Google Cloud

Technical customer case study
November 2023

Inside the eDreams ODIGEO data mesh

A platform engineering view

Authors: Carlos Saona (eDreams ODIGEO), Luis Velasco (Google Cloud)

Table of contents

Chapter 1

The history of data at eDreams ODIGEO: from a data monolith to a data mesh

Chapter 2

The eDreams ODIGEO data mesh architecture

Chapter 3

Lessons learned and the future of the eDreams ODIGEO data mesh

Chapter 1

The history of data at eDreams ODIGEO: from a data monolith to a data mesh

Introduction

For three years, eDreams ODIGEO (eDO) has been working on an ambitious transformation program inspired by Zhamak Dehghani's article[1], *Data Mesh Principles and Logical Architecture*. This program harnesses the full benefits of the data mesh approach to data analytics.

eDO is one of the world's leading online travel companies, serving 20 million customers in 44 countries. The evolution of its ecommerce platform has been driven by canary and AB testing, with data as an integral part of the culture.

This whitepaper shares eDO's experience of building a data mesh. Rather than outlining a specific process, it provides a framework for companies considering data mesh options. It provides information on how the project evolved, what decisions were made and why, and their positive and negative outcomes.

The document is structured in three chapters. In the first chapter, we will follow the evolution of the data architecture of eDO. We look at how it started, why it evolved to include a traditional data warehouse and several silos, and why this ecosystem eventually became limiting. We look at why a data mesh was chosen over a data lake, and at how the transformation project was planned and executed.

In the second chapter we look at the technical architecture of the platform and the taxonomy of data contracts that the platform supports. We describe the process of creating new data products, as well as discussing data quality and governance.

The third chapter is structured as a learning exercise where we will review the outcomes of the decisions made and explore some alternative paths. We will also share the planned roadmap for the data mesh rollout at eDO and end with our conclusions on the project.

EVOLUTION STAGES

Stage I - The central data warehouse

Like most startups at the time, eDO's founding architecture blueprint was based on a modular monolithic application shared by several teams. The construction of an analytical informational vision was relatively simple. Every few minutes, database physical replication was used to copy the production database to a read-only replica. It was then transformed into a usable model for a traditional data warehouse system (DWH) using ETLs.

At this stage, the key characteristics were:

- A unified, company-wide data model for analysts.
- A unified data model owned by a central team (the Business Intelligence team, or BI).
- The central data team had most of the responsibility for governance and had a comprehensive understanding of the business. They translated information from the operational data model (the production database) to the unified data model, which catered for analyst teams.

[1] https://martinfowler.com/articles/data-mesh-principles.html

The central team was in charge of adapting the data warehouse to match changes in the operational data model. This was a challenge because of the frequent monolith releases, but the mandatory canary release policy alleviated this. Because there could be two versions of the monolith in production, development teams had to change the database in backward and forward compatible ways. This gave the central team more time to adapt.

This initial stage worked well, even when the monolith underwent migrations and re-writes. The underlying method was used during the development of new features in separate services (SOA) and with separate database schemas. It survived because physical replication worked as long as the database instance was shared. The limiting factors were functional complexity and the rate of change, as the data team needed a high-level view of the whole business and had to keep up-to-date with all changes. As the business grew, it became clear that a new approach was needed.

EVOLUTION STAGES

Stage II - Data silos

To scale with rapid business growth over time, the application architecture used a distributed paradigm to support fast release cycles. The monolith eventually broke down into over 300 microservices structured around 30 business domains such as flights, hotels, insurance, payments, and fraud. This removed the release bottleneck and allowed the company to scale the development team. *Figure 1* shows how breaking the platform into microservices allowed release speeds to be sustained and improved as the number of developers increased.

The need for data analytics increased as the business grew. The microservice transformation put stress on the DWH setup because physical replication created a tight coupling between the DWH and the operational microservices. Tables and columns could not be changed or renamed without risking a catastrophic impact on the ETLs and the DWH.

To resolve this, a set of dedicated microservices replaced physical replication. These periodically pushed the operational data into a staging DB with an intermediate data model between the operational world and the DWH world. A new set of ETLs transformed the staging DB model into the final DWH.

As the development workforce expanded in line with commercial growth, the DWH team struggled to keep up with the proliferation of business domains and features in their centralized model. This led teams to look for alternatives. In some cases, this meant creating their own services and pushing data into silos, but the most popular approach was based on logical table replication. DBAs set a simple job in one of the read-only production replicas to make partial copies of some schemas to a DB owned by an analytical team. This let them create their own ETLs or query the copy directly. Some analytical teams even coordinated to share raw and transformed data between them to improve efficiency. The main reason behind the popularity of this option was that it did not need involvement from the development teams.

	2008	2015	2022
Developers (Teams)	20	~150 (11)	300 (50)
Architecture	Monolith	1 monolith + 29 services	250 microservices
Avg Daily releases	1	10 (coordinated)	60 (autonomous)

Figure 1 - The evolution of the eDO e-commerce platform architecture driven by its release life cycles

The key characteristics of this stage were:

- A central DWH owned by a central team and providing a unified, company-wide model. Over time, this no longer provided a comprehensive view of the company.
- Silos grew (or were created) organically in response to demands for new or differently structured data.
- The central DWH operated in parallel with multiple data systems. Some contained subsets of the DWH data or catered for a specific purpose, others contained data not present in the DWH, and others contained both.
- Different silos employed different data sharing mechanisms. Data replication was most common, followed by ad-hoc contracts and data pipelines.
- As a general rule, data silos were not interoperable. Partial interoperability was sometimes accomplished with ad-hoc ETLs.

Data silos were limited by poor interoperability and governance, especially for silos without contracts where producers had minimal information on how data sharing worked.

Overall, the data silo stage increased analytical teams' data accessibility while reducing and hiding development costs. This fueled a data-driven culture but also created future challenges. eDO looked for ways to address these issues.

EVOLUTION STAGES

Stage III - Data mesh

Over time, the limitations of the data silo approach became clear. These included high maintenance costs, no common standards for data quality, absence of a unified global data source, and data trust issues causing discrepancies between silos.

The most visible challenge was data producers' lack of ownership. For example, when one product team changed their tables, this often created a chain reaction in analytical teams. In many cases individual teams were unaware tables were being copied and used by others. Even when they were aware of this, waiting until other teams had time to adapt to changes in their tables created friction.

Silos were problematic because there was no contract between producers and consumers, and data sharing was achieved by coupling operational and analytical data. Without an agreement between consumers and producers, lack of alignment priorities eventually eroded data quality in the silo (*Figure 2*). The need for a unified company-wide view with discovery and search was required to provide data science teams a higher level of insight.

In order to meet these challenges, eDO adopted a new data strategy, beginning with an RFI process in 2018. Professional consulting advice suggested that new data strategies should be built around the concept of a data lake, unifying all data in one place under the control of a centralized data team.

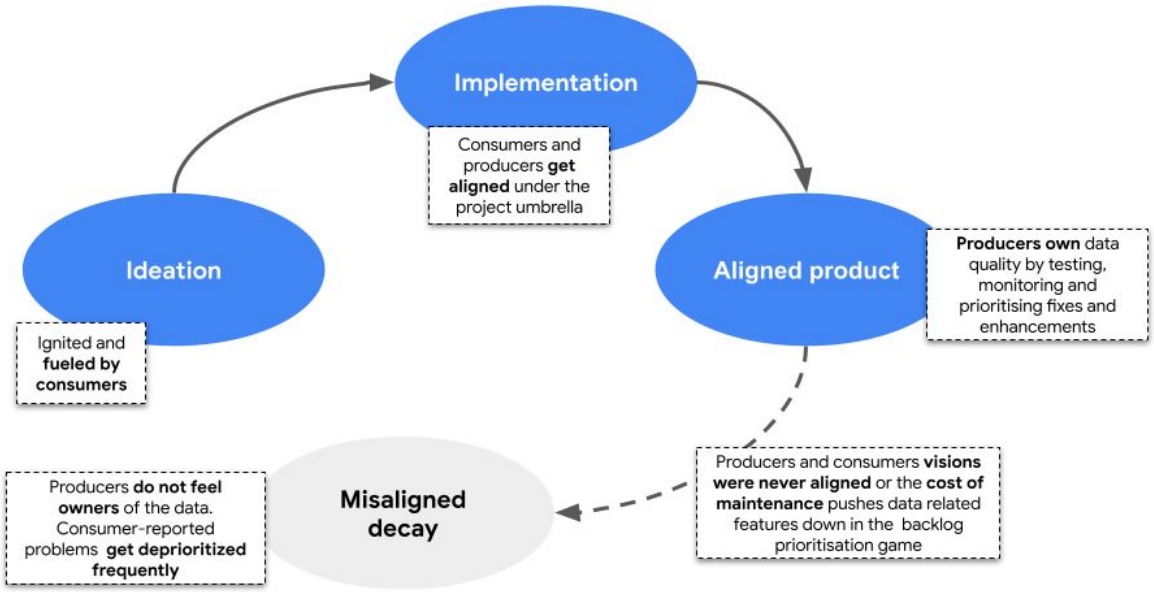


Figure 2 - The two possible final states of a data silo. When the missions of the consumers and producers match, the data silo stays aligned. Otherwise, the silo will eventually decay (dotted line) because alignment was purely the result of a management push and was thus short lived. Decay will also occur when the cost for the producer gets too high: backlog prioritization pushes back items with a low gain/cost ratio.

In environments where software release cycles are measured in months, the centralized management structure of a data lake is ideal. This approach was not suitable for eDO because of the much faster release speed of the microservices platform. This made it impossible to design a stable data layout in the data lake without affecting development speed.

A data lake was also unsuitable because of the degree of functional dependencies between eDO business domains. There were strong functional relationships between domains such as flights and hotels because they were part of the same customer journey, rather than completely isolated entities. This made it difficult for a central data team to bring together knowledge from all domains.

When Zhamak Dehghani's article[1], *Data Mesh Principles and Logical Architecture* was published in 2020, it validated eDO's doubts about the feasibility of centralized data lakes in domain-oriented microservices architectures. It also provided a new conceptualization for a distributed data architecture - the data mesh. This is a sociotechnical paradigm that builds on the concept of decentralizing data ownership. It is achieved by leveraging a domain-oriented data model and priming a self-service data infrastructure.

The data mesh core principles (*"Domain-oriented decentralized data ownership", "Data as a product", "Self-serve data infrastructure as a platform",* and *"Federated computational governance"*) aligned with eDO's culture as a product-focused organization structured around business-oriented domains. eDO also has software engineering at its core and a strong data-driven mindset.

Implementing a data mesh is both a technological and cultural change because it alters the data paradigm in the organization. *Figure 3* shows how ownership of activities in the end-to-end data process changes in a data mesh.

Instead of being monitored for quality by a centralized team, data in a mesh is treated as a first-party asset by producers. Teams producing and sharing data treat it as part of their product. They model data, and are accountable for its quality once they have agreed on data quality SLAs with consumers.

The data team no longer owns data, but instead focuses on building a domain-agnostic, self-service data platform that can be used by producers and consumers. Governance is still needed to ensure consistency between the data shared by domain teams (i.e., data products), but this is accomplished by automation and by federation of domain and data modeling experts. By implementing a data mesh, the data team stops being a bottleneck that prevents an increase in the number of domain-oriented development teams.

After adopting the data mesh paradigm, eDo had to plan the transition away from data silos. This involved changes in technology, shifts in ownership of critical functions, and a general change in the culture around data. This process is still in progress at eDO, but can be divided into the following phases: envision, foundation, sponsorship, minimum viable product, limited adoption, general adoption, and silo decommission.

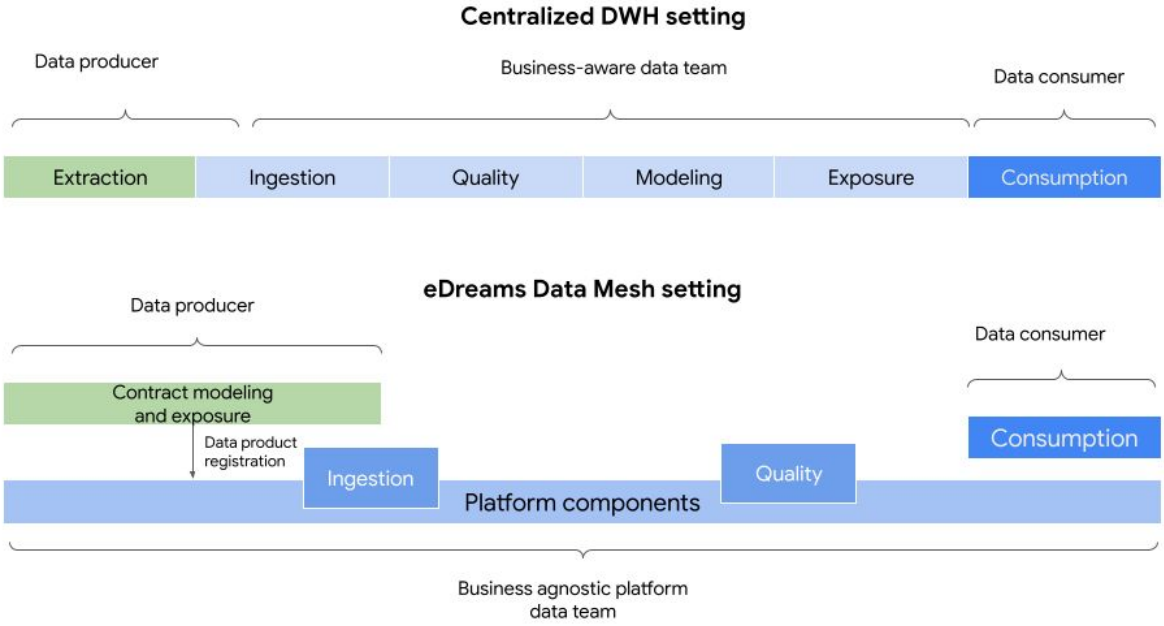


Figure 3 - Task distribution in a data mesh setting vs a centralized deployment (consumers can agree SLAs directly with producers)

Envision

The first step of the envision phase was to pinpoint problems with the current data setup and find better options. For eDO, this meant choosing a data mesh over a data lake. The next step was to map the general principles of the data mesh paradigm onto eDO's specific needs and to establish goals and constraints for the project.

On top of the established data mesh foundational principles, eDO added one of its own, based on data silo experience: **favor shifting effort from producers to consumers**. The primary reason for drift in the data silos was misalignment of priorities between consumers and producers of data, and the second was the excessive burden on producers (*Figure 2*).

Typically, data systems are created after a business case is pushed by data consumers. If the producer's mission is not initially aligned with the consumer's, or if its mission changes, the data silo will eventually suffer. This frustrates consumers because their improvements and fixes do not get prioritized.

The same happens if production costs get too high, as other items in the backlog with a better gain/cost ratio take priority. Following this logic, one solution to improve prioritization of data product features is to reduce cost on the producer's side as much as possible by increasing automation and by transferring the cost to consumers. Because of this, eDO data contracts are tailored to align with producers, whether they are source or consumer aligned. Consumers carry out any required transformation, aggregation, or stitching themselves. This helps prevent misalignment and reduces costs for producers.

Foundation

The previous phase helped clarify ideas and enabled eDO to create a draft proposal. The initial vision was then shared with two key stakeholder groups: the central data team in charge of the data warehouse (BI) and the analytical team with the greatest data needs (Data Science). They were told about the advantages of the new paradigm and asked to help refine the project to make it more feasible. This stakeholder feedback provided real-life details about ownership, data quality, and data transformation rules. This improved the vision, goals, principles, and constraints of the project. It also made it possible to create a high-level overview of how the implementation would take place. Another important outcome was a set of principles for the transformation itself:

- **No data products without consumers.** A mandatory requirement for any dataset added to the mesh was the existence of a real consumer and use case (the consumer and producer could be the same team).
- **Slow, controlled bootstrap.** Rather than designing a unified model from scratch, an iterative approach was used. Initially, datasets were hand-picked to allow freedom to make mistakes and roll back decisions without severe impacts on the business. This also enabled learning from real use cases and created generalized modeling and federation principles.
- **Federated governance.** Experienced engineers were selected to review all data contracts. Every business domain was represented by at least one expert. Because one of the principles stated that the modeling would be aligned with the producers, this expertise was limited to knowledge of the data produced, not of the use cases for consumption (which was planned to happen case-by-case as data contracts were produced).

Sponsorship

This phase involved establishing executive team buy-in, which was essential due to the magnitude, impact, and scope of the transformation. The team were presented with the analysis of the current data situation and the outline previously agreed on by the initial stakeholders. Other stakeholders were added to the conversation at this stage, and posed additional questions and challenges.

The general approach to the transformation rollout was to start small, as per the transformation principles discussed previously, and expand along with maturity. One of the biggest issues was the future of data silos. To avoid creating uncertainty and wasted effort around silo decommissioning, forced decommissioning was postponed for several years and it was agreed that the cost of migrating to the data mesh would be evaluated before improving data silos or creating new ones.

The rollout plan was divided into three phases. The first phase created a minimum viable product and tested it with a simple, low-risk data product. The second phase increased testing with higher-risk data products until the system was considered mature enough to add data products without constraints.

Minimum viable product

The first step of the rollout was to create the new domain-agnostic data platform. Keeping the data platform and its team separate from business domains ensured that they could grow without facing scaling issues if requirements did not grow in line with domain complexity.

The data team focused on developing self-service systems to reduce cost for producers and enable data governance for consumers. For producers, the system needed to provide a self-service platform that minimized the cost of sharing data.

For consumers, the system needed to automate data governance so that they could trust the data in the system. Data quality parameters formed part of the contract between each consumer and producer instead of being globally imposed.

This task was given to a team that already owned generic business-agnostic microservices for the company. This ensured that the solution was decoupled from business knowledge.

On the business-aware side, the federated team in charge of reviewing all data contracts started working on an initial set of guidelines. One of the key decisions was modeling policy as data, forcing contracts to be declarative, as a combination of Avro schemas for the data structure and a YAML file with policies expressed as properties, such as which fields contain personal or financial information, or for how long data should be retained.

The first data contract implemented was a simple dataset that helped the production team monitor the business performance of a microservice. Choosing a use case that was new, and where the producer and consumer were on the same team, helped reduce risk. The new data team carried out the implementation to ensure they had experience of using the required tools.

The data platform was not fully automated at this stage, which meant that the process of registering new data products required manual intervention from the data team. This was an acceptable compromise because it accelerated feedback from data users, while also keeping the process moving forward.

Limited adoption

Once the first data product was in production, more use cases were selected and teams began implementing their own data products using the tools developed in the previous phase.

As the number of live data products increased, automation of the whole system was improved to meet demand. Initially, only one type of data product, based on domain events, was supported (see Chapter 2). Although this limited scope, it made learning easier. As the system gained stability, support was added for more data product types. Data quality measurement and alert features were also added in this phase. Finally, the initial modeling guidelines generated by the federated governance team improved as more use cases were added.

General adoption

Once the catalog of data product types covered all known use cases, the system was considered mature enough to allow unlimited addition of data products as long as there was a consumer behind them.

This phase involved both organic and directed data product growth. Organic growth came from teams creating new data contracts. The main limitation of organic growth is that it does not always result in complete coverage of all the data products needed. The most complex use cases require data products from multiple teams and domains, and this requires coordination from multiple producers in different parts of the organization. This coordination sometimes happens organically, but is not guaranteed. Directed projects were established to complement organic growth and ensure complex cases were also addressed.

During this phase, the data platform was already in a regular product dynamic that was centered on improving data governance with features such as additional methods to measure data quality.

Silo decommission and transformation

A key initial project principle was that data silos would not be immediately decommissioned or altered. However, as the system matured, the trust level of some of the teams consuming data increased, and plans were made to decommission some of the silos after all the required datasets migrated to the new data mesh.

In some cases, silos were transformed rather than decommissioned because they were tailored to specific business domains. Provisioning pipelines were refactored to source data from the data mesh, which became the source of truth.

This phase is ongoing and is expected to last several years. Some silos have already started transformation, but none have yet been decommissioned.

Summary

Data at eDO has gone through three different stages (*Figure 4*). The driving factor behind the evolution of the system has always been the increase in the number of people developing it. As teams grow, people and data communication channels can break.

Systems often start with a centralized data team in charge of a data warehouse. Strong coupling is not a major problem because the number of teams is small enough to enable good communication between developers and the data team. The functional complexity is simple enough for one team to understand and manage. Data is easily available to everybody in the company, and there is a high level of trust and quality.

As teams and businesses grow, adding new data to the warehouse or adapting it to increasing functional complexity creates a bottleneck. Some analytical teams respond by creating their own pipelines and data systems. This creates data trust, quality, and ownership problems, but these are mitigated by the advantages of data availability and team autonomy.

The problems with data silos are also exacerbated as businesses grow, teams get larger, and functional complexity increases. This highlights the need for a unified company-wide data model. The growth of AI and data science teams further emphasizes this need.

In 2019, the natural next step for eDO would have been to revert to a centralized model overseen by a single data team. However, this was incompatible with the distributed nature of its ecommerce platform, which is organized around business domains and autonomous teams. Instead, eDO chose a business-agnostic data team and a distributed data architecture aligned with that of the platform and teams.

This chapter has outlined the history of data at eDO. In the next chapter we will examine the technical implementation of this new data mesh paradigm. We will look at technical challenges, data contracts and products, and the business-agnostic data platform. We will also look at how this links with the overall architecture of the company.

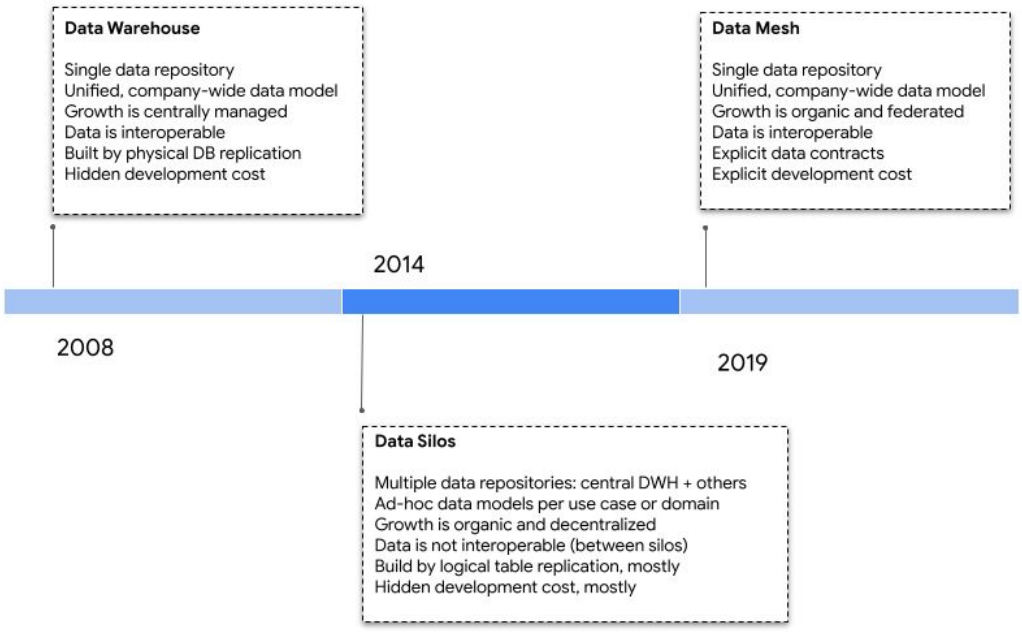
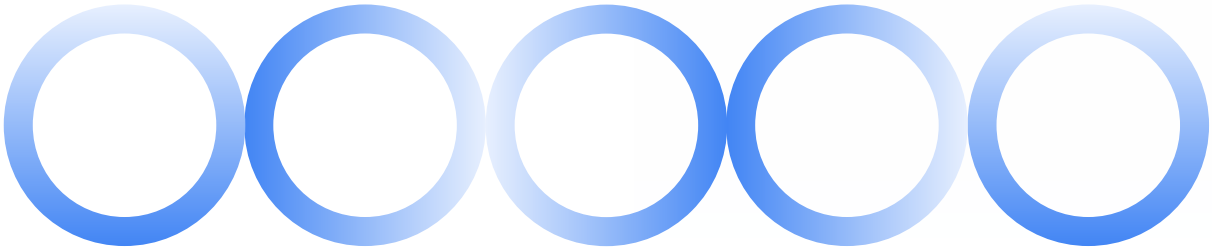


Figure 4 - The evolution of data architectures at eDO



Chapter 2

The eDreams ODIGEO data mesh architecture

Introduction

In the last chapter we looked at why eDO decided to take a fresh approach to data analytics based on data mesh principles. In this chapter we will focus on the architectural solution from both data and platform perspectives.

First, we will describe the context of the eDO ecommerce platform. We will then explain the main technical challenges of implementing a data mesh at scale in terms of data volumes as well as functional and organizational complexity. We will review the architecture of the components of the data platform and examine data product taxonomy. We will explain each of the four data product types, why they exist, and how they are typically used by consumers. Finally, we will explain the reasons behind the choices made in the technical stack that underlies the whole system.

Context: The eDO ecommerce platform

Before discussing the data platform, we will provide a brief overview of the ecommerce platform. Many of its characteristics influenced the architectural design of the data mesh.

- **Microservices architecture:** The platform is built around microservices (over 300 as of early 2023), breaking down the entire ecommerce model into independently deployable services grouped by business domains. Microservices can communicate synchronously using REST or asynchronously using Apache Kafka. Both styles can coexist in the same microservice depending on the degree of coupling accepted or the tolerance to failures.
- **Fast-paced release cycles:** To speed up time-to-market and ensure platform availability, eDO implements strict continuous integration and deployment practices. All microservices use canary releases and the release pipeline provides tools to compare technical and functional KPIs against benchmark canary and stable versions. Canary versions can be automatically promoted to stable versions when all indicators are neutral or positive. There is no release coordination function and teams deploy when they are ready without external synchronization.
- **AI and data-driven:** Data and AI are pervasive across teams. Around 1000 AB tests are performed each year, and the multiple ML models in production execute almost two billion daily predictions.
- **100% cloud:** eDO started migrating to the cloud over a decade ago, first with SaaS workspace software (Google Mail, Docs, etc.). This was followed by IaaS for testing and continuous integration and data warehouse infrastructure. Migration continued with PaaS for the complete ecommerce platform running on Kubernetes. Traffic demand on the platform is seasonal, with daily customer searches peaking at well over 100 million and CPU core usage peaking at around 30,000. This makes extensive use of Google Cloud's autoscaling capabilities to optimize latency and cost.
- **DevOps**: eDO has a long history of DevOps practices and automation, having used Puppet for over a decade and embracing Terraform and Helm when the on-prem Mesos system was migrated to Google Cloud and Kubernetes. Automation is key to keeping the cost of creating and maintaining microservices as low as possible.

Technical challenges and requirements

The data mesh community agrees that the biggest challenges for adoption relate to culture, organization, and ownership. However, there are also significant engineering challenges because the design of automated processes often influences people to change their existing work habits.

- **Data volumes.** A unified repository requires the data system to operate with terabytes or petabytes of data. This scale of data creates two challenges:
  - **Data access.** Data users expect to be able to query large amounts of data in a few seconds.
  - **Capacity planning and budgeting.** Cost and infrastructure resources grow with the amount of data and with the number of users consuming it. From a capacity planning perspective, the system needs to provision infrastructure on-the-go. From a budgeting perspective, the team in charge of the system needs to know each data user's expected consumption in advance. Because the aim of a data mesh is to provide a single data system, system budgeting must consider the data needs of all the teams in the company.

- **Data interoperability.** Interoperability poses two challenges. From a pure data point of view, consumers need to be able to link datasets modeled in different business domains. From a platform point of view, the system needs to be able to efficiently join data from multiple sets, preferably without requiring users to know in advance which queries they will perform (i.e., without needing to include indexing in the system design).

- **Data versioning.** Change management is a key challenge because the system needs to operate in self-service mode for both producers and consumers. Producers need to be able to change their contracts without being blocked by consumers updating their queries, jobs, or dashboards. Consumers need to be able to rely on stable data contracts so their queries, jobs, and dashboards do not break due to the actions of data producers. Typical data retention (over months or years) makes this balance more challenging because multiple versions of a data schema may coexist in each single dataset or data product. For existing queries to work properly in this scenario, the system needs to guarantee both backward and forward compatibility. This includes compatibility between one version and the next as well as between each version present in a dataset (i.e., full transitive compatibility). Generally, data versions can only be decommissioned when data retention policies apply and eliminate them from the dataset.

- **Documentation availability and trustworthiness.** Documentation is essential for the data mesh to function effectively. Traditionally, data pipelines have only been responsible for processing data, with documentation handled separately. This can lead to outdated documentation. To avoid this, each product team is responsible for writing and maintaining documentation for their data product.

- **Maintenance overhead.** The cost of maintaining data products needs to be reduced so that product teams embrace the new paradigm and own the data they publish.

- **Self-service, business-agnostic data platform.** The new paradigm only works if the data platform can scale independently of the number of teams and domains using it. This requires no knowledge of the specifics of domains in the platform, making it easy for producers and consumers to publish or access data.

Data platform architecture

This section outlines the design of the data mesh system. The relationships between the high-level components are shown in *Figure 5*. The sub-sections below describe each component in the system. They follow the data life cycle from data ideation on the producer side to its ingestion and storage and finally to its use on the consumer side.

DATA CONTRACTS

A data product starts with a contract. Most of the key features of the data mesh are enabled by the way data contracts are modeled:

- Each data contract is owned exclusively by a single microservice, its producer. By extension, each data contract is owned exclusively by a single team, as microserves are not shared.
- Data contracts are implemented declaratively using a file with the Avro schema and a YAML file.
- Documentation is part of the data schema as supported by Avro. It is written and maintained by the team owning the contract.
- YAML files contain policies and metadata such as data retention, data quality SLAs, and tags marking fields that contain private or financial data. They also contain hints for physical optimizations of data at rest, such as which fields should be used for partitioning (in declarative form).
- Avro and YAML files are maintained in a source code repository associated with the microservice that owns the contract.

Declarative contracts allow automation of ingestion and data quality in the ingestion pipeline without requiring any knowledge of business domains. Making contracts exclusive to one microservice and recording them as human-readable files associated with the microservice git repository helps producers own the contract, and embedding documentation in the data schema makes it easier to keep it updated.

STREAMING PLATFORM

Microservices publish data to the streaming platform in real time. This platform is built on Kafka and Kafka Registry. Kafka dynamically validates messages carrying data products and rejects any that do not comply with the data schema defined in their contract. Kafka Registry is used to define a company-wide schema compatibility policy. This policy needs to be set to mandate backward and forward compatibility between all schema versions that are live in the mesh. The streaming platform enforces the policy by rejecting any messages that violate it. If consumers and producers agree to break compatibility, the streaming platform can be directed to use a more relaxed policy for a given contract. Developers also have tools and tests to verify that their changes are compliant.

INGESTION PIPELINE

As data is published to the streaming platform in real time, the ingestion sub-system consumes it and updates the data repository in semi-real time. This sub-system is composed of several pieces. The key component is a microservice that reads messages from Kafka and writes their content in BigQuery. This microservice holds a registry of all the data contracts approved by the federated governance process. It can recognize which messages to process and ignores the rest. The goal of this process is to provide interoperability by using a company-wide federated governance to ensure all data contracts use consistent terminology and language within the constraints of each domain.

When data from a contract is published for the first time, the service automatically creates a table in BigQuery to match the schema described in the Avro file, with retention, partitioning, and clustering policies as defined in the YAML file. The service also uses the metadata in the YAML file to tag columns and enforce data mesh policies regarding personal and financial information.

Defining data contracts declaratively is the key to enabling the ingestion pipeline to be business agnostic.

DATA REPOSITORY

Data at rest is implemented as a set of tables in BigQuery (one per data product). BigQuery lets data consumers use standard SQL to query terabytes of data in a few seconds. At the platform level, it allows security policies to comply with privacy and financial regulations such as GDPR and PCI.

The design of the ingestion architecture guarantees that the data in BigQuery and the data in the streaming platform is identical. The only difference is the type of read access provided and its duration. In effect, the eDO data mesh offers consumers two different interfaces. Data products can be consumed within the operational platform by reading from Kafka, or outside the operational platform by using SQL queries in BigQuery.

Kafka offers shorter retention periods and fewer querying capabilities than SQL, but data quality is the same. How a microservice publishes data impacts both the operational platform (because other microservices may be consuming it directly from Kafka) and the analytical platform (because multiple analytical teams may be consuming it via SQL queries or ETLs). This helps producers improve data quality because the more issues a potential fix solves, the easier it is to prioritize it in the backlog.

REPORTING

Most product and analyst teams use Google Looker Studio to create self-service, serverless dashboards for reporting. The eDO data team plans to speed up the creation of new dashboards by providing templates for the most common use cases. These can be copied and altered by teams to suit their needs.

DATA CATALOG

Dataplex offers consumers a centralized web-based UI for the documentation of all data contracts. Producers document their contracts within the Avro schema. The ingestion pipeline automatically creates and updates entries in the data catalog for each data contract. This guarantees that the data catalog is always in sync with the distributed documentation source. It also reconciles the distributed ownership of data contracts with the convenience of having a single entry point for all documentation. The catalog encourages data accountability because it shows the ownership of each data set. The ingestion pipeline updates this from the YAML of the data contract.

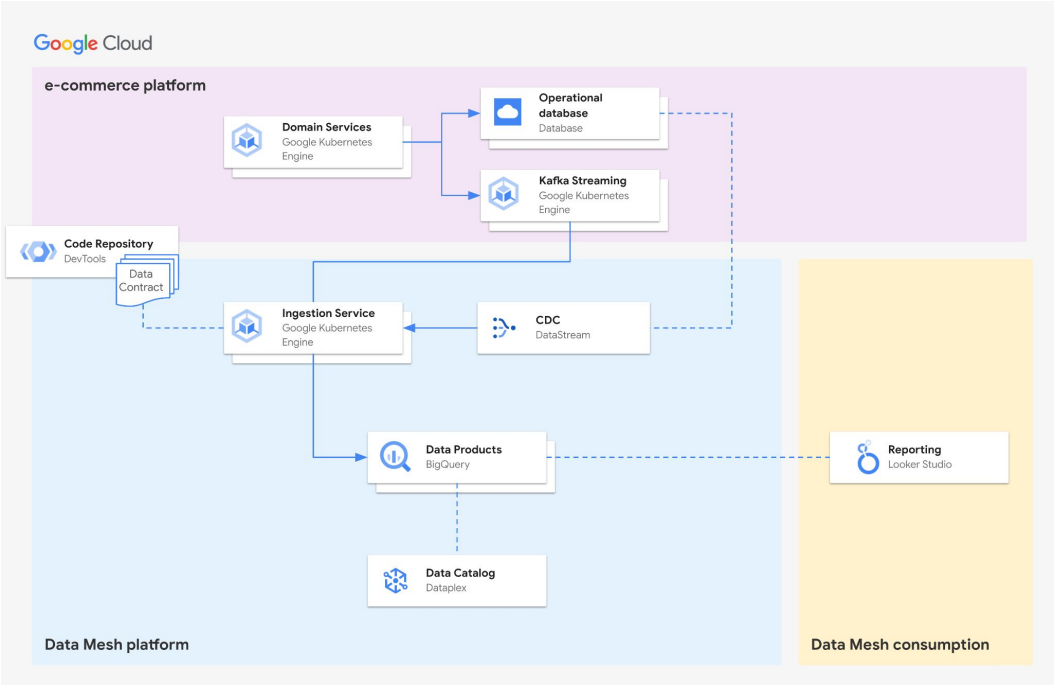


Figure 5 - The data mesh architecture at eDO

DATA QUALITY MONITORING

The quality sub-system is built with a set of microservices and automated processes that allow producers and consumers to self-register their data quality agreements as part of their declarative data contracts. Quality contracts can be declared as validations of foreign keys or as custom SQL queries that output indicators.

Foreign keys can be used to detect data losses in the pipeline. Delays in fixing data losses are often caused by problems diagnosing the location of the loss. Without this information, it is unclear whether the fix needs to be done by the platform data team or by the owner of the contract. Teams can inform to the quality sub-system of the presence of foreign keys in other tables that link to their own primary key. The quality sub-system will regularly query foreign keys and try to match them to primary keys. When a foreign key is not present as a primary key (BigQuery does not enforce foreign key integrity) this is usually because the row with the primary key is lost. This is more likely than the foreign key having the wrong value, as each table is owned by a different microservice. If data losses are occurring at the pipeline, they will suddenly increase in multiple tables due to the business agnosticism of the ingestion pipeline. If losses are only occurring in one table, the problem is likely to be with the microservice that sources the data.

Alternatively, teams can create their own custom indicators by registering a SQL query and establishing an execution frequency. The system will run the query at the indicated frequency, store the result in a dedicated table in the data repository, and export it as a normal metric in the operational monitoring system. This means both producers and consumers can set up standard alerts for when the indicator diverges from the expected values. Product teams can use this to monitor business KPIs that involve data from multiple microservices or domains. This helps address data quality problems with the same level of care as technical or functional problems in the platform.

The data platform is independent of changes in business domains and can scale as development teams grow. Data validation starts with consumers and producers agreeing on their SLAs, and ends with producers declaring these SLAs as part of the data contract. The data platform developers only need to be involved if the agreed SLA uses a type of metric not yet implemented by the platform.

Data products architecture

Now that we have a good understanding of the architecture of the data system, we will discuss the data it manages. The platform currently has **four types of data products: domain events, snapshots, derivatives, and quality records**. We will explain each in the sub-sections below.

DATA PRODUCT: DOMAIN EVENTS

Domain events represent something that happened within the boundaries of a business domain. They are immutable. They were the first data product type used in the system because of their simplicity. They are modeled according to the following principles:

- Facts: The name of an event shall end with a verb in the past tense (e.g., `BookingConfirmed`).
- Accountability: An event shall be owned and published by one microservice only.
- Integrity: An event shall have an explicit, versioned schema, enforced by the pipeline.
- Interoperability: All events shall share a common set of company-wide data (UUID, occurrence, publisher, versions, etc.) and functional relationships shall be modeled and cross-referenceable for aggregation.
- Single source of truth: Events shall only include fact data generated by their microservice.
    - Events can reference other entities by ID, or by ID+version when the state of the other is relevant.
    - Exceptionally, events can include pieces of data from other entities if the other entity is mutable, lacks versioning, and its temporary state at the time of the event is relevant to the nature of the event.
- Maintainability: Events shall only include minimal data. They shall not include the entire state of the associated entities (e.g., a `BookingCancelled`event should include the ID of the booking, but should not include the destination as this is irrelevant to the cancellation itself).

These principles differ significantly from event sourcing. There are two possible approaches to modeling events: semantic and changelog. The semantic approach captures the context and meaning of changes. The changelog approach abstracts all changes to one of three types: creation, update, and deletion. Typically, the entire state of the entity or the subset that has changed is included. The changelog approach is usually used for for event sourcing because the final state of an entity can be recreated by subscribing to the stream of events. The semantic approach provides more information, but recreating the final state is not straightforward and requires modeling all possible semantic changes.

eDO chose the semantic approach for several reasons. First, abstracting changes as inserts, updates, and deletes loses contextual information, even when the entire final state is included. For example, a booking can be canceled by the customer or by the airline. If the microservice in charge of the booking state is sufficiently granular, the resulting state of the booking might be the same, even though the business process is very different. Domain events can be used to represent this information by modeling `BookingVoluntarilyCancelled`and `BookingInvoluntarilyCancelled`separately. It is possible to have this in a changelog approach by adding context to the entity model, but it complicates the model and raises costs for producers.

DATA PRODUCT: DOMAIN EVENTS

Because the semantic approach minimizes the amount of data in events and makes them highly granular, it maximizes maintainability for producers. This is aligned with the goal of minimizing production costs. For example, an event called `BookingPaid` includes the booking ID and data about the payment (i.e., amount, currency, buyer) but does not include data about what was been booked (these details can be retrieved from a different event if needed). This imposes extra work on consumers who want an overview of business transactions. On the positive side, it also reduces coupling. The frequency of schema changes in granular events is much lower than changes to full-fledged data products representing a complete entity such as Booking. When high granularity is too much of a burden, the system offers a different kind of data product, called snapshots, which are covered in the next subsection. Note also that, unlike event sourcing, there is no aspiration to model all business events. Instead, only the ones that are requested by data consumers are modeled.

Data producers use the following process to create a new domain event:

1. The producer creates the data contract after discussing it with the prospective consumer. It is composed of two files:
    a. An Avro schema file with the event model and documentation.
    b. A YAML file with declarative policy metadata such as fields containing personal or financial information, retention policies, the microservice owning the event, hints for optimization in BigQuery, etc. (see *Figure 6*).
2. The new event is reviewed, changed if needed, and registered in the pipeline.
    a. Registration materializes as a commit in a git repository owned by the data team.
    b. Because the event is independent of the internal representation of data in the microservice, this step does not block the producer from releasing new versions of their code.
3. The producer integrates a common library into their microservice to publish the event. The library is maintained by the data team and guarantees that the standard data part of the event is present and correct (UUID, timestamp, name of the microservice publishing the event, version, etc.).
4. If applicable, the library can guarantee transactional consistency. The event can be attached to a running database transaction and will only be published if the transaction commits. This guarantee is critical to ensure data quality in transactional environments. The library uses a temporary table for publishing that is read asynchronously. Using a sidecar instead of a library was discarded, despite having several advantages, because it would have made it harder to ensure transactional safety.
5. Once the first version of the microservice that is publishing the event is released to production, the team is accountable for data quality as agreed with consumers. The team is in charge of revising data with every microservice release. As data is part of the microservice code, it can be tested like any other code in the microservice. This makes data and code harder to desynchronize.

DATA PRODUCT: DOMAIN EVENTS

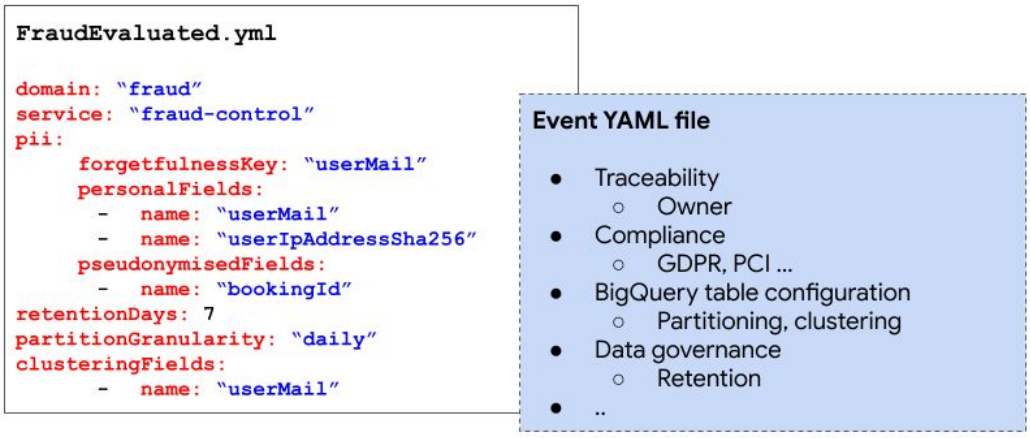


Figure 6 - An example of a YAML file implementing "policy as data" for a domain event

Data mesh products are not the only messages shared in the Kafka streaming system. If teams want to share their data with the rest of the company via the data mesh, they can publish and consume asynchronous messages in Kafka whether or not they are data mesh events. They can do this without going through the federated governance process required by the data mesh. The self-service automations of the data platform (versioning, schema validations, monitoring, etc.) provide efficiencies, although this means extra responsibilities for teams in terms of data quality and data contract interoperability.

DATA PRODUCT: SNAPSHOTS (AGGREGATES)

Even though many data consumers benefit from the reduced coupling of domain events, this is not always the case. Some data consumers need to model a parallel representation of entire business entities. Using domain events for this is cumbersome and increases coupling because it requires understanding the semantics of all the possible changes in the entity. It also forces producers to model all possible changes as domain events. This is unfeasible for business entities that can be modified by a high number of processes or use cases, such as flights or hotel bookings.

This consumption use case is better served by a snapshot, which typically corresponds with an aggregate in domain-driven design [2] parlance. A snapshot is a data product that shows the state of an entity owned by a microservice at a specific moment in time. It exposes a subset of the internal state and typically uses a different data schema.

[2] https://en.wikipedia.org/wiki/Domain-driven_design

DATA PRODUCT: SNAPSHOTS (AGGREGATES)

The main challenge with snapshots is minimizing the effort of publishing them. Aspect Oriented Programming (AOP) offers a solution by intercepting all operations against the database. However, this has several problems:

- It is not uncommon to use SQL scripts to update rows *en masse* (e.g., as part of a bug fix), bypassing the microservice code. This makes using AOP unfeasible.
- For snapshots to be self-service for producers, they need to integrate a library with the AOP code (as they do for domain events). There is a difference between a library that is used explicitly on demand for the sole purpose of publishing a data product and a library that intercepts all operations within the database. The latter can alter transactions and degrade performance.

Trigger-based interception is another possible solution, but it requires developers to implement triggers table by table. This is time consuming and risky because bugs in the triggered code can cause transaction problems and performance issues.

CDC is another database-level solution. It is transparent for the development team and does not alter the execution of transactions because it is typically implemented by asynchronously reading the database's transaction journal. In most cases, the production database will be physically replicated as a read-only copy, allowing CDC activation in the replica and eliminating performance impacts on the production database.

All three solutions share one major challenge: it is not straightforward to take a snapshot of an aggregate using data from a transaction. In most non-straightforward cases, business aggregates are modeled with multiple tables. The code is optimized to implement business use cases by altering just a few tables, without loading the full state in memory. Transaction information passed through AOP (or with triggers, or in the CDC stream) will not always include the aggregate primary key. Inferring this from the primary key of a child table requires logic that understands foreign keys and can traverse them until it reaches the root table. This logic cannot be domain-agnostic. It is completely coupled to the business domain and to the internal database model of the microservice.

The ideal solution splits the logic coupled to the domain (i.e., the internal representation) from the remaining logic, leaving business-agnostic logic as a separate self-service pipeline (see *Figure 7*). The data team is in charge of a generic snapshot aggregator service that listens to CDC events and publishes data products to the streaming platform.

DATA PRODUCT: SNAPSHOTS (AGGREGATES)

Data producers use the following process to create a new snapshot:

1. The producer creates the data contract.
2. The new snapshot contract is reviewed, changed if needed, and registered in the pipeline. Registration involves two extra steps:
    a. The producer provides the name of the database schema.
    b. The data team enables CDC in the database of the service.
3. The producer implements two public endpoints in the microservice that owns the snapshot:
    a. One endpoint returns the value of the ID of the snapshot when given the name of a table in the schema of the service and a primary key for that table. The service needs to know how to navigate relationships in its own schema so that it can calculate which row in the root table is the ancestor of the row passed as a parameter. *Figure 7* shows a case where the transaction only updates a table that is a grandchild of the root table of the aggregate.
    b. The other endpoint returns the snapshot when given its ID (primary key).
4. Once the first snapshot is published, the data producer is accountable for data quality (as agreed with consumers).

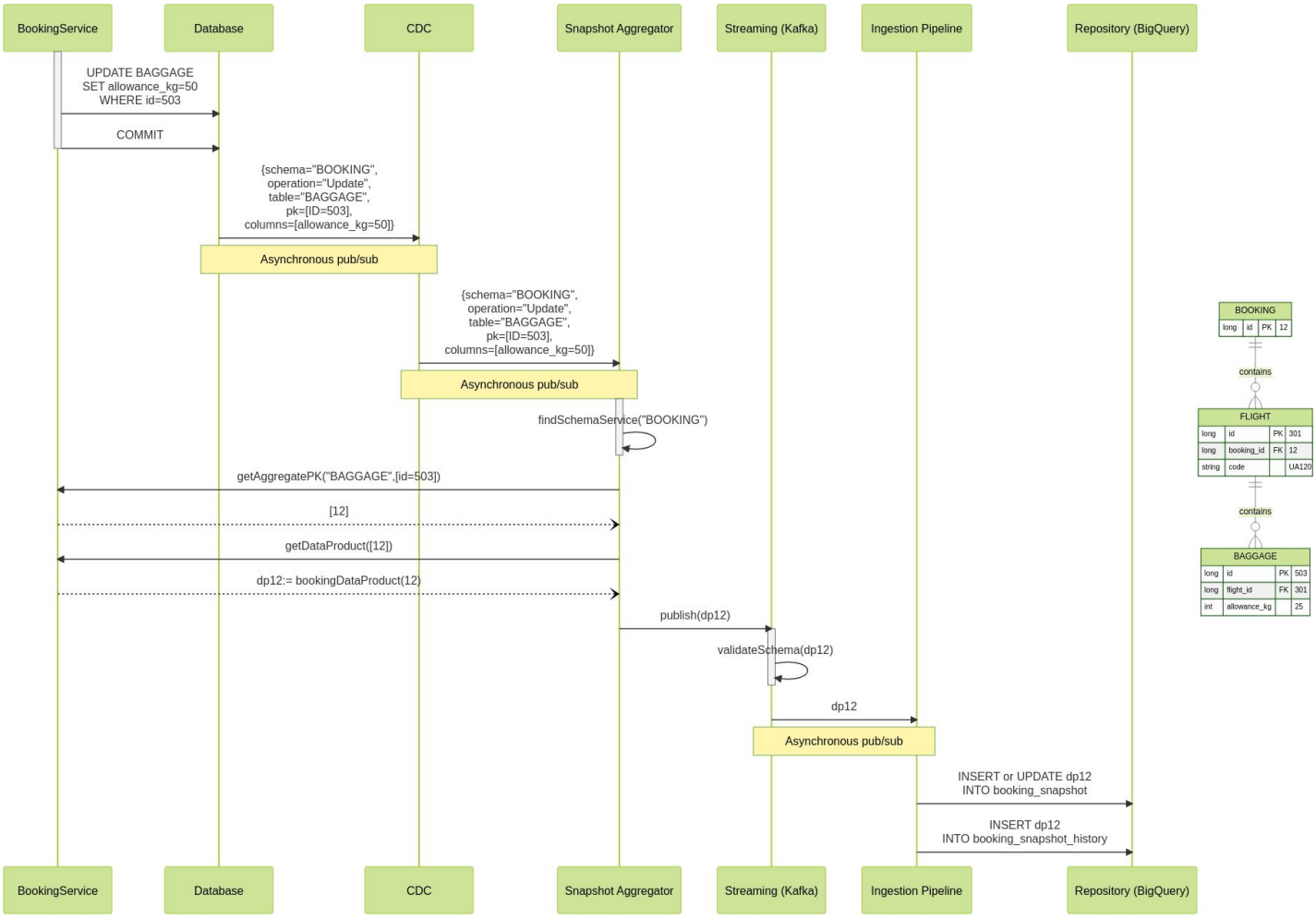


Figure 7 - Example: Generating a snapshot for a fictitious booking aggregate

DATA PRODUCT: SNAPSHOTS (AGGREGATES)

The ingestion pipeline consumes snapshots from the streaming platform and writes them to BigQuery verbatim. The pipeline also maintains two BigQuery tables for each snapshot data product. One contains the latest snapshot and the other contains the history of all snapshots. This solution has several limitations:

- Snapshots are only supported for microservices with persistent and transactional storage.
- Snapshots are eventually consistent (in the sense that there is a latency between the aggregate changing and its snapshot being available). There is no option to make snapshot generation transactional.
- There is no guarantee that all intermediate states will be published due to the latency between an aggregate changing and its snapshot being taken. If several transactions occur very quickly one after the other and change the state of a given aggregate instance, the intermediate states will have disappeared by the time the pipeline retrieves the state. This means the resulting snapshot will accumulate multiple transactions.

Despite using CDC, the ingestion pipeline is decoupled from internal microservice data representation. Teams are free to change their DB schemas if they update the two methods imposed by the contract. In exchange, they are free from all the logic regarding updates and from the publication itself in the streaming platform. Teams can also configure a table filter to avoid their endpoints being queried for tables that do not alter the snapshot. In this case coupling is restricted to table names; teams only need to review the pipeline configuration when adding or renaming tables.

DATA PRODUCT: DERIVATIVES

Derivative data products are generated inside the analytical part of the data mesh (i.e., BigQuery) and use other data products as sources. As the number of data products grows and their usage increases, some data transformations start repeating on the consumer side. Letting consumers reuse these transformations increases efficiency. If this reuse is generic enough to be applied broadly, it can be implemented as a new derivative data product type that encapsulates the transformation logic.

Derivatives have several key characteristics:

- Instead of being aligned with a microservice, they are aligned with a processing logic that reads and transforms data from one or more data products. The table schema is predefined and agreed up front.
- The owner is usually an analytical team willing to take on the burden of maintaining the data product to benefit other teams.
- Derivatives are implemented using materialized BigQuery views. BigQuery automatically keeps the view in sync with its source tables without any user input.

DATA PRODUCT: SNAPHOSTS (AGGREGATES)

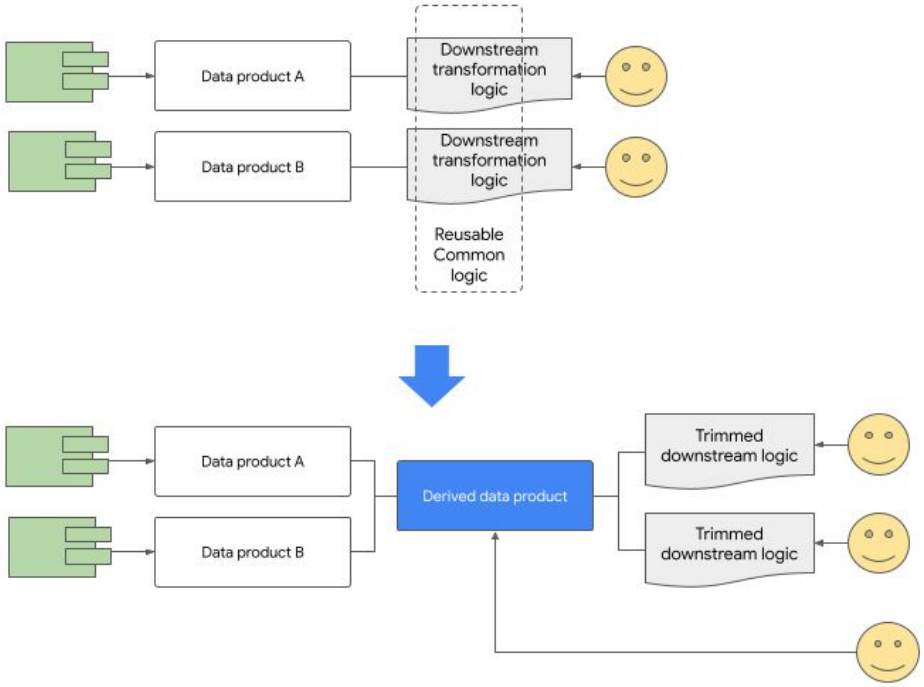


Figure 8 - Derivative data product construction detail

DATA PRODUCT: QUALITY RECORDS

The final data product is used to store the quality metrics of the other data products. The system architecture includes a self-service component where producers and consumers can register contracts on data quality. The system automatically evaluates the contracts, stores the results in BigQuery, and exports them to the monitoring system of the company to allow alert configuration.

Consumers carrying out historical data analysis need to be able to query historical data quality. This can be accessed like any other data product using SQL. Having data quality in the same repository as data itself also allows users to write SQL queries that filter data and quality metrics together.

The data mesh currently supports two types of data metrics:

- The system can detect loss of data using foreign keys (although BigQuery does not currently support enforcing foreign key integrity). By regularly querying which foreign keys in table X do not have the corresponding primary key in table Y, the platform can detect the percentage of rows “lost” in table Y. This helps identify whether data loss is caused by the pipeline itself or by a bug in the microservice that owns table X. When the cause is the pipeline, the loss will increase across all tables.
- Teams can register their own custom SQL queries into a YAML file. The system will regularly execute the SQL query and export it to the monitoring system. Analytical and product teams can use this to monitor data quality and stay alert to problems. Development teams can use the data mesh to detect operational or functional issues in the ecommerce platform as well as data pipeline problems (*Figure 9*).

DATA PRODUCT: QUALITY RECORDS

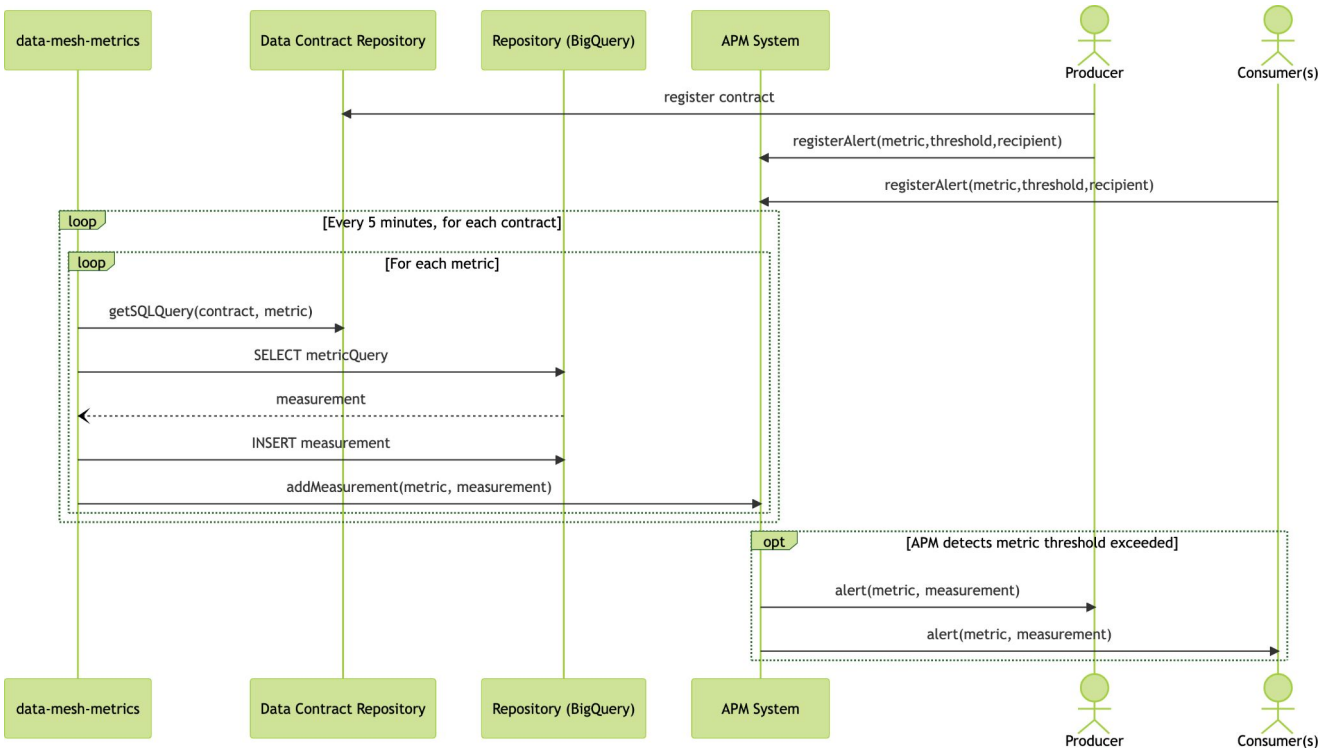


Figure 9 - Measuring data quality, recording it as another type of data product, and integrating it into the APM system

Data products usage

Consumers often use multiple data products for a single use case. This follows the principle of modeling producer data and moving the stitching effort to consumers. For example, an analytical team interested in eDO Prime subscription performance might use a snapshot of the Prime membership aggregate along with domain events relating to successful and failed signups and renewals. This information can be used to calculate renewal success ratios.

It is more straightforward to use events (rather than snapshots) to calculate these ratios, especially in conjunction with payment attempt events published by the collection domain. The analytical team might also use one or more quality records to monitor data quality and get alerts when it drops below certain thresholds agreed with the producers.

Domain events are the most frequent data product on the platform, with over 200 deployments. Snapshots are much less frequent. Derivatives are rare because aggregation logic that is useful for analytics is also useful on the online platform. Aggregation is typically implemented in two ways, neither of which use derivatives. For the simplest cases, it is implemented as a dedicated microservice that consumes the source snapshots and domain events and joins them locally (e.g., to compute ranks such as the most popular routes booked) In more complex cases, such as when data volumes require parallel serverless stream processing, it is implemented using Apache Beam.

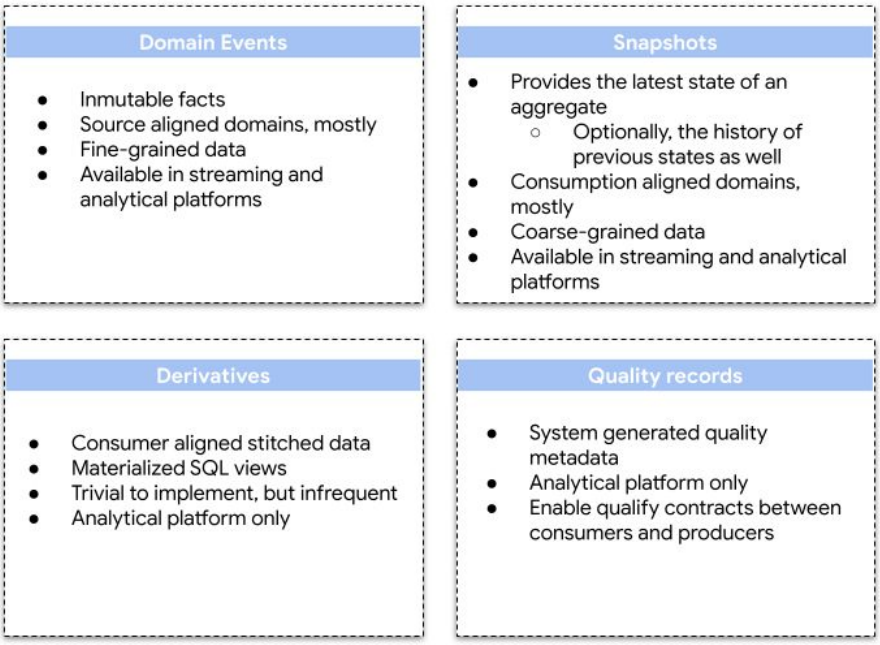


Figure 10 - The four main types of data mesh products architecture

Technical stack rationales

There are several reasons behind eDO's product choices for their data mesh technical stack:

- **BigQuery** sits at the center of the architecture because all data products are stored as BigQuery tables. eDo chose BigQuery for storing data at rest for several reasons:
    - It can query terabytes of data in a few seconds without requiring prior knowledge of the type of joins between tables (removing the need to pre-index).
    - It supports standard SQL, which is widely used for analysis by both developers and data users.
    - It is serverless, which means no maintenance cost and no need to plan capacity in advance. This is valuable at the start of the project because it lowers up-front costs and remains useful once the system is stable.
    - It allows costs to be attributed to producers and consumers separately. This solves for budgeting challenges because the cost was previously owned by the producers despite largely being driven by consumer queries.
    - It has native support for Avro payloads.
    - It was already being used for one of the eDO data silos with good results.
- **Data Catalog** provides a simple but powerful web-based UI. As it is part of the **Dataplex** suite, it is simple to integrate with the ingestion platform and BigQuery. It is integrated with the Google console, which makes it easy for data users to access data documentation, lineage, and ownership.

- **Datastream** was selected as the CDC system because it is serverless and supports all the transactional databases currently used at eDO.
- **Google Dataflow** was chosen as the execution runner for Apache Beam because it provides fast, serverless parallel stream processing.
- **Google Looker Studio** was chosen because it is serverless and can be used in self-service mode by any team wanting to create its own dashboards using data in the mesh. It has not replaced the BI systems in eDO because it lacks many features present in the standard reporting heavyweights, but its simplicity has enabled some product and analytical teams to create their own dashboards autonomously.
- **Apache Avro** was selected as a schema language and binary serialization protocol for several reasons:
    - Kafka can validate messages that fulfill the schema and reject those that do not.
    - The schema language supports custom logic types.
    - The schema language helps with version compatibility changes, particularly with enums via default values.
    - Documentation is written as part of the schema, which makes it much easier for producers to keep it up-to-date.
    - Avro is supported by Kafka, Kafka Registry, and BigQuery.
- **Apache Kafka** was already used extensively in eDO before the data mesh project, so it was a natural choice. The versioning enforcement features of Kafka Registry cemented this decision.
- All the microservices in the data platform run in **Google Kubernetes Engine**, as it was already running the entire ecommerce platform.
- **Terraform** and **Helm** were chosen to implement infrastructure-as-code, taking advantage of Terraform's integration with Google Cloud. Again, they were already used for the entire ecommerce platform

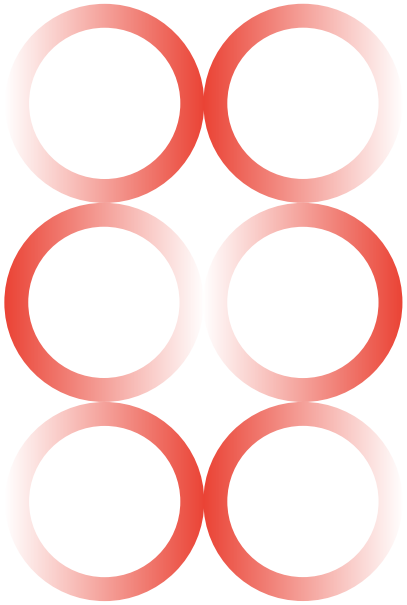
Summary

So far, this whitepaper has outlined how eDO implemented a business-agnostic data platform, how it operates in self-service mode, and how it isolates development teams from data infrastructure concerns. We have also looked at how eDO isolates its data platform team from business knowledge to avoid overloading them with the ever-increasing functional complexity of the system.

We have examined how eDO achieves its data mesh goals by using declarative data contracts expressing both data schema and data policies, and by using CDC technologies as a trigger for data sharing without coupling the data pipeline to the internal data representation of the microservice producing the data.

We have also covered the four types of data contracts in eDO (*Figure 10*) and examined how domain events and snapshots can be used for different consumer use cases.

In the final chapter, we will discuss the main lessons learned during the deployment of the data mesh and consider the platform's future.



Chapter 3

Lessons learned and the future of the eDreams ODIGEO data mesh

Introduction

In this chapter we will cover the lessons learned in the four years since the data mesh project began. We will also look at improvements and new features planned for the future.

Key takeaways can be categorized into three main areas: cultural and transformation management, software engineering processes, and software design and architecture. We will discuss the decisions taken and look at the positive and negative outcomes, as well as exploring alternative options.

LESSONS LEARNED

Culture and transformation management

We learned important lessons about dealing with people, culture, and the large-scale transformation that a data mesh entails. This is important even in companies like eDO with a strong data-driven culture. A data mesh is a paradigm, and the most difficult part of the journey is changing the culture of the organization to ensure successful adoption of this new paradigm.

Decision: Hand-pick consumer stakeholders that are willing and able to be early adopters.

The good: This speeds up the process and allows for earlier implementation testing with real problems. The ideal early adopter is a team whose main pain points are addressed by the data mesh and who are willing to absorb the extra cost of early adoption. They also need a data mindset strong enough to provide constructive feedback. In eDO's case, the data science team was the perfect match because they needed a cohesive global data source to meet the challenges of implementing AI across multiple business domains. Their frustration with data silos was higher than other analyst teams, and their affinity with engineering culture ensured actionable and constructive feedback.

BI was part of the project from its inception, showing that early involvement did not need to be limited to teams adopting the data mesh. As the owners of the initial centralized data warehouse, they had a unique view on many implementation challenges, such as data quality measurements and data corner cases tied to their domain-specific knowledge. This experience helped make the initial design fit for purpose.

The bad: Not involving all data consumers from the start may have come at a cost. The teams most satisfied with the changes were those involved from the beginning. Some teams that adopted a data mesh later were satisfied with the results, but others were more skeptical. Later involvement seemed to correlate to skepticism.

Potential improvement: While it is not feasible to involve all consumers from the beginning without risking the whole project, it is clear that there were some mistakes, frustrations, and misunderstandings with consumers who became involved later in the process. These could have been avoided with more frequent communication. Explanations given at the beginning of a multi-year transformation need to be refreshed periodically. More communication in this area would have helped increase satisfaction.

Decision: Make onboarding optional.

The good: Pushing decommissioning into the future and making data silo replacement optional helped unblock project progress. Taking migration concerns out of the conversation made teams more open to the benefits of the new system. Teams that joined when they were comfortable with the maturity of the platform were more satisfied than teams that had to use it because other teams had adopted it.

The bad: Organic adoption creates problems with transitive dependencies. Until all data is present in the data mesh, teams can face a situation where part of the data they need is in a silo and another part is in the mesh. This can happen when developing new functionalities that share data using the mesh. In complex use cases where analysis is done over multiple domains across transitive dependencies, this gap may not be immediately obvious to producers. eDO is addressing this by handling the most complex use cases with specific migration projects that complement organic growth.

Potential improvement: Timing of specific migration projects is crucial for avoiding problems caused by some data being present in the data mesh but not in silos. It is best to assume a worst-case scenario of organic growth never succeeding except for in small use cases.

Decision: Foster feedback loops with data producers.

The good: The best way to ensure data quality is to have data owners consume their own data. eDO created feedback loops that accelerated the rate of domain teams deciding to create new data contracts. They created these feedback loops by providing self-service dashboards, fast SQL access to data, common libraries to lower the cost of producing data, and self-service alerting and operational indicators,

The bad: Some producer teams prioritize data contracts for their own benefit. This can give a false measure of progress if most data contracts are feedback loops. This activity can be detected by measuring the number of consumers in each function or division, as producers are typically in the engineering division.

Decision: Data quality is variable and agreed between producers and consumers, instead of decided by a central team.

The good: Giving consumers and producers control over the degree of acceptable data errors makes the system scalable. Centralizing control requires a single team to understand what all datasets mean and how they are used, which does not scale as datasets and consumers grow.

The bad: Consumers are sometimes reluctant to implement cultural changes because of the extra workload involved in talking with producers (*"I just want the data to be there and be right"*). The same is true of producers, particularly because in their case the required data quality is the maximum between all their consumers. Even though the new explicit cost is lower than the implicit previous costs, the latter are more intangible.

Potential improvement: Template SLAs could help producers and consumers align. The templates could be based on business impact or on degrees of overall data quality. They could also use decision trees suggesting indicators and thresholds based on business impact. However, there is a risk that templates will not be useful without real, complex cases to base them on.

LESSONS LEARNED

Software engineering processes

There are many companies with experience of implementing a data mesh, but not yet enough to provide a well-established set of patterns and practices to follow. Currently, data mesh implementation is a complex journey that requires careful planning and an adaptable mindset with the ability to react quickly to unforeseen obstacles.

Decision: Plan for iterative creation of data products, driven by real consumption.

The good: Despite the recent popularity of the data mesh paradigm, data mesh transformation is still far from being a predictable software project. In the case of eDO, it helped that the project started small and with realistic expectations about its maturity. Early adopters were willing to accept that the platform would have issues in the beginning and could not guarantee perfect backward compatibility, and knew they might be asked to redo some work. It also helped that the first data products were domain events that had short retention periods and a small number of columns. Finally, not having to design a complete unified data model for the whole company made getting feedback much faster and required a much lower initial investment.

The bad: Small domain events solved more cases than anticipated, but sometimes this came at the cost of unwanted complexity for consumers. Source-aligned domain events carried the project a long way before snapshots and derived data products were added, despite conveying less data than snapshots and requiring more work for consumers. In some cases consumer teams achieved this by building very complex queries - the exact scenario that snapshots avoid. Because the first consumers were data scientists and data engineers, this behavior might not generalize to other companies with less SQL-experienced analytical teams.

Controversial alternative: More bias towards data product type diversity (rather than towards actual consumer satisfaction) would have been useful. For example, snapshots were added after quality records because actual consumers of domain events wanted to measure data quality. Had snapshots taken precedence, more teams would have been involved earlier on.

Decision: Make data backlogs part of team product backlogs.

The good: It is simpler to use a single backlog to prioritize all the work of a team because decisions take all concerns into consideration (customer features, tech debt, security, performance, data, fixes, etc.). This elevates data to a first-class product. The best way to get data prioritized in a backlog is to lower the cost of production with automation and increase its value by re-using it as much as possible.

The bad: Letting data concerns compete in the backlog on equal terms can leave some neglected. This is less likely to happen with normal maintenance because automation lowers the cost. It is more likely to happen with the initial investment needed to create data products. Critical data products can be tackled with ad-hoc projects, but not all data products have enough organizational visibility to achieve this.

Controversial alternative: Consumers typically push for a reserved allotment in the prioritization process to avoid issues being neglected. However, setting up and managing capacity reservation is not straightforward.

LESSONS LEARNED

Design and architecture

The relative novelty of the data mesh paradigm means there is a lack of commodity software available to build the platform. Being cloud native goes a long way, but it is not enough by itself. Strong engineering capabilities are also needed because developing a business-agnostic data platform that can build a cohesive data repository iteratively is technically much more difficult than building multiple data silos aligned with each business domain.

Decision: Include modeling of data contracts in federated governance, as it is the hardest and riskiest part.

The good: Achieving the promise of a company-wide, trustworthy data repository requires interoperability. Having multiple and diverse domain experts review the models in the contracts does not guarantee this, but helps promote it. Data modeling is one of the higher-risk elements of the project because of the complexity of managing data over time. Retention makes changes to data contracts much more difficult than service contracts (APIs). Unlike APIs, old data versions need to remain supported for years, and company-wide SQL consumption means that the most pragmatic way to decommission old schema versions is usually to wait until they expire.

The bad: Federated governance overhead caused frustration with some development teams. This is highly correlated with teams that are not aware of the problems created on the consumer side by data contracts that are not explicit, not documented, or not interoperable. This is aggravated in teams that (inadvertently) share data via DB replication, because the mesh involves more (explicit) work. Although the overall workload balance is positive for producers because changes in their databases will no longer cause them problems in the form of a data crisis, some feel as if their workload has increased.

Potential improvement: As with consumers, it is not enough to communicate only at the beginning of the project. Ongoing communication needs to clearly outline why the overhead of federated governance is better than the alternatives.

Decision: Approach data mesh adoption as a challenging engineering project for both domain and platform teams.

The good: The technical solution has not had any significant issues, and there has been no need for a re-write or re-architecture of the system. While cultural change is the hardest part of the data mesh shift, this does not mean that the technical part is easy.

The bad: This approach requires a significant investment to get to a state where adoption is feasible in the first place, as well as during the data mesh project itself. For eDO, who deal with hundreds of millions of searches per day, a data mesh could not have been deployed effectively if the platform had not already been 100% on the cloud and teams had not already had substantial cloud experience. People with strong technical skills are vital, particularly at the data platform level.

Potential alternative: eDO is an ecommerce company where software engineering is a core asset and data is part of the culture. Companies where software engineering is not a core asset or that do not have an engrained data culture will need a technology partner with strong engineering skills.

Decision: Embed documentation into data contracts and tailor it to readers who are knowledgeable about the domain of the contract.

The good: Embedded documentation is less likely to become outdated because developers will see the documentation when they are changing the contract. Leaving the context of the domain out of documentation makes it easier to write and easier to read for consumers who are already familiar with it.

The bad: If documentation is attached to each contract, it cannot provide an overview of the domain. Domain concepts cannot be documented unless the same explanation is repeated in multiple contracts. This would make maintaining documentation impossible because any change in the concept would mean changing multiple contracts. For the same reasons, business processes cannot be documented in this way. At eDO, these problems existed before the data mesh but were highlighted when it was rolled out.

Potential improvement: Implement several layers of documentation. At the contract level, the current approach works and is optimal in terms of maintenance. However, more abstract layers are needed to enable full self-service consumption.

Future work

eDO does not consider the data mesh initiative finalized, even after four years of work. First, there are still teams using data silos. Second, teams' requirements for data quality and availability are increasing as they benefit from better data in the mesh. eDo has several plans for addressing these issues.

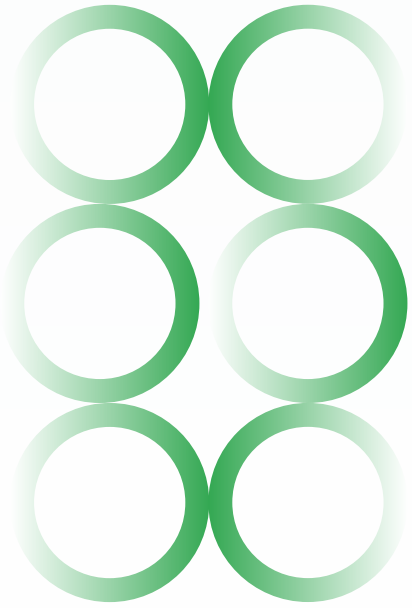
- **Improved data quality SLAs.** The existing system uses foreign keys to measure data completeness. Teams can write SQL queries to detect anomalies in data products. eDO plans to enrich the quality system to include ways to measure inconsistencies between analytical and operational data (i.e., data accuracy) caused by either bugs in the pipeline or bugs created in producer team code (e.g., when calculating the primary key of a snapshot given the primary key of a child table). This will enable better SLAs between data consumers and producers.

- **Self-motivated decommissions.** Instead of forcing analytical teams to decommission their data silos, eDO is experimenting with a more outcome-oriented way of achieving data convergence by piloting a new Data Maturity Assessment framework. This gives analytical teams an evaluation of their current data use cases in relation to data availability, completeness, accuracy, lineage, accountability, trustworthiness, stability, timeliness, and compliance. These metrics are measured from the consumer point of view. The evaluation also suggests a plan to overcome any gaps. Typically, this will involve a migration to the data mesh. The aim is to provide consumers with information on the value they gain by moving towards a better data environment.

- **Third-party aligned data contracts.** Most of eDO's data originates within the organization, but some comes from third-party providers. Governance of this data cannot operate within the same parameters, so eDO plans to create different pipelines or types of data contracts for these scenarios.

- **Faster data timeliness.** Domain events and snapshots are available in real time in the streaming platform, but it takes up to 15 minutes to get them in BigQuery. eDO plans to reduce maximum latency to one minute, to help reduce the overall mean-time-to-repair metric of the company (remember mesh data is also available for operation teams).

- **Multi-layered documentation.** One layer of documentation is not enough. Extra levels are required to document data contracts along with business domain concepts and processes. This capability is independent of data mesh implementation, but is necessary to enable self-service consumption of a large multi-domain data repository.



Summary and conclusions

To end this technical use case, here is a summary of what we have shared:

- **We presented a data journey in three stages.** This journey started with a data warehouse where functional complexity and organizational scale were manageable by a central team, before moving to decentralized, spontaneous, and disconnected data silo growth. The journey ended with a data mesh where data is distributed and aligned with domain data teams. We believe this journey is typical of any organization where the software architecture has migrated to microservices based in autonomous teams organized around business domains.
- **We demonstrated a successful and fully-functional implementation** that has improved data availability, quality, and governance in a complex real-world use case. It is too early in the existence of the data mesh paradigm to provide definitive patterns, but we believe discussing our implementation will help advance data mesh principles.
- **We shared our key takeaways from the beginning of the project.** Even though more information and help is available than when we began, adopting a data mesh is still a challenging process. It requires strong transformational skills, organizational awareness, and capable engineering, operational, and project management practices.

Adopting the data mesh paradigm is complex and high risk. Companies who use uses microservices in a similar way to eDO are likely encounter the problems described here at some point in their journey. We hope that the ideas and solutions we have shared here can help others solve challenges around their data.

Inside the eDreams ODIGEO data mesh

A platform engineering view

November 2023

Interested in getting started?

Contact us to learn more.

Google Cloud