. If ownership works well, boundaries of responsibility are clear, and ownership is brought into action – both within and outside of the data team. If not, it's only sporadically defined and rarely actioned.

Whether you're just getting started with ownership or have existing owner processes in place already, we recommend thinking through these steps.



1. **Integrate metadata**–before you begin, you need a central place where you define ownership. In the most basic approach, this may be using a tool like dbt's built-in metadata. In a more sophisticated approach, this may be bringing all your data assets together – from source systems to data warehouse tables and BI tools – in a tool such as a data catalog
2. **Define data products**–start by identifying your most important data assets as data products. After all, that's where the stakes are highest and should be your starting point for defining ownership
3. **Assign ownership**–assign ownership based on responsible teams or individuals, ideally using existing ownership structures such as Google Groups
4. **Deploy data controls**–with ownership definitions in place, strategically place monitors based on the owners' domain knowledge of the data they own
5. **Notify relevant owners**–activate ownership through relevant alerting or escalations to incident management tooling

Defining ownership

In the ideal world, you'd neatly group your stack into well-defined areas with clear boundaries. But in reality, ownership lines can get blurry, so don't be discouraged if you can't easily assign ownership to all assets. Data rarely stops or ends with the data team. Instead, data is ingested from 1st and 3rd party data sources, loaded and transformed in the data warehouse, and exposed to end-users in a BI tool or use cases such as ML/AI.

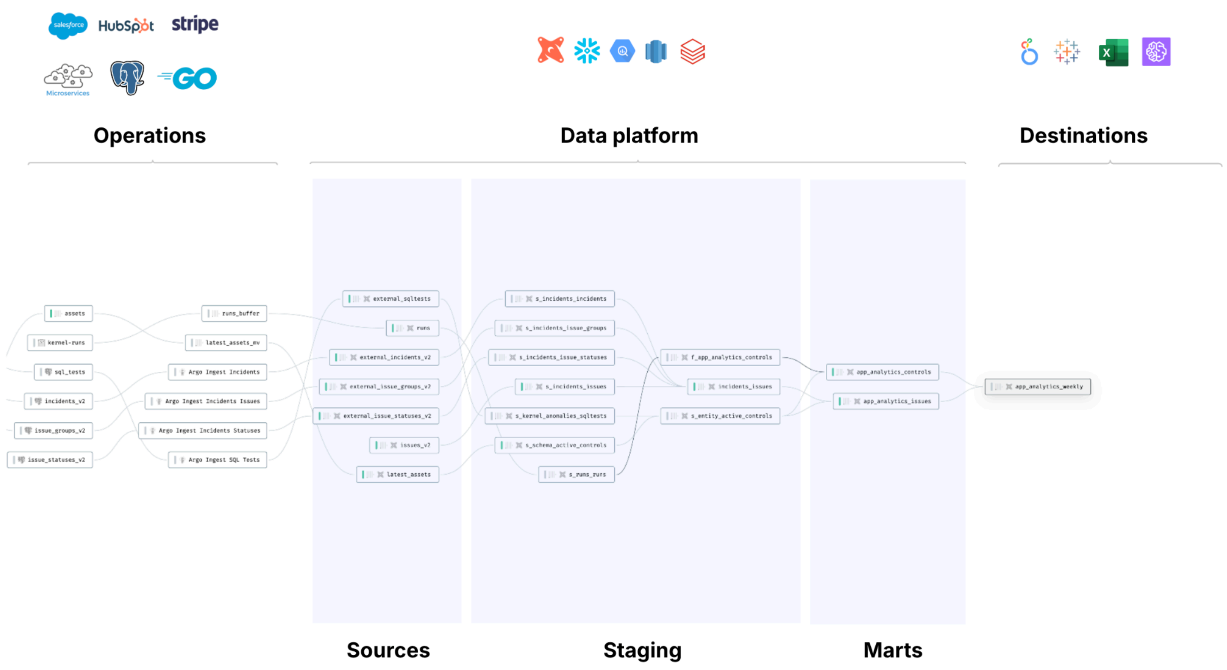
This is how we manage ownership at SYNQ:

- **Well-defined ownership at the input layer–**we ensure that ownership at `sources` is clearly defined so that escalation paths to upstream engineering teams are unambiguous, and so that they can be notified directly of issues on source systems before any data transformations are done.
- **Clearly defined boundaries on staging marts–**within the staging and transformation layer, our analytics engineers assign ownership of models based

on `dbt metadata` definitions using the `owner tag` and `dbt groups`. We enforce that ownership tags are set using CI checks.

- **Stakeholder ownership on consumer-facing marts–**for our consumer-facing data marts and products, we've organized them into `mart folders` based on their use case and assigned relevant owners such as our Technical Account Management team who get notified if there are issues with data they rely on.

"More than 50% of our data assets above are already encapsulated into data products making the ownership seamless to define."



The data lineage of an internal data product at SYNQ. Ownership can be mapped and overlaid across all dependencies

The result is that we manage ownership of data across the entire company and not just as something that's owned by the analytics engineering team.

Below are step-by-step instructions for how to define ownership.

Use existing owner groups

Use groups that already exist in your company, such as Google Groups or Slack team channels formed around existing teams. These will always be up-to-date as *people leave or join without you having to manage groups for data ownership specifically.*

Defining ownership based on data products

Data products should be your starting point–while you may not want to set ownership for all your data assets, we recommend you at least do it for important data. Having assets with clear ownership makes it more likely that the right people act on issues or are notified if the data they use is wrong. If you've defined data products with well-established priority levels, the highest-priority data products are a great place to start.

Ownership definitions will closely follow your data product definitions. For example

- The Marketing Attribution Data Product will be owned by *marketing data*
- The Users Data Product will be owned by *analytics engineering*
- The ARR Data Product will be owned by *finance data*

Case study: Ownership should be set at the right level

Set ownership at a too-high level and you risk that no individuals take responsibility. For example, by defining the owner as "data-team" you risk the definition is too broad to act on. Setting ownership on an individual level creates a lot of accountability but little scalability. You run the risk that people move around to different teams, go on holidays, or leave the company.

At SYNQ, we assign ownership based on teams and their associated Slack channels. Where possible, we use dbt groups so we only have to keep ownership metadata such as Slack channel updated in one place. As all alerts on input sources go to one channel, we also assign individuals based on sources they own, so that they're tagged in Slack to bring attention to these issues.

Defining ownership using metadata

Use existing folder structures to tie into your existing architecture design

Use this option if you've already organized your dbt project or data warehouse schemas to resemble your ownership structure, such as marketing, finance, and operations. With folder-based ownership, you typically need less time to get set up, and as you add new data models, they, by design, fall into an existing owner group, reducing your upkeep. If you work with non-technical stakeholders who don't contribute to your code base, such as business analysts or data stewards, this approach makes it easier for them.

Use dbt owner meta tags to manage ownership through code

dbt has built-in support for designating owners using the meta: owner tag. Owners are displayed in dbt Docs, making it easy for everyone to see who's responsible.

```
models:
  - name: users
    meta:
      owner: "analytics-engineering"
```

You can extend this to dbt sources to define ownership to upstream teams. If issues happen on sources before any data transformations, it indicates that upstream teams should own the issue.

An added benefit is that this approach lets you use CI checks, such as check-model tags from the pre-commit dbt package, to ensure that each data model has an owner tag assigned.

Use dbt groups to enable intentional collaboration

With dbt 1.5, dbt launched support for groups. Groups are helpful for larger dbt projects where you want to encapsulate parts of the internal logic only to be accessible to members of that group – similar to how you'd only expose certain end-points in a public API to end-users. If a model's access property is private, only owners within its group can reference it.

```
models/marts/finance/finance.yml

groups:
  - name: finance
    owner:
      # 'name' or 'email' is required; additional properties allowed
      email: finance@acme.com
      slack: finance-data
      github: finance-data-team
models/schema.yml
```

```
models:
  - name: finance_private_model
    access: private
    config:
      group: finance

  # in a different group!
  - name: marketing_model
    config:
      group: marketing
```

Defining cross-tool ownership

There are situations where you want to manage ownership across multiple tools – from databases to multiple data warehouses and dashboarding tools. This is useful when you want to find dashboards owned by specific teams or build out capabilities to notify downstream impacted stakeholders when you have a data incident. Managing cross-tool ownership in code can be difficult as there's often no coherent way to define this. Tools such as a data catalog or data reliability platform are built for this.

Notifying the right people

Data ownership doesn't have to stop with the data team. Below, we'll look at ways you can notify: (1) the data team, (2) upstream teams, and (3) business stakeholders.

"One of the top pitfalls we see is when teams spend a lot of time mapping out and defining ownership, but let it sit stale on a Confluent page that gradually gets out of sync with reality."

Managing alerts within the data team

Managing ownership within the data team is the most straightforward. Your team is in control; typically, the tools are within the stack you manage. You can use your existing ownership definitions to ensure the right owner knows about the right issue. The two most effective ways to do this, assuming you use a communication tool like Slack, is by tagging owners and routing alerts based on your ownership definitions:

- Tagging owners – associate owners with Slack handles to tag groups or individuals and drive awareness of issues.
- Routing alerts – tie Slack channels with ownership and send alerts to the relevant team's channel. This is a great way to overcome alert overload in the central channel.

"As a rule of thumb, you'll see the most impact once your data team moves past a handful of people, and everybody no longer has full visibility into all data assets."

Notifying upstream teams

We typically see two kinds of upstream teams that produce data and need to be alerted differently: technical teams, such as engineering, and non-technical teams, such as a SalesOps team owning Salesforce customer data.

a. Technical teams – the alerts you send don't need to look different from those in the example from the data team above. If you've placed tests at your sources and detected issues before any data transformations, engineers should be able to connect the dots between the source and the error message and trace back the issues to their systems. For larger teams with a clear split between teams that ingest data (e.g., data platform) and teams that produce data (e.g., frontend engineers), it can be helpful to compliment the error message with details about what event or service it relates to.

b. Non-technical teams – bringing ownership of quality of source systems to non-technical teams is underrated. Too often, tedious input errors such as an incorrect customer amount or a duplicate `employee_id` end up on the data team to debug, triage, and find the right owner. With the right context, these teams can start owning this without the data team being involved.

Case study: The cultural challenge of upstream ownership

One thing is starting to send alerts to an upstream team. But getting them to consistently act on them is another challenge. "Before we started to send alerts to our operations and lending team about faulty customer records, we got our COO to buy into the initiative. Only then were we able to ensure that the team was incentivized to act on and prioritize alerts from the data team"

Notifying stakeholders

Sometimes, you can alert your stakeholders to notify them of issues proactively with alerts similar to those you send to the data team. This works best if your stakeholders are data-savvy teams, such as a group of analysts in a business domain.

"Sending Slack alerts to your marketing director that five rows are failing a unique test on the orders data model is not the best idea."

The best way to notify impacted stakeholders is most often for the person with the relevant context to "declare" it unless the stakeholder is technical. In most cases, we recommend that the data owner notify the stakeholder directly and link to an ongoing incident page so they can follow along on the issue resolution. Another way is displaying issues directly in dashboards so end-users are aware of issues – but this can be more risky and difficult to interpret for non-technical users and recommend you only use this with caution.

At SYNQ, our Technical Account Managers rely on a 'Usage Report' data product to see the usage across our customers' workspaces. As our Technical Account Managers are technical stakeholders, alerts on or upstream of this data product are sent directly to the #tech-ops channel. This helps them be the first to know if data is unreliable and not make decisions based on incorrect information.

Beware of alert overload

If you spam a Slack channel with alerts, chances are people will stop paying attention to them. Preventing alert overload is solved throughout the data reliability workflow – from architecture design to monitoring and testing deployment, and alerting & ownership rules. Not all issues have to be sent to the #data-team Slack channel. A better workflow is to be deliberate with what's sent to the main alerting channel. Issues on less critical data assets can be sent to a different channel, or not sent as alerts at all, and managed in weekly backlog review.

Adopting Incident management

With adequate ownership in place, you're well-positioned to start streamlining responses to issues and adopting incident management for more severe issues.

If done well and combined with well-defined ownership definitions, this has several benefits–(1) Time to resolution is slowed as the issues are brought to the right owners with the relevant context. (2) Important issues are prioritized based on established incident management response processes. (3) Time spent resolving data issues is reduced as ownership lines are less blurred. (4) You start building an institutionalized way or adopting learnings from incidents and symmetrically improve.

There's no set way to manage issues and incident response, and you should always look to adapt to other ways of working in your companies. With that being said when adopting incident management for rapid responses in data teams, we recommend working through these five steps

1. Getting the data team on call
2. Detecting issues
3. Triaging issues
4. Handling the incident
5. Post-incident analysis

Getting the data team on call

Start by defining expectations from the data team for what it means to be on call. For smaller data teams, this may be the sole data person being the "data responder duty" for the week. For others, it may mean that relevant owners address issues within a predefined SLA as they come up. For teams owning core business processes, this may involve being paged or notified outside business hours, closely tied to the SLA definitions of your data products. We recommend you consider them across these groups:

- Overseeing data-related incidents (e.g., P1 data product errors or failures in dbt core pipelines)
- Managing smaller failures such as a dbt test warning to ensure they're addressed promptly and maintain their relevance

If the existing ownership activation rules you've set up are not working out of hours, you can create a "@slack-responder" group and only have the people on-call in it to avoid tagging the entire data team when there are issues out of hours.

Be explicit about on-call expectations

If you're not specific about expectations for on-call (e.g., we only look at issues within business hours), people will start adopting different expectations which can create an uneven workload across the team.

At SYNQ, our data platform powers core in-product functionality such as our

Quality Analytics tab. If there are issues here, we strive to detect the issue within an hour and resolve it within a few hours.

Detecting issues

An incident begins when something goes wrong—whether it's a critical dbt job failing, a table no longer receiving new data, or an SQL test for data integrity breaking down. At this point, it's still just an issue, and you should be able to achieve these three objectives:

1. Detect the issue promptly through predefined tests and monitors
2. Alert the appropriate person through the ownership model you've defined
3. Provide relevant context for resolution

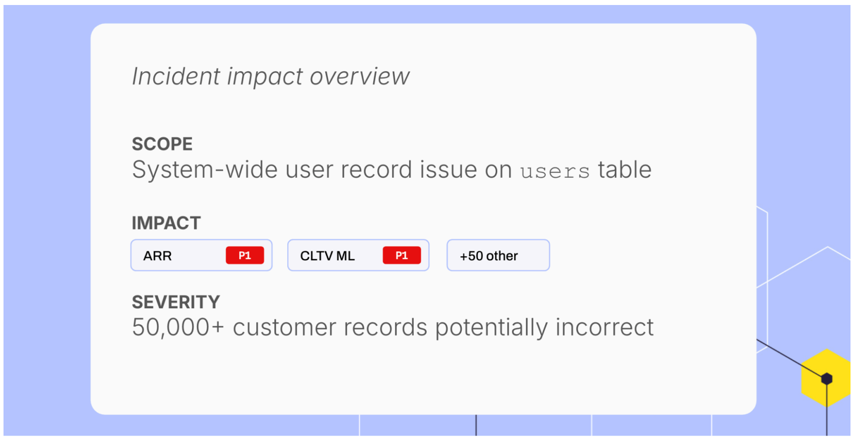
"Without sufficient testing and monitoring in place, you'll be caught on the backfoot, only learning about issues when you've critical system failures or issues detected directly by stakeholders."

The chapter Testing & Monitoring goes into more detail about how to set up the right monitoring to help you avoid this.

Triaging issues

When an alert is triggered, you should assess the situation thoroughly. The alert may not provide all the context needed, and a system failure might be connected to other relevant issues. At this stage, you should have an understanding of all other issues and their connectivity. Internally at SYNQ, we always aim to be able to answer three questions:

1. **Scope**–Is this an isolated failure, or is it related to other issues?
2. **Impact**–What is the potential effect on critical data products and assets?
3. **Severity**–Does this involve data being unavailable, corrupted, or unreliable?



Incident impact overview

SCOPE
System-wide user record issue on `users` table

IMPACT

SEVERITY
50,000+ customer records potentially incorrect

Example summary card–system-wide issue impacting critical data products

Answering these questions provides the context needed to evaluate the incident's urgency and decide whether it warrants a full incident response. This triage step helps separate critical issues from minor ones that don't require full incident management.

"An efficient triage workflow should alert relevant team members using your ownership definitions and offer a comprehensive view of failures and their context. This enables data engineers and analysts to assess issue severity and interrelations effectively."

Linking incidents to data products, or your other business-critical datasets is especially useful, as it highlights the potential impact of each issue and automatically identifies affected datasets, streamlining the triage process. This is a process that can otherwise be nearly impossible to do, especially if you've many hundreds of data assets downstream of an issue.

Once you're able to answer these questions, we recommend that you set clear expectations levels closely tied to your on-call setup. At SYNQ, we use the following benchmarks for our MTTD (mean time to detect) and MTTR (mean time to resolve).

Severity	MTTD (mean time to detect)	MTTR (mean time to resolve)
P1	1h	asap (hours)
P2	12h	1d
P3	24h	3d
P4	24h	7d

SYNQ's internal MTTD and MTTR metrics

If your data team or engineering team already uses an incident management tool like PagerDuty, Opsgenie, or incident.io, we recommend you link your data-related issues to these, so that an alert can be automatically linked to existing workflows, as well as bring on core metadata around the impact directly to the existing platforms.

Handling the incident

While there's no universal approach to handling data-related incidents, a structured and well-documented incident management process significantly helps make the response smoother and more effective.

When to declare an incident

Not all issues should be incidents but erring on the side of declaring too many rather than too few incidents gives you a tracable log of issues, where if something goes wrong, you can go back and look at what happened last time. You can always adjust the incident severity and prioritize it accordingly.

Creating a dedicated space for communication (such as a document, a Slack channel, or an incident in an external tool) ensures that stakeholders are kept in the loop and avoids having to switch between multiple apps and tabs, allowing you to focus your efforts on identifying and resolving the root cause of the issue.

At SYNQ, we use these steps for coordinating key communication and root cause analysis.

- Inform stakeholders promptly, focusing on the owners of critical data assets impacted.
- Organize data team efforts by clearly tracking who is responsible for each part of the issue.
- Attach GitHub pull requests to monitor the fixes made and their deployment status.
- Look for similar past incidents to learn from how they were handled.
- Document key steps taken and insights gained during the root cause analysis.

Each of these steps will be much easier with well-defined data products and ownership in place–you'll know who to communicate or escalate issues to, you'll be able to see if important data products are impacted downstream, and have a log to trace down previous related incidents, and how they were solved last time.

Post-incident analysis

It can be tempting to set an incident aside once resolved, but doing so may mean missing valuable insights and failing to establish processes to prevent it from reoccurring—or leaving someone else to grapple with a similar issue later on.

For lower-severity incidents, such as a test failure, follow-up checklists with assigned task owners can help ensure that guardrails are put in place to prevent recurrence. For more critical incidents, like a critical data product failure, a postmortem review can offer deeper insights and guide process improvements.

```
INC-123: Data Product Incident Title

ℹ️ Key Information                      ⏱️ Timestamps
+----------------------+--------------+
+------------------------+----------------+
| Data Product        | …            |  | Reported at            | …
| Priority            | P1, P2, …    |  | Impact started at      | …
| SLA                 | 99.9%, …     |  | Resolved at            | …
+----------------------+--------------+
+------------------------+----------------+

👪 Ownership & Teams                    ⏳ Durations
+----------------------+--------------+
+------------------------+----------------+
| Product Owner       | …            |  | Time to Identify       | …
| Slack Channel       | #team-name   |  | Time to Fix            | …
+----------------------+--------------+
+------------------------+----------------+

🪞 Related Incidents                   🔗 Useful Links
+---------------------------+        +---------------------------+
| INC-456                   |        | Incident Homepage         |
| INC-789                   |        | Slack Channel             |
+---------------------------+        +---------------------------+

🖊️ Summary
+--------------------------------------------------------------+
| Summary of incident impact on "Marketing Attribution" data   |
| product affecting downstream assets…                          |
+---------------------------------------------------------------+

📆 Incident Timeline
+---------------------+-----------------+--------------------------+
| Date        | Time   | Event                                      |
+------------+--------+--------------------------------------------+
| 2024-01-01 | 12:00  | Reported by Data Engineer                  |
| 2024-01-01 | 12:30  | Priority escalated to P1                   |
| 2024-01-02 | 12:00  | Incident Resolved                          |
+------------+--------+--------------------------------------------+

⬆️ Root Cause                             ⬇️ Mitigators
+--------------------------+             +-----------------------------+
| - Missing upstream checks |              | - Add source freshness      |
| - No quality assurance    |              | - Raise priority of source  |
|   in source system        |              |   data product              |
+--------------------------+             +-----------------------------+

👀 Risks                                ⏩ Follow-up Actions
+--------------------------+             +-----------------------------+
| - Low ownership of       |             | - Add review with core      |
|   source data quality    |             |   engineering team          |
| - Weak completeness SLIs |             | - Raise priority of issue   |
+--------------------------+             +-----------------------------+
```

Example template for a post-mortem of a critical data incident

Additionally, a periodic review of incident trends can reveal patterns—such as recurring errors in specific data assets—or highlight imbalances such as specific upstream teams being responsible for a disproportionally large amount of issues.

You should tie these metrics together with your wider data reliability workflow and be able to measure metrics such as

- **Mean time to resolution** – what's the average time from an issue is detected to it being resolved, broken down by severity
- **# issues and incidents by team** – are there specific teams or team members who have an outsized number of incidents. And how do these trace back (e.g., are

some source systems systematically triggering incidents outside the data team's control)

- **Issue to incident rate –** to identify if there are low signal data controls in place that could potentially be removed

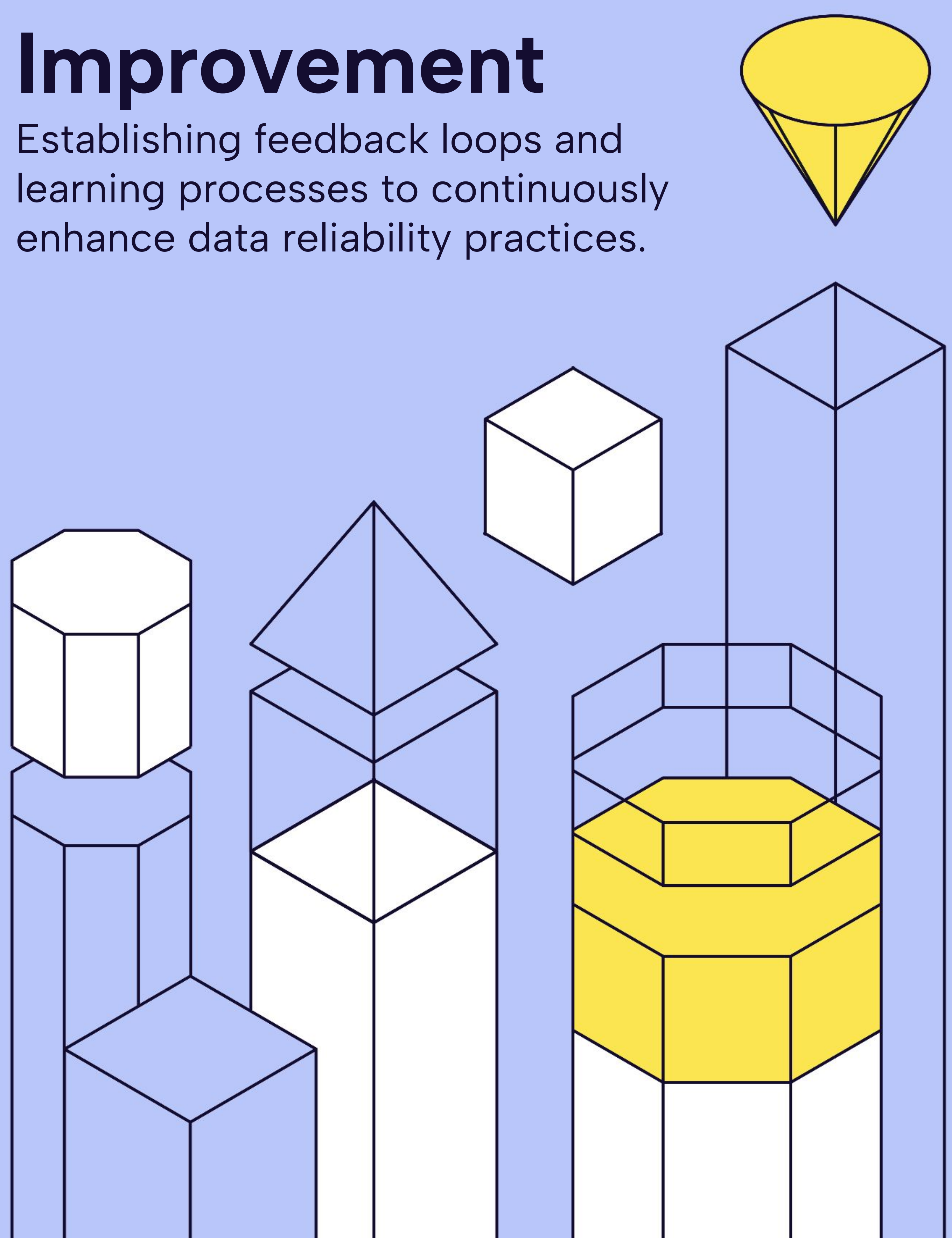
Read the next chapter to understand how you can establish feedback loops and learning processes to continuously enhance data reliability practices.

SYNQ

CHAPTER 5

Continuous Improvement

Establishing feedback loops and learning processes to continuously enhance data reliability practices.



While it's common for engineering teams to have established a set of metrics to monitor the performance, uptime, and velocity over time, it's less common for data teams. It is increasingly important to be able to report on the SLA, performance, and uptime of your data as you take on business-critical data products–but even if the main output of your team is dashboards and analysis for decision-making, it's still a good idea to establish benchmarks for when data should be ready and feedback loops for learning.

Here are some indicators that it's time to put metrics in place

- **Business-critical data** – Your team now owns data products like customer-facing dashboards where any downtime impacts customers directly.
- **Data quality perception** – You're hearing complaints about data "unreliability" or slow dashboard readiness without being able to systematically pinpoint issues.
- **Inconsistent data quality** – You're seeing inconsistencies across data teams and want to establish consistent, higher standards.
- **External accountability** – You need to objectively assess data quality and dependencies for regulators or external board members.
- **Low signal-to-noise controls** – You want to understand and improve the ratio of data control alerts that indicate real business issues.

Get buy-in outside the data team for your metrics

If it's only the data team that cares about the metrics you picked, you've likely picked the wrong ones. Get buy-in across stakeholders who are impacted by the metric and understand how they're impacted. For example, a product or account management team may be directly impacted if a customer-facing dashboard is down and contractually committed with an SLA towards the customers. Work closely with these teams on

Picking the right metrics

With your use case in mind, you should assess a list of metrics that you can track. It helps to group them into key areas–if your goal is to improve the SLA of key data products, focus on *High-level* metrics *and SLIs.* If your goal is to improve the usability of data, focus on *Usability*-related metrics.

"Measuring metrics such as test model test coverage without a clear end goal in mind can create a false sense of security and lead teams to optimize toward the wrong goals"

Metric Group	Metric	Description
High-Level	Coverage	% of assets with required data controls in place
	Quality Score / SLA	% of SLIs passing, calculated as (passed SLIs) / (total SLIs)
Specific Quality (SLIs)	Accuracy	Data reflects real-world facts
	Completeness	All required data is present and available
	Consistency	Data is uniform across systems and sources
	Uniqueness	No duplicate records exist within the dataset
	Timeliness	Data is updated and remains fresh
	Validity	Data conforms to required formats and business rules
Usability	Ownership Defined	% of assets with a defined owner
	Priority Level	% of assets with an assigned priority level
	Data Product Association	% of assets belonging to a data product
	Description	% of assets with a description
	Active Users	Number of users actively interacting with the asset
	Dashboard Load Time	Average time for dashboards to load
Operational Metrics	Mean Time to Resolution	Average time to resolve incidents
	Mean Time to Detection	Average time to detect an incident
	Number of Incidents	Total incidents impacting data products
	Number of Issues	Total issues logged (not escalated to incidents)
	The issue-to-incident rate	The signal-to-noise ratio for different data controls

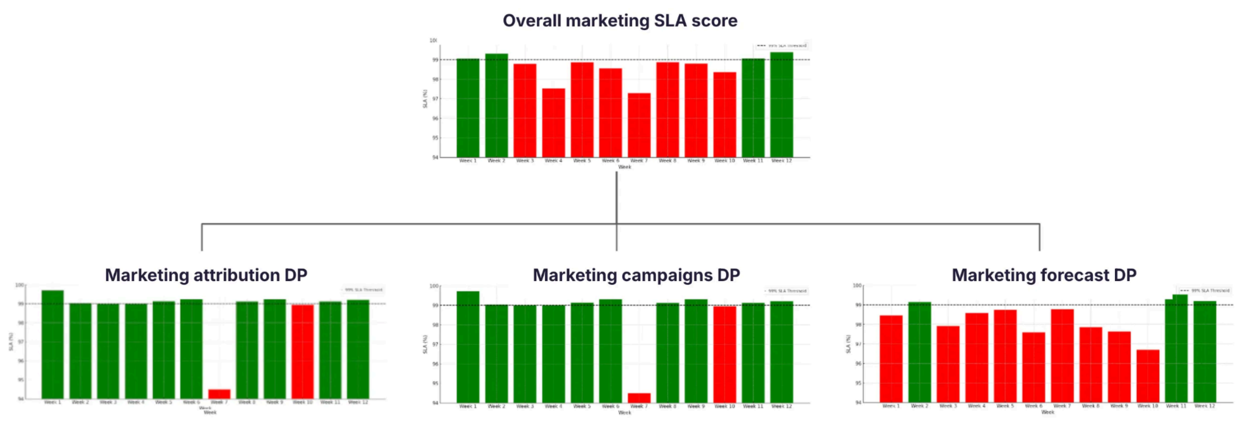
Consider the metric availability

If you're just starting, you likely have little to no metrics at all. Some metrics will be easier to get – for example, if you have existing tests and monitors in place, you'll be able to get the group these into SLIs. On the other hand, if you don't have an established incident management process in place, tracking incident mean time to resolution may not be the right place to start.

Selecting your North Star metrics

Start with just a few metrics based on the use case you have in mind. If you support a business-critical data product such as a customer-facing dashboard, you'll likely want to be able to track the coverage and quality score/SLA. It's important to consider both – if you're only tracking quality score/SLA without considering the coverage of data controls, you'll establish a false sense of security of the actual quality of the underlying data.

With this in mind, start decomposing your key metrics into dimensions. In the example below, you can see that SLA is only satisfied for 4 of the last 12 weeks. But this is largely due to the Revenue Forecasting data product consistently falling below the SLA target, giving you a good sense of where to focus and improve.



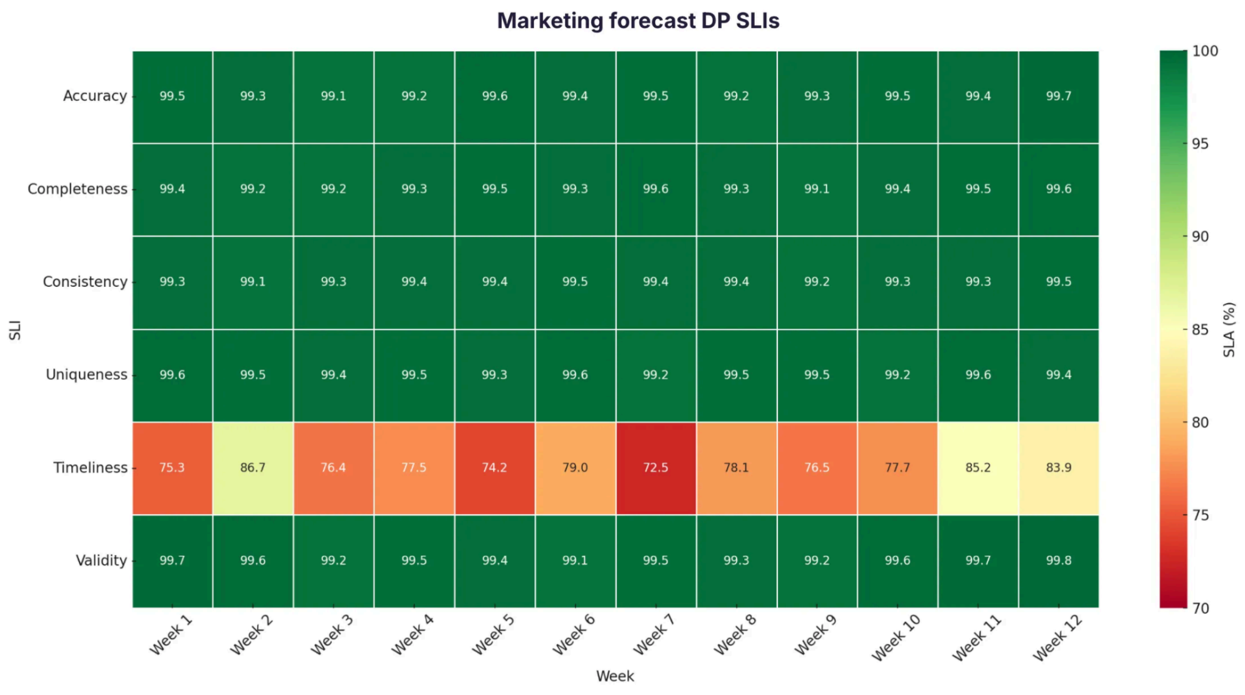
Establishing service level indicators (SLIs)

Think about SLIs as groups of data quality controls. By grouping the SLIs you can zoom in on if there are specific areas that are causing the SLA to fall behind. The six SLI areas we identified earlier provide a good starting point for grouping your existing data controls and are also the ones we've decided to use internally at SYNQ.

- **Accuracy**: Ensures data correctly represents real-world facts (e.g., `accepted_values` test for valid statuses, custom SQL checks for calculated metrics).
- **Completeness**: Confirms all necessary data is present (e.g., `not_null` test for critical columns, row count checks).
- **Consistency**: Verifies uniform data across sources (e.g., `relationships` test to check foreign key integrity, `unique` test across datasets).
- **Uniqueness**: Ensures no duplicate entries exist (e.g., `unique` test on primary key columns).

- **Timeliness**: Checks data freshness and update frequency (e.g., `dbt source freshness` test, custom timestamp lag checks).
- **Validity**: Confirms data adheres to formats and rules (e.g., `accepted_values` test for categorical data, regex-based custom tests for formatting).

With clearly established SLIs, you can go a step further to understand what's causing the SLA to fall behind for our Revenue Forecasting data product.



SLI	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12
Accuracy	99.5	99.3	99.1	99.2	99.6	99.4	99.5	99.2	99.3	99.5	99.4	99.7
Completeness	99.4	99.2	99.2	99.3	99.5	99.3	99.6	99.3	99.1	99.4	99.5	99.6
Consistency	99.3	99.1	99.3	99.4	99.4	99.5	99.4	99.4	99.2	99.3	99.3	99.5
Uniqueness	99.6	99.5	99.4	99.5	99.3	99.6	99.2	99.5	99.5	99.2	99.6	99.4
Timeliness	75.3	86.7	76.4	77.5	74.2	79.0	72.5	78.1	76.5	77.7	85.2	83.9
Validity	99.7	99.6	99.2	99.5	99.4	99.1	99.5	99.3	99.2	99.6	99.7	99.8

With these insights at hand, the next step is clear – you should focus on the timeliness of data, especially for sources feeding into the Revenue Forecasting data product to reach your SLA goals.

In our case, we equally weigh all SLIs as components to calculate the SLA. In some cases, you want to set different SLI levels for each SLI. For example, for an ML model, fresh data may be less important causing you to accept a 95% threshold in terms of times that data is loaded in time while you have a lower tolerance for completeness or accuracy issues, causing you to set the SLI target to 99.9%.

Tracking and obtaining the metrics

You may already have the data available to start measuring the key metrics, or you may be starting from scratch uncovering where data lives in source systems, or starting by defining processes to define the metrics.

As you build out the metrics, do it with the following four principles in mind

1. Metrics – select metrics that fit the business outcome you're optimizing for
2. Action – the insights your metrics provide should lead to action
3. Segment – metrics should be segmentable by key dimensions (owner, data product, …)
4. Trend – your metrics should be measured consistently and measurable over time

Below are some ways how you can obtain the data based on the tools you use.

High-level group	**Metrics**	**How to obtain**
High-level	Coverage, Quality Score/SLA	Export from dbt artifacts, dashboards from data observability tools
Specific quality (SLIs)	Accuracy, Completeness, Consistency, Uniqueness, Timeliness, Validity	dbt test results, dbt artifacts, data quality monitoring tools (e.g., SYNQ, Great Expectations)
Usability	% Ownership Defined, % Priority Level, % Belonging to a Data Product, % Descriptions, Number of Active Users	Data catalog exports (e.g., Atlan, Collibra), manual assessments, usage logs
Operational metrics	Mean Time to Resolution, Number of Incidents, Number of Issues	Incident management tools (e.g., PagerDuty, Opsgenie), internal ticketing system reports

"Internally at SYNQ, we've automated the SLA, coverage, and SLI tracking, so that we can monitor and report on the uptime on all data products at any given time. This helps make sure that monitoring uptime is not an afterthought, but instead something we review regularly."

The 2024 MAD (ML, AI & Data) Landscape gives a good overview of all tools and vendors across data and AI tooling.

Operationalizing insights

You'll want to put the insights you uncover from monitoring data quality into action. Whether it's to improve a particular area, share with stakeholders how you're improving, or something else.

While there's no one-fit-all solution, we've seen these work well.

Automated accountability with a weekly email digest – being the person having to slide into other teams' Slack channels to tell them that their data quality is not great is not always fun (we've been there). Scheduling an automated weekly email with the quality score over time and per owner domain and data product is a great way to bring accountability without one person having to point fingers (It does wonders when people see their team scoring lower than their peers).

Be religious about including metadata – the most common reason we see data quality initiatives failing is that everybody owns data quality, and thus, nobody feels responsible. Only by enforcing metadata such as data product definitions and owner or domain can you hold people accountable for data quality in their area. Build it into key processes such as using the check-model tags CI checks to enforce that certain tags are present.

Beware of the broken windows theory – the broken windows theory can be traced back to criminology and suggests that if you leave a window broken in a compound, everything else starts to fall apart. If residents start seeing that things are falling apart, they stop caring about other things. We can draw the same analogy to data quality.

If you've got many failing tests, it's often a symptom that the signal-to-noise ratio is too low or that you don't implement tests in the right places. Don't let failing data checks sit around. Instead, set aside dedicated time, such as "fix-it Fridays" every other week, to work on these types of issues and remove data checks that are no longer needed.

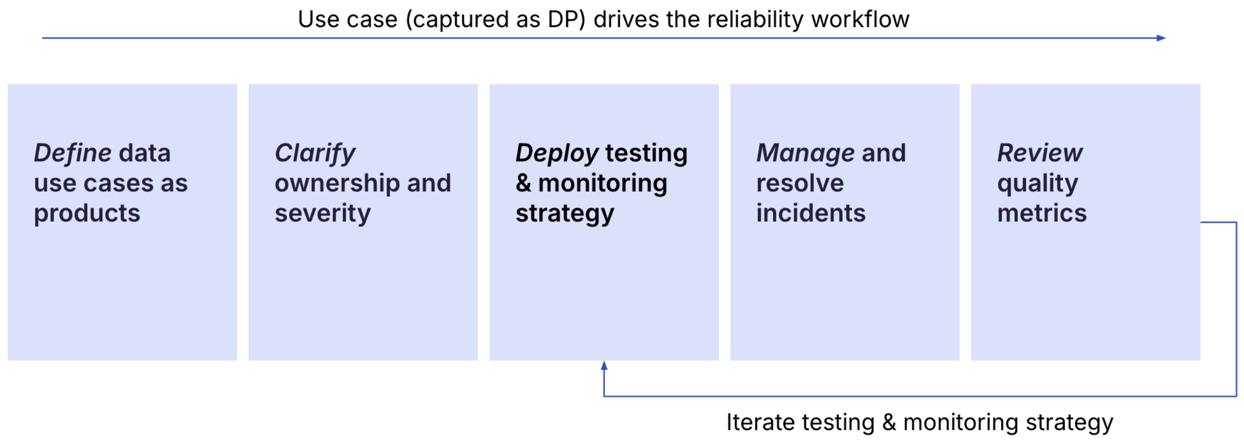
Case Study: Present metrics to stakeholders with a regular cadence

If you're only sporadically looking at your data quality metrics, it's harder to establish a benchmark and systemically track improvements. The data team at Lunar meets with key C-level stakeholders every 3 months to update them on data quality KPI progress, new initiatives, and any regulatory risks.

Create run books for data quality – if you're in a larger team, include clear steps around addressing each data quality dimension so it's clear for everyone. For example, if the Timeliness score is low, you can recommend steps such as adding a dbt source freshness check or an automated freshness monitor.

Data Product reliability workflow for continuous improvement

If you've made it this far, you've understood the key components of building a reliable data stack – from defining data use cases as products, setting ownership & severity, deploying strategic tests & monitors, and establishing quality metrics.


Use case (captured as DP) drives the reliability workflow
Define data use cases as products
Clarify ownership and severity
Deploy testing & monitoring strategy
Manage and resolve incidents
Review quality metrics
Iterate testing & monitoring strategy

Maintaining a reliable, high-quality data platform is not a one-off exercise but requires continuous investment.

New data use cases should be defined as data products with ownership and severity clearly defined.

Tests and monitors should be evaluated on an ongoing basis based on quality metrics, adding new checks in cases where issues are caught by stakeholders, removing low signal-to-noise tests, and keeping teams that score low on quality metrics accountable.

If you have questions about specific chapters or want to book a free consulting session with the authors of this guide, Petr Janda or Mikkel Dengsøe, schedule it here.

SYNQ

Book free consulting session

Book a platform demo: **synq.io**

Read the guide online: **synq.io/guide**

Discover more insights: **synq.io/blog**

SYNQ