Practical Data Mesh

Building Decentralized Data Architectures with Event Streams

Adam Bellemare
Foreword by Ben Stopford


ABWASIA.ORG

Practical Data Mesh

Building Decentralized Data Architectures with Event Streams

Adam Bellemare

Foreword by ***Ben Stopford***

Practical Data Mesh

Building Decentralized Data Architectures with Event Streams

by **Adam Bellemare**
Foreword by **Ben Stopford**

October 2022 First Edition

Table of Contents


ABWASIA.ORG

Foreword

The software industry has always been susceptible to trends. Some stick, some pass, but all vie for the limelight in one way or another. Agile, SOA, cloud, data lakes, microservices, DevOps, and event streaming have all had a fundamental effect on the software we build today.

Data mesh may well be the next innovation we can add to this list. I see it as a kind of microservices for data architecture. Part technology and part practice, this socio-technical theory aims to let large, interconnected organizations avoid putting all their data into one single place: a pattern that can lead to paralysis. In a data mesh, different applications, pipelines, databases, storage layers, etc., are instead connected through self-service data products, creating a network, or "mesh," of data that has no central point where teams are forced to wait in line or step on each other's toes. Such problems plague the monolithic application and the monolithic data warehouse alike.

Building systems that solve socio-technical problems like these has always been a bit of a dark art. I got this sense early in my career, working on data infrastructure in large enterprise companies. Investment banks, in particular, have complex data models with thousands of individual applications reusing the same common datasets: trades, customers, books, risk metrics, etc. Their architectures inevitably become unruly, and the natural instinct is to put everything in one place and install tight governance. While this can work, it stifles progress.

After moving to Silicon Valley, I was surprised to see younger companies facing the same problems, although being younger the effects were less pronounced. Two things stuck with me through these experiences. First, the approaches that worked best had some elements of being decentralized. Second, they emphasized getting the people part right, not just the technology.

While data mesh is a solid theoretical framework that touches both of these, it is essentially un-opinionated on what technology it should be implemented in. Event streams have emerged over the last five to 10 years as the primary implementation technology that fills this type of gap by embracing data in motion. In this book, Adam Bellemare has created a definitive resource that marries these two together. In fact, it may be the missing manual I always needed back in my enterprise days: a practical guide for building data systems that treats data as decentralized products and puts these products at the heart of your architecture.

This book also does a great job of making data mesh more tangible. Adam uses the data mesh principles as a set of logical guardrails that help readers understand the trade-offs they need to consider. He then dives deep into practical and opinionated implementation. This adds meat onto the bones of what is otherwise an abstract theoretical construct. I think this down to earth take will be valuable to architects and engineers as they map their own problem domain to the data mesh principles.

Of course, whether data mesh proves to be the next big trend that rocks the software industry remains up for debate, at least at the time of writing, but Adam's book takes you further than anyone has along this particular journey.

– Ben Stopford, Lead Technologist, Confluent

Introduction

Data mesh is a set of social and technological principles for designing modern data architectures. Data mesh makes data a first-class citizen by treating data sources as products, a key component for an organization's success. The data in a data mesh is easily accessible, interconnected across an entire business, and provides its users with the means to discover, access, and consume it reliably.

While the data mesh concept is relatively new, the problems it proposes to solve are not. This book covers the historical issues of data access, including why these issues remain relevant to this day. It examines how data mesh architecture can solve these historical problems and how event streams play into this modern data stack. In addition, it explores your options for building and designing data products served by event streams, and the necessary decisions you'll need to make for building your self-service tooling.

The principles underpinning the data mesh, while detailed and prescriptive, leave a lot of leeway for taking different approaches to the underlying implementation. For this reason, we have included a review of Confluent's opinionated data mesh prototype, explaining the design decisions and implementation choices we made in its creation. We also cover a data mesh implementation created by Saxo Bank, showcasing the design choices and architectural trade-offs that they decided upon.

Before we get into those, let's start with why data mesh is relevant. What are the main problems that we're still trying to solve today?

A Brief History of Data Problems

Data, as a discipline, is often treated as a separate domain from engineering. A data team composed of *data engineers, data scientists,* and *data analysts* extracts data from engineering systems and does "something useful" with it for the business. "Something useful" typically includes answering analytical questions, building reports, and structuring data from disparate systems into queryable form. An example of this might include correlating sales with various patterns of user behavior observed on a website. Making real-time product suggestions based on pages a user recently browsed would be a more contemporary example.

The data team usually functions in a highly centralized capacity. While the data engineers remain responsible for obtaining data from other systems, they do not own those systems or the data inside. Yet they must find the means to couple on the systems, extract the data, transform it, and load it (ETL) into a location for the data scientists to remodel. This data is typically stored in a large cloud storage bucket.

The data lake is a common form of analytical data storage. It contains vast amounts of unstructured and semi-structured data pulled in from all corners of an organization. Data scientists remodel, clean, and standardize the extracted data, before committing it back to the data lake as a set of clean or usable data, often according to a data-quality tier system (e.g., Bronze, Silver, or Gold quality data). The analysts access this data either by pointing their BI tooling to the specific area of the data lake, or by ETLing it themselves (or with the help of data engineers).

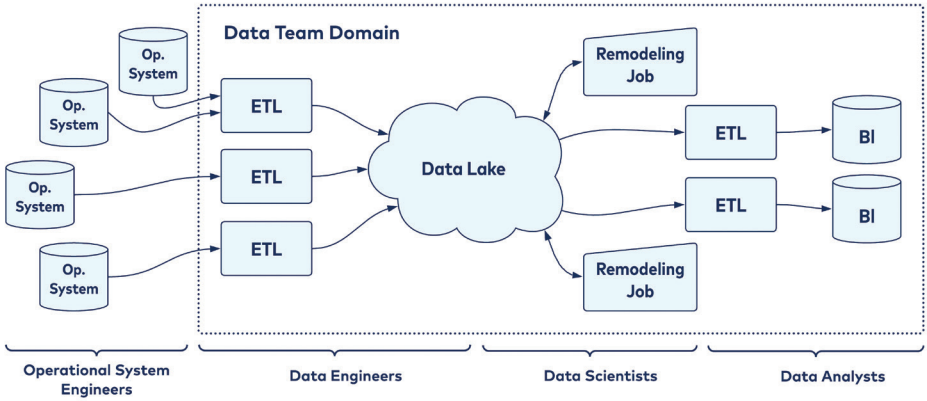

Figure 1-1. A typical data lake architecture, including the areas of responsibility for the various roles

Despite the notable success of this architecture in general practice, there are five clear areas where it falls short of the ideal: data quality, scalability, bottlenecks, cost, and timeliness.

Data quality is a broad-brush term, encapsulating the responsibility for ensuring clean, available, and reliable data. Traditionally, data quality is the responsibility of the centralized data team and not the team that initially created it. Pushing responsibility for data quality issues downstream was accentuated further during the rise of big data when practitioners were encouraged to *write unstructured data as-is and restructure it later with schema-on-read.* Schema-on-read was a popular way to market big data solutions, as it proved to be a low-effort way for teams to export data to the central data repository. However, this led to low quality and inconsistent data, and put extra work on the data team.

In practice, these perceived savings were often a false economy. Operational teams would effectively absolve themselves from the data needs of downstream users, particularly for analytical purposes, offloading that responsibility squarely onto the data team. Unfortunately, this shift in responsibilities proved extremely difficult to execute well in practice.

The second major hindrance related to the traditional data approach is in scaling the architecture. ETLs are notoriously brittle, largely because of tight coupling to the source system's internal private data model. Internal

model changes, data schema changes, configuration changes, and credential changes are only a few common failure modes. Any breakage in an ETL job ripples downstream to the data scientists and analysts trying to use the data, resulting in dashboard outages, delayed reports, incorrect insights, and stoppages to machine-learning model training and deployments. Scaling becomes even more difficult as the number of datasets grows, because each new dataset increases the fragility and complexity of the dependency graph.

Third, the traditional data team forms a bottleneck. They are responsible for extracting, standardizing, and cleaning data, deriving and clarifying relationships between datasets from multiple domains, and formulating and serving data models to a wide array of end customers. They are also responsible for building, maintaining, and fixing ETL jobs, a highly reactive process when combined with their lack of ownership over the source domain models. The data team often spends significant time tracking down failures and remediating incorrectly computed results. The lack of official dataset ownership makes it difficult to pinpoint who is responsible for making data available to other teams. One quantification from the IDC suggests that data analysts waste about 50% of their time simply trying to identify or build reliable sources of data.

Fourth, when using traditional data, cost is often also an issue. While historically you may only have had a few data sources, data-centric companies typically have hundreds of data sources, each with its own dedicated, yet fragile, ETL job. The competitive needs for data have only increased, with daily jobs replaced by hourly jobs, and hourly jobs replaced by event stream processing.

This leads to the fifth issue with the traditional approach: timeliness. Real-time data is better than batch. Data is increasingly being generated in real time by mobile phones, servers, sensors, and internet-of-things devices. Our customers expect us to react to this data quickly and efficiently. To do so, we must shed the unresponsive batch processing in favor of real-time, stream-processing solutions.

Data warehouses exemplify a very similar approach to data lakes: Extract data, transform it, and load it into storage for analytical querying. But unlike data lakes, data warehouses require a well-defined data model and schema at write time, which is usually defined and fulfilled by the data team. While this strategy may resolve many data quality issues, the onus of maintaining a full, canonical data model, and its accompanying ETL jobs, remains with the data team, and not with those who own the production of the upstream data. Scaling, bottlenecks, cost, and timeliness remain an issue, along with maintaining a data model stitched together from independently evolving, disparate datasets from across the organization.

Thus, centralized data lake and warehouse architectures, while successful, are not perfect. Access to real-time, high-quality data from across an organization remains a challenge. Decentralized architectures have become much more popular recently, with microservices providing a great example. And much like microservices provide an alternative to centralized, monolithic business applications, data mesh provides an alternative to the traditional centralized data architectures embodied in data warehouses and data lakes.


ABWASIA.ORG

The Principles of Data Mesh

Data mesh is based on four main principles: **data as a product, domain ownership, self-service,** and **federated governance.** While each of these principles is interrelated and plays an essential role, treating **data as a product** is a fundamental shift in how organizations create, store, and communicate important business data.

Data mesh moves the responsibility of providing reliable and useful access to data back to the data's owner from the centralized data team. Data is no longer treated as an application's byproduct, but instead is promoted as a first-class citizen on par with other products created and used within an organization. This requires a shift in responsibilities with respect to how data is created, modeled, and made available across an organization.

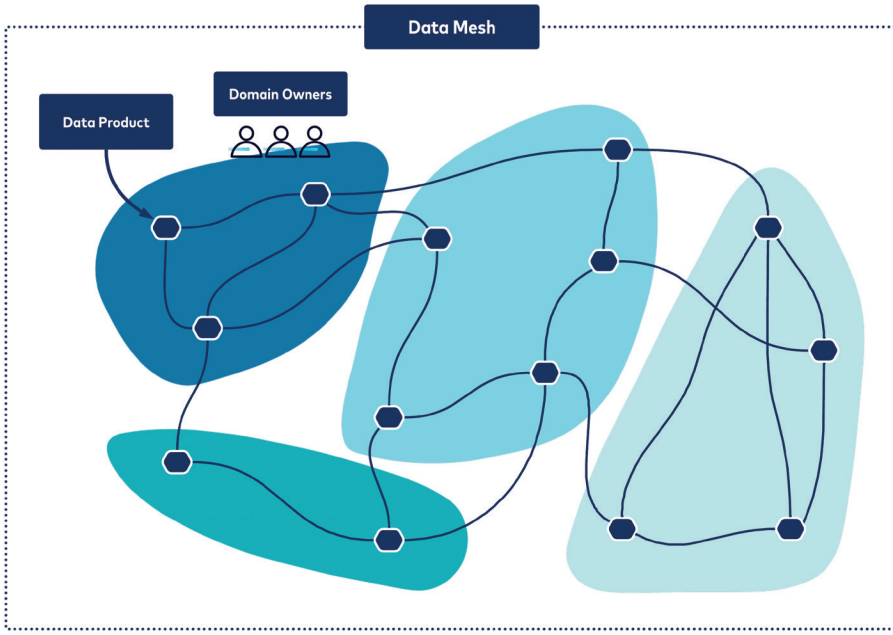

Figure 2-1. A data mesh connected across multiple business domains

Each domain can choose which data to expose to the others through well-structured and formally supported data products. By coupling data product ownership to its source domain, we can apply the same rigor to data products that we do to other business products. Consumers can simplify their dependencies and eliminate coupling on internal source models, instead relying on data products as trustworthy mechanisms for accessing important business data.

Let's take a closer look at the four main principles of data mesh.

Data as a Product

Each team is responsible for publishing their data as a fully supported product. That team must engage in product thinking about the data they're serving: They're wholly responsible for its contents, modeling, cohesiveness, quality, and service-level agreements. The resultant data products form the fundamental building blocks for communicating important business data across the company.

Under the hood, a data product consists of a subset of domain data, the code for creating and serving the data, and the infrastructure components supporting its creation and delivery. Because data products are multimodal, you may have the same data product served by different delivery mechanisms—via a SQL-queryable dataset, a synchronous API, or an event stream. All of the code used to compose and deliver the data belongs within the boundary of the data product.

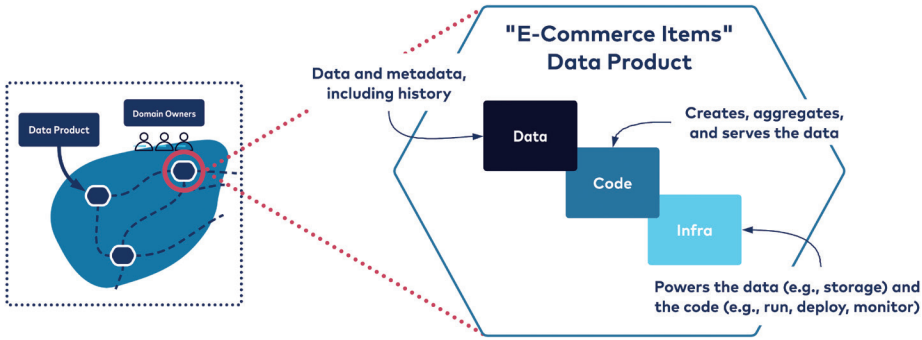

Figure 2-2. A data product contains data, code and infrastructure components

Be careful about selecting the domain data to include in the data product and do not share any internal private state. Determining what data ends up in a data product requires consulting both known and prospective consumers. It is also common to create multiple data products within a single domain, each serving the unique needs of its consumers.

Data Ownership by Domain

In much the same way that microservices each own a specific business function, data products are composed around particular business domains. Access to that data is decentralized: There's not a central data bureau, a data team, an analytics team, etc., holding that data in captivity. Instead, the data lives in its domain, and it is from the boundary of that domain that you can access it as you need for building your business solutions.

Domain owners are also responsible for ensuring that data products are discoverable and documented. This typically involves registering data products with a discovery service and providing additional information such as metadata, service-level agreements, and domain-specific semantics. Explicit scoping of the data product context is essential. It is not uncommon for two domains to have similar-yet-different definitions of certain elements (e.g., a User may mean slightly different things in two distinct domains).

Self-Service Data Access

Data products are readily available everywhere in the company by self-serve means. Governance and control of information remain a factor, but in principle, once a data product is published, it can be immediately used by any domain.

For example, if you're producing a sales forecast for Japan, you could find all of the data you need to drive that report—ideally in a few minutes. You'd be able to quickly get all of the data you need, from all of the places it lives, into a database or reporting system that you control.

Alternatively, perhaps you're computing real-time interest in products to optimize your advertisements selection. Data discovery services can lead you to the data you need, letting you select the products to consume as inputs into your prediction model.

Federated Governance

Governance and control of your data remain a concern when it's made widely available across your organization. It's a collaborative affair, with individuals from across the organization joining together to create appropriate standards and policies. Shared responsibilities include outlining info security policies, data handling requirements, and right to be forgotten legislation. Another part of it is balancing a data product owner's right to autonomy versus the ease with which consumers can access and use the data.

A formal federated governance committee also provides recommendations and requirements for interoperability, self-service tooling, and metadata requirements. Implementation of governance policies occurs at multiple levels across the organization. For example, requesting and managing access to data products may be handled through standard self-service tooling, while enforcing info-security policies requires a commitment from each data product owner.

The Relationships Between Principles

Each data mesh principle relates to and influences the others. The federated governance group defines officially supported data product formats, which guide the features of the self-service platform. Data product owners rely on this platform to create, publish, and manage their data products. For example, serving a data product via a SQL API may be officially supported and streamlined with self-service tools, while serving it via a gRPC API may not.

In the next section, we'll look at creating a data product from an existing domain, illustrating the interrelationship of the data mesh principles. We'll cover domain boundaries, modes of data product access, and governance, and we'll also discuss the role of event streams in data mesh.


ABWASIA.ORG

A data product is composed within its domain and output across its boundary for other teams to consume and use. Figure 3-1 shows the components for building a data product.

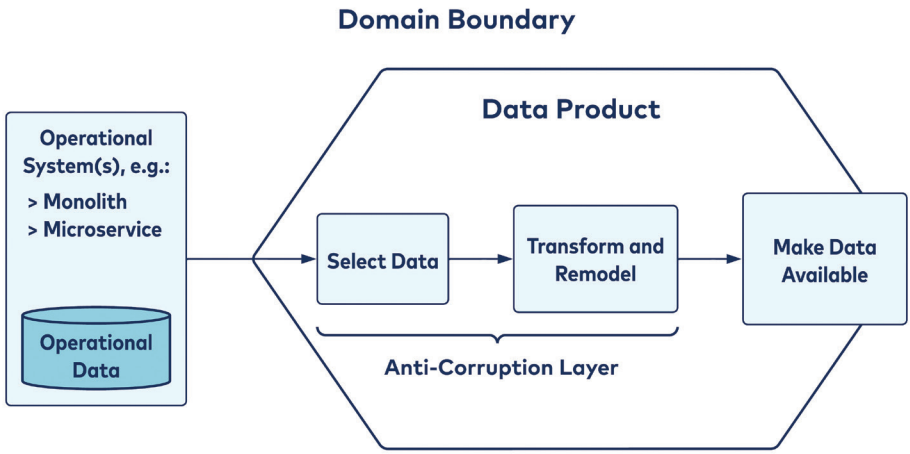

Figure 3-1. Components of a data product

From left to right, data is extracted from one or more operational systems, then transformed and remodeled according to the data product's requirements. Finally, it is made available in a format suitable for consumer use cases. The extraction and transformation steps form an *anti-corruption layer* that isolates the internal data model from the data published to the external world. Consumers coupled with the data product API need not concern themselves with internal model changes. This significantly contrasts the frequent breaking changes inherent in the responsibility model of a centralized data team.

Data products are first and foremost meant to serve *analytical use cases*. This means selecting the necessary operational data and remodeling it into *facts*. These facts comprise a record of precisely what happened within that domain's operations, without directly exposing the internal data model of the domain.

Three key characteristics are common to all data products:

- **Facts are read-only:** Only the data product owner can add new facts to the data product.
- **Facts are immutable:** Data product owners can append new facts as an addendum to previous facts, but cannot modify, overwrite, or delete them.
- **Facts are timestamped:** Each fact contains a timestamp representing when it occurred, such that time-based ordering is made possible.

Modifying already-written data is prohibited as it can invalidate results for any consumer that has read and used the data. A silent correction of existing data, without reprocessing by the consumers, will often lead to divergent computations between each consumer. Thus, corrections are published and propagated as part of new entries into a data product.

The three key characteristics of facts—that they are read-only, are immutable, and are timestamped—provide the essential property of **reproducibility**: A single consumer can read and reprocess the same data for a given time range as many times as it chooses, and will always yield the same result. Similarly, two independent consumers can access the same data range at different points in time and obtain consistent results. Consumers can leverage these properties to produce projections or materialized views of the data to serve their own specific business needs.

Source, Aggregate, and Consumer Data Products

Data products are built with an alignment to a source domain, to an aggregate of domains, or to a consumer domain. Each of these subtypes provides its own benefits and trade-offs.

Source Aligned

Source-aligned data products reflect business facts generated by an operational system. For example, a storefront domain may emit click facts containing the full set of details about each product that a user clicked on.

Aggregate Aligned

An aggregate-aligned data product consists of data from one or more upstream data products. An example would be a daily aggregate of clicks per e-commerce item enriched with each item's category classification.

Consumer Aligned

A consumer-centric data product is built to serve a highly specialized use case for a given consumer.

The three alignments make different trade-offs regarding the effort involved in creating them, their scope of application, and the effort involved by the consumer in applying them. At one end of the spectrum are source-aligned data products, which are general purpose and typically require only simple business logic to construct. Consumers can take these source-aligned facts and apply their own business logic to remodel and transform the data to suit their needs.

Figure 3-2 shows two consumer-aligned examples: a) Item Entity data extracted and converted into an event stream, and b) click events collected from client endpoints, filtered of personally identifiable information (PII), and converted into an externally facing event stream.

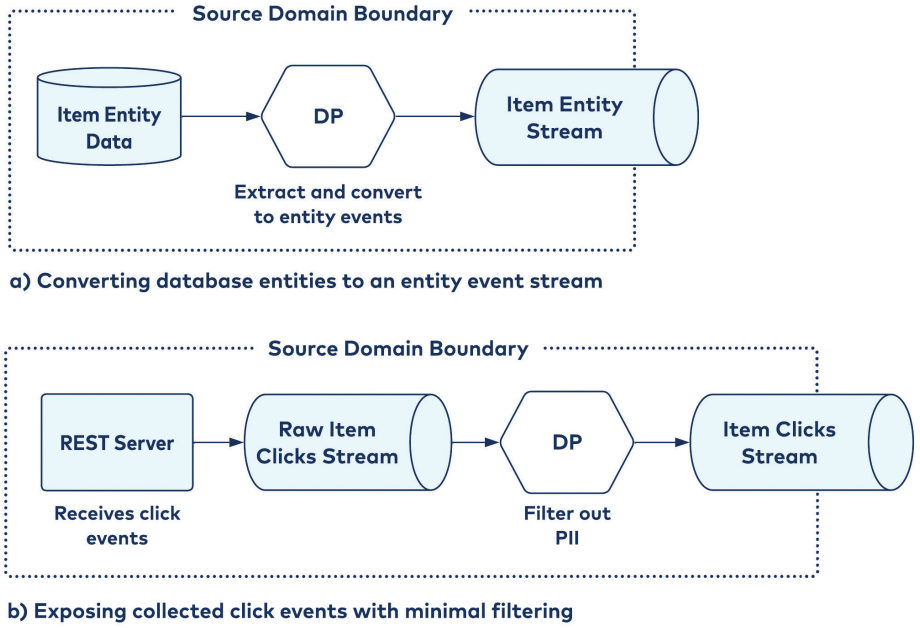

Figure 3-2. Creation of source-aligned data products

In the middle of the data product spectrum are data products that are built around an aggregation for its consumers, and may or may not include enrichment and denormalization of data. Figure 3-3 shows the aggregation of item clicks into a daily count, joined on item entity data for ease of use by downstream consumers.

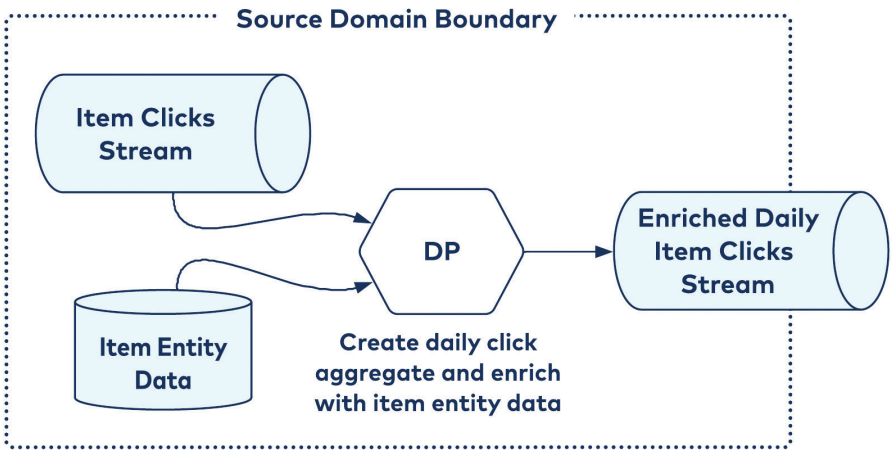

Figure 3-3. Creation of an aggregate-aligned data product

At the far end of the spectrum are consumer-aligned data products, which frequently rely on data products from multiple disparate domains, combined with consumer-specific business logic. Consumer-aligned data products tend to be so specific that they are only useful for the consumer domain, and so they reside within that domain under the consumer team's responsibility. Figure 3-4 shows the composition of a Daily Gross Sales Volume data product for the accounting domain, sourced from data products published in the sales and finance domains.

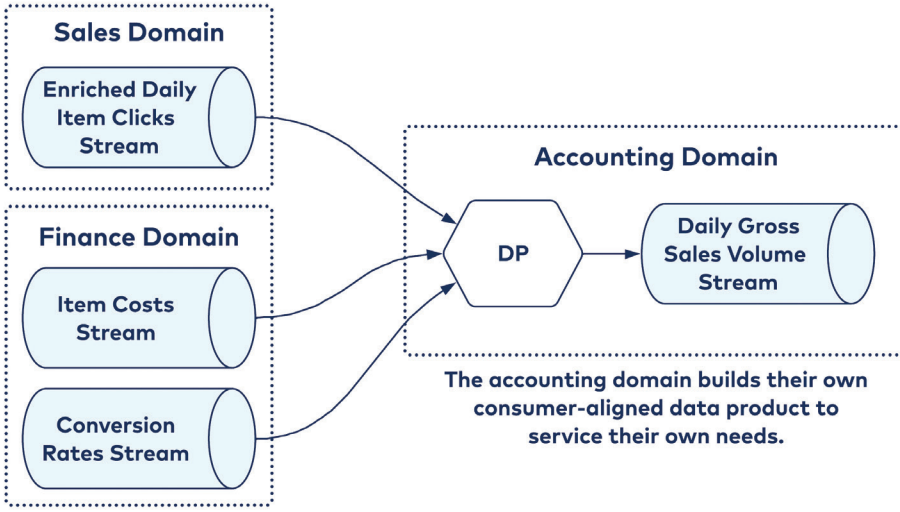

Figure 3-4. Creation of a consumer-aligned data product

A data product owner may publish multiple, similar-yet-different data products, each suiting a different consumer need. For example, several disparate consumers may be accessing the source-aligned click data product, with each consumer subsequently computing a daily aggregate using its own business logic. In this case, it would make sense for the storefront domain to go ahead and perform the aggregation work, producing a second data product of daily-aggregated clicks per e-commerce item. Creating an aggregation data product saves significant cumulative effort and eliminates the possibility that one of the consumers computes the aggregate incorrectly.

While consumer needs should drive the domain alignment of data products, data product owners must balance needs against available resources. While a simple consumer-aligned data product may be reasonable to implement, another may be too complex for the domain owner to support. In this case, it will be up to the consumer to build and support a consumer-aligned data product to fulfill its needs.

Accessing a Data Product

Data products are inherently multimodal: There are many different ways by which a data product can be made available to external consumers, provided the data is both **timestamped** and **immutable**. Selecting an access mode is a function of the needs of your consumers, your self-service infrastructure support, and the officially sanctioned modes determined by your governing body.

A big data compatible mode is the most common way to access a data product. An example would be a set of Parquet files written to S3 for periodic access via an analytics job. Another would be a set of domain events published to an event stream. Other options could include accessing the data product through SQL queries, REST APIs, or gRPC.

A single data product can be served through many different modes. Figure 3-5 shows a single data product served via three separate mechanisms:

- A SQL API serving data from an underlying relational database (note: this data represents business facts, not relational operational data)
- Cloud storage, where the data product is updated based on hourly batch jobs
- A durable event stream for real-time consumption

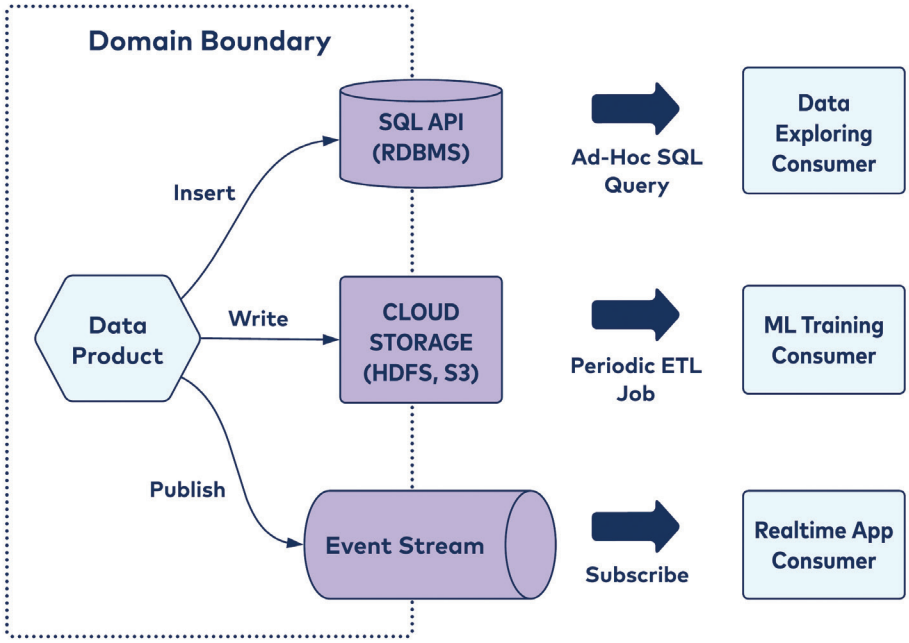

Figure 3-5. Data mesh supports multimodal data products

Selecting a mode in which to publish a data product to the wider world is largely up to the data product owner, though the set of available options will be heavily influenced by the federated governing body and self-service tooling. At the same time, a single access mode is unlikely to fit the needs of every data product producer and consumer. A balance is inevitably required, and where that balance lands will differ from one organization to another.

However, event streams are unique among access modes. In contrast to periodically querying a set of Parquet files or a SQL interface, the consumer instead maintains a persistent connection to an event stream, receiving new events as they are published. Event streams provide real-time access to important business data not only within a domain but across them too.

Figure 3-6 shows multiple domains and data products, communicating important business data both within and between domains—using event streams. Two of these domain owners have chosen to use event streams to communicate both internally and externally, while the others only use event streams for external communication.

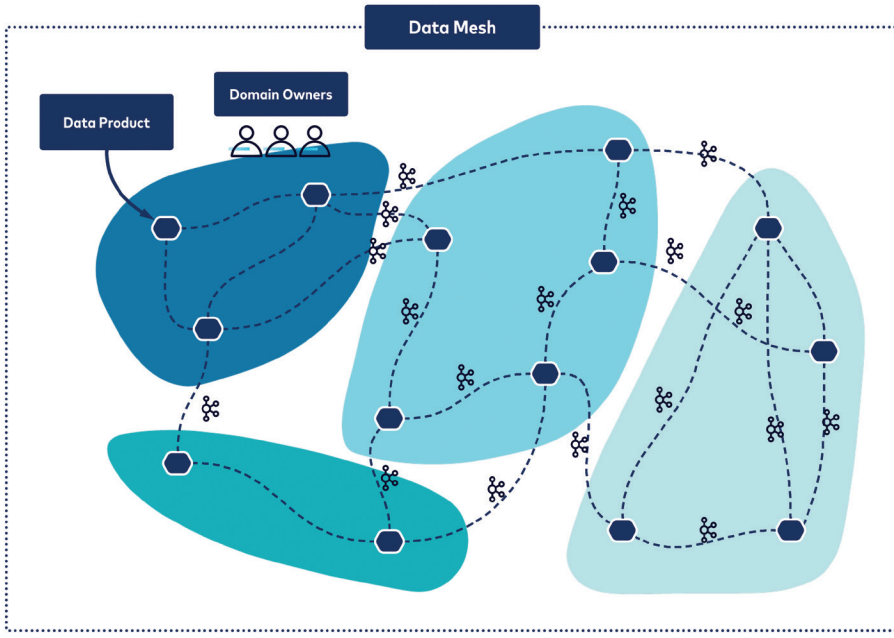

Figure 3-6. Multiple domains and data products, communicating data within and between domains

Using event streams as the primary mode of data product access unlocks several powerful options, which we'll look at in detail in the next section.

Presenting Data Products as Event Streams

Though there are many ways to provide access to data products, the push semantics of event streams make them the ideal choice. An event stream in the scope of this paper is defined as an Apache Kafka® topic, a well-defined schema (Avro or Protobuf), and accompanying metadata.

A Kafka topic is an immutable log of events in the order in which they occurred, where new events can only be appended to the end of the log. Events cannot be modified once written. Each event contains an optional **key** and a **value**: the key is typically a unique identifier of an entity, used as a partition key to ensure that all data of the same key is written to the same partition. This provides data locality and scalability for consumers,

such that a single consumer instance can be used to process each partition. In the following example, you can see the partitioning at work for a very basic fruits topic, where the "Apple" value for key 100 is updated to "Red Apple" following the appending of a new event to partition 0.

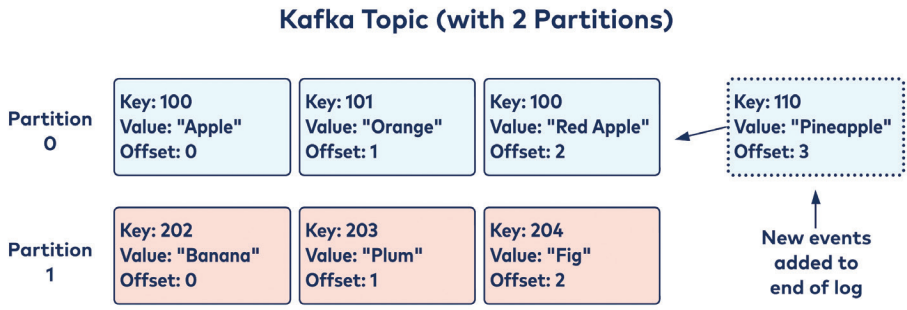

Figure 3-7. A Kafka topic with two partitions, showcasing the append-only update of Key 100 from "Apple" to "Red Apple"

If you would like to learn more about the fundamentals of Kafka, including partitions, topics, production, and consumption, please check out Confluent's Kafka 101 course.

There are several major factors that make event streams optimal for serving data products:

- **They are immutable and append-only:** Event streams communicate a series of occurrences in the form of stateful events. Data pertaining to any given entity can only be updated or modified in the form of a new event. This provides the basis for eventually consistent communication between the event stream publisher and each of its consumers.

- **They bridge the operational/analytical divide:** Event streams enable all forms of operational, analytical, and hybrid workflows by providing a single data product source. Streaming use cases can consume directly from the event stream, while analytical use cases can sink the data to a batch query system. This addresses a long-standing problem whereby analytical and operational data is often served from different sources. For example, orders are created in an orders service and queried via a REST API, but the same orders are also available in a data warehouse for reporting. This can lead to hard-

to-solve inconsistencies and divergent sources of truth, resulting in a loss of trust. In contrast, event streams provide a common source of data for both operational and analytical use cases.

- **They get data where it needs to be, in real time:** Event streams are a scalable, reliable, and durable way of storing and communicating important business data. Streams are updated as new data becomes available, propagating the latest data to all consumers. Real-time data enables both streaming and batch use cases, including those that span data centers, clouds, or geographies.

- **They empower consumers:** Consumers can independently remodel event streams, for example mixing multiple data products to develop domain-specific use cases. You might pull in customers and orders data from their respective data products, but then embellish the data with your own customers categorization schema. You can then store the enriched data in a data store best suited for your query patterns, such as a key-value store for fast, customer-based lookups. Figure 3-8 shows an example of what this could look like in practice.

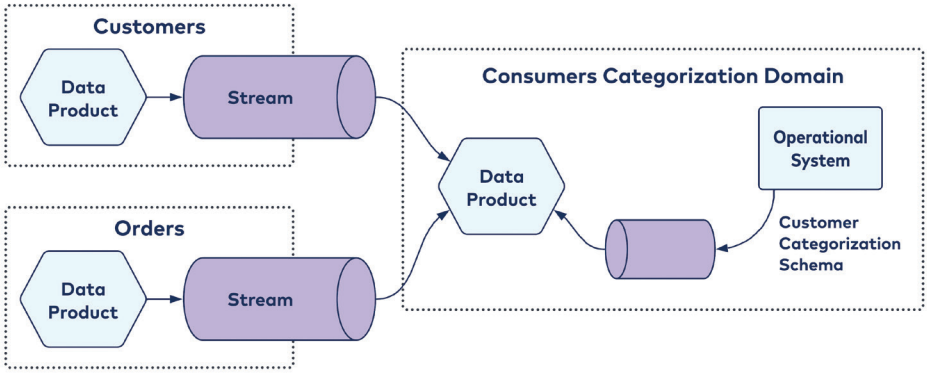

Figure 3-8. Sourcing customer and order data through event streams

Event streams provide data product consumers with the ability to use the latest events, or to rewind to a specific point in time and replay events. New consumers can begin from the start of the event stream and build up historical state, while existing consumers can replay events to account for bug fixes and new functionality. Each consumer autonomously controls its stream consumption progress.

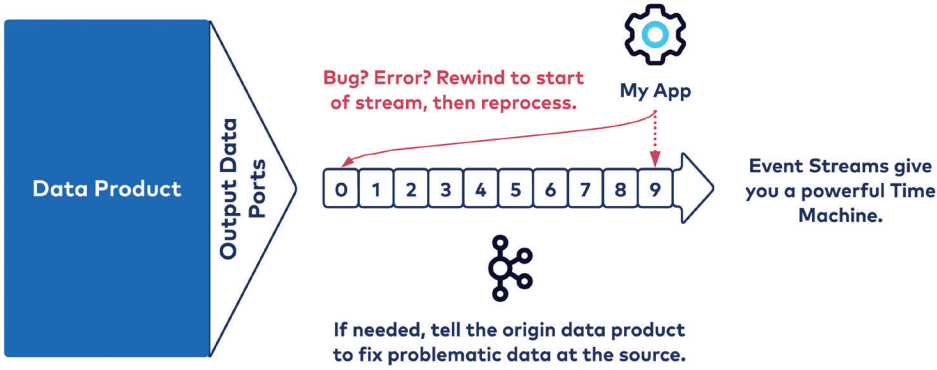

Figure 3-9. Rewind and replay events as needed

Consumers can debug and inspect event stream contents using precisely the same consumer interface. For example, a producer of an event stream may want to validate the contents of what they have written. In contrast, consumers may wish to obtain actual sample data to validate their expectations. Inspecting the contents of an event stream can be done using a stand-alone application like kcat or by leveraging built-in SaaS UI, such as Confluent's message viewer.

Event streams play a unique role in data mesh because they provide an opportunity for the consumer to have access to a stream of business facts. Source-aligned and aggregate-aligned event streams offer the opportunity for real-time reaction, and provide enterprising consumers with an easy to use, general-purpose data product API they can use to build their own use case specific models featuring personal business logic.

This approach has a trade-off: leaving data remodeling and interpretation to the consumer may lead to inconsistent interpretations. But clear documentation, including a detailed schema and metadata, can help prevent accidental misinterpretation. Depending on the use case, a difference in understanding may be inconsequential—or it can cause problems, such as when two consumers of the same data, interpreted in similar yet divergent ways, try to merge their results.

For example, two different consumers may compute daily sales inconsistently: one may base their computations on UTC-0 time, while the other uses UTC-5 time. One option to prevent this scenario is prescriptive governance (e.g., time-based aggregations must use UTC-0). A second option is to issue a feature request to the data product owner, requesting that they compose an aggregate-aligned data product for both consumers to use.

The data product owner could choose to generate an aggregate-aligned event stream of daily sales to fulfill this request. The product providers take on the work of adapting their data so that consumers don't have to repeat that work in each of their own domains. But, in working with the consumers, the data product owner may choose to publish the data instead to another mode of access, such as a set of Parquet files in cloud storage.

Regardless of how the event stream data product is aligned, the need to have a well-defined schema to enforce the data format remains, along with metadata to describe it.

Event Schemas and Metadata

An event schema serves several purposes. First and foremost, it defines the contents of each event, providing a well-defined data definition for consumer coupling that is separate from the producer's internal data model. It also provides a framework for evolving and changing the public data model over time as the needs of the business change, as well as a means of enforcing schema validation at runtime, to prevent bad data from being written to a topic. Schemas and their associated metadata enable a platform for data product discovery, registration, and tracking, and also allow the assessing of consumer interest.

Apache Avro and Google's Protobuf are the leading options for event stream schemas. The value of an event must have explicitly defined schemas to simplify discovery, validation, documentation, and code generation. An event's key may also have a schema, or it may use a primitive type if the key is a simple, unique identifier. Schemas are stored in the Schema Registry and comprise part of the metadata required for discovering the contents of the data product.

Metadata contains information about the data product, including the owner, the service-level agreement, a timestamp of the last published event, a quality rating, information about the number of events and size of the data product, and compaction information, to list a few. You may also choose to include metadata about security-related subjects, such as the presence of personally-identifiable information (PII) and financial details. The precise metadata of your data product will vary depending on your business needs.

The event schema and metadata are published to a central location as part of the data mesh, enabling a standard mechanism of data discovery, formalized schema evolution, and enforcement of federated requirements.

What About Non-Streaming Data Products?

The vast majority of data products can, and should, be served through event streams. Source-aligned and aggregate-aligned data products can easily be provided through event streams, providing fundamental building blocks for consumers to use as they see fit. Event streams are exceptionally well suited to a vast range of data domains, given their incrementally updating nature, their ease of consumption, and the freedom they provide to mix streams of disparate data products. Simply put, event streams are the best option for serving your data products, particularly when multiple teams require real-time, replayable data access.

But data mesh is multimodal, and other interfaces may better serve some data products. Further, you can serve the same data product through both an event stream and another interface, since using one does not preclude you from the other.

Your data product APIs will depend heavily on your organization's existing tech stack, the resources you have dedicated towards data mesh, and the needs of your data users. Your organization may be at a stage where it only needs a daily aggregate of user analytics data, met with a nightly export to cloud storage for batch computation. With no real-time use cases, you may find it unnecessary to invest in event-driven data products until such use cases arise.

Non-streaming data product APIs may include:

- **Cloud file access,** such as Amazon's S3 hosting a set of Parquet files. Organizations with extensive investments in existing batch processing frameworks can reap the benefits without retooling their platforms.

- **Human-readable data,** such as weekly and quarterly financial reports, or any sort of human-accessed system with low load and the need for a friendly UI.

- **A geolocation-based API** that serves queries related to users and their locations at a given time. The extreme similarity and narrowness of query patterns in this scenario may make it worthwhile to provide this as a single queryable data product for all consumers.

Our experience shows that the ever-increasing need to communicate data, at scale across an organization, means that event streams will play an essential role in data mesh implementations. The ability to deliver updated data, in real-time, across a network of connected data mesh consumers has proven to be an invaluable step in data communications.

Data mesh provides producers and consumers with a scalable way to communicate important business data between domains. But to do this well depends on self-service tooling for creating, publishing, discovering, and using data products.


ABWASIA.ORG

Making Data Products Self-Service Using Kafka

A self-service platform is essential for implementing a data mesh, providing self-service capabilities to data product owners, producers, and consumers alike. Guided by the requirements of federated governance, self-service platform capabilities must make it easy for all participants in a data mesh to publish, access, and use data products across the organization.

The data mesh self-service implementation you build depends on your own unique needs, existing tooling, and governance choices. While there is no single correct solution, you'll know you're on the right track when your data mesh users can do the following:

- Create a data product access point, define a schema, set up access controls, publish, and deprecate data products with just a few button clicks.
- Transform domain data into a data product by focusing only on the business logic, relying on the self-service platform for infrastructure, durability, scalability, and monitoring.
- Explore, discover, and evaluate data products using a schema registry or stream catalog.
- Register consumers with read-only access using self-service controls.
- Consume events into your business logic for either operational or analytical purposes.

Using Kafka topics as the basis for your data products relies heavily on stream processing and stream connectors. While there are multiple technologies you can choose to support building data products, here are the two that we have found to be essential for creating and consuming *event stream* data products:

- **Kafka Connect,** as a means of integrating non-streaming systems with event streams.
- **ksqlDB** for consuming, transforming, joining, and using event streams to drive business logic and create new data products.

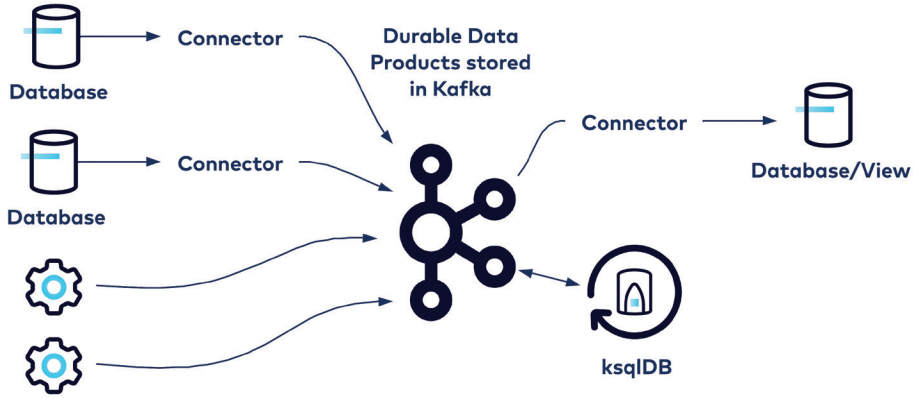

Figure 4-1. Kafka Connect and ksqlDB make it easy to use event streaming data products

Let's review the roles of each in providing self-service capabilities.

Kafka Connect Source Connectors

Kafka Connect source connectors provide the means of extracting, transforming, and loading data from an operational system into an event stream. The connectors serve as the anti-corruption layer, isolating the internal operational system from the data provided to the event stream.

While you can download Kafka Connect and run your own cluster, Confluent makes self-service a reality by providing it as a service. Data product owners specify the data sets they want to extract in the connector configuration, including targeted table and column names, SQL queries, or other database-specific query options. The connector is automatically scaled up and down depending on resource needs. All you need to do is choose from the large selection of managed connectors, provide your configurations, and start it up.

Figure 4-2 shows a query-based connector. A SQL query is periodically executed on the target database, with the results packaged up into events and emitted to Kafka topic.

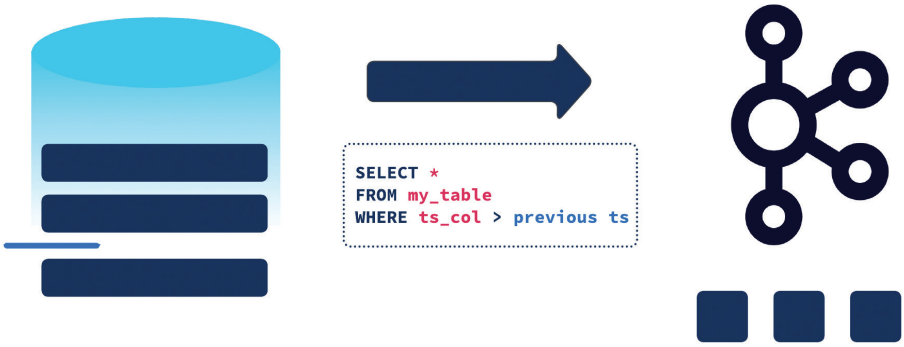

Figure 4-2. A query-based Kafka connector

Sourcing Data Using Change-Data Capture

One popular mechanism of extracting data from an operational system is change-data capture. Events are captured from the transaction log of the database system, completely forgoing the use of the query interface. This connection method significantly reduces the load put on the source database and provides access to change events with very low latency. Figure 4-3 shows transactional logs from the database extracted into events stored in a Kafka topic.

Figure 4-3. Extracting events from the transaction log into Kafka

Debezium is an example of a popular and robust CDC connector that works natively with Kafka Connect.

Sourcing Data Using the Outbox Pattern

The outbox pattern is another popular option for producing a data product. For consistency, this pattern requires that the domain database support transactions.

To use it, you create a dedicated new table (the outbox) to store events temporarily. Next, you modify your code by wrapping selected database modifying commands within a transaction. Thus, a corresponding record is also published to the outbox whenever you insert, update, or delete data. The following pseudocode shows a user insertion wrapped in a transaction, such that the outbox is also populated:

Before Adding an Outbox	After Adding an Outbox
user.insert(username, age, pin, country, …)	Transaction.start() user.insert(**username**, age, pin, **country**, …) outbox.insert(**username**, **country**) Transaction.commit()

The transaction ensures that any update made to the internal domain model is reflected in the outbox table on a successful commit, and correctly rolled back in the case of a failure. You may also have noticed that the outbox does not write the user.age or user.pin. This is a deliberate choice: the outbox table acts as an anti-corruption layer, isolating the internal domain model from the data product model.

Once data is in the outbox, a process must write it into the corresponding Kafka topic, and self-service Kafka Connect is the easiest way to do this. You can set up a CDC connector to tail the database log or a periodic query connector to query the table for the latest results. After emitting the captured data to the event stream, be sure that you delete it from the outbox table to keep the disk usage low and to avoid republishing old data.

Temporary Stream Storage and Push-Button Republishing

Push-button republishing of domain data enables bootstrapping of historical data into a new consumer. The option is complex and error-prone, and is usually only selected due to the inability of the event broker to support infinite retention.

The premise is that the event stream does not contain the entire history of data but only a tiny recent subset. A new consumer who wants the whole data history would simply start up their application, then push a button in a UI to have the source domain republish the entire dataset.

There are several reasons that we recommend using infinite event stream retention instead:

- Republishing places a very high load on the data source, which may be unacceptable for production systems.
- There may be an inability to create the historical data due to a lack of retention in the source system.
- Existing consumers will also receive all of the republished events, resulting in unwanted side effects and inconsistent state.

A variation of this pattern instead republishes the push-button data into a new dedicated stream. The consumer reads this history, then swaps back to the main event stream once its backfill is complete. Though this removes the disruption to existing consumers, it introduces significantly more complexity, while failing to remove the overhead load on the producer.

Instead, it is best to retain all data product facts in the event stream as long as they remain relevant to the business. Consumers simply reset their offset to the beginning of the event stream to access the history.

Materializing Event Streams into Databases

An event stream can be composed into a materialized view by consuming, transforming, and loading each event into a database. One common form of materialization is to write each event into a key-value store, overwriting old key values with new ones as they arrive. Figure 4-4 shows an example of materialization.

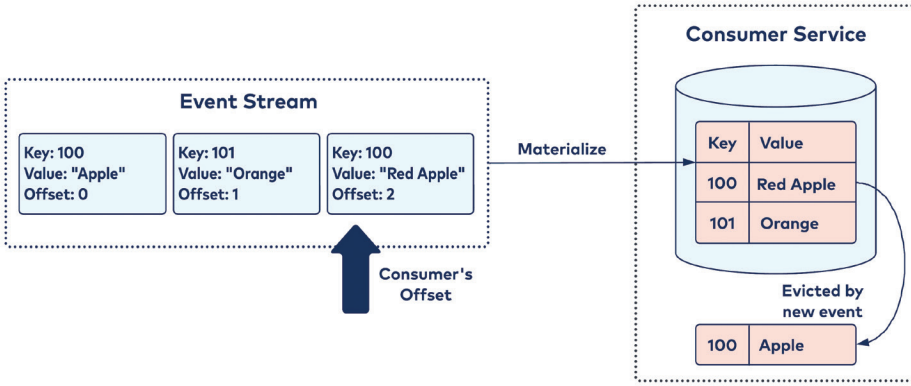

Figure 4-4. Materializing an event stream into a consumer's database

In this example, the consumer has fully materialized the event stream into its domain boundaries. In materializing (key: 100, value: "Red Apple", offset: 2), the consumer service simply overwrote the existing entry, evicting (key: 100, value: "Apple", offset: 0).

Consumers can materialize event streams from any number of data products, selecting, mixing, and merging data from multiple sources. Every consumed event can trigger business logic, create or update aggregates, and contribute to an eventually consistent, read-only state.

Materializing to Existing Databases with Kafka Connect

Kafka Connect makes it easy to consume event streams, transform them, and write them to a wide range of endpoints. Purpose-built sink connectors provide self-service options for getting the data product data into your domain. Need to write an event stream into a JDBC database? No problem. Want to sink it into S3? Kafka Connect has you covered.

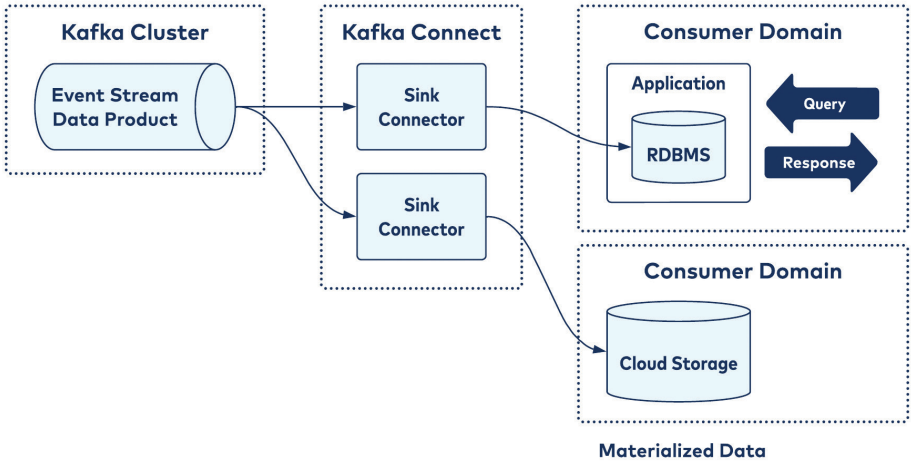

Figure 4-5. Using Kafka Connect to materialize into both an RDBMS and cloud storage

Kafka Connect also provides essential self-service tooling to reduce the difficulty of using data products. Consumers gain the autonomy to select the event streams and sink connectors they need, in order to ingest and materialize data products into their domain. Businesses can continue to use their existing business intelligence tools, data warehouses, data marts, and data lake deployments without being forced to adopt stream processing.

The sink connector converts the event stream into the format required by the existing systems. This native ability to power both real-time streaming jobs **and** offline batch jobs is one of the most significant advantages of publishing data products to event streams.

Materializing with ksqlDB Native Streaming Applications

ksqlDB is an example of a native, event-driven framework that supports materialization out of the box. It relies on the principles of the Kappa Architecture, as first described by Jay Kreps, co-creator of Apache Kafka and co-founder of Confluent. Event streams with infinite retention provide ksqlDB applications with the means to materialize the entire historic state from a single source.

This provides us with several powerful capabilities. First, we can declaratively generate materialized views inside of our ksqlDB applications, as shown in the following code:

```
CREATE TABLE users (
    userid VARCHAR PRIMARY KEY,
    registertime BIGINT,
    gender VARCHAR,
    regionid VARCHAR
  )
WITH ( KAFKA_TOPIC = 'users', VALUE_FORMAT='JSON');
```

Second, we can use these materialized views to join, aggregate, and enrich other events, composing complex business logic with streaming primitives. Updates are made in real time as the ksqlDB application receives new events.

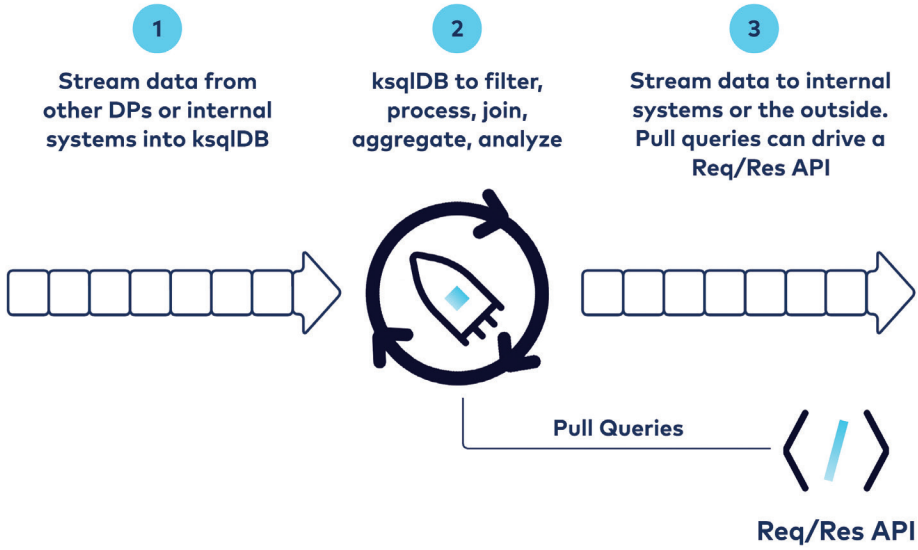

Figure 4-6. A ksqlDB application consuming data product event streams

Finally, native stream processors like ksqlDB enable our applications to write *new* events to their dedicated event streams, making them a mainstay for creating new data products.

Self-Service Management and Discovery Tooling

Building and hosting new event stream data products should be easy—data product creators should not need to struggle. At the same time, a minimum set of requirements, including dedicated ownership, documentation, metadata, SLAs, and sufficient differentiation from existing data products, should also be met. While you want to make it easy to create data products, you also want to avoid creating large numbers of similar-yet-barely-different or ill-supported products.

The key takeaway is that the overhead associated with accessing, consuming, and storing data derived from data products needs to be managed and simplified for all consumers. They should be able to spend their efforts focusing on using data products instead of struggling with access and management overhead. These self-service tools enable a clean separation of responsibilities for the data product creator and the consumers accessing the event stream output port.

But what does such a self-service platform look like in practice? Let's find out.


ABWASIA.ORG

The Confluent-Powered Data Mesh

The Confluent data mesh is built on the concept that data products are best served as event streams. In this section, we're going to examine the Confluent data mesh prototype to get a better idea of what an opinionated implementation actually looks like. We'll investigate how the principles of the data mesh map to the prototype, exploring alternative options and their trade-offs. Finally, we'll cap it off with a case study from one of Confluent's customers, Saxo Bank.

The Confluent data mesh prototype is built on top of the fully managed, cloud-native, data streaming platform Confluent Cloud. As the event stream is the primary data product mode in the reference architecture, we rely heavily on several of Confluent's commonly used services:

- Apache Kafka acts as the central broker for hosting the event stream topics, including handling all producer writes and consumer reads.
- The Confluent Schema Registry contains the schemas for each event stream and manages the schema evolution rules for handling changes over time.
- The Confluent Stream Catalog stores the metadata related to data products for easy discoverability.

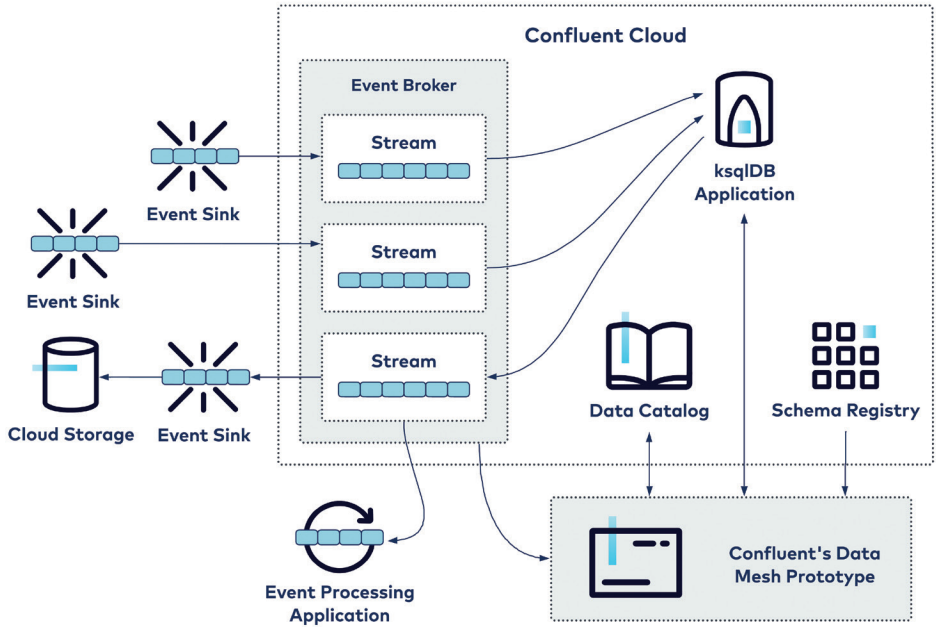

Figure 5-1. Confluent's data mesh prototype

The data mesh prototype is a proof of concept application with a graphical user interface that was built specifically to showcase what a real-life data mesh can look like. It was built in consultation with customers, domain experts, and field architects, in order to distill common patterns and use cases, and it focuses on illustrating the operations and tools that should be available to users of a data mesh.

The prototype is written in Spring Boot Java with an Elm frontend. You can run and fork your own copy of the prototype by following the steps outlined in the GitHub repo. The source code is freely available for you to use however you like.

> While this book will cover some of the features of the data mesh prototype, it will not be an exhaustive tour of all components. If you are interested in a detailed video tour, please see the Data Mesh 101 course.

Deploying the prototype locally will require a Confluent Cloud account. The automated script will create a cluster, several Kafka topics, a ksqlDB application, and several data generators to populate the topics. Schemas are

registered to the Schema Registry as events are produced, while the Stream Catalog stores the data product's metadata. The prototype is organized by the steps you might take to consume and produce data products as a user of the data mesh, organized around the common user roles:

- Tab 1 is for Data Product Consumers, where they can explore the available data products, including the metadata that describes them.

- Tab 2 is for Application Developers, who can use the available data products to build new applications.

- Tab 3 is for Data Product Owners, who can manage published data products and their advertised metadata.

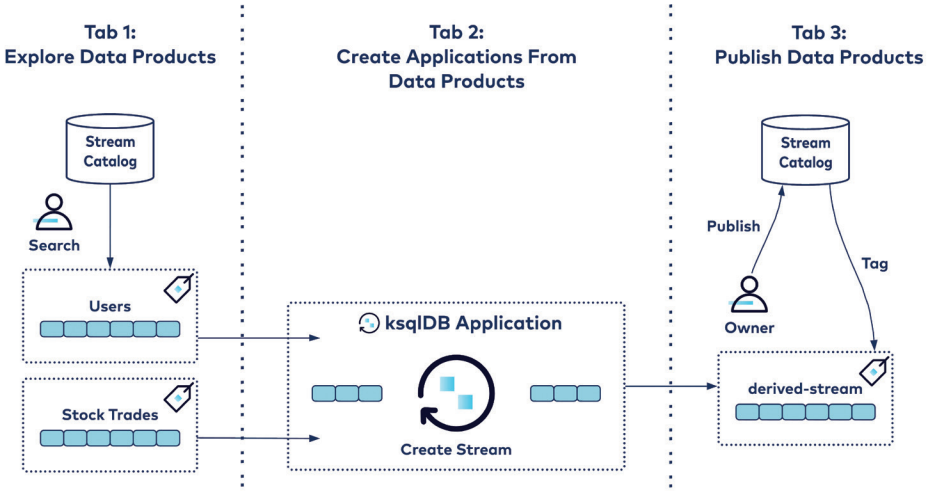

Figure 5-2. Confluent's data mesh prototype overview

The downloadable demo starts with several data products already registered in the system. How did they get there? Let's take a look at how you'd go about publishing data products in your own data mesh. We're going to work a bit out of order for this part, starting with **Tab 3: Publish Data Products**.

Publishing Data Products

There are a few steps to take before we can get down to the action of publishing a data product. First, the **federated governance** group should determine the minimum requirements for each supported data product format. For an event stream data product, this would consist of a Kafka topic, a schema, and predefined metadata such as owner, description, quality, and SLA. Secondly, your data mesh implementation must enforce these federated governance requirements, preventing registration of anything not meeting the minimum requirements.

Our prototype data mesh only allows the registration of Kafka topics that already have an associated schema. You cannot register internal topics, nor topics that are outside of your domain. Figure 5-3 shows the full selection of data necessary to register a new data product.

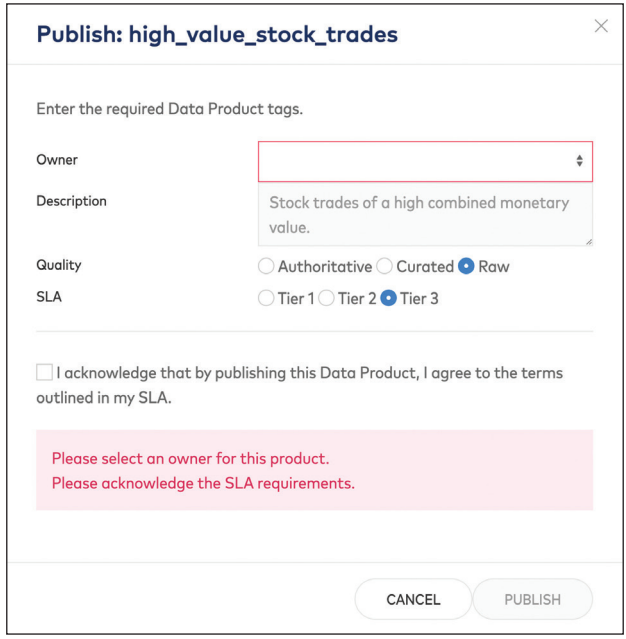

Figure 5-3. The required metadata gatekeeps the publishing of a data product

The **Owner** and **Description** help consumers identify where the data product comes from. In this example, the owner is a proxy for both the domain that the data comes from, and the person or team to reach out to should you have a question about the product. In your own fully featured data mesh, ownership should be managed using the same mechanisms that are used for other services in your company.

Data **Quality** and Service Level Agreement **(SLA)** are the two other required metadata fields. The first helps you identify the data product quality, differentiating canonical data products (**Authoritative**) from those with lower (**Curated**), or no (**Raw**) standards. SLAs provided by the domain owner indicate the service level a consumer can expect when using the data product: **Tier 1** indicates that someone will get out of bed to fix the data product in the middle of the night, whereas **Tier 3** indicates you'll have to wait until the next business day.

As a publisher of data products, you are responsible for ensuring that all required guarantees can be met. Figure 5-4 shows a completed metadata record, including explicit acknowledgment of responsibility by the data product owner.

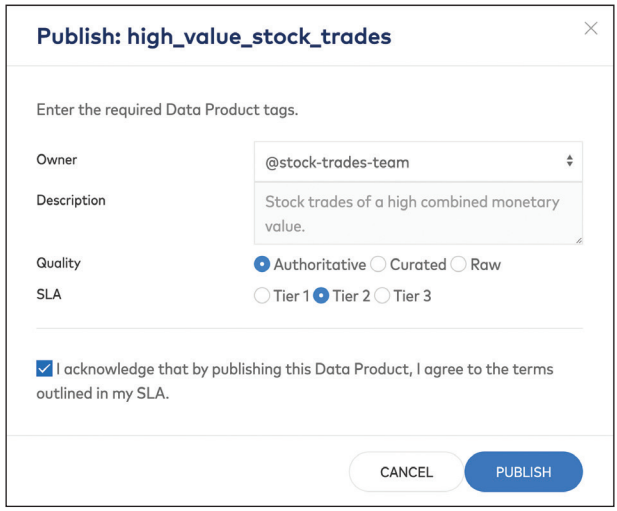

Figure 5-4. A completed metadata record for an event stream ready for publishing

Publishing a data product is a commitment to maintaining the stipulated data quality and SLAs with the same level of dedication as any other product in the organization.

Discovering Data Products

A key feature of the data mesh is that it makes it easy to find the relevant data for your business use cases. It achieves this in a couple of ways. First, a data mesh is an opt-in selection of data products. You won't see any event streams that aren't explicitly published for usage by others. This simple in-or-out barrier substantially increases the signal-to-noise ratio, and prevents consumers from accessing data that is not intended for cross-domain use.

Second, the metadata collected during publishing is stored in the Stream Catalog, and is provided to the user in a human-readable and searchable format. Discovery options include a plain-text search of the description and schema as well as filtering on specific metadata fields. For example, you may want to filter out all non-authoritative data products that are less than Tier 1, leaving you with a selection of narrower, but higher performing and extremely reliable data products to choose from.

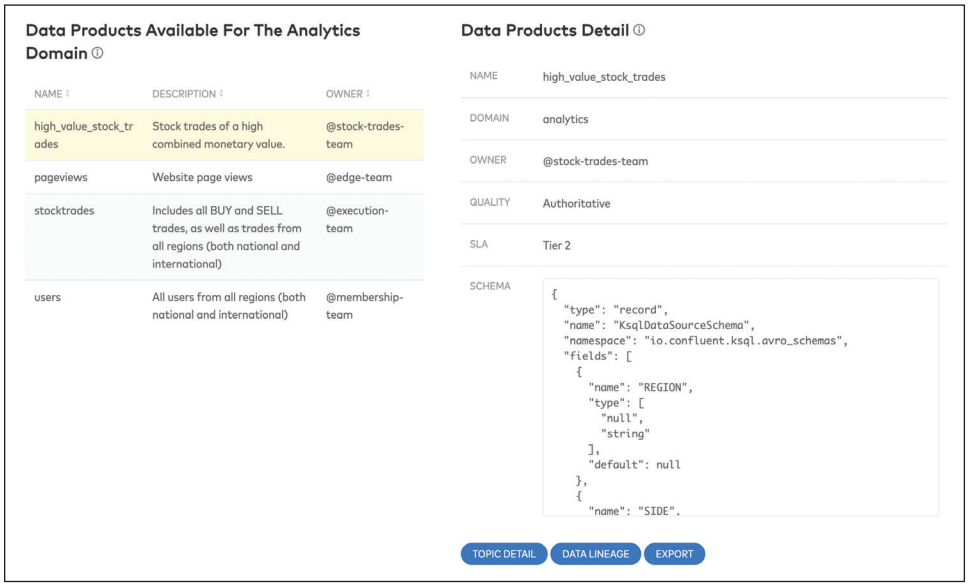

Figure 5-5. Available data products and their details

This self-service view of available data products for consumers requires accessing the **metadata** from the Stream Catalog, and the **schemas** from the Schema Registry, as mapped to the Kafka topic name.

Determining the lineage of data is another essential component of Confluent's stream governance. Clicking on the **Data Lineage** button brings up the Stream Lineage associated with the selected data product, including the upstream producer and downstream consumers. An example of this is shown in Figure 5-6.

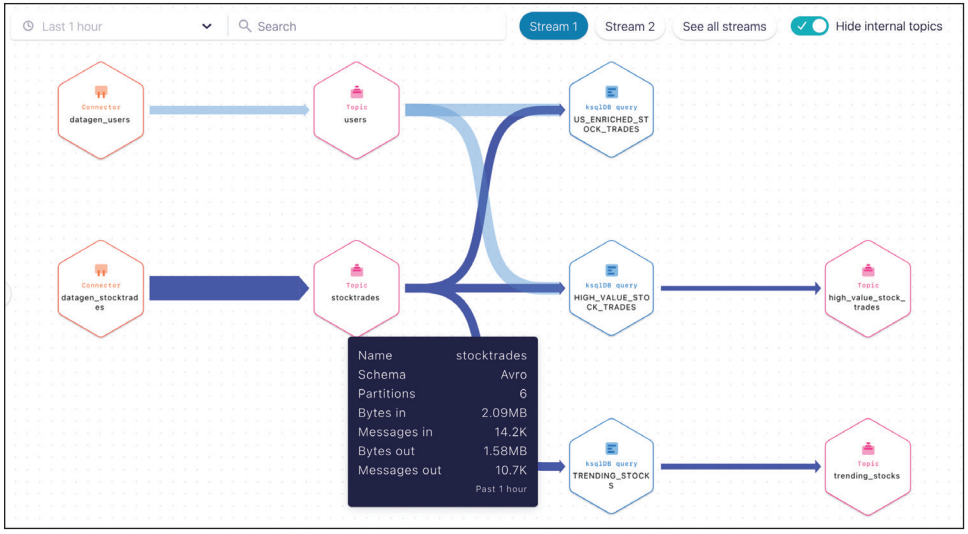

Figure 5-6. Stream Lineage showcasing the lineage of the stocktrades topic, including ksqlDB queries and connectors

By ensuring data quality via schema support, metadata via Stream Catalog, and lineage options via Stream Lineage, we're able to provide users with reliable self-service tooling to discover, understand, and trust the data they need.

Consuming and Using Data Products

Consuming and using data products depends heavily on the developer tooling used in your organization. The most important step is to integrate access to data products into the self-service tooling used by your application developers. Developers should be able to effortlessly specify the data products they need for their job at hand, with as minimal effort as possible.

At a minimum, a consumer needs to know the Kafka topic name, the hosting event broker URI, the Schema Registry URI, and the subject name (to obtain the schema). The consumer must also be able to register themselves as a consumer of the data product, using Kafka ACLs or Confluent's RBAC.

A strong permissions model can then be used to track dependencies, generate lineage, and enable interteam communications such as notifying data product consumers of upcoming breaking changes.

This principle of easy access is exemplified in the prototype by the **Export** button, as shown at the bottom of Figure 5-5. It links you directly to Confluent's Connect ecosystem, where you can quickly set up a fully managed connector to get the data directly into the data store of your choice.

The barrier to accessing data products through Kafka Connect is very low and abstracts the complexity away from the consumer. However, as we saw earlier with the Kappa Architecture, there are other ways to consume and use event streams, and your needs will vary with your organization and tooling. As an implementer of the data mesh, your focus needs to remain on simplifying the barriers to entry and integrating with your organization's workflows and tooling.

Using Data Products with ksqlDB

For the purposes of our prototype, we chose to illustrate consuming data products with ksqlDB. This is in part to showcase the native integration between Schema Registry, the Kafka brokers, and ksqlDB, but more importantly, to showcase another example of how *easy it should be* for your users to use data products.

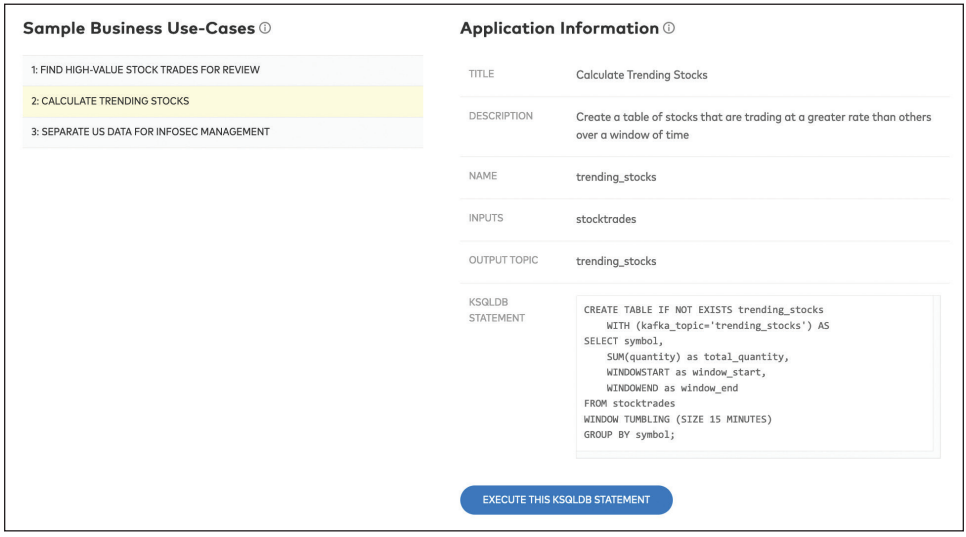

Figure 5-7. Consuming data products with ksqlDB

Figure 5-7 shows a ksqlDB application in our prototype ready for execution. Clicking on the **Execute This ksqlDB Statement** button ships the code off to the ksqlDB instance in Confluent Cloud and kicks off the processing. Please refer to Confluent Developer's ksqlDB 101 course for more information on how to get started writing your own ksqlDB applications.

Integrating with a Microservices Platform

There are a number of parallels between data mesh and microservices architectures. For one, both concern themselves with domain ownership and well-defined bounded-contexts, upholding autonomy within the domain, while requiring interdomain compatibility via well-defined APIs and event streams. Second, both require a form of federated governance, to balance individual autonomy with cross-domain compatibility, self-service tooling, and interoperational standards.

Event-driven microservices, as powered by the data mesh, provide a whole host of benefits to their users. The data mesh provides a data-centric way of thinking about intersystem communications, swapping out synchronous service APIs for well-structured streams of events. Promoting data products to first-class citizens on par with synchronous service APIs gives your application developers the means to select the streams they need to power their services. They can rely on schemas, SLAs, quality specifications, and other federated standards, reducing the overhead of getting access to important real-time business data to weave into their applications.

To make this a reality, the data mesh federated governance group, together with your microservices standards group, must focus on streamlining microservice integration with the data mesh. One form of this is in reducing the overhead on common tasks, as part of facilitating self-service. This could include:

- Establishing read-only permissions on data products via RBAC or ACLs, to preserve lineage and dependencies
- Automatically downloading and integrating data product schemas into projects, including generating code
- Automatically generating test data based on the input schemas
- Automatically validating schema evolution upon creating a pull request
- Self-service offset management and state reset tooling

Integrating data mesh with microservices is an area rich in opportunity, but unfortunately is beyond the scope of this book. Look to us in the future for a paper addressing this topic.

Managing and Deprecating Data Products

Having already discussed publishing data products, let's turn our attention to management and deprecation. Each data product owner is responsible for managing the life cycle of their own data product, and once it's published, there are two scenarios that may come up.

First, the data product may need to be refactored, possibly due to a shift in domain responsibilities, a renegotiating of the public domain boundary with consumers, or perhaps due to an oversight in the original schema. This is referred to as a breaking change, and depending on the magnitude of the change, it could mean as little as minor accommodations by existing consumers, or something as large as a full-out migration to a new data product. Stream Lineage, in conjunction with strict read-only access controls, helps you identify the affected services and plan migrations accordingly.

A data product owner needs to be able to communicate with all of its registered consumers to notify them about breaking changes and the compensating actions that will be necessary. This is a communications-intensive process and is best resolved by getting representatives from all participating teams to come up with a joint migration plan. This can include creating new data products, populating them with new schemas, reprocessing old data into the new format, and swapping existing consumers over to the new data products. Consumers may also need to reset and rebuild their own internal state, depending on the severity of the change.

The second scenario is that of deprecation, either as part of a breaking change or due to the termination of a data product. The first step in deprecation is similar to that of a breaking change: Notify your consumers (if any), and inform them that the data product is now deprecated. A deprecated data product should no longer be available for new consumer registration, should be hidden by default in search results, and should carry a "deprecated" tag in its metadata. It is also a best practice to specify the date that all read access will be terminated, sending out reminders to teams that have yet to migrate off.

Once all consumers have been migrated off, it is up to the data product owner to decide when to finally delete the data product. As part of federated governance, and in conjunction with laws relating to data storage, you may need to keep the data provided by that data product around for a period of time. It can remain as an event stream, though you may also choose to sink it to long-term storage using a Kafka connector.


ABWASIA.ORG

The Multi-Cloud Data Mesh

Many organizations today have a hybrid approach to their computing infrastructure. This could be due to an ongoing migration from on-premises to cloud, or expanding from a single-cloud to a multi-cloud deployment. Integrating a data mesh across multiple infrastructure and cloud boundaries can be a challenge, especially as services and deployments evolve over time.

A data mesh can provide a layer of abstraction over an organization's various on-prem and cloud deployments. When implemented with Apache Kafka, the mesh becomes one logical cluster made up of many physical ones spread around the globe. You can set up this replication using Mirror Maker 2.0 with basic Apache Kafka, or with Confluent's Cluster Linking.

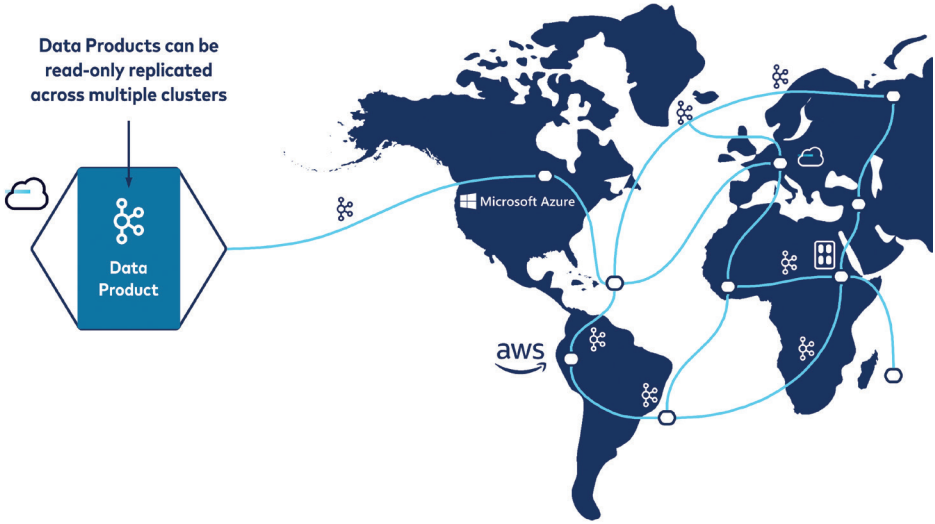

Figure 6-1. A global data mesh

Event streams provide a significant replication advantage over other data product APIs thanks to their incremental, eventually consistent, and append-only nature. As an event is published to the data product's host Kafka cluster, the replication tooling can immediately route a copy of the new event to the desired destination cluster.

While the concept of replicating an event stream is simple, there are still a host of related concerns to attend to. We still need to address syncing of topic definitions, configurations, consumer offsets, access control lists, and the actual data itself. The reality is that accurately synchronizing a data product in real-time and across multiple environments and cloud providers is difficult to do well. It is further complicated by intermittent errors, network connectivity problems, privacy rules, and the idiosyncrasies of cloud providers.

Mirror Maker relies on replicating event streams between clusters using the same publish-subscribe mechanisms as our consumer use cases. Events are consumed from the source and republished to the destination, just as with any other producer/consumer.

Confluent's Cluster Linking, on the other hand, has a substantial advantage over Mirror Maker. Instead of using the pub-sub connection, Cluster Linking taps into Kafka's replication protocol, providing a perfect byte-for-byte replica of the event stream in the destination cluster. Not only does this ensure that the partitioning and ordering of your data product remain unchanged, but it is also completely self-service: Data product consumers can choose which topics they want to sync, with Cluster Linking doing the rest of the work.

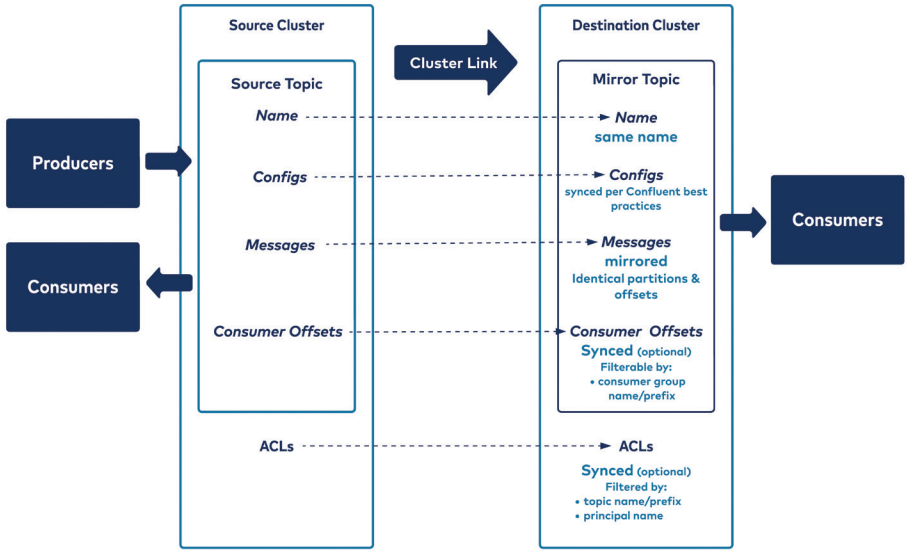

Figure 6-2. Confluent's Cluster Linking

Self-service access to data products is essential for making a data mesh work at scale. A consumer needs to be able to find data products, spin up a new Kafka cluster, and sync over each of the event stream data products they need. Barriers to cross-deployment data product access will stymie innovation and development, as prospective consumers will revert back to trying to find ways to access data products, instead of finding ways to use them.


ABWASIA.ORG

Saxo Bank Data Mesh Case Study

Saxo Bank is a Danish investment bank specializing in online trading and investment. Their representative Paul Makkar presented their data mesh journey in a Data in Motion talk entitled Kafka and the Data Mesh.

https://cnfl.io/practical-data-mesh-video

Saxo Bank is an early adopter of data mesh. Having followed conventional ETL into the data lake architecture common in many businesses, they found themselves encountering the problems that we outlined at the beginning of this book: Data was hard to find and access, data ownership was spread across teams, their ETL jobs were brittle and difficult to manage, and teams spent an inordinate amount of time finding problems and fixing them instead of putting the data to use.

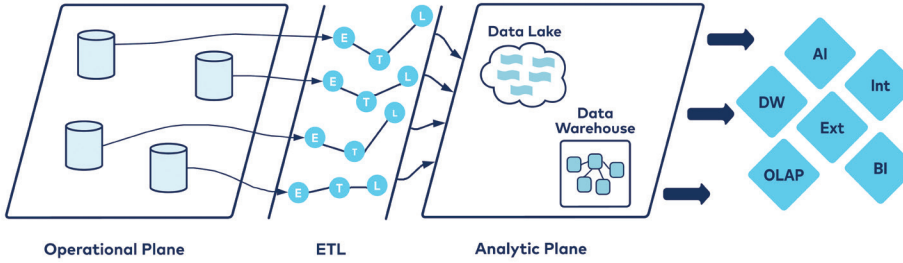

Figure 7-1. Saxo Bank's starting architecture, bridging analytics and operations through batch ETL

Thus, Saxo Bank began their journey to a data mesh architecture. Instead of narrating through their entire journey, which Paul Makkar covers in the video, we will focus on a few of their design decisions.

Saxo Bank is a .NET shop, and this heavily influenced their design decisions. For one, Apache Avro isn't supported as well in .NET as it is in the Java world, so Saxo made the decision to go with Protobuf for their event schemas. This decision was made by their federated governing body, who fenced out a poly-schema approach to ensure simplicity of tooling for both producer and consumer alike.

Aside from the problems of data quality and ownership, Saxo Bank's migration to data mesh was also driven by their need to provide well-formulated operational data to multiple consumer systems. Kafka shines in asynchronous event-driven architectures, and as it was coupled with strong domain ownership and delineation of data ownership responsibilities, it led to a common data plane for all teams to access.

A noteworthy point regarding the design of their schema modeling is that it includes both centrally defined components as well as those specific to a domain. They have correctly identified that some entities and models in their business should share a common schema across all domains in their business, while others can remain specific to a domain. This is codified in their automated schema tests, which check for both cross-domain and evolutionary compatibility, and thus prevent unintentional breaking changes.

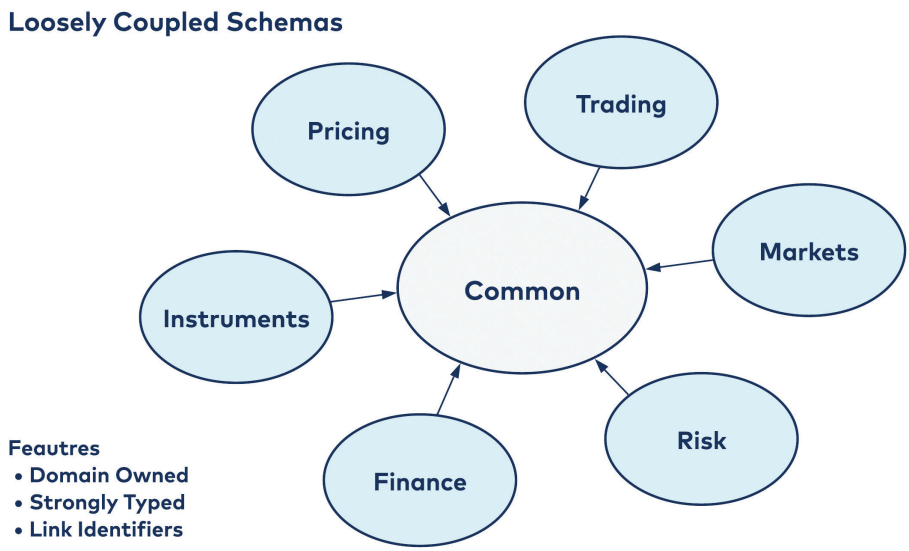

Figure 7-2. Saxo Bank relies upon both domain specific and common schemas

While the Confluent data mesh prototype uses Confluent's Stream Catalog to store the metadata and provide the backing for its data discovery UI, Saxo Bank uses Acryl Data. The principles of publishing, managing, and discovering data products remain identical: Metadata is collected as part of the publishing process, with prospective users able to browse and discover the data they need.

Moving to data mesh is not a big-bang migration. Saxo Bank's migration is enabled by Kafka Connect and its suite of connectors, along with stream processors for merging and formatting data into a suitable data product format. This allows its teams to incrementally add new data products as they need to, yielding immediate benefits without having to undertake a complicated and risky all-or-nothing switchover. And perhaps most importantly, this iterative process turns up valuable lessons in what works well, what to avoid for the next data product, and the next best area to address.

There is also a significant social transformation required to support data mesh. Responsibilities are renegotiated and boundaries are readdressed, along with historic precedents that may once have served well, but now are dysfunctional. New roles, like that of data product owner, must be taken on to ensure success. The formation of the federated governing body must be inclusive and collaborative, balancing the need for top-down decisions (e.g., use Protobuf and Kafka topics) with existing tooling, autonomy, and independence.

One decision that Paul highlights in the Saxo Bank migration is the intersection of information security with the standards established by federated governance, as mandated to all data product owners. All personally identifiable information (PII) is encrypted end-to-end and integrated with the schemas, using Protobuf tags. They use format-preserving encryption to ensure that while data remains encrypted, the format of the schemas remains unchanged. This preserves functionality for clients that do not have encryption access, eliminating the need to maintain multiple topics of similar-yet-encrypted data.

Saxo Bank shifts the data modeling and data quality responsibilities to the left, upstream to the teams that own the data. Data product owners assume the responsibilities of creating consumer-friendly data models and event schemas, working with current and prospective consumers to ensure their needs are met. Data products are populated to the operational plane, backed by Apache Kafka, where both operational domain and analytics domain teams and services can discover, subscribe to, and consume them as they need.

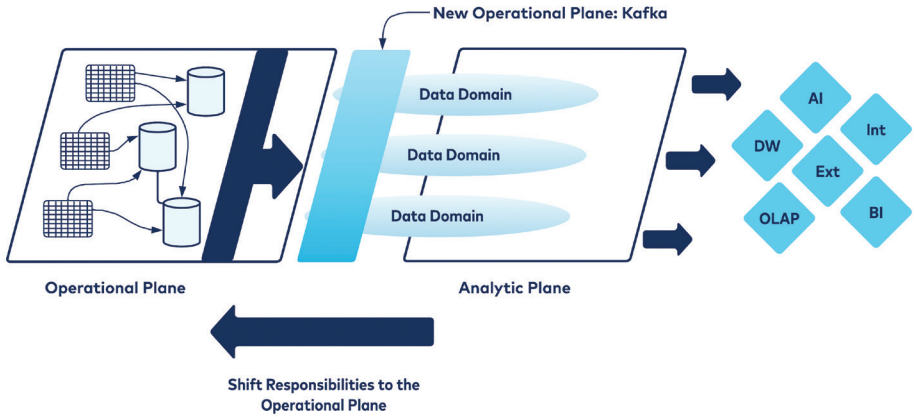

Figure 7-3. Serving both operational and analytical use cases from Kafka-based data products

Saxo Bank's data mesh architecture relies upon both the operational and analytical planes to source important business data products through Apache Kafka. Consumers rely on asynchronously updating data products to power their use cases, using stream processors, Kafka Connect, or basic consumer clients to consume and remodel data for usage in their own domains. New data products can be emitted back into Apache Kafka, registered for usage and consumption by other teams.

Saxo Bank has been gracious enough to publish additional data mesh experiences in a blog and a podcast, each describing different aspects of their journey. Graham Stirling's blog dives into the problems, theories, solutions, learnings, and decisions of their data mesh journey.

https://cnfl.io/practical-data-mesh-blog

Graham further discusses the challenges and successes of data mesh at Saxo on the Streaming Audio podcast. Using a combination of Kafka GitOps, pipelines, and metadata, Graham intended to free domain teams from having to think about the mechanics, such as connector deployment, language binding, style guide adherence, and data handling of personally identifiable information (PII). Graham also covers the role of the schemas, the schema registry, and how they fit into their business operations.

https://cnfl.io/practical-data-mesh-podcast

Conclusion

An event-driven data mesh unlocks a number of significant benefits. First, it provides the means for real-time communication between domains, allowing your consumers to react to important business events as they occur. Data product thinking ensures that consumers have a reliable and trustworthy set of information to make their business decisions.

Second, an event-driven data mesh enables you to power operational and analytical workflows, either streaming or batch, from the same data products. Operational systems can build event-driven (micro)services that operate off of the event stream, while data analysts can build streaming models or sink the data to the BI tool of their choice, using self-service options such as Kafka Connect.

Third, consumers are empowered with the operational mobility to select the data products they need to do their work. Consumers can weave streams from multiple domains together to power their own business use cases without asking others to do custom work on their behalf. And a data product consumer can in turn generate their own data products for others to couple on and use. Self-service tooling such as Confluent's ksqlDB makes it easy for domain owners to compose data products for their own use cases.

Finally, collaborative federated governance provides guidance and standards for everyone working in the data mesh. Self-service platform engineers obtain clarity on their user requirements, data product owners obtain clarity on their roles and responsibilities, and would-be consumers obtain clarity on identifying products, using self-service tooling, and change request processes.

Data mesh is a rethinking and reformulation of many of the best principles of data architectures, emphasizing data as a first-class citizen and not as an afterthought. Building and operating a modern data architecture at scale requires addressing the needs, roles, and responsibilities of everyone involved with the data. Data products provided through event streams help fulfill these needs, such as real-time processing, unification of operational and analytical sources, and integration with existing batch-based systems. Together, an event-streaming data mesh forms a new socio-technological compact that makes it easy to access and use trustworthy data to power your organization.

About the Author

Adam Bellemare is a Staff Technologist at Confluent, and author of O'Reilly's *Building Event-Driven Microservices*.

Adam has spent his career focusing on data, events, and microservices. First starting out at RIM (now BlackBerry) in 2010, he designed and developed event-driven processors and data pipelines to identify and analyze device failures. Since then he has worked primarily in e-commerce, first in designing and implementing big data platforms, before moving on to event-driven architectures and microservices. Adam is a firm believer in the tenets of data mesh, especially when combined with event streams, microservices, and Apache Kafka.