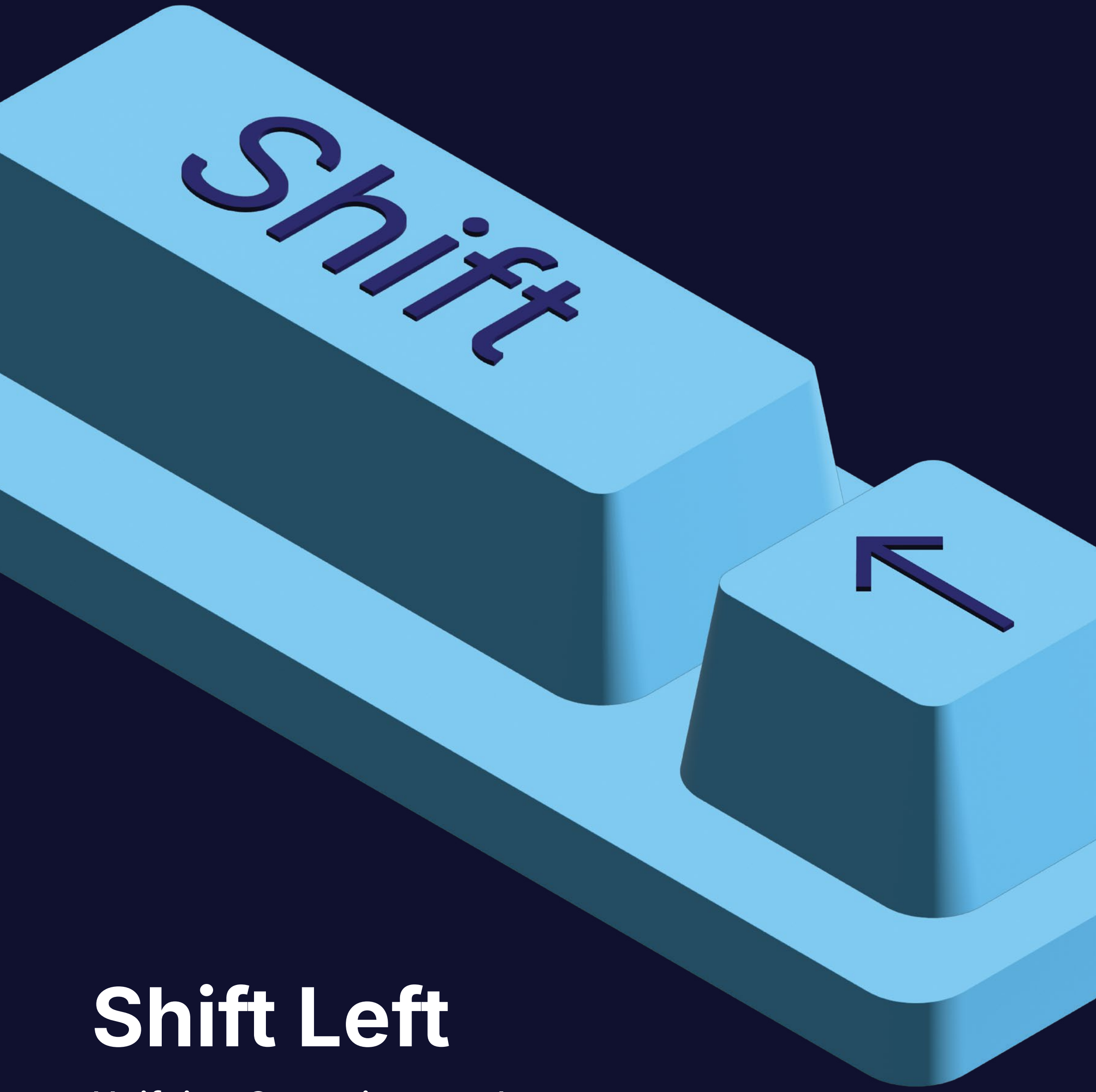
CONFLUENT



Shift Left

Unifying Operations and Analytics With Data Products

Adam Bellemare

Table of Contents

Foreword

It should be no secret that I've been heavily invested in data streaming since my days at LinkedIn as one of the original co-creators of Apache Kafka® many years ago. I've been fortunate enough to meet many great people, see the wonderful things that they've built with Kafka, and learn what has made them succeed, as well as the problems and pain points they've experienced.

There's a common trend that separates the companies that find success with their data strategy from those that don't. They invest in making their data easy to publish, find, and share all across their organization—for operations, analytics, back office, and wherever else it's needed.

Historically, there's been a major divide between the operational side and the analytical side of the business, as each team ends up working on its own to get, clean, structure, and use data. While simple to get started, this strategy causes significant hardship in the long run for the business, its customers, and the developers and support staff in between.

In this book, Adam explores exactly what these problems are, why they exist, and how you may find yourself approaching the same challenges today. He does a great job of explaining how shifting left—an incremental selection and shifting of the important data sets that underpin your business—works in practice so that everyone can benefit from clean, simple, and reliable data access.

I think that this practical guide to shifting left will be valuable to architects, developers, and data engineers, as Adam covers the relevant considerations for all of the members of the team that wrangle data to find insights and power operations.

The first part explores the historical divide between the operational and the analytical plane, as well as the challenges that existing solutions fail to solve. In the second part, Adam explains what we can do about it today—focusing on the same measures that we've seen come from the most successful companies: making data easy to publish, find, share, and use. In other words, Confluent makes it so you can plug your data wherever you need it, as you need it. We take the work that you're doing downstream, far to the right, and shift it upstream or to the left so that you can reuse the results wherever you need them.

This is a very timely book, as we enter a new and exciting era of mature cloud computing, cheap storage, and cheap processing. New technologies, such as Apache Iceberg®, are rising to take advantage of these trends. Iceberg and related open table formats make it easy to decouple table storage, management, and queries with the same ease that we have streams with Kafka today. We have the opportunity to rethink and reinvent the ways we produce and use data within our organizations. I think you'll find this to be a powerful envisioning of the future and a handy reference to help guide you on your way.

–Jay Kreps, CEO of Confluent Inc.

Introduction

Data is the lifeblood of a company. Every single bit stored across your organization represents something about your business—be it an order, a payment, a customer, a shipment, or an interaction. Data is first and foremost stored in the systems that created it, in a file on disk somewhere, purpose-built and modeled to serve the immediate business use case. These transactional systems are part of the *operational plane* (populated by online transaction processing systems or OLTP) and are extremely efficient at serving their specific business use cases.

But business requirements expand far beyond handling transactions. Are we making money? Is our product strategy succeeding? Where should we focus our efforts in the next quarter and the next year? Answering these questions requires collecting data from across the organization and *analyzing it in data warehouses, data lakes, or data lakehouses*. The *analytics plane* (online analytical processing systems or OLAP) is primarily concerned with evaluating business success, providing actionable insights, and the multipurpose analysis of historical data.

The reality is that most businesses have a rift between the operational plane and the analytical plane. Access to operational data is necessary for computing analytics, but the data practitioners in the analytics plane are left to create their own means of access.

Data pipelines, such as the extract-transform-load (ETL) and the extract-load-transform (ELT) processes have long been the primary means for getting data into the analytics plane. Analytics engineers reach into the OLTP database (extract), pull out and model the data they need (transform), and then push it to the destination data lake or warehouse (load). ETL pipelines are traditionally brittle, expensive, and require chronic maintenance and break-fix work, but they remain in common use nonetheless due to the significant need to access data outside the source.

But the pipeline's owners in the analytics domain have little to no control or involvement over the source data model, which is under the ownership of the application developers of the operational domain. Evolution and changes in the application space cause chronic breakages in the data pipelines, resulting in significant break-fix work, delayed results, bad data, and unhappy customers. The root of this problem is a combination of social responsibilities, technological limitations, and historical conventions based on legacy tooling.

To make matters worse, the analytics plane is not the only plane that needs access to high quality business data. In this era of decoupled cloud computing, microservices, managed SQL engines, and software as a service (SaaS) applications, the need for high quality business data is greater than ever all across your business—including the operational layer.

Operational systems often require access to the exact same data as their analytical systems but are unable to leverage the same ETL/ELT data pipelines to get it.

Emerging technologies such as generative AI (GenAI) have driven a surge in demand for data access across the business. From tuning models to providing query context, easy and reliable access to well-formatted data makes the difference between great and lackluster results.

Simply put, operational and analytical use cases all face a common problem: **The inability to reliably access relevant, complete, and trustworthy data.** Instead, they're left to their own devices to cobble together their own means, periodically querying APIs, databases, and SaaS endpoints.

ETL pipelines may provide a partial solution for data access for data analytics use cases, while a REST API may serve some ad hoc data access requests for operational use cases. But each independent solution requires its own implementation and maintenance, resulting in duplicate work, excessive costs, and similar-yet-slightly-different data sets.

There is a better way to make data available to the people and systems who need it, regardless of if they're operational, analytical, or somewhere in between. It involves rethinking those archaic yet still commonly used ETL patterns, the expensive and slow multi-hop data processing architectures, the "everyone for themselves" mentality prevalent in data access responsibilities. It's not only a shift in thinking but also a shift in *where* we do our data processing, *who* can use it, and *how* to implement it. In short, it's a *shift left*.

This book covers how to make this shift left a reality. We'll look at the problems with the existing methodologies, how they fall short, and how their failures lead to costly mistakes. We'll take a detailed look at *shifting left*, the role of data primitives such as streams, tables, and schemas, and best practices for success. And finally, we'll look at some examples that illustrate how to apply it to your own space so that you can get on with using your data instead of just struggling with it.

Part I

How Shifting Left Solves the Analytical-Operational Divide

The crux of shifting left is relatively simple. Take the work that you're already doing to extract, transform, and remodel data out of your downstream systems (such as your analytics plane) and *shift it left*. Shift it upstream as close to the source as possible, such that all of your downstream consumers can share a single source of well-defined and well-formatted data. Provide *consistent*, *reliable*, and *timely* access to important business data, treating it as a first-class building block for powering services, analytics, and AI capabilities.

Shifting left reduces complexity and costs by eliminating duplicate data pipelines, processing, and data storage. It removes the need for parallel maintenance, eliminates data pipeline break-fix work, and frees your engineers to work on more valuable tasks.

You and your colleagues can reuse the data across your business because you've front-loaded the work to clean, standardize, and structure it out of the gate, instead of downstream in a siloed location.

In some cases, this is as simple as lifting SQL cleanup code that you're running in your data lake and shifting it left towards the source system that originally created the data. For other use cases, it may take more work, both in identifying the most valuable data and in implementing the shift left. But the idea remains the same—move the data cleaning, structuring, and data transformation as close to the data creation as possible so that all may benefit from the result.

Shifting left is not an "all-or-nothing" proposition, and not everything needs to be shifted to the left. There are many small iterative steps you can take to make it a reality, reaping value along the way. We'll cover the techniques and strategies to identify and shift your data left in Part II of this book, as well as key data concepts like the stream-table duality, data products, data contracts, and the prevention and handling of bad data.

We'll also take a look at the headless data architecture (HDA), a modular and composable decoupling of data storage (in any form) from the services that read and write it. HDA is included as part of the shift-left strategy as it's the outcome of shifting data processing and governance to the left. It relies on modern cloud primitives such as cheap storage, ultra-fast networking, and cloud-to-cloud partnerships that make it easy to create data in one cloud ecosystem yet share it and use it in another. Ultimately, adopting a headless data architecture lets you benefit from your shift-left work by enabling you to plug your data into any system, anywhere, be it operational, analytical, or anything in between.

Accessing data *that you don't own* is hard.

The age-old problem of accessing data

Once upon a time, you'd likely have stored all of your business's data in a single monolithic database. If you needed to access any data you just queried the database directly. It wasn't costly as the data wasn't very large, so there really wasn't anything you couldn't do. Data access is easy if it's all located in just one fully consistent location.

But fast forward to today: it's highly likely that you can no longer fit everything in just one single database. After all, your company has grown; your tech stack has changed and migrated; and the quantity, granularity, and availability of data has multiplied. While you may still be able to meet most (or some) of your OLTP needs with your monolith, you're going to need an entirely different set of tools to provide OLAP solutions—not every problem is a nail, and not every solution is a hammer.

Perhaps you're using Javascript and MySQL or Go and Postgres for low-latency OLTP services. Meanwhile, your OLAP use cases rely on Java, Apache Flink®, and S3 storage. The tools that you use to solve one set of problems are unlikely to be the same tools that you use to solve another problem. And therein lies a problem—how do you get the data you need to do your work?

The analytical-operational divide

The analytical-operational divide presents the most common data-sharing challenge facing most companies.

On one side, you have the operational plane, characterized by applications and services optimized for low-latency transactions. In many businesses, the operational plane is where the vast majority of data is created and initially stored for other operational systems to query and update. The problem for analytical use cases is that the data models and database technologies are optimized for OLTP but not for OLAP, which typically requires complex querying and characterizations of large quantities of data.

On the other side, you have the needs of the data analysts, scientists, and engineers. The querying of large sets of data is the norm, including historical data and data from disparate systems across the organization. Large, high-latency queries feature regularly but rely on optimization techniques innovated particularly for analytics.

One good example of an analytics optimization is the columnar data model (such as Apache Parquet™), in which data is stored and accessed in columns instead of rows. Analytical queries that aggregate on a column value—for example, a sum of all values in the "paid_amount" column—can complete very quickly by just loading that single column of data.

In contrast, an OLTP system typically uses a row-based model in which updates to a specific row are saved on a per-row basis. This approach is much faster for random read/write access typical of operational workloads but less efficient for analytical use cases.

The analytical-operational divide exists not only due to technology requirements but also due to social factors. The historical data analytics approach has been one of isolation, with little to no cooperation between the operational source system owners and the data analysts, scientists, and engineers. Why is this?

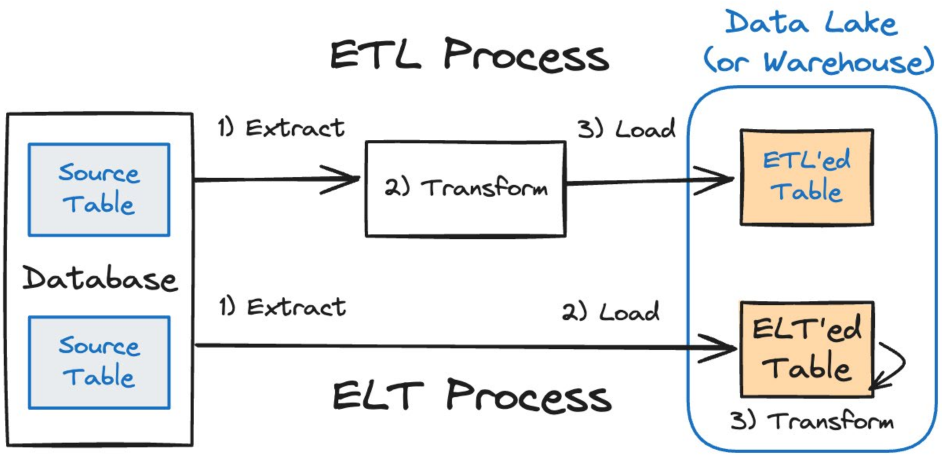
The data teams are usually siloed off in their own part of the company—with their own leadership, their own business objectives, and their own cost centers. Similarly, the operational teams have their own leadership, objectives, and cost centers as well, which can make inter-team cooperation a bit clumsy and slow. Unfortunately, Conway's Law strikes again. Essentially, the goals and incentives of the operational space traditionally do not align with the goals and incentives of the analytical space, resulting in diverging data and technology dependencies.

Thus, a booming ETL/ELT industry emerged to accommodate the isolation of the data practitioner. The promise: build and manage your own pipelines to copy data from the source systems into the analytics space. How does this hold up in practice? Let's take a look.

ETLs and ELTs: Getting data from the operational plane to the analytical plane

The most data-intensive use cases require special tools that sit outside of the realm of the operational systems that generate a significant majority of a business's data. Because these purpose-built data-intensive tools sit so far away, we've historically needed a way to plug the operational data into the system, so we used (and still use) ETLs and ELTs. But the further from the source you move, copy, and transform data, the greater the frequency of breakages, the higher the latency, the more complex the dependencies, and the greater chance that the humans in the loop will make some well-meaning but ill-fated decisions on how to clean, transform, and process the data.

The main goal of setting up an ETL (and ELTs—please assume I'm talking about both unless otherwise noted) pipeline is to get data from your source system into the analytical plane so you can do analytics work on it. In the case of an ETL, you may do some initial transformation between the **E**xtraction and **L**oad steps. In the case of ELT, you'll pull the data into the analytics plane exactly as it looks in the source and then perform the initial **T**ransform steps once it lands.



How ETL and ELT processes generate a new tables of data from the source

Once in the analytics plane, the data practitioners are traditionally charged with reshaping the data, adding schemas, and bringing some semblance of order to it. The medallion architecture is a common and widely used model for organizing this work, which we'll look at in more detail in the next section.

ETL pipelines typically rely on one of several common methods for connecting to the source systems:

- **Direct access to internal domain model:**
  The ETL process is given credentials to access the internal domain model of the database, coupling tightly to the model in disregard for data encapsulation. It issues periodic queries on the internal domain model's tables, selecting either all the data that has changed since the last query (incremental) or everything in the table. The ETL remains directly coupled to the internal domain model of the source, such that any changes made to it often result in a broken pipeline. Note that the source database must accommodate this query and load.

- **Direct access via a read-only replica:**
  Same as above but via a read-only replica. The ETL no longer impacts the source database's performance. However, the ETL remains directly coupled to the underlying data model.

- **Access via the database change log (change data capture):**
  Many databases use a write-ahead log or a binary log to append

changes made to both the data and the table definitions. A long-running process continually tails the log and creates events that reflect everything that has occurred in the database. The changelog is tightly coupled to the internal database model, and so are the resulting events.

- **Access via a request-response API:**
  The database internal model remains isolated from the ETL, which must periodically poll the API to obtain new data. This mode is more common in SaaS offerings where the internal data models are deliberately isolated from the user.

There's a common thread between each of these strategies: each must contend with the abstraction between the internal data model of the source system and what is exposed externally to the rest of the world. Directly accessing the data via a query on the internal data model results in a very high level of coupling. Querying a view provides lower coupling, but you must maintain the view and the mapping to the internal data model.

Change data capture (CDC) provides a replica of the internal data model via the database change log. You can choose to filter, transform, and drop columns, but much like a view, you'll also need to maintain the processing logic. We'll revisit the role of both views and CDC later in Part II when we look at how to build reusable data products.

The REST API is the only option in this short list that forces the application developer to make a conscious decision as to what data they are willing to expose *outside* of the application versus that which they choose to conceal inside. They may also choose to simply expose their entire internal data model via GET and PUT operations, providing no guarantees against breaking changes made to the internal model. These wide-open APIs may then be used as part of an ETL pipeline to get data into the analytical domain.

We use ETLs to get data into the analytical domain so that our data practitioners can put it to work to do useful things. I'll skip a full treatment of the detailed personas of the data practitioners of the analytical space and instead split it into two simplified major roles:

- **Data engineers responsible for getting data:** They create ETLs ("pipelines") from external systems to pipe data into the analytics plane. Once in the analytics plane, they are also responsible for cleaning it up, standardizing it, applying schemas, and denormalization.

- **Data analysts, data scientists, and ML and AI practitioners:** They require reliable base data sets to compose reports, analyses, machine learning (ML) models, artificial intelligence (AI) agents, and other data-intensive projects. They may also be responsible for some degree of cleanup, transformation, and remodeling, but they typically work more on using the data than on acquiring it from source systems.

This model is of course quite simplified, and there is no rule saying that one may not do any of the tasks in the other. But the gist is there are people responsible for pulling data into the analytics plane from the "outside" via ETLs—ETLs for which they must maintain, debug, evolve, monitor, and provide on-call "wake up in the middle of the night" services.

Change is constant. If we didn't have change, our ETLs would never break. We wouldn't have to worry about changing business objectives, changing product lines, or financial changes (like mergers or layoffs). People wouldn't quit, and we wouldn't have orphaned and abandoned ETLs. Data models wouldn't change, tables wouldn't be refactored, and tech stacks would never need to evolve. But since we do have change, then it's probably not too hard for you to see why an ETL is simply not a "set it and forget it" sort of thing.

ETLs (and ELTs) are a natural progression of the adoption of OLAP services, machine learning, and GenAI. But the ETLs that power them remain a major source of chronic pain, even after decades of technology changes and revision. Why? What happened?

Let's defer to Ralph Kimball of the Kimball and Ross methodology, an expert in the domain of data modeling and data warehousing (emphasis mine):

> *"When asked about the best way to design and build the ETL system, many designers say, 'Well, that depends.' It depends on the source; it depends on limitations of the data; it depends on the scripting languages and ETL tools available; it depends on the staff's skills; and it depends on the BI tools. But the 'it depends' response is dangerous because it becomes an excuse to take* ***an unstructured approach to developing***

an ETL system, which in the worse-case scenario results in an undifferentiated spaghetti-mess of tables, modules, processes, scripts, triggers, alerts, and job schedules. *This 'creative' design approach should not be tolerated. With the wisdom of hindsight from thousands of successful data warehouses, a set of ETL best practices have emerged. There is no reason to tolerate an unstructured approach."*[1]

— Ralph Kimball, one of the original architects of data warehousing

To paraphrase Kimball, you need rigor, best practices, and a structured approach to avoid building your ETLs incorrectly. While I do not doubt that Kimball knows ETLs, the beauty of our shift-left approach is that we do away with ETLs *entirely*. Instead, we rely on a no-copy approach to provide the same data from the OLTP side as a pluggable read-only table suitable for OLAP needs.

But we're getting ahead of ourselves. Before we can see why this pluggable table approach works so well, we'll need a better idea of just what these ETL jobs are powering. Let's take a look at the medallion architecture, one of the most popular ways to structure a data lake or data warehouse today.

Analytics with data lakes and data warehouses

There are several leading architectural and conceptual models in the data space today, but for simplicity we'll just touch briefly on data lakes and data warehouses.

Data lakes store all types of raw data, which is then transformed step-by-step into a usable data product for a specific business use case. The result is that you have multiple tiers of data, each of different quality and reliability, that you can process and use for your own results.

Data warehouses have historically been where you would load all of your cleaned and structured data for analytical purposes—slicing and dicing data to answer queries. Star and snowflake schemas have been

1 Kimball, R. (2002). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*. John Wiley & Sons Inc.

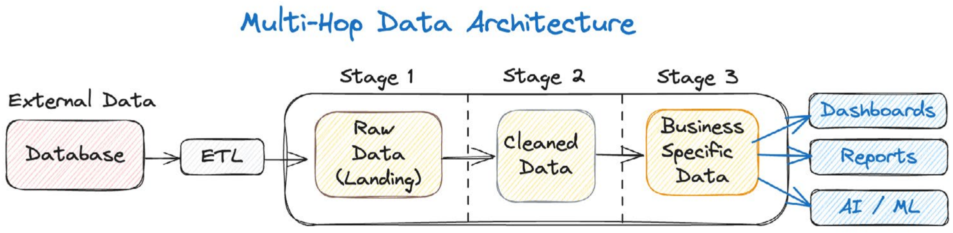
historically popular data modeling choices for query efficiency.

The thing is that today's data architectures have pretty blurry lines between data lakes and data warehouses. You can serve performant star and snowflake schema queries from a data lake. Conversely, you can also build data processing pipelines that use the data warehouse to store raw, intermediate, and business specific data sets.

For our purposes, we're going to focus on the **multi-hop data architecture** as the common basis for the data analytics plane and how it affects the tasks and duties of data practitioners.

Multi-hop data architectures

The term multi-hop data architecture is used for architectures where data is processed and copied multiple times before eventually arriving at some level of quality and organization that can power a specific business use case. Data flows from left to right in the diagram below, beginning with some form of ETL from the source system into the data plane.



Data flowing through the stages of a multi-hop architecture

This diagram is a 3-stage architecture, commonly found in most "how to" literature on data lakes and data warehouses.

- **Stage 1–Landing or staging zone:** Put raw, unstructured, and semi-structured data into a staging location, usually within the boundaries of the data lake or warehouse. This data typically comes from an external system as part of an ETL, a direct file copy, or other ingestion mechanism. Stage 1 data is not considered reliable or trustworthy yet.
- **Stage 2–Cleaned basic data sets:** Apply filtering, remodeling, and data quality enforcement to convert the data from stage 1 into stage 2.

Stage 2 data is considered trustworthy and reliable and is suitable for composing the more advanced models in stage 3.

- **Stage 3–Curated business data sets:** Consists of business specific data sets built from data sourced from stage 2. Reports (e.g., sales/day), dashboards, AI/ML models (e.g., user interests), and more complex data models are the provenance of stage 3.

Though multi-hop architectures have been around for decades as part of bridging the operational-analytical divide, they are inherently inefficient. While they do provide the means to move data from the operational side to the analytical side, they remain slow, expensive, and difficult to reuse.

The medallion architecture nomenclature has become a popular substitute for the generic "stage 1-2-or-3" terminology I used to introduce this section. **We're going to use medallion architecture terminology to describe multi-hop architectures for the rest of this book**, since it is fairly well-defined, is widely used, and can map to data lakes or data warehouses regardless of the actual technology in use. We'll also take a deeper look at the inherent problems behind multi-hop and its limitations through the lens of medallion.

Let's get started.

The medallion architecture

The medallion architecture is a design pattern used by data practitioners to organize and delineate data sets. It is divided into three different medallion classifications or layers, according to the Olympic Medal standard: **bronze, silver,** and **gold**. Each of the three layers represents progressively higher quality, reliability, and guarantees—with bronze being the weakest and gold being the strongest.

The medallion architecture is a tidy formalization of multi-hop or stage-gate architectures, which have existed for decades under various names and forms. Medallion is the most recent and popular adaptation and for good reason: it makes a lot of sense from the perspective of a data practitioner that receives little to no help from the teams that create and maintain the source data models.

The ultimate goal of adopting a medallion architecture is the unification and standardization of importing, cleaning, modeling, aggregating, and serving data for analytical purposes. Let’s look at the primary use cases and objectives of each layer:

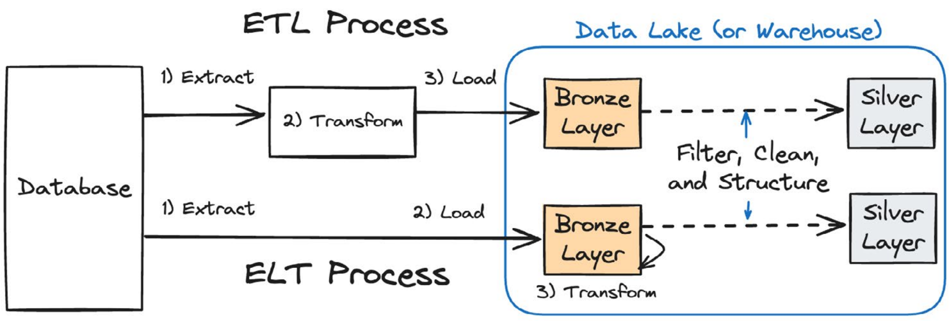
Bronze layer:

The bronze layer is the landing zone for raw imported data sourced from OLTP systems, SaaS endpoints, Kafka topics, CSVs, XMLs, or even plain text files to name just a few. The data commonly arrives as a mirror of the source data model but may also undergo minor transformations as part of the ETL process. Data practitioners then add structure, schemas, enrichment, and filtering to the raw data. The bronze layer is the primary data source for the higher-quality silver layer data sets.

Silver layer:

The silver layer stores filtered, cleaned, structured, standardized, and (re)modeled data, processed to a minimally acceptable level of quality. The silver layer provides well-defined data suitable for analytics, reporting, and further advanced computations. Silver layer data represents important business entities, models, and transactions. For example, it may contain data sets representing all business customers, sales, receivables, inventory, and customer interactions, each well-formed, schematized, deduplicated, and verified as “trustworthy canonical data.” The processing patterns, standardization rules, degrees of normalization, and file write/read optimizations will vary from use case to use case, and organization to organization.

The silver layer provides the building blocks for calculating analytics, building reports, and populating dashboards. It enables ad hoc querying and exploration and allows for data practitioners to build the richer data sets found in the gold layer.

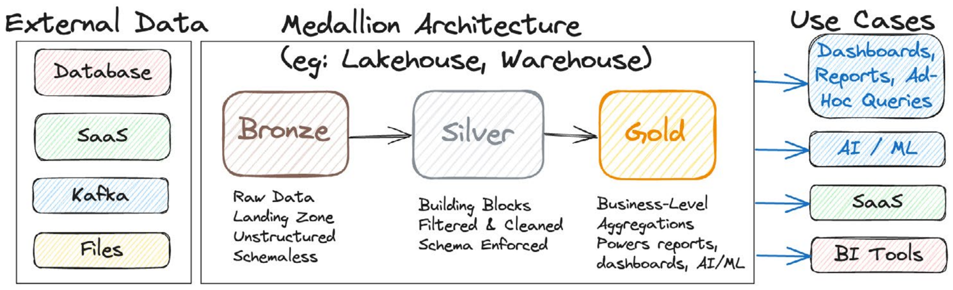


How data retrieved via ETL or ELT processes reach the silver layer in the medallion architecture

Gold layer:

This layer delivers "business-level" and "application-aligned" data sets, purpose-built to provide data for specific applications, use cases, projects, reports, or data export. The gold layer data is predominantly de-normalized and optimized for reads, but you may still need to join against other data sets at query time. While the silver layer provides "building blocks", the gold layer provides the "building," built from the blocks and cemented together with additional logic.

Here's an overview of what the medallion architecture looks like from end to end. Each step in the architecture, as you go from external to bronze to silver to gold necessitates processing and data copying. In some cases, exporting the data to a SaaS endpoint or a business intelligence (BI) tool may require copying the data yet again. In other cases, you may be able to plug your external tool into the data without copying.



The medallion architecture, a popular version of the multi-hop architecture

The downside of the medallion architecture is that all of the work you do to get the data into a clean, reliable, and well-formatted state (the silver layer) is locked away within the data lake or the data warehouse. The services, teams, and people in the operational space are unable to use this high-quality data for their own use cases, particularly if they care about getting it in a timely manner.

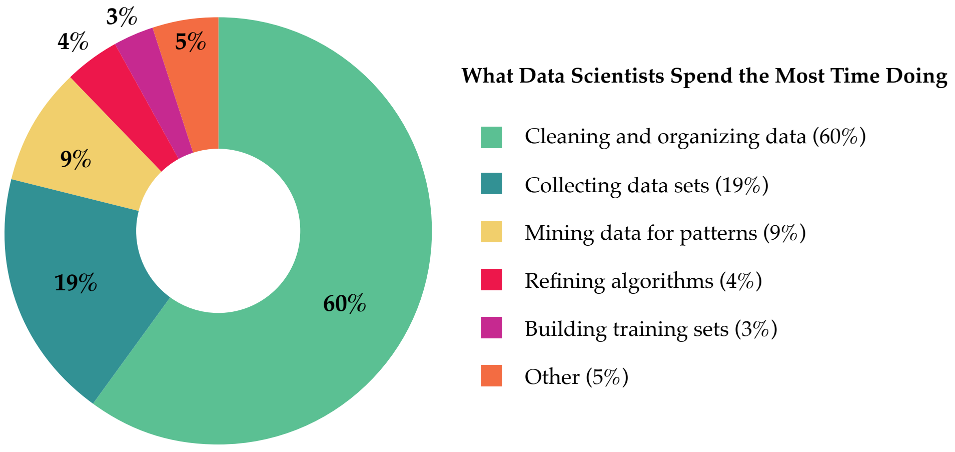
Shifting this silver data to the left (i.e., out of the data lake or data warehouse) means we can publish it as a first-class data product and provide it for all consumers, operational, analytic, or in between.

The medallion architecture was an attempt to provide some sanity for the mess that is the ETL/ELT. Isolate the mess in the bronze layer and hope that it's all sorted out by the time it gets to the silver layer. But the reality is that ETLs and the bronze layer have proven to be quite a mess, despite all of our best intentions.

5 flaws accelerating the end of the bronze age

People and systems outside of the source domain often need (read-only) access to the data. As we've discussed earlier, data engineers get the data from across the business and funnel it into the analytics space. Then, they, or other data practitioners (e.g., analysts, scientists, etc.) further clean it up and put it into use. But how much work are these data practitioners putting into getting their data ready to use?

It's challenging to nail down the precise portion of a data practitioner's work that goes into extracting, loading, and cleaning up the data for further use. But we have some insights and measurements from a few research organizations.



Insights on practitioners' work from CrowdFlower's 2016 Data Science Report[2]

While this Crowdflower report is from 2016, I chose it because it illustrates time-percentages that we still see today. Empirically, we regularly see this report popping up time and time again, with data practitioners pointing to it and saying, "Yes! It's true!" We have seen ranges as low as 40% and as high as 80% spent on data cleaning and organizing. Each organization is different, and they may spend more or less time acquiring, cleaning, and organizing their data.

A more recent article from 2024 indicates that this problem has not yet been solved (emphasis mine):

[2] CrowdFlower. (2016). Data Science Report.

"Operations consume nearly 20% of global banks' $1.4 trillion cumulative budgets and, according to Broadridge, ***30-40% of back office costs are spent on reconciliation processes****. That means financial institutions are* ***spending tens of billions of dollars per year cleaning up broken data****. While efforts have been made to improve reconciliation with optimized processes or technology, relatively little has been achieved to ensure data is correct at the outset."*[3]

— Greg Schvey, CEO of Axoni

Schvey goes on to include that "the reconciliation costs described above do not include the downstream costs incurred by the industry as a result of that incorrect, late, missing, or otherwise broken data." Indeed, "one of the largest global banks was recently fined $400M by the Office of the Comptroller of the Currency for operational deficiencies, with a dozen references to data quality in the Consent Order, including an explicit requirement to 'strengthen procedures and processes for the continuous improvement of data quality.'"

The unfortunate part is that the bronze-layer model is deeply flawed, incurs significant recurring costs, and puts your organization at significant risk. Data is not reusable by others outside of the data lake or data warehouse without yet another ETL (a "reverse ETL") to pipe the data out, and operational use cases can seldom tolerate the high latencies associated with it. Operational application developers are effectively left to their own devices to build their own means of accessing data.

Flaw #1: The consumer is responsible for data access

The medallion architecture buys into the idea that downstream data users are solely responsible for accessing, acquiring, and storing the data in the data lake or data warehouse for future use. This is a reactive and untenable position, as the consumer bears all the responsibility of keeping the data available and running, without having ownership or even influence of the source model.

[3] Schvey, G. (2024)."Wall Street's Multi-Billion Dollar Bad Data Problem." Tabb FORUM.

What are the main problems that a data practitioner faces when they build an ETL to pipe data into their data lake?

- **The ETL is coupled to the source data model.**
  The internal private model of the source system is exposed in its entirety to the downstream consumers. While you can certainly choose to exclude some fields from ETL, the decision of what to include and exclude remains the choice of the consumer. This leads to inappropriate coupling on the source model.

- **ETLs are brittle due to the strong source coupling.**
  Any changes to the source model, such as those from a refactoring or expansion into new business opportunities, can and often does cause the ETL to break. There are seldom guarantees of schema or structure. Data models and formats are subject to change whenever the upstream source system changes. It is common for unexpected data changes from the source to show up in the bronze layer, breaking dependent data pipelines and processes.

- **ETLs rely on the source's compute resources.**
  Querying the source database to extract data uses up resources, causing a degradation in production performance. While adding read-only replicas can alleviate resource contention, it doesn't eliminate resource contention between ETL jobs (e.g., 50 ETL jobs all kicking off at midnight to get the daily data).

- **ETLs are high latency and not suitable for all workloads.**
  While some analytical jobs may be fine with waiting for an hour for data to sync, others may need sub-minute or even sub-second latency.

In short, the ETL (or ELT) pipelines act as an abstraction layer between the source database model (data on the inside) and the outside world where other teams, people, and systems may couple on it. But the reality is that bronze layer architecture ETLs are brittle, fragile, and reactive, and the roles and responsibilities of those involved do not align for a healthy data ecosystem.

Flaw #2: The medallion architecture is expensive

The second major flaw is that the medallion architecture is expensive.

- **Populating a bronze layer requires lots of copying and processing power.**
  The bronze layer prescribes copying data into the data lake with minimal transformations. Once in the lake, it can then be processed (and reprocessed) to a sufficient level of quality. Each step incurs costs—loading data, network transfers, writing data, and computing resources—and these costs add up into a large bill.

- **Everyone just builds their own pipelines.**
  The medallion architecture puts the responsibility of obtaining data on the data consumer. A consequence is that the consumer will tend to focus on building their own pipelines for their own use cases since they remain solely responsible. Finding similar data sets for reuse can be difficult, as the ad hoc approach to accessing data tends to lead to silos and fragmented discovery.

  Sharing is uncommon because people have been burned by bad data sets and zombie pipelines before.

  - *"Build your own pipe from the source. I don't want to be responsible for your SLAs."*
  - *"If I build my own pipe, I know someone else won't come along to screw it up with changes and edits."*

- **Outages and reprocessing are expensive.**
  When an ETL fails, every single downstream job must pause until the ETL is fixed. Any in-process jobs may need to be rerun once the new data is processed. This can lead to surge pricing, for example, when an extensive nightly data job fails and must be reprocessed during the day when compute costs are three times higher.

- **The bronze model also incurs significant opportunity costs.**
  Break-fix work is also dreadfully boring. Humans will leave jobs they would otherwise stay at if all they do is fix the same broken pipelines over and over. Similarly, all the time spent fixing the pipelines are not being spent on other more interesting things—like integrating your data into ML models, GenAI, or rich analytical tools.

The medallion architecture is expensive because it is defensive as the responsibilities for making data accessible lay with the wrong people. Instead of relying on the consumer to craft clever pipelines and react quickly to inevitable breakages, involve those who actually created the data—the operational system owners. We'll cover this in more detail in Part II.

Flaw #3: Restoring data quality is difficult

So you ETL/ELT the data out of your system into your data lake. Now you need to denormalize it, restructure it, standardize it, remodel it, and make sense out of it—without making any mistakes.

A formerly popular TV show in the United States, entitled Crime Scene Investigators (CSI) gave their audiences an unrealistic sense of restoring structure and form to data. In one scene, one investigator tells another to "enhance" the image on the computer screen—the other investigator zooms in (significantly), and the blurry pixelated image suddenly reforms into a sharp rendition of a reflection of the suspect's sunglasses. In reality, we would just get a close-up of a handful of very blocky pixels that no amount of "enhancing" would fix.

This is a cautionary tale. The bronze layer creators are not experts on the source domain model, yet they are charged with recreating a precise, picture-perfect rendition of the data. Any of the work that they perform in pursuit of this goal is challenging, and though they may come close to recreating what the source data model represents, it may not be precisely correct. And this is where the next problem comes in.

Say we're back in the CSI world, but now there are three computers total, each with their own investigator or agent. Each agent "enhances" their copy of the data, and each agent gets a similar-yet-different result. Which one is correct? Whose lead is the one that they should pursue next? Is it license plate BFEZ838? Or is it BEFZ833? BEEZ888?

There is no substitution from getting the correct high-fidelity data from the source, as tack-sharp and optimally constructed as possible. And if you do it correctly, as we'll see in Part II, you'll be able to reuse it for all your use cases—operational, analytical, and everything in between.

Flaw #4: The bronze layer is fragile

The bronze layer is supposed to be the basis of your entire medallion architecture. It forms the base of the pyramid, with each of silver and gold layers plugging into the layer beneath it. The silver layer, where good quality reusable data building blocks are to be found, relies on a shaky chain of processing dependencies and a poorly-aligned responsibility model. But the impact can be found the most in the gold layer, the data sets that your business customers access to view their important business stats.

Trust is the utmost in business. All the data orgs I've worked in had some variation of a saying that boiled down to this: "Trust is easy to lose and hard to regain."

Bad data will make your customers lose trust in you. If you show them one thing on the dashboard but charge them another thing in the bill, they won't trust you. If they find confusing data that doesn't match other data sets (similar-yet-different, remember?), they won't trust you. If they receive email after email notifying them of reporting delays, or they can't see when their products have shipped, they won't trust you. It is easy to lose trust; it is hard to regain it.

The fortunate part about the bronze layer fragility is that it's very easy to fix. Instead of putting the onus of the consumers and would-be users of the data to access it, clean it up, standardize it, and make it available, you just shift it to the left. You do the same thing you're already doing in your data lakes, warehouses, and operational endpoints to copy, reconstruct, and remodel data, and instead you front-load it into your processes. Then, you share *that* data as a first-class citizen for all to use as they see fit.

Flaw #5: No data reusability for operational workloads

Data pushed to an analytics endpoint largely remains only in the analytical space, as does any cleanup, standardization, schematization, and transformation work. It is typically processed by periodic batch jobs, which prove to be too slow to handle operational use cases. (Not to mention that the ETLs themselves are also periodic batch-based jobs.)

Instead, operational workloads develop their own techniques and tools to access the data they need, further enlarging the divide between the operational and analytical space.

A trend toward something better

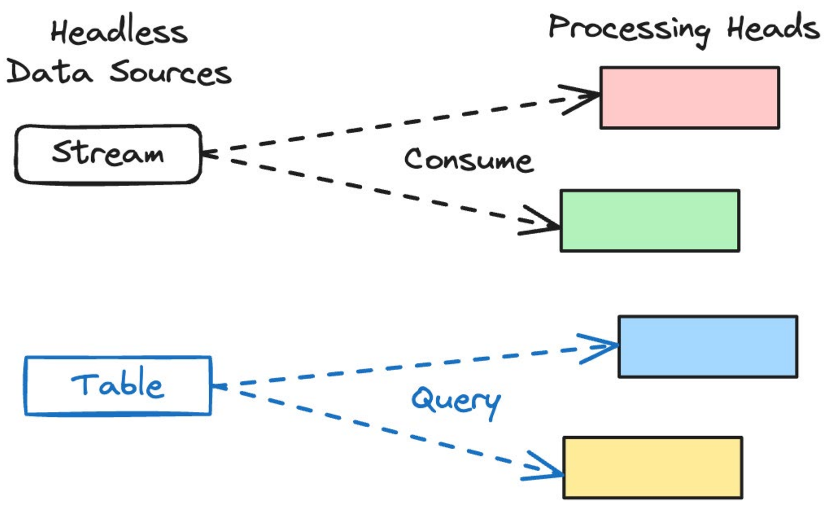
One emerging trend that we're seeing as part of shifting left is the unification of streams and tables under a common umbrella for organization, discovery, management, and access controls. But it also goes a step further into a complete decoupling of storage from the applications, jobs, and processes that use it. While it may go by many names, we're hearing it called **the headless data architecture** by both customers and thought leaders alike—and it forms the perfect complement to a shift-left strategy.

Let's take a look.

The headless data architecture

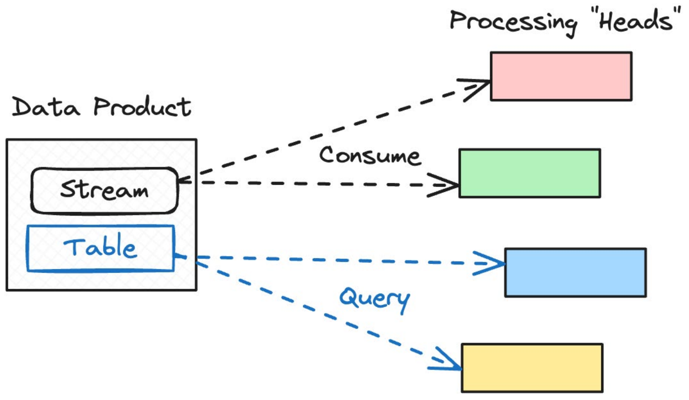
The headless data architecture is a rethink of how systems create, use, and share data. It relies on modern cloud primitives such as cheap storage, ultra-fast networking, and cloud-to-cloud partnerships that make it easy to create data in one cloud ecosystem yet share it and use it in another.

The headless data architecture is the formal separation of storage, management, and access of data from the services that write, process, and query it. It's called a *headless* data architecture because of its similarity to a headless server, where you have to use your own monitor and keyboard to log into and access the server. We can effectively write the data once to the headless data architecture, then plug in access to the underlying data to any head we choose.



Headless data, consumed and queried by independent "heads"

In a headless data architecture, data can be further organized into **data products**. A data product is a trustworthy data set purpose-built to share and reuse with other teams and services. Furthermore, you'll have the option to build data products that provide the same data as a **stream** *and* as a **table**.



A stream-table data product providing headless data

Apache Kafka® plays a significant role in a headless data architecture. Writing events to Kafka is completely decoupled from consuming events—and not just in time and space but in technology too. For example, a producer may be a Flink, Go, or Rust application. Meanwhile, you have consumers as their own independent heads, consuming the events at their own rate and written in whatever language suits their business needs.

For tables, we have Apache Iceberg®. You can plug in open source processors like Flink, Trino, Presto, Apache Spark®, and DuckDB directly into Iceberg tables. You can also plug in SaaS options like BigQuery, Redshift, Snowflake, and Databricks. This is only a short list of compatible OLAP-capable heads, and increasing Iceberg adoption will see this list only continue to grow.

Iceberg is a leading open source project that is, in effect, a file system manager for columnar data including the popular Parquet format.

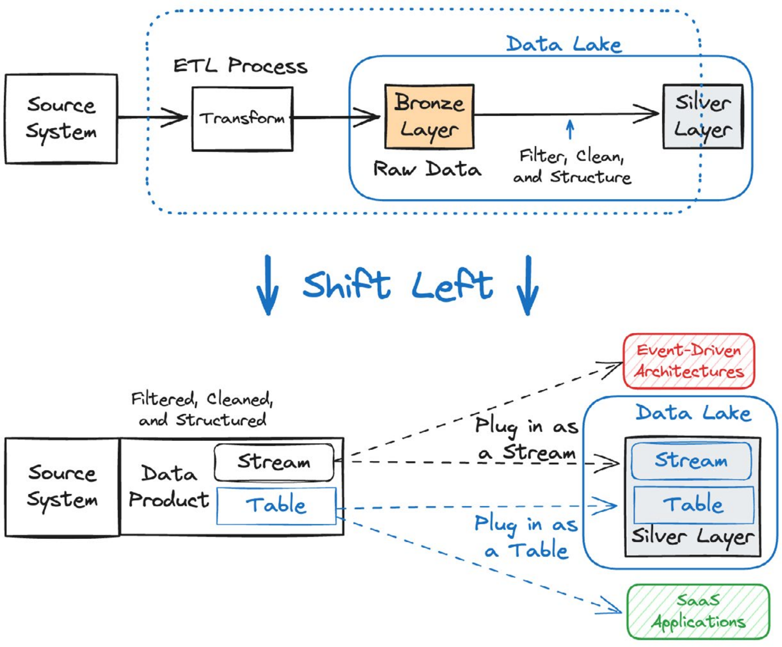
Iceberg provides several key features enabling table formats in the headless data architecture, including storage, metadata cataloging, transactional support, query time-travel, and housekeeping functions like compactions, cleanup, and optimizations. While Kafka has long since been the de facto leader of data streaming, it was only very recently that a critical mass of adoption formed behind Iceberg.

Think of a data product as a first-class data citizen, published with the rigor, control, and quality of any other product that you or your business creates. You'll be able to both publish your own data products for others to use, as well as rely on and use data products published by others.

Shifting left into a headless data architecture works because it's a simple remapping of work that you're already doing. The only difference is that you're going to do it further upstream. For example, you're already:

- Mapping internal domain models to external domains in the data lake or data warehouse
- Trying to restore fidelity to unstructured and low-fidelity data to the high-fidelity state of its source
- Building an appropriate silver-layer data set to be used as a building block for more complicated work

Instead of doing that work downstream, where only a few people may benefit from it, you do that work upstream—where multiple people, teams, and systems can benefit. You shift it to the left.



Shifting left into a headless data architecture

The first thing to note is that the bronze layer is no longer present. All of the work that goes into ETL-ing to the bronze layer, then refactoring into the silver layer is encapsulated into the data product logic. Shifting the modeling decisions, data structuring, and interface building into a data building block makes it accessible to everyone.

Second, note that you can use either a stream or a table to access the data from the data product. From our Iceberg-compatible data lake, this just means pointing your table reference to the Iceberg table provided in the data product and relying on the Iceberg standards for interoperation.

Third, note that you can also power other applications and services—be they OLTP or OLAP—from the same source of data. The beauty of the headless data architecture is that by front-loading the data preparation into a data product, we gain the freedom and mobility to plug in new heads and new data sets as our business needs change.

Furthermore, you reduce complexity and duplication in the following ways:

- No more ETLs. Just plug the data in via Iceberg or Kafka.
- No more copies. Eliminate duplication and break-fix work.
- No more similar-yet-different data sets.
- Automatically propagate updates and modifications to both streams and tables.
- Reject data errors at the source to eliminate bad data from seeping in.
- Maintain a single source of truth to power real-time applications and batch analytics.

What's the difference between a data lake and the headless data architecture?

This is a common question, particularly from people who operate in the OLAP plane.

Headless Data Architecture	Data Lake
For OLTP and OLAP use cases: Enables a common source of pluggable data building blocks. Powers applications, SaaS endpoints, **data lakes**, data warehouses, and databases as needed.	**For OLAP-centric use cases:** Analytics, reporting, ML, and GenAI. Continuous refinement, aggregation, and building of business-specific data sets.
Single layer: This architecture provides the data building blocks but does not provide additional analytical processing capabilities.	**Multi-layer:** The data lake provides tooling and processing to orchestrate complex data pipelines and multi-hop architectures.
No copies needed: Data is not copied into consumer applications but is accessed in place.	**Copies required:** Data is copied into a data lake for further processing by data lake processing jobs. Note that some data lakes also allow registry of external data sets with limited no-copy functionality.
Provides multi-modal data access, including (but not limited to) streams and tables.	**Relies predominantly on table-based data access**, including using external services like Kafka to provide full streaming support.

Figure 1. A comparison of data lakes and headless data architectures

The headless data architecture is largely *agnostic* to the work of its downstream users. It provides foundational data that is precisely aligned with the source data. It is boring. It is unexciting. But it is predictable, reliable, and completely isolates the internal domain from the external use cases. And while it provides you with the ability to access data, it doesn't force you into using a specific head to read or write to it. That is entirely up to you.

The headless data architecture is predicated on making it easy to access your data as tables or streams, however your consumers may need it. Any Kafka data stream is also fully accessible as an Iceberg table. You can plug your table or stream into any compatible head, without having to move, transfer, or copy the data.

Finally, shifting left is not an all-or-nothing strategy. You are free to continue using your existing ETLs, pipelines, and data movement systems and services. The beauty of shifting left is that you can selectively apply it to the most critical and commonly used data sets in your organization. You can try out a headless data architecture, see how it integrates into your operational and analytical services, and iteratively improve it as your business dictates. There’s no big bang shift, and everything is completely reversible if you choose to keep things as you had it.

In the rest of this book, we’ll talk about *how* to accomplish it.

Part II

Building the Headless Data Architecture

The headless data architecture makes it easy to access data wherever you need it as a table or a stream. Kafka provides the data streaming functionality, where it can then be converted directly into a table. In this section, we'll cover how to convert streams into tables, how to organize data, and all the ins and outs of building reusable data building blocks. We'll also look at a practical example to stitch it all together.

In this part, we're going to look at several major components of shifting left and the headless data architecture. These include:

- **Data products:** The fundamental reusable data building block
- **Stream-table duality:** Creating a table from the stream
- **Data streaming platform:** The six tools you need to make it all work together
- **What makes a good shift-left candidate:** Identifying the data sets that will give you the best return on investment for shifting left
- **Building data products:** Some examples of how you can create your stream-table data products
- **Evolving schemas and handling changes:** Navigating changing business and data needs over time
- **Preventing and fixing bad data:** Strategies and techniques to avoid bad data and enforce good practices

Let's take a look at the fundamental data unit of the headless data architecture, the data product.

The data product: A reusable and trustworthy data asset

You may have heard about data products before, likely as part of a data mesh. A data product is a trustworthy data set purpose-built to share and reuse with other teams and services. It's a formalization of responsibilities, technology, and processes to make getting the data you and your services need easy. Data products are a useful way to think about and structure the reusable data assets of your headless data architecture.

A **data product owner** is responsible for maintaining the code and processes that generate the data product, particularly the mapping between the internal data model and the external data product model. This mapping provides decoupling between the two models and ensures that the internal and external models can update largely independently of each other.

Anyone can be a data product owner—you don't need to hire someone specifically for the job. The owner is responsible for managing the changes and features of the data product—for example, evolving schemas, changing semantics, and deprecations. They work with the data product users or consumers and act as a point of contact for discussing changes and requests. In most cases, the data product owner is someone who is intimately familiar with the data set, understands the internal and external domain models, and is willing to act as a spokesperson and point of contact for the data's usage. It is not a full time role but rather a modest addition to usual duties.

Adopting data products requires a shift in social responsibilities. The main challenge in adoption usually comes from defenders of the status quo—the operational systems developers who often want no part in providing data access, and the data practitioners who have a familiar (albeit difficult) solution already in front of them.

The key is to come together and discuss the problems currently being faced and identify how to work together to solve them. Limiting liability is an important consideration—for example, the operational system developers are only responsible for providing a few specific data sets, and any other data access is not within their realm of responsibility (e.g., ad hoc query access).

Working together is also important. The data practitioners need to provide insights into their needs but also stay involved as an active consumer of the data product—such as validating changes made to the external models and ensuring that testing and validation is being performed. After all, the impact of bad data flows downstream!

The reality is that shifting left will fail if you don't get buy-in from upstream. In fact, this is precisely what has happened with the conventional device between operations and analytics—isolation, a lack of communication, and an acceptance of the status quo.

A successful shift left will see some social changes in your organization about who is responsible for what. But it'll also open up a window for conversations about best practices, how to watch each other's backs, and how to come up with mutually beneficial outcomes.

Once you've established a data product owner, they can begin to focus on defining the data contract. Major components include:

- **Schemas:** These formal declarations establish the data's structure, types, fields, defaults, constraints, rules, evolutionary capabilities, and embedded documentation.
- **Data quality rules:** These are formal restrictions about the content of a field. For example, you could enforce that a phone number must be 10 digits long, or that the email address be in the format of "<string>@<string>.com". Data quality rules prevent malformed data from accidentally getting into your data streams.
- **Metadata:** This component provides additional information about the data product, including business tags, classification tags, data access location, policies, and versioning information.
- **Service-level agreements (SLAs):** SLAs define the degree of service offered in supporting the data product, often tied to the same SLA scale of other applications and services in the business.

A data product is accessible via one or more modes—an endpoint that a data product user connects to to access the underlying data. Common modes include:

- **Kafka topics**, containing a stream of near real-time events. Reminder: A topic is an immutable append-only log of events. New events are created and appended to the log. Any corrections or updates to data already published are written as a new event. You cannot selectively delete or edit records in the data stream.

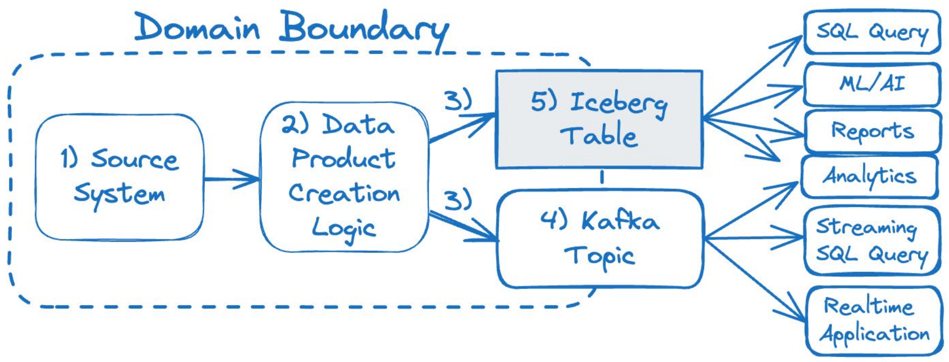
- **Parquet tables (e.g., via Apache Iceberg)**, containing a materialization of the stream or as a standalone periodically updated data set.

Less common modes include:

- **REST/gRPC API**, providing a selection of pre-fabricated data sets. However, this is usually semantically identical to just providing the table itself.

- **Alternate data models,** for example, graphs and documents. You may choose to present your data in a variety of other structures, but these rarely offer the decoupling of storage and compute necessary for a headless data architecture. However, there is no reason why a decoupled graph or decoupled document model could not be possible in the future.

For the sake of simplicity, space, and time, let's focus our efforts on just the first two—data provided by a Kafka topic and data provided by Iceberg tables. Let's take a look at what a stream-table data product looks like:

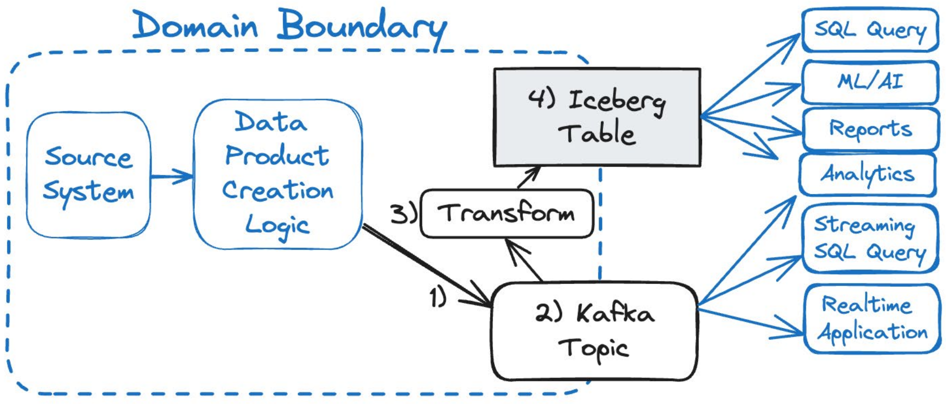


Multiple systems accessing a data product via a Kafka topic or an Iceberg table

The first element (1) is the source system in the OLTP plane. We couple the data product creation logic (2) directly to the source system. The two elements are going to live in lockstep now—changes to the source system will need to validate against the data product creation logic and vice versa. We'll explore an example of what this interface looks like and the technologies involved in the remainder of this book.

Third, we have to write the data (3) to the data products modes (4 & 5). This part can be challenging if you choose to write to each mode at the same time. Distributed transactions are hard to implement and even more difficult to do so with high performance and low latency.

Instead, we can take advantage of **stream-table duality** to generate the table from the stream itself. Instead of writing to both the stream and table at the same time, we instead write to the stream first, and then generate the table from the stream.



A streams-first data product with an Iceberg table generated from a Kafka topic

First, the data product logic (1) writes data to the Kafka topic (2). Note that it does not write any data to the table directly. Then, a purpose-built consumer reads the data from the topic, transforms it (3) into the suitable Iceberg table format, and writes it to the Iceberg table (4). Let's take a closer look at how materializing a stream into a table works in practice.

Stream-table duality and materialization

The biggest benefit of a streams-first approach to data products is that you effectively get the table representation for free. You've already done all the challenging work of providing a well-defined and contextualized data product with a contract, schema, evolution rules, metadata, tagging, ownership, documentation, and support channels.

The stream-table duality posits that *any stream can be represented as a table, and any table can be represented as a stream*, which means:

- **To create a table from a stream**, consume each event from the stream in sequence. Then, append (or upsert) that data to a table on an event-by-event basis.
- **To create a stream from a table**, capture each modification made to the table (create, update, delete) as a discrete event and append it to an immutable log (a data stream).

While we're primarily interested in creating the table from the stream, it is important to note that you can also create a stream from a table. Change data capture mechanisms, which we discuss later in this section, do precisely that—capture the changes from the table and emit it to a stream. We'll come back to that later.

Next, there are two commonly used table types in headless data architectures.

1. **Append-only tables**
   Acts as a stream, but as a table format. Data can only be added but never updated or edited once written. Deletes, if required, are typically accommodated with a soft delete strategy. You simply append yet another row that provides deletion information. Append-only tables make an excellent store of record for everything that has happened in the past.
2. **Materialized tables**
   A materialized table applies some sort of logic to the conversion of the data stream into a table row. There are several common subtypes, including the:

- **Upsert type:** Data is upserted based on a **primary key**. Old data is overwritten when a new event with the same primary key is processed. Data can also be deleted from the table when indicated, typically by using a tombstone event. Upsert tables are the table-based equivalent of a compacted Kafka topic.
- **Aggregation type:** Data is aggregated with records already in the table. For example, you may keep an aggregation of *sales_amount per store_id*, summing the value of each sale to the existing store_id as new data arrives. Aggregations may also be more complex, including representing state machines and time-based windowing.

Append-only tables tend to be the easiest way to get started in the headless data architecture. They require very little code, are very high performance, and are semantically identical to data streams. Streams and tables represent the same data, but a stream-table data product allows you to access it through the Kafka API or via Iceberg, whichever works best for your use case.

The fact of the matter is that **it is far easier to take data in motion and slow it down into data at rest than it is to take data at rest and spur it into motion**. Shifting left to a streams-first approach makes making important time-sensitive decisions easier, cheaper, and safer. Shifting left allows you to take your well-defined streaming data products and convert them into other formats, be they columnar, graph, document, or relational.

Shifting left with the 6 components of a data streaming platform

Our customers have told us that they love the idea of stream-table data products—it's been one of the most discussed subjects with our most forward-thinking customers. In fact, many of them have independently converged on a similar idea but lacked the tooling to implement it on their own. For some, it was difficult to simplify and standardize the stream-to-table process. For others, the difficulty lay in establishing data discovery, metadata tracking, data management, and lineage functionality.

Confluent provides a complete data streaming platform (DSP) that gives you all of the components you need to accomplish a shift left. You may choose to build your own platform as well, but be warned that it's not just the components themselves that matter but also how they relate to one another. But without further ado, let's take a look at these components, what they do, how they relate to one another, and the role they play in simplifying your data architectures.

#1 Apache Kafka

Kafka provides reliable, scalable, and high-performance data streaming. Kafka topics provide the fundamental data streaming component in the form of an append-only log. Producers append data to the topic log, and consumers read the data and do their own business work. Kafka doesn't have any notion of content—it's just bytes in, bytes out.

For richer functionality, we'll need to rely on the next components in the DSP.

#2 Schema Registry and data quality

The schema registry provides explicit schema structures for your events, such as the popular Avro®, Protobuf, and JSON Schema formats. It also lets you manage schema evolution, such as setting requirements for forwards and backwards compatibility. Stream quality features like data validation ensure that you can only write data to topics that conform with the expected schema—no more accidentally writing malformed data or writing data to the wrong stream.

Adopting good governance also allows you to adopt data quality rules. While a schema may specify that a field must be a string, a data quality rule takes it a step further and specifies additional requirements about that string. For example, that it must be exactly nine characters long, in the case of a unique national identity number, or that it must contain an "@" symbol, in the case of an email. We're going to take a close look at solving this type of problem in the "Preventing and fixing bad data in a streams-first world" section.

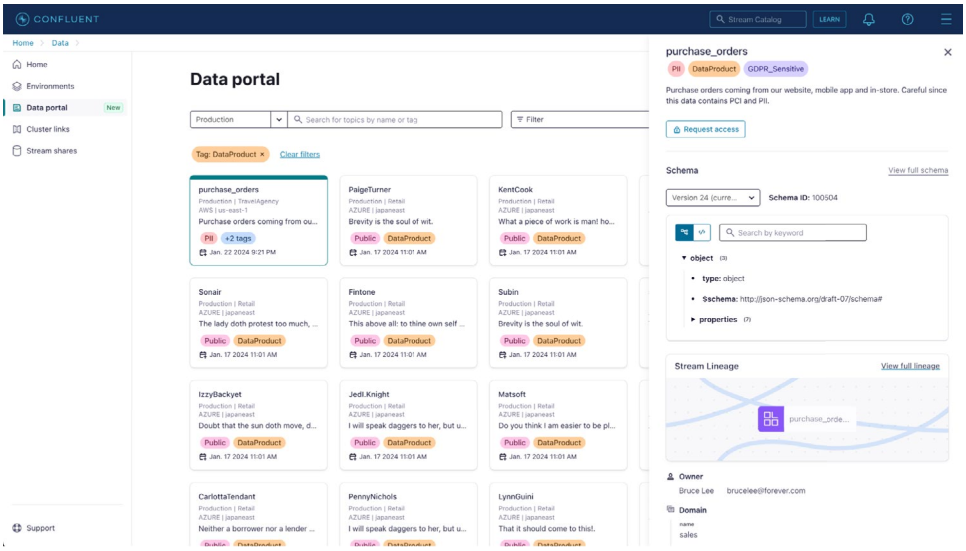
The schema registry, along with its host of data quality features, ensures that both the producers and the consumers of the data have a clear set of expectations. Further, it provides the foundation for integrating into human-centric workflows, including discovery, lineage exploration, and ad hoc queries.

#3 Data Portal

Data Portal combines several essential components to make building, sharing, discovering, and using streaming data products easy. Major sub-features include:

- Stream Catalog–Increase collaboration and productivity with self-service data discovery that allows teams to classify, organize, and find the needed data streams.
- Stream Lineage–Understand complex data relationships and uncover more insights with interactive, end-to-end maps of data streams.
- Access controls–Manage user access to Schema Registry subjects and topics, allowing multiple users with different access levels to various resources to collaborate.

Data Portal combines these components into an easy-to-use portal for your data product producers, consumers, and managers. Users can discover, search, inspect, manage, and access data products across your organization. Only those data you deliberately promote to a data product will show up in the portal, significantly reducing clutter and easing your users' data discovery.



Purchase orders coming from our website, mobile app and in-store. Careful since this data contains PCI and PII.

purchase_orders Production | TravelAgency AWS | us-east-1 Purchase orders coming from ou... PII +2 tags Jan. 22 2024 9:21 PM	PaigeTurner Production | Retail AZURE | japaneast Brevity is the soul of wit. Public DataProduct Jan. 17 2024 11:01 AM	KentCook Production | Retail AZURE | japaneast What a piece of work is man! ho... Public DataProduct Jan. 17 2024 11:01 AM
Sonair Production | Retail AZURE | japaneast The lady doth protest too much, ... Public DataProduct Jan. 17 2024 11:01 AM	Fintone Production | Retail AZURE | japaneast This above all: to thine own self ... Public DataProduct Jan. 17 2024 11:01 AM	Subin Production | Retail AZURE | japaneast Brevity is the soul of wit. Public DataProduct Jan. 17 2024 11:01 AM
IzzyBackyet Production | Retail AZURE | japaneast Doubt that the sun doth move, d... Public DataProduct Jan. 17 2024 11:01 AM	Jedi.Knight Production | Retail AZURE | japaneast I will speak daggers to her, but u... Public DataProduct Jan. 17 2024 11:01 AM	Matsoft Production | Retail AZURE | japaneast Do you think I am easier to be pl... Public DataProduct Jan. 17 2024 11:01 AM
CarlottaTendant Production | Retail AZURE | japaneast Neither a borrower nor a lender ...	PennyNichols Production | Retail AZURE | japaneast I will speak daggers to her, but u...	LynnGuini Production | Retail AZURE | japaneast That it should come to this!.

```
▾ object (3)
  • type: object
  • $schema: http://json-schema.org/draft-07/schema#
  ▸ properties (7)
```

Owner: Bruce Lee bruceelee@forever.com
Domain: sales

The Confluent Data Portal

#4 Kafka connectors

There are several ways to create tables from data streams provided by Kafka. One is by relying on Kafka connectors.

Connectors play two major roles in shifting left:

- **Source data from tables:** Create data streams from tables via periodic querying, change data capture, or deeper integration with the database's logging mechanism.
- **Sink streaming data into tables:** Write a data stream to a table, for example, as an append-only or materialized table.

Confluent offers fully-managed connectors for all the most popular (and even the less popular) databases and table formats. Confluent's fully managed connectors expand on the original open source Kafka Connect, providing all of the scaling and power of the original with added monitoring, logging, fault tolerance, and customizations.

Open source Kafka Connect is another option, coming packaged with Kafka by default. However, just like Kafka, you will have to run and manage your own infrastructure, including scaling and monitoring.

Plus you'll have to manage your connectors. Further, you'll also have to integrate it into your own data portal, particularly if you shift left into stream-table data products.

#5 Tableflow

Tableflow is a fully managed Kafka-topic-to-Iceberg-table converter. Simply push a button to convert your Kafka topic into an Iceberg-compatible append-only table. All operational overhead, management, scaling, and fault handling are taken care of for you so that you can focus on simply using the data instead. Tableflow-enabled topics are discoverable via the data portal.

Why use Tableflow over an Iceberg connector?

- Built-in table maintenance (e.g., compaction, garbage collection, snapshot management)
- External catalog service synchronization to send metadata to other Iceberg catalogs
- Very low overhead to get started—just push the button and the table is automatically created
- No need to manage connect resources or connector configurations

Tableflow is a major step in streamlining the production of stream-table data products. It makes it very easy for you to create data products so you can plug in the most suitable heads for processing and querying. Furthermore, it provides schema evolution capabilities for evolving your stream and table in lockstep to ensure that any changes you make won't inadvertently break established data contracts.

#6 Apache Flink

Last but not least is Flink. Flink provides full data stream processing capabilities, powering applications and data pipelines alike. Transform events, materialize them to tables, join them together, build aggregations, or choose from a whole other range of processing options. Flink can also read and process data directly from Iceberg tables in both batch and streaming modes. Consult the documentation for the latest Flink processing capabilities in Confluent.

You are free to choose other processing engines—after all, that's the whole point of shifting left into a headless data architecture. Both Kafka and Iceberg offer tons of compatibility with many different engines, languages, frameworks, and services. But to get the most out of your shift-left initiative, you're going to need to make sure that they plug into your schema registry and data portal and that they respect data contracts, data validation requirements, and schema evolutions.

Summing up the parts of the DSP

A data streaming platform is intended to make it easy to build, share, and use data streams. It's also built to serve the most common use cases, such as iteratively sourcing data from databases and providing it in table formats for other non-streaming engines to process. The DSP makes it *much* easier to shift left into stream-table data products. And while you can choose to build your own, the Confluent DSP provides everything you need right out of the box.

Now that we've covered data products and DSPs, let's take a look at some practical considerations for selecting a data set to shift left.

What makes a good shift-left candidate?

Most readers of this book will already have an established set of data pipelines that carry data from the operational plane to the analytics plane. The prospect of having to replace them wholesale is too daunting to face—plus it would take forever and cost a lot of people-hours.

The beauty of shifting left is that you can selectively apply the strategy to the data sets that matter most. There is no all-or-nothing. You can adopt it by incremental degrees, prototyping, trialing, testing, and committing to the shift on a case-by-case basis. However, there are some data sets that form the best candidates for getting your biggest bang-for-your-buck. Criteria to look for includes:

#1 High-demand data

This is the data that many systems and people rely on. It's in high demand across your organization because it's *foundational* to how your business operates—the tasks and actions undertaken by people and systems. Operational systems require this data to do their tasks, as do many analytical processes.

High-demand data is typically associated with the **nouns** that describe the area a business operates within. For example, e-commerce would include **products, orders, accounts, reviews, inventory,** and **payments**. A package delivery company would include **packages, vehicles, drivers, accounts, warehouses,** and **payments**.

High-demand data is often copied to multiple locations for local use. For example, a large organization with multiple divisions might use ETL/ELTs to copy data from the source to multiple data lakes, data warehouses, and SaaS endpoints. The result is multiple copies of data, each with accompanying pipeline creation and maintenance costs, network transfer costs, and the risk that data between the sets becomes out of sync.

#2 Critical data

Critical data is data where data correctness and reliability is essential. It cannot be open to misinterpretation nor can it be committed to or written haphazardly or carelessly. Schemas, data validation, data quality checks, and evolution rules are all essential in creating and communicating critical data.

ETL/ELT pipelines apply criticality checks after the fact, once the data has already left the operational plane and entered the analytics plane. Any checks performed from this point are only useful to downstream dependents in the analytics layer. Shifting all of this validation, schema enforcement, and quality checking upstream benefits every single consumer in the organization.

Examples of critical data include things like **payments, personally identifiable information (PII), credit information, banking transactions, legal contracts,** and **accounts receivable**.

Critical data also encompasses anything that can cause a loss of customer and business partner trust. For example, providing customers with conflicting reports comparing two similar-yet-different data sets from two different ETLs or from two different teams. Other examples include financial misrepresentation, including issuing bills for the wrong amount, charging for services not used, or conversely undercharging for actual services used.

Financial information often finds itself in the path of lost customer trust, predominantly because the operational side tends to be responsible for issuing the bills—and possibly collecting payment—while the analytics side tends to be responsible for tallying up revenue and reporting on gross trends. Unifying these use cases under a single common source eliminates a major source of issues.

#3 High-volume data

High-volume data can incur high expenses, particularly ETL/ELTs where data is copied again and again for each stage in the pipeline. For example, a source→bronze→silver pipeline results in at least two copies of data (one for bronze, one for silver), whereas a headless model provides silver-level quality with just a single copy of data.

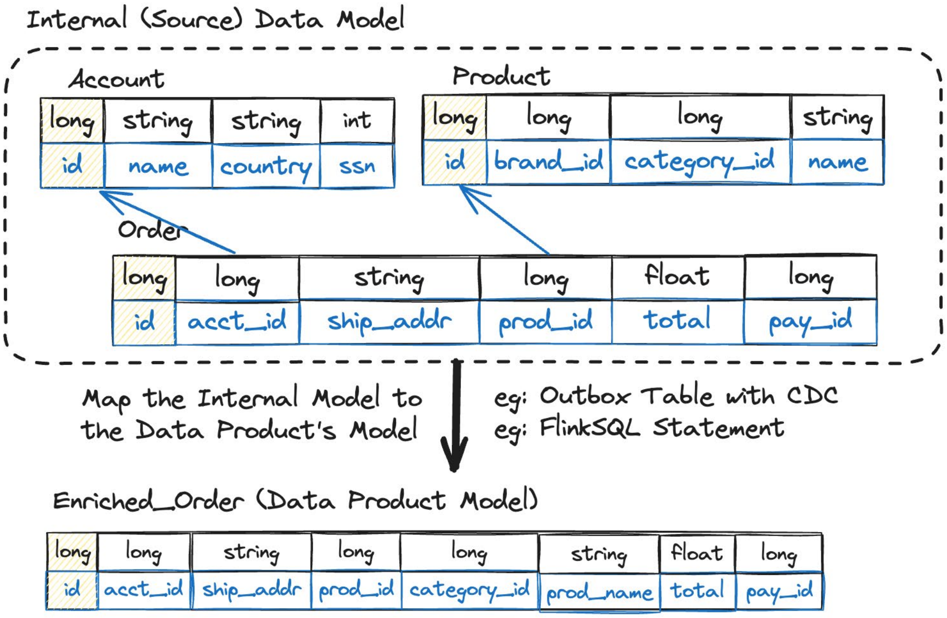
High-volume data is expensive to replicate to multiple endpoints. It also tends to fall out of sync with other copies due to latency between creation and replication and varying intervals between synchronizations.

Now that we've outlined some of the main identification criteria, let's take a look at an example of how we may choose to shift the creation of an existing data set to the left.

Example: Defining a data product's data model

Consider an e-commerce company. Their fundamental unit of sale is an **order**, which is used all across the company from fulfillment, to inventory management, to revenue, and to predictive modeling. Right now, each team is responsible for getting this data on their own—through a variety of ETL/ELTs, direct database access, periodic snapshots, and other ad hoc means. It is a prime candidate for shifting left.

The internal model is a normalized schema of **account**, **product**, and **order**. The external data model is what we choose to expose to the outside world, forming the schema definition of the data product.



Mapping a data source's internal model to the external data product model

The external model denormalizes the internal data model and provides a flat **enriched_order**. It eliminates **country**, **ssn**, and **brand_id** from the **enriched_order** output, isolating the internal model fields from coupling by downstream consumers. These fields could be added at a later point if required.

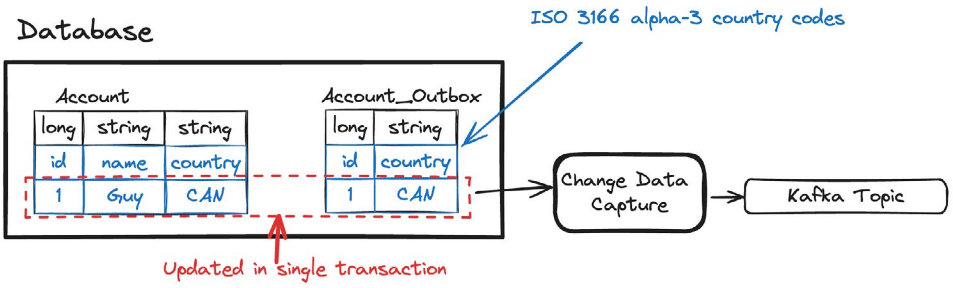
Now that we have a model in mind, let's look at a few options for how we'd actually build it.

Building a data product using the outbox pattern and change data capture

Change data capture (CDC) is one of the most common and powerful ways to get data from an existing system into a data stream. CDC captures data from the source database, typically by tailing the underlying transactional log. (Check out our detailed blog post for more info.) As a result, you can access a stream of near-real-time updates to the internal domain model of the database tables, letting you power rich streaming applications and analytics.

However, the resultant streams are coupled directly to the source's internal data model. Any changes to the internal model can affect your data streams. Simple changes like adding a column to a table may not cause any problems, whereas combining and deleting tables may cause your CDC connector and consumers to come to a screeching, error-filled halt.

The outbox table pattern is a popular technique to isolate the internal data model. You declare a database table to act as an outbox, define the schema to match the desired event schema, and populate it whenever you update your internal model. Note that you must use atomic transactions to ensure that your internal data model remains consistent with the outbox. Finally, you use a CDC connector to emit the outbox events into a data stream.



Capturing events with the outbox pattern requires the use of transactions

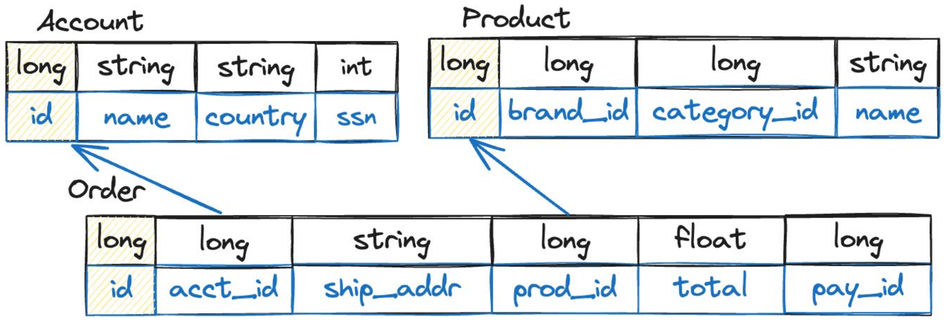
What's great about the outbox pattern is that you use the source database's query engine and data type checking to handle consistency. You can't write any data to the outbox if it doesn't match the schema, which provides you with a strong, well-defined schema for your data product right from the source.

You can also leverage the query engine for richer operations, such as denormalizing data from the source. Outbox tables work with *any database that supports transactions* and are a simple way to get started with well-defined data contracts without investing in yet another tool.

Keep in mind that the outbox pattern requires database processing and storage resources. Requirements will vary depending on transaction volume, data size, and query complexity (such as denormalizing or aggregating by time). Test the performance impact before you deploy to production to ensure your system still meets its SLAs and avoid unpleasant surprises.

Let's look at an example.

Back to our original data model. Each table has a primary **id** key. The **order** table has two foreign-key references, one to **account** and one to **product**.



The order table contains foreign-key references to account and product tables

We want to enrich an **order** event with **account** and **product** information and write it to the outbox whenever we create a new order. We'd modify our code by wrapping it into a transaction like so:

```
def insert_order(long acct_id,
                 String ship_addr,
                 long prod_id,
                 float total,
                 long pay_id){
    //Create the `order` object
    val order = Order(acct_id, ship_addr, prod_id, total, pay_id)

    //Transaction is atomic
    openTransaction {
          //Update the internal model
          writeToOrderTable(order)
          //Get the `account` and `product` for the outbox
          //Note that these lookups consume database resources
          val account = getAccount(acct_id)
          val product = getProduct(prod_id)

          //Denormalize and select fields to write to outbox
          //Note that the extra write consumes database resources
          writeToOutboxTable(order, account, product)
    }
    //Event is in the outbox, according to outbox Schema.
}
```

Note that in this code block, we have decided to **`get`** the data from each of the Account and Product tables. The **`writeToOutboxTable`** function takes all three rows as parameters, then denormalizes, selects, and forms the data according to the outbox schema and then writes it to the outbox. Finally, set up a CDC connector to pull the data out of the outbox asynchronously and write it to Kafka.

Coming Soon: Two-phase commit (2PC) support with KIP-939

Confluent and Kafka will soon offer more flexibility in sourcing data products from operational data stores. With KIP-939, the addition of a two-phase commit protocol to Kafka means data processing systems can safely write data to both operational databases and a data stream.

With the outbox table, we had to rely on a transaction to update both Order and Outbox, as Kafka doesn't currently support participation in 2PC. However, once KIP-939 is complete (target release pending), you'll be able to rewrite your code to simply write directly to Kafka in your transaction.

```
//Transaction is atomic
openTransaction {
      //Update the internal model
      writeToOrderTable(order)
      //Get the `account` and `product` for the outbox
      //Note that these lookups consume database resources
      val account = getAccount(acct_id)
      val product = getProduct(prod_id)
      //Denormalize and select fields for outbox,
      //then write to Kafka!
      //
      //Note: This is the only change!
      //
      writeToKafka(order, account, product)

}
//Data in both the order table and
//kafka topic are updated atomically!
```

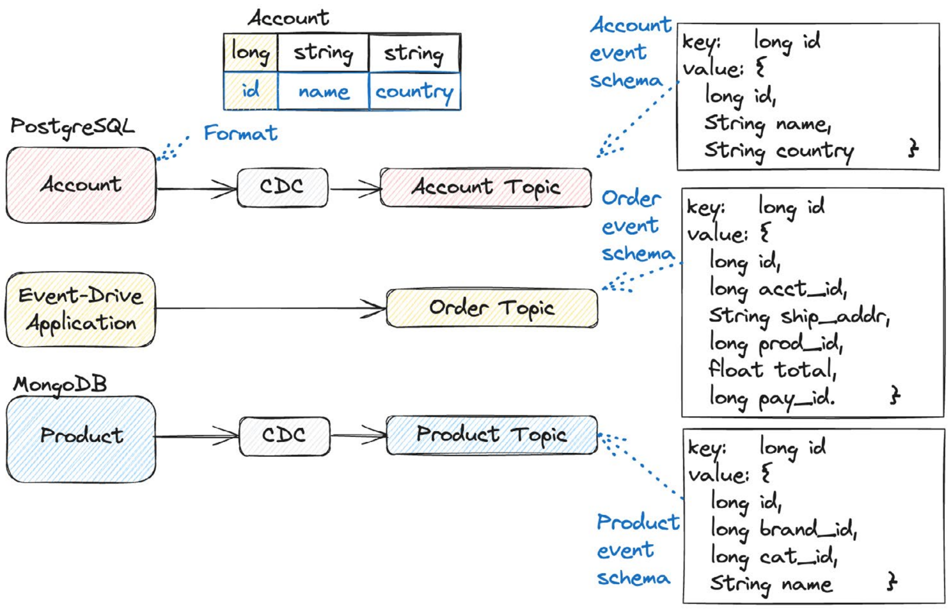
Transactions ensure eventual data consistency in both your database and your streams. Post KIP-939, you'll be able to rely on the transaction coordinator to ensure atomic data production to both table and stream. However, you'll still need to consider latency requirements and performance tradeoffs.

So far, we’ve looked at the scenario where all of your source data is within a single database, and you can leverage its query engine to denormalize and structure your data product. But let’s look at another common scenario: where the data you want to use to build your data product is scattered across multiple sources. This is where Flink SQL can really help us out.

Building a data product using Flink SQL

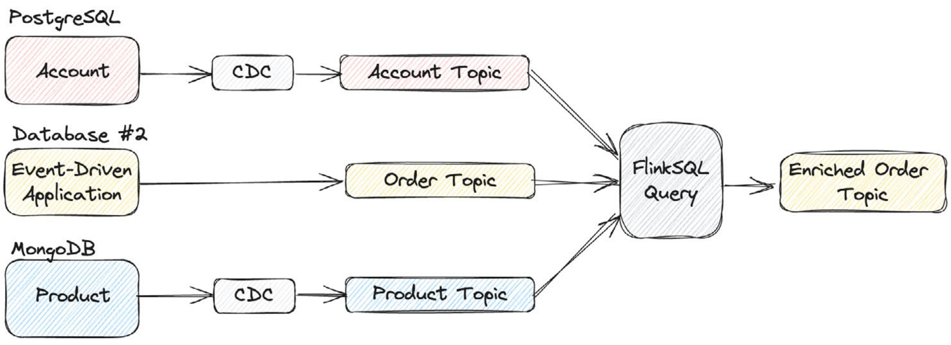
Let's revise our outbox table example. Instead of a single database, the **account**, **product**, and **order** data are each stored in different locations—one in a PostgreSQL database, one in a native data stream, and the third in a MongoDB document database.

For the **account** data in PostgreSQL, we'll use the PostgreSQL Debezium CDC connector, then simplify and flatten the data within the connector. Similarly, for the **product** schema in MongoDB, we'll use the MongoDB Debezium CDC connector and emit it to a topic. And for **order**, we won't need to do anything as it's already in a topic.



Using connectors to emit account and product data as topics ready to join with the order topic

Each topic contains an event-carried state based on its primary key. (I have previously called these events "facts".) What's great about this is that we can materialize the **account**, **product**, and **order** topics into their own Flink tables and then join them together whenever a new event arrives.



Joining the account, order, and product topics into an enriched order topic using Flink SQL

The Flink SQL code looks like this:

```
CREATE TABLE enriched_orders AS
SELECT o.id as order_id,
      a.id as account_id,
      a.country,
      o.ship_addr,
      o.total,
      o.pay_id,
      p.brand_id,
      p.cat_id,
      p.name
FROM orders AS o
LEFT JOIN accounts FOR SYSTEM TIME AS OF `o.$rowtime`AS a
ON a.id = o.account_id
LEFT JOIN products FOR SYSTEM TIME AS OF `o.$rowtime`AS p
ON p.id = o.product_id
```

Assumptions:

- Each record is partitioned according to its primary key.
- Rowtime represents the Kafka record timestamp.

The resulting **enriched_orders** topic contains denormalized events from three separate sources. When the upstream system creates, updates, or deletes an **order**, the Flink SQL logic will create a new **enriched_order** event. We can also compact the **enriched_orders** topic if we would like to reduce the amount of data stored.

Consumers can read the **enriched_order** data from the beginning of time, read the data from a given offset or timestamp, or simply jump to the latest data.

In this example, the data product is made up of these components:

1. The CDC connectors
2. The intermediate Kafka topics
3. The Flink SQL table and stream declarations
4. The Flink SQL join job
5. The **enriched_orders** topic as the output port

One helpful technique in organizing the components that make up your data products is to tag them with organizational metadata. For instance, you can tag the connectors, topics, schemas, and Flink SQL jobs (coming soon!) with a unique ID (such as the data product name). This can help you find all the components associated with your data product and help you organize and manage your dependencies.

What is the best method for building a data product?

We've covered the outbox pattern, the Flink SQL pattern, and introduced KIP-939 as an upcoming option. But when would you choose one over the other?

The outbox pattern is a good choice when:

- The data required for the data product is entirely within the database.
- You require a lot of denormalization.
- You don't want to run another downstream process to compile the data product.

The outbox pattern has some drawbacks that you will need to consider:

- It imposes extra processing loads on the database, sometimes excessively so.
- It must periodically clean out the outbox table to avoid unbounded growth.
- It requires deep integration with the application source code.
- It cannot join data outside of the database.
- There is no replayability if you built the data product incorrectly.

In general, the outbox is a good pattern for emitting denormalized data from a singular database, provided the performance and software modification impact is minimal.

The Flink SQL pattern is a good choice when:

- You want to isolate database performance from data product composition resources.
- Your data product requires data from multiple sources.
- Your database administrator will not let you build an outbox, due to privacy, security, or performance concerns.
- You can rewind the Flink SQL job to reprocess Kafka topics, letting you rebuild a data product in the case of an error or a poorly defined schema.
- You want a more granular view into how your data products are composed via Stream Lineage.

KIP-939 remains a promising future option that will give us more flexibility. In many use cases, we'll be able to remove connectors and outbox tables entirely, publishing directly to the stream and using Flink SQL to construct the data product. Alternatively, we can leverage the source database's engine to denormalize and construct the data product in code, writing it to the data product's topic directly in the transaction. We'll look into this subject more once KIP-939 is accepted and moving into beta release.

Evolving schemas and handling changes

It would be nice if we could declare our data products once and never have to change them, the reality is that the one big constant in any organization is *change*. Data changes over time, be it due to business requirements, organizational structures, or the emergence of new technologies.

How do we minimize disruption to our data product users without unduly burdening the creator? Let's look at how data contracts provide us with various options for evolving and changing our data products.

Schemas play an outsized critical role in data streaming. They define the data format, including names, types, fields, defaults, and documentation. The most popular streaming formats include Avro®, Protobuf, and JSON Schema.

The producer writes the data according to the schema, while the consumer reads and uses the data according to the schema. If the producer tries to write data that doesn't adhere to the schema, the Kafka client throws an exception and forces the producer to try again using the correct format. Meanwhile, the consumer is isolated from any data that does not adhere to the schema. It is a very elegant separation of concerns. The consumer doesn't need to guess or interpret the format and meaning of the underlying data, while the producer has a precise and well-defined definition for the data it must produce.

But while schemas provide a strong data definition, the form of the data can change over time. Let's briefly take a look at the two main types of changes:

- **Compatible changes:** Some changes to your data product schema may be forward, backward, or fully compatible. These are relatively easy evolutions, such as adding a field with a default value (check out more information in this course, including a hands-on guide to schemas). In a compatible change, existing consumers are not forced to update their code.
- **Breaking changes:** Other changes are not evolutionarily compatible and can incur much work (Avro, Protobuf, and JSON Schema vary in what they consider breaking changes). These changes typically include significantly restructuring the event's schema and requiring

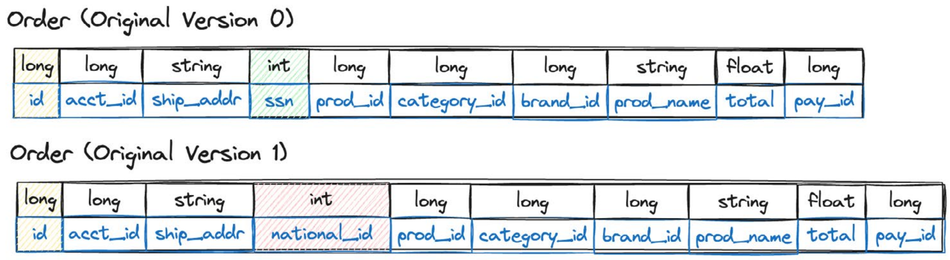
consumers to refactor their code. In a breaking change, existing consumers are forced to update their code to accommodate the breakage.

While compatible changes are relatively easy to handle (again, see the documentation or video on schema evolution), breaking changes can pose more difficulty. This is where our data contract capabilities can reduce friction, taking incompatible changes and smoothing them over for your consumers.

Schema changes with data streams

Let's take a look at an example: Our previous internal **account, product, and order** data model was pretty good, but we failed to account for non-U.S. citizens. Other countries have identifications that are not Social Security numbers (SSN), which doesn't match our schema declaration. (We're using ISO-3166 country codes in 'country').

If we decide to change the event schema, we'd have to update our schema from Version 0 to Version 1 (note the changes from **ssn** in Version 0 to **national_id** in Version 1). We'd also have to update our outbox definition or the Flink SQL query.



*A breaking schema change due to renaming **ssn** to **national_id***

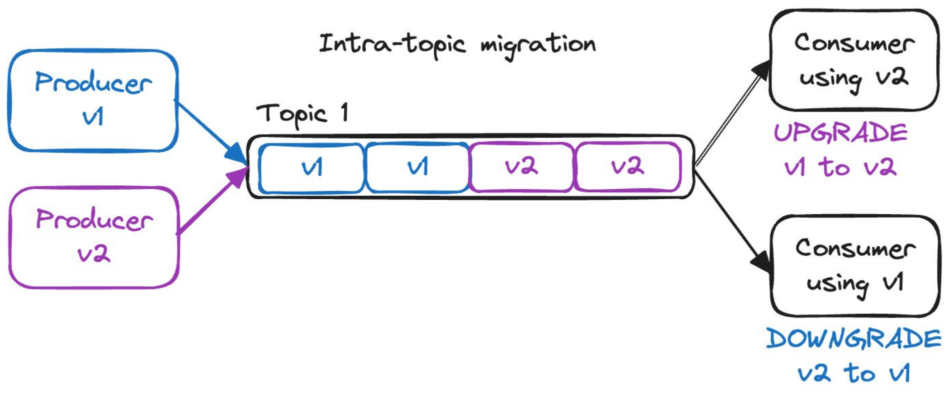
Data contracts provide a framework for rules and enforcement to reduce the complexity of migrating data during a breaking schema change. Confluent's **complex data migration** rules introduce major versioning for schemas that allow breaking compatibility without having to do messy data migrations between topics—the only other option available until now.

Here's a migration rule to convert a field name from `ssn` to `national_
id`, a typically incompatible change:

```
{
  “schema”: “...”,
  “ruleSet”: {
    “migrationRules”: [
      {
        “name”: “changeSsnToNationalId”,
        “kind”: “TRANSFORM”,
        “type”: “JSONATA”,
        “mode”: “UPGRADE”,
        “expr”: “$merge([$sift($, function($v, $k) {$k != ‘ssn’}),
{‘national_id’: $.’ssn’}])”
      },
        {
        “name”: “changeNationalIdToSsn”,
        “kind”: “TRANSFORM”,
        “type”: “JSONATA”,
        “mode”: “DOWNGRADE”,
        “expr”: “$merge([$sift($, function($v, $k) {$k !=
‘national_id’}), {‘ssn’: $.’national_id’}])”
        }
    ]
  }
}
```

The upgrade rule allows new consumers to read old messages, while the downgrade rule allows old consumers to read new messages. The consumer clients use these rules to convert incompatible events into a suitable format for their use cases.

The following figure shows two consumers, each written against the **v1** or the **v2** schema. The consumer written against **v1** of the schema **DOWNGRADES** the newer **v2** events to **v1**, which its business logic can understand and process. Meanwhile, the consumer written against **v2** **UPGRADES** the historical **v1** events to **v2**.



Consumers upgrade and downgrade their schemas as required

Schema changes with data streams is a mature and well-defined process. You can safely update your schemas while ensuring that you won't cause any unintentional outages or failures for your downstream consumers.

Propagating schema changes to Iceberg tables

Changes made to the Kafka data stream can automatically propagate to the Iceberg table. Tableflow, built as part of the Confluent DSP, is fully integrated with the schema registry. Tableflow-created Iceberg tables are linked with their corresponding topics, such that schema evolutions made to streams must also correspond to valid and supported table evolutions in Iceberg. In other words, you won't accidentally break your Iceberg tables due to a well-intentioned but erroneous update to your stream's schema.

If you're using an open source Iceberg connector made for Kafka Connect, you'll have to consult the project's documentation to see available functionality. Not all modes of schema evolution may be compatible between streams and tables, which can be problematic if you don't evolve the two in lockstep. An incompatible evolution can result in a **breaking change**, which, if unexpected, is equivalent to an incident. Whether it's a low-severity or a high-severity incident will depend on what you broke.

Navigating breaking changes

Breaking changes typically require rewriting your data product and negotiating a transition with your consumers. Breaking changes aren't very common in practice. Causes may include a data product refactoring, perhaps due to new security requirements, carelessness in the original creation of the data product, or a shifting business domain that causes an extensive redefinition of the data model.

Navigating breaking changes for your data products are much like any other breaking API change, including those for a REST API (e.g., /**v1**/ and /**v2**/ APIs). Support both the old data model and the new data model for a fixed period of time. Next, deprecate the old one so that it is no longer accessible to new consumers, and migrate its existing consumers onto the new data product. Finally, archive or delete it depending on your data retention policies.

This section is deliberately brief. There isn't a "magic bullet" for easily navigating breaking changes. You'll have to get everyone involved and affected together to discuss the changes, and come up with a plan to navigate through the implementation.

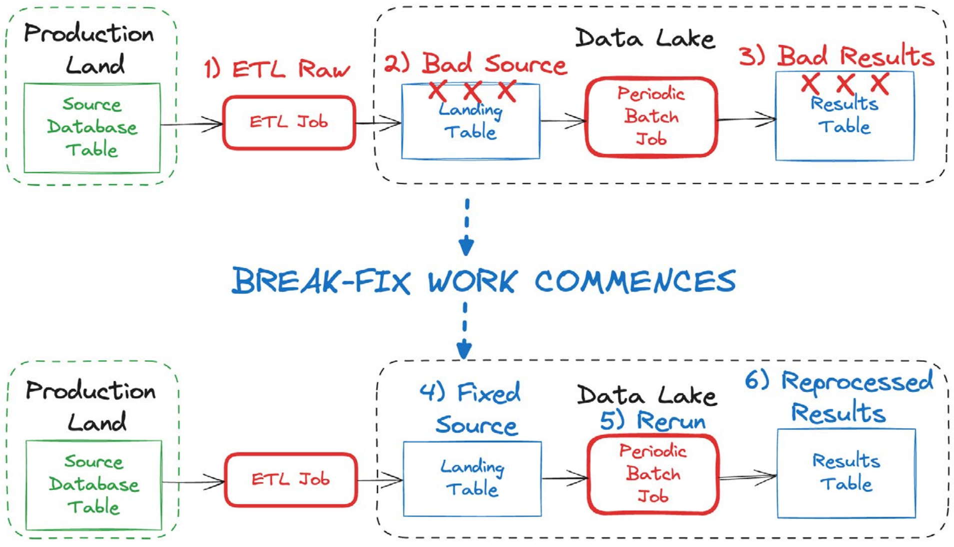
Preventing and fixing bad data in a streams-first world

Bad data is data that doesn't conform to *what is expected*. For example, an email address without the "@", or a credit card expiry where the MM/YY format is swapped to YY/MM. "Bad" can also include malformed and corrupted data, such that it's completely indecipherable and effectively garbage. In any case, bad data can cause serious issues and outages for all downstream data users.

Data streams are an *immutable log*, where data, once written, cannot be edited or deleted (outside of expiry or compaction—more on this later). The benefit is that consumers can read the events independently, at their own pace, and not worry about data being modified after they have already read it. The downside is that it makes it trickier to deal with "bad data," as we can't simply reach in and edit it once it has been written.

In construct, batch processing relies *extensively* on cutting out bad data and selectively replacing it with good data. You reach in, rip out whatever you deem as bad, then fill in the gap with good data—via reprocessing or pulling it back out of the source.

The following figure shows how copying raw data via an ETL job (1) can cause bad data to show up in the landing table (2), eventually resulting in (3) bad results. Let's just say in this case it's due to a type change in the **source database table** (an **int** is now a **string**) that causes type-checking errors downstream.



Copying data and figuring out the schema later can result in bad data requiring break-fix work to avoid bad results

As part of the declaration of an incident, the data engineer must account for the change of **int⟶string** by updating their periodic batch code (5) to account for the change in types. They reprocess the landing table data (4) via their cleaning job (5) to recompute the results table (6) for the affected rows.

Say the results table (6) is a Hive-declared table containing daily sales aggregates. We won't delete the whole table and recompute the aggregates—we'll just surgically remove the bad days (e.g., April 19 to 21, 2024), reprocess for that time range, and then move on to reprocessing the affected downstream dependencies. Basically, we stop the world, do all of our surgery and cleanup work, then restart things.

Streams-first data can't rely on stopping the world, cutting out the data, then reprocessing the work of everyone who used it.

First, it's slow—too slow to have any hope of powering any operational or time-sensitive analytical cases.

Second, it's expensive, in both people and service hours. You'd have to have an elaborate setup to halt all consumers, purge their affected data, and undo any actions they may have taken based on the bad data. That is not going to happen for any business at scale.

Third, it's not really possible anyways—append-only immutable logs don't allow editing. While deleting and recreating an entire stream just to fix a few records is *technically* possible, it's far too heavy handed (and slow, and expensive) to do every time you have bad data. There are much better ways.

Here are the important takeaways as we head into the streaming section:

- **There is little prevention against bad data.**
  The data engineering space has typically been very reactive. Import data of any and all quality now and let those poor data engineers sort it out later. Enforced schemas, restrictions on production database migrations, and formalized data contracts between the operations and data plane are rarely used.

- **The batch world relies on deleting bad data and reprocessing jobs.**
  Data is only immutable until it causes problems, and then it's back to the drawing board to remutate it into a stable format. This is true regardless of incremental or full refresh work.

Beware the side effects of processing bad data

Bad data can lead to bad decisions, both by humans and by services. Regardless of batch processing or streaming, bad data can cause your business to make incorrect decisions. Some decisions are irreversible, but other decisions may not be.

For one, reports and analytics built on bad data will disagree with those built on good data. Which one is wrong? While you're busy trying to figure it out, your customer is losing confidence in your business and may choose to pull out completely from your partnership. While we may call these false reports a side effect, in effect, they can seriously affect the affectations of our customers.

Alternatively, consider a system that tabulates vehicle loan payments but incorrectly flags a customer as non-paying. Those burly men that go to repossess the vehicle don't work for free, and once you figure out you've made a mistake, you'll have to pay someone to go give it back to them.

Any decision-making that relies on bad data, whether batch or streaming, can lead to incorrect decisions. The consequences can vary from negligible to catastrophic, and real costs will accrue regardless of if it's possible to issue corrective action. There can be significant negative impacts from using bad data, and the reality is that only some results may be reversible.

With all that being said, we won't be able to go into all of the ways you can undo bad decisions made by using bad data. Why? Well, it's *primarily a business problem*. What does your business do if it makes a bad decision with bad data? Apologize? Refund? Partial refund? Take the item back? Cancel a booking? Make a new booking? You will have to figure out what your business requirements are for handling things when decisions are made with bad data.

But all is not lost! There exist many strategies for *preventing* bad data from getting into your streams (and tables) in the first place, and some options for *fixing* it when it does.

Strategies for handling and preventing bad data in streams

The most successful strategies for mitigating and fixing bad data in streams include:

1. **Prevention:** Prevent bad data from entering the stream in the first place—use schemas, testing, and validation rules. Fail fast and gracefully when data is incorrect.
2. **Event design:** Use event designs that let you issue corrections, overwriting previous bad data.
3. **Rewind, rebuild, and retry:** When all else fails and deeper intervention is required.

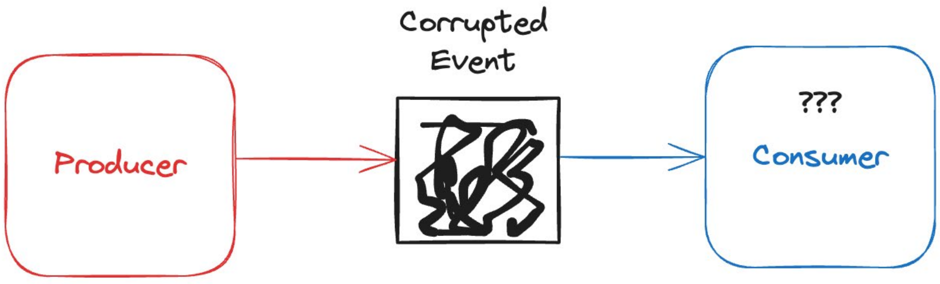
We'll need to explore *what* kind of bad we're dealing with and *where* it comes from to properly explore the roles of these strategies. So let's take a quick side trip into the main types of bad data you can expect to see in a data stream.

The main types of bad data in data streams

As we go through the types, you may notice a recurring reason for how bad data can get into your data stream. We'll revisit that at the end of this section.

#1 Corrupted data

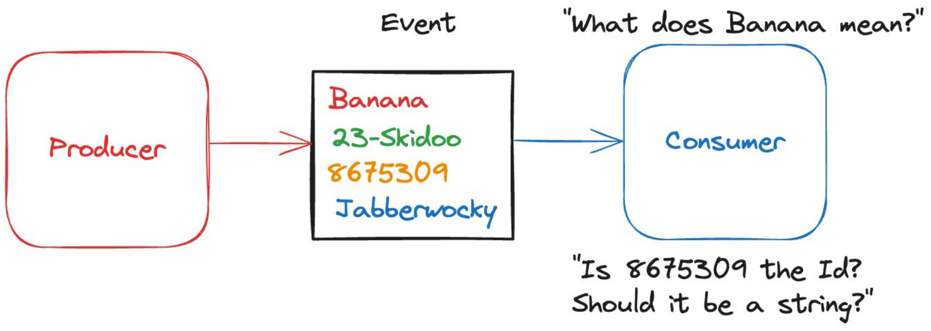
The data is simply indecipherable. It's garbage. It turned into a sequence of bytes with no possible way to retrieve the original data. Data corruption is relatively rare, but may be caused by faulty serializers that convert data objects into a plain array of bytes for Kafka. Luckily, you can test for that.



How corrupted events impact consumers

#2 Event has no schema

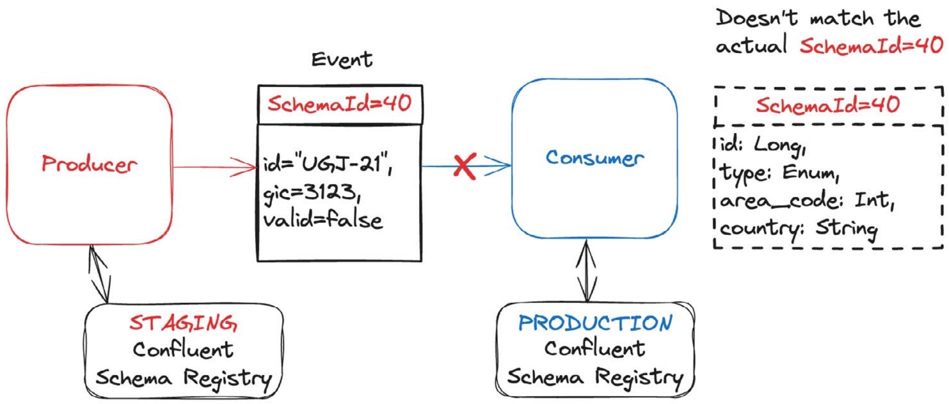
Someone has decided to send events with no schema. How do you know what's "good data" and what's "bad data" if there are no structures, types, names, requirements, or limitations? **NOTE:** This is an extremely common source of bad data for ETL/ELT pipelines—write it first, then figure it out later is expensive.



How events with no schema are interpreted by consumers

#3 Event has an invalid schema

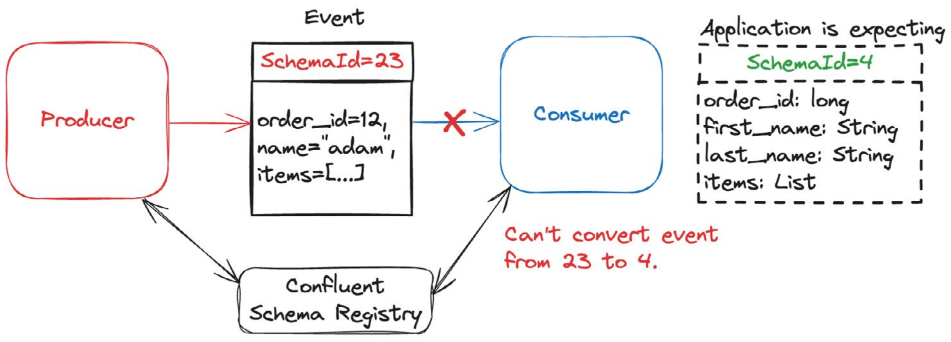
Your event's purported schema can't be applied to the data. For example, you're using the Confluent Schema Registry with Kafka, but your event's schema ID doesn't correspond to a valid schema. It is possible you deleted your schema, or that your serializer has inserted the wrong schema ID (perhaps for a different schema registry, in a staging or testing environment?).



Consumer is unable to interpret the data due to an invalid schema

#4 Incompatible schema evolution

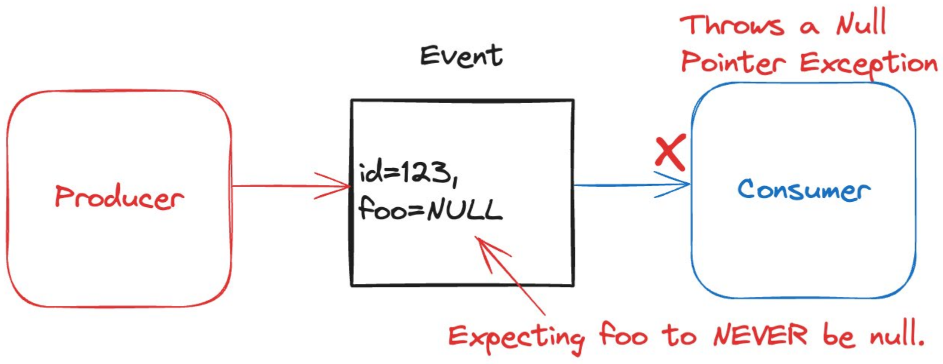
You're using a schema but the consumer cannot convert the schema into a suitable format. The event is deserializable but not mappable to the schema that the consumer expects. This is usually because your source has undergone breaking schema evolution , but your consumers have not been updated to account for it. (Note that evolution rules vary per schema type.)



Consumer is unable to convert the data due to a breaking schema evolution

#5 Logically invalid value in a field

Your event has a field with a value that should never be. For example, an array of integers for "first_name," or a null in a field declared as a NPE (see below).


Event
id=123,
foo=NULL
Producer
Consumer
Throws a Null Pointer Exception
Expecting foo to NEVER be null.

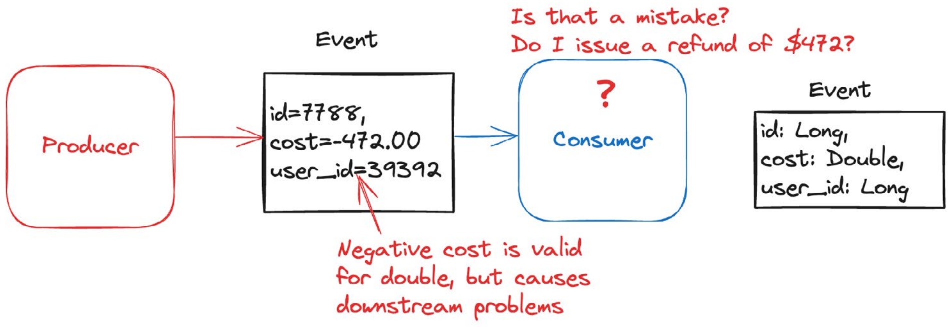
Consumer throws an exception due to a logically invalid value in a field

This error type arises when you are not using a well-defined schema but rather a set of implicit conventions. It can also arise if you are using an invalid, incomplete, old, or homemade library for serialization that ignores parts of your serialization protocol.

#6 Logically valid but semantically incorrect

These types of errors are a bit trickier to catch. For example, you may have a serializable string for a "**first_name**" field, but the name is "**Robert**"); **DROP TABLE Students;—**". A SQL injection prompt. While little Bobby Tables may be a valid string, and an invalid first name—and in this case it's downright damaging.

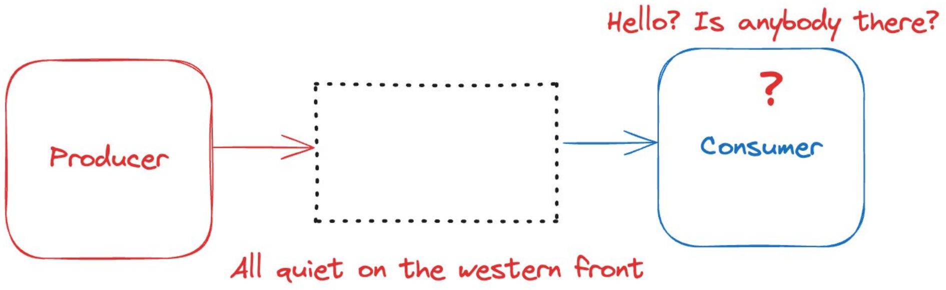
Here's another example. The following shows an event with a negative "cost." What is the consumer supposed to do with an order where the cost is negative? This could be a case of a simple bug that slipped through into production, or something more serious. But since it doesn't meet expectations, it's bad data.



A negative cost is an example of logically valid but semantically incorrect data in an event

#7 Missing events

This one is pretty easy. No data was produced, but there should have been something. Right?



What happens when an event is missing?

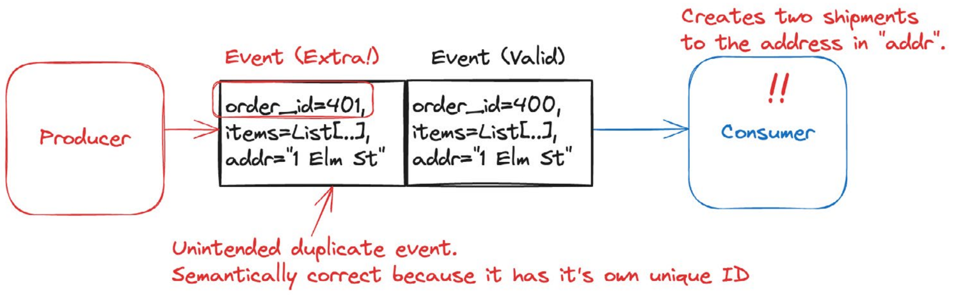
The nice thing about this type of bad data is that it's fairly easy to prevent via testing. However, it can have quite an impact if only *some* of the data is missing, making it harder to detect. More on this in a bit.

#8 Events that should not have been produced

There is no undo button to call back an event once it is published to a data stream. We can fence out one source of duplicates with idempotent production, meaning that intermittent failures and producer retries won't accidentally create duplicates. However, we cannot fence out duplicates that are logically indistinguishable from other events.

These types of bad events are typically created due to bugs in your producer code. For example, you may have a producer that creates a duplicate of:

- An event that indicates a change or delta ("add 1 to sum"), such that an aggregation of the data leads to an incorrect value.
- An analytics event, such as tracking which advertisements a user clicked on. This will also lead to an overinflation of engagement values.
- An e-commerce order with its own unique **order_id** (see below). It may cause a duplicate order to be shipped (and billed) to a customer.


Creates two shipments to the address in "addr".
Event (Extra!)
Event (Valid)
Producer
order_id=401, items=List[..], addr="1 Elm St"
order_id=400, items=List[..], addr="1 Elm St"
!!
Consumer
Unintended duplicate event. Semantically correct because it has it's own unique ID

What can happen when an event has been unintentionally duplicated

While there are likely more types of bad data in data streams that I may have missed, this should give you a good idea of the types of problems we typically run into. Now let's look at how we can solve these types of bad data, starting with our first strategy: *prevention*.

Preventing bad data with schemas, validation, and tests

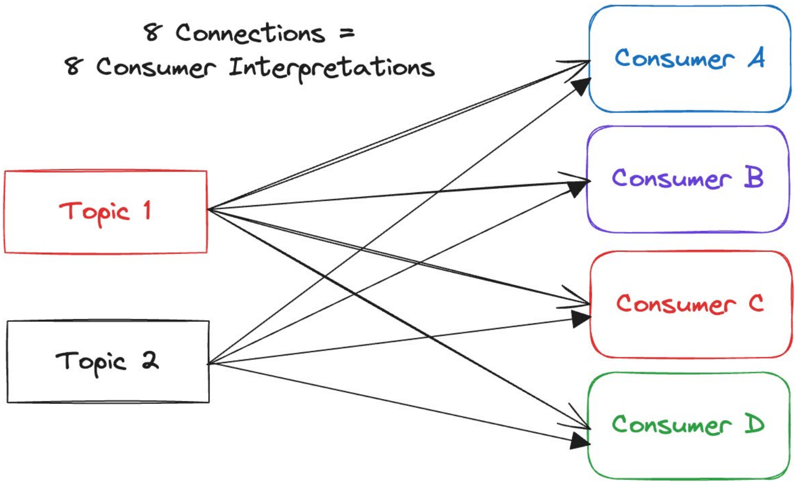
Preventing the entry of bad data into your system is the number one approach to making your life better. Diet and exercise are great, but there's no better feeling than watching well-structured data seamlessly propagate through your systems.

First and foremost are schemas. Confluent Schema Registry supports Avro, Protobuf, and JSON Schema. Choose one and use it. Do yourself, your colleagues, and your future selves a favor. It's the best investment you'll ever make.

Preventing bad data types #1–5 with schemas and schema evolution

Schemas *significantly* reduce your error incident rates by preventing your producers from writing bad data, making it far easier for your consumers to focus on using the data instead of making best-effort attempts to parse its meaning. Schemas form a big part of *preventing bad data*, and it's far, far, *far* easier to simply prevent bad data from getting into your streams than it is to try to fix it after the damage has already started.

Let's say we have two topics with no schemas and four consumers consuming them.



Eight independent interpretations of data

There are eight possible chances that a consumer will misinterpret the data. And the more consumers and topics you have, the greater the chance they misinterpret data when compared to their peers. Not only will your consumers get loud, world-stopping exceptions, but they may also get silent errors—miscalculating sums and misattributing results that lead to undetected divergence of consumer results.

These discrepancies regularly pop up in data engineering, such as when one team's engagement report doesn't match the other team's billing report due to divergent interpretations of unstructured data.

Adopting schemas provides a massive reduction in the chance that bad data will get into your production stream.

Preventing type #6 (logically valid but semantically incorrect) with data quality rules

The data quality rules that we introduced in the data streaming platform section let us build a safety net for the *content* of data in a given field. You can find more information under Confluent's data quality rules documentation.

Here's an example of a Confluent data quality rule for a U.S. Social Security number (SSN).

```
{
 "schema": "…",
 "ruleSet": {
 "domainRules": [
     {
       "name": "checkSsnLen",
       "kind": "CONDITION",
       "type": "CEL",
       "mode": "WRITE",
       "expr": "size(message.ssn) == 9"
     }
   ]
  }
}
```

This rule enforces an exact length of nine characters for the SSN. If it's an integer, we could also enforce that it must be positive and, if a string, that it must only contain numeric characters.

The data quality checks are applied when the producer attempts to serialize data into a Kafka record. If the **message.ssn** field is not exactly 9 characters in length, then the serializer will throw an exception. Alternatively, you can also send the record to a dead-letter queue (DLQ) upon failure.

Approach DLQ usage with caution. Shunting data into a side stream means that you'll still have to deal with it later, often by manual intervention. If you choose to use a DLQ, ensure that you have a process in place to deal with those events.

Preventing types #7 (missing data) and #8 (data that shouldn't have been produced) with testing

Third in our trio of prevention heroes is **testing**. Write unit and integration tests that exercise your serializers and deserializers, including schema formats (validate against your production schema registry), data validation rules, and the business logic that powers your applications. Integrate producer testing with your CI/CD pipeline so that your applications go through a rigorous evaluation before they're deployed to production.

Both type seven (**missing data**) and type eight (**data that shouldn't have been produced**) are actually pretty easy to test against. One of the beautiful things about data streaming systems is that it's pretty easy to test them. For event-driven applications, this is as simple as writing a single event to the input and observing what comes out the output. When integrating with a database, say via CDC, it's as simple as modifying a row in the database and seeing what comes out the other side. Once you find the bug in your logic, write another test to ensure that you don't get a regression.

Preventing bad data is essential for shifting left

Bad data can creep into your data sets in a variety of ways. **Prevention is the single most effective strategy for dealing with bad data, bar none**. Preventing bad data from getting into your system will be the simplest and most cost effective solution you can implement. But mistakes will happen, so let's take a look at the next major component in preventing and fixing bad data—**event design**.

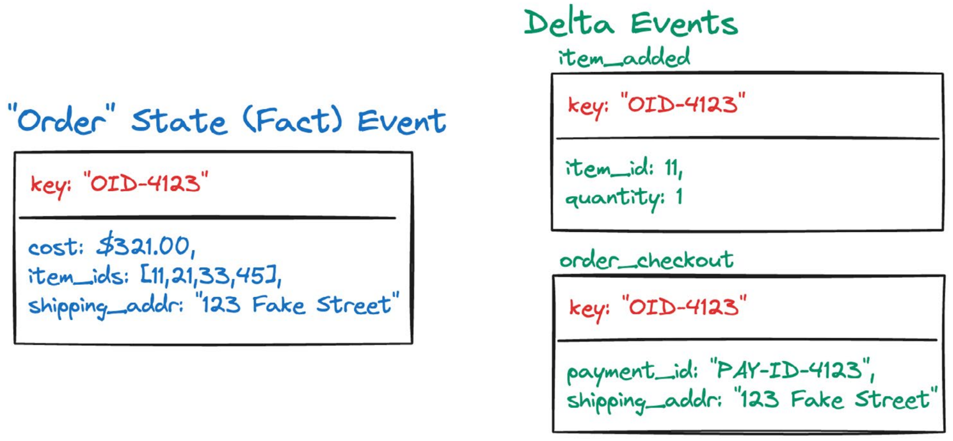
Fixing bad data through event design

Event design plays a big role in your ability to fix bad data in your data streams. And much like using schemas and proper testing, this is something you'll need to think about and plan for during the design of your application. Well-designed events significantly ease not only bad data remediation issues, but also related concerns like compliance with GDPR and CCPA.

Event design heavily influences the impact of bad data and your options for repairing it. First, let's look at **state (or fact)** events, in contrast to **delta (or action)** events.

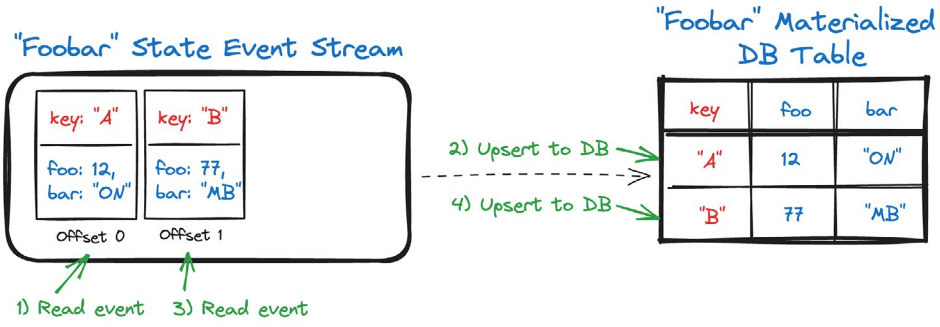
State (fact) and delta events

State events contain the entire statement of fact for a given entity (e.g., order, product, customer, shipment). Think of state events exactly like you would think about rows of a table in a relational database—each presents an entire accounting of information, along with a schema, well-defined types, and defaults (not shown in the picture for brevity's sake).



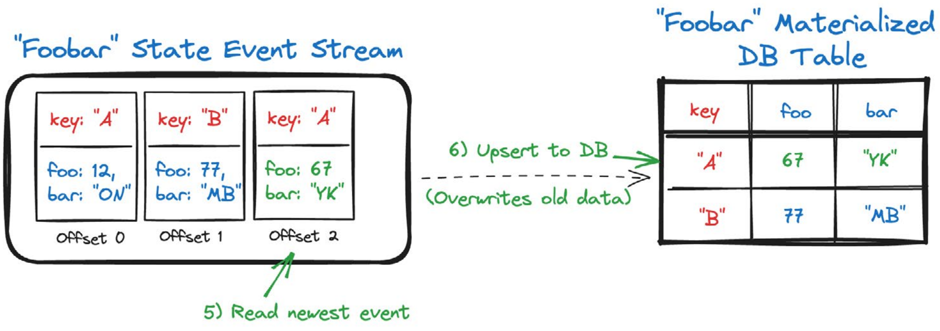
State shows the entire entity state, whereas delta just shows the change

State events enable event-carried state transfer (ECST), which lets you easily build and share state across services. Consumers can *materialize* the state into their own services, databases, and data sets, depending on their own needs and use cases.



A consumer service materializing a data stream made of state events into a table

Materializing is pretty straightforward. The consumer service reads an event (1) and then upserts it into its own database (2), and you repeat the process (3 and 4) for each new event. Every time you read an event, you have the option to apply business logic, react to the contents, and otherwise drive business logic.



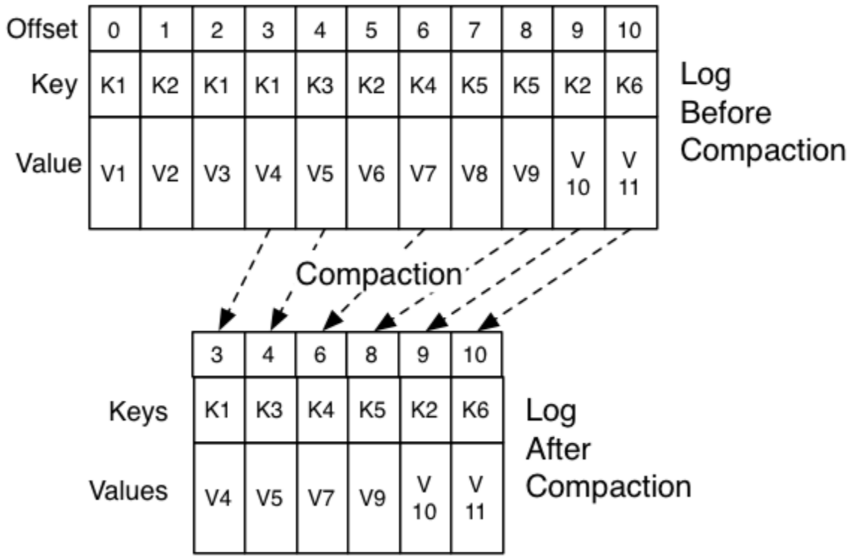
A consumer service reading new events to upsert into its own database

Updating the data associated with a key A (5) results in a new event. That event is then consumed and upserted (6) into the downstream consumer data set, allowing the consumer to react accordingly. Note that your consumer is not obligated to store any data that it doesn't require—it can simply discard unused fields and values.

Deltas, on the other hand, describe a change or an action. In the case of the order, they describe **item_added**, and **order_checkout**, though reasonably you should expect many more deltas, particularly as there are many different ways to create, modify, add, remove, and change an entity.

While there are tradeoffs of these two event design patterns, the important thing for this post is that you understand the difference between delta and state events. Why? Because *only* state events benefit from topic compaction, which is critical for deleting bad, old, private, and/or sensitive data.

Compaction is a process in Apache Kafka that retains the latest value for each record key (e.g., key = "A", as above) and deletes older versions of that data with the same record key. Compaction enables the complete deletion of records via tombstones from the topic itself—all records of the *same key* that come *before* the tombstone will be deleted during compaction.



Compacting a data stream–an asynchronous process[4]

Aside from enabling deletion via compaction, tombstones also indicate to registered consumers that the data for that key has been deleted and they should act accordingly. For example, they should delete the associated data from their own internal state store, update any business operations affected by the deletion, and emit any associated events to other services.

Compaction contributes to the eventual correctness of your data, though your consumers will still need to deal with any incorrect side-effects

[4] Confluent Documentation. "Log Compaction." Confluent. Retrieved July 22, 2024.

from earlier incorrect data. However, this remains identical as if you were writing and reading to a shared database—any decisions made off the incorrect data, either through a stream or by querying a table, must still be accounted for (and reversed if necessary). The eventual correction only prevents future mistakes.

State events: Fix it once, fix it right

It's *really easy* to fix bad state data. Just correct it at the source (e.g., the application that created the data), and the state event will propagate to all registered downstream consumers. Compaction will eventually clean up the bad data, though you can force compaction too if you cannot wait (perhaps due to security reasons).

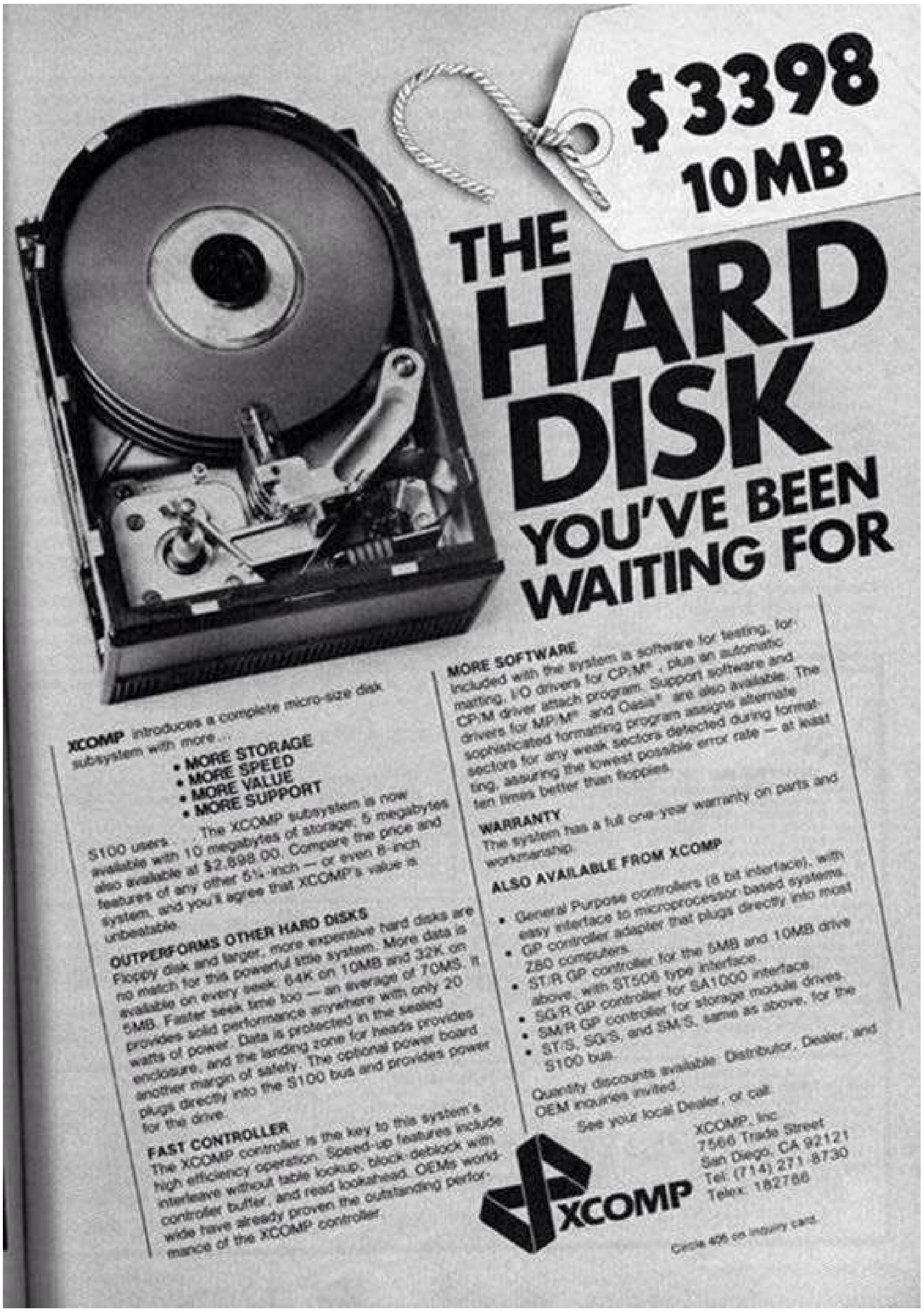
You can fiddle around with compaction settings to better suit your needs, such as compacting as soon as possible or only compacting data older than 30 days (min.compaction.lag.ms= 2592000000). Note that active Kafka segments can't be compacted immediately, the segment must first be closed.

State events are easy to use and map to database concepts that the vast majority of developers are already familiar with. Consumers can also *infer* the deltas of what has changed from the last event (n-1) by comparing it to their current state (n). And even more, they can compare it to the state before that (n-2), before that (n-3), and so forth (n-x), so long as you're willing to keep and store that data in your microservice's state store.

A common objection:

"Shouldn't we store as little data as possible so that we don't waste space?"

It depends on what you mean by "little." You should be careful with how much data you move around and store, but only after a certain point. This isn't the 1980s, and you're not paying $339.8 per MB for disk. You're far more likely to be paying $0.08/GB-month for AWS EBS gp3, or you're paying $0.023/GB-month for AWS S3.



$3398
10 MB

THE HARD DISK YOU'VE BEEN WAITING FOR

XCOMP introduces a complete micro-size disk subsystem with more…

- MORE STORAGE
- MORE SPEED
- MORE VALUE
- MORE SUPPORT

S100 users… The XCOMP subsystem is now available with 10 megabytes of storage; 5 megabytes also available at $2,898.00. Compare the price and features of any other 5¼-inch — or even 8-inch system, and you'll agree that XCOMP's value is unbeatable.

OUTPERFORMS OTHER HARD DISKS
Floppy disk and larger, more expensive hard disks are no match for this powerful little system. More data is available on every seek: 64K on 10MB and 32K on 5MB. Faster seek time too — an average of 70MS. It provides solid performance anywhere with only 20 watts of power. Data is protected in the sealed enclosure, and the landing zone for heads provides another margin of safety. The optional power board plugs directly into the S100 bus and provides power for the drive.

FAST CONTROLLER
The XCOMP controller is the key to this system's high efficiency operation. Speed-up features include interleave without table lookup, block-deblock with controller buffer, and read lookahead. OEMs world-wide have already proven the outstanding performance of the XCOMP controller.

MORE SOFTWARE
Included with the system is software for testing, formatting, I/O drivers for CP/M®, plus an automatic CP/M driver attach program. Support software and drivers for MP/M® and Oasis® are also available. The sophisticated formatting program assigns alternate sectors for any weak sectors detected during formatting, assuring the lowest possible error rate — at least ten times better than floppies.

WARRANTY
The system has a full one-year warranty on parts and workmanship.

ALSO AVAILABLE FROM XCOMP

- General Purpose controllers (8 bit interface), with easy interface to microprocessor based systems.
- GP controller adapter that plugs directly into most Z80 computers.
- ST/R GP controller for the 5MB and 10MB drive above, with ST506 type interface.
- SG/R GP controller for SA1000 interface.
- SM/R GP controller for storage module drives.
- ST/S, SG/S, and SM/S, same as above, for the S100 bus.

Quantity discounts available: Distributor, Dealer, and OEM inquiries invited.

See your local Dealer, or call:

XCOMP

XCOMP, Inc.
7566 Trade Street
San Diego, CA 92121
Tel: (714) 271-8730
Telex: 182786

Circle 406 on inquiry card.

An example of a faulty mental model of storage pricing[5]

[5] Fisher, T. (2022). "21 Things You Didn't Know About Hard Drives." Lifewire.

State is *cheap*. Network is *cheap*. Be careful about cross-availability zone (AZ) and cross-region data replication costs, though even those are starting to disappear. Note that many massive cloud providers, like Azure, have done away with cross-AZ charges entirely (May 2024). By and large, most state events won't cost you much at all—but in the case that you have a very large payload and do not want to replicate it everywhere, you can also consider using the claim-check pattern.

Maintaining per-microservice state is very cheap these days thanks to cloud storage services. And since you only need to keep the state your microservices or jobs care about, you can trim the per-consumer replication to a smaller subset in most cases.

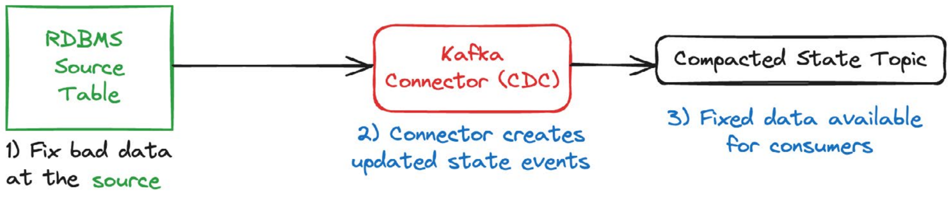
State events provide you with a ton of flexibility and let you keep complexity to a minimum. Embrace today's cheap compute primitives, and focus on building useful applications and data products instead of trying to slash 10% of an event's size. (And feel free to use compression if you haven't already.)

So how do state events help us fix the bad data? Let's take a look at a few simple examples, one using a database source, one with a topic source, and one with an FTP source.

State events and fixing bad data at the source

Kafka Connect is the most common way to bootstrap events from a database. Updates made to a registered database's table rows (e.g., create, update, delete) are emitted to a Kafka topic as discrete state events.

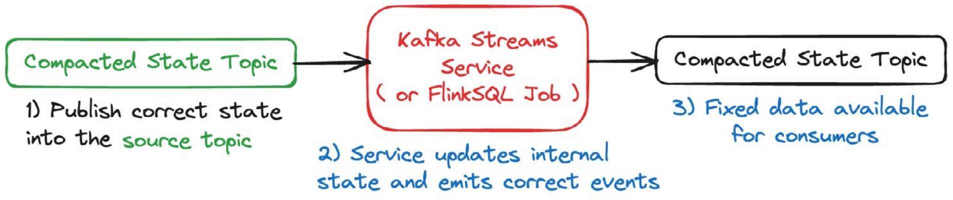
You can, for example, connect to a MySQL, PostgreSQL, MongoDB, or Oracle database using Debezium (a change data capture connector). Change-data events are **state-type** events, and feature both **before** and **after** fields indicating the before state and after state due to the modification. You can find out more in the official documentation, and there are plenty of other articles written on CDC usage on the web.


RDBMS Source Table
1) Fix bad data at the source
Kafka Connector (CDC)
2) Connector creates updated state events
Compacted State Topic
3) Fixed data available for consumers

Fix the bad data at the database source and propagate it to the compacted state topic

To fix the bad data in your Kafka Connect powered topic, simply **fix the data in your source database (1)**. The change-data connector (CDC, 2a) takes the data from the database log, packages it into events, and publishes it to the compacted output topic. By default, the schema of your state type maps directly to your table source—so be careful if you're going to go about migrating your tables.

Note that this process is *exactly the same as what you would do for batch-based ETL*. Fix the bad data at source, rerun the batch import job, then upsert or merge the fixes into the landing table data set. This is simply the stream-based equivalent.

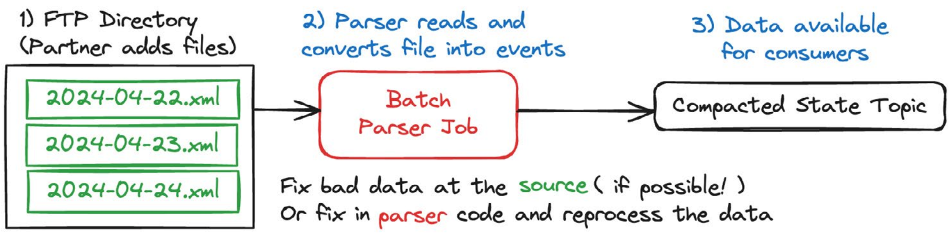

Compacted State Topic
1) Publish correct state into the source topic
Kafka Streams Service (or FlinkSQL Job)
2) Service updates internal state and emits correct events
Compacted State Topic
3) Fixed data available for consumers

Fix the bad data in the compacted source topic and propagate it to the downstream compacted state topic

Similarly, for example, a Kafka Streams application (2) can rely on compacted state topics (1) as its input, knowing that it'll always get the eventually correct state event for a given record. Any events that it might publish (3) will also be corrected for its own downstream consumers.

If the service itself receives bad data—say a bad schema evolution, or even corrupted data—it can log the event as an error, divert it to a dead-letter queue (DLQ), and continue processing the other data.

Lastly, consider an FTP directory where business partners (i.e., the ones who give us money to advertise or do work for them) drop documents containing information about their business. Let's say they're dropping in information about their total product inventory, so that we can display the current stock to the customer. (Yes, sometimes this is as close to data streaming as a partner is willing or able to get.)


1) FTP Directory (Partner adds files)
2024-04-22.xml
2024-04-23.xml
2024-04-24.xml
2) Parser reads and converts file into events
Batch Parser Job
3) Data available for consumers
Compacted State Topic
Fix bad data at the source (if possible!)
Or fix in parser code and reprocess the data

Fixing bad data dropped into an FTP directory

We're not going to run a full-time streaming job just idling away waiting for updates to this directory. Instead, when we detect a file landing in the bucket, we can kick off a batch-based job, parse the data out of the .xml file, and convert it into events keyed on the **productId** representing the current inventory state.

If our partner passes us *bad data*, we're not going to be able to parse it correctly with our current logic. We can, of course, ask them nicely to resend the correct data(1), but we might also take the opportunity to investigate what the error is, to see if it's a problem with our parser(2), and not their formatting. Some cases, such as if the partner sends a completely corrupted file, require it to be resent. In other cases, they may simply leave it to us data engineers to fix it up on our own.

So we identify the errors, add code updates, and new test cases, and reprocess the data to ensure that the compacted output (3) is eventually accurate. It doesn't matter if we publish duplicate events since they're effectively benign (idempotent), and won't cause any changes to the consumer's state.

That's enough for state events. By now you should have a good idea how they work. I like state events. They're powerful. They're easy to fix. You can compact them. They map nicely to database tables. You can store only what you need. You can infer the deltas from any point in time so long as you've stored them.

Can I fix bad data for delta-style events?

What about if I write some bad data into a delta-style event? Am I just straight out of luck?

Not quite. But the reality is that it's *much harder* to clean up delta-style events than it is state-style events. Why?

The major obstacle to fixing deltas (and any other non-state event, like commands) is that *you can't compact them—no updates, no deletions*. Every single delta is essential for ensuring correctness, as each new delta is in relation to the previous delta. A bad delta represents a change into a bad state. So what do you do when you get yourself into a bad state? You really have two strategies left:

1. **Undo** the bad deltas with new deltas. This is a **build-forward** technique, where we simply add new data to undo the old data. (**WARNING:** This is very hard to accomplish in practice.)

2. **Rewind, rebuild, and retry** the topic by filtering out the bad data. Then, restore consumers from a snapshot (or from the beginning of the topic) and reprocess. This is the final technique for repairing bad data, and it's also the most labor-intensive and expensive. We'll cover this more in the final section as it technically also applies to state events.

Both options require you to identify every single offset for each bad delta event, a task that varies in difficulty depending on the quantity and scope of bad events. The larger the data set and the more delta events you have, the more costly it becomes—especially if you have bad data across a large keyspace.

These strategies are really about making the best out of a bad situation. I won't mince words: *Bad delta events are very difficult to fix without intensive intervention!*

But let's look at each of these strategies in turn. First up, **build-forward** and then, to cap off this book, **rewind, rebuild, and retry**.

Build-forward: Undo bad deltas with new deltas

Deltas, by definition, create a tight coupling between the delta event models and the business logic of consumer(s). There is only one way to compute the correct state, and an infinite amount of ways to compute the incorrect state. And some incorrect states are terminal—a package, once sent, can't be unsent, nor can a car crushed into a cube be un-cubed.

Any new delta events, published to reverse previous bad deltas, must put our consumers back to the correct good state without overshooting into another bad state. But it's very challenging to guarantee that the published corrections will fix your consumer's derived state. You would need to audit each consumer's code and investigate the current state of their deployed systems to ensure that your corrections would indeed correct their derived state. It's honestly just really quite messy and labor-intensive, and will cost a lot in both developer-hours and opportunity costs.

However, you may find success in using a delta-strategy if the producer and consumer are tightly coupled and under the control of the same team. Why? Because you control entirely the production, transmission, and consumption of the events, and it's up to you to not shoot yourself in the foot.

Fixing delta-style events sounds painful

Yeah, it is. It's one of the reasons why I advocate so strongly for state-style events. It's so much easier to recover from bad data, to delete records (hello GDPR), to reduce complexity, and to ensure loose coupling between domains and services.

Deltas are popularly used as the basis of event sourcing, where the deltas form a narrative of all changes that have happened in the system. Delta-like events have also played a role in informing other systems of changes but may require the interested parties to query an API to obtain more information. Deltas have historically been popular as a means of reducing disk and network usage, but as we observed when discussing state events, these resources are pretty cheap nowadays, and we can be a bit more verbose in what we put in our events.

Overall, I recommend avoiding deltas unless you absolutely need them (e.g., event sourcing). Event-carried state transfer and state-type events work extremely well and simplify so much about dealing with bad data, business logic changes, and schema changes. I caution you to think very carefully about introducing deltas into your inter-service communication patterns, and I encourage you to only do so if you own both the producer and the consumer.

Hybrid events: Can I just include the state in the delta?

Hybrid events technically contain both state and delta, but in practice, they provide guarantees that are effectively identical to **state events**. Hybrid events give your consumers some options as to how they store state and how they react. Let's look at a simple money-based example.

Key	{ accountId: 6232729 }
Value	{ debitAmount: 100, newTotal: 300 }

Figure 2. An example of a hybrid event containing both the debit amount and the new total of funds

In this example, the event contains both the **debitAmount** ($100) and the **newTotal** of funds ($300). But note that by providing the computed state (**newTotal**=$300), it frees the consumers from computing it themselves, just like plain old state events. There's still a chance the consumer will build a bad aggregate using debitAmount, but that's on them—you already provided them with the correct computed state.

There's not much point in only *sometimes* including the current state. Either your consumers are going to depend on it all the time (state event) or not at all (delta event). You may say you want to reduce the data transfer over the wire—fine. But the vast majority of time we're only talking about a handful of bytes, and I encourage you not to worry too much about event size until it's costing you enough money to bother addressing. (And again, if you're REALLY concerned, you can always invest into a claim-check pattern.)

But let's move on now to our last fixing-bad-data strategy.

The last resort: Rewind, rebuild, and retry

Our last strategy is one that you can apply to any topic with bad data, be it delta, state, or hybrid. It's expensive and risky. It's a labor-intensive operation that costs a lot of people-hours. It's easy to screw up, and doing it once will make you never want to do it again. If you're at this point, you've already had to rule out our previous strategies.

Let's just look at two example scenarios and how we would go about fixing the bad data.

Rewind, rebuild, and retry from an external source

In this scenario, there's an external source from which you can rebuild your data. For example, consider an nginx or gateway server, where we parse each row of the log into its own well-defined event.

What caused the bad data? We deployed a new logging configuration that changed the format of the logs, but we failed to update the parser in lockstep. (Tests, anyone?) The server log file remains the replayable source of truth, but all of our derived events from a given point in time onwards are malformed and must be repaired.

Solution:

If your parser/producer is using **schemas and data quality checks**, then you could have shunted the bad data to a DLQ. You would have protected your consumers from the bad data, but delayed their progress. Repairing the data in this case is simply a matter of updating your parser to accommodate the new log format and reprocessing the log files. The parser produces correct events, sufficient schema and data quality, and your consumers can pick up where they left off (though they still need to contend with the fact that the data is late).

But what happens if you didn't protect the consumers from bad data and they've gone and ingested it? You can't feed them hydrogen peroxide to make them vomit it back up, can you?

Let's check how we've gotten here before going further:

- No schemas (otherwise would have failed to produce the events)
- No data quality checks (ditto)

- Data is not compactable and the events have no keys
- Consumers have gotten into a bad state because of the bad data

At this point your stream is so contaminated that there's nothing left to do but purge the whole thing and rebuild it from the original log files. Your consumers are also in a bad state, so they're going to need to reset either to the beginning of time or to a snapshot of internal state and input offset positions.

Restoring your consumers from a snapshot or savepoint requires planning ahead. (*Prevention*, anyone?). Examples include Flink savepoints, MySQL snapshots and PostgreSQL snapshots to name just a few. In either case, you'll need to ensure that your Kafka consumer offsets are synced up with the snapshot's state. For Flink, the offsets are stored along with the internal state. With MySQL or PostgreSQL, you'll need to commit and restore the offsets into the database, to align with the internal state. If you have a different data store, you'll have to figure out the snapshotting and restores on your own.

As mentioned earlier, this is a very expensive and time-consuming resolution to your scenario, but there's not much else to expect if you use no preventative measures and no state-based compaction. You're just going to have to pay the price.

Rewind, rebuild, and retry with the topic as the source

If your topic is your one and only source, then any bad data is your fault and your fault alone. If your events have keys and are compactable, then just publish the good data over top of the bad. Done. But let's say we can't compact the data, because it doesn't represent state? Instead, let's say it represents **measurements**.

Consider this **scenario**. You have a customer-facing application that emits **measurements of user behavior** to the data stream (think clickstream analytics). The data is written directly to a data stream through a gateway, making the data stream the single source of truth. But because you didn't write tests nor use a schema, the data has accidentally been malformed directly in the topic. So now what?

Solution:

The only thing you can do here is reprocess the "bad data" topic into a new "good data" topic. Just as when using an external source, you're going to have to identify all of the bad data, such as by a unique characteristic in the malformed data. You'll need to create a new topic and a stream processor to convert the bad data into good data.

This solution assumes that all of the necessary data is available in the event. If that is not the case, then there's little you can do about it. The data is gone. This is not CSI: Miami where you can yell "enhance!" to magically pull the data out of nowhere.

So let's assume you've fixed the data and pushed it to a new topic. Now all you need to do is port the producer over, then migrate all of the existing consumers. But don't delete your old stream yet. You may have made a mistake migrating it to the new stream and may need to fix it again.

Migrating consumers isn't easy. A polytechnical company will have many different languages, frameworks, and databases in use in their consumers. To migrate consumers, for example, we typically must:

1. Stop each consumer and reload their internal state from a snapshot made before the timestamps of the first bad data.
2. That snapshot must align with the offsets of the input topics, such that the consumer will process each event exactly once. Not all stream processors can guarantee this—but it is something that Flink is good at, for example.
3. But wait! You created a new topic that filtered out bad data (or added missing data). Thus, you'll need to map the offsets from the original source topic to the new offsets in the new topic.
4. Resume processing from the new offset mappings for each consumer.

If your application doesn't have a database snapshot, then you must delete the entire state of the consumer and rebuild it from the start of time. This is only possible if every input topic contains a full history of all deltas. Introduce even just one non-replayable source and this is no longer possible.

Bad data: Prevent it, repair it, rebuild it

In streaming, data is immutable once written to the stream. The techniques that we can use to deal with bad data in streaming differ from those in the batch world.

- First is to **prevent bad data from entering the stream**. Robust unit, integration, contract testing, explicit schemas, schema validation, and data quality checks each play important roles. Prevention remains one of the most cost-effective, efficient, and important strategies for dealing with bad data—to just stop it before it even starts.
- Second is **event design**. Choosing state-type event design allows you to rely on republishing records of the same key with the updated data. You can set up your Kafka broker to compact away old data, eliminating incorrect, redacted, and deleted data (such as for GDPR and CCPA compliance). State events allow you to fix the data once, at the source, and propagate it out to every subscribed consumer with little to no extra effort on your part.
- Third and final is **rewind, rebuild, and retry**. A labor-intensive intervention, this strategy requires you to manually intervene to mitigate the problems of bad data. You must pause consumers and producers, fix and rewrite the data to a new stream, and then migrate all parties over to the new stream. It's expensive and complex and is best avoided if possible.

Prevention and good event design will provide the bulk of the value for helping you overcome bad data in your data streams. The most successful streaming organizations I've worked with embrace these principles and have integrated them into their normal event-driven application development cycle. The least successful ones have no standards, no schemas, no testing, and no validation—it's a wild west.

Conclusion

A shift left is a rethink of data architecture. Take the very same work you're already doing in your data lakes and warehouses, and shift it to the left so that everyone can benefit from it. Shifting left eliminates duplicate pipelines, reduces the risk and impact of bad data, and lets you leverage high-quality data sources for both operational and analytical use cases. Shifting left is iterative and modular—you can choose to shift just one data set to the left, or you can choose to shift many, depending on your needs.

Multi-hop architectures—as popularized by medallion architecture—are inherently slow, expensive, and brittle. They rely on those who need the data to create and manage ETL pipelines that couple deeply into the source data domain, resulting in unexpected breakages whenever the source system changes its database models. Multi-hop architectures have been around for decades, but so have their problems.

Break-fix work is common, and a lack of trust propagates due to the unreliability of the data pipelines. Working defensively, each team tends to create their own data pipelines so that they can be sure of their data's provenance, leading to many similar-yet-different data sets. They result in significant duplication of data and can lead to the propagation of similar-yet-different data sets, a lack of trust in others' data, and loss of confidence.

Data products provide a powerful innovation in the data space. By building data products as part of a shift-left movement, you unlock both near-real-time and batch-based use cases all across your business. Your data product provides a single source of well-defined, high-quality, interoperable, and supported data that your systems and peers can freely consume, transform, and remodel as needed. Provide your data as a **Kafka topic**, an **Iceberg table**, or both to form the bedrock of a healthy modern data ecosystem.

A **headless data architecture** formalizes the interfacing between stream-table data products and your consumer and processing heads. It is the natural evolution of decoupling data from processing, with table formats such as Apache Iceberg providing the same sort of headless model that Apache Kafka has had since day one.

Establishing healthy data products as part of shift-left relies on **data contracts**. They provide the formal agreement of the form and function of the data product and its API to all of its users. They provide a barrier between the internal and external data models, providing the data producer users with a stable yet evolvable API and a well-defined boundary into other business domains. Users can browse through contracts and find the data product of your choice in the **Data Portal**, or register their own data products for others to use.

Flink, CDC, and Tableflow form a powerful triplet for realizing stream-table data products. Isolate your internal domain models, join with data from across your company, and eliminate costly, brittle, point-to-point pipelines. Rely on fully managed connectors, stream processing, and governance so you can focus on actually using your data instead of struggling just to make it usable.

Event design is a critical component of building stream-table data products. **State (fact) event models** are the best option for easily materializing your stream into a table via Tableflow or a connector. You can issue corrections and deletions so that your consumers don't have to independently identify, fix, and mitigate bad data. **Prevention** against bad data getting into your stream in the first place remains the best way to ensure that your shift-left initiative is successful.

The ultimate goal of shifting left is to provide clean, reliable, and accessible data to all who need it in your organization. It is ultimately a reduction in complexity, overhead, and break-fix work and will free you and your colleagues to work on other, more valuable problems.