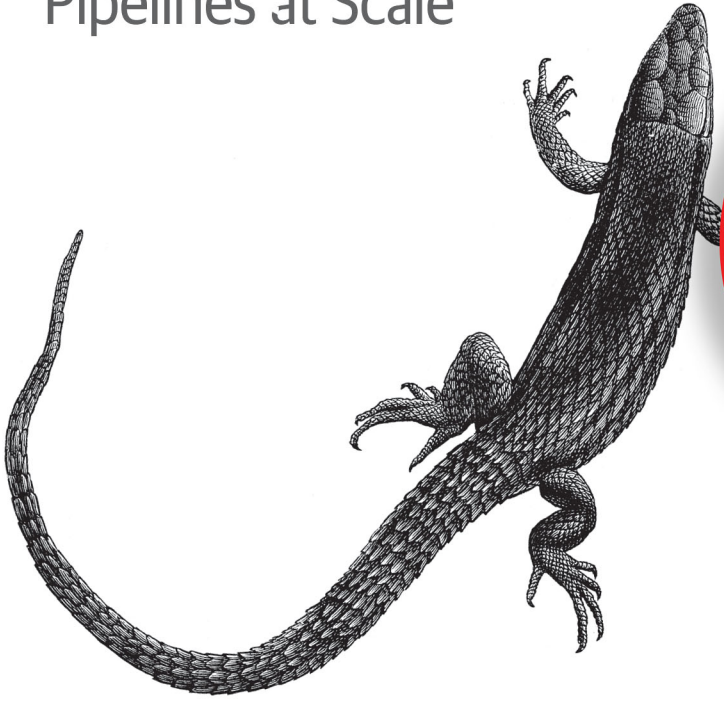
O'REILLY®

Data Contracts

Developing Production Grade Pipelines at Scale



Early Release
Raw & Unedited

Compliments of

gable

Chad Sanderson
& Mark Freeman

Data Contracts

Developing Production Grade Pipelines at Scale

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Chad Sanderson and Mark Freeman

O'REILLY®

Data Contracts
by Chad Sanderson and Mark Freeman

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Melissa Potter and Aaron Black
Production Editor: Katherine Tozer
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

March 2025: First Edition

Revision History for the Early Release

2024-02-26: First Release
2024-03-26: Second Release
2024-05-09: Third Release
2024-06-17: Fourth Release
2024-09-09: Fifth Release
2025-01-27: Sixth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098157630* for release details.

978-1-098-15763-0

Table of Contents

Brief Table of Contents (*Not Yet Final*)

CHAPTER 1

Why the Industry Now Needs Data Contracts

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

We believe that data contracts, an agreement between data producers and consumers that is established, updated, and enforced via an API, is necessary for scaling and maintaining data quality within an organization. Unfortunately, data quality and its foundations, such as data modeling, have been severely deprioritized with the rise of big data, cloud computing, and the Modern Data Stack. Though these advancements enabled the prolific use of data within organizations and codified professions such as data science and data engineering, its ease of use also came with a lack of constraints–leading many organizations to take on substantial data debt. With pressure for data teams to move from R&D to actually driving revenue, as well as the shift from model-centric to data-centric AI, organizations are once again accepting the merits of data quality being a must-have instead of a nice-to-have. Before going in depth about what data contracts are and their implementation, this chapter highlights why our industry forgone data quality best practices, why we're prioritizing data quality

again, and the unique conditions of the data industry post-2020 that warrant the need of data contracts to drive data quality.

Garbage-In Garbage-Out Cycle

Talk to any data professional and they will fervently state the mantra of "garbage-in garbage-out" as the root cause of most data mishaps and or limitations. Despite us all agreeing on what the problem is within the data lifecycle, we still struggle to produce and utilize quality data.

Modern Data Management

From the outside looking in, data management seems relatively simple—data is collected from a source system, moved through a series of storage technologies to form what we call a pipeline, and ultimately ends up in a dashboard that is used to make a business decision. This impression could be easily forgiven. The consumers of data such as analysts, product managers, and business executives rarely see the mass of infrastructure responsible for transporting data, cleaning and validating it, discovering the data, transforming it, and creating data models. Like an indescribably large logistics operation, the cost and scale of the infrastructure required to control the flow of data to the right parts of the organization is virtually invisible, working silently in the background out of sight.

At least, that should be the case. With the rise of machine learning and artificial intelligence, data is increasingly taking the spotlight– yet organizations still struggle to extract value from data. The pipelines used to manage data flow are breaking down, the data scientists hired to build ML models and deploy them can't move forward until data quality issues are resolved, executives make million dollar "data-driven" decisions that turn out to be wrong. As the world continues its transition to the cloud, our silent data infrastructure is not so silent anymore. Instead, it's groaning under the weight of scale, both in terms of volume and organizational complexity. Exactly at the point in time when data is poised to become the most operationally valuable it's ever been, our infrastructure is in the worst position to deliver on that goal.

The data team is in disarray. Data engineering organizations are flooded with tickets as pipelines actively fail across the company. Even worse, silent failures result in data changing in backwards incompatible ways with no one noticing resulting in multi-million dollar outages and worse, a loss of trust in the data from employees and customers alike. Data engineers are often caught in the crossfire of downstream teams who don't understand why their important data suddenly looks different today than it did yesterday, and data producers who ultimately are responsible for these changes have no insight into who is leveraging their data and for what reason. The business is often confused about the role data engineers are meant to play. Are they

responsible for fixing any data quality issue, even those they didn't cause? Are they accountable for rapidly rising cloud compute spend in an analytical database? If not, who is?

This state of the world is a result of *data debt*. Data debt refers to the outcome of incremental technology choices made to expedite the delivery of a data asset like a pipeline, dashboard, or training set for machine learning models. Data debt is the primary villain of this book. It inhibits our ability to deploy data assets when we need them, destroys trust in the data, and makes iterative data governance almost impossible. The pain of data engineering teams is caused by data debt - either managing it directly, or its secondary impacts on other teams. Over the subsequent chapters you will learn what causes data debt, why it is more difficult to handle than software debt, and how it can ultimately cripple the data functions of an organization.

What is data debt?

If you have worked in any form of engineering organization at scale you have likely seen the words 'tech debt' repeated dozens of times from concerned engineers who wipe sweat from their brow discussing a future with 100x the request volume to their service.

Simply put, tech debt is a result of short term decisions made to deploy code faster at the expense of long term stability. Imagine a software development team working on a web application for an e-commerce company. They have a tight deadline to release a new feature, so they decide to take a shortcut and implement the feature quickly without refactoring some existing code. The quick implementation works, and they meet their deadline, but it's not a very efficient or maintainable solution.

Over time, the team starts encountering issues with the implementation. The new feature's code is tightly coupled with the existing codebase which makes it challenging to add or modify other features without causing unintended side effects. Bugs related to the new feature keep popping up, and every time they try to make changes, it takes longer than expected due to the lack of proper documentation. The cost incurred to fix the initial implementation with something more scalable is *debt*. At some point, this debt has to be paid or the engineering team will suffer slowing deployment velocity to a crawl.

Like tech debt, data debt is a result of short term decisions made for the benefit of speed. However, data debt is *much worse* than software oriented tech debt for a few reasons. First, in software, the typical tradeoff which results in tech debt is speed in favor of maintainability and scale: Meaning how easy is it for engineers to work within this codebase, and how many customers/requests can we service? The operational function of the application is still being delivered which is intended to solve a core customer problem. In data however, the primary value proposition is trustworthiness. If the data that appears in our dashboards, machine learning models,

and embedded in customer facing applications can't be trusted then it is worthless. The less trust we have in our data, the less valuable it will be. Data debt directly affects *trustworthiness*. By building data pipelines quickly without the core components of a high quality infrastructure such as data modeling, documentation, and semantic validity, we are directly impacting the core value of the data itself. As data debt piles up, the data becomes more untrustworthy over time.

Going back to our e-commerce example, imagine the shortcut implementation didn't just make the code difficult to maintain, but also every additional feature layered on top actually made the product increasingly difficult to use until there were no customers left. That would be the equivalent of data debt. Second, data debt is far harder to unwind than technical debt. In most modern software engineering organizations teams have moved or are currently moving to microservices. Microservices are an architectural pattern that changes how applications are structured by decomposing them into a collection of loosely connected services that communicate through lightweight protocols. A key objective of this approach is to enable the development and deployment of services by individual teams, free from dependencies on others.

By minimizing interdependencies within the code base, developers can evolve their services with minimal constraints. As a result, organizations scale easily, integrate with off-the-shelf tooling as and when it's needed, and organize their engineering teams around service ownership.The structure of microservices allow for tech debt to be self-contained and locally addressed. Tech debt that affects one service does not necessarily affect other services to the same degree, and this allows each team to manage their own backlog without having to consider scaling challenges for the entire monolith.

The data ecosystem is based on a set of entities which represent common business objects. These business objects correspond to important real world or conceptual domains within a business. As an example, a freight technology company might leverage entities such as shipments, shippers, carriers, trucks, customers, invoices, contracts, accidents, and facilities. Entities are nouns - they are the building blocks of questions which can ultimately be answered by queries.

However, most useful data in an organization goes through a set of transformations built by data engineers, analysts, or analytics engineers. Transformations combine real world domain level data into logical aggregates called facts, which are leveraged in metrics used to judge the health of a business. A "customer churn" metric for example combined data from the customer entity, and the payment entity. "Completed shipments per facility" would combine data from the shipment entity and the facility entity. Because constructing metrics can be time consuming, most queries written in a company depend on both core business objects and aggregations. Meaning, data teams are tightly coupled to each other and the artifacts they produce - a distinct difference from microservices.

This tight coupling means that data debt which builds up in a data environment can't be easily changed in isolation. Even a small adjustment to a single query can have huge downstream implications, radically altering reports and even customer facing data initiatives. It's almost impossible to know where data is being used, how it's being used, and the level of importance the data asset in question is to the business. The more data debt piles up between producers and consumers, the more challenging it is to untangle the web of queries, filters, and poorly constructed data models which limits visibility downstream.

To summarize: Data debt is a vicious cycle. It depreciates trust in the data, which attacks the core value of what data is meant to provide. Because data teams are tightly coupled to each other, data debt cannot be easily fixed without causing ripple effects through the entire organization. As data debt grows, the lack of trustworthiness compounds exponentially, eventually infecting nearly every data domain and resulting in organizational chaos. The spiral of *data debt is the biggest problem to solve in data, and it's not even close.*

Garbage In / Garbage Out

Data debt is prominent across virtually all industry verticals. At first glance, it appears as though managing debt is simply the default state of data teams: a fate to which every data organization is doomed to follow even when data is taken seriously at a company. However, there is one company category that rarely experiences data debt for a reason you might not expect: startups.

When we say startup, we are referring to an early stage company in the truest sense of the word. Around 20 software engineers or less, a lean but functioning data team (though it may be only one or two data engineers and a few analysts) and a product that has either found product market fit or is well on its way. We have spoken to dozens of companies that fit this profile, and nearly 100% of them report not only having minimal data debt, but of having virtually no data quality issues at all. The reason this occurs is simple: The smaller the engineering organization, the easier it is to communicate when things change.

Most large companies have complex management hierarchies with many engineers and data teams rarely interacting with each other. For example, Convoy's engineering teams were split into 'pods,' a term taken from Spotify's product organizational model. Pods are small teams built around core customer problems or business domains that maximize for agility, independent decision making, and flexibility. One pod focused on supporting our operations team by building machine learning models to prioritize the most important issues filed by customers. Another worked on Convoy's core pricing model, while a third might focus on supplying real-time analytics on shipment ETA to our largest partners.

While each team rolled up to a larger organization, the roadmaps were primarily driven by product managers in individual contributor roles. The product managers rarely spoke to other pods unless they needed something from them directly. This resulted in some significant problems arising for data teams when new features were ultimately shipped. A software engineer managing a database may decide to change a column name, remove a column name, change the business logic, stop emitting an event, or any other number of issues that are problematic for downstream consumers. Data consumers would often be the first to notice the change because something looked off in their dashboard, or the machine learning began to produce incorrect predictions.

At smaller startups, data engineers and other data developers have not yet split into multiple siloed teams with differing strategies. Everyone is part of the same team, with the same strategy. Data developers are aware of virtually every change that is deployed, and can easily raise their hand in a meeting or pull the lead engineer aside to explain the problem. In a fast moving organization with dozens, to hundreds, or thousands of engineers this is no longer possible to accomplish. This breakdown in communication results in the most often referenced phenomena in data quality: Garbage In, Garbage Out (GIGO).

GIGO occurs when data that does not meet a stakeholder's expectations enters a data pipeline. GIGO is problematic because it can *only be dealt with retrospectively*, meaning there will always be some cost to resolve the problem. In some cases, that cost could be severe, such as lost revenue from an executive making a poor decision off a low quality dashboard whose results changed meaningfully overnight or a machine learning model making incorrect predictions about a customer's buying behavior. In other cases, the cost could be less severe - a dashboard shows wrong numbers which can be easily fixed before the next presentation with a simple CASE statement. However, even straight forward hotfixes underlie a more serious issue brewing beneath the surface: the growth of data debt.

As the amount of retroactive fixes grows over time, institutional knowledge hotspots begin to build up in critical areas within the data ecosystem. SQL files 1000 lines long are completely indecipherable to everyone besides the first data engineer in the company. It is not clear what the data means, who owns it, where it comes from, or why an active_customers table seems to be transforming NULLs into a value called 'returned' without any explanation in the documentation.

Over time, data debt caused by GIGO starts to increase exponentially as the ratio of software to data developers grows larger. The number of deployments increase from a few times per week, to hundreds or thousands of times per day. Breaking changes become a common occurrence, while many data quality issues impacting the contents of the data itself (business logic) can go unnoticed for days, weeks, or even months.When the problem has grown enough that it is noticeably slowing

down analysts and data scientists from doing their work, you have already reached a tipping point: Without drastic action, there is essentially no way out. The data debt will continue to mount and create an increasingly more broken user experience. Data engineers and data scientists will quit the business for a less painful working environment, and the business value of a company's most meaningful data asset will degrade.

While GIGO is the most prominent cause of data debt, challenges around data are also rooted in the common architectures we adopt.

The Death of Data Warehouses

Beginning in the late 1980s and extending through today, the Data Warehouse has remained a core component of nearly every data ecosystem, and the foundation of virtually all analytical environments.

The Data Warehouse is one of the most cited concepts in all of data, and is an essential concept to understand at a root level as we dive into the causes of the explosion of data debt. Bill Inmon is known as the "Father of the Data warehouse." And for good reason: he created the concept. In Inmon's own words:

> A data warehouse is a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process.

According to Bill, the data warehouse is more than just a repository; it's a subject-oriented structure that aligns with the way an organization thinks about its data from a semantic perspective. This structure provides a holistic view of the business, allowing decision-makers to gain a deep understanding of trends, patterns, ultimately leveraging data for visualizations, machine learning, and operational use cases.

In order for a data structure to be a warehouse it must fulfill three core capabilities.

- First, the data warehouse is designed around the key subjects of an organization, such as customers, products, sales, and other domain-specific aspects.
- Second, data in a warehouse is sourced from a variety of upstream systems across the organization. The data is unified into a single common format, resolving inconsistencies and redundancies.This integration is what creates a *single source of truth* and allows data consumers to take reliable upstream dependencies without worrying about replication.
- Third, data in a warehouse is collected and stored over time, enabling historical analysis. This is essential for time bounded analytics, such as understanding how many customers purchased a product over a 30 day window, or observing trends in the data that can be leveraged in machine learning or other forms of predictive analytics. Unlike operational databases that constantly change as new

transactions occur, a data warehouse is non-volatile. Once data is loaded into the warehouse, it remains unchanged, providing a stable environment for analysis.

The creation of a Data Warehouse usually begins with an Entity Relationship Diagram (ERD), as illustrated in Figure 1-1. ERD's represent the logical and semantic structure of a business's core operations and are meant to provide a map that can be used to guide the development of the Warehouse. An *entity is* a business subject that can be expressed in a tabular format, with each row corresponding to a unique subject unit. Each entity is paired with a set of dimensions that contain specific details about the entity in the form of columns. For example, a *customer* entity might contain dimensions such:

Customer_id
: Which identifies a unique string for each new customer registered to the website

Birthday
: A datetime which a customer fills out during the registration process

FirstName
: The first name of the customer

LastName
: The last name of the customer

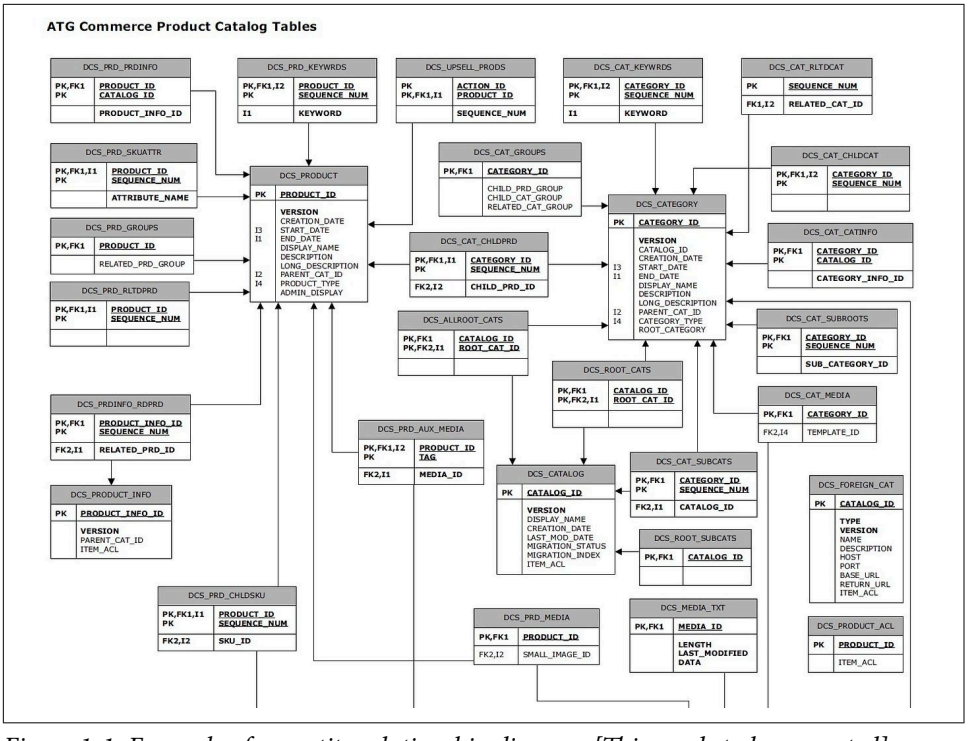


Figure 1-1. Example of an entity relationship diagram [This needs to be recreated]

An important dimension in ERD design are foreign keys. Foreign keys are unique identifiers which allow analysts to combine data across multiple entities in a single query. As an example, the customers_table might contain the following relevant foreign keys:

Address_id
: A unique address field which maps to the address_table and contains city, county, and zip code.

Account_id
: A unique account identifier which contains details on the customers account data, such as their total rewards points, account registration date, and login details.

By leveraging foreign keys, it is possible for a data scientist to easily derive the number of logins per user, or count the number of website registrations by city or state.

The relationship that any entity has to another is called its **cardinality**. Cardinality is what allows analysts to understand the nature of the relationship between entities, which is essential to performing trustworthy analytics at scale. For instance, if the

customer entity has a 1-to-1 relationship with the accounts entity, then we would never expect to see more than one account tied to a user or email address.

These mappings can't be done through intuition alone. The process of determining the ideal set of entities, their dimensions, foreign keys, and cardinality is called a **conceptual data model**, while the outcome of converting this semantic map into a tables, columns, and indices which can be queried through analytical languages like SQL is the **physical data model**. Both practices taken together represent the process of data modeling. It is only through rigorous data modeling that a Data Warehouse can be created.

The original meaning of Data Warehousing and Data Modeling are both essential components to understand in order to grasp why the data ecosystem today is in such disrepair. But before we get to the modern era, lets' understand a bit more about how Warehouses were used in their heyday.

The Pre-Modern Era

Data Warehouses predate the popularity of the cloud, the rise of software, and even the proliferation of the Internet. During the pre-internet era, setting up a data warehouse involved a specialized implementation within an organization's internal network. Companies needed dedicated hardware and storage systems due to the substantial volumes of data that data warehouses were designed to handle. High-performance servers were deployed to host the data warehouse environment, wherein the data itself was managed using Relational Database Management Systems (RDBMS) like Oracle, IBM DB2, or Microsoft SQL Server.

The use cases that drove the need for Data Warehouses were typically operational in nature. Retailers could examine customer buying behavior, seasonal foot traffic patterns, and buying preferences between franchises in order to create more robust inventory management. Manufacturers could identify bottlenecks in their supply chain and track production schedules. Ford famously saved over $1 billion by leveraging a Data Warehouse along with Kaizen-based process improvements to streamline operations with better data. Airlines leveraged Data Warehouses to plan their optimal flight routes, departure times, and crew staffing sizes.

However, creating Data Warehouses was not a cheap process. Specialists needed to be hired to design, construct, and manage the implementations of ERDs, data models, and ultimately the Warehouse itself. Software engineering teams needed to work in tight coordination with data leadership in order to coordinate between OLTP and OLAP systems. Expensive ETL tools like Informatica, Microsoft SQL Server Integration Services (SSIS), Talend and more required experts to implement and operate. All in all, the transition to a functioning Warehouse could take several years, millions of dollars in technology spends, and dozens of specialized headcount.

The supervisors of this process were called Data Architects. Data Architects were multi disciplinary engineers with computer science backgrounds with a specialty in data management. The architect would design the initial data model, build the implementation roadmap, buy and onboard the tooling, communicate the roadmap to stakeholders, and manage the governance and continuous improvement of the Warehouse over time. They served as bottlenecks to the data, ensuring their stakeholders and business users were always receiving timely, reliable data that was mapped to a clear business need. This was a best-in-class model for a while, but then things started to change…

Software Eats the World

In 2011, Venture Capitalist and founder of legendary VC firm Adreesen Horowitz, Marc Andreessen wrote an essay titled "Why Software Is Eating the World," published in The Wall Street Journal. In the essay, Andreessen explained the rapid and transformative impact that software and technology were having across various industries best exemplified by the following quote:

> Software programming tools and Internet-based services make it easy to launch new global software-powered start-ups in many industries — without the need to invest in new infrastructure and train new employees. In 2000, when my partner Ben Horowitz was CEO of the first cloud computing company, Loudcloud, the cost of a customer running a basic Internet application was approximately $150,000 a month. Running that same application today in Amazon's cloud costs about $1,500 a month
>
> —M. Andreesen

The 2000s marked a period of incredible change in the business sector.. After the dotcom bubble of the late 90s, new global superpowers had emerged in the form of high margin, low cost internet startups with a mind boggling pace of technology innovation and growth. Sergey Brin and Larry Page had grown Google from a search engine operating out of a Stanford dorm room in 1998, to a global advertising behemoth with a market capitalization of over $23 billion by the late 2000s. Amazon had all but replaced Walmart as the dominant force in commerce, Netflix had killed Blockbuster, and Facebook had grown from a college people-search app to a $153 million a year in revenue in only 3 years.

One of the most important internal changes caused by the rise of software companies was the propagation of AGILE. AGILE was a software development methodology popularized by the consultancy Thoughtworks. Up until this point, software releases were managed sequentially and typically required teams to design, build, test, and deploy entire products end-to-end. The waterfall model was similar to movie releases, where the customer gets a complete product that has been thoroughly validated and gone through rigorous QA. However, AGILE was different. Instead of waiting for an entire product to be ready to ship, releases were managed far more iteratively with a heavy focus on rapid change management and short customer feedback loops.

Companies like Google, Facebook, and Amazon were all early adopters of the AGILE. Mark Zuckerberg once famously said that Facebook's development philosophy was to 'move fast and break things.' This speed of deployment allowed AGILE companies to rapidly pivot their companies closer and closer to what customers were most likely to pay for. The alignment of customer needs and product features achieved a sort of nirvana referred to by venture capitalists as 'product market fit.' What took traditional businesses decades to achieve, internet companies could achieve in only a few years.

With AGILE as the focal point of software oriented businesses, the common organizational structure began to evolve. The software engineer became the focus of the R&D department, which was renamed to Product for the sake of brevity and accuracy. New roles began to emerge which played support to the software engineer: UX designers that specialized in web and application design. Product Managers, that combined project management with product strategy and business positioning to help create the roadmap. Analysts and Data Scientists that collected logs emitted by applications to determine core business metrics like sign-up rates, churn, and product usage. Teams became smaller and more specialized, which allowed engineers to ship code even faster than before.

With the technical divide growing, traditional offline businesses were beginning to feel pressure from the market and investors to make the transition into becoming AGILE tech companies. The speed that software-based businesses were growing was alarming, and there was a deep seated concern that if competitors adopted this mode of company building could emerge out the other end as a competitor with too much of an advantage to catch. In around 2015s, the term 'digital transformation' began to explode in popularity as top management consultant firms such as McKinsey, Delloitte, and Accenture pushed offerings to modernize the technical infrastructure of many traditionally offline companies by building apps, websites, and most importantly for these authors - driving a move from on-premise databases to the cloud.

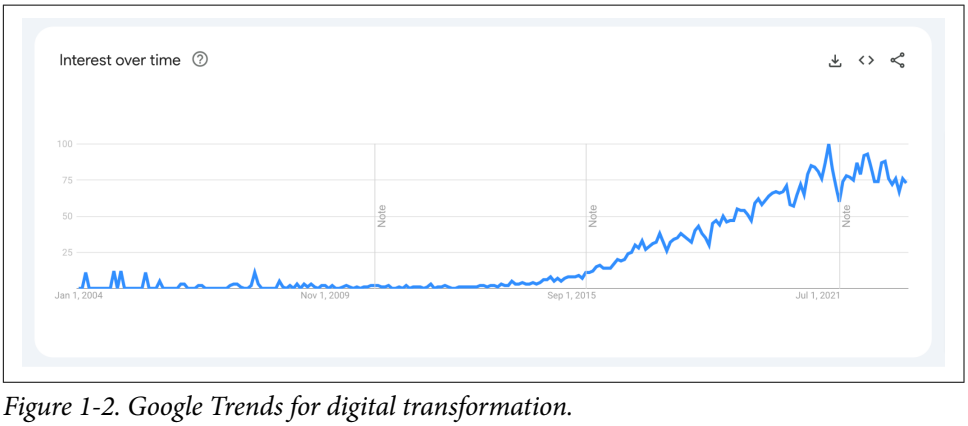


Figure 1-2. Google Trends for digital transformation.

The biggest promise of the cloud was one of cost savings and speed. Companies like Amazon (AWS), Microsoft (Azure), and Google (GCP) removed the need to maintain physical servers which eliminated millions of dollars in infrastructure and human capital overhead. This fit within the AGILE paradigm, which allowed companies to move far faster by offloading complexity to a service provider. Companies like McDonalds, Coca-Cola, Unilever, General Electric, Capital One, Disney, and Delta Airlines are all examples of entrenched Fortune 500 businesses that made massive investments in order to digitally transform and ultimately transition to the cloud.

A Move Towards Microservices

In the early 2000s, by far the most common software architectural style was a monolith. A **monolithic architecture** is a traditional software design approach where an application is built as a single, interconnected unit, with tightly integrated components and a unified database. All modules are part of the same codebase, and the application is deployed as a whole. Due to the highly coupled nature of the codebase, it is challenging for developers to maintain monoliths as they become larger leading to a significant slowdown in shipping velocity. In 2011 (The same year as Marc Andreesen's iconic Wall Street Journal article) a new paradigm was introduced called Microservices.

Microservices are an architectural pattern that changes how applications are structured by decomposing them into a collection of loosely connected services that communicate through lightweight protocols. A key objective of this approach is to enable the development and deployment of services by individual teams, free from dependencies on others. By minimizing interdependencies within the code base, developers can evolve their services with minimal constraints. As a result, organizations scale easily, integrate with off-the-shelf tooling as and when it's needed, and organize their engineering teams around service ownership. The term *Microservice* was first introduced by Thoughtworks consultants and gained prominence through Martin Fowler's influential blog.

Software companies excitedly transitioned to microservices in order to further decouple their infrastructure and increase development velocity. Companies like Amazon, Netflix, Twitter, and Uber were some of the fastest growing businesses leveraging the architectural pattern and the impact on scalability was immediate.

> Our services are built around microservices. A microservices-based architecture, where software components are decoupled and independently deployable, is highly adaptable to changes, highly scalable, and fault-tolerant. It enables continuous deployment and frequent experimentation.
>
> —Werner Vogels, CTO of Amazon

Data Architecture in Disrepair

In the exciting world of microservices, the cloud, and AGILE - there was one silent victim - the data architect. Data architects original role was to be bottlenecks, designing ERDs, controlling access to the flow of data, designing OLTP and OLAP systems, and acting as a vendor for a centralized source of truth. In the new world of decentralization, siloed software engineering teams, and high development velocity the dta architect was perceived as (rightly) a barrier to speed.

The years it took to implement a functional data architecture was far too long. Spending months creating the perfect ERD was too slow. The data architect lost control - they could no longer dictate to software engineers how to design their databases, and without a central data model from which to operate the upstream conceptual and physical data models fell out of sync. Data needed to move fast, and product teams didn't want a 3rd party providing well curated data on a silver platter. Product teams were more interested in building MVPs, experimenting, and iterating until they found the most useful answer. The data didn't need to be perfect, at least not to begin with.

In order to facilitate more rapid data access from day 1 the **data lake** pattern emerged. Data lakes are centralized repos where structured, unstructured, and semi structured data can be stored at any scale for little cost. Examples of data lakes are Amazon S3, Azure Data Lake Storage, and Google Cloud Storage. Analytics and Data Science teams can either extract data from the lake, or analyze/query it directly with frameworks like Apache Spark, or SaaS vendors such as Looker or Tableau.

While data lakes were effective in the short term, they lacked well defined schemas required by OLAP databases. Ultimately, this prevented the data from being used in a more structured way by the broader organizations. Analytical databases emerged like Redshift, BigQuery, and Snowflake where data teams could begin to do more complex analysis over longer periods of time. Data was pulled from source systems into the data lake and analytical environments so that data teams had access to fresh data when and where it was needed.

Businesses felt they needed **data engineers** who built pipelines that moved data between OLTP systems, data lakes, and analytical environments more than they needed data architects and their more rigid, inflexible design structures.

With the elimination of the data architect, so too resulted in the gradual phasing out of the Data Warehouse. In many businesses, Data Warehouses exist in name only. There is no cohesive data model, no clearly defined entities, and what does exist certainly does not act as a comprehensive integration layer that is a near 1:1 of business units reflected in code. Data Engineers do their best these days to define common business units in their analytical environment, but they are pressured from both sides - data consumers and data producers. The former is always pushing to go

faster and 'break things' and the latter operate in silos, rarely aware of where the data they produce is going or how it is being used.

There is likely no way to put the genie back in the bottle. AGILE, software oriented businesses, and microservices add real value, and allow businesses to experiment faster and more effectively than their slower counterparts. What is required now is an evolution of both the Data Architecture role and the Data Warehouse itself. A solution built for modern times, delivering incremental value where it matters.

Rise of the Modern Data Stack

The term 'Modern Data Stack' (MDS) refers to a suite of tools that are designed for cloud based data architectures. These tools have overlap with their offline counterparts but in most cases are more componentized and expand to a variety of additional use cases. The Modern Data Stack is a hugely important tool in a startups toolkit. Because new companies are cloud native - meaning they were designed in the cloud from day 1 - in order to perform analytics or machine learning at any scale will require an eventual adoption of some or all of the Modern Data Stack.

The Big Players

Snowflake is the largest and most popular of the Modern Data Stack, often being cited as the company that first established the term. A cloud oriented analytical database, Snowflake went public with its initial public offering (IPO) on September 16, 2020. During its IPO, the company was valued at around $33.3 billion, making it one of the largest software IPOs in history. Its cloud-native architecture, which separates compute from storage, offers significant scalability, eliminating hardware limitations and manual tuning. This ensures high-performance even under heavy workloads, with dynamic resource allocation for each query. Snowflake could save compute-heavy companies hundreds of thousands to millions of dollars in cost savings, providing a slew of integrations with other data products.

There are other popular alternatives to Snowflake like Google BigQuery, Amazon Redshift, and Databricks. While Snowflake has effectively corned the market on cloud-based SQL transformations, Databricks provides the most complete environment for unified analytics built on top of the worlds most popular open source large scale distributed data processing framework - Spark. With Databricks, data scientists and analysts can easily manage their Spark clusters, generate visualizations using languages like Python or Scale, train and deploy ML models, and much more. Snowflake and Databricks have entered a certifiable data arms race, their competition for supremacy ushering in a new wave or data-oriented startups over the course of the late 2010s and early 2020s.

The analytical databases however, are not alone. Extract, Load, and Transform (ELT) tool Fivetran has gained widespread recognition as well. While not publicly traded as of 2023, Fivetran's impact on the modern data landscape remains notable. Thanks to a collection of user-friendly interface and pre-built connectors, Fivetran allows data engineers to connect directly to data sources and destinations, which organizations can quickly leverage to extract and load data from databases, applications, and APIs. Fivetran has become the defacto mechanism for early stage companies to move data between sources and destinations.

Short for Data Build Tool, dbt is one of the fastest growing open source components for the Modern Data Stack. With modular transformations driven by YAML, dbt provides a CLI which allows data and analytics engineers to create transformations while leveraging a software engineering oriented workflow. The hosted version of dbt extends the product from just transformations, to a YAML based metrics layer that allows data teams to define and store facts and their metadata which can leveraged in experimentation and analysis downstream.

This is but a sampling of the tooling in the MDS. Dozens of companies and categories have emerged over the past decade, ranging from Orchestrations systems such as Airflow and Dagster, Data Observability tooling such as Monte Carlo and Anomalo, cloud-native Data Catalogs like Atlan and Select Star, Metrics repositories, feature stores, experimentation platforms, and on and on. (Disclaimer: These tools are all startups formed during the COVID valuation boom - some or all of them may not be around by the time you read this book!)

The reason why the velocity of data tooling has accelerated in recent years is primarily due to the simplicity of integrations most tools have with the most dominant analytical databases in the space: Snowflake, BigQuery, Redshift, and Databricks. These tools all provide developer friendly APIs and expose well structured metadata which can be accessed and leveraged to perform analysis, write transformations, or otherwise queried.

Rapid Growth

The Modern Data Stack grew rapidly in the ten years between 2012-2022. This was the time teams began transitioning from on-prem only applications to the cloud, and it was sensible for their data environments to follow shortly after. After adopting the core tooling like S3 for data lakes and an analytical data environment such as Snowflake or Redshift, businesses realized they lacked core functionality in data movement, data transformation, data governance, and data quality. Companies needed to replicate their old workflows in a new environment, which led data teams to rapidly acquire a suite of tools in order to make all the pieces work smoothly.

Other internal factors contributed to the acquisition of new tools as well. IT teams which were most commonly responsible for procurement began to become supplan-

ted with the rise of Product Led Growth (PLG). The main mode of selling software for the previous decade was top down sales. Sales people would get into rooms with important business executives, walk through a lengthy slide deck that laid out the value proposition of their platform and pricing, and then work on a proof of concept (POC) over a period of many months in order to prove the value of the software. This process was slow, required significant sign-off from multiple stakeholders across the organization, and ultimately led to much more expensive day 1 platform fees. PLG changed that.

Product Led Growth is a sales process that allows the 'product to do the talking.' SaaS vendors would make their products free to try or low cost enough that teams could independently manage a sign-off without looping in an IT team. This allowed them to get hands-on with the product, integrate with their day-to-day workflows and see if the tool solved the problems they had without a big initial investment. It was good for the vendors as well.

Data infrastructure companies were often funded by venture capital firms due to the high upfront R&D required. In the early days of a startup, venture capitalists tend to put more weight on customer growth over revenue. This is because high usage implies product market fit, and getting a great set of "logos" (customers) implies that if advanced, well known businesses were willing to take a risk on an early, unknown product then many other companies would be willing to do the same. By making it far easier for individual teams to use the tool for free or cheap, vendors could radically increase the number of early adopters and take credit for onboarding well known businesses.

In addition to changing procurement methodologies and sales processes, the resource allocation for data organizations grew substantially over the last decade [source]. As Data Science evolved from a fledgling discipline into a multi-billion dollar per year category, businesses began to invest more than ever into headcount for scientists, researchers, data engineers, analytics engineers, analysts, managers and CDOs.

Finally, early and growth stage data companies were suddenly venture capital darlings after the massive Snowflake initial public offering. Data went from an interesting nice to have, to undisputable longterm opportunity. Thanks to the low interest rates and a massive economic boost to tech companies during COVID, billions of dollars were poured into data startups resulting in an explosion of vendors across all angles of the stack. Data technology was so in demand that it wasn't unheard to invest in multiple companies that might be in competition with one another!

Problems in Paradise

Despite all the excitement for the Modern Data Stack, there were noticeable cracks that began to emerge over time. Teams who were beginning to reach scale were complaining - The amount of tech debt was growing rapidly, pipelines were breaking,

data teams weren't able to find the data they needed, and analysts were spending the majority of their time searching for and validating data instead of putting it to good use on ROI generating data products. So what happened?

First, software engineering teams were no longer engaging in proper business-wide data modeling or entity relationship design development. This meant that there was no single source of truth for the business. Data was replicated across the cloud in many different microservice. Without data architects serving as the stewards of the data, there was nothing to prevent unique implementations of the same concept, repeated dozens of times uniquely with data perhaps providing opposite results!

Second, data producers had no relationship to data consumers. Because it was easier and faster to dump data into a data lake than build explicit interfaces for specific consumers and use cases, software engineers threw their data over the fence to data engineers whose job it was to rapidly construct pipelines with tools like Airflow, dbt, and Fivetran. While these tools completed the job quickly, they also created significant distance between the production and analytical environments. Making a change in a production database had no guardrails. There was no information provided about who was using that data (if it was being used at all), where the data was flowing, why it was important, and what expectations on the data were essential to its function.

Third, data consumers began to lose trust in the data. When a data asset changed upstream, downstream consumers were forced to bear the cost of that change on their own. Generally that meant adding a filter on top of their existing SQL query in order to account for the issue. For example, if an analyst wrote a query intended to answer the question "How many active customers does the company have this month," the definition of *active* may be defined from the visits table, which records information every time a user opens the application. Secondly, the BI team may decide that a single visit is not enough to justify the intention behind the word 'active.' They may be checking notifications but not using the platform, which results in the minimum visit count to be set to 3.

```
 WITH visit_counts AS (
    SELECT
        customer_id,
        COUNT(*) AS visit_count
    FROM
        visits
    WHERE
        DATE_FORMAT(visit_date, '%Y-%m') =
        DATE_FORMAT(CURDATE(), '%Y-%m')
    GROUP BY
        customer_id
)
SELECT
    COUNT(DISTINCT customers.customer_id) AS active_customers
```

```
FROM
    customers
LEFT JOIN
    visit_counts ON
    visit_counts.customer_id = customers.customer_id
WHERE
    COALESCE(visit_counts.visit_count, 0) >= 3;
```

However, over time changes upstream and downstream impact the evolution of this query in subtle ways. The software engineering team decides to distinguish between visits and impressions. An impression is ANY application open to any new or previous screen, whereas a visit is defined as a period of activity lasting for more than 10 seconds. Before, all 'visits' were counted towards the active customer count. Now some percentage of those visits would be logged as impressions. To account for this, the analyst creates a CASE WHEN statement that defines the new impression logic, then sums the total number of impressions and visits to effectively get the same answer as their previous query using the updated data.

```
 WITH impressions_counts AS (
    SELECT
        customer_id,
        SUM(CASE WHEN duration_seconds >= 10 THEN 1 ELSE 0 END) AS visit_count,
        SUM(CASE WHEN duration_seconds < 10 THEN 1 ELSE 0 END) AS impression_count
    FROM
        impressions
    WHERE
        DATE_FORMAT(impression_date, '%Y-%m') =
        DATE_FORMAT(CURDATE(), '%Y-%m')
    GROUP BY
        customer_id
    HAVING
        (visit_count + impression_count) >= 3
)
SELECT
    COUNT(DISTINCT customers.customer_id) AS active_customers
FROM
    customers
LEFT JOIN
    impressions_counts ON
    impressions_counts.customer_id = customers.customer_id
WHERE
    COALESCE(impressions_counts.visit_count, 0) >= 3;
```

The more the upstream changes, the longer these queries become. All the context about why CASE statements or WHERE clauses exist is lost. When new data developers join the company and go looking for existing definitions of common business concepts, they are often shocked at the complexity of the queries being written and cannot interpret the layers of tech debt that had crusted over the in analytical environment. Because these queries are not easily parsed or understood, data teams

review directly with software engineers to understand what data coming source systems meant, why it was designed a particular way, and how to JOIN it with other core entities. Teams would then recreate the wheel for their own purposes leading to duplication and growing complexity beginning the cycle anew.

Fourth, the costs of data tools began to spiral out of control. Many of the MDS vendors use usage based pricing. Essentially that means 'pay for what you use.' Usage based pricing is a great model when you can reasonably control and scale your usage of a product over time. However, the pricing model becomes venomous when growth of a service snowballs outside the control of its primary managers. As queries in the analytical environment became increasingly more complex, the cloud bill grew exponentially to match. The increased data volumes resulted in higher bills from all types of MDS tools - which were now individually gouging on usage based rates that continued to skyrocket.

Almost overnight, the data team was larger than it had ever been, more expensive than it had ever been, more complicated than it had ever been, and delivering less business value than they ever had.

The Shift to Data-centric AI

While there are earlier papers on arXiv mentioning "data-centric AI," its widespread acceptance was pushed by Dr. Andrew Ng's and DeepLearningAI's 2021 campaign advocating for the approach. In short, data-centric AI is the process of increasing machine learning model performance via systematically improving the quality of training data either in the collection or preprocessing phase. This is in comparison to model-centric AI which relies on further tuning an ML model, increasing the cloud computing power, or utilizing an updated model to increase performance. Through the work of Andrew Ng's AI lab and conversations with his industry peers, he noticed a pattern where data-centric approaches vastly outperformed model-centric approaches. We highly encourage you to watch Andrew Ng's referenced webinar linked in the further resources section, but two key examples from Ng best encapsulate why the data industry is shifting towards a data-centric AI approach.

First, Andrew Ng highlights how underlying data impacts fitting ML models for the following conditions represented in Figure 1-3:

Small Data, High Noise:
: Results in poor-performing models, as numerous best-fit lines could be applied to the data and thus diminish the ability for the model to predict values. This often requires ML practitioners to go back and collect more data or remediate the data collection process to have more consistent data.

Big Data, High Noise:

Results in an ML model being able to find the general pattern, where practitioners utilizing a model-centric approach can realize gains by tuning the ML model to account for the noise. Though ML practitioners can reach an acceptable prediction level via model-centric approaches, Ng argues that for many use cases, there is better ROI in taking the time to understand why training data is so noisy.

Small Data, Small Noise:

Represents the data-centric approach where high-quality and curated data results in ML models being able to easily find patterns for prediction. Such methods require an iterative approach to improving the systems in which data is collected, labeled, and preprocessed before model training.

We encourage you to try out the accompanying Python code in Example 1-3 to get an intuitive understanding of how noise (i.e. data quality issues) can impact an ML model's ability to identify patterns.

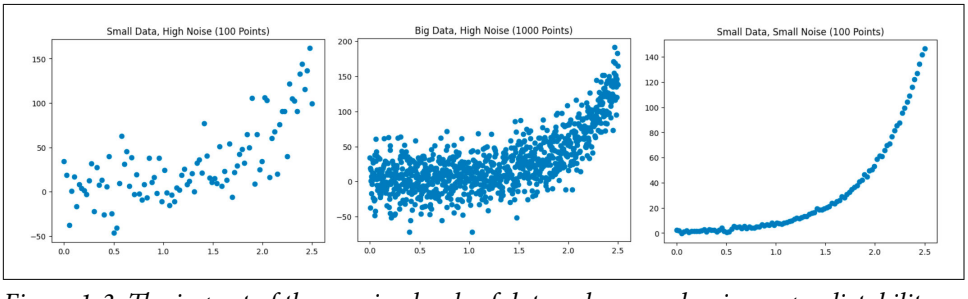


Figure 1-3. The impact of the varying levels of data volume and noise on predictability.

Example 1-3. Code to generate data volume and noise graphs in Figure 1-3.

```
import numpy as np
import matplotlib.pyplot as plt
def generate_exponential_data(min_X, max_x, num_points, noise):
    x_data = np.linspace(min_X, max_x, num_points)
    y_data = np.exp(x_data * 2)
    y_noise = np.random.normal(loc=0.0, scale=noise, size=x_data.shape)
    y_data_with_noise = y_data + y_noise
    return x_data, y_data_with_noise
def plot_curved_line_example(min_X, max_x, num_points, noise, plot_title):
    np.random.seed(10)
    x_data, y_data = generate_exponential_data(min_X, max_x, num_points, noise)
    plt.scatter(x_data, y_data)
    plt.title(plot_title)
    plt.show()
example_params = {
    'small_data_high_noise': {
        'num_points':100,
        'noise':25.0,
```

```
        'plot_title': 'Small Data, High Noise (100 Points)'
    },
    'big_data_high_noise': {
        'num_points':1000,
        'noise':25.0,
        'plot_title': 'Big Data, High Noise (1000 Points)'
    },
    'small_data_low_noise': {
        'num_points':100,
        'noise':1.0,
        'plot_title': 'Small Data, Small Noise (100 Points)'
    },
    # 'UPDATE_THIS_EXAMPLE': {
    #     'num_points':1,
    #     'noise':1.0,
    #     'plot_title': 'Your Example'
    # }
}
for persona in example_params.keys():
    persona_dict = example_params[persona]
    plot_curved_line_example(
        min_X=0,
        max_x=2.5,
        num_points=persona_dict['num_points'],
        noise=persona_dict['noise'],
        plot_title=persona_dict['plot_title']
        )
```

Second, Andrew Ng provided the analogy of comparing an ML engineer to a chef. The colloquial understanding among ML practitioners is that 80% of your time is spent preparing and cleaning data, while the remaining 20% is actually training your ML model. Similar to a chef, 80% of their time is spent sourcing and preparing ingredients for mise en place, while the remaining 20% is actually cooking the food. Though a chef can improve the food substantially by improving cooking techniques, the chef can also improve the food by sourcing better ingredients– which is arguably easier than mastering cooking techniques. The same holds true for ML practitioners, as they can improve their models via tuning (model-centric AI approach) or improve the underlying data during the collection, labeling, and preprocessing stages (data-centric AI approach). Furthermore, Ng found that for the same amount of effort, teams that leveraged a data-centric approach resulted in better-performing models than the teams using model-centric approaches.

Diminishing ROI of Improving ML Models

Incrementally improving machine learning models follows the Pareto Principle, where 80% of the gains in improving the model itself is achieved through 20% of the effort. Via a model-centric approach, every improvement grows exponentially harder for the needed effort, such as going from 93% to 95% accuracy.

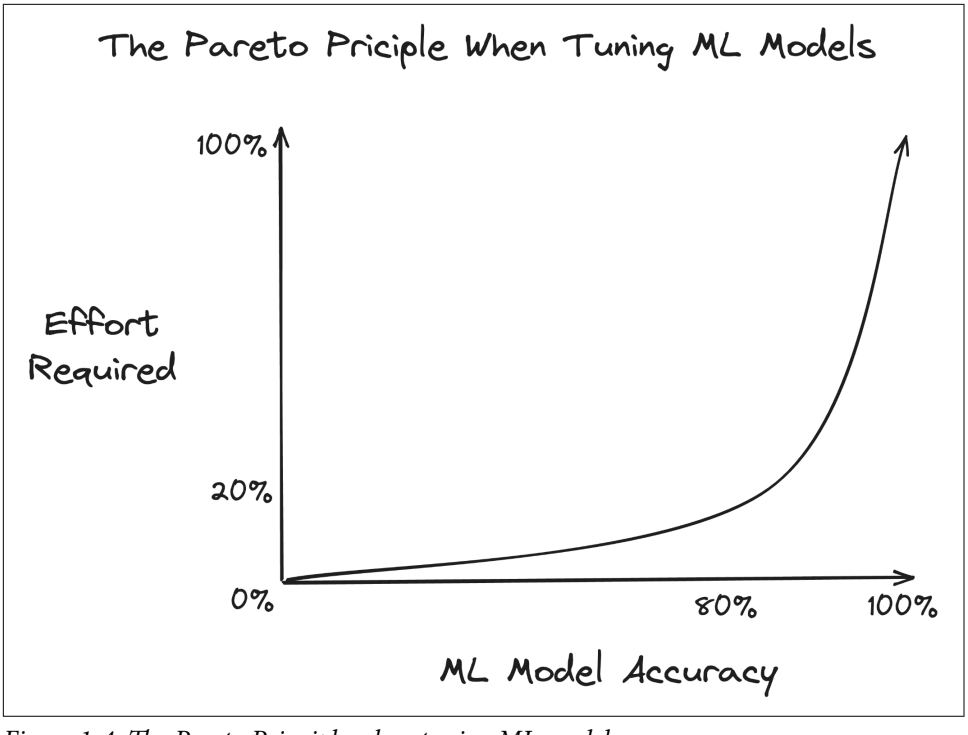


Figure 1-4. The Pareto Principle when tuning ML models.

Furthermore, taking a model-centric approach to AI often requires a substantial amount of data, hence why big tech SaaS companies have been the first successful adopters of machine learning at scale given their access to massive amounts of weblogs. Ng argues that as AI branches outside of these big tech domains, such as healthcare and manufacturing, ML practitioners are going to need to adopt methods that account for having access to substantially less data. For example, on average a single person generates about ~80MB of healthcare imaging and medical record data a year, as compared to the ~50GB of data a single user generates a month on average via their browsing activity.

In addition, Ng also argues that even with big data use cases, ML practitioners still need to wrestle with the challenges of small data. Specifically, after ML models are tuned on large datasets, gains come from accounting for the long-tail of use cases, which is ultimately a small data problem as well. Taking a data-centric AI approach to these long-tail problems within big data can provide more gains with substantially less effort than optimizing the ML model.

Commoditization of Data Science Workflows

While machine learning and AI has been developed for decades, it wasn't until around 2010 when the practice gained widespread utilization within industry. This is apparent in the number of ML vendors growing from ~5 companies in 2014 to ~200 plus companies in 2023, as illustrated in Matt Turck's yearly data vendor landscapes in Figure 1-5.

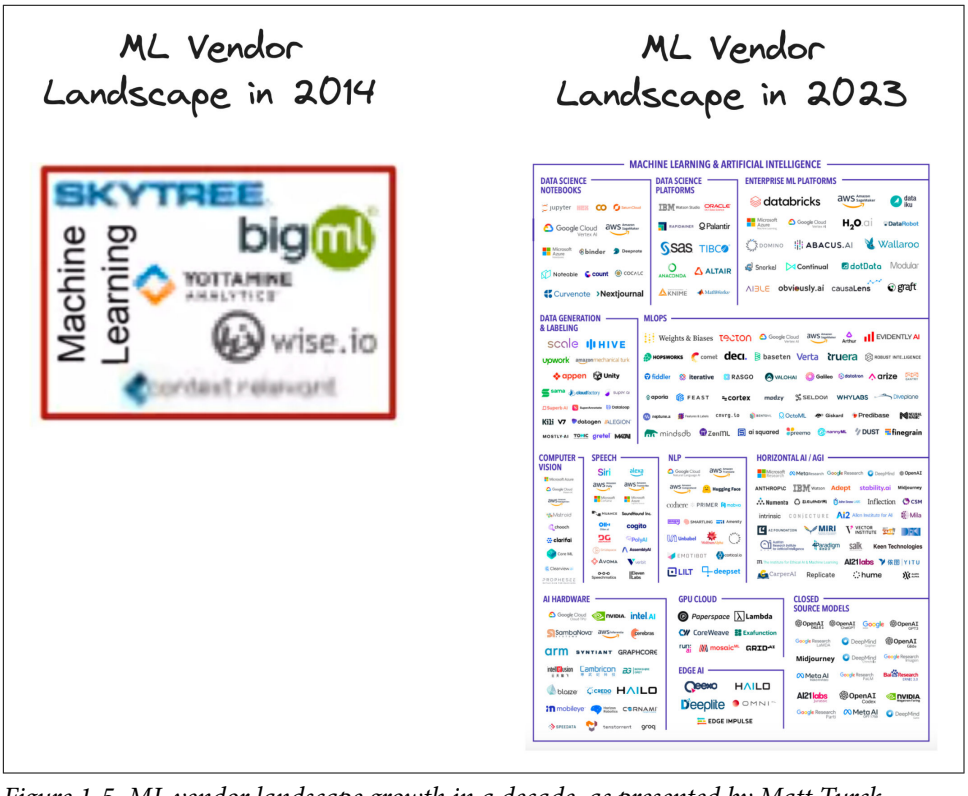


Figure 1-5. ML vendor landscape growth in a decade, as presented by Matt Turck.

Furthermore, as the data industry matured, less emphasis has been placed on developing ML models and instead the focus has turned to putting ML models in production. Early data science teams could get by with a few STEM PhDs toiling away in jupyter notebooks for R&D purposes, or sticking to traditional statistical learning methods such as regression or random forest algorithms. Fast forward to today, and data scientists have a plethora of advanced models they can quickly download from GitHub or can leverage auto-ml via dedicated vendors or products within their cloud provider. Also, there are entire open-source ecosystems such as scikit-learn or TensorFlow that have made developing ML models easier than ever before. It's simply not enough for a data team to create ML models to drive value within an organiza-

tion, the value is generated in their ability to reliably deploy machine learning models in production.

Finally, the rise of generative AI has further entrenched this trend of the commoditization of data science workflows. In a matter of an API call, that costs fractions of a cent, anyone can leverage the most powerful deep learning models to ever exist. For context, in 2020 Mark put an NLP model in production for an HR tech startup looking to summarize employee survey free text responses, utilizing the spaCy library. At the time, spaCy abstracted away the need to fine-tune an NLP model, hence why it was chosen for our quick feature development cycle. If tasked with the same project today, it would be unsound to not strongly consider using a large language model (LLM) for the same task, as no amount of tuning spaCy NLP models could compete with the power of LLMs. In other words, the process of developing and deploying an NLP model has been commoditized to a simple API call to OpenAI.

Data's Rise Over ML in Creating a Competitive Advantage

In parallel with the commoditization of data science workflows, the competitive advantage of ML models in themselves is decreasing.The amount of specialized knowledge, resources, and effort necessary to train and deploy an ML model in production is significantly lower than what it was even five years ago. This lower barrier of entry means that the realized gains of machine learning are no longer relegated to big tech and advanced startups. Especially among traditional businesses outside of tech, the implementation of advanced ML models is not only possible but expected. Thus, the competitive advantage of ML models in themselves have diminished.

The best representation of this reduced competitive advantage is once again the emergence of generative AI. The development of ChatGPT, and other generative AI models from big tech, was the culmination of decades of research, model training on expensive GPUs, and an unfathomable amount of web-based data. The requirements to develop these models were cost prohibitive for most companies and thus the models in themselves maintained a competitive advantage, up until recently. At the time of writing this, open-source and academic communities have been able to replicate and release similarly powerful generative AI models in a matter of months of the releases of their closed-source counterparts.

Therefore, the ways in which companies can maintain their competitive advantage, in a market where machine learning is heavily commoditized, is through their underlying data itself. Through a data-centric AI approach, taking the time to generate and or curate high quality data assets unique to a respective business will extract the most value out of these powerful but commoditized AI models. Furthermore, the data generated or processed by businesses are unique to the businesses themselves and hard, if not impossible, to replicate. The winners of this new shift in our data

industry won't be the ones who can implement AI technology, but rather the ones who can control the quality of the data they leverage with AI.

Conclusion

In this chapter we provided an overview of historical and market context as to why data quality has been deprioritized in the data industry for the past two decades. In addition we highlighted how data quality is again being deemed as integral as we evolve from the Modern Data Stack era and shift towards data-centric AI. In summary, this chapter covered:

- What is data debt and how it applies to garbage-in garbage-out
- The death of the data warehouse and the subsequent rise of the Modern Data Stack
- The shift from model-centric to data-centric AI

In Chapter 2, we will define data quality and how it fits within the current state of the data industry, as well as highlight how current data architecture best practices creates an environment that leads to data quality issues.

Additional Resources

- "The open-source AI boom is built on Big Tech's handouts. How long will it last?" by Will Douglas Heaven
- "The State Of Big Data in 2014: a Chart" by Matt Turck
- "The 2023 MAD (Machine Learning, Artificial Intelligence & Data) Landscape" by Matt Turck
- "A Chat with Andrew on MLOps: From Model-centric to Data-centric AI" by Andrew Ng

References

CHAPTER 2
Data Quality Isn't About Pristine Data

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

One of the early mistakes Mark made in his data career was trying to internally sell data quality on the merits of what pristine data *could* provide the organization. The harsh reality is that, beyond data practitioners, very few people in the business care about data– they instead care what they can do with data. Coupled with data being an abstract concept (especially among non-technical stakeholders), screaming into the corporate void about data quality won't get one far as it's challenging to connect it to business value, and thus data quality is relegated to being a "nice-to-have" investment. This dynamic changed dramatically for Mark when he stopped trying to internally sell pristine data and instead focused on the risk to important business workflows (e.g. often revenue driving) due to poor data quality. In this chapter, we expand on this lesson by defining data quality, highlight how our current architecture best practices create an environment for data quality issues, and what the cost of poor data quality is for the business.

Defining Data Quality

"What is data quality?" is a simple question that's deceptively hard to answer, given the vast reach of the concept, but its definition is core to why data contracts are needed. The first historically recorded form of data dates all the way back to 19,000 BCE, with data quality being an important factor for every century thereafter ranging from agriculture, manufacturing, to computer systems– thus, where does one draw the line? For this book, our emphasis is on data quality in relation to database systems, where 1970 is the cutoff given that's when Edgar F. Codd's seminal paper *A Relational Model of Data for Large Shared Data Banks* kicked off the discipline of relational databases.

During this time, the field of Data Quality Management emerged with prominent voices, such as Richard Y. Wang from MIT's Total Data Quality Management program, formalizing the discipline. In Dr. Richard Wang's and Dr. Diane Strong's most cited research article, they define data quality in 1996 as "data that are fit for use by data consumers…" among the following four dimensions: 1) conformity to the true values the data represents, 2) pertinence to the data user's task, 3) clarity in the presentation of the data, and 4) availability of the data.

References to these early works defining data quality can be found in the "further reading" section at the end of this chapter.

Throughout the academic works of Wang and colleagues, there is a massive emphasis on the ways in which the field is interdisciplinary and is greatly impacted by "...new challenges that arise from ever-changing business environments, ...increasing varieties of data forms/media, and Internet technologies that fundamentally impact how information is generated, stored, manipulated, and consumed." Thus, this is where this book's definition of data quality diverges from the 1996 definition above. Specifically, our viewpoint of data quality is greatly shaped by the rise of cloud infrastructure, big data for data science workflows, and the emergence of the modern data stack between the 2010s and the present day.

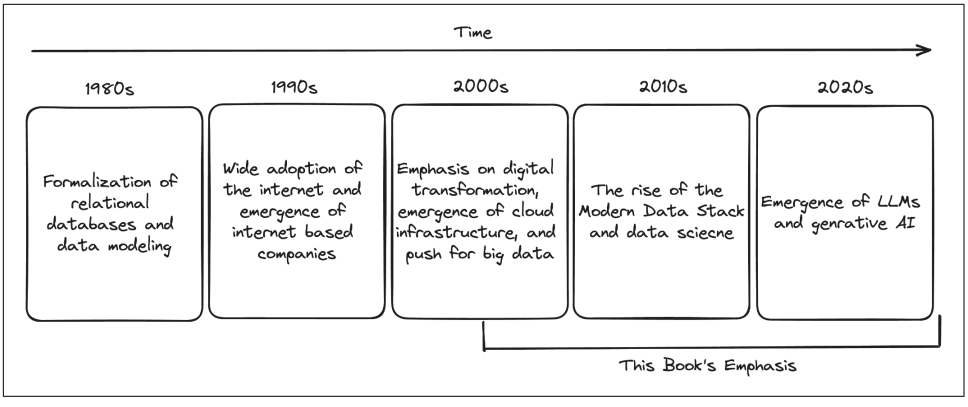


Figure 2-1. Timeline of data industry phases and where this book focuses on.

We define data quality as "an organization's ability to understand the *degree of correctness* of its data assets, and the tradeoffs of operationalizing such data at various degrees of correctness throughout the data lifecycle, as it pertains to being fit for use by the data consumer."

We especially want to emphasize the phrase "... tradeoffs of operationalizing such data at various degrees of correctness…" as it's key to a major shift in the data industry. Specifically, NoSQL was coined in 1998 by Carlo Strozzi and popularized again in 2009 by Johan Oskarsson. (source: https://www.quickbase.com/articles/timeline-of-database-history) Since then, there has been a proliferation of the various ways data is stored beyond a relational database, leading to increased complexities and tradeoffs for data infrastructure. As noted earlier, one popular tradeoff was the rise of the Modern Data Stack that opted for ELT and data lakes. Among this use case, many data teams have forgone the merits of proper data modeling to instead have vast amounts of data that can be quickly iterated on for data science workflows. Though it would be easier to have a standard way of approaching data quality for all data use cases, we must remember that data quality is as much of a people and process problem as a technical problem. Being cognizant of the tradeoffs being made by data teams, for better or worse, is key to changing the behavior of individuals operating within the data lifecycle.

In addition, we also want to emphasize the phrase "...ability to understand the degree of correctness…" within our definition. A common pitfall is the belief that *perfect data* is required for data quality; resulting in unrealistic expectations among stakeholders. The unfortunate reality is that data is in a constant state of decay that requires consistent monitoring and iteration that will never be complete. By shifting the language from a "desired state of correctness" for data assets to instead a "desired process for understanding correctness" among data assets, data teams account for the ever-shifting nature of data and thus its data quality.

In conjunction, with the ability to understand the *degree* of correctness among a data asset, a data team can make properly informed *tradeoffs* that balance the needs of the business and the effort of maintaining a level of data quality that supports their respective data consumers. But what's the cost of poor data quality when an organization gets this tradeoff wrong?

OLTP Versus OLAP and Its Implications for Data Quality

One of the most common data architecture patterns in our industry is the use of OLTP (online transaction processing) and OLAP (online analytical processing) databases as a way to separate the workloads of data access for transactions and analytics. Surprisingly, many individuals throughout the data lifecycle are intimately aware of their respective pieces of the architecture but not the other. Specifically, those who work in OLTP databases often primarily only work in such systems, and the same is true among those who work in OLAP databases. With the exception of roles such as data engineers or architects, few individuals within the data lifecycle of a respective company have an all-encompassing view of the entire data system through their work. We argue that the silos of OLTP and OLAP systems are the catalyst for many data quality issues among organizations, while also maintaining the notion that this data architecture design is valuable and has withstood the test of time.

A Brief Summary of OLTP and OLAP

As the names imply, OLTP databases are optimized for quick data transactions, such as updating user information on a website, while OLAP databases are optimized for scanning and aggregating large swaths of data for analytics workflows. In the early stages of a company's data infrastructure, OLTP databases alone often meet the needs of the business as there isn't enough data even to consider analytics. Furthermore, while the data is small, using SQL on top of these databases to answer simple questions about the logs doesn't cause enough strain to warrant concerns. This changes when the business begins to ask historical questions about the transaction data stored within the OLTP database, as analytical queries often require vast amounts of scanning that can bring the production database to a grinding halt. Thus, the need to replicate the transactional data into another database to prevent the production database from going down. Figure 2-2 illustrates at a high level the data flow of a data-driven organization utilizing OLTP and OLAP databases.

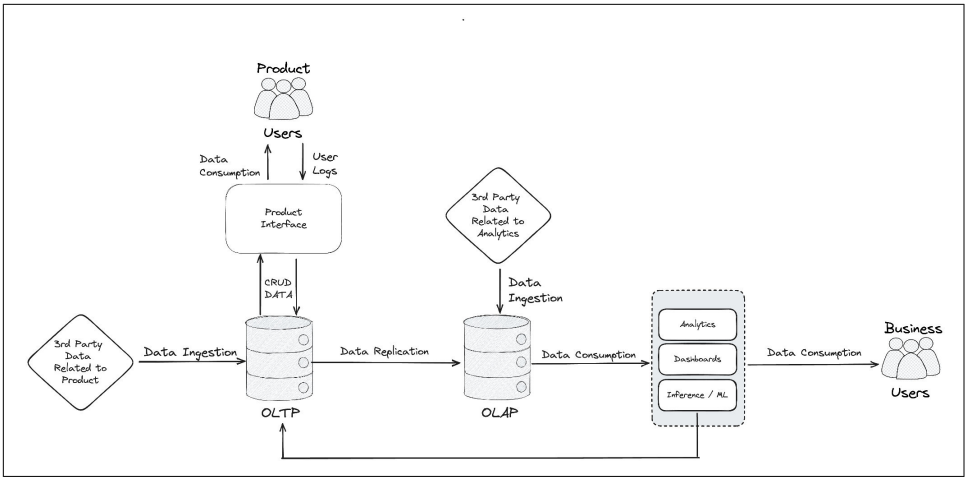


Figure 2-2. High level data flow of an organization utilizing OLTP and OLAP databases.

This splitting of databases marks an important inflection point in the data maturity of an organization. This split provides substantial gains in understanding organizational data at the tradeoff of increased complexity. This split also creates silos in the respective OLTP and OLAP "data world-views" that lead to miscommunications. Please note that there are other database formats, such as NoSQL or data lakes, but we've placed our emphasis here on relational OLTP and OLAP databases for simplicity.

Under the OLTP world-view, databases focus heavily on the speed of transactions of user logs, with emphasis on three main attributes:

1. Create, read, update, and delete tasks (CRUD).
2. Upholding compliance for atomicity, consistency, isolation, and durability (ACID).
3. Modeled data should be in the third-normalized form (3NF).

The above three components enable low latency of data retrieval, allowing user interfaces to quickly and reliably show correct data in sub-seconds rather than minutes to a user.

On the OLTP side of the data flow, illustrated in Figure 2-2, you will see software engineers as the main persona utilizing these databases, and are often the individuals implementing such databases before a data engineer is hired. This persona's role heavily emphasizes the maintainability, scalability, and reliability of the OLTP database and the related product software– data itself is a means to an end and not their main focus. Furthermore, while product implementations will vary, requirements and scoping are often clear with tangible outcomes.

Under the OLAP worldview, databases focus heavily on the ability to answer historical questions about this business via large scans of data where the data consumers care about the following:

1. Using denormalized data to find unique relationships.
2. Validating and or improving the nuances of business logic being applied to upstream data.
3. The ability to work iteratively on complex business questions that don't have clear requirements or may lead to dead ends.

Though this additional database increases the complexity of the data system, the tradeoff is that this increased flexibility enables the business to discover new opportunities not fully apparent in the product's CRUD data format.

On the OLAP side of the data flow, illustrated in Figure 2-2, you will see data analysts, data scientists, and ML engineers as primary roles working solely within OLAP data systems. Key to these roles is the iterative nature of analytics and ML workflows, hence the flexibility in the resulting data models compared to OLTP's third-normalized form data.

Translation Issues Between OLTP and OLAP Data Worldviews

To illustrate the different data worldviews of the OLTP and OLAP silos and how this leads to translation issues, we have a mock example of an aquarium business review website called Kelp. In this use case, the Kelp engineering team has recently enabled users to submit updated reviews, and the data analysts want to understand if this added feature increases user session duration. Yet, the data analyst must determine an important nuance with the business logic– "How are average review stars calculated?"

In Figure 1-2 below, we have Kelp's review data for an aquarium in Monterey, with three reviews represented in third-normalized form within an OLTP database. Note that the engineering team used the average of all reviews for simplicity within the V1 feature release.

Example 2-1. Kelp's Aquarium Review Data Organized in 3NF

```
-- Kelp's Aquarium Review Data Organized in 3NF

-- Aquarium Table
+----------+-------------+----------------+-------------+--------------------+
|aquarium_id|aquarium_name|aquarium_location|aquarium_size|adult_admission_price|
+----------+-------------+----------------+-------------+--------------------+
|123456     |'Monteray...'|'Monteray, CA'   |'Large'      |25.99               |
+----------+-------------+----------------+-------------+--------------------+
```

```
-- User Table
+-------+---------+-------------+
|user_id|user_name|demographic_n|
+-------+---------+-------------+
|1234   |Mark     |'<user info>'|
|5678   |Chad     |'<user info>'|
+-------+---------+-------------+

-- Reviews Table
+-----------+---------+-------+-----------+---------------+-----------+
|aquarium_id|review_id|user_id|number_stars|review_timestamp|review_text|
+-----------+---------+-------+-----------+---------------+-----------+
|123456     |00001    |1234   |1          |2019-05-24 07...|'Fish we..'|
|123456     |00002    |1234   |5          |2019-05-25 07...|'I revi...'|
|123456     |00003    |5678   |5          |2019-05-29 05...|'Amazin...'|
+-----------+---------+-------+-----------+---------------+-----------+

-- Average Stars Table
+-----------+-------------+
|aquarium_id|average_stars|
+-----------+-------------+
|123456     |3.67         |
+-----------+-------------+

-- User Activity Table
+-------+----------------------+--------------------+
|user_id|session_start_timestamp|session_end_timestamp|
+-------+----------------------+--------------------+
|1234   |2019-05-24 T07:46:02Z  |2019-05-24 T07:59:01Z|
|1234   |2019-05-25 T07:31:01Z  |2019-05-25 T07:48:03Z|
|5678   |2019-05-29 T05:14:08Z  |2019-05-29 T05:21:12Z|
+-------+----------------------+--------------------+
```

In Figure 1-3 below, we have Kelp's review data represented as a denormalized wide table created by the data analyst in the OLAP database, along with an ad-hoc table of various ways in which average stars can be represented. Beyond questions around average stars calculations, the data analyst would also consider other nuances of the business logic such as:

- How to calculate website session duration where multiple sessions are near each other.
- Are there instances of single users with multiple Kelp accounts?
- What session duration change is relevant to the business?
- Is it reviews directly or a combination of attributes leading to changes in session duration?

These questions represent a decoupling of business logic from the data represented in the OLTP worldview– resulting in a multitude of understandings that can lead to downstream data quality issues and data debt.

Example 2-2. Example of Kelp's Denormalized Wide Table Used by Data Analyst

```
-- Denormalized Wide Table Used by Data Analyst

-- Reviews Wide Table
+-----------+---------+-------+------------+----------------+---+
|aquarium_id|review_id|user_id|number_stars|review_timestamp|...|
+-----------+---------+-------+------------+----------------+---+
|123456     |00001    |1234   |1           |2019-05-24 07...|...|
|123456     |00002    |1234   |5           |2019-05-25 07...|...|
|123456     |00003    |5678   |5           |2019-05-29 05...|...|
+-----------+---------+-------+------------+----------------+---+

-- Reviews Wide Table Continued...
+---+-----------+-------------+-----------------+-------------+---+
|...|review_text|aquarium_name|aquarium_location|aquarium_size|...|
+---+-----------+-------------+-----------------+-------------+---+
|...|'Fish we..'|'Monteray...'|'Monteray, CA'   |'Large'      |...|
|...|'I revi...'|'Monteray...'|'Monteray, CA'   |'Large'      |...|
|...|'Amazin...'|'Monteray...'|'Monteray, CA'   |'Large'      |...|
+---+-----------+-------------+-----------------+-------------+---+

-- Reviews Wide Table Continued...
+---+---------------------+-----------------------+---------------------+
|...|adult_admission_price|session_start_timestamp|session_end_timestamp|
+---+---------------------+-----------------------+---------------------+
|...|25.99                |2019-05-24 T07:46:02Z  |2019-05-24 T07:59:01Z|
|...|25.99                |2019-05-25 T07:31:01Z  |2019-05-25 T07:48:03Z|
|...|25.99                |2019-05-29 T05:14:08Z  |2019-05-29 T05:21:12Z|
+---+---------------------+-----------------------+---------------------+

-- Adhoc Table: Average Stars
+-----------+-------+---------+----------+
|aquarium_id|avg_all|avg_first|avg_latest|
+-----------+-------+---------+----------+
|123456     |3.67   |3        |5         |
+-----------+-------+---------+----------+
```

Given these differences in data worldviews, how can a data team determine which perspective for calculating average stars is correct? In reality, both data worldviews are correct and dependent on the constraints the individual, and ultimately the business, cares about. On the OLTP side, Kelp's software engineers cared about the simplicity of the feature implementation and wanted to avoid any additional complexity unless deemed necessary; hence the default to averaging all the star reviews rather than applying business logic– in other words, it's a product decision rather than a

data decision. On the OLAP side, the data analyst is privy to unique combinations of data that don't make sense in an OLTP format but inform ways to improve the existing business logic. The data analyst may make suggestions that are sound from an analytics perspective but difficult to implement upstream in the OLTP database.

This difference in constraints and goals between data producers and consumers gets to the crux of why we believe data contracts are important. This book will go in great depth about data contracts, but in short data contracts are an agreement between data producers and consumers that is established, updated, and enforced via an API. The process of defining data contracts codifies the tradeoffs that both data producers and consumers are willing to make in respect to a data asset and why such configuration is important to the business.

Though this example highlights the OLTP and OLAP data architecture, this same pattern of miscommunication between different databases persists. All of which was exacerbated by the shift from on-prem data warehouses to cloud analytics databases that were inexpensive and easy to scale.

The Cost of Poor Data Quality

The foundational requirement of any software is that it is functional: In essence, does the program behave in the way it was intended to be designed? A software bug is a defect in the expected operating behavior of the codebase. Because the software does not function as expected, there is some business facing risk which has now been introduced. The risk might be transactional: The system might crash a customer's app in the middle of a purchase causing the business to lose a much needed revenue. The risk might be experience degrading: Perhaps the app loads too slowly, which causes a customer to drop off the page and potentially use a competitor instead. Or, the risk might relate to internal scalability: A tremendous amount of server load could be put onto cloud infrastructure causing costs to spiral out of control - and so on and so forth.

Data is not the same as software. At its core, data is not functional, it is descriptive. Data is a signal that is meant to effectively describe the state of the world around us. It can then be leveraged for an operational purpose, such as Artificial Intelligence, or an analytical one, such as creating a dashboard. However, its foundational requirement is that it accurately reflects the real world and can be trusted by other members of the business. Without this core truth, data means nothing. To put it simply, there is no operational value in incorrect data. A violation of data quality then, is of equal impact on data products as bugs are to customer facing software.

In software there are a variety of mechanisms to measure the impact of bugs or scalability issues on production systems: Error rates, downtime, latency, and incident rate are all examples of **trailing indicators.** Trailing indicators measure an outcome,

either positive or negative. In an e-commerce shop, a trailing indicator might be revenue. Revenue (money in the bank) is the last step in a long process that begins with a customer registering on a website, browsing the website, adding items to their cart, and checking out. In quality, trailing metrics indicate that the damage is done and we are measuring the blast radius. Reactive quality is extremely effective for diagnosing, prescribing next steps, and uncovering gaps in operational processes.

For example, a high number of errors being returned from a Javascript application typically means some aspect of the customer experience has regressed from a previous release. A software engineer may attempt to root cause the problem by first tracking the error history: At what time did the errors start? What was the error code being returned? Were there commonalities between each error that could narrow down where the problem was occurring? From there, the engineer might check logging or clickstream events to understand where unexpected behavior began to arise: If the number of page loads for a particular screen has dropped to zero, or the purchase of a particular item has been cut in half these would be great places to start an investigation.

The second mechanism of measurement tracks **leading indicators**. A leading indicator is an input to a success metric that precedes the metric itself. In the e-commerce world, a leading indicator might be customer registrations. The act of registration itself yields no value for the business, but if there is a quantifiable relationship between registrations and purchases, an increase in the former will eventually lead to more of the latter. In quality too, there are leading indicators which can be used to predict the future increase or decrease of a trailing indicator. An example is code coverage. Code coverage is a common mechanism used by software engineering teams to measure how well services follow best practices in terms of security, scalability, and usability. Low code coverage means greater risk!

Measuring Data Quality

Data follows a similar measurement pattern as software when it comes to quality. There are leading indicators, such as trust, ownership, and the existence of expectations & testing. There are also lagging indicators, such as the number of data incidents, data downtime, latency requirements, replication, and more. Let's dig into each a bit more, starting with leading indicators:

Data debt

Data debt is a measure of how complex your data environment is and what is its capacity to scale. While the debt itself doesn't represent a breaking change, there is a direct correlation between the amount of data debt and the development velocity of data teams, the cost of the data environment as a whole, and its ultimate scalability.

There are a few heuristics for measuring data debt.

1. How many data assets have documentation (and how much is serviceable)
2. The median number of dependencies per dataset in the data environment
3. The number of backfilling jobs performed in the past year
4. The average number of filters per query

Trustworthiness

The trustworthiness of the data is an excellent leading indicator because it correlates strongly with an increase in replication and (as a result) rising data costs. The less trust data consumers have in the data they are using, the more likely it is they will recreate the wheel to ultimately arrive at the same answer.

Trustworthiness can be measured through both qualitative and quantitative methodologies. A quarterly survey to the data team with the following questions is a strong temperature check, such as the following example.

How much do you agree or disagree with the following questions:

1. I trust our company's data
2. I am confident the data I use in my own work represents the real world truth
3. I am confident the data I use will not change unexpectedly
4. I trust that when I use data from someone else, it means exactly what they say it means
5. I am not concerned about my stakeholders receiving incorrect data

Additionally, the amount of replicated data assets is a fuzzy metric that is correlated with trust. The more a dataset is trustworthy, the less likely it is it will be rebuilt using slightly different logic to answer the same question. When this scenario does occur, it usually means that either A.) the logic of the dataset was not transparent to data developers, which prompted a lengthy amount of discovery that ultimately ended in replication, or B.) the data asset was simply not discoverable - meaning that it likely would have been reused if only the data asset in question had been easier to find. In this author's personal experience, it is more rarely the latter case than data teams might assume. A motivated data developer will find the data they need, but no matter how motivated they are they won't take a dependency on it for a business outcome if they can't trust it!

Ownership

Ownership is a predictive metric, as it measures the likelihood and speed errors will be resolved upstream of the quality issue when they happen. Ownership can be measured at the individual table level, but I recommend measuring the ownership as

a percentage of data upstream of a specific data asset with explicit ownership in place. This requires first cataloging data sources, identifying the owners or lack of owners, and aggregating the total number of owned sources divided by the total number of registered sources.

Ownership metrics are great to bring up during Ops meetings. Even better is when the lack of ownership can be tied to a specific outage or data quality issue, and even better still is when a lack of ownership can be framed with the risk of outages for important downstream data products. As an example, imagine that a CFOs executive dashboard takes dependencies on 10 data sources. If only 3 of these data sources have clearly defined ownership, it is fair to say that the CFO has a 70% chance of extended time to mitigation in the event of data outage. The more important the data product, the more critical ownership becomes.

Data downtime

Data downtime is becoming one of the most popular metrics for data engineering teams attempting to quantify data quality. Downtime refers to the amount of time critical business data can't be accessed due to data quality, reliability, or accessibility issues. We recommend a three-step process for tracking and actioning on data downtime:

Track and Analyze Downtime Incidents
: Keep a log of all data incidents, including their duration, cause, and resolution process. This data is crucial for understanding how often downtime occurs, its common causes, and how quickly your team can resolve issues.

Calculate Downtime Metrics
: Use the collected data to calculate specific metrics, such as the average downtime duration, frequency of downtime incidents, mean time to detect (MTTD) a data issue, and mean time to resolve (MTTR) the issue. These metrics provide a quantitative measure of your data's reliability and the effectiveness of your response strategies.

Assess Impact
: Beyond just measuring the downtime itself, assess the impact on business operations. This can include the cost of lost opportunities, decreased productivity, or any financial losses associated with the downtime.

Once completed, data engineering teams should not only have a comprehensive view of how their critical metrics are changing over time but clear impact on the business from downtime. This impact can be used to leverage the business to implement additional tooling for managing quality, roping in additional headcount, or driving greater upstream ownership to prevent problems before they occur.

Violated expectations

Expectations refer to what data consumers expect from their upstream data systems. Here, upstream can mean a table used as direct input for a query, or a transactional database maintained by production engineers supporting production applications. Depending on the team and use case, the specific expectations of the consumer might differ. A data engineer responsible for orchestrating an series of Airflow pipelines may have expectations on a PostgreSQL database in cloud storage, while an analyst who relies on a set of well-defined business entities to construct their dashboard view would have expectations on a set of analytical database tables in Snowflake SQL.

Expectations can take a variety of forms:

Schema
: Definition: The structure of the data, data types, and column names.

  Example: We always expect the primary key customer_ID to be a 6 character string.

Semantics
: Definition: The underlying business logic of the data and the entity itself.

  Example: We always use the email field to include an "@" symbol.

Service level agreements (SLAs)
: Definition: The latency and volume requirements of the data.

  Example: We expect a minimum of 1000 events from data sources per hour.

Personally identifiable information (PII)
: Definition: Information that can be used, individually or in combination of data values, to identify a person and or entity.

  Example: We expect this customer_name field is PII, and will be masked to all consumers.

Ideally, all violated expectations should be centrally logged. Teams can record when these violations occurred, who was responsible, the data source in question, and any downstream impacted assets. The success of both the data engineering and data producer teams can be measured against the total number of violations, which represent a range of data quality and governance issues. If this number gradually decreases quarter over quarter, it is an indication that the business is becoming more holistically aware of data quality and responsive to issues.

Quarterly incident count

While not substantially dissimilar from certain Data Downtime metrics, a quarterly incident count is a great way for business and technical leaders to gain a robust understanding of the impact of data quality on the company and its data products.

A data incident technically refers to any event that compromises the integrity, availability, or confidentiality of data. This could include unauthorized access to data (a security breach), loss of data due to system failures or human error, corruption of data because of software bugs or hardware malfunctions, or any situation where data is rendered inaccurate, incomplete, or inaccessible.

The most meaningful types of data incidents have some type of real-world impact. In software, incidents often cause a steep drop off in revenue, spurning valuable customers, or even facing legal and or political blowback. Therefore, not every outage should necessarily be treated as an incident. If an impact to data quality only causes a few dashboards to show incorrect numbers, it's difficult to argue this had a tangible impact on the business. However, if a decision was made that led to a negative outcome, based on those incorrect numbers, that's a very different story.

Here are some common examples of data incidents worth reporting:

Incorrectly allocating marketing spend:

During the COVID pandemic, the marketing team of one popular tech startup were on edge - concerned that users may spike or decline due to the unpredictability of the virus and surrounding legislation. During a monthly report, an analyst noticed that new user sign-ups were down by over 25%, marking a massive decrease in potential revenue. To get ahead of the issue, marketing received approval to significantly increase their budget for email campaigns and public outreach. A few months later, however, it was determined that the cause of the issue was not user behavior at all, but an improper change to how the product engineering team was recording the user-sign-up event. Whoops!

Real world recalls:

A consumer goods company that manufactured expensive home utilities used the item_ID in their primary transactional database as the serialization number printed on the bar code for each of their products. One day, production engineers tested adding a character to the item_ID not realizing how it would affect teams downstream. By the time they realized their mistake, hundreds of incorrect barcodes had been printed, put onto a myriad of different household items and shipped to homes, hotels, and restaurants. The company had to pay delivery drivers to revisit each dropoff and exchange the barcodes which was both time-consuming and expensive for such a seemingly small change!

While often less impressive in terms of raw quantifiable data, these stories best demonstrate the incredible importance of data quality and have the highest success rate in driving investment. Even with the best measures, we must remember that data quality is not only about *what* is impacted but also *who* is impacted within these data workflows.

Who Is Impacted

Data quality issues impact stakeholders in a variety of ways depending on the persona of the user. Each persona has a different relationship with data, and therefore feels the pain in a related but unique way.

Data engineers. Data Engineers often pull the shortest end of the stick, Because the word 'data' is in their name business teams make the assumption that any data issue can be dumped on their plate and promptly resolved. This is anything but true! Most data engineers do not have a deep understanding of the business logic leveraged by product and marketing organizations for analytics and ML. When requests to resolve DQ issues arise on the data engineers backlogs, it takes them days or even weeks to track down the offending issue through root cause analysis and do something about it.

The time it takes to resolve issues leads to an enormous on-call backlog, and frequent tension with the analytics and AI teams are high due to a litany of unresolved or partially resolved outages. This is doubly bad for data engineers, because when things are running smoothly they are rarely acknowledged at all! Data Engineers unfortunately fall into the rare category of work whose skills are essential to the business, but because they can't claim *quick wins* in the same way a data scientist, software engineer, or product manager might, given their visibility in the organization is comparatively diminished. Life is not good when the only time people hear about you is when something is failing!

Data scientists . Data Scientists are scrappy builders that often come from academic backgrounds. In academia, the emphasis is more on ensuring that research is properly conducted, interesting, and ethically validated. The business world represents a sudden shift from this approach to research - focusing instead of money making exercises, executing quickly, and tackling the low hanging fruits (see: boring problems) first and foremost. While data scientists all know how to perform validation, they are much less used to data suddenly changing, not being able to trust data, or data losing its quality over time.

This makes the data scientist particularly susceptible to the impacts of data quality. Machine Learning models frequently make poor or incorrect predictions. Data sets developed to support model training and other rigorous analysis are not maintained for long periods of time until they suddenly fail. Expected to deliver tangible business value, Data scientists may find themselves in a bind when the model they have been reporting was making the business millions of dollars was actually off by an order of magnitude, and they just recently found out.

Analysts. Analysts is a broad term. It could refer to financial analysts making decisions on revenue data, or product analysts reviewing web logs, clickstream events,

and measuring the impact of new features on user experiences. Most analysts use the same set of tools: 3rd party Visualization software like Looker, Tableau, Mixpanel, Amplitude, or of course the trusted Microsoft Excel. Analysts need to have more than a passing familiarity with the underlying data. It is essential they understand how customers is JOINed to items purchased, why a long SQL file seems to be filtering out users that haven't visited the website in the last 45 days, and exactly what a purchase_order event refers to. More than any other job function, analysts are the most connected to the business logic of each company.

Data Quality impacts Analysts by throwing their understanding of the business logic into disarray. If a column in a production database refers to kilometers today, but miles tomorrow it will effectively double the values of any downstream user leveraging that table. Analysts are also in the unfortunate position of being blamed when things change that they can't control. Their worst nightmare is getting an email at 4:45pm on Friday titled: "Why is this data wrong?"

Software engineers. Software engineers are impacted by data quality in a more roundabout way, in the sense that their changes are usually the root cause of most problems. So while they may not be impacted in the same way a data engineer, analyst, or data scientist might be - they often find themselves being shouted at by data consumers if an incompatible change is made upstream that causes issues downstream.

This isn't (usually) for lack of trying. Software engineers will often announce potentially breaking changes on more public channels hoping that any current or would-be users will notice and prepare for the migration accordingly. Inevitably though, after receiving very little feedback and making the change, data teams immediately start screaming and asking to rollback.

Business teams. Business teams refer to (typically) non-technical end customers of the data. This might be a marketing lead using a dashboard on SEO attribution to measure the impact of their content marketing strategy, or a product manager reviewing the click through funnels for a new feature they launched on the website.

Business Teams have very little agency when it comes to resolving identifying data quality issues. When something is noticeably wrong all they can do is message analysts and ask them to look into it. Worse, it is very challenging for a business user to differentiate between a legitimate unexpected change and a data quality problem.

At a previous company Chad worked at, an application team was using a common 3rd party vendor for event instrumentation. When COVID happened, the analytics in the application began reporting an incremental, but noticeable drop in active customer sessions. This drop continued day by day, until it stabilized at around a 25% decrease from the peak pre COVID. The product managers and marketers were beside themselves. Convinced the pandemic had permanently impacted churn rates, they funneled money into marketing to rebalance the numbers. This turned out

to be a false alarm. An analyst who had been looking into a totally separate issue discovered there was a similar drop in average time on page. It was oddly coincidental that two seemingly unrelated metrics experienced the exact same decrease! After looking into the issue and contacting the vendor, the team discovered the initial drop was due to an implementation mistake that ultimately resulted in a bug. All that marketing spend - wasted. Oops.

Conclusion

In this chapter we defined data quality and contextualized it to the current state of the data industry including data architecture implications and the cost of poor data quality. In summary, this chapter covered:

- Defining data quality in a way that looks back to established practices but accounts for recent changes in the data industry.
- How OLTP and OLAP data architecture patterns create silos that lead to data miscommunications.
- The cost of poor data quality is a loss of trust in one of the business's most important assets.

In Chapter 3, we will discuss the challenges of scaling data infrastructure within the era of the Modern Data Stack, how scaling data is not like scaling software, and why data contracts are necessary to enable the scalability of data.

Additional Resources

"The open-source AI boom is built on Big Tech's handouts. How long will it last?" by Will Douglas Heaven

"The State Of Big Data in 2014: a Chart" by Matt Turck

"The 2023 MAD (Machine Learning, Artificial Intelligence & Data) Landscape" by Matt Turck

"A Chat with Andrew on MLOps: From Model-centric to Data-centric AI" by Andrew Ng

References

CHAPTER 3

The Challenges of Scaling Data Infrastructure

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In this chapter we'll dive further into the challenges of the everyday workflows of data producers and consumers in building and maintaining complex software and data systems. Furthermore, we'll elaborate on how data development is different from software development, and the implications of these differences on our ability to drive value with data in an organization. Being aware of these challenges will equip you to understand how data contracts fit within the workflows of developers, what problems it mitigates, and how you can help developers understand their role within a data contract architecture.

How Data Development Is Not Like Software Development

As mentioned in Chapter 1, data development best practices proceeded software development by decades. There existed previously a set of best practices, design philosophies, and management methodologies wholly separate from the software engineering workflows which came after. Naturally, there are unique differences between data development and software development that prevent one from being a carbon copy of the other in terms of tools or processes. It is useful to walk through these flows to better understand why data requires a similar, but unique paradigm for managing scale.

Perhaps the largest of these differences can be found when comparing pre-development patterns. Pre-development refers to engineering time spent on a project outside of writing code. In software engineering, the most important pre-development steps are gathering requirements and designing architecture. However for data developers, the most essential pre-development steps are formulating the right hypothesis and exploring/understanding the data.

How Software Engineers Build Products

Good software engineering always begins the same way: with a requirements document. A requirements document is a set of customer needs created by the software engineer or someone else on the product team responsible for user research. Most commonly this role belongs to UX designers or Product Managers. The requirements document focuses on a particular problem that, if solved, would hypothetically result in delivering value for the customer and meaningfully increasing a target success metric.

Software engineers review requirement documents and after performing a feasibility analysis convert the user stories into architectural specifications. If the requirement document defines the why, the spec defines the how. Here, engineers decide which technologies to use, why those technologies make the most sense for their application, list out assumptions, and make decisions on whether to buy or build tooling in-house.

The technical spec is usually subject to the most scrutiny by other engineers. During the review phase, other engineers will challenge the primary architect on why they made certain decisions and potentially bring up issues that hadn't yet been considered. Is it necessary to have a real-time data feed? Is this going to require an extensive security review? How will users authenticate? How will we keep latency low on the front-end? These are all common questions during the architecture design meetings.

At this point, engineers begin building. While the actual act of writing code certainly requires a level of expertise and speed, what separates great engineers from the not-so-great is how much thought and planning they put into the initial architecture. If the technical spec was well built, there isn't much work to do after the code has been deployed and QA'd in a developer environment. The feature goes live, the team announces their work, and the company cheers. It's rare that (in the exception of bugs) the deployed feature would ever need to significantly change if it was well designed.

How Data Developers Build Products

While there certainly are similarities between data development and software development, the workflows have a significant amount of disparity.

For one, most data development *begins with a question* rather than immediately building an application around a user experience. These questions could come from the data developer themselves, but more often a business partner. Here are a few examples:

- Operations Lead: "How many resources do we need to deal with the Christmas buying surge?"
- Chief Revenue Officer: "How is our new pricing strategy going? Do customers like it?"
- Marketing Manager: "How many people are downloading our Ebook and does it lead to sales?"
- Sales Lead: "Did our team make their quotas? Which region performed the best?"
- Product Manager: "What was the impact on revenue for the new feature we launched?"
- Head of HR: "What is our employee churn rate? What are the most common causes of attrition?"

First, however, before even beginning to understand what operational data products we can build on top of these queries, data developers must understand if the question can even be answered from the data that exists. This involves data discovery. Data discovery is the process of searching for data in an attempt to answer a question. There are multiple components a data developer must keep in mind when beginning the journey towards discovery:

- Does this data exist?
- Where is the data located?
- Who is the owner of the data?

- Can the owner explain what it means?
- How do I use the data?
- Do I have permission to access the data?

Each of these discovery oriented questions may take hours or more realistically days to answer.

Second, depending on the answer to the above, it could result in a radically different next step. For instance, if the data doesn't exist, the data developer must decide if they would rather set down the path on trying to have a data producer generate the data on their behalf or report back on the failed project to the team lead. Similarly, if data doesn't have any owner, do you push through with a prototype even though no-one Is around to take care of quality, or do you let the asking team know that no answer you could provide them would be trustworthy.

Third, the data developer needs to construct the query to answer the question. But here, yet another difference arises. Just because someone has provided an answer does not yet mean it is useful on its own. As an example, a product designer may ask the question "how are people interacting with the new chatbot we shipped? Is it leading to more sales?"

To answer that question, first the data developer needs to find the user behavior data that shows the number of times a chatbot was shown to a user, and the number of clickstream events recorded when they interacted with it. Next they need to understand the nature of those interactions. Asking the chatbot a question is different from closing out the chatbot window, after all. Next the analyst would need to understand more details about the user's session. How long did they interact with the chatbot? Did they add anything to their cart afterwards? Did they purchase what was in their cart? Finally, the developer would need to do a comparative exercise in order to understand if the number of purchases was significantly different from the previous state of the world to the current state of the world. Things get even more complicated when you factor in that some percentage of users who saw the chatbot probably would have purchased anyway! How do we deal with those individuals in our research and how can we identify them?

During this phase the data developer is typically not looking for extreme accuracy. They are simply attempting to discover the data as fast as they can, attempting to understand it as best they can, and querying it as efficiently as they can en route to an answer. Things like data quality checks, testing, and monitoring, and semantic validity are the furthest thing from their mind. Once all these queries are developed and the final report is handed to the designer, it's possible that the signal is simply not strong enough to warrant a deeper dive or follow up steps. The data may show that there are many more customer interactions, but overall purchase rate seems relatively flat. This type of outcome would yield more questions rather than a tangible

operational application. “Why is it that the purchase rate is flat even though chatbot interactions are up? Is the chatbot not helpful, or is there no change in purchases for another reason?”

This means the data developer is in a constant state of flux and change. They are responding to new questions being asked, conducting discovery, and providing analysis - which is significantly different from the more structured and rigid workflow of software engineering.

However, when answered questions are useful, other teams and users begin to take a dependency. Answers to common questions like: “how many active customers does the business have over the past month?” are useful for almost anyone to know, and the output would likely be leveraged in a large variety of other queries. Once an answered question has been proven to be valuable, there comes with it an expectation of trust (to a certain degree). At this point the quality checks become much more meaningful!

It should be clear from these examples that software developers and data developers have unique workflows, and thus different needs in the tools and processes they use. In addition to these, each group requires a specific environment suited to their needs.

Software engineers need an environment that allows them to seamlessly write code, test for errors, and easily push to production. Data developers need an environment that facilitates prototyping and experimentation - discovering the data that exists, identifying what is trustworthy and what is not, selecting the useful answers, then creating data quality checks on what is expected to generate value.

Core Challenges for Modern Data Engineering Teams

Change Management has existed in the world of software engineering as a discipline for many years. In the early 2010s version control frameworks and the platforms that supported them such as GitHub and GitLab went mainstream. In addition to being user friendly platforms to host code, they provided AGILE, iterative mechanisms of conducting code review through a feature called Pull Requests (PR).

A PR is a request to merge a development branch back into the main branch, effectively productionizing a code change. Because code changes can have negative impacts if not thought through carefully, the PR allowed for other engineers to register themselves reviewers for any change within a repo. This allowed code review to be managed without long meetings and painfully scheduling conflicts. A developer could file a PR and then work on other projects while in review as demonstrated in Figure 3-1.

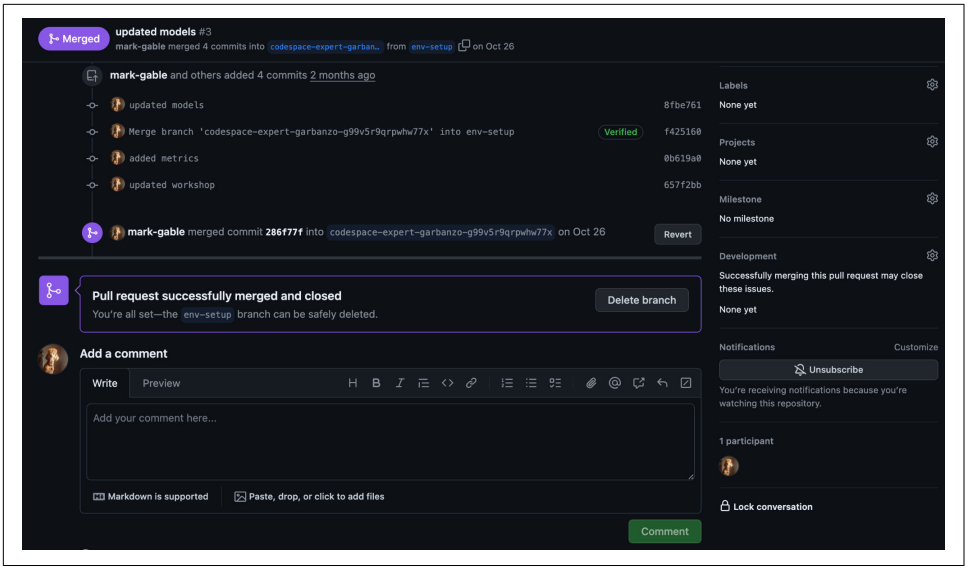


Figure 3-1. : GitHub's UI for a pull request.

Version control platforms themselves have become code change management systems, and the features they possess are built to reflect the range of use cases where change management is essential. Such features include:

- Version Control: Undo changes
- Pull Requests: Directly review changes
- Code diff: Visually review changes
- Views & Reviewers: Be alerted when changes happen

These systems allow the right people to be in the loop when the code evolves. However, there is no such system for data! It is incredibly difficult, sometimes impossible, to understand how changes to an application code base will impact data teams. There is no good way for those who would be impacted by changes to advocate for themselves in the way a PR reviewer might, nor is there any indication for the engineer of what is going to happen to their dependents once a change is made. This lack of context makes change management an extremely difficult endeavor in the data space. But why?

A core reason for this has to do with the organizational structure of most tech inclined businesses. Generally, software engineering teams want their code to be reviewed by other members on the same team. This is because your teammates are the ones who understand your services the best! It wouldn't make much sense to ask the owner of an unrelated back-end service to review a change to the front-end

settings page of your application. The two engineers would likely not be speaking the same language, would be missing context, and would not have a grasp on the gotchas, potential security issues, and other best practices within the organization.

For that reason, change review is easy to maintain. Every time a software engineer joins a new team they immediately subscribe to the repos their team owns. Many engineering teams have Slack channels through which they push new PRs from GitHub so teammates are actively kept in the loop. While this system may not 100% cover their bases, it is good enough to catch most major quality issues and allow teams to be functionally self sufficient.

However, in data this within-team oriented system does not work (as illustrated in Figure 3-2)! Downstream data teams do not produce data themselves, and are therefore dependent on upstream teams that maintain transactional systems to provide them with data, or 3rd party applications operated by business users or even the customers themselves. Changes to those systems are invisible to the data teams.

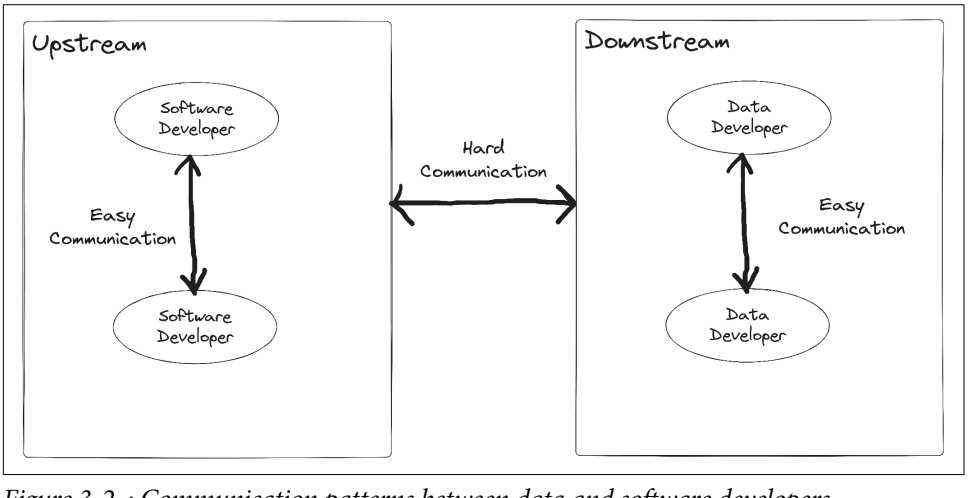


Figure 3-2. : Communication patterns between data and software developers.

The biggest challenge in creating effective data change management is that it must operate *between teams*, effectively forming a link between the data producer and data consumer. What increases the challenge is that different teams may have different requirements for their data. A team leveraging customer order information for a weekly report to the CMO only needs the data refreshed from production every 7 days, while a data science team leveraging order information for a real-time Recommendation system may need their training data refreshed daily to avoid drift.

The challenges with creating a data change management system are both cultural and technical. These include:

No visibility:

Data producers and data consumers both lack visibility of each other in different ways. Data producers do not understand who is using their data, why it's being used, how it's being transformed, and its tier of importance. Data consumers, on the other hand, don't understand where their data is coming from, why it's changing, who is changing it, and when changes are coming. Because there is nothing linking the producers and consumers together - it is challenging for this awareness to ever develop organically.

Not everything needs visibility:

As we covered earlier, data developers may go through many rounds of prototyping and discovery before settling on a use case that requires production grade data quality. In some organizations these production use cases may represent 10% of the total data assets or less! Making ALL data visible could lead to an overwhelming amount of information. This causes *alert fatigue*.

Alert fatigue:

Alert fatigue is the result of teams creating too many monitors that either do not communicate enough information to be actionable, or do but no one is willing or capable of taking action. Alert fatigue is by far the best way to make testing a useless exercise, creating a Pavlovian response in which any data monitor is ignored by default. Unless the change management system can discriminate between what is important and not important, and what is actionable vs. inactionable it is not useful to either data producers or consumers.

Not the right time:

The biggest cultural challenge with between-team change management is prioritization. Typically product teams define their roadmap at the start of each quarter after which it is usually set in stone. Asking another team to consider data quality as a first-class citizen at the expense of their own features is unlikely to happen! Simply asking others to be good citizens will not work. Data teams must be empowered to initiate a resolution on their own.

"Hands off my stuff":

Some teams are highly territorial and (unfortunately to say) inconsiderate. These organizations or individuals also may have very good reasons to be stand-offish. The service they maintain might be critically important to the business and the data pales in importance by comparison.

If you're reading this from the perspective of an enterprise level company it may seem as though there's no light at the end of the tunnel. The sheer scale and scope of the data quality problem you are facing feels inescapable - at every turn there is a new mess to clean up and a different (but related) problem to solve. If you're a growth stage company it may not be any better! You likely joined a tech company expecting the data to be high quality, easily accessed, and highly usable only to find the exact

opposite - a disjointed jumbled mess constantly breaking. Would you be interested to know then, that there is a unique class of company that has almost no data quality issues at all?

Early stage startups. To be even more specific: Startups with ~20 engineers or fewer.

We've spoken to dozens of early stage startups and in almost all cases data quality issues were minimal if not non-existent. The reason why is clear: As highlighted in Chapter 1, the smaller the engineering team, the easier it is for the data team to be represented when new features are launched.

Opposite to what some data teams might feel, data producers are actually not out to get you. They are intelligent, thoughtful people who have a rabid appetite for risk mitigation. Breaking things due to a code deployment they introduced is one of their greatest fears. When data engineers are able to clearly explain why and how a new feature can cause problems *before* a deployment occurs, it is extremely unusual for data producers to not be receptive towards that!

So if it works at small companies, then what's the problem for the rest of us? Well, companies don't stay small. They grow. And these days the greatest growth lever for tech oriented businesses is software engineers. According to Mikkel Dengsøe, the median data developer to software developer ratio is 1:4 among the top 50 European tech companies. At Chad's previous companies, he had an engineer team of over 200 with a data engineering team of only 5! There's a few reasons for this:

- Data engineers work on infra, software engineers work on product: It's always easier to get more resources when you're making the company money compared to running pipelines.
- C-level executives don't know what data engineers do: They know they are important, but due to a lack of visibility their team is rarely staffed properly.
- It's hard to hire data engineers: The sad truth is that there just aren't enough data engineers compared to software engineers. The good ones have options!
- It's hard to retain data engineers: The more data quality becomes a problem, the harder it is to keep them around.
- No one knows quite where they fit: We've seen data engineers as part of the data platform team, part of the data science team, and even (shockingly) part of the product team. Most organizations don't seem to have a great handle on how to manage data engineers which leads to them leaving for greener pastures.

And even if a reasonable amount of data engineers were hired, it would require a multiple order of magnitude improvement before data engineers could be present in every single meeting for every feature. Data teams can't move as fast as software teams - we have a giant source of truth monolith to think about after all.

But there's good news: Data teams don't need to be everywhere. We only need to be where it matters, when it matters, in other words the "right place, right time." If data teams can inject themselves into the development lifecycle at the point where data producers are likely to be the most receptive to their feedback, we can replicate what small startups are doing at tremendous scale. The trick is how to do it programmatically.

Why Data Development Needs a Design Surface

We'll be frank– any system that requires production quality data must be preceded by some form of change management framework. The purpose of data change management is to ensure that the right teams are looped into the review at the right time. The outcome of such a system is to prevent the "Garbage In / Garbage Out" problem, increasing the amount of trust in data by creating visibility across data producers and data consumers.

With theoretical alignment in place, we can discuss the practical requirements of data change management, and how we might leverage technology to achieve these outcomes over the following chapters.

Prevention first

The majority of data solutions today rely on reactive measures. These include monitoring, testing, and anomaly detection. Preventative systems are designed to catch breaking changes before they occur. From an implementation perspective, this means integration with a data producers CI/CD pipeline.

Communicative

The best way to prevent breaking changes is to connect people: namely data developers who understand their own data and the pipelines supporting them and the data producers who are familiar with how their upstream sources will change. Bridging the gap between data consumers and data producers is at its core an exercise in communication efficiency.

Contextual

Simply providing alerts is not enough. Data producers must understand more details about how the data will change in order for it to be truly actionable. This includes information such as:

- What is the data expected to look like versus what does it actually like?
- What is the schema of the data?
- What are the semantics of the data?

- Does the data contain PII or not?
- How is the data actually utilized downstream - a dashboard, a training set for a machine learning model, or something else?

The more information that someone can provide then the easier it is for consumers to be able to act accordingly.

At the right time

The right time of communicating changes is right before something breaks, not significantly before nor after. If someone approaches a producer about changes too far in advance it will be ignored and placed onto the backlog for later. If it happens after the change has been made then usually the engineer has already moved on to greener pastures and is working on more interesting projects. The best time to communicate a breaking change is before it is deployed, not after it's detected– often by others such as downstream data consumers within the business. This allows you to inject at the moment the data producer is most likely to listen.

Including the right people

It is critically important that the right people are brought into the change management process. This refers to the consumers of data who are going to be impacted by an upstream change. These consumers range from highly technical data engineers and data scientists, to non technical product managers, designers, and other business level personnel. These individuals need to be able to interact with data producers in a background agnostic interface, in an abstraction layer outside of products that non technical teams may not have access to, as illustrated in Figure 3-3.

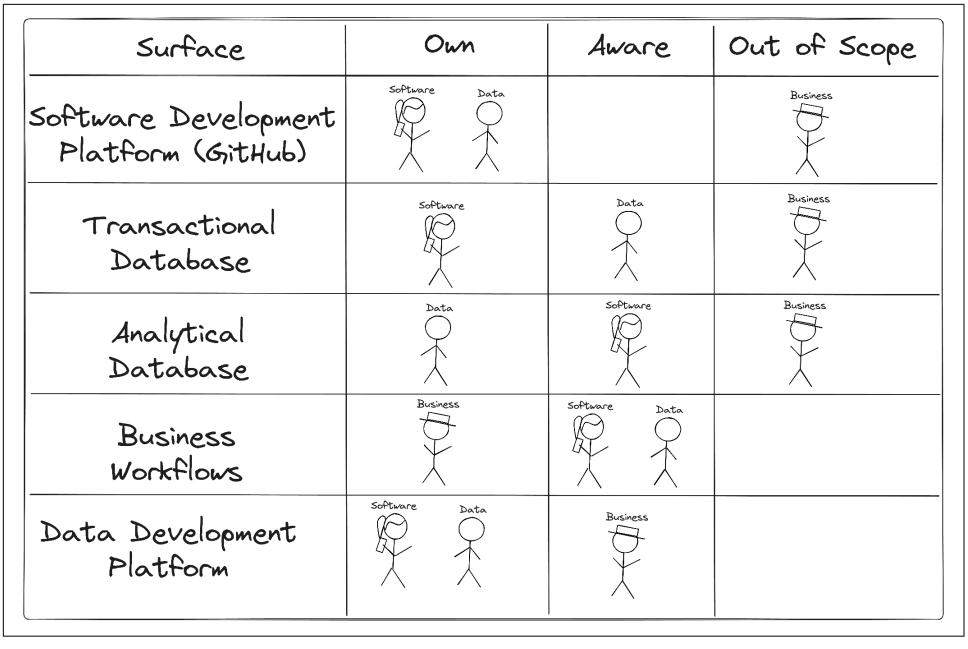


Figure 3-3. : Scope of stakeholders across surfaces within the data lifecycle.

In the next sections, we will cover one of the processes where change management and collaboration is essential: large-scale refactors.

The Cost of Large-Scale Refactors

Refactoring is an expected step within the software lifecycle to continually improve a codebase. Martin Fowler, one of the leading voices on refactoring practices, defines refactoring as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior." (Refactoring: Improving the Design of Existing Code, Chapter 2) Typically, these refactors are opportunistic in that small changes happen within the development workflow, as such changes make it easier to implement new features within the codebase. Yet sometimes, refactors require massive effort if an organization finds itself in too much technical debt.

Refactors that are on the opportunistic side are considered "floss refactoring" as its small efforts over time to maintain cleanliness and prevent issues long-term. On the opposite spectrum are "root canal refactors" which are extensive procedures to deal with a rotting section of the codebase. These root canal refactors are often large-scale refactors required to unblock the codebase's ability to implement new features or make the developer experience less painful. Drs. Murphy-Hill and Black utilized this metaphor of oral hygiene as it best represented the behaviors of software developers,

where it's known that flossing or refactoring is a practice one should do every day, yet many put off the practice and ultimately pay a much higher price later to resolve the issue.

Large-Scale Refactor Considerations

According to a survey conducted by Dr. Ivers et al., responding organizations had spent 1,500 staff days on average working on large-scale refactors. Assuming an average salary of $115K and 260 work days in a year, such companies are spending over $650K a year on large-scale refactors alone. Given this large investment and months to years of time committed, why would an organization consider a large-scale refactor to be worth the effort?

Where "floss refactoring" is considered good code hygiene that's expected among engineers, large-scale refactors are more a business decision rather than a desire to improve the codebase. As the scale of changes and cost to implement a refactor increases, so does the purview of the change in which more senior technical staff need to approve the change, as well as the need for buy-in among non-technical executives. Thus, a clear ROI is needed to warrant the approval of such an initiative. In the same Dr. Ivers et al. survey, respondents noted that the top three reasons for going through with a large-scale refactor were the following:

1. Reducing the cost of implementing a change to the codebase.
2. Increase the speed at which engineers can deliver code.
3. Migrating to a new architecture that is often driven by an external business decision.

One of the survey respondent best captured the pain that pushes organizations to make such a hefty investment in the following quote:

> "...modernization cycle was held back by 4 years....maintenance cost stayed high....cost to implement, deploy, and validate continue to increase."

All of this leads to a poor developer experience that makes it both harder to ship meaningful products for the business and to retain top engineering talent.

Use Case: Alan's Large-Scale Refactor

Finally, Dr. Ivers et al. survey was able to surface seven distinct steps for a large-scale refactor among responding organizations:

1. "Determining where changes are needed"
2. "Choosing what changes to make"
3. "Implementing the changes"

4. "Choosing what changes to make"
5. "Validating refactored code (inspection, executing tests, etc.)"
6. "Re-certifying refactored code"
7. "Updating documentation"

We will apply these steps to Alan, a health insurance company founded in 2016 but needed to conduct a large-scale refactor in 2021 to account for a major shift in their product's assumptions and ultimately their underlying data. These major changes often unearth data quality issues from the previous system, or embed hidden assumptions in the new infrastructure that will potentially become new data quality issues.

For Alan's use case, they made a major assumption in that there is a one-to-one relationship between a company and its contracts (not data contracts) and that a company will only have one contract at a time. This assumption permeated throughout the codebase, documentation, and beyond in non-technical roles. Hindsight is twenty-twenty, but such assumptions are part of the startup journey, where business hypotheses are placed with the goal of trying to disprove them as quickly as possible to iterate to the correct assumption.

1. "Determining where changes are needed"

As a startup grows, the population of available customers increases, and thus more assumptions are broken. In the case of Alan, 2019 was when they first started seeing customers break the one contract assumption, but not enough customers to warrant a large-scale refactor. Thus, Alan strategically took on technical debt by creating multiple company entities for companies that needed multiple contracts. By 2021, the number of companies breaking the one-contract assumption increased enough to warrant a large-scale refactor for anywhere that assumed a one-contract relationship for companies.

2. "Choosing what changes to make"

Alan's engineering team was able to determine that the most important change that would enable multiple company contracts was to update their data model. Before the data model relationship was Company–Contract and was limited to a one-to-one relationship. The refactor aimed to have the relationship changed to Company–ContractPopulation and ContractPopulation–Contract, which enabled one-to-many relationships.

3. "Implementing the changes"

Key to Alan's success was getting their large-scale refactor prioritized on the product roadmap and creating a team to conduct the refactor. With the decision to update the

data model, it now became clear what pieces of code needed to be deprecated, but not easy to do. In one instance, Alan noted how one property was called over 500 times and subsequently inherited by numerous other properties. Due to this complexity, having a single team to manage the learned domain knowledge and changes was essential.

4. "Generating new tests and migrating existing tests"

While testing is unfortunately viewed as a nice to have within data, it's a requirement within engineering workflows. Many engineering teams ascribe to test-driven development, where unit tests are written in conjunction with new code implemented. The same applies to large-scale refactors. In addition to creating new tests, teams also have to account how their refactor will break existing tests. Though a painful and iterative process, initially creating these unit tests is what enables engineers to confidently implement updates, such as a major refactor.

5. "Validating refactored code (inspection, executing tests, etc.)"

With the complexity of a large-scale refactor, it would be wise to also leverage tools to monitor the progress of the refactor, the utilization of the newly refactored components, and new bugs potentially introduced. Alan specifically used an application observability tool and created monitors for each use case they were refactoring. In addition, Alan ran both the old and new implementation to ensure that the refactor did not introduce a deviation from the original code's output.

6. "Re-certifying refactored code"

Though this is not discussed by Alan, I can speak to this from my own experience working in a health-tech startup in the insurance space. Health insurance data is a highly regulated space as it's at the intersection of medical and HR data. Going from a one-to-one to a one-to-many assumption means that the Alan team had to be extremely mindful of the risk of data leakage, or data being exposed in areas it doesn't belong. I would assume that this refactor was a major point of discussion for their yearly SOC 2 security compliance review.

7. "Updating documentation"

Once again, large-scale refactors are not a technical problem but a business problem that expands well beyond the codebase. For five years, the notion of "one active contract per customer" was embedded in the culture, training, and documentation of Alan. Thus, in addition to refactoring the codebase, you also need to "refactor" the company culture by including non-engineering roles in the process. Alan updated documentation for every impacted party and communicated these changes repeatedly to ensure they were adopted outside the codebase as well.

We highly recommend reading Chaïmaa Kadaoui's article on Alan's large-scale refactor to learn more. In the next section we move away from an engineering perspective experienced with large-scale refactors, and instead focus on the type of codebase change often experienced by data professionals: database migrations.

The Dangers of Database Migrations

While software developers focus on building software systems that are maintainable and easy to scale, data developers unfortunately struggle to do such given the amount of unknowns in our workflows in respect to data. Thus, data infrastructure focuses on the ability to not only maintain the codebase but also provide consistent and trustworthy data that can be iterated on. When one's data is unable to do this we call it data debt, and therefore consider the possibility of a database migration. Even with the benefit of reducing data debt, potential pitfalls of database migrations include:

Data Loss

When moving data from database A to database B, there runs the risk of data loss if an error occurs during run time or because of faulty logic. For example, we've experienced instances where data was missing for some of our earlier customers as that specific data's date was greater than the implemented date filter, and thus never made it into the new database. Fast forward a year after the migration and as a data consumer I'm confused as to why I can't answer historical questions for particular organizations.

Introduction of Data Quality Issues

We live by the mantra that data degrades every time you move it. While it's a conservative notion, this skepticism is crucial for being cognizant of the fragile nature of data. In the case of a database migration, you are often moving data at sizes where it's infeasible to determine a 1:1 match between tables and thus have to rely on aggregate metrics such as row counts. Depending on the data, a level of error may be acceptable but this cutoff must be determined by the business need.

Massive Amounts of Change Management

As stated in the large-scale refactor section, changes at this scale is a business decision more than a technical decision. Thus, change management needs to go beyond the technical staff implementing the change but also to impacted stakeholders (e.g. business user reviewing dashboards). Ideally a database migration is an improvement, but regardless change is happening and will require extensive and repeated communication.

Staff Pulled Away From Main Roles

As stated earlier, it's much easier to get budget allocation for revenue driving workflows rather than infrastructure. This lack of prioritization is often why data debt can build for so long before a database migration is put on the roadmap– resulting in many data teams spending too much time being reactive rather than focusing on their main roles. This pitfall is less about the migration, but rather shining a light on how database migrations can grow into a major problem to solve.

Untangling Business Logic is Painful

As noted in the Alan refactor use case above, it was five years before a refactor was implemented. Similarly, database migrations are conducted in roughly 3-5 year intervals (need citation), and in that time team-members leave, assumptions about the business change, and earlier processes are less mature. What results is trying to piece together complex business logic that expands for years, and ensuring it's still maintained in the new database. Doing such perfectly is unlikely, and thus data developers need to be aware of this limitation.

Much like a large-scale refactor, database migrations are a complex and challenging experience for developers.

Despite these pitfalls, organizations still engage with database migrations on a regular basis. Again, such large changes to software and data systems are a business decision rather than a technical decision, and thus the ROI warrants such initiatives. Such considerations include:

Data debt pain

While infrastructure projects are overlooked for revenue-driving projects, it becomes much easier to sell infrastructure projects when the pain of data debt is higher than the pain of migrating. Specifically, as data debt increases, trust in the data decreases and limits an organization's ability to extract value from data. An example of this is a database approaching its memory limit and thus risk data pipelines going down for key executive dashboards.

Changing business models :

The business model changed, thus, there are new requirements for revenue-driving work. For example, a company's product may move from batch reporting to real-time analytics as its main product feature. This completely changes the underlying assumptions of the business and thus the technology and databases need to be migrated to meet technical requirements.

Regulatory changes:

Many companies are experiencing the impact of EU's General Data Protection Regulation (GDPR) on data processing and storage, resulting in major database migrations to adhere to regulations and avoid fees. As the US catches up to Europe for data privacy laws, such as the California Consumer Privacy Act (CCPA), we see database migrations becoming more prominent.

Skyrocketing cloud costs :

One of the earliest wins a new data team can reach is doing an audit of their cloud spend. With uncertainty in the wider market, CFOs are looking at budgets across the organization and seeing the obscene prices of cloud spend. Database migrations can help teams slash cloud spend by migrating to cheaper storage.

Opportunities with new technologies :

Finally, as new technologies emerge, so do new use cases in which data can be used. In 2023 the emergence of large language models (LLM) in production has spurred companies to start looking at vector databases to manage their own LLMs in production, and thus another database migration is needed.

Again, the above points must be viewed within the context of business needs. Therefore, it's essential for data teams to translate how these technical considerations apply to the business as means of change management.

The Role of Change Management in Data Quality

In Chapter 2 we defined data quality as "an organization's ability to understand the degree of correctness of its data assets, and the tradeoffs of operationalizing such data at various degrees of correctness throughout the data lifecycle, as it pertains to being fit for use by the data consumer." A simplified way to view this definition is "an organization's ability to handle change management for processes related to data." Specifically, poor data quality is the symptom of the data, representing a technology and or process, diverging from the real-world truth. Thus, data teams need to be vigilant of both 1) externally driven changes that impact their workflows (e.g. new product feature), and 2) the impact of changes implemented by their data team (e.g. new data transformation logic). While change management is necessary within software engineering as well, it is amplified for data-related processes given its management needs to be handled around both code and data– unfortunately, data is also significantly less stable than code as it's in a constant state of decay. This decaying of data is what makes it difficult to work with as compared to the stability of software systems.

The Entropic Behavior of Data

Much like entropy, data is in a constant state of change and decay from the moment it's recorded. While some data's decay rate is relatively slow, such as a social security number, other forms of data, such as telemetry data, will decay rapidly. Understanding where one's data falls on this spectrum of data entropy is integral to understanding the required change management for your data system. With that said, the decay rate of data does not imply a level of value but rather the constraints of the data for its utilization. As an analogy, chemical elements experience entropy, also known as half-lives which results in valuable quirks for industry use. For example, the slow decay of carbon-14 enables scientists to date artifacts and fossils thousands of years old, while the fast decay of technetium-99m enables its use for medical imaging. Similarly, a social security number always being unique and unlikely to change for an individual makes it a great unique identifier (PII aside). At the same time, the quick decay of telemetry data enables it to be extremely useful for real-time streaming.

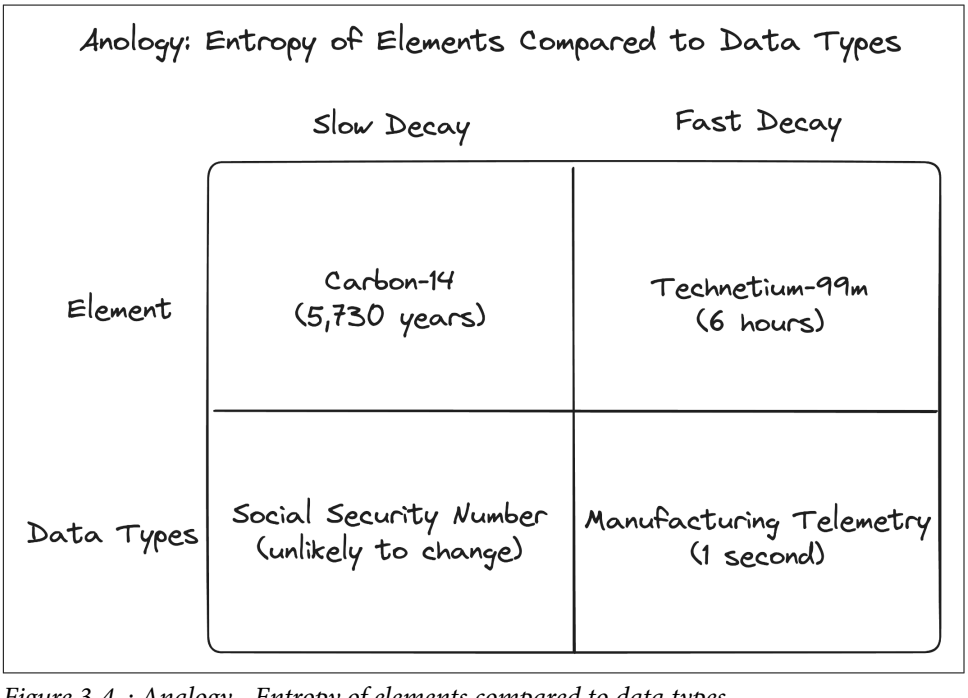


Figure 3-4. : Analogy - Entropy of elements compared to data types.

How Data Drifts from Established Business Logic

Given that data experiences entropy, resulting in organizations needing to manage these changes to continually extract value from data, what type of patterns of change can organizations expect? We once again defer to the field of machine learning, which

has established the concept of "data drift" to describe how changes in data and or understandings of data impact model performance. While there are multiple types of data drift, the subset of "concept drift," best aligns with the lens of data quality and change management.

In the highly cited paper, *Learning under Concept Drift: A Review*, Dr. Jie Lu and colleagues define concept drift as "unforeseeable changes in the underlying distribution of… data over time." As illustrated in Figure 3-5, sourced from the above research paper, concept drift can come in the form of sudden drift, gradual drift, incremental drift, or recurring concepts. Below are real-world examples of these forms of concept drift impacting data.

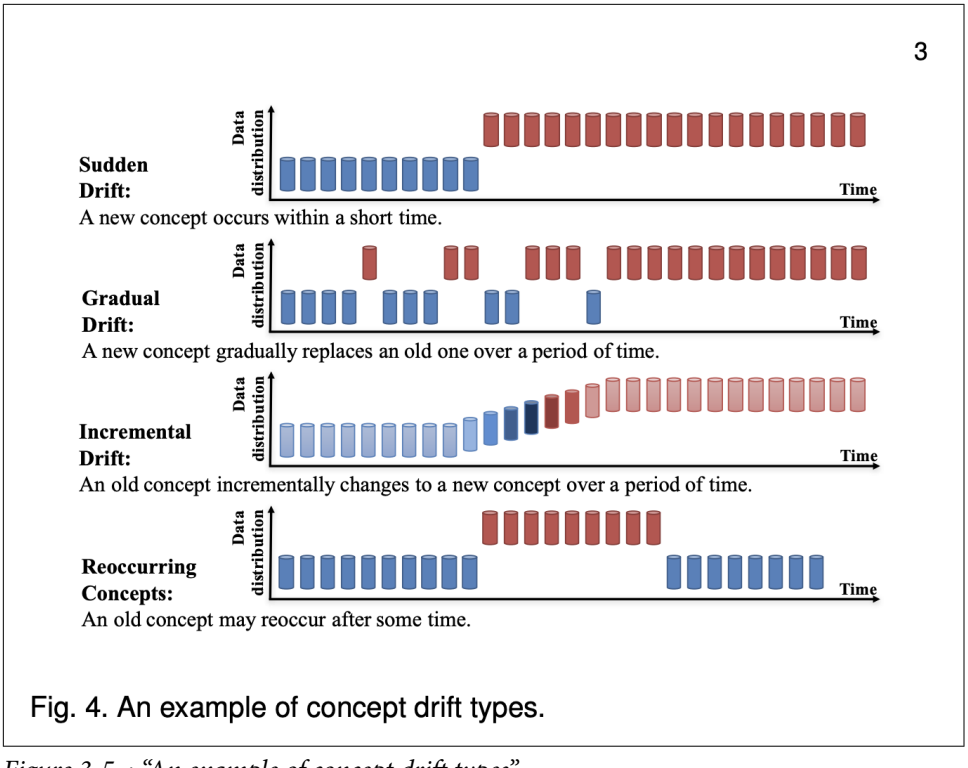


Figure 3-5. : "An example of concept drift types"

Examples of applying the four forms of concept drift to a data quality lens:

Sudden Drift

To become GDPR compliant, user data must be removed or changed to meet privacy requirements. On the extreme side, some US news outlets block EU countries from accessing their websites to avoid the regulation, and thus completely removing a demographic.

Gradual Drift

Over time, a product's target customer changes as a company achieves product-market fit. For example, Netflix moving from DVD sales to streaming and eventually no longer offering DVDs.

Incremental Drift

A userbase's median age gradually changes, such as the gaming company Roblox whose initial demographic was young children. As the same children grew up with the platform, the median age is over thirteen, and open-source projects on the platform reflect that.

Recurring Concepts

Recessions in the economy are simultaneously expected to recur while also being unable to predict precisely. A significant marker of a recession is the drop in discretionary consumer spending; thus, consumer organizations that have existed for decades will have recurring concepts present within their data.

While the four above examples are legit changes to data, they can potentially lead to data quality issues if the data teams are not aligned with the business in their change management.

Change Management Needs to Align with the Needs of the Business for It to Be Accepted

While the above four forms of data concept drift may be clear to a technical team, it is unlikely for non-technical business stakeholders to pick up on these nuances until after they are directly impacted. Specifically, stakeholders outside of data struggle with the abstract nature of data and thus struggle to connect data infrastructure to value. For example, ask a stakeholder about a row in an Excel spreadsheet, and they can quickly provide the specific data about the record. Ask the same stakeholder about 100,000 rows in the same Excel spreadsheet, and they will likely struggle. This same level of scale required for data infrastructure is similarly abstract and far removed from the workflows of the business stakeholder. Thus, data professionals need to clearly map data infrastructure needs to business outcomes as part of their change management processes.

For example, diving deeper into the "sudden drift" GDPR use case, why would US news outlets block EU countries from accessing their websites to avoid the regulation? According to the GDPR regulatory body, "less severe infringements could result in a fine of up to €10 million, or 2% of the firm's worldwide annual revenue from the preceding financial year, whichever amount is higher." In the case of US news outlets whose primary revenue-providing audience is in the US, making the business case for

data quality practices that align with GDPR would fall flat as the data quality initiative doesn't align with their business model. In comparison, for a major US news outlet with a worldwide audience that drives revenue, making the necessary data quality investments is a substantially easier sell.

It's not about data quality for data quality sake, it's about data quality to drive business value. As an additional example, data quality is similar to surveys. While an exhaustive survey of every possible person in a population would ideally get the most accurate results, most don't pursue such methods, given how infeasible or cost-prohibitive it is. The same is true for data quality, and its associated change management, in that perfect data is theoretically ideal, but achieving such is both improbable and not worth the ROI to the business.

Thus, data teams need to work with the business to align data quality practices with the needs of the business. Specifically, data quality infrastructure needs to enable organizations to scale change management via:

- Codifying domain knowledge of business stakeholders with respect to valuable data use cases and disseminating such knowledge across the organization.
- Creating meaningful constraints on the data systems to uphold data standards to the expected domain knowledge.
- Identifying when data drift occurs, what type of drift occurs, and its impact on the business.
- Alerting key stakeholders when drift impacts an agreed-upon threshold to data quality.
- Rectifying data quality or updating the business logic to better match real-world truth after data drift.

These five requirements are why we strongly believe in data contracts as a needed mechanism to enable data quality at scale.

How Infrastructure Needs Change at Scale

When considering the scale of technical systems, developers often primarily emphasize the system's technical capacity over the need to scale people and processes through technology. While technical capacity is essential for meeting established requirements, technical teams need to also account for how new technology will change interpersonal relationships and processes, both among their respective team and other internal stakeholders. Two patterns that best reflect the impact of people and processes on technical scale are Dunbar's Number and Conway's Law. Technical teams that account for these two patterns are best equipped to not only successfully implement scaling projects, but also ensure such projects align with the business.

Dunbar's Number & Conway's Law

Anthropologist Dr. Robin Dunbar posited that respective species have a "...information-processing capacity and that this then limits the number of relationships that an individual can monitor simultaneously. When a group's size exceeds this limit, it becomes unstable and begins to fragment. This then places an upper limit on the size of groups which any given species can maintain as cohesive social units through time." For humans, it is believed that this number of relationships is around 150 people on average, as this number has been observed among businesses, military units, and referenced in popular culture such as Malcolm Gladwell's book *Tipping Point*. Chris Cox, Chief Product Officer at Meta, even noted in a 2016 interview that "It's one of the magic numbers in group sizes…I've talked to so many startup CEOs that after they pass this number, weird stuff starts to happen…The weird stuff means the company needs more structure for communications and decision-making."

Key to understanding this phenomena is that the number of potential relationships within an organization grows exponentially while the number of employees grows linearly– as seen in Figure 3-6. At 150 employees, there are 11,175 potential relationships within an organization, and simply adding 10 employees increased the number of potential relationships by ~1,500. This is roughly ten times more than the increase in potential connections going from 10 to 20 employees. To reiterate, this is why technical teams need to also account for how new technology will change interpersonal relationships, especially as they pass Dunbar's Number.

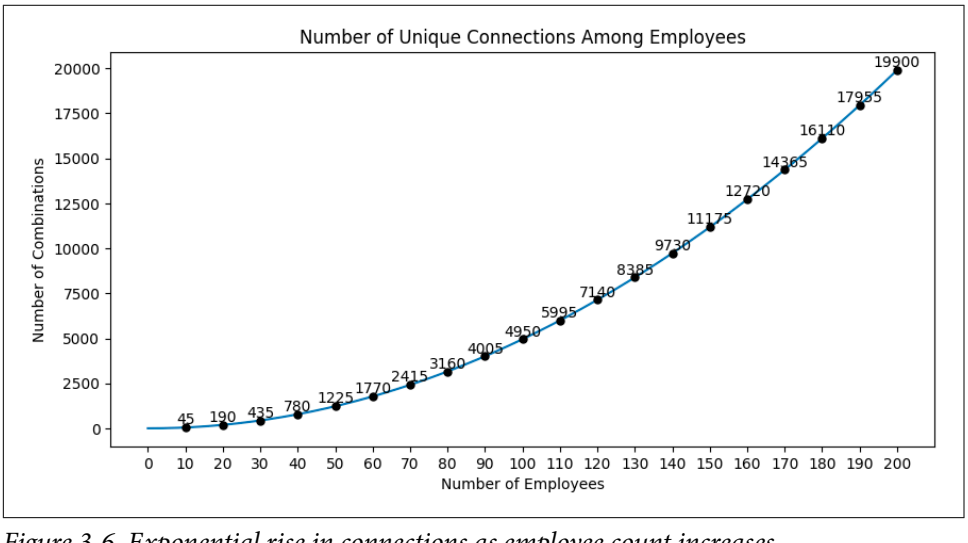


Figure 3-6. Exponential rise in connections as employee count increases.

In addition to the number of connections, how organizations communicate amongst these connections is also important. Dr. Melvin Conway stated in his 1968 paper,

How Do Committees Invent?, "Any organization that designs a system… will inevitably produce a design whose structure is a copy of the organization's communication structure." Even Martin Fowler, who we referenced earlier regarding refactoring, noted how even skeptical engineers accept the power of Conway's Law and that it's "[important] enough to affect every system [he's] come across, and powerful enough that you're doomed to defeat if you try to fight it." Ultimately, these two laws reflect the challenges of maintaining complex technical systems within organizations as they scale– and why change management among software and data developers is so important. A great example of this at play is the organization, Atlassian, who confronted both laws while scaling their engineering.

Case Study: Atlassian Engineering Team

Even engineering teams at vendors that specialize in software development collaboration and communication are still bound to the impacts of Dunbar's Number and Conway's Law. Atlassian, the company behind the widely used issue tracking product Jira, had to account for these phenomena when they scaled their engineering team.

When Atlassian moved their Confluence Cloud team, one of the first things leadership accounted for was Dunbar's Number by scaling down the team into a manageable number. By scaling the team down, engineers were able to build stronger rapport and collaboration practices. During this phase, Atlassian's Confluence Cloud team focused on relationship building and work sharing across teams, ensuring managers had less than five direct reports, and that direct teammates were co-located. Before they even considered scaling the team back up, they ensured that their tools, systems, and processes could handle the increased complexity of adding more people.

While Atlassian had success with overcoming Dunbar's Number, they initially struggled with the power of Conway's Law. Specifically, Atlassian tried a "squads and tribes" model but quickly learned that such a team organization was ineffective for managing complex software systems. The individual groups from the squads and tribes model didn't reflect the system they were trying to build, and engineers struggled to ramp up and be effective wherever they moved to a different group within the organization. From this hard lesson, Atlassian instead focused on being able to pick up signals that their team organization didn't match up with the system they were trying to build, or when there were duplicative efforts. In instances where such signals were picked up, emphasis was placed on having a single team work on duplicative systems as a catalyst to have the two systems converge and reduce complexity.

Atlassian's use case expands beyond these two examples, and we highly encourage you to learn more via their published blog written by Stephen Deasy.

How Data Contracts Enable Change Management at Scale

In summary, Dunbar's Number implies that communication and relationships start to break down as an organization reaches 150 employees. This level of scale requires new processes and team structures to enable effective communication while scaling; especially when building complex technical systems such as those that rely on data. Furthermore, Conway's Law emphasizes the impact of an organization's communication and team structure on the output of its produced system.

Together, Dunbar's Number and Conway's law highlights that scaling complex data systems cannot be achieved via technical requirements alone. Furthermore, simply scaling a team or process only adds further complexity and thus results in a breakdown of systems. We argue that data teams need to first scale their ability to communicate and collaborate amongst a growing team, before they focus on scaling their technical requirements. Specifically, data contracts enable organizations to scale collaboration and change management within data systems.

To illustrate this, please refer to Figure 3-7 where we have an example of an organization network where the nodes represent individuals and the lines represent a connection between two individuals. When an issue is identified by a specific node, it doesn't happen in a vacuum but is instead tied to the contingencies of their respective network--such as an upstream data producer changing the schema of a data asset and thus breaking downstream dashboards. Without data contracts, this individual has a plethora of first and second degree connections in which the issue needs to be coordinated with. This often looks like widespread messages to teams either informing of an issue or requesting help. Data contracts significantly limit the problem scope by programmatically monitoring changes for known constraints, keeping track of the data asset owners, and only notifying relevant parties. Thus, data contracts enable organizations to overcome the limitations of Dunbar's Number by limiting the number of relationships an individual needs to keep track of. In addition, data contracts leverage the power of Conway's Law to optimize communication among relevant parties within complex systems.

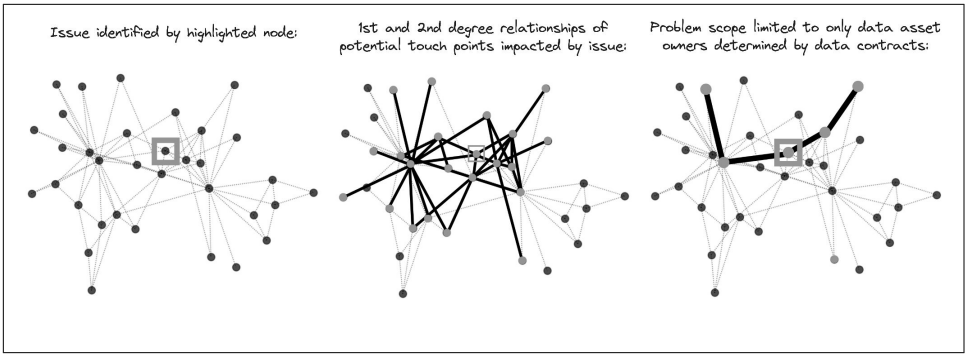


Figure 3-7. : The degrees of relationships in resolving issues.

Going back to Martin Fowler, he also noted that "If I can talk easily to the author of some code, then it is easier for me to build up a rich understanding of that code. This makes it easier for my code to interact, and thus be coupled, to that code. Not just in terms of explicit function calls, but also in the implicit shared assumptions and way of thinking about the problem domain." This gets to the crux of the problem that data contracts are aiming to solve. With respect to data change management, data contracts are the "glue" that holds people and processes together among increasingly complex technical systems.

Conclusion

In this chapter we discussed the challenges of the everyday workflows of data producers and consumers in building and maintaining complex software and data systems. In summary, this chapter covered:

- How data development is different from software development.
- The implications of these differences on our ability to drive value with data in an organization.
- How businesses approach large-scale refactors such for managing technical debt and database migrations.
- Why scaling makes managing incredibly difficult, and how Dunbar's Number and Conway's law gives insight as to why it's so challenging.

Among all of these challenges, we argue that data contracts are necessary for scaling these workflows among technical teams and managing the necessary change management of data workflows. In the next chapter, we begin to go in-depth as to what data contracts are and its various components.

Additional Resources

- Industry Experiences with Large-Scale Refactoring
- James Ivers Presentation, Scaling Refactoring (October 14, 2022)
- "Refactoring Tools: Fitness for Purpose" by Emerson Murphy-Hill and Andrew P. Black
- Software Engineering at Google - Large-Scale Changes
- Is This the End of Data Refactoring? - The New Stack
- Database Refactoring: Improve Production Data Quality
- Refactoring Databases: Evolutionary Database Design
- What we learned from a large refactoring | by Chaïmaa Kadaoui | Alan Product and Technical Blog | Medium

References

n.d. General Data Protection Regulation (GDPR) Compliance Guidelines. Accessed December 17, 2023. https://gdpr.eu/.

"." 2022. . - YouTube. https://arxiv.org/pdf/2004.05785.pdf.

"," 2023. , - YouTube. https://dl.acm.org/doi/10.1145/3540250.3558954.

"," 2023. , - YouTube. https://www.sciencedirect.com/science/article/abs/pii/004724849290081J.

"," 2023. , - YouTube. https://www.sciencedirect.com/science/article/abs/pii/004724849290081J.

"California Consumer Privacy Act (CCPA) | State of California - Department of Justice - Office of the Attorney General." 2023. California Department of Justice. https://oag.ca.gov/privacy/ccpa.

Conway, Mel. n.d. "How Do Committees Invent?" Mel Conway's. Accessed December 17, 2023. https://www.melconway.com/Home/Committees_Paper.html.

"Data to engineers ratio: A deep dive into 50 top European tech companies." n.d. Towards Data Science. Accessed December 17, 2023. https://towardsdatascience.com/data-to-engineers-ratio-a-deep-dive-into-50-top-european-tech-companies-58abc23e36ca.

Deasy, Stephen, and Ashley Faus. 2020. "3 research-backed principles that help you scale your engineering org - Work Life by Atlassian." Atlassian. https://www.atlassian.com/blog/technology/3-research-backed-principles-scaling-engineering.

Delaney, Kevin J. 2016. "Something weird happens to companies when they hit 150 people." Quartz. https://qz.com/846530/something-weird-happens-to-companies-when-they-hit-150-people.

Fowler, Martin. 2022. "ConwaysLaw." Martin Fowler. https://martinfowler.com/bliki/ConwaysLaw.html.

"Half-Lives and Radioactive Decay Kinetics." 2023. Chemistry LibreTexts. https://chem.libretexts.org/Bookshelves/Physical_and_Theoretical_Chemistry_Textbook_Maps/Supplemental_Modules_(Physical_and_Theoretical_Chemistry)/Nuclear_Chemistry/Nuclear_Kinetics/Half-Lives_and_Radioactive_Decay_Kinetics.

Jensen, Mathias. 2022. "What we learned from a large refactoring | by Chaïmaa Kadaoui | Alan Product and Technical Blog." Medium. https://medium.com/alan/what-we-learned-from-a-large-refactoring-85291cb4457c.

"Mapping decline and recovery across sectors." 2009. McKinsey. https://www.mckinsey.com/capabilities/strategy-and-corporate-finance/our-insights/mapping-decline-and-recovery-across-sectors.

McCracken, Harry. 2023. "Roblox's metaverse is growing up." Fast Company. https://www.fastcompany.com/90850387/roblox-users-growing-up.

""Refactoring Tools: Fitness for Purpose" by Emerson Murphy-Hill and Andrew P. Black." n.d. PDXScholar. Accessed December 17, 2023. https://pdxscholar.library.pdx.edu/compsci_fac/109/.

Sarandos, Ted. 2023. "Netflix DVD - The Final Season." About Netflix. https://about.netflix.com/en/news/netflix-dvd-the-final-season.

Satariano, Adam. 2018. "U.S. News Outlets Block European Readers Over New Privacy Rules (Published 2018)." The New York Times. https://www.nytimes.com/2018/05/25/business/media/europe-privacy-gdpr-us.html.

CHAPTER 4

An Introduction to Data Contracts

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

The previous three chapters focused heavily on the *why* of data contracts and the problem statement that data contracts aims to solve. Starting in Chapter 4, we shift to defining what exactly the data contract architecture is and what its workflow looks like. This chapter will serve as the theoretical foundation before transitioning into discussing real-world implementations in Chapter 5 and implementing it yourself in Chapters 6 and 7. In addition to the theoretical foundation, we will also discuss the key stakeholders and workflows when utilizing data contracts.

Collaboration is Different in Data

Depending on who you ask, 'collaboration' could either refer to a nebulous executive buzzword or a piece of software that makes vague and lofty promises about improving the relationships between team members who already have quite a bit of work to do.

For the purposes of this book, our definition of technical collaboration is the following: "Collaboration refers to a form of distributed development where multiple individuals and teams, regardless of their location, can contribute to a project efficiently and effectively."

Teams may collaborate online or offline, and with comprehensive working models or not. The goal of collaboration is to increase the quality of software development through human-in-the-loop review and continuous improvement.

In the modern technology ecosystem, there are many components to collaboration that have become part of the standard developer lifecycle and release process. Version control, which allows multiple contributors to work on a project without interfering with each other's changes, is a fundamental component of collaboration facilitated through source code management systems like Git.

Pull Requests (PR) are another common collaborative feature that have become an essential component of DevOps. Contributors can make changes in the code branches they maintain, then propose these changes to the main codebase using PRs. PRs are then reviewed by software engineers on the same team in order to ensure high code quality and consistency.

With automation, collaboration around the code base can provide short feedback loops between developers that ensure bugs are mitigated prior to release, the team has a general understanding of which changes are being merged to production and how they impact the codebase, ensuring greater accountability for releases which are now often made multiple times per day.

All of this is ultimately done to improve code quality. If you are an engineer, imagine what would life be like if your company wasn't using software like GitHub, GitLabs, Azure DevOps, or BitBucket? Teams would need to make manual backups of their code base and leverage centralized locking systems to ensure only one developer was working on a particular file at a time. Teams would also have to consistently check in with each other to plan which aspects of the codebase were being worked on by whom, and all updates would be shared through email or FTP systems. With space for human error, a significant amount of time would be spent dealing with bugs and software conflicts as illustrated in Figure 4-1.

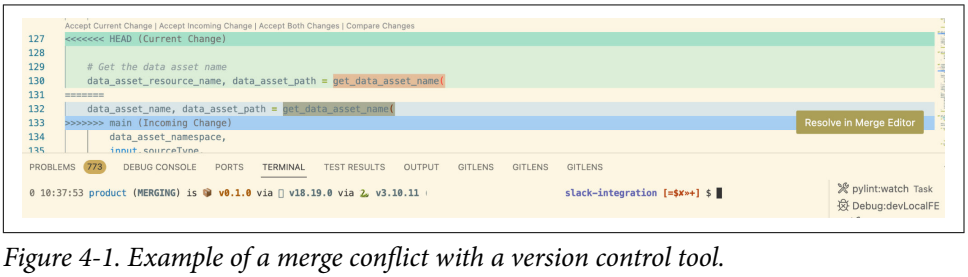


Figure 4-1. Example of a merge conflict with a version control tool.

However, effective collaboration is not only about implementing core underlying technology—it ultimately hinges on the interaction patterns between change-makers and change-approvers. These interaction patterns will differ depending on the change being made, the significance of the change, and the organizational design of the company.

GitHub, the world's most popular version control system, was launched in 2008 by Tom Preston-Werner, Chris Wanstrath, and PJ Hyett. By 2011, GitHub was home to over 1 million code repositories, and grew tenfold over the following two years. GitHub not only solved a significant technical challenge in creating a simple to use interface for Git and providing a host of collaboration oriented functionality, but it also allowed leadership teams who were excited by the principles of AGILE software development to meaningfully switch to new organizational structures that were better in line with rapid, iterative release schedules and federation.

These organizational structures placed software engineers at the center of *product teams*. Product teams were composed of a core engineering team, reinforced by a variety of other roles such as product and program managers, designers, data scientists, and data engineers. Each product team acted as a unique microorganism within the company, pursuing distinct goals and objectives that, while being in line with the company's top level initiatives, were tailored to specific application components. GitHub allowed developers on these teams to further subdivide their tasks, many of which overlapped, without consequence.

Despite federated collaboration becoming a core component of a product development team's workflow, this same cycle of review and release is not present for data organizations, or software engineers who maintain data sources such as production databases, APIs, or event streams. This is primarily because software engineers operate within teams, while data flows *between* teams, as illustrated in Figure 4-2 .

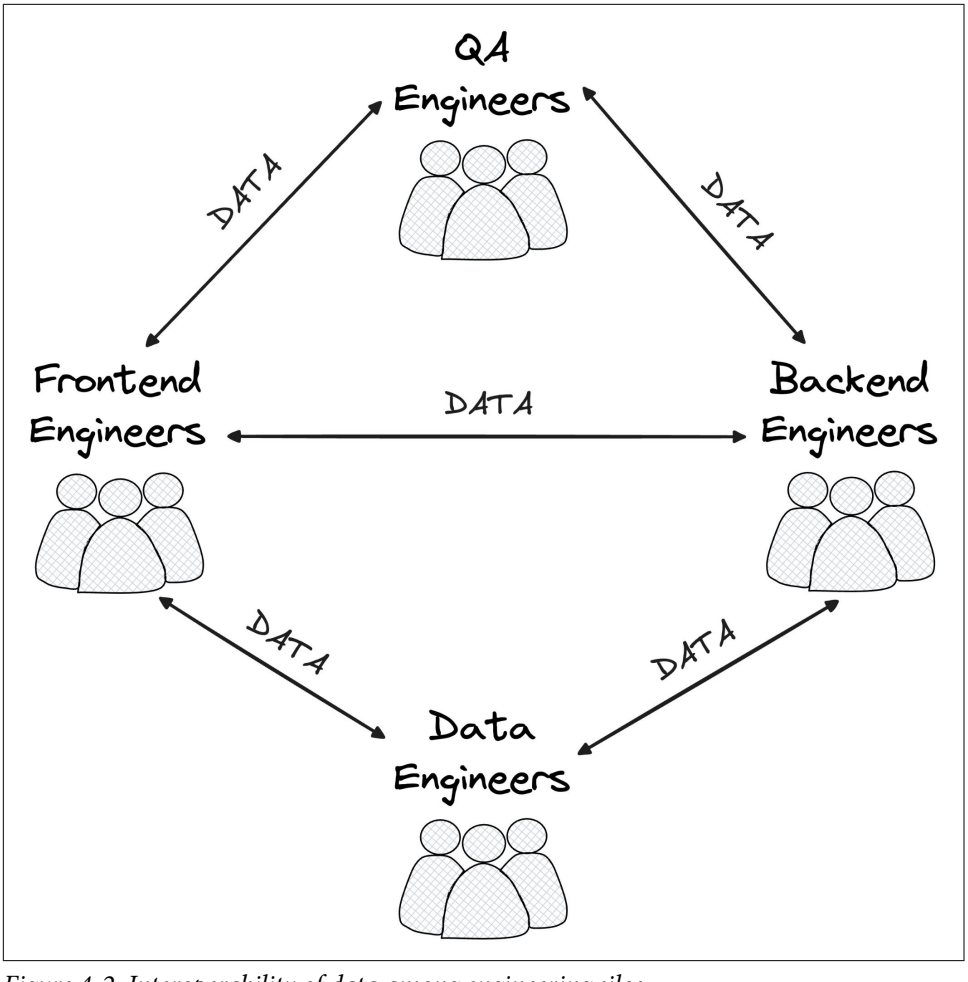


Figure 4-2. Interoperability of data among engineering silos.

Each member of a product team has an in-built incentive to take collaboration seriously. Bad code shipped by an engineer could cause a loss of revenue or delays to the feature roadmap. Because product teams are judged according to their component level goals, each engineer is equally accountable to changes in their codebase.

However, data is a very different story. In most cases, data is stored in source systems owned by product teams which flow downstream to data developers in finance, marketing, or product.

Upstream and downstream teams have unique goals and timelines. Due to the highly iterative and experimental nature of constructing a query (or, asking/answering a question) it is often unclear at the moment the query is created whether or not it

will be useful. This leads to a split in the way data is being used and a divergence of responsibilities and ownership.

When upstream software engineers make changes to their database, these changes are reviewed by their own team but *not* the downstream teams that leverage this data for a set of totally different use cases. Thanks to the isolated nature of product teams, upstream engineers are kept in the dark about how these changes affect others in the company, and downstream teams are given no time to collaborate and provide feedback to their data providers upstream. By the time data teams detect something has changed,it is already too late—pipelines have been broken, dashboards display incorrect results, and the product team who made the change has already moved on.

The primary goal of data contracts is to solve this problem. *Contracts are a mechanism for expanding software-oriented collaboration to data teams, bringing quality to data through human-in-the-loop review, just as the same systems facilitated code quality for product teams.* The next section goes into more detail about who these stakeholders are and how they collaborate.

The Stakeholders You Will Work With

As stated in the previous section, data contracts center around collaboration in managing complex systems that leverage software and data. To best collaborate, one must first understand who they are working with. The following section details the various stakeholders and their role in the data lifecycle in respect to data contracts.

The Role of Data Producers

Thus far we have been talking quite a bit about the gaps between data producers and data consumers which culminate in data quality issues. But before we go further into the *why* of data contracts as a solution, it is essential to understand the responsibilities of all those who are involved with collecting, transforming, and using data. By understanding the incentive structures of key players in this complex value chain of data movement, we can better identify how technology helps us address and overcome challenges with communication between each party.

Data does not materialize from thin air. It must be explicitly collected by software engineers building applications leveraged by customers or other services. We call engineers responsible for collecting and storing this data, *data producers*. While unique roles like data engineers can be both producers and consumers, or non-technical teams such as sales people or even customers themselves can be responsible for data entry, for now we will focus primarily on the role of software engineers who almost exclusively take on the work of most data producers in a company.

Most structured data can be broken down into a collection of *events* which correspond with actions undertaken either by some customer leveraging the application

or the system itself. In general, there are two types of events which are useful to downstream consumers.

Transactional Events:
: Transactional events refer to individual transaction records logged when a customer or technology takes some explicit behavior that typically (but not always) results in a change in state. An example of a transactional event might be a customer making an order, a back-end service processing and validating that order, or the order being released and delivered to a customer's address. Transactional events are either stored in Relational Database Management Systems (RDBMS) or emitted using stream processing frameworks such as Apache Kafka.

Clickstream Events:
: Clickstream events are built to capture a user's interactions with a web or mobile application for the purpose of tracking engagement with features. An example of a clickstream event might be a user starting a new web session, a customer adding an item to their cart, or a shopper conducting a web search. There are many tools available to capture clickstream events such as Amplitude, Google Analytics, or Segment.

Clickstream events are typically limited in their shape and contents by Software Development Kits provided by 3rd party companies to emit, collect, and analyze behavioral data. These events are often stored in the vendor's cloud storage environment where product managers and data scientists can conduct funnel analysis, perform A/B tests, and segment users into cohorts to see behavioral trends across groups. Clickstream data can then be loaded from these 3rd party providers into 1st party analytical databases such as Snowflake, Redshift, or Big Query.

Transactional events depend entirely on the role of the applications and can vary wildly in terms of their schema, data contents, and complexity. For example, in some banks, transactional events may contain 100s or even 1000s of columns with detailed information about the transaction itself, the account, the payer, the payee, the branch, the transaction channel, security information (which is highly leveraged for fraud detection models), any fees or changes, and much more. In many cases, there is so much data recorded in transactional events that much of a schema goes unused or is simply not leveraged in a meaningful way by the service.

Because transactional events record data that is seen as only being relevant to the functioning of the application owned by the engineering team (as opposed to clickstream events, which are designed specifically for product analytics) it is often said that transactional data is a *by-product* of the application. Data producers treat their relational databases as an extension of their applications, and will generally modify them using controlled integration & controlled deployment (CI/CD) processes similar to their typical software release process.

In the modern data environment, data producers' role stops once the data has been collected, processed, and stored in an accessible format. In the past, the responsibility of the data producer was also heavily influenced by design and *data architecture*. However, in federated product environments this function is becoming increasingly rare and seen as unneeded. It is easier and faster for the data producer to collect whatever data they would like from the application, store it all in their database in the most beneficial format for their application, and then make the data accessible to data consumers so that they might leverage the data for analysis.

To be clear, this is not a statement on what we think is right or wrong, but simply how a significant amount of data producers in technologically advanced organizations actually behave.

The Role of Data Consumers

A data consumer is a company employee that has access to data provided by a data producer, and leverages that data to construct pipelines, answer questions, or build machine learning models. Data consumers are split into two categories, technical and non-technical.

Technical data consumers are what you might call data developers. Data developers are engineers or analysts that leverage data-oriented programming languages like SQL, Spark, Python, R, and others to either analyze data in large volumes in order to discover trends and insights, or construct pipelines which allows data to flow through a series of transformations on a schedule before being leveraged for decision making. Technical data consumers deal with the nitty gritty of data management, from data discovery, query generation, documentation, data cleaning, data validation, and much more.

Non-technical data consumers are often business users or product managers that lack the technical ability to construct queries themselves (or can only do so on pre-cleaned and validated data). This category of user typically are the ones asking the questions, and ultimately hold the decision making power on which outcome should be taken depending on the directionality of the data.

For the remainder of this chapter, we'll be focusing on the technical data consumer, though we'll get back to the role non technical teams play later. There are many different types of technical data consumers and where they fit into the data lifecycle as illustrated in Table 4-1:

Data Engineers:

Data Engineers are most commonly known as *software engineers with a data specialty*. They focus on extracting data from source systems, moving data into cheap file storage (such as data lakes or delta lakes) then eventually to analytical databases, cleaning and formatting the incoming data for proper use,

and ensuring pipelines have been orchestrated properly. Data Engineers form the backbones of most data teams and are responsible for ensuring the entire business gets up-to-date high quality data on a schedule.

Data Scientists:
Once one of the sexiest jobs in the world, data scientists are computer engineers with a higher focus placed on statistics and machine learning. Data scientists are most frequently associated with creating features to be leveraged by ML models, conducting and evaluating hypothesis tests, and developing predictive models to better forecast the success of key business objectives.

Data Analyst:
Data analysts are SQL specialists. They primarily leverage SQL to construct queries in order to answer business questions. Analysts are on the front-lines of query composition - it is their responsibility to understand where data is flowing from, if it's trustworthy, and what it means.

Analytics Engineers:
Analytics engineering is a relatively new discipline that revolves around applying software development best practices to analytics. Analytics Engineers or BI Engineers leverage data modeling techniques and tooling like dbt (data built tool) to construct well defined data models that can be leveraged by analysts and data scientists.

Data Platform Engineer:
Even newer than analytics engineers, platform engineers are responsible for the implementation, adoption, and ongoing growth of the data infrastructure and core datasets. Platform engineers often have data engineering or software engineering backgrounds, and are mainly concerned with selecting the right analytical databases, streaming solutions, data catalogs, and orchestrations systems for a company's needs.

Table 4-1. Table of data consumers and their data lifecycle touch points

Data Consumer Role	Data Infra Management	Data Ingestion	Transaction Data Operations	Data Replication	Analytical Data Operations	Analytics, Insights, and Dashboards	ML Model Building	Actioning on Insights and Predictions
Data Platform Engineer	X	X	X	X	X			
Data Engineer		X	X	X	X			
Analytics Engineer				X	X			

Data Analyst	X			
Data Scientist	X	X		
Internal Business Stakeholder				X
External User of Data Product				X

Most technical data consumers follow a similar workflow when beginning a new project:

1. Define or receive requirements from non-technical consumers
2. Attempt to understand what data is available and where it comes from
3. Investigate if data is clearly understood and trustworthy
4. If not, try to locate the source of the data and set time with data producers to better understand what the data means and how it is being instrumented in the app
5. Validate the data
6. Create a query (and potentially a more comprehensive pipeline) that leverages valuable data assets

Data consumers access data through a variety of different channels depending on their skillset and background. Data engineers extract data in batch from multiple sources using open source ELT technologies like Airbyte, or or closed source tooling such as Fivetran. Data engineers can also build infrastructure which moves data in real-time. The more common example of this is Change Data Capture (CDC). CDC captures record level transformations in upstream source systems before pushing each record into an analytical environment, most commonly using streaming technologies such as Apache kafka or Redpanda. Once data arrives in an analytical environment, data engineers then clean, structure, and transform the data into key business objects that can be leveraged by other consumers downstream.

Data Scientists, Analytics Engineers, and Analysts typically deal with data after it has already been processed by data engineers. Depending on the use case, their most common activities might include dimensional modeling and creating data marts, building machine learning models, or constructing *views* which can then be used to power dashboards or reporting.

The output of most data consumers work are *queries*. Queries are code, written in a language designed for either analytical or transactional databases - most commonly SQL, though Python is also a popular data science alternative. Depending on the complexity of the operation queries can range from simple 5-10 line code blocks to incredibly complex files with hundreds or even thousands of lines full of CASE WHEN statements. As these queries are leveraged to answer business questions they might be extended, replicated, or other modified in a variety of ways.

The Impact of Producers and Consumers

As you might imagine, data producers play a large role in the day to day work of data consumers. Data producers control the creation of and access to source data. Source data represents as close to the ground truth as possible, given that it is collected directly from applications. Data consumers often prefer to use upstream data because re-leveraging queries built by other data consumers can be challenging. The meaning of queries is ultimately subjective - a data scientist may have an opinion on what makes an active customer 'active' or a lost order 'lost.' These opinions are often baked into the query code with very little explanation or documentation provided. Depending on the length and complexity of the query, it might be almost impossible for other data scientists to fully understand what one of their fellow data team members meant.

This problem is exacerbated with time. Data consumers regularly leave their place of employment and often take the knowledge of their code along with them. This is called *institutional knowledge*.

For these reasons, data consumers love going to the source. In the same way it's always easier to get information about a particular person directly from that individual than rely on rumors and hearsay. However, going to the source can be challenging. This can require understanding the lineage of your data ecosystem. Lineage refers to the spider-web of connections that tie data assets to each other within an analytical environment. The older and more dense the environment, the more challenging it can be to track lineage across nodes in the graph. The lineage graph creates an additional problem: Change management.

As data producers update their application they regularly make changes to the code of their software. Software changes may or may not affect data structures, such as schema or the contents of data generating objects like transactional events, logs, or analytical events. Because there is no baseline which sets the expected state of these data objects, data producers make their changes effectively *in the dark*, as illustrated in Figure 4-3. While integration testing can help check integration errors with the code itself, and production management software like LaunchDarkly (feature management) or DataDog (Observability) can detect or prevent issues from degrading

a customer experience, this quality assurance only applies to the application layer, NOT the data layer, where data consumers do their work. The data layer is hidden behind the lineage graph. Such convoluted and tangled data environments make it very difficult for application developers to understand how and where their data is being used.

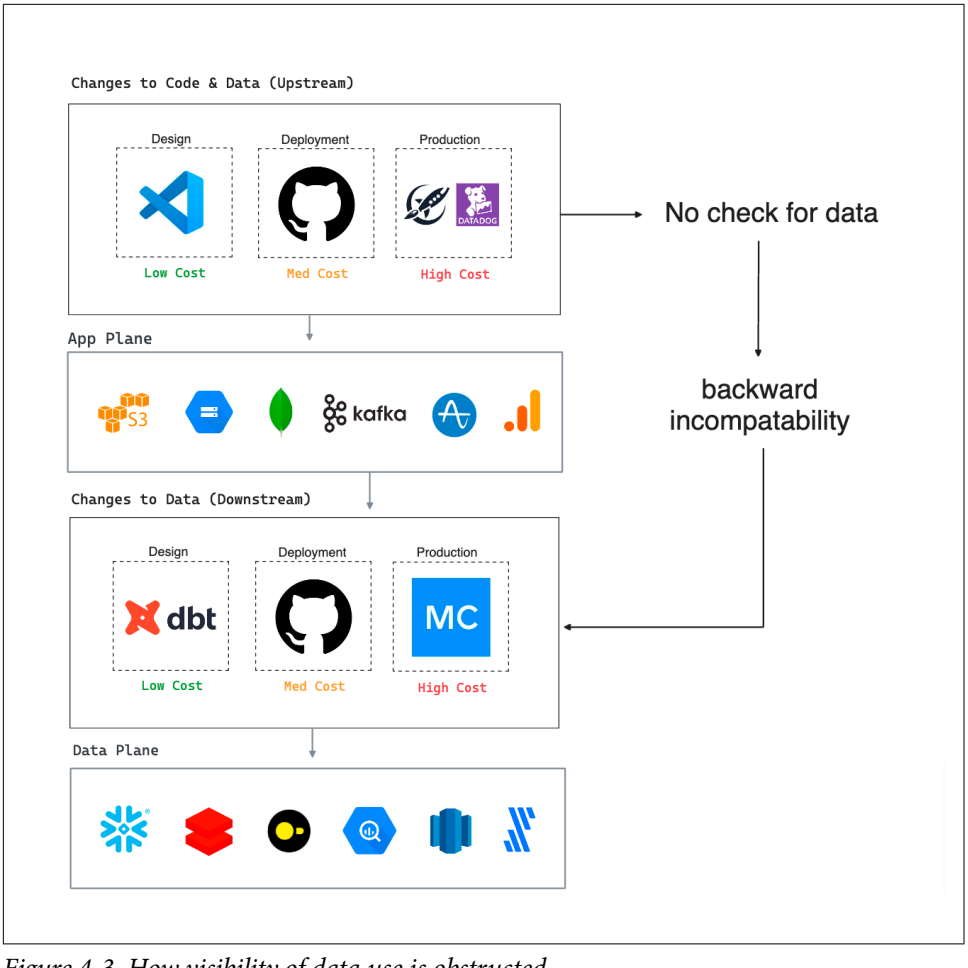


Figure 4-3. How visibility of data use is obstructed.

One option for data producers might be to treat ANY impact to the data layer as part of CI/CD. Unfortunately, this rarely works out well. At most large scale businesses the amount of data in a data lake is so large that it is rare for even 25% of the data to have meaningful utilization within the company. Slowing down your engineering team to ensure integration testing for data that isn't even relevant or useful for data consumers is a waste of time. Producers must be free to iterate so long as they have limited dependencies.

Second, oftentimes backwards incompatible changes are hard requirements. Product teams regularly ship new features, refactor old code, and modify events according to a more grand architectural design which has buy-in from across the organization. While these changes may catch downstream consumers unaware, they often have too much momentum to be stopped. Data producers might attempt to communicate migrations to data consumers by sending emails, announcing their intentions during design reviews, or posting plans in Slack channels. Unfortunately, this feedback loop rarely reaches data consumers who (again, due to complex lineage) rarely realize they will be impacted by an upstream change even if they knew the context.

When change does happen, it is rarely detected in advance. More often than not downstream consumers (and in the worst case scenario, not technical business stakeholders) are the first to notice a problem exists, which sets them on a long-winded goose chase of identifying and resolving errors caused by an upstream team which barely knows they even exist.

The Trials and Tribulations of Data Consumers Managing Data Quality

Note to editors - based on the following article: https://dataproducts.substack.com/p/the-data-quality-resolution-process

Most data practitioners will have a chill run down their spine when they hear the phrase "These numbers look off." Such statements often lead to hours or days of digging into data and their respective systems to unearth and fix data quality issues. Often, data teams are the "face of failure" for these data quality issues within the business, despite many issues arising from upstream changes outside of their control. In Mark's time in startups as a data scientist and data engineer, he came up with a repeatable pattern in solving these data quality issues in organizations where data maturity was relatively early. While this process is relatively manual, many organizations find themselves in similar positions. The data quality resolution process consisted of the following steps, which we have written in more detail about previously:

- "0. Stakeholder Surfaces Issue"
- "1. Issue Triage"
- "2. Requirements Scoping"
- "3. Issue Replication"
- "4. Data Profiling"
- "5a. Downstream Pipeline Investigation"
- "5b. Upstream Pipeline Investigation"
- "5c. Consult Technical Stakeholders"

- "6. Pre-Deploy - Implement DQ Fix"
- "7. Deploy - Implement DQ Fix"
- "8. Stakeholder Communication"

Note that a majority of these steps are not technical but rather center around communication across the business to triage breaks within the data lifecycle and coordinate a solution among multiple teams. In more detail, each step consists of the following from the perspective of the analytical database:

Stakeholder Surfaces Issue:
: Best case scenario is that a data consumer, such as a data analyst, surfaces a data quality issue before downstream business stakeholders notice. Getting ahead of the data quality issue is less about avoiding the downstream users noticing, but rather making sure your stakeholders are confident that you are handling issues in a timely manner. How you respond to such requests shapes the data culture among your stakeholders.

Issue Triage:
: While it's important to respond quickly, data teams should not try to solve the data issue until they properly triage the issue as an urgent fix, assign for later, or to not work on it. Key to this is having managers on the data team serve as a buffer and set expectations. In addition, requests should never be accepted within individual chat channels (e.g. direct Slack message) but rather directed towards a shared channel with visibility such as Jira.

Requirements Scoping:
: Mark's early career mistakes in data often stemmed from jumping straight into solving the problem without properly scoping the issue. Often there are tradeoffs between the effort and impact of the fix, and thus you need to consult with the downstream stakeholder to understand why this data is important and how it impacts their workflows. In addition, this step further fosters trust with your stakeholder as it shows the due diligence you are taking as well as including them in the resolution process.

Issue Replication:
: Once the problem is scoped, problem replication provides the first clues as to where the data quality issue is stemming from. In addition, it prevents one from pursuing a data quality issue that is the result of a human error– which instead implies a communication or process issue. Typically this can be done by either using the data product in question or pulling data from the source table via SQL.

Data Profiling:
: At this stage one can do a series of aggregates and cuts of the tables in question. While not exhaustive, and context specific, the following are great starting points:

- Data Timeliness
- Null Patterns
- Data Count Spikes or Drops
- Counts by Aggregate (e.g. Org Id)
- Reviewing the data lineage of impacted tables

The goal isn't to find the issue, but rather reduce the scope of the issue surface so you can deep dive in a targeted fashion. These quick data searches become the hypotheses to test out.

Downstream Pipeline Investigation:

With the hypotheses created from data profiling, assumptions are tested via downstream investigation in the analytical databases and data products. While the issue may be caused by an upstream change, its visibility is often most present downstream. Again, emphasis is building a complete picture of the data quality issue and potentially discovering other surfaces impacted by the issue that was not in the original problem. Two common issues that surface in this stage are the following:

- A bug was introduced in the SQL transformation code such as a misuse of a JOIN, a WHERE clause missing an edge case, or pulling data from the wrong table (e.g. *user_table* vs. *user_information_table*).
- The SQL code no longer aligns with evolving business logic, and thus transformations need to be updated accordingly.

Upstream Pipeline Investigation:

After exploring downstream impacts, the next step is to trace the data lineage upstream to the transactional database. Specifically, going beyond the database and reviewing the underlying code generating data, conducting the CRUD operations, and capturing logs. In this step, lineage moves from tables and databases to instead also reviewing function calls and the inheritance of these functions. For example, the `product_table` in the database is updated by the `product_sold_count()` function within the `ProductSold` class within the `product_operations.py` file (example below):

```
 # product_operations.py example code
import db_helper_functions as db_helper
class ProductSold:
    def __init__(self) -> None:
        # Connect to the database
        self.db_connection = db_helper.connect_to_database()
        pass
    def product_sold_count(self, product_id, sold_count):
        # Update the product_table in the database with new count
        <python code implementing logic>
```

```
# Update database
db_connection.commit()
db_connection.close()
```

Often the most "hidden" data quality issues lurk within the codebases outside the scope of the data team. Without this step, data teams many times create additional transformations downstream to deal with it quickly.

Consult Technical Stakeholders:

With the facts collected, one may think the solution is apparent, but knowing the solution is only half the battle if the source issue is outside your technical jurisdiction (e.g. limited read-write access to the transactional database). The other half is convincing upstream teams that the proposed solution is correct and that it's worth prioritizing over their current work. Thus, emphasis is on *consulting* (rather than *requesting*) with stakeholders to make them a stakeholder in the solution and understand where it fits within their priorities. Furthermore, there may be some nuances that only those who often work in the upstream system would be aware of.

Pre-Deploy - Implement DQ Fix:

Once a solution is determined, take the results from the data profiling stage for a baseline reference and identify which values should be changing and staying the same. This is key in both ensuring that one's solution does not introduce further data quality issues, and to document the due diligence of resolving the issue for your impacted stakeholders.

For downstream data quality issues, this typically looks like changing the underlying SQL code making the transformations until one is satisfied with the expected behavior of the data. Ideally these SQL files are version controlled via a tool like Data Build Tool, and thus will go under code review before changing the database. For upstream data quality issues, changes will certainly go through code review, but the challenge is instead getting a separate team to implement the fix. This will hopefully not be an issue if the "Consult Technical Stakeholders" stage goes well, but one does need to account for a timeline outside of their control.

Deploy - Implement DQ Fix:

Once changes are confirmed and pushed into the main branch of the code repository, the solution needs to be deployed into production and then monitored to ensure changes work as expected. Once the change for the underlying code has been deployed, backfilling the impacted data should be considered and implemented if warranted.

Stakeholder Communication:

> One area many technical teams forget to consider is the role of communication among stakeholders, especially business leaders impacted by data quality. Resolving data quality issues in a timely manner is mainly an effort in mitigating lost trust in the data organization, thus simply just resolving the issue silently is not enough. Key stakeholders need to be continually informed of the status of the data quality issue, its timeline to being resolved, and its ultimate resolution. The way in which a data quality issue is handled is just as important as the issue being resolved in maintaining the trust in the data organization.

Though this process is tremendously useful in handling data quality issues, it is still quite manual and reactive. While there are numerous alternatives to automating and scaling data quality (e.g. data observability), we believe that data contracts is the ideal choice for implementing a solution that involves multiple teams along the same data lifecycle within one's respective organization.

An Alternative: The Data Contract Workflow

The current state of resolving data quality issues revolves around reactive processes that require considerable iteration during breaking changes. Furthermore, the largest bottleneck in this data quality process is the process of coordinating various parties to resolve a breaking change– especially among parties where data quality is not their focus.

The data contrat workflow moves the data quality resolution process from reactive to proactive, where constraints, owners, and resolution protocols are established well before a breaking change. In addition, while data contracts ideally prevent a breaking change, in the case where a contract violation is unavoidable, the relevant parties are automatically made aware to resolve accordingly and prevent the stakeholder coordination bottleneck.

The Data Contract Workflow

As highlighted in Figure 4-4, the data contract workflow consists of the following steps:

1. Data constraint identified by data consumer.
2. Data consumer requests a data contract for an asset.
3. Data producer confirms the data contract is viable.
4. Data contract confirmed as code.
5. Data producer creates a pull request to change a data asset.
6. Automatically check if requested change violates a data contract.

7. 7a) Data asset owners are notified of data contract violation and change follows failure protocol.
8. 7b) Data asset updated for downstream processes.

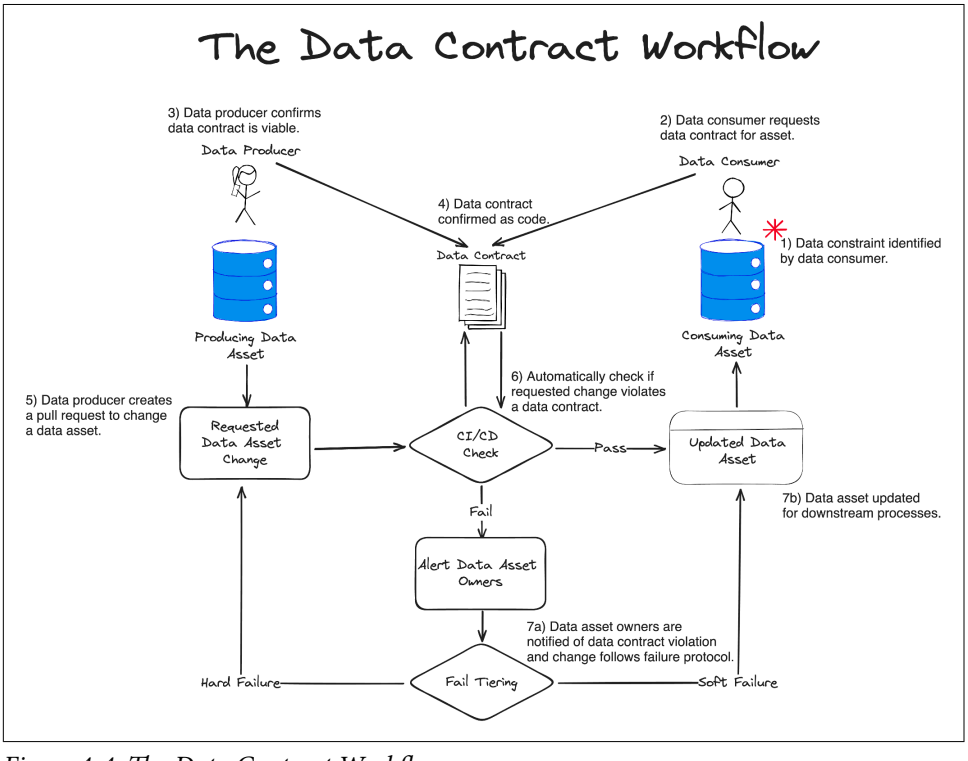


Figure 4-4. The Data Contract Workflow

Let's go look at each step in detail:

1. Data constraints identified by a data consumer.

As mentioned in Chapter 1, data revolves around the needs of data consumers and thus sets forth the data quality requirements. This is because the data consumer is the interface between available data assets and the operationalization of such assets to drive value for the business. Though it's possible for a data producer to be aware of such business nuances, their work is often far removed from the business stakeholders and their needs. A great example of this divide is comparing the business knowledge of a data analyst and software engineer. While both can be business savvy, the data analyst's job revolves around answering questions for the business with data and thus will likely be more abreast of pertinent requirements of the business in real-time.

2. Data consumer requests a data contract for an asset.

One of the most important roles of a data consumer is to translate business requirements into technical requirements in respect to data, as highlighted in Figure 4-5. This is reflected in the fact that business stakeholders are often relegated to interfacing with data via dashboard rather than accessing raw data directly. Therefore, we need to differentiate between "technical data consumers" and "business data consumers" when thinking of the data contract workflow. Thus, technical data consumers will take their knowledge of business and data requirements to request a data contract for data producers to abide to.

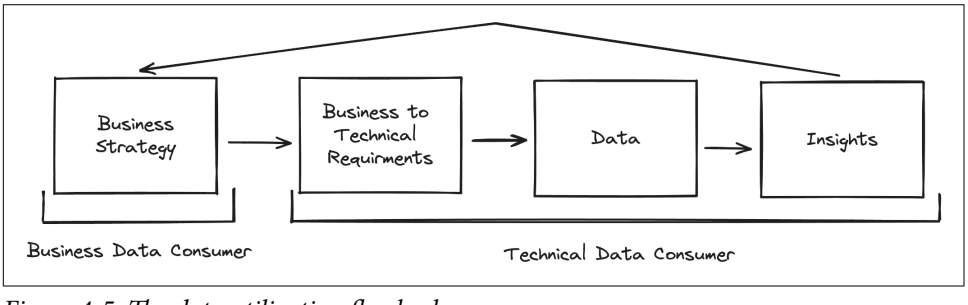


Figure 4-5. The data utilization flywheel.

3. Data producer confirms the data contract is viable.

While technical data consumers are skilled in matching business requirements to technical requirements, their limitation is understanding how technical requirements align with the entire software system. Thus, the data producer will be the one to determine the viability of this request and make necessary adjustments of the proposed data contract.

For example, a data consumer may be aware that the degree of data freshness required by the business is one day for a specific business need. This can be a simple update to a data pipeline schedule, or require a massive refactor to scale capabilities of the data pipeline. The data producer can support the data consumer in becoming aware of these constraints and communicating such limitations to the business.

4. Data contract confirmed as code.

Data contracts have a heavy emphasis on the automatic prevention and alerting of data quality violations, but the step of creating the data contract is actually the most important. Specifically, this step serves as a forcing function for data teams to communicate their needs, inform producers of the business implications of data assets, and establish owners and workflows when a violation is encountered. As noted in the previous step, it's not a one-way request but rather a negotiation among stakeholders to align on how to best serve the business with data.

Furthermore, since the data contract is stored as version-controlled code (typically YAML files), the evolution of historical business to technical requirements mappings are saved. This historical information is gold for technical data consumers who often work with data spanning across timelines of various product and or business changes.

5. Data producer creates a pull request to change a data asset.

This step is self explanatory as version-controlled code and code reviews are a minimum requirement for any software system. With that said, changes to data assets to meet changing software requirements may seem innocuous, but these changes are the fuel for major technical fires that start off as silent failures. This is because data producers are often not privy to downstream business implications and are removed from the fallout of such failures until after a root cause analysis is conducted. Data contracts move these requirements from being downstream and obscured to being readily available for any technical stakeholder to review.

6. Automatically check if requested change violates a data contract.

Once the data contracts are in place for the relevant data assets, data quality prevention can happen in the developer workflow rather than being a reactionary response. Specifically, CI/CD requires new pull requests for code changes to pass a set of tests, and data contract checks fit within this workflow.

7a. Data asset owners are notified of data contract violation and change follows failure protocol.

As stated earlier in this chapter, it's not enough to just be aware of data quality issues. Instead, only the relevant stakeholders need to be notified and given the context to motivate them to take action to resolve the data quality issue. While data quality is a requirement for data consumers, the state of data has limited impact on the constraints of data producers who primarily focus on software. As illustrated in Table 4-2, data contracts realigns the impact of data quality to the motivations of data producers via alerts on their pull request.

Table 4-2. Differences between data producers and consumers

	Data Producer	Data Consumer
Problem	• Need to update the underlying software system to align with changing technical requirements.	• Need to ensure underlying data is trustworthy to drive meaningful insights for the business.
Motivations	• Need to pass CI/CD checks to merge pull requests. • Don't want to have a business critical failure point back to code change where they were notified of the risk.	• Ensuring proper due diligence surrounding the data has been conducted to know limitations of a data asset. • Having insights accepted by the business, especially among executives.
Outcome	• Robust software that not only accounts for technical tradeoffs, but also critical business logic tied to data.	• Reduced time spent investigating and resolving data quality issues for critical business workflows.

It's important to note that a data contract violation does not equate to an automatic full block of the change. Again, the role of data contracts is to serve the needs of the business in respect to data. It may be the case that the contract itself needs to be changed or that there is a technical tradeoff being made where the data quality is a lower priority. Regardless, the contract spec itself will inform if the violation results in a hard or soft failure.

7b. Data assets updated for downstream processes.
After data contract CI/CD checks have passed, the code with the requested change to the data asset will be merged into the main branch and eventually deployed to production. Ideally this workflow will happen with minimal intervention, but in the case of a data contract violation, the resolution process will be documented on the pull request itself given that relevant stakeholders have been notified to engage.

What makes this workflow powerful is that it's not relegated to a single section of the data lifecycle (e.g. tools that only focus on the data warehouse), but rather works anywhere there is movement of data from a source to a target. In the next section, we'll discuss where one can implement data contracts and their various tradeoffs throughout the data lifecycle.

An Overview of Where to Implement Data Contracts

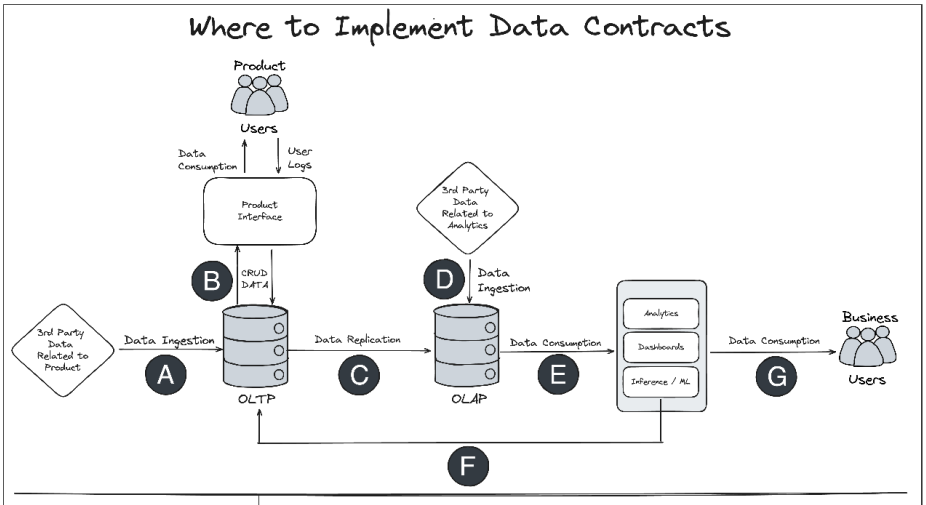
Emphasis of the data contract workflow is abstracting away the business logic and data quality surrounding data assets as an API. By creating this abstraction, stakeholders no longer need to interact with a multitude of touchpoints to understand data and resolve quality issues. Instead, stakeholders now only need to interact with the contract itself and only engage when alerted by a contract violation.

Data contract architecture can be bundled into four distinct components:

- Data Assets
- Contract Definition
- Detection
- Prevention

We will deep dive into each component in Chapter 5 from a conceptual level, Chapter 6 will provide the open source tools we recommend to build data contracts, and Chapter 7 will provide an end-to-end implementation.

In addition to the numerous components, there are also multiple stages in the data lifecycle in which you can implement data contracts, as illustrated in Figure 4-6.



Data Lifecycle Stage	Considerations
A	While we recommend going as upstream as possible for data contracts, one of the most challenging areas to enforce data contracts are with third-party data. While possible, it is unlikely that a third-party would agree to additional restrictions without your organization having leverage. With that said, while you can't control the third-party data, data contracts can be utilized between ingestion and your transactional database as a way to triage data that violates a contract and provide early alerting.
B	Changes to the underlying schema and semantics for CRUD operations are often the root of data quality issues, and would be an ideal location for the first implementation of data contracts. One huge consideration though is whether your team oversees the transactional database; often software engineering teams control this database, which means this would not be an ideal location for a first implementation if you are a data team.
C	The data pipeline between transactional databases and analytical databases is where we recommend most organizations start implementing data contracts. Specifically, both software engineering and data teams are active stakeholders in this stage of the data lifecycle, with data teams often having considerably more autonomy for implementing changes. In addition, this lifecycle stage is often the most upstream source of data for analytical and ML workflows, thus having the highest probability of preventing major data quality issues.
D	Similar to third-party data entering transactional databases, putting contracts between third-party data and an analytical database is often not feasible. One major exception are cloud-based customer relationship management (CRM) platforms (e.g. Salersforce and Hubspot), which are often synced with an analytical database with data connectors. While the data is third-party, CRMs still allow customizability of the underlying data models and columns which can be controlled with data contracts. This is especially true given these data sources.
E	From our conversations with organizations interested in data contracts, many data teams strongly consider first placing data contracts on their analytical database to control data transformations used for analytics, machine learning, and dashboards. While valuable, we strongly encourage teams to first focus on the stage between transactional and analytical databases and then work downstream. Specifically, starting too far downstream will diminish your ability to prevent data quality issues before they are stored in the analytical database. With that said, this is an excellent location after placing the upstream data contracts.
F	While it's possible to place data contracts for "reverse ETL" workflows, it's less common as ideally data contracts on the analytical database are preventing data quality issues from data transformations. Furthermore, unless the engineering team is spearheading the implementation of data contracts, it's unlikely for data contract requests as it often falls in the purview of the data team.
G	Placing data contracts between analytical databases and downstream consumers serve as a way to ensure trust and consistency for the data being served via data products. While too downstream to prevent data quality issues, it enables data teams to tier the data they serve and set expectations. For example, would you want to rely on making key business decisions on a dashboard that was or wasn't using data under contract?

Figure 4-6. The various locations to implement data contracts within the data lifecycle.

Below are the considerations for each stage of the data lifecycle:

A. Third-Party Data → Transactional Database (OLTP):
While we recommend going as upstream as possible for data contracts, one of the most challenging areas to enforce data contracts are with third-party data. While possible, it is unlikely that a third-party would agree to additional restrictions without your organization having leverage. With that said, while you can't control the third-party data, data contracts can be utilized between ingestion and your transactional database as a way to triage data that violates a contract and provide early alerting.

B. Product Surface → Transactional Database:
Changes to the underlying schema and semantics for CRUD operations are often the root of data quality issues, and would be an ideal location for the first implementation of data contracts. One huge consideration though is whether your team oversees the transactional database; often software engineering teams control this database, which means this would not be an ideal location for a first implementation if you are a data team.

C. Transactional Database → Analytical Database (OLAP):
The data pipeline between transactional databases and analytical databases is where we recommend most organizations start implementing data contracts. Specifically, both software engineering and data teams are active stakeholders in this stage of the data lifecycle, with data teams often having considerably more autonomy for implementing changes. In addition, this lifecycle stage is often the most upstream source of data for analytical and ML workflows, thus having the highest probability of preventing major data quality issues.

D. Third-Party Data → Analytical Database:
Similar to third-party data entering transactional databases, putting contracts between third-party data and an analytical database is often not feasible. One major exception are cloud-based customer relationship management (CRM) platforms (e.g. Salersforce and Hubspot), which are often synced with an analytical database with data connectors. While the data is third-party, CRMs still allow customizability of the underlying data models and columns which can be controlled with data contracts. This is especially true given these data sources.

E. Analytical Database → Data Products:
From our conversations with organizations interested in data contracts, many data teams strongly consider first placing data contracts on their analytical database to control data transformations used for analytics, machine learning, and dashboards. While valuable, we strongly encourage teams to first focus on the stage between transactional and analytical databases and then work downstream. Specifically, starting too far downstream will diminish your ability to prevent

data quality issues before they are stored in the analytical database. With that said, this is an excellent location *after* placing the upstream data contracts.

F. Analytical Database → Transactional Database:
: While it's possible to place data contracts for "reverse ETL" workflows, it's less common as ideally data contracts on the analytical database are preventing data quality issues from data transformations. Furthermore, unless the engineering team is spearheading the implementation of data contracts, it's unlikely for data contract requests as it often falls in the purview of the data team.

G. Data Products → Business Users:
: Placing data contracts between analytical databases and downstream consumers serve as a way to ensure trust and consistency for the data being served via data products. While too downstream to prevent data quality issues, it enables data teams to tier the data they serve and set expectations. For example, would you want to rely on making key business decisions on a dashboard that was or wasn't using data under contract?

The key deciding factor of where to implement data contracts is the relationship between upstream and downstream producers. While ideally this will always be on good terms, realistically it's more nuanced and challenging. In Chapters 10 through 12, we will detail how to navigate these nuances in team dynamics within the business and build buy-in.

The Maturity Curve of Data Contracts: Awareness, Ownership, and Governance

While Data Contracts are a technical mechanism for identifying and resolving Data Quality and Data Governance issues upstream, cultural change is equally as important to address as technology. Culture change here refers to the shifts in behavior required by data producers, data consumers, and leadership teams in order to help their company onboard to data contracts and successfully roll out federated governance at scale.

It's important to acknowledge that not all companies are equally ready for data contracts from day 1. Some businesses are already data-driven: These companies understand and appreciate the value of data for its capacity to add operational and analytical functions from the top of the organization to the bottom. Others are simply data aware: They understand data is used at their company, but outside the data organization there isn't much recognition of the need to invest in infrastructure, tooling, and process. Others are data ignorant: Their journey in data has barely started!

To make matters more complex, these differences in maturity and communication may not only differ between organizations, but WITHIN organizations across teams.

For example, a machine learning team may have a sophisticated appreciation for the value of data with the Sales team may not. In our experience, it is important to acknowledge these differences exist. Companies cannot be treated as monolith! To succeed with the implementation of data contracts, data heroes must take a pragmatic approach that relies on rolling out contracts across a *maturity curve* depending on organizational and team-based readiness.

The data contract maturity curve has three steps: Awareness, Collaboration, and Ownership. We'll outline each step, as well as their corresponding goals.

1) Awareness

Goal: Create visibility into how downstream teams use upstream data

The goal of the Awareness phase is to help both data producers and data consumers become aware of their responsibilities as active stakeholders in the data supply chain. As mentioned above, data contract requirements must start from the consumers who are the only ones with an explicit understanding of their own use cases and data expectations. In order to limit the surface over which data teams must begin implementing contracts, it is advisable to select a subset of useful downstream data assets known as tier-one data products.

Data product is a term that has many different definitions. We prefer looking to our software engineering counterparts for directional guidance. A software product is the sum of many engineering systems organized to serve an explicit business need. Products have interfaces (APIs, User interfaces) and backends. In the same way, a data product is the sum of data components that are organized to serve a business need, as illustrated in Figure 4-7.

A dashboard is a data product. It is the sum of many components such as visualizations, queries, and a data pipeline. The interface: a drag-and-drop editor, charts, and data tables. The back-end: data pipeline and data sources. This framing rings true for a model's training set, embedded data products, or other data applications. *The data contract should be built in service of these products*, not the other way around.

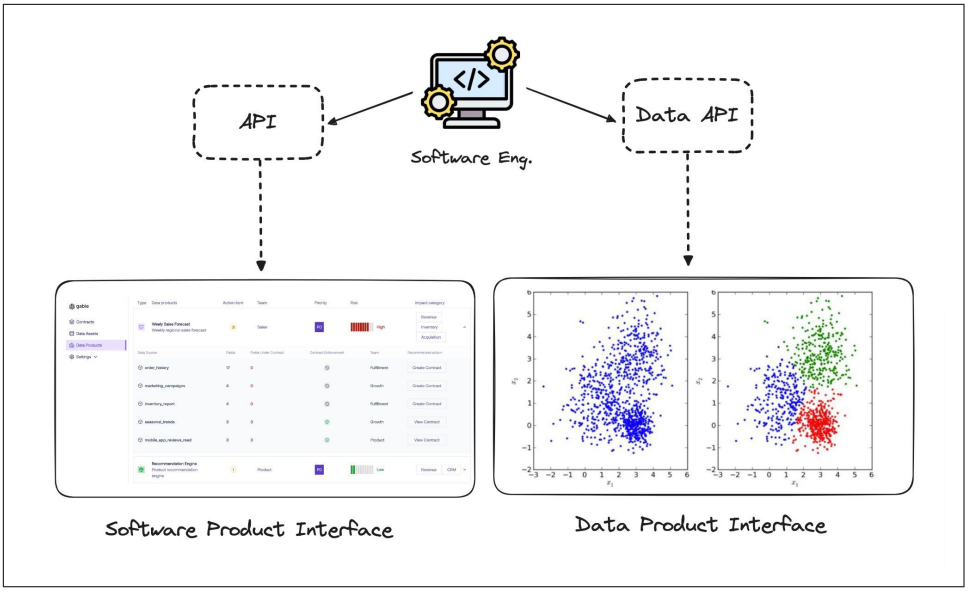


Figure 4-7. Software vs. data product interfaces.

In the Awareness phase, data producers must understand that changes they make to data will harm consumers. Most data producers are operating in a black box regarding the data they emit. They don't *want* to cause outages, but without any context provided pre-deployment, it is incredibly challenging to do so.

Even without the implementation of a producer-defined data contract, producers should still be aware of when they are making code changes that will affect data, exactly what those changes will impact, and who they should speak to before shipping. This pre-deployment awareness drives accountability and most importantly - conversation.

2) Collaboration

Goal: Ensure data is protected at the source through contracts

Once a data producer has some understanding of how their changes will impact others in the organization, they are faced with a set of choices. A.) Make the breaking change and knowingly cause an outage or B.) Communicate with the data consumers that the change is coming. The second option is better for a wide variety of reasons that should be obvious!

This resolves most problems for data consumers. They are informed in advance before breaking changes are made, have plenty of time to prepare, and can potentially delay or deter software engineers' deployment by advocating for their own use cases of the data. This sort of change management functions in a similar way to pull

requests. Just as an engineer asks for feedback about their code change, with a consumer-driven contract, they may also "ask" for feedback about their change to data.

It can't be stressed enough the importance of this collaboration happening pre-deployment driven by context. Once code has been merged it is no longer the responsibility of the engineer. You can't be accountable for a change you were never informed about!

3) Contract Ownership

This shift-left towards data accountability resolves problems but also creates new challenges. Consider you are a software engineer who regularly ships code changes that affect your database. Every time you do so, you see that there are dozens of downstream consumers, each with critical dependencies on your data. It's not impossible to simply communicate with them which changes are coming, but to do that for *each* consumer is incredibly time-consuming! Not only that, but it turns out certain consumers have taken dependencies on data that they shouldn't be, or are misusing data that you provide.

At this point, it is beneficial for producers to define a data contract, for the following reasons:

1. Producers now understand the use cases and consumers/customers
2. Producers can explicitly define which fields to make accessible to the broader organization
3. Producers have clear processes in place for change management, contract versioning, and contract evolution

Data producers clearly understand how changes to their data impact others, have a clear sense of accountability for their data, and can apply data contracts where they matter most to the business. In short, consumer-defined contracts create problem visibility, and **visibility creates culture change**. The next section details the outcomes of enabling this change.

Outcomes of Implementing Data Contracts

There are several extremely important outcomes which occur as a result of data contracts being implemented across your organization. Some of them are obvious and can be measured quantitatively while others are softer, but nevertheless have some of the greatest impact on culture change and working conditions as a data developer. The three core metrics are the following:

Faster Data Science and Analytics Iteration Speed:

When Mark was a data scientist, a majority of his time for any project was spent on 1) sourcing data that was of high enough quality to use within a data lake, and 2) spending considerable time understanding the quirks of the data in question and validating it. Specifically in his previous role at an HR-tech company, one of the most important data assets was "manager status of an employee." Despite this being a critical data asset for an HR company, it was constantly changing as new customers created various edge cases or the product evolved. For example, an enterprise customer would change employee management systems and thus ingested employee data would change from a daily batch job to a monthly– unfortunately management status doesn't align with a monthly cadence. Thus, the same exploration and validation stage was present on every new project on the same data asset.

With implementing data contracts, the outcome would be considerably cutting this exploration and validation stage. First, having data assets under contract creates a shortlist of data to use for data science projects and ensures that proper due diligence has already taken place. Second, the data contract itself documents the quirks of the data one needs to be aware of and its expectations. Third, since data contracts are version controlled, data teams also have a log of past changes in constraints and assumptions, as well as a mechanism to document and enforce new ones that arise and or evolve. In short, data contracts provide a mechanism to handle the discovery and validation of data assets at scale while also disseminating this information across teams in a manner that's version controlled.

Developer and Data Team Communication:

In Chapter 3, we referenced Dunbar's Number and Conway's Law as two powerful phenomena within businesses that shape how technical teams communicate (or lack of) with each other. Specifically how an increase in employee count corresponds with an exponential rise of potential connections that ultimately breaks down communications– which are reflected in the technical systems built by the organization. Data contracts aim to overcome this challenge via automation and embedding itself within existing CI/CD workflows.

First, data contracts increase visibility of dependencies related to data assets generated and or transformed by upstream producers (e.g. application engineers) in the three following ways:

1. For a data contract to be enforceable, both the consumer and producer parties need to agree to the data contract spec before implementing it within the CI/CD pipeline– serving as a forcing function for communication between both parties.

2. Since an enforceable contract is within the CI/CD process, violations generate failed test notifications within pull requests and notify data asset owners, thus supporting code reviews with relevant information and the people who can help resolve issues.
3. There may be cases where a contract violation is no longer relevant due to evolving needs, implying that a new constraint is needed, thus changes to the contract specs inform downstream data asset owners rather than finding out reactively with the data itself.

Mitigation of Data Quality Issues:

Ultimately, the reason for going through the effort of implementing data contracts and coordination amongst technical teams is to reduce business critical data quality issues. But to reiterate, data quality is not about pristine data, it's about fitness of use by the consumer and their relevant tradeoffs. Furthermore, poor data quality is a people and process problem masquerading as a technical problem. Data contracts serve as a mechanism to improve the way in which people (i.e. data producers and consumers) communicate about the captured process of the business (i.e. data). Specifically, resolving data quality issues shift from being a reactive problem to instead a change management problem in which teams can iterate in as the consumer use cases become clearer over time.

While this section highlights high-level outcomes of data contracts, we encourage reviewing Chapter 11 where we go into the specific metrics used to measure the success of a data contract implementation.

Data Contracts vs. Data Observability

Through our hundreds of conversations with companies about data contracts, one question was often asked: "How are data contracts different from data observability, and when would I need data contracts or observability?" The below section aims to answer this question.

According to Gartner, data observability is defined as the following:

"... [The] ability of an organization to have a broad visibility of its data landscape and multi-layer data dependencies (like data pipelines, data infrastructure, data applications) at all times with an objective to identify, control, prevent, escalate and remediate data outages rapidly within acceptable data SLAs."

The only caveat we will make to this definition is that data observability can't be preventative in itself, as an event needs to happen for it to be observable, but it can definitely inform preventive workflows such as data contracts themselves. Table 4-3 below provides further information on the comparison of the two:

Table 4-3. Data contracts vs data observability

Data Contracts	Data Observability
Prevent specific data quality issues: Data contracts emphasize preventing changes to metadata that would result in breaking changes in related data assets.	**Highlight data quality trends:** The main value proposition of data observability is that it gives you visibility of your data system and how that system is changing in real-time.
Included in CI/CD workflow: Data contracts checks are embedded within the developer workflow, specifically CI/CD, so that breaking changes can be addressed before a pull request is merged.	**Compliments CI/CD workflow:** Observability serves as a measurement of your data processes, and thus is not within the developer workflow; but the output of observability will inform which additional tests you need to have within your CI/CD workflow.
Informed by business logic: Data contracts is technology to scale communication among data producers and consumers. One of the most important data consumers are business stakeholders who provide domain knowledge that enables value generation from data.	**Reflects how data captures business logic:** While business logic is an input into data contracts, observability insteads measures the output of business logic within data systems and how well they align with reality (e.g. capturing data drift).
Targeted visibility: Data contracts should not be on every data asset and pipeline, but rather only on the most important ones (e.g. revenue generating or utilized by executives). This prevents alert fatigue, as the goal of data contracts is not only to alert, but also encourage the data producer to take action when a contract is violated.	**Broad visibility:** Data observability should reach into every aspect of your data stack from ingestion and processing to serving. While at later stages of implementation, one can refine thresholds, the broad visibility of
Alerts before change: Data contracts shift agreements between data producers and consumers from implicit to explicit. This enables the prevention of metadata changes that would result in breaking related data assets for known issues.	**Alerts after change:** By its name, to observe means that an event has already taken place, and thus can't be fully preventative. With that said, such workflows *after* an event is essential for surfacing unknown issues to inform future data contracts.

The key difference between data contracts and data observability is that contracts emphasize data quality prevention of known issues, while observability emphasizes detection of unknown data quality issues. One is not a replacement of the other, but rather, both data contracts and data observability compliment each other. Another way to think about the two is in terms of the flashlight and laser pointer, where both illuminate an area to bring attention to it yet serve different purposes. As illustrated in Figure 4-8, data observability is similar to the flashlight where it illuminates the entire data system and workflows, where the alternative is being "left in the dark" about your data and waiting to bump into data quality issues. While data observability serves as a flashlight in this analogy, data contracts can be viewed as the laser pointer. While its light is relegated to a small area, its value is in its ability to target and bring attention to a specific area within a system.

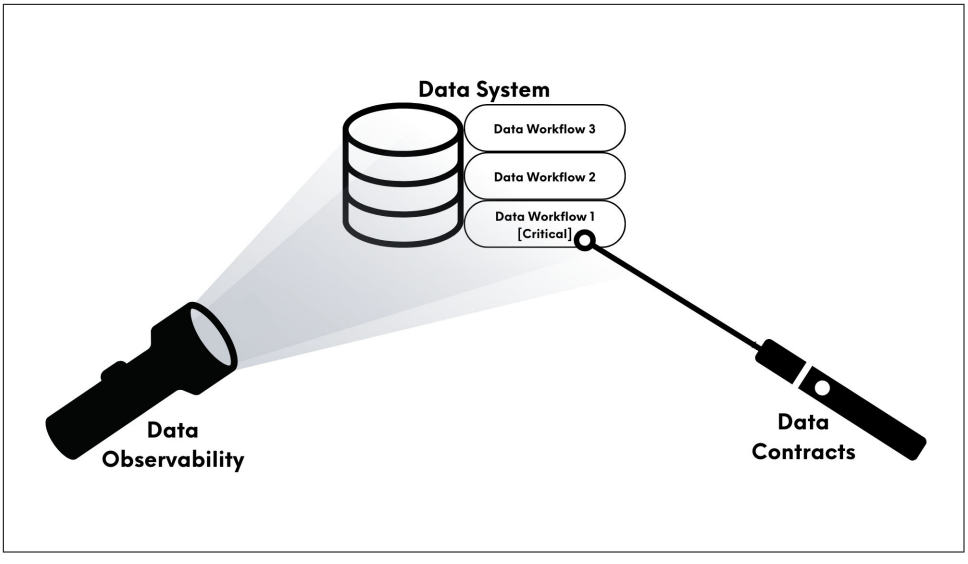


Figure 4-8. How data contracts and observability compliment each other.

Observability and contracts are individually useful for data teams, but using them together enables teams to work more efficiently in that they are able to automate understanding their entire data system with observability and use this information to inform enforceable constraints with contracts.

Conclusion

This chapter provided the theoretical foundation of the data contract architecture, as well as discussed the key stakeholders and workflows when utilizing data contracts. Specifically, we covered:

- How collaboration is different in data.
- The roles of data producers and consumers within the data lifecycle.
- The current state of reactively resolving data quality issues.
- A high level overview of the data contract workflow.
- The various tradeoffs between implementing data contracts throughout the data lifecycle.
- The maturity curve of implementing data contracts and its outcomes.

In the next chapter, we will move from theoretical and into providing real-world case studies of implementing the data contract architecture.

Additional Resources

CHAPTER 5

The Data Contract Components: Data Assets and Contract Definition

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In Chapters 1 through 5 we discussed data contracts from a theoretical perspective, and in Chapter 6 we will shift into actually applying data contracts with a real-world project. Specifically, in Chapter 6 and 7 we will provide you with the components of data contracts and our preferred open source tools needed to implement them. In Chapter 8 we will build on Chapters 6 and 7 with a guided project tying these open source components together to create the architecture for data contracts. We highly encourage you to reference our public GitHub repository that compliments these chapters and walks you through the technical implementation.

Let's start with our overview of the components of data contracts.

Overview of Components

What makes data contracts powerful, is what also makes them difficult to implement. The power of data contracts is that they're designed to unite teams and disciplines across an entire company, while also integrating seamlessly into the individual tools and workflows at all stages of the data lifecycle. Thus, data contract architecture requires using a toolbox of open source infrastructure, in which these components can be broken up into four main categories and their respective subcategories illustrated in Figure 5-1.

In Chapter 6 we will only cover Data Assets and Contract Definition in depth, but will provide a high level of all components so it's clear how they all fit within the larger picture. The next chapter will cover the remaining components of Detection and Prevention.

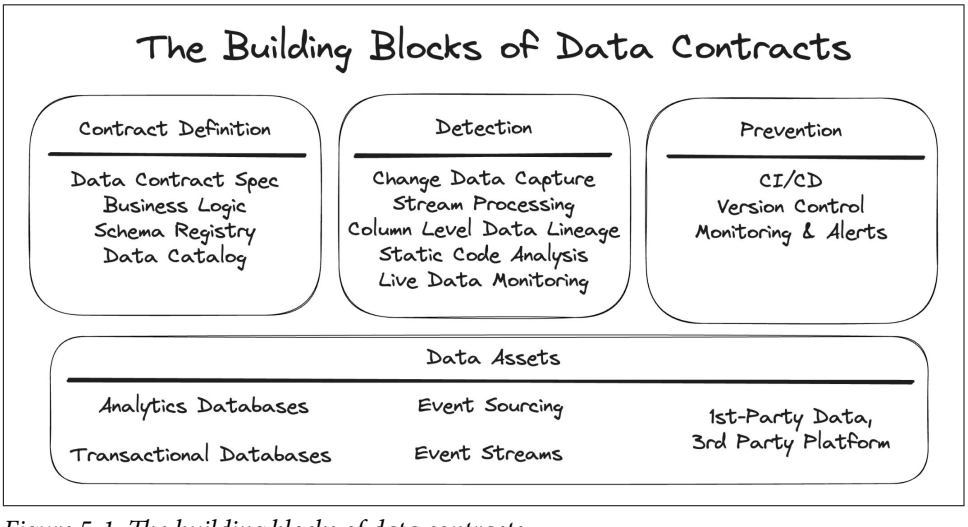


Figure 5-1. The building blocks of data contracts.

Let's look at each building block in a little more detail:

Data Assets
: The foundation of all data infrastructure are databases, and while the types of databases are vast, we will focus on the following four database categories for this book.

  - Analytics Databases
  - Transactional Databases
  - Event Sourcing and Event Streams

- First Party Data, Third Party Platform

Contract Definition
: Data contracts are codified via the "contract spec" which serves as a surface to embed the expectations around schema and business logic (i.e. semantics) for a respective data asset. The data contract spec is then compared to centralized sources of metadata, such as schema registries and data catalogs. The three components are as follows.

  - Data Contract Spec
  - Business Logic
  - Schema Registry and Data Catalogs

Detection
: Detection needs to happen at the code level for schema, and at runtime for data semantics to ensure changes are captured ideally before data is written to the target database. The following five categories represent the areas in which one can detect changes to data.

  - Change Data Capture
  - Stream Processing
  - Column Level Data Lineage
  - Static Code Analysis
  - Live Data Monitoring

Prevention
: While it's ideal for prevention to be automatic in handling contract violations, this is not possible given that the nature of many data quality issues are rooted in people and processes. Thus the automation is in alerting the right people at the right time within the developer workflow. The below three categories are typically where one can inform of potential breaking changes.

  - CI/CD
  - Version Control (Data Code Review)
  - Monitoring & Alerts

While these are quite a few components, many of them are components already used within software and data infrastructure– thus, we're mostly repurposing already existing infrastructure and or adding additional workflows.

We'll start our in-depth review of each building block by digging deeper into data assets.

Data Assets

In Chapter 4, Figure 4-6 highlighted the considerations of implementing data contracts in the various touchpoints of the data lifecycle. We are bringing an iteration of that diagram with Figure 5-2, as we dive into the four categories of data assets: transactional databases (A), analytical databases (B), events data (C), and first-party data within third-party platforms (D).

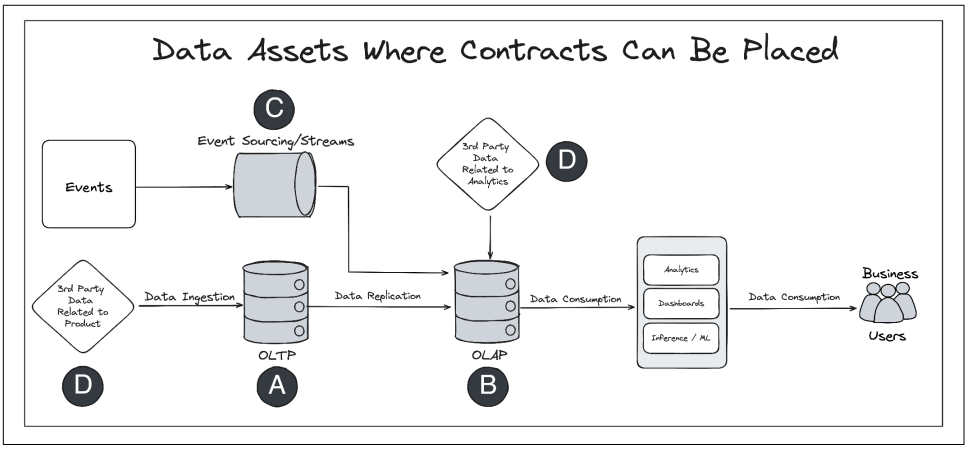


Figure 5-2. Data assets where contracts can be placed

Analytics Databases

Data developers are most likely found working within analytical databases—given their central role in analytics (as the name implies), machine learning, and AI—and storing the underlying data of dashboards presented to non-technical business stakeholders. Analytical databases can include data warehouses, data lakes, data lake houses, and other variations where the emphasis is on being able to scan and aggregate large swaths of data. Furthermore, performance is centered around data quality and reliability rather than speed; for example, processing an extensive SQL query in minutes as compared to retrieving a specific data point in subseconds. Given this distinction, data contracts become quite useful in maintaining the reliability and quality of the data within an analytical database.

While earlier implementations of analytical databases centered primarily around data warehouses– which were subject-oriented, integrated, time-variant, and non-volatile–the rise of the now-common Modern Data Stack (MDS) changed these assumptions. Specifically, MDS relies heavily on extract-load-transform (ELT) where minimal to no transformations are implemented to make upstream data useful for analytics. This has resulted in analytical databases becoming company-wide dumping grounds of data, via replication, where the hope is that one day the data may become valuable.

In reality, it makes the small percentage of data that is truly valuable to the business much more complex to utilize and harder to find– like a needle in a haystack.

Many data teams will often put a patch on the issue by building numerous layers of transformations via Data Build Tool (dbt) to quickly iterate and provide value to the business… but this doesn't solve the root problem. Specifically, modern analytical databases do not adhere to being "non-volatile", thus resulting in these layers of transformation being brittle. For example, an upstream table that's replicated into the analytical database has a schema change, resulting in all subsequent downstream transformation breaking.

This challenge is exacerbated even further by scaling both teams and infrastructure, where the number of assumptions and dependencies grows exponentially (as highlighted in Chapter 3). Mark experienced this first-hand in overseeing the analytical database at an HR-tech startup. When he was initially hired to the data science team, it was relatively easy to manage the analytical database as its use was relegated to just the few on the data team, and the upstream engineering team was all under one manager. Two years later the analytical database was used by a data team that had doubled in size, powered the majority of dashboards used by the business, and engineering was split between three different teams. What was once a simple set of tables had become a complex system that relied on three different upstream teams, supported numerous R&D initiatives and analytical workflows among the data science team, and was a source of contention among business users receiving the wrong data within their dashboards. While our manual workflows on the analytical database sufficed in the earlier stage, our ability to maintain these manual workflows suffered as we scaled.

While these scaling issues were challenging, they are part of the growing pains of an evolving organization. As the number of dependencies increase for a respective data asset, so does the awareness of various edge cases and data quality issues among business stakeholders. It's not a matter of *if* a data issue will arise, but rather a matter of *when*, and increased scope leads to an increased probability of issues being highlighted. Though painful to deal with, these issues serve as a litmus test to determine if the underlying data aligns with the business stakeholders expectations; especially when they are domain experts (e.g. a doctor reviewing clinical data). Specifically, such instances inform whether a data team's downstream transformations are wrong, if upstream data is being captured wrong, or the underlying assumptions of the data pipelines don't align with reality in which the data represents. All of which makes data contracts useful in both documenting this business logic and enforcing such constraints automatically as issues are surfaced.

Transactional Databases

While data developers are found working within analytical databases, software developers are often found upstream in transactional databases, as this is most pertinent to the application and services layer of most businesses. As mentioned in Chapter 1, these transactional databases focus on CRUD operations, ACID compliance, and with data in third-normalized form– all of which enable blazing fast information retrieval (e.g. website actions taking subseconds). More importantly, transactional databases are where "snapshots of the truth" are captured by the business, such as product logs and user information (i.e. Section "B" of Figure 5-2). This is why transactional databases serve as an ideal location for data contracts—they're the most upstream data source an organization controls given that analytical databases are often replications. The challenge is that data quality is not a top priority for many software engineering teams as their constraints revolve around the maintainability of the code rather than the underlying data, thus a data contract initiative would be deprioritized unless driven by the engineering team itself.

Diving further into the concept of ACID is essential for understanding the importance of data contracts in maintaining data quality. ACID stands for atomicity, consistency, isolation, durability but the emphasis for data contracts will be on "consistency" for the following reason as highlighted in Martin Kleppmann's book Designing Data-Intensive Applications:

> …[This] idea of consistency depends on the application's notion of invariants [(i.e. true conditions about your data)], and it's the application's responsibility to define its transactions correctly so that they preserve consistency. This is not something that the database can guarantee: if you write bad data that violates your invariants, the database can't stop you. (Some specific kinds of invariants can be checked by the database, for example using foreign key constraints or uniqueness constraints. However, in general, the application defines what data is valid or invalid—the database only stores it.)
>
> Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database's atomicity and isolation properties in order to achieve consistency, but it's not up to the database alone.

A great way to think of invariants in a database is to consider the question "Who is a customer?"

- Is it an enterprise company with an active contract with your organization?
- Is it a company that ever had a contract with your organization?
- Is it a company without a contract but has made up-to-date monthly payments?

As you can see, the devil is in the details as to *what* exactly needs to stay consistent within your database, and data contracts are best equipped to uphold these constraints as the business (and ultimately the invariants) evolve.

An excellent real-world example of this comes from the hundreds of calls we have had with companies interested in data contracts. Specifically, an unnamed consumer app company had the assumption that their business would be business-to-consumer (B2C) and thus this assumption was embedded everywhere from their software architecture, databases, and documentation. Yet ~10 years later the company realized they needed to expand their offering to businesses, and thus became business-to-business (B2B) in addition to B2C. In other words, their invariant of "each individual customer and user is a one-to-one relationship" ceased to be true as users could quickly switch between individual and company accounts (e.g. a freelancer joining a company for a 3 month contract). Their ~10 years of revenue driving assumptions and architecture quickly became data and tech debt with a single major business decision– resulting in their inability to properly bill their customers and therefore a loss in revenue. Thus, they explored data contracts as an avenue to put guardrails on their data system as they did a major refactor (refer to Chapter 3 on refactors). Specifically, by establishing contracts for their desired end state of data (e.g. B2C only to B2C and B2B customers) they could identify where in the transactional database these contracts were violated, and iterate until violations ceased to exist.

In addition to CRUD operations, transactional databases also serve as the ingestion point for third-party data relevant to the application layer (i.e. Section "A" of Figure 5-2). For example, Mark previously worked in a HR-tech startup that ingested sensitive employee demographic data given directly by their respective enterprise customers. A key consideration through this workflow was: How can we handle differential data loading patterns from customers? While it's unlikely you can enforce a data contract on a third-party, due to lack of leverage (e.g. an important enterprise customer unwilling to change), data contracts are powerful between the raw data staging and ingestion into the transactional database.

For Mark's HR use case, customers provided data as manual CSV files, SSH File Transfer Protocol (SFTP), or a direct API– all of which were provided in varying intervals such as daily, monthly, or quarterly. This ended up causing painful data quality issues, as this employee data determined org charts and manager status which were key factors for product features. We had to deal with issues such as: Why is a file missing? Did the customer shift from daily to monthly intervals? Did they change the format of the CSV file? These unexpected changes created data quality issues visible in the application and resulted in the engineering team spending copious amounts of time reactively debugging data quality issues in the transactional database, manually fixing the third-party data, and doing backfills. In this situation, data contracts would enable:

- Documentation of differential workflows of the underlying third-party data assets.

- Preventing data that violates a contract from entering the transactional database and creating customer facing data quality issues in the application.
- Logging contract violation patterns of provided third-party data, which can be used by the business team to highlight to customers the impact of providing poor quality data.

Not surprisingly, we have talked to numerous companies experiencing challenges with third-party data providers and see data contracts as an avenue to relieve their pain.

Finally, a less common pattern is pushing the outputs of data products in the analytical database back into the transactional database (i.e. Section "F" of Figure 4-6). With the emerging category of generative AI, we expect this pattern to grow considerably as more companies expand to AI use cases. A great example of this is the "batch prediction" design pattern for machine learning models as highlighted in Figure 5-3.

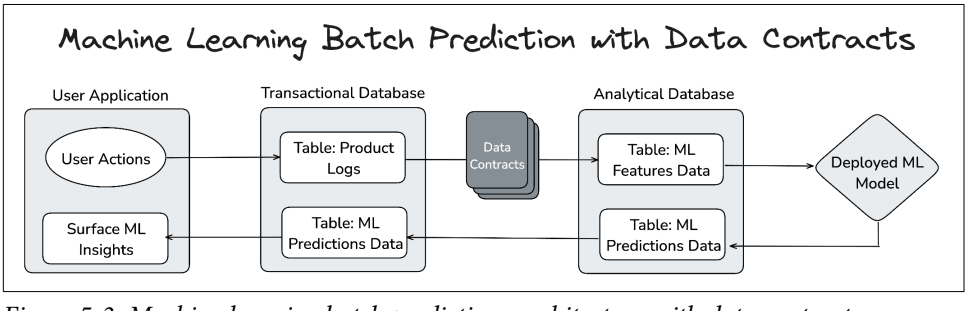


Figure 5-3. Machine learning batch predictions architecture with data contracts.

There is already a tremendous amount of literature on machine learning (out of scope for this book), but where data contracts fit within this use case is between the replication of a data asset between the analytical and transactional databases. In other words, a "data product" was developed using the analytical database assets, and now the transactional database has a major dependency on this data product for customer facing surfaces. Some considerations include the following:

- Is the format of this data product output suitable for the transactional database?
- Would new data for the same users be overwritten or appended to?
- What happens if machine learning monitoring flags outputs with poor predictive power?
- Is the prediction data asset going to be versioned with the accompanying ML model versions?
- Are there any regulatory requirements for the predictions data (e.g., health or financial data)?

By placing constraints between the analytical database and transactional database, organizations can ensure data maintains expectations before it reaches customer facing surfaces.

Event Sourcing and Event Streams

Event driven architecture (EDA) is a dense topic that deserves an entire book in itself, thus for sake of brevity we will mainly focus on two facets: 1) event sourcing, and 2) event streams and how they relate to data contracts. For a more in-depth discussion on streaming, we highly recommend reading *Chapter 11. Stream Processing* within Martin Kleppmann's *Designing Data-Intensive Applications*.

It's best to understand streaming data in relation to batch data, where batch data provides dumps of data from system A to system B in predetermined intervals (e.g. one day). This approach is the most simple and what many data teams default to. Yet, there are cases where a set interval of time doesn't suit the use case, such as tracking changes when they happen in real-time or when there is a high volume of data in a continual short period of time (e.g. telemetry data). In these use cases, the additional complexity of EDA is warranted. Examples of this additional complexity, and thus data quality issues, include:

Race conditions and concurrency
: Multiple conflicting writes causing overwriting our data targets becoming out of sync.

Fault tolerance and durability
: Ability for a system to account for a node going offline and thus losing data (e.g. distributed computing).

Retention
: How long should data within a stream be retained to ensure accurate data but doesn't cause memory issues.

Latency
: What's an acceptable window of time the data can represent (e.g. seconds vs millisecond)

Note that while these challenges can be present for batch processes, it's more pronounced within streaming.

These considerations thus bring us to the differences between event sourcing and event streams highlighted in Table 6-1. Both of which were used by Chad at a previous company, further described in his articles *An Engineer's Guide to Data Contracts*, *Part 1* and *Part 2*.

Table 5-1. : Comparison of Event Sourcing and Streaming

	Event Sourcing	Event Streaming
Use Case	Immutable log that has every recorded event appended to the log.	High volume of data captured in set intervals of times (i.e. "windows").
Data Examples	• Updates to self-reported user demographic information in a application. • Online shopping cart transactions • Shipping status across a supply chain.	• Telemetry data for devices • High-volume financial transactions (e.g. stock market) • Click tracking on a high-traffic website.

While there are numerous ways to implement event driven architecture, we highly recommend the publish-subscribe pattern for data contract implementation– Apache Kafka is an example open source tool that enables this. Under this pattern, you have a service that sends events, a message broker that manages these events and essentially provides a "buffer," and a set of event subscribers that read from the broker. Where event sourcing and streams differ is in how the event broker is utilized. As illustrated in Figure 5-4, event sourcing creates an immutable log of every event from the event publishing service; an example being updates to shipping status across a supply chain.

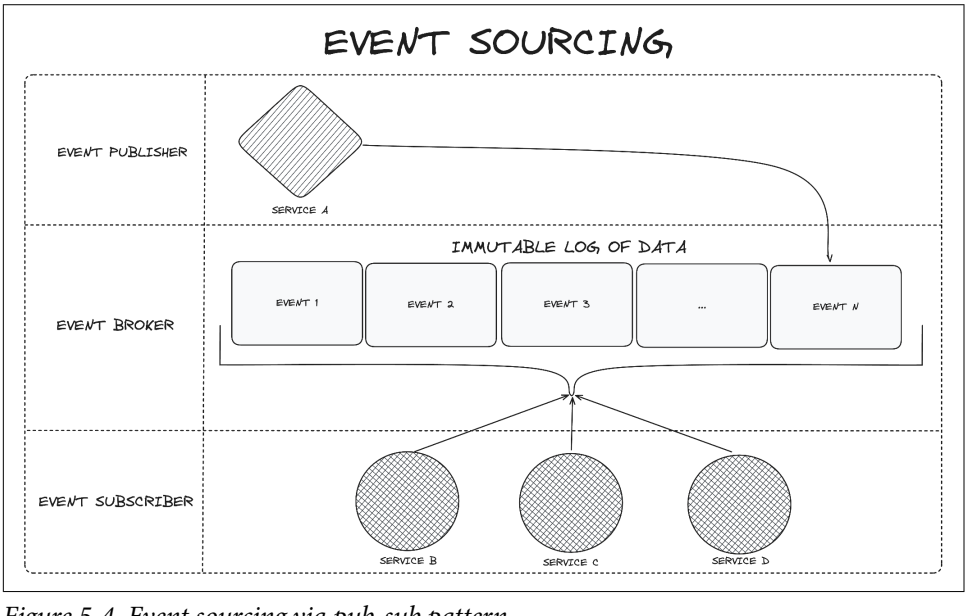


Figure 5-4. Event sourcing via pub-sub pattern.

What makes this powerful from a data quality perspective is that the immutable log can recreate databases and event history in the instance that a downstream database goes down. Furthermore, different downstream consumers can utilize the log for their respective use case, such as analytics reviewing all logs in history or an user-facing app only presenting the most recent log in the UI without impacting each

other. Pulling from the supply chain example again, the analytics team may want to know the average shipment wait time between each step in the supply chain, while an end user only wants to know where the shipment is currently located.

While event sourcing and streaming have many similarities, there are instances where creating an immutable log of every event is infeasible and would either result in an out-of-memory error or a very expensive cloud service bill. In instances where there is a high level of volume of data (e.g. telemetry data), the utilization of "windowing" is leveraged to capture a log of data in set time intervals (e.g. every second) as illustrated in Figure 5-5. While not all the data is captured, its general trends and patterns can be aggregated into a single value in a log, which then follows the similar pattern of a "buffer" via the event broker and data consumer via event subscribers.

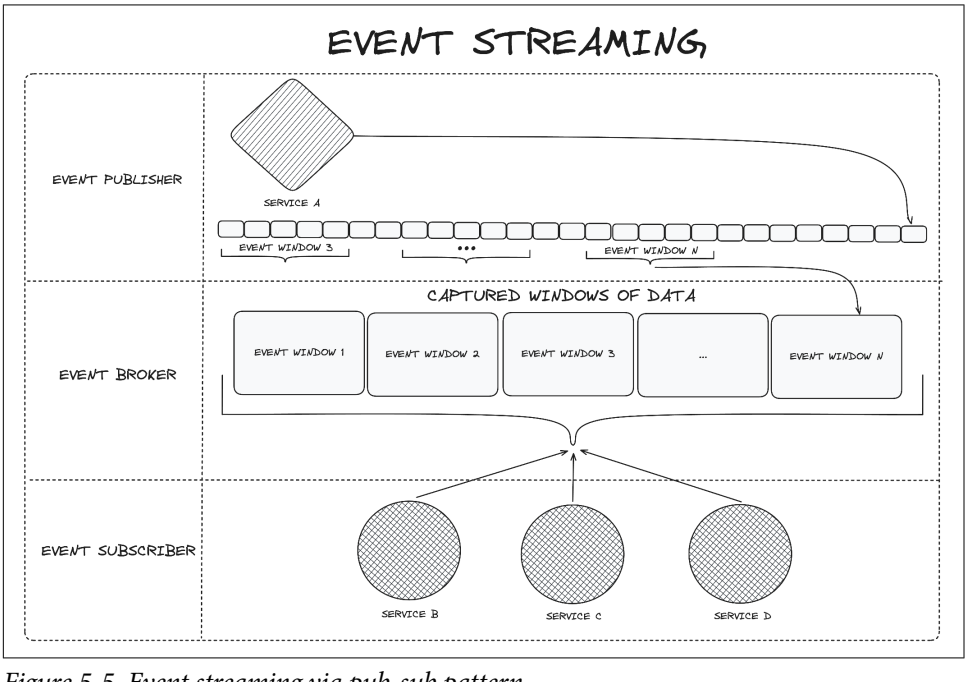


Figure 5-5. Event streaming via pub-sub pattern.

Oftentimes, organizations will employ both event sourcing and streaming in the same data stack. Furthermore, event driven architecture becomes more common as an organization scales and increases in data maturity. In Mark's previous role at a startup, he accomplished replication from a transactional database into an analytical database via daily batch loads. In the earlier days, the batch operations were lightweight and reliable, but this changed once the company scaled the number of enterprise customers. The once simple batch operations took hours to run, errors in the pipeline caused jobs to fail, resulting in stale data, and the batch would do full replacements of the table. The last point of replacements was most troubling,

as breaking changes to schema and semantics were propagated into the analytical database and would require an hours-long backfills. This resulted in days or weeks of the data science team being stalled on their work, and it was clear that it was time to start paying off the technical debt that was acquired via a tradeoff for an initial simpler solution (rightfully so at the time). The team heavily considered event sourcing via change data capture, as it would enable transactional integrity and shift them away from hours-long batch jobs towards incremental updates into the analytical database.

First-Party Data, Third-Party Platform

Typically one cannot enforce data contracts on third parties given the lack of leverage; but there is one edge case that doesn't apply: instances where it's first-party data of the business, you can control the underlying data model, but the platform in which the data is stored and processed is a third party. Examples of this include customer relationship management software such as Hubspot and Salesforce, or enterprise resource planning (ERP) software such as SAP. These data sources are often business critical and many predate the implementation of analytical databases, thus are embedded with a tremendous amount of business logic without considerations of an underlying data model. While these systems don't have the same level of customizability as a traditional database, the underlying data objects can be modified and are replicated into an organization's analytical database (refer to "D" in Figure 5-2), typically via batch jobs.

For example, in a former position, Mark replicated Salesforce data into BigQuery using the ELT tool Fivetran. He then generated reports that linked product usage, time tracking data, and customer deal size and health (from Salesforce) to assist in reallocating staffing resources within the Customer Success team– ultimately aiming to optimize the number of employee hours dedicated per dollar of annual recurring revenue (ARR).

Given this nature of third-party platforms, CI/CD workflows are not typically possible and thus prevention shifts from "preventing violations within source data" to "preventing source data violations replicating into a target database." Figure 5-6 illustrates this, specifically steps "6," "7a," and "7b" where there is a "prevention window" between staging and execution of data replication into the target database.

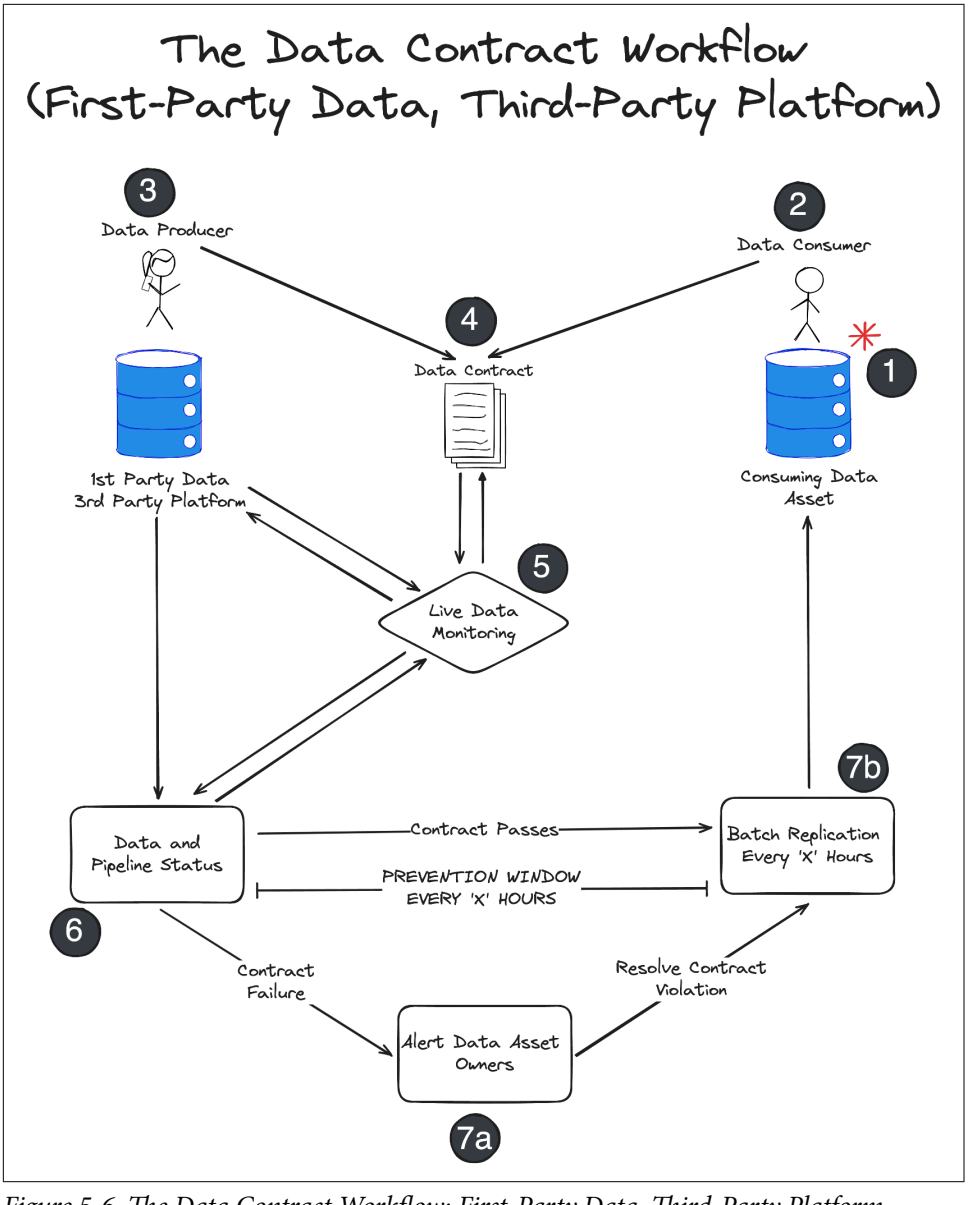


Figure 5-6. The Data Contract Workflow: First-Party Data, Third-Party Platform

This process in detail looks like the following:

1. Data constraints identified by the data consumer.
2. The data consumer requests a data contract for an asset.
3. Data producer confirms data contract is viable.

4. Data contract confirmed as code.
5. Continuous monitoring of data and relevant contracts.
6. Queue of batch replication jobs.
7. Data asset owners are notified of data contract violation for a batch job and follow failure protocol.
8. Data assets updated for downstream processes when the data contract passes tests for the batch job.

Please refer to Figure 4-4 to further visualize the difference between first-party data and platform v.s. first-party data and third-party platform use cases.

In Mark's example of optimizing employee hours and ARR, there was a one week (i.e. 168 hours) window in which incorrect data in Salesforce could reach the reporting workflow and erroneously alter business decisions related to revenue. Without data contracts in place, these issues often fail "silently" until a business stakeholder notes in the report that the numbers look off given their domain expertise. Data contracts serve as a mechanism to surface the issue within the "prevention window" and implement a fix or block the replication before reaching the target database.

In this section we covered the four data asset categories you will likely encounter when considering data contracts and where to place them. In the next section, we will expand to utilizing metadata to align expectations with the data assets via data contract specs.

Contract Definition

Now that we know what data assets we want to put under contract, we need to a) establish our expectations as code via a contract spec, b) codify business logic within the contract spec, and c) compare the expectations within the contract spec to the metadata of the data assets under contract via schema registries and or data catalogs. The next section details each of these and the various considerations one should account for.

Data Contract Spec

In our conversations with practitioners, a common but misplaced question revolves around the standardization of data contract specs. While the spec is an integral component of data contracts, they are a tool to help implement the architecture of data contracts– the spec is not data contracts themselves. An analogy would be

the question of "What database should we use to implement a data lakehouse?" Regardless of whether it's Snowflake, Databricks, BigQuery, etc., all that matters is what you have available and what works for your business use case.

As of this writing, open source data contract specifications are available, but they are not widely adopted. In our conversations with companies, many are developing their own specifications to address the unique nuances of their data systems. As this architectural pattern gains further adoption and matures, we anticipate that various standardized formats will emerge as popular choices. However, the purpose of this book is not to recommend a specific format, but rather to detail the underlying mechanisms, regardless of the specification used. Figure 5-7, xkcd.com comic 927, further illustrates why directing you towards a specific spec is a futile task.

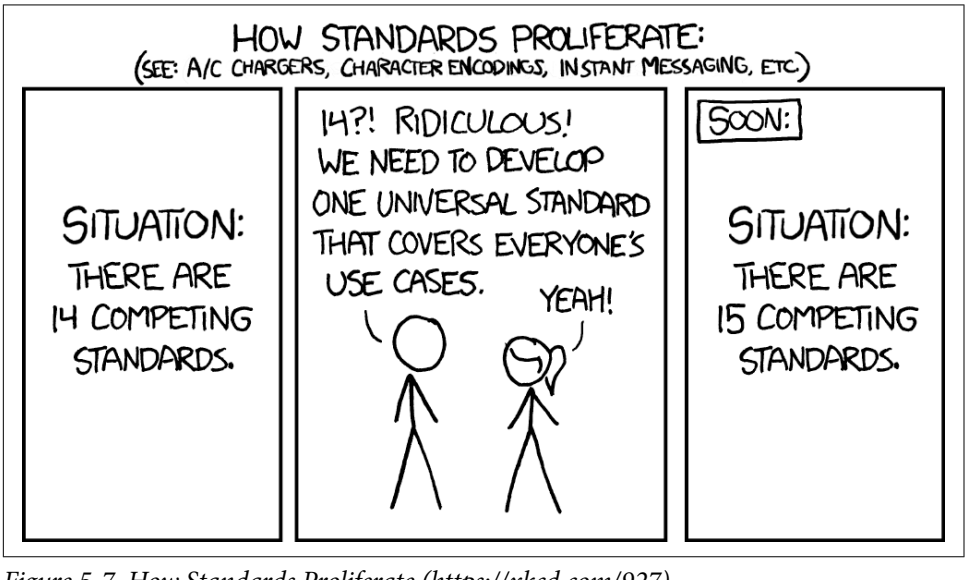


Figure 5-7. How Standards Proliferate (https://xkcd.com/927)

To create your own data contract spec, we recommend utilizing JSON Schema– "...a declarative language for defining structure and constraints for JSON data," and pairing it with YAML. In addition, JSON Schema is also part of a larger ecosystem of tools that extend its power, such as validators for various languages and databases. This chapter will only provide a high level overview, but we will provide further details and code in Chapter 8.

Key considerations of a data contract spec include the following:

- Ability to be technology agnostic to enable integration across the entire data lifecycle.

- Ability to describe expectations of a data asset in a human readable format–hence why YAML is a common format.
- Ability to cover both the schema of data as well as the semantics.
- Ability for the contract spec itself be semantically versioned, as contracts will need to evolve over time.
- Ability to account for the heterogeneity of the ways in which technologies define or constrain data (e.g. required typing in JavaScript vs optional typing in Python).

Below is a YAML data contract spec for the imaginary aquarium analytics company Kelp, who stores aquarium exhibit data in S3. Note that while we are using JSON Schema, JSON and YAML can be easily converted one-to-one.

```
spec-version: 0.1.0
name: AquariumExhibitStatus
namespace: Kelp
dataAssetResourceName: s3://kelp/region/aquarium/aquarium_exhibit_status.json
doc: Contract representing the status of a aquarium exhibit in Kelp's system.
owner: mark@kelp.com
schema:
  - name: aquarium_id
    doc: The id of the aquarium
    type: string32
    constraints:
      - charLength: 32
      - isNull: FALSE
      - isNotEmpty: TRUE
  - name: exhibit_id
    doc: (Optional) The id of an exhibit within a specific aquarium.
    type: union
    types: ['null', 'string32']
    default: 'null'
    constraints:
      - isNullThreshold: 0.8
  - name: exhibit_status
    doc: The status of the aquarium exhibit.
    type: enum
    symbols: ['OPEN', 'CLOSED', 'MAINTENANCE', 'FEEDING']
    constraints:
      - isNullThreshold: 0.3
      - length: 1
  - name: exhibit_location
    doc: (optional) The location of an aquarium exhibit.
    type: union
    types:
      - type: 'null'
      - type: struct
        alias: Location
        name: location
```

```
        doc: A geographic location
        fields:
          - name: latitude
            doc: The latitude of the location
            type: float64
            constraints:
              - isNull: False
          - name: longitude
            doc: The longitude of the location
            type: float64
            constraints:
              - isNull: False
    constraints:
      - isNullThreshold: 0.45
  - name: last_exhibit_update_time
    doc: The last known real-time update from the aquarium exhibit status (in milliseconds since the U
    type: date64
    constraints:
      - isNull: FALSE
      - max: today
```

In the subsequent subsections, we will break this contract spec apart into its key components of a) contract management, b) data schema, and c) data semantics.

Contract management

Key to infrastructure as code is the ability to manage the metadata being stored (i.e. meta-metadata) and keep track of its changes as it inevitably evolves. While technically this can be tracked via git blame and the version controlled history, that only partially satisfies the necessary requirements of a data contract. In addition, one must be able to build automations on top of these changes, hence the following values.

spec-version
: The version of the data contract spec utilized within your system, where this value will change with each iteration to the spec structure as your use case changes or becomes more complex (e.g. adding further semantic constraints).

name
: The user-defined name of the respective data contract, where `name` and `name space` form a combined unique identifier.

namespace
: A user-defined name representing a collection of data contracts, similar to a higher level "folder" within a repository.

dataAssetResourceName
: The URL path of the data source under contract, where it aligns with the naming pattern of the data source (e.g. `s3://<your path>/<your file name>` or `postgres://db/database-name`)

doc
: The documentation that describes what the data contract represents, enforces, and any other pertinent information.

owner
: The assigned owner (either an individual or group) of the data contract, and the contact information used to notify when a contract is violated, needs a change, or if an individual is trying to determine a point of contact for further context.

```
spec-version: 0.1.0
name: AquariumExhibitStatus
namespace: Kelp
dataAssetResourceName: s3://kelp/region/aquarium/aquarium_exhibit_status.json
doc: Contract representing the status of a aquarium exhibit in Kelp's system.
owner: mark@kelp.com
```

Data schema

Enforcement of data schemas will likely be the first use case implemented on a company's data contract journey and will protect the most obvious yet most disastrous upstream changes. With utilizing JSON Schema as the underlying language, one will need to map data typing to their standard and or utilize available extensions for additional languages (e.g Go, Rust, etc.). The standard JSON schema includes the following:

- string
- number
- integer
- object
- array
- boolean
- null

In addition, to the above standard types, it is also beneficial to enable `optional` and `union` typing to handle more complex typing use cases such as where a `null` value is valid for a `string` value for example.

```
schema:
  - name: aquarium_id
    doc: The id of the aquarium
```

```
    type: string32
  - name: exhibit_id
    doc: (Optional) The id of an exhibit within a specific aquarium.
    type: union
    types: ['null', 'string32']
  - name: exhibit_status
    doc: The status of the aquarium exhibit.
    type: enum
    symbols: ['OPEN', 'CLOSED', 'MAINTENANCE', 'FEEDING']
  - name: exhibit_location
    doc: (optional) The location of an aquarium exhibit.
    type: union
    types:
      - type: 'null'
      - type: struct
        alias: Location
        name: location
        doc: A geographic location
        fields:
          - name: latitude
            doc: The latitude of the location
            type: float64
          - name: longitude
            doc: The longitude of the location
            type: float64
  - name: last_exhibit_update_time
    doc: The last known real-time update from the aquarium exhibit status (in milliseconds since the U
    type: date64
```

Data semantics

As a company moves forward with their data contract use cases, one may want to implement constraints beyond schema and on the underlying data itself, as well as specific if-else conditions based on multiple data values. JSON Schema supports this via schema composition and subschemas.

```
schema:
  - name: aquarium_id
    constraints:
      - charLength: 32
      - isNull: FALSE
      - isNotEmpty: TRUE
  - name: exhibit_id
    constraints:
      - isNullThreshold: 0.8
  - name: exhibit_status
    constraints:
      - isNullThreshold: 0.3
      - length: 1
  - name: exhibit_location
    types:
      - name: location
```

```
      fields:
        - name: latitude
          constraints:
            - isNull: False
        - name: longitude
          constraints:
            - isNull: False
  constraints:
    - isNullThreshold: 0.45
- name: last_exhibit_update_time
  constraints:
    - isNull: FALSE
    - max: today
```

This section serves as an introduction to the data contract spec, but we will provide a more in depth description and use case in Chapter 8, where we will create a data contract spec from scratch and utilize it to prevent a breaking data change. In summary, data contract specs at minimum need to support contract management and schema enforcement. For more complex implementations, one may consider the requirement of semantic enforcement that puts constraints on the data values themselves and conditional properties of one or more data values within the schema. While there are numerous ways to implement a contract spec, and growing number of standards, we believe the JSON Schema framework provides a great avenue to understand the underlying mechanisms of a contract spec that you can continue using or enable you to have a strong understanding of what to look for when choosing an emerging open source standard.

Business Logic

In the previous section we highlighted the role of managing data semantics via the contract spec, which serves as a means to codifying business logic. We define business logic as the "the domain specific knowledge of a particular business function and/or sets of procedures that underline the operation of an organization." Similar to there being a data lifecycle, we argue that there is a business logic lifecycle as represented in Figure 5-8. Furthermore, like the data representing business logic, this logic is not static and is constantly iterated upon– where data quality issues arise when the data doesn't keep up in representing the business logic.

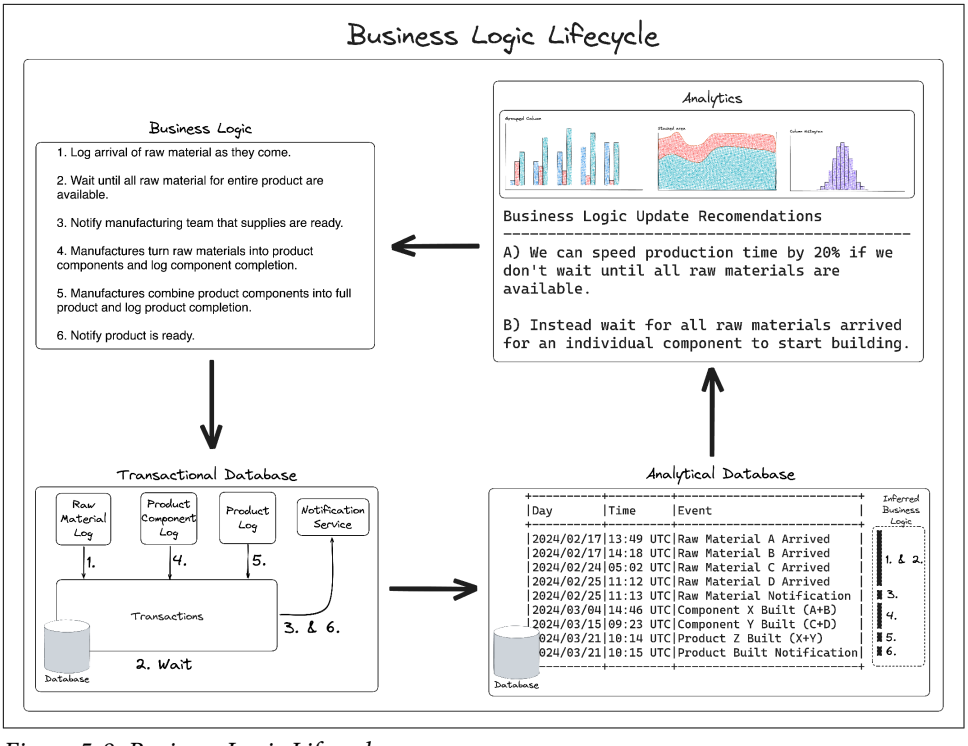


Figure 5-8. Business Logic Lifecycle

To summarize the business logic lifecycle:

1. Business logic that details the functions and procedures of the organization is established, such as "standard operating procedures."
2. Upstream engineers operationalize this business logic and store the various steps and their timestamps within the transactional database.
3. The data from the transactional database (and other sources) are replicated into the analytical database, where the data is transformed to infer the business logic.
4. Analytics are performed on top of the data within the analytical database to provide recommendations to the business and inform how the business logic should be iterated upon.

While this business logic is critical to the business, it's often maintained via "institutional knowledge" within teams and departments. If one is lucky, this knowledge is documented, maintained, and easily discoverable across the organization. Sadly, most people are not lucky. It's not uncommon for this knowledge to be shared via word of mouth, documented only in obscure Slack messages or emails, or worse only known by a single person who left the organization years ago. This is what makes data

contracts powerful, and why management fields within the contract spec are critical. While data contracts are not a full replacement for documenting business processes, nor do they try to be, they do provide a means to maintain the critical business logic as it pertains to data processes.

Specifically, the "contract management" section of the data contract spec creates a composite unique identifier for a particular data asset and its respective business logic. Furthermore, since this logic is stored as a YAML file within a version controlled repository, a history of changes to the data's business logic is maintained as discoverable code by any developer with access to the repository.

Schema Registry and Data Catalogs

The data contract spec captures the *expected* schema and business logic, but there needs to be a means of capturing the *actual* schema and business logic within an organization's data systems. While there are numerous ways to achieve this ranging from hacky implementation to managed vendor tools, the two we will focus on are a) schema registry, and b) data catalogs– where schema registry emphasizes stream processing and data catalogs emphasizes batch processing.

Schema registry

Schema registry provides a means for event publishers and subscribers to maintain consistency and compatibility between, and thus are primarily focused on event sourcing and streaming data assets. Figure 5-9 below provides a high-level overview of how we can utilize schema registry with data contracts within the CI/CD process. Confluent, the maintainers of Kafka Schema Registry, define the tool as the following:

> Schema Registry provides a centralized repository for managing and validating schemas for topic message data, and for serialization and deserialization of the data over the network. Producers and consumers to Kafka topics can use schemas to ensure data consistency and compatibility as schemas evolve. Schema Registry is a key component for data governance, helping to ensure data quality, adherence to standards, visibility into data lineage, audit capabilities, collaboration across teams, efficient application development protocols, and system performance.

It is important to note that while we've placed our emphasis for this book on Kafka and Schema Registry, as they are both open source and widely adopted, the overarching concepts still apply to managed tools such as AWS Kinesis and corresponding AWS Glue Schema Registry.

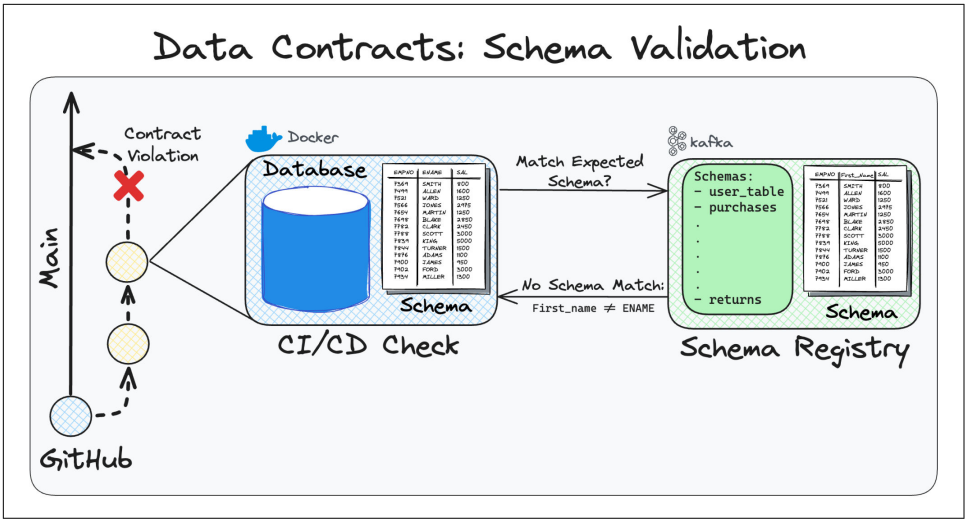


Figure 5-9. Data Contracts Schema Validation

Regardless of the tool, we'll use the same pattern to enforce schema via data contracts:

1. A developer creates a pull request where a data asset under a contract is changed.
2. The pull request kicks off the CI/CD workflow where a Docker image is spun up with a database.
3. The Docker database replicates a fraction of the changing data asset, where schema info can be extracted.
4. The extracted schema from the Docker database is compared to the source of truth of the respective data asset's schemas (e.g. Kafka Schema Registry).
5. The CI/CD test passes if the two schemas match, but will fail and block merging if there is a discrepancy between the two schemas.

This same workflow also applies to data catalogs, where the replicated database within Docker is compared to the metadata within the catalog. We will detail this further in the next section as well as highlight when you would want to use Schema Registry over a data catalog for their data contracts.

Data catalogs

Data catalogs serve as a centralized inventory of metadata within the organization with emphasis on discoverability, providing a data glossary, data lineage between assets, and defined governance. As of writing this chapter in 2024, data catalogs have become a hot topic with the major data infrastructure vendors making their data catalogs open sourced; specifically, Databricks open sourced their existing Unity

Catalog and Snowflake released Polaris Catalog. With that said, any tool will suffice for data contracts as long as there is a means to extract the stored metadata within the CI/CD workflow.

Schema registries and data catalogs both store schema metadata, but they differ in their implications for stream and batch processing. Specifically, data catalogs can consume metadata from schema registries, raising the question, "Which source should one reference for data contract enforcement?" Typically, data catalogs consume metadata via batch processing, which limits data contract enforcement to the timing of the batch intervals at the entire data asset level. In contrast, streaming data processes events in real-time, allowing schema registries to enforce data contracts at the record level. Thus, if your architecture warrants it, we suggest utilizing both as represented in Figure 5-10 below, where CI/CD both references Schema Registry and the data catalog.

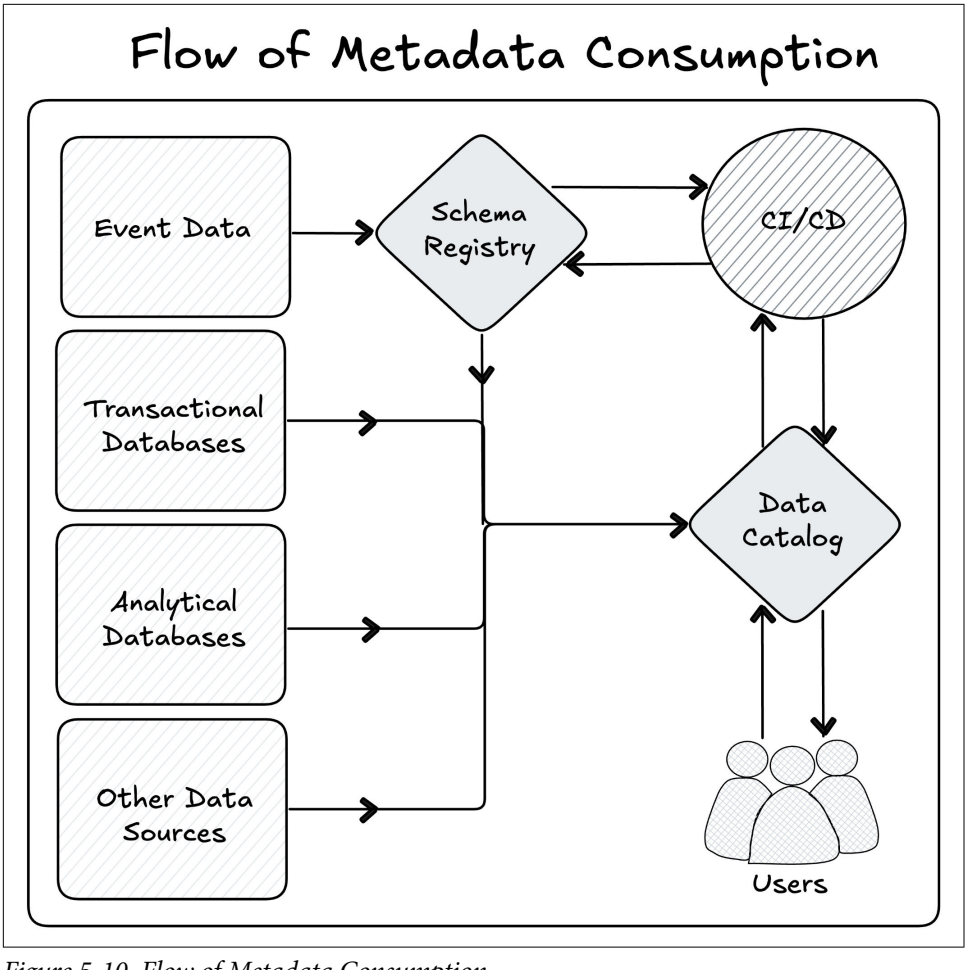


Figure 5-10. Flow of Metadata Consumption

Furthermore, data catalogs increase in importance for data contract enforcement once organizations move beyond just schema validation. As noted earlier, in addition to schema, data catalogs include other categories of metadata such as lineage, definitions, governance policies, and locations of various data assets within different databases. Chapter 9 details advanced applications of data contracts, but examples include:

- Enforcement of PII protection on data asset changes based on documented governance policies.
- Impact analysis of upstream changes based on data lineage.

- Data profiling thresholds to ensure data stays within expected ranges or follows specific regex patterns.

These advanced use cases are ideal, but we still recommend a crawl, walk, and run approach to data contracts where each iteration of implementation builds on each other to handle further complexity.

Conclusion

In this chapter we covered the initial two components of the data contract architecture: Data Assets and Contract Definition. Among the data assets, we detailed the differences between analytical and transactional databases, event sourcing and streaming, and first-party data on third-party platforms. While the data itself has wide variability, these categories of data assets are the ones you will most likely encounter and want to place data contracts on. In addition to data assets, we also described the underlying foundations for data contract definitions via the data contract spec, business logic, and centralized metadata sources such as schema registry and data catalogs.

Furthermore, we highlighted the underlying structure of a data contract spec with following key considerations:

- Ability to be technology agnostic to enable integration across the entire data lifecycle.
- Ability to describe expectations of a data asset in a human readable format.
- Ability to cover both the schema of data as well as the semantics.
- Ability for the contract spec itself to be semantically versioned.
- Ability to account for the heterogeneity of the ways in which technologies define or constrain data.

Chapter 6 focused heavily on the underlying data and metadata utilized by data contracts. In Chapter 7, we will continue describing the final two components of the data contract architecture– Detection and Prevention– which emphasizes how to take action on the data and metadata covered in Chapter 6.

Additional Resources

- CAP Twelve Years Later: How the "Rules" Have Changed
- Event Streaming and Event Sourcing: The Key Differences
- Domain Events versus Change Data Capture
- Making Sense of Stream Processing

- An Engineer's Guide to Data Contracts - Pt. 2

CHAPTER 6

Change Management: The Crux of People, Process, and Technology

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *mpotter@oreilly.com*.

In Part II of this book we covered how to implement data contracts from a technological perspective. While that's extremely useful, the reality of getting a new process adopted within an organization relies heavily on people– this is especially true for data contracts since it serves to bridge silos across the organization, where each silo has different motivations and constraints (e.g. software and data teams).

Part III of this book will detail how to get leadership buy-in for the data contract architecture, and specifically covers how to create a data quality strategy, creating your first wins with data contracts, and measuring your impact. We have spent years learning how to build adoption of data contracts within organizations, ranging from Chad implementing the architecture within his previous job at a startup, to us working on implementations at Fortune 100 companies. We hope this section of the

book helps you navigate the journey of adoption, and ideally avoid the hard lessons we learned along the way.

One lesson that stood out the most to us was that while it was relatively easy to get buy-in for data contracts from data engineers , leadership was still hesitant to invest. It's obvious to us now that leaders don't invest in technology– they invest in solving strategic problems, and technology is one of many levers to do such. Time and time again, the strategic problem that got the attention of leadership and ultimately investment in data contracts was *change management*.

Chapter 6 details this key lesson, and will serve as framing for subsequent sections of Part III of this book. To be clear, the idea of change management is not new, but it's extremely powerful in serving as a concept that easily translates pains felt between software engineers, data developers, and non-technical business stakeholders. Building upon the foundation of change management, we will finish the chapter with how you can create your data quality strategy.

The Importance of Change Management

What type of product is GitHub? Most people might say it's a "DevOps Platform." After all, everything GitHub facilitates is a component of DevOps: code review, unit tests, integration tests, code diffs, repos, merging and branching, subscribers, changelogs. But we see it differently. GitHub might be a DevOps tool, but it's a federated change management platform. Its raison d'être is to create automated processes that reduce the risk of changing code across a complex, heterogeneous codebase.

Branching and merging? You're parallelizing change. Pull requests, code review, and code diff? Human-in-the-loop change review. Subscribers, changelogs, notifications? Systems for being notified of change.

Why is change management important? Because change is the primary cause of quality and governance problems. Whether it's data quality issues, governance issues, compliance, or other regulatory problems—every one of these can be tied back to change: change to the codebase, change to business logic, change in data users, change in expectations, change in producers, change in consumers.

This is a problem for software systems because technology doesn't prevent change. In fact, it shouldn't. Change is an inevitable part of our industry. As highlighted in the *Business Logic Lifecycle* within Chapter 6, the product and company will evolve, as will our tools, infrastructure, frameworks, and ideas. This very book represents a change that may motivate you to create systems for managing change.

Given that change is such a ubiquitous part of the data industry, it's surprising that change management isn't discussed more openly. After all, according to Al Lee-Bourke's book on organizational change management, "the most common percentage

given to adoption and change management [within large projects] is 10% [of its budget]." Many companies have undertaken data migration projects from on-prem systems to the cloud costing billions, yet data change management remains an elusive component of those projects.

Data contracts, data products, DevDataOps, and other techniques we've discussed all fall under change management. As mentioned earlier, the intent of data contracts is not to prevent change but to help teams understand it, communicate when changes are happening and their impacts, and prepare for changes that would otherwise cause downstream systems to fail. That is change management, through and through.

Companies that need change management the most are typically larger organizations. The larger a company becomes, the more data it generates. The more federated a business becomes, the more complexity and replication occur. As data becomes more valuable, more spaghetti code emerges over time, increasing the impact a single change can have on downstream consumers. We covered this in more detail within Chapter 3, where we discussed the implications of Dunbar's Number and Conway's Law.

That isn't to say small companies don't need change management. After all, GitHub is the global standard for code change management. Even in small companies, code change management is necessary from day one because the code is crucial to the company's existence, so processes like pull requests and code review are seen as necessary to ship any functionality. The risk of simply SSHing into main is too great.

In companies where the core product is a physical good, like a sofa or television, change management is an important part of the product lifecycle. Want to change how the brake system works in a car? Prepare for months or even years of reviews, testing, regulatory confirmation, communication to customers, and so on. In other words, the more critical the system is to the company, the more change management becomes a no-brainer.

While it might surprise some readers in 2025, many engineering teams still don't use a DevOps platform or have version-controlled code. Can you guess why? If you guessed because those companies prioritize physical products or service-based relationships that don't rely on code, you'd be right.

So what does that mean in the data space? Fortunately, thanks to data products, we're seeing a rise in the utility and value of data. With generative AI on the rise, we expect AI will become a revenue engine equivalent to applications and websites. Furthermore, since data is absolutely essential to all models, the criticality of data increases significantly, and concepts of change management will likely become more prevalent.

Meanwhile, framing your data management initiatives in terms of business-critical data products provides the strongest rationale for investing in this space. We'll cover

that more later in Chapter 11 where we discuss creating your first wins with data contracts. For now, let's switch gears to discuss another subject close to our hearts: the data supply chain.

Data as Supply Chains

Gunpei Yokoi from Nintendo coined a term in 1975 called "Lateral Thinking with Withered Technology." He developed the concept after a string of hits from the video game company at a time when most games were straightforward and fell into a few distinct categories: puzzle games, fighters, racing games, and so on. But Yokoi found fun gaming experiences in unexpected places—fields with repetitive tasks that weren't seen as enjoyable at all! For example, running a farm (*Harvest Moon*), cooking for many hungry customers (*Overcooked*), or being a lawyer (*Ace Attorney*).

Yokoi discovered that an old idea, applied in a new context, could breathe new life into it. He found inspiration in everyday surroundings instead of following common patterns in the video game industry. We think the data field could use a bit of lateral thinking as well. It's common for us data developers to look to our cousins in software engineering and try to learn from their best practices. However, even software engineers are novices in change management! Other industries like healthcare, manufacturing, and logistics have been dealing with quality issues for hundreds of years, with far greater stakes.

Changing how you treat a patient for a dangerous disease risks harm if you're not extremely careful. Changing a shipping lane might delay delivery, or the ship could hit rocky waters and potentially capsize. When lives are at stake, change processes are taken seriously.

One of the most sophisticated areas where change management comes into play is the supply chain. A supply chain is an interconnected network of producers and consumers working together to ensure a physical product reaches a customer with a certain degree of quality and timeliness (sound familiar?). Producers start by extracting raw materials from the Earth or another source—think farming, mining, and woodcutting. Once collected, the material is transported to a manufacturer. Since people can't use raw logs or iron ore, a processing facility turns the goods into something useful. Usually, there are steps between transforming the raw good into something consumable by an end user. One plant may treat a chemical, another might handle bottling and packaging, and a third might test the quality of the mixture. After manufacturing, the product is shipped to a distributor. The distributor typically owns a warehouse or central store where they hold various transformed materials, making them available to retailers on demand.

Retailers are consumer-facing. They are the interface between buyers and products. When you walk into somewhere like Walmart, consider the problems they must

solve: maintaining a large inventory with high availability; ensuring the products they sell are of good quality and not contaminated; understanding who is purchasing which products and which supplier provided a certain product at what price. They must track all their costs, including storing goods and moving them between physical locations. They also need to ensure their wares are clearly labeled and discoverable so any shopper can find what they need quickly and check out.

Finally, consumers are the stakeholders getting utility from the product. They are the ones going to the store, handling the browsing, shopping, and item selection. If what they picked doesn't meet their needs, they might return it or complain.

Figure 6-1 below illustrates these parallels between supply chains and data workflows. Every component of a supply chain is present in the data space. The miners or farmers are equivalent to data producers; the trucks transporting goods between processing centers and retailers are the data pipeline; the manufacturers are the data engineers, accepting raw data and turning it into something usable by a downstream customer; the distributor is our data warehouse. Heck, it's even called a *warehouse* if it wasn't obvious enough. The retailers are the tools and systems providing utilities to leverage data in valuable ways like Tableau, Power BI, Amazon SageMaker, and so on. And of course, our end customers are the data scientists, analysts, and product managers we serve every day.

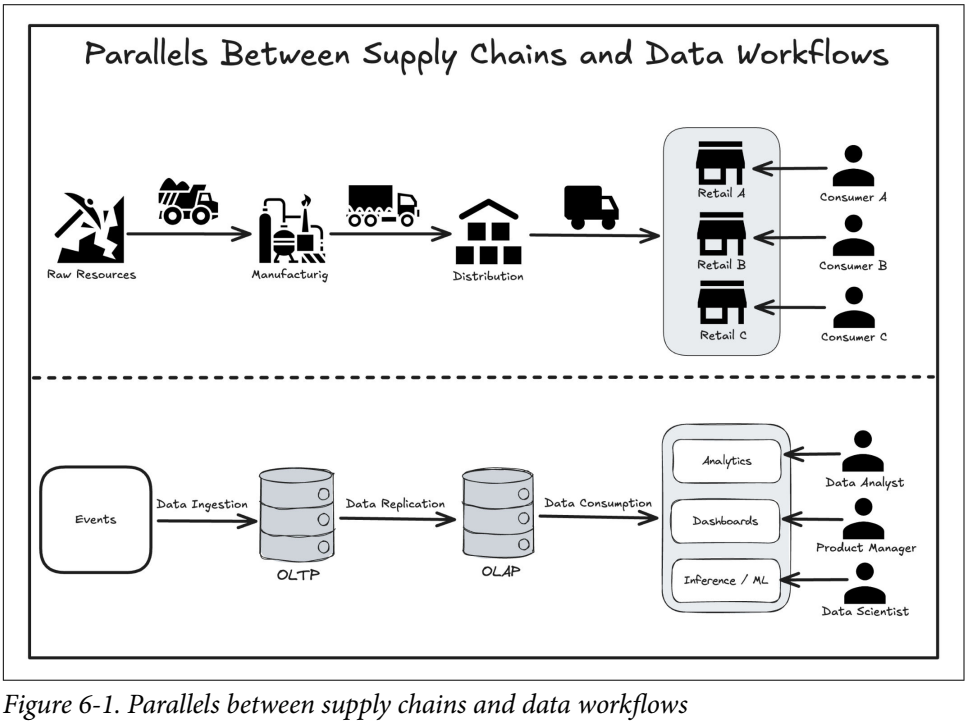


Figure 6-1. Parallels between supply chains and data workflows

Why is it useful to think about our work in the context of data supply chains? It starts to dawn that supply chain management is as much (if not more) a people problem as a tech problem! In fact, supply chain managers leveraged contracts between producers and consumers long before they existed in software engineering. A farmer will sign a contract with a cereal company, for example, to provide a certain amount of grain on a particular schedule, to a degree of quality defined by the consumer. After all, it's the restaurants that understand what is fit for human consumption, not the farmers who sit far from the line cooks!

What is essential is understanding how changes ripple across the supply chain, who is impacted, what that means for affected teams, and what action each party must take next. Freely communicating this information, tracking how changes have been applied, and creating open communication is a must in a fast-moving supply chain where disruptions are common and teams need the ability to collaborate in real time as climate disasters or geopolitical events change their ability to produce or consume.

Good data change management puts people at the core—not tech. Facilitating conversation and a free flow of information is more important than tests! Helping people understand how they are impacted is more important than catalogs, and giving all impacted parties the right distribution of power for what decisions to make next is essential. In the following section, we will discuss what this actually looks like across the stages of implementing data contracts.

Levels of Data Contract Implementations

The first step with implementing contracts is to require data contracts on source data in order for that data to be pushed into a platform. This is a great starting point for a few reasons. First, it's within the control of the data platform team. You built the platform, so you decide what goes into it. Simple enough! If a data producer wants their data to be available for querying—perhaps because a product manager needs it for analytics—they have to go through the process of creating a contract, taking ownership, and managing change communication to the broader team.

A great pattern we've seen implemented successfully at several large-scale enterprises is for the data platform team to maintain a group of data product managers. When someone fills out and submits a new data contract through a portal (or catalog), the data product manager collaborates with the producer to define the data, clarify its meaning, establish the producer's perspective on ownership, and even create the YAML version of the contract stored in Git. There are more sophisticated implementations where teams require producers to upload a snapshot of their data, and the platform team automatically collects useful statistics and descriptions about the data. This can be used as a baseline for building the initial specifications of the contract and data quality rules.

There are some major challenges to this approach, which we will cover in a moment. But first, there's a metaphor we are going to use that we hope will make the different levels of implementation easier to understand. These categories are not unique to data contracts but represent how teams typically think about internal implementation of technologies in various fields, potentially explaining why many of these initiatives tend to fail.

Airplane and Airline Projects

In 1903, Orville and Wilbur Wright invented the first functioning airplane after taking off from Kitty Hawk, North Carolina, and landing 120 feet later. This invention was a seminal moment in aviation history. The brothers discovered that arranging wings with a specific airfoil shape, control surfaces like elevators and rudders, and a lightweight yet strong frame allowed the vehicle to achieve lift, maintain balance, and be steered in flight. This design causes air passing over and under the wings to create differential pressure, generating lift. These factors together enabled flight.

A decade later saw the dawn of the first airline, the St. Petersburg-Tampa Airboat Line. Created by Percival Fansler, the airline faced different problems. How do you make people feel safe enough to fly between cities? After all, airplane safety during the early 1900s wasn't at today's level. Airplanes lacked advanced navigation systems, were made of wood and fabric, and accidents were common. Fansler needed to address this. They invested in safety measures like rigorous pilot training, regular aircraft maintenance, and establishing standard operating procedures.

But this was just the beginning. Airlines had to consider the ***incentives*** that drove people to purchase tickets: Great customer experience, affordability, and awareness relative to competing options. Comfort is extremely important in travel. A person is often willing to sacrifice time for an enjoyable experience. Seating that wasn't stiff or painful, in-flight meals, and attentive flight attendants made the experiences superior to train rides or sea-travel. They then adopted technologies like pressurized cabins and more reliable aircraft to reduce turbulence and noise.

Second came ticket prices. Airlines addressed affordability through offering classes like economy to make flying accessible and government subsidies to lower ticket prices. They made it a goal to achieve an economy of scale, offering flights to many cities, states, and countries in order to drive down prices to be competitive with other forms of transportation. To do this they needed to create schedules, follow strict flight paths, and improve ticketing systems and boarding speed to get people on and off board faster.

To *convince* people to fly more frequently, airlines launched marketing campaigns that emphasized safety records, showcased the speed and convenience of air travel over trains or ships, used celebrity endorsements, and portrayed flying as a glamorous, modern experience.

Figure 6-2. Eastern Airlines News Paper Ad - Evening Star (Washington, D.C.), May 8, 1950

In short, the Wright brothers focused on proving *technical viability*, while Percival Fansler focused on *business viability,* or adoption. Technical viability is a subset of business viability. Just because you've proved something *can* be done does not mean you've solved the myriad requirements necessary for customers to regularly use your invention instead of alternatives.To that end, what we've termed an *airplane project* focuses on implementing a particular technology without considering other factors, while *airline projects* go beyond providing a technological capability and focus on what incentives customers to use that technology.

Some projects are primarily technical endeavors. For example, most ETL/ELT tools are airplane projects. As long as you *can* move data in near real-time between systems, the data engineering or platform team that built the technology can roll it out wherever it makes sense. Other examples might be cost optimization tools. If you can optimize the cost of your Snowflake instances, then you've solved the most critical problem. You don't need to think much about the interaction layer because you're unlikely to have users outside the team who created the project.

However, approaching an airline project as an airplane project can be disastrous. One reason most data catalog implementations fail is because they are airplane projects. It is less about whether or not some system *can* catalog all the data in your analytical database, and more so what will incentivise business stakeholders and data teams to use and derive value from the catalog. There's a long tail of problems and challenges your customers might experience that have nothing to do with technical feasibility. For example, maybe your tool can facilitate adding descriptive metadata for each table, but do customers go through the effort of doing that? Why or why not?

Data contracts fall into the airline project bucket. On the surface, implementing a contract is simple, perhaps even trivial. An Excel spreadsheet can contain your contract, or a Confluence page. But the purpose of a contract is for producers to take ownership and for consumers to gain value from it. Software engineers must be willing to create and manage these contracts iteratively. Thus, data teams need to ask themselves:

- Why would software engineers work with data contracts?
- Why might working with data contracts be challenging?
- Where might software engineers resist that ownership?
- Do software engineers have the time to own more surface area?
- Do software engineers even know what the data contract should be?
- How do software engineers decide what goes into a data contract?
- Are the created contracts useful to consumers?
- Does implementing data contracts solve downstream problems?
- Are they applied to the data sets consumers actually need?

- Are violations actually addressed in a timely fashion?

The answers to these hypothetical questions are dependent on your specific use case and organization, but regardless, all data teams need to consider the implications of the data contract beyond the data team itself.

Forward and Backward Looking Problems

With the airline context in mind, let's return to our earlier implementation of data contracts. For an initial implementation, it's not bad as an airline project because it accounts for the *why*. The reason a producer fills out a contract is because they want to move their data into the data platform team. Great. But that raises the following questions:

- What if the data producer doesn't want to expose their data to the broader business?
- What if they don't have a product manager pushing them to use the data for analytics?
- What happens if the producer decides to change their data from the initial contract?
- What happens when contracts are out of date?
- What about all the old data that doesn't have a contract?
- How do data producers communicate when changes have to happen?
- How do data consumers know which data has a contract and which doesn't?

We call these the *forward-looking problem* and the *backward-looking problem*. The forward-looking problems are about how we deal with changes to data or new data being added that isn't included in the platform. The backward-looking problems deal with how consumers access the data they want that isn't under contract, or even learn what source data is available and how they should use it.

Another big problem is a lack of scalability. Generally, there aren't many data product managers or data engineers on a team. These central data platform organizations are already bottlenecks, overwhelmed by data quality issues and outages they have to handle. If this already swamped team now becomes the central change control agents for every data change made in the entire company, it results in a situation not much different from centralized DevOps teams responsible for approving every PR. What we need to do is *shift left* (there's that phrase again) and distribute some of the responsibility of data management onto everyone.

Challenges of Technology First Implementations

The second implementation we've seen is more technological. Teams might build an automated testing framework or use tools like Protobuf and Avro to manage schemas, then document those schema-based contracts in some central catalog that the data producer maintains. This is definitely *shifting left*, but these implementations typically run into a different set of challenges, such as:

- Why would the data producer adopt a system like this? What's the incentive?
- How does the producer understand what the contract should be?
- What happens when the producer needs to make a breaking change?
- What happens when non-schema changes occur? Semantic or context changes?
- How do you scale a system like this to every engineer in the company?

Companies going down this route usually say the same thing: "We built the technology, but we're struggling with onboarding..." In engineering-driven organizations with lots of build decisions being made, you encounter this phrase often. This is because the problem is being treated as an airplane when it's actually an airline project. It's equivalent to asking: "Hey, we built a rickety plane that can cram a couple of people inside with no safety precautions and doesn't take people where they want to go... why isn't anyone flying with us?"

To answer that question better, it helps to explore a concept called *friction*– or blockers that prevent your target audience from engaging in a desired action and or workflow. Chad first learned about friction from Peep Laja, who ran the CXL Institute for Conversion Optimization back in the early to late 2010s. Click rate optimization (CRO) is a discipline of analytics and website development that focuses on examining user experiences, creating hypotheses on how to improve those experiences, testing various options, and measuring the results with the end goal of increasing "conversion," or the percentage of users that complete a task from the total population of users that *could* complete that task.

Let's use an example from this author's personal experience: Donating to charity. I (Chad) do not consider myself a small-hearted person, but I rarely donate to charity regularly (Let's just say it's not challenging to fill up time on my calendar!). However, whenever I go shopping at a grocery store, the credit card scanner frequently asks if I'd like to round up my purchase to support one of two or three charities that have been pre-selected for me. When this happens, I almost always donate. Why? Friction.

If I were to donate to charity on my own, I'd need to first find a charity to donate to. This can be overwhelming given there are thousands of charities with some being significantly more reputable than others working on projects that may be more or less important to me. I'd need to research these charities using search engines and find

one with a compelling cause. Then I need to choose the donation amount. Should I donate a dollar (seems too low) or enough that it could be a tax write-off? If it's high then do I need to have a conversation with my partner first? Then I need to go through the actual process of making the donation. Hopefully it's easy, and there is a PayPal integration, but if not I'd need to add my credit card number and other details then at minimum fill in the billing address plus potentially additional details.

Compare that to the donation experience when shopping. I've swiped my credit card to pay so I don't need to enter any additional information. The charities and the amount are preselected for me. The amount is negligible such that it doesn't require any additional outside confirmation. The cost of donation (outside the 45 cents) is pressing a button on the screen that says 'I'd like to donate.' The difference in friction between the two options is night and day.

The concept of friction is incredibly important in airline-oriented projects. Whenever you are asking a customer or stakeholder to do something they are not naturally inclined to do, any friction in the user experience will significantly decrease the likelihood of a successful event. If you are attempting to encourage data producers to be own their data and be more thoughtful of downstream consumer teams, there are many examples of potential friction points you will run into:

- Asking engineers to manage systems they are not familiar with
- Asking engineers to be thoughtful about the data they are producing
- Asking engineers to understand who is using their data and derive contracts from that
- Asking engineers to adopt a process that slows them down from shipping code
- Asking engineers to adopt a solution that prevents them from making needed changes
- Asking engineers to adopt a process that relies on humans making consistently correct decisions
- Asking engineers to care about consumers when they don't know who those data consumers are

Ultimately, adoption of data contracts boils down to incentives and friction. If you can align the incentives of the various stakeholders in the data supply chain while *making the right thing to do the easy thing to do*, you will be far more effective than simply proposing technologies and hoping that "if we build it, they will come." (Pro tip: This never happens.)

The Trap of Standards

We want to spend a bit of time addressing "standards." Inevitably, for all new software engineering or data engineering paradigms, some system emerges that claims to be a standard or aims to become one. We can respect the effort of the teams pushing these standards onto others—whether it's Protobuf or Avro, Iceberg, or one of the many opportunistic data contract standards that have emerged over the last few years as this concept has gained popularity. However, in our experience, the winners of standard wars emerge naturally over time as developers understand the use cases and actively solve problems using that standard.

For example, OpenAPI has become a standard because it provides a unified way to describe RESTful APIs, making it easier for developers to build, test, and integrate services. dbt is another open-source technology rapidly becoming the standard for analytics engineering. Standardization didn't arise because the industry made an explicit decision to follow the standard but because the tool added so much value that everyone adopted it to solve their problems. We have worked with many companies following a data contract standard, and to be frank—it adds very little additional value compared to simply defining your own YAML specification with your own use cases baked in. The far larger problem is not which spec the industry decides to use but how you solve the problems of adoption, ownership, visibility, and change management we've been discussing throughout this chapter.

In other words, focus on the difficult existential problems that will determine whether your data contract implementation succeeds. Once the above issues are solved, the industry will rally around a specification that makes sense for the common use cases we've uncovered collectively.

Requirements of a Successful Data Contract Implementation

Now that we know the challenges, let's lay out some of the requirements of a successful system. In the following section, we'll get into the weeds about how to implement such a system starting from scratch, but let's start by focusing on the high-level needs based on what our data producers need for adoption and what data consumers need to improve their data usability.

Visibility

Data producers need to know where their data is going and who is using it. In other words—a dependency graph! Engineers are familiar with the concept of a dependency graph and use it in many areas of their day-to-day work, like API management and security. If you understand the services, systems, applications, teams, and products that leverage the data being produced, you can begin to make choices proactively with the context of how your system is impacting others in the organization.

The second phase of data visibility is impact analysis. As mentioned in other chapters, data producers rarely understand how their data is being used. Without this context it is difficult to make informed decisions. If I am making a change to schemas or business logic, who should know about it? When should I tell people? Who should I tell? What information must I communicate? What actions should I take to minimize any damage being done?

Change communication

When data sources change, producers need to understand who to speak to, what to communicate, and when to communicate. Engineers are good at communicating their changes, but if the change management process is to push updates to a central Slack channel or email, that is simply not going to get the job done. The primary reason is that there are many transformations between the producer and the team that ultimately uses the data. If software is making changes in Java, which pushes data into a MySQL database, which pushes data to a Kafka topic, which pushes data to an S3 bucket, where an ELT pipeline is set up to dump data into Snowflake, which is transformed in a data model again and again, and ultimately ends up in training data for a critical pricing model—neither side at either end of the data supply chain will have the context that is useful to them.

One option to solve this problem is to funnel all change communication to the central data platform that operates as the 'gate' to the data warehouse, but ultimately this runs into the challenges of scale we mentioned before. If data is changing all the time, and it is the responsibility of an over-encumbered platform organization to manage communication on behalf of the producers, they become the bottleneck. We recommend this approach for smaller companies and startups where the rate of change is low or the number of downstream use cases is fewer.

Policies

A quote from a Distinguished Engineer at a large bank: "We have lots of policies. The problem is enforcement." A policy is a requirement for how data is managed, analyzed, modified, or created. Policies lay out regulatory and compliance requirements, access-control specifications, querying allowances, and more. In order to have a successful data contract implementation, you must have a way to define the rules and policies that can be cataloged and referenced in data contracts. These policies must be created centrally and are explicitly tied to a mechanism of enforcement in the data management system of choice. Policies should operate at a higher level of abstraction, with the ability to transpile to and from the underlying technologies where enforcement occurs. This allows governance teams to build policies based around business rules without needing to become full-fledged software or data engineers themselves.

Context evolution

In most cases, the evolution of schemas is relatively straightforward. Types might evolve, columns may be subject to standard to CRUD operations, new schemas might be created or removed depending on the events being populated by application developers. However, there are other changes to business logic that fall outside this context. For example, what happens when the underlying meaning of a field changes, or features are built that modifies how a property should be leveraged elsewhere in the codebase? As an example, imagine the field `promo_code` changed so that it is only valid if `promo_expiry_date` is in the future. This is a logic change that could significantly affect the number of valid promo codes moving forward. Simple checks against schemas would not be enough to detect this change, as no types are being modified or removed. Systems must exist to not only catch schema and row-level data shifts, but also tools that can detect semantic drift in important business objects as well.

Contract management

Even if all the other requirements to implement data contracts are put in place, adoption will stall if there is no system created to manage the evolution of the contracts themselves. Contracts must be owned, highly visible, and versioned. There must be a way to associate data contracts with the underlying assets they support - systems to track how those assets have evolved over time, and whether the changes were in compliance with policies. Changes of ownership should be recorded, incidents documented, and outages communicated. The contract is your source of truth for what a data product represents, but that truth is only valuable if the full context of change is made visible and clear to all parties. Equivalently, there must be mechanisms to both detect when data contracts are missing or need to evolve. There must be systems to make requests for data that currently does not exist, or requests to alter existing data. As we have stated previously, the power of data contracts is in their ability to promote collaboration around what the data means and how it is governed. However, this requires processes that allow a variety of different technical and non-technical actors to make their voices heard and participate in the process of data management.

In the next section, we will discuss in more depth how to bring technical and non-technical actors together by providing a framework for introducing data contracts to your organization.

Developing a Strategy for Introducing Data Contracts

In the previous sections of this chapter, we detailed the considerations of a successful data contract implementation, but how do you actually get buy-in from leadership to start? The following sections will illustrate a set of frameworks one can use to start building buy-in through the lens of introducing data contracts within your

company. Specifically, these are the frameworks that the management consulting firm McKinsey & Company uses called the *hypothesis-led approach*, as described by former McKinsey consultants as the following steps:

1. "Define problem - What key question do we need to answer?"
2. "Structure problem – What could be the key elements of the problem?"
3. "Prioritize issues – Which issues are most important to the problem?"
4. "Develop issue analysis [and] work plan – Where and how should we spend our time?"
5. "Conduct analyses – What are we trying to prove [or] disprove?"
6. "Synthesize findings – What implications do our findings have?"
7. "Develop recommendations – What should we do?"

We will use this framework to conduct a thought exercise to illustrate how you could potentially develop your data contract strategy.

Define the Problem

A phrase we use with each other when formulating problem statements is the idea of "going deeper" when trying to understand what's worth solving. This process is deceptively hard, as one can quickly settle on an appealing problem statement that ultimately doesn't address the root of the issue. This process is exactly what led us to "change management" as the key driver of data contracts among organizations that have adopted the architecture.

The below list of statements is example of how "going deeper" led us to change management as the core issue, where each subsequent statement goes deeper into the issue:

1. Data quality is difficult for data teams despite data being their specialization and this problem has persisted for decades.
2. Data governance exists and is effective in providing a framework to manage data quality issues, but is limited in its enforcement.
3. Data governance is challenging to enforce as many breaking changes to data come from upstream workflows– such as application code and business logic– both of which are not owned by data teams.
4. Teaching data best practices to upstream teams would be ineffective, as there is high inertia to change current workflows; especially when downstream issues don't directly impact those who made the change.
5. There needs to be a mechanism to automatically enforce expectations of data to all parties who plan to make data-impacting changes (i.e. data contracts).

6. Again, inertia in changing workflows and taking on additional dependencies will likely not be well met by upstream teams without proper context of "what's in it for them."
7. Having data contracts managed programmatically and enforced via the CI/CD workflow is key for fitting within the developer workflow and addressing the issue of inertia.
8. If we position data contracts to enforce workflows only on the most important data assets (e.g. revenue driving or risk mitigating), we can provide the *why* it's important and *what's* in it for them.
9. The root of the change that we are requesting of our upstream stakeholders is the process of change management.
10. How do we create the most effective change management workflow in respect to changes that impact the data that's important to the organization?

Now you may be asking yourself, "at what point do I stop going deeper in trying to find the root of a problem?" You can always "go deeper," but we found that it's mainly warranted when you have not found traction with the level you are currently engaging with. For example, the above chain of sequentially deeper thoughts was the culmination of four-plus years of trial and error in implementing data contracts at various organizations.

Specifically, we once stopped at *Statement E* "There needs to be a mechanism to automatically…" as we initially thought it was sufficient in resolving data quality issues. While this was true among data teams, expanding implementation beyond them left us stuck once again, prompting us to revisit and validate our assumptions more deeply. In a few years after publishing this book, we may very well have found that we *still* need to "go deeper" as we uncover more information through additional implementations.

Structure the Problem

From the above process of "going deeper," we have identified the problem statement of "How do we create the most effective change management workflow in respect to changes that impact the data that's important to the organization?" With this question in mind, we must identify what are the key "levers" one can pull to solve this question. The folks at McKinsey suggest you utilize an "issue tree" where you break a problem down into its subcomponents. Figure 6-3 below provides a simplified example of this exercise .

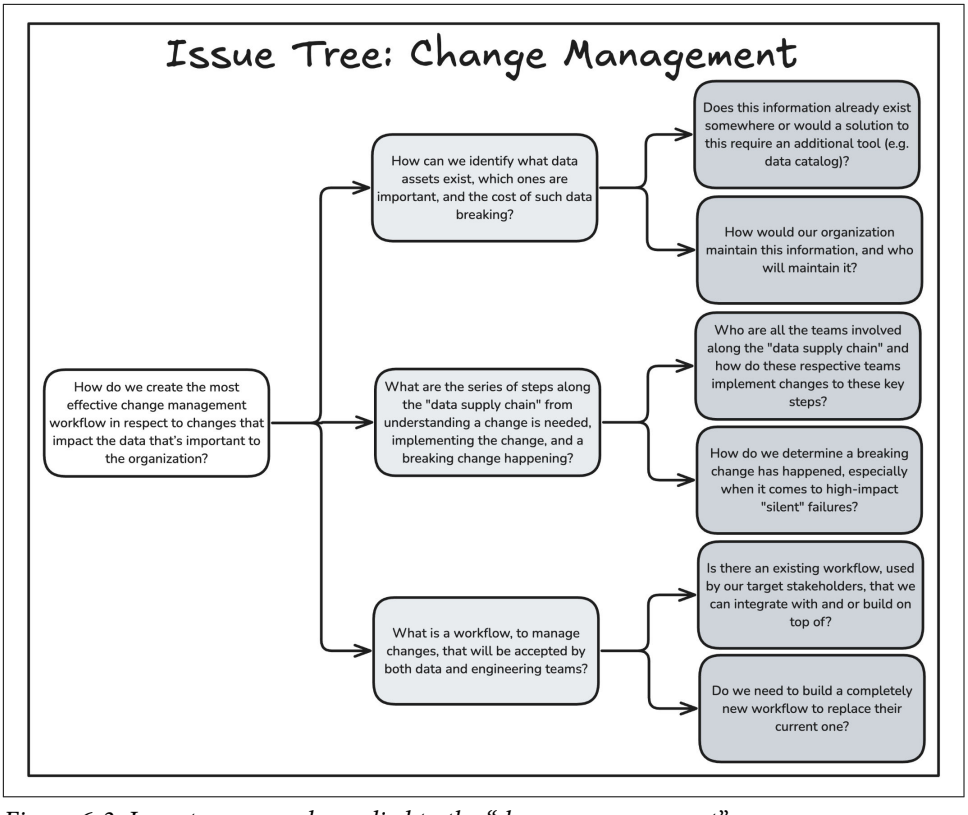


Figure 6-3. Issue tree example applied to the "change management" use case.

One important note for creating this structure is the presence of a hierarchy in how the issues are organized. Having a hierarchy further informs one of the prerequisites of solving a particular issue and thus a starting point to explore a question further.

Prioritize the Issues

In the previous step in the thought exercise, we determined what were the key elements of a problem that needs to be understood to take action upon. While our simple example resulted in only six elements, a real-world implementation would likely result in substantially more elements, and with even more limited resources constraining which questions one pursues. Thus, it's best to take these elements and prioritize them via a matrix, such as Figure 6-4, with relevant criteria as the axis (e.g. "impact" and "ease of implementation").

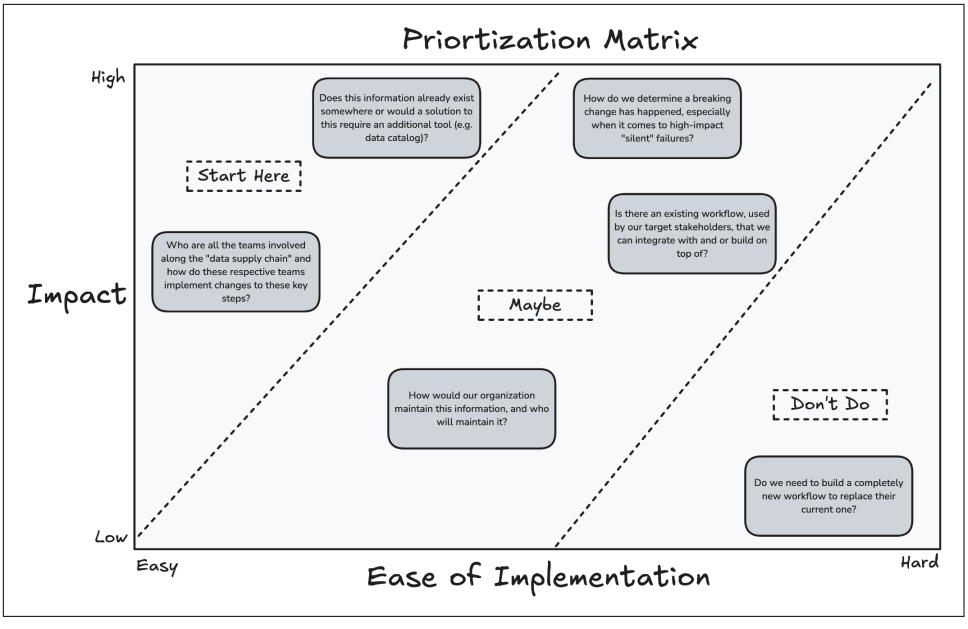


Figure 6-4. Prioritization matrix applied to the "change management" use case.

Based on the position of the elements of the matrix, one can determine where to start and delegate tasks across a team used in the subsequent step of "analysis and work plan." Furthermore, such prioritization also quickly identifies which elements one would not pursue.

Develop an Issue Analysis Work Plan Conduct the Analyses

This is the stage where the strategy development moves from thinking to execution, and where one will establish a work plan to validate the various questions and assumptions within the strategy. Below are examples with the "start here" elements we identified within the prioritization matrix:

"Does this information already exist somewhere or would a solution to this require an additional tool (e.g. data catalog)?"

- Search for existing documentation related to the data assets we utilize.
- Determine if the organization already uses an existing tool to manage this information, and whether this tool is used across the entire organization or within specific teams.
- Among the data assets we utilize, identify the various databases within the organization where the data is created, ingested, replicated, and transformed.

- If an existing tool isn't present, scope out the requirements of such a tool and determine if it's a build or buy decision.
- If determined to be a buy decision, begin researching various vendors and starting early procurement conversations.

"Who are all the teams involved along the "data supply chain" and how do these respective teams implement changes to these key steps?"

- Among the data assets we utilize, identify the various databases within the organization where the data is created, ingested, replicated, and transformed.
- Among the identified databases, identify which teams within the organization interact with the databases, manage the databases, and are held accountable for the respective data within the databases.
- Among the identified teams, identify who we already have an existing relationship with, who we need to build a relationship with, and who are the key decision makers among these teams.
- Identify any existing internal documentation regarding how these respective teams implement changes for workflows related to the databases.

All of these tasks will generate further information that will need to be synthesized in the next step.

Synthesize Your Findings

Another framework heavily utilized by McKinsey consultants is the Pyramid Principal, where information is synthesized in a way that emphasizes persuasion. Specifically, the Pyramid Principal is structured in three layers:

1. What decision you want the audience to make.
2. Supporting arguments for the above.
3. Supporting data for the arguments.

In Figure 6-5 below, we illustrate the pyramid principle utilizing our previous steps in this thought exercise.

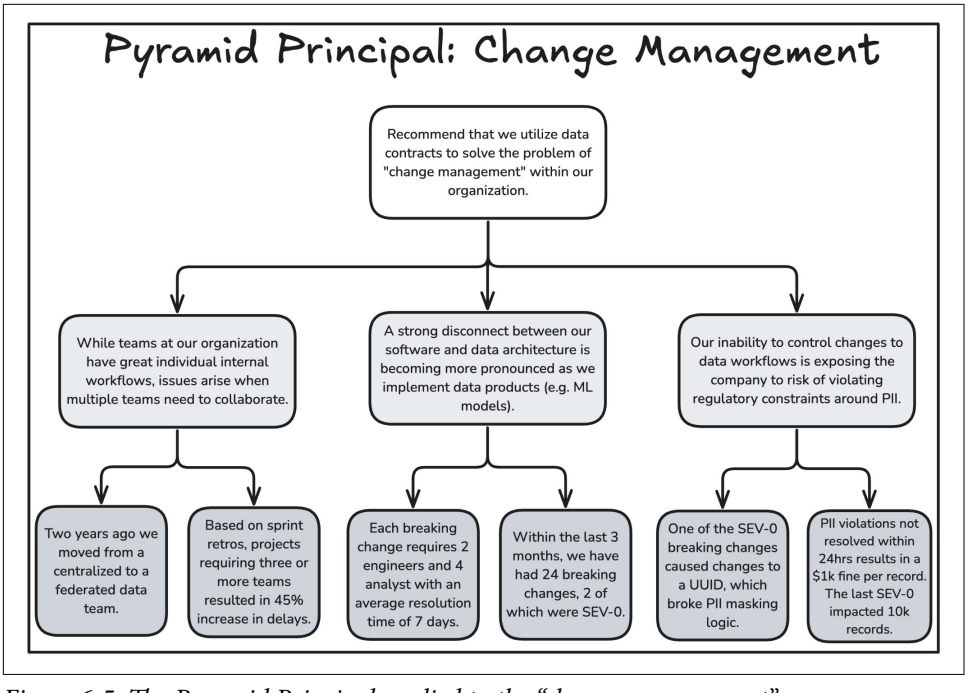


Figure 6-5. The Pyramid Principal applied to the "change management" use case.

Note that you will need to adapt what arguments and evidence you present based on the audience you want to persuade. With that said, the above is a means to structure your synthesis, but the next step will focus on how to present it.

Develop Recommendations

The final framework used by McKinsey consultants is the SCQA (also known as the SCR Framework), which stands for "situation, complication, question, and answer." Applied to our above use case, it would look like the following:

Situation:

> "Numerous breaking changes have occurred in the past 3 months that at best requires ~8k person-hours to resolve the issues, and at worst exposed the company to a $10M regulatory fine."

Complication:

> "A strong disconnect between software and data teams has arisen with our shift to federated team structures, where it has been identified that specifically change management between teams is the root cause– as seen in sprint retros where it was noted that projects requiring three or more teams resulted in a 45% increase in delays."

Question:

> "Is there a way to resolve this issue without reverting our decision on federated teams that we have invested heavily in?"

Answer:

> "Yes, we can better handle change management across multiple teams at scale via data contracts, which would prevent breaking changes such as the recent SEV-0 that exposed the company to a potential $10M fine if it wasn't resolved in less than 24 hours."

Throughout this thought exercise we applied the *hypothesis-led approach,* utilized by McKinsey & Company consultants, to a) develop a strategy to introduce data contracts into an organization, and b) develop messaging to persuade stakeholders that this strategy is worth pursuing. If you want another example of this process, Mark wrote the following article *How To Make Leaders Pay Attention to Your Next Data Initiative* where he applied the same process to a failed AI initiative use case.

Conclusion

In this chapter we discussed the importance of change management when considering the implementation of data contracts, as well as how we came to this conclusion via our implementation of data contracts at various companies. In addition, we applied the frameworks used by the management consulting firm, McKinsey & Company, to illustrate how to develop a strategy to introduce data contracts within your organization. In summary, this chapter covered:

- The importance of change management and how it's the root of the problem that data contracts solve..
- Viewing data workflows as supply chains within your organization.
- The levels of data contract implementations and their various tradeoffs.
- Developing a strategy for introducing the implementation of data contracts.

While we provided concrete examples, it is paramount that you apply these concepts through the lens of your organization's specific nuances. We believe data stacks are akin to thumbprints, where there are categories of thumbprint patterns, but each thumbprint is different. The specific nuances of your organization, and how your organization manages change, are the key levers you can pull to get adoption of data contracts.

About the Authors

Chad Sanderson is one of the most well-known and prolific writers and speakers on data contracts. He is passionate about data quality and fixing the muddy relationship between data producers and consumers. He is a former head of data at Convoy, a LinkedIn writer, and a published author. Chad created the first implementation of data contracts at scale during his time at Convoy, and also created the first engineering guide to deploying contracts in streaming, batch, and even oriented environments. He lives in Seattle, Washington, and operates the Data Quality Camp Slack group and the "Data Products" newsletter, both of which focus on data contracts and their technical implementation.

Mark Freeman is a community health advocate turned data engineer interested in the intersection of social impact, business, and technology. His life's mission is to improve the well-being of as many people as possible through data. Mark received his M.S. from the Stanford School of Medicine and is also certified in entrepreneurship and innovation from the Stanford Graduate School of Business. In addition, Mark has worked within numerous startups where he has put machine learning models into production, integrated data analytics into products, and led migrations to improve data infrastructure.