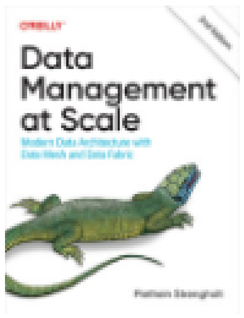


Data Management at Scale, 2nd Edition

Piethein Strengholt

Published by O'Reilly Media, Inc.

Chapter 4. Data Product Management

You may be wondering about the term *data product*—is it just another buzzword? In this chapter, we'll be cutting through the confusion to discuss what data products really are. I'll cover all the essential information you need to have for serving large quantities of data to other domains. We'll start with a disambiguation of the term because practitioners have many different interpretations and definitions. Next, we'll examine the pattern of Command Query Responsibility Segregation (CQRS) and why it should influence the design of your data product architectures. We'll discuss various design principles for data products, and you'll learn why a well-designed, integrable, and read-optimized data model is essential for your consumers. Then we'll look at data product architectures: what they are, how they can be engineered, what capabilities are typically needed, and the role of metadata. I'll try to make this as concrete as I can by using a practical example. By the end of this chapter, you'll have a good understanding of how data product architectures can help make vast amounts of data available to data consumers.

What Are Data Products?

Making the producers of data accountable for it and decentralizing the way data is served is a great way to achieve scalability. Dehghani uses the phrase "data as a product" and introduces the concept of "data products," but there's a big difference between the two. *Data as a product* describes the thinking: data owners and application teams must treat data as a fully contained product they're responsible for, rather than a byproduct of some process that others manage. It's about how data providers should treat data consumers as customers and provide experiences that delight; how data should be defined and shaped to provide the best possible customer experience.

A *data product* differs from data as product thinking because it addresses the architecture. Within the data community, you see different expectations and interpretations of how data products relate to architecture. Some practitioners say a data product isn't just a dataset containing relevant data from a specific bounded context; it also contains all the necessary components to collect and serve data, as well as metadata and code that transforms the data. This interpretation aligns with Dehghani's description of a data product as an architectural quantum, a "node on the mesh that encapsulates three structural components required for its function, providing access to [the] domain's analytical data as a product." Those three components, according to Dehghani, are code, data and metadata, and infrastructure. So, her focus is clearly on the solution architecture, which may include all the components needed for obtaining, transforming, storing, managing, and sharing data.

Some practitioners have different views on data products. For example, Accenture refers to datasets, analytical models, and dashboard reports as data products. Such a view differs significantly from Dehghani's because it focuses on the physical representation of data, and doesn't necessarily include any metadata, code, or infrastructure. So, before we can design an architecture, we first need to align on a common terminology for data products and determine what must be included or not.

Problems with Combining Code, Data, Metadata, and Infrastructure

In an ideal world, data and metadata are packaged and shipped together as data products. Ferd Scheepers, chief information architect at ING Tech Group Services, gave a presentation at Domain-Driven Design Europe 2022 on the subject of metadata management. During this presentation, he explained that metadata is critical for data management because "metadata is data that provides information about other data." To reinforce the importance, he draws an analogy to how flowers are distributed within the Netherlands.

The Aalsmeer Flower Auction is the largest flower auction in the world. From Monday to Friday, some 43 million flowers and 5 million plants are sold every day. Traders from all over the world participate in the auction each working day. On arrival, flowers are labeled and sent to designated facilities. For buyers, the entire process is seamless: you digitally place orders, submit quantities, and provide a price and a destination. The great part of the story is that the auction is fully metadata-driven. All flower boxes are equipped with barcodes that include essential information about their contents, such as country and city of origin, quantities, starting bid price, weight, production and expiration dates, producer's name, and so forth. When flowers are sold, another barcode is added with information for shipping: buyer ownership details, destination, shipping instructions, and so forth. Without metadata, the flower auction wouldn't be able to function.

Within the flower auction, flowers and metadata are always managed and distributed together because the physical objects are connected to each other. However, in the digital world, it doesn't work that way! Let me provide some examples to make this clearer. If you use a central data catalog for managing and describing all your data, then data and metadata are managed apart from each other. In this example, all the metadata sits in your data catalog and all the physical data is stored elsewhere, likely in many other locations. The same separation applies, for example, for a metadata-driven ingestion framework that manages the extract and load process of data. Such a framework doesn't crawl metadata from hundreds of data product containers before it starts processing data. A metadata-driven ingestion framework, as an architectural component, generally manages metadata in a stand-alone database, which is separated from the data itself. It uses this metadata to control the process and dependencies through which data is extracted, combined, transformed, and loaded.

The same goes for lineage and data quality information. If the associated metadata were to be packed within the data product container itself, you would need to perform complex federated queries across all data products to oversee lineage or data quality from end to end. Alternatively, you could replicate all the metadata to another central location, but that would dramatically overcomplicate your overall architecture. And how would you handle security, privacy classifications, sensitivity labels, or ownership information for the same semantic data that has several physical representations? If you were to strongly glue metadata and data together, the metadata would be duplicated when data products are copied because both sit in the same architecture. All metadata updates would then need to be carried out simultaneously over several data product containers. This would be a complex undertaking, requiring all of these containers to be always available and accessible. You could implement an overlay architecture to carry out these updates asynchronously, but this would again dramatically overcomplicate your overall architecture.

To conclude, viewing a data product as a container gluing code, data and metadata, and infrastructure together is actually far too naive. The definition of a data product needs a revision that better reflects the reality of how data platforms work these days.

Data Products as Logical Entities

If the previous definition isn't a good one, how should we define a data product? I firmly believe that managing data and technology should be done from different architectural viewpoints: one that addresses the concerns of managing data, and another that addresses the concerns of managing the underlying technology architecture. Figure 4-1 shows what this might look like.

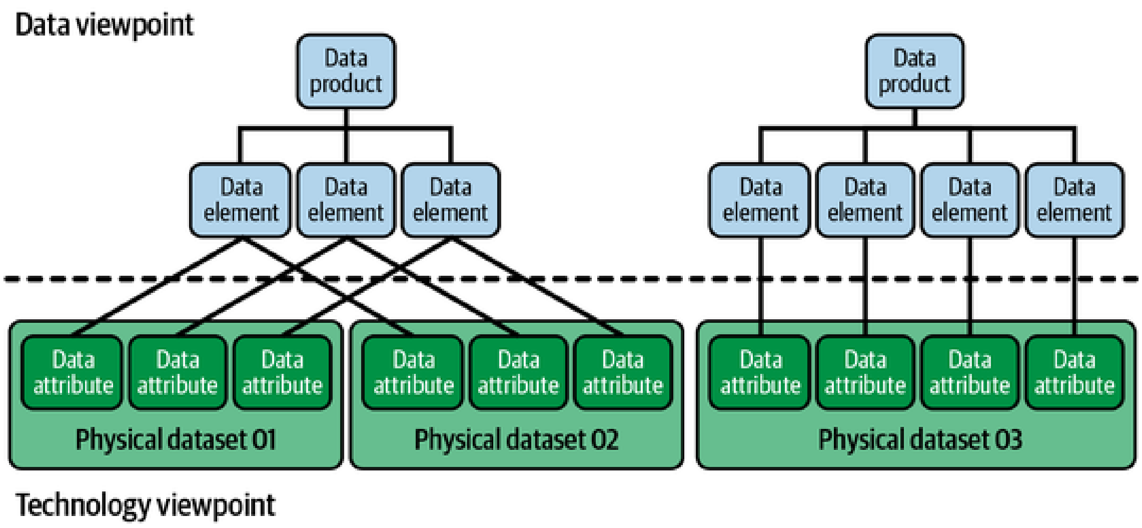


Figure 4-1. Separate viewpoints for managing the data and the technology

Why would you separate the concerns of managing data and technology? First, because you can make the architecture more cost-effective and reduce the overhead of managing infrastructure complexity. You enable scenarios of using "data product architectures" in which multiple "data products" are managed.

Second, a separation of data and architecture better facilitates scenarios in which the same (semantic) data must be distributed. Distributing data might be needed, for example, when performing cross-domain data validation or enrichment. Duplicating and preprocessing data is also sometimes required for facilitating multiple use cases at the same time; for example, when storing the exact same data in Parquet and Delta Lake file formats. When you do this, you duplicate data without changing the underlying semantics. This scenario is reflected in Figure 4-1: physical datasets 01 and 02 share the same semantics, and the individual physical data attributes link to the same elements.

Third, you avoid tight coupling between data and metadata. Imagine a situation in which data and metadata are always stored together in the same container. If, for example, owners, business entities, or classifications change, then all corresponding metadata within all data product architectures will have to change as well. In addition to that, how do you ensure consistent data ownership? If you duplicate metadata, there's a potential risk of data ownership changing as well. This shouldn't be possible for the same semantic data that has different physical representations in different locations.

Separating data and metadata changes the definition of a data product: *a data product is an autonomous logical entity that describes data that is meant for consumption*. From this logical entity you draw relationships to the underlying technology architecture—that is, to the physical locations where read-optimized and consumer-ready physical data assets are stored. The data product itself includes a logical dataset name; a description of the relationships to the originating domain, the unique data elements, and business terms; the owner of the dataset; and references to physical data assets (the actual data itself). It's semantically consistent for the business, but it could have multiple different shapes and representations on a physical level. These changes require a data product to be defined as technology-agnostic. This metadata is kept abstract for the sake of flexibility.

Tip

In Chapter 9, I'll make the logical viewpoint of a data product concrete by showing screenshots of a metamodel for data product management. If you don't want to wait to learn how this works, I encourage you to read "The Enterprise Metadata Model", then come back and continue reading here.

When you define data products as logical entities, you're able to describe, classify, label, or link data (e.g., to domains, data owners, organizational metadata, process information, and other metadata) without being specific on the implementation details. One level below a data product, there are data elements: atomic units of information that act as linking pins to physical data, interface metadata, application metadata, and data-modeling metadata. To describe the physical data, you need technical metadata, which includes schema information, data types, and so on.

Now that we have a clear idea of what data products are, let's continue by examining what impacts their design and architecture. We'll start with CQRS, then look at read-optimized modeling and other design principles. After all this, we'll zoom out a bit to examine the data product architecture.

Data Product Design Patterns

Changing the definition of a data product doesn't mean we should abandon the concept of managing data as a product. Application owners and application teams must treat data as self-contained products that they're responsible for, rather than a byproduct of some process that others manage. Data products are specifically created for data consumers. They have defined shapes, interfaces, and maintenance and refresh cycles, all of which are documented. Data products contain processed domain data shared with downstream processes through interfaces in a service level objective (SLO). Unless otherwise required, your (technical) application data should be processed, shaped, cleansed, aggregated, and (de)normalized to meet agreed-upon quality standards before you make it available for consumption.

In Chapter 1, you learned about the engineering problems of limiting data transfers, designing transactional systems, and handling heavily increased data consumption. Taking out large volumes of data from a busy operational system can be risky because systems under too much load can crash or become unpredictable, unavailable, or, even worse, corrupted. This is why it's smart to make a read-only and read-optimized version of the data, which can then be made available for consumption. Let's take a look at a common design pattern for this: CQRS.

What Is CQRS?

CQRS is an application design pattern based on making a copy of the data for intensive reads.

Operational commands and analytical queries (often referred to as *writes* and *reads*) are very different operations and are treated separately in the CQRS pattern (as shown in Figure 4-2). When you examine the load of a busy system, you'll likely discover that the command side is using most of the computing resources. This is logical because for a successful (that is, durable) write, update, or delete operation, the database typically needs to perform a series of steps:

1. Check the amount of available storage.
2. Allocate additional storage for the write.
3. Retrieve the table/column metadata (types, constraints, default values, etc.).
4. Find the records (in case of an update or delete).
5. Lock the table or records.
6. Write the new records.
7. Verify the inserted values (e.g., unique, correct types, etc.).
8. Perform the commit.
9. Release the lock.
10. Update the indexes.

Reading the database, compared to writing, takes a smaller amount of computing resources because fewer of those tasks have to be performed. For optimization, CQRS separates writes (commands) from reads (queries) by using two models, as illustrated in this figure. Once separated, they must be kept in sync, which is typically done by publishing events with changes. This is illustrated by the "events arrow" (the lightning bolt icon) in Figure 4-2.

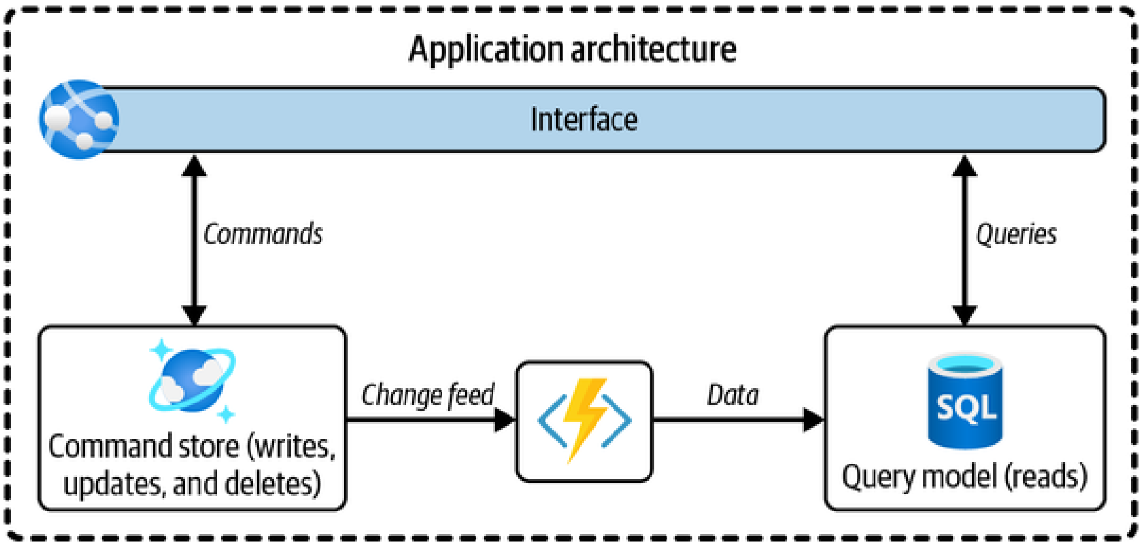


Figure 4-2. An application that uses CQRS separates queries and commands by using two different data models: a command model for the transactions and a query model for the reads

A benefit of CQRS is that you aren't tied to the same type of database for both writes and reads.[1] You can leave the write database objectively complex but optimize the read database for read performance. If you have different requirements for different use cases, you can even create more than one read database, each with an optimized read model for the specific use case being implemented. Different requirements also allow you to have different consistency models between read stores, even though the write data store remains the same.

Another benefit of CQRS is that there's no need to scale both the read and write operations simultaneously. When you're running out of resources, you can scale one or the other. The last major benefit of not having a single model to serve both writes and reads is technology flexibility. If you need reading to be executed very quickly or need different data structures, you can choose a different technology for the read store while still respecting the ACID database properties (atomicity, consistency, isolation, and durability), which are needed for the command store.

Although CQRS is a software engineering design pattern that can help improve the design process for specific (and possibly larger) systems, it should greatly inspire the design and vision of your data product architecture. *The future model of an architecture must be that at least one* read data store *per application is created or used whenever other applications want to read the data intensively.* This approach will simplify the future design and implementation of the architecture. It also improves scalability for intensive data consumption.

Note

CQRS has a strong relationship with *materialized views*.[2] A materialized view inside a database is a physical copy of the data that is constantly updated by the underlying tables. Materialized views are often used for performance optimizations. Instead of querying the data from underlying tables, the query is executed against the precomputed and optimized subset that lives inside the materialized view. This segregation of the underlying tables (used for writes) and materialized subset (used for reads) is the same segregation seen within CQRS, with the subtle difference that both collections of data live in the same database instead of two different databases.

Read Replicas as Data Products

Using replicas as a source for reading data isn't new. In fact, every read from a replica, copy, data warehouse, or data lake can be seen as some form of CQRS. Splitting commands and queries between online transaction processing (OLTP) systems, which typically facilitate and manage transaction-oriented applications, and an operational data store (ODS, used for operational reporting) is similar. The ODS, in this example, is a replicated version of the data from the OLTP system. All these patterns follow the philosophy behind CQRS: building up read databases from operational databases.

Note

Martin Kleppmann uses the "turning the database inside-out" pattern, another incarnation of CQRS that emphasizes collecting facts via event streaming. We'll look at this pattern in Chapter 6.

Data product architectures should follow the CQRS philosophy too. They take the position of a replicated copy of the data and are used to allow data consumers to read intensively. They're highly governed and inherit their context from the domains and underlying applications. At a high level, this means that whenever data providers and data consumers want to exchange data, as shown in Figure 4-3, a data product architecture must be used.

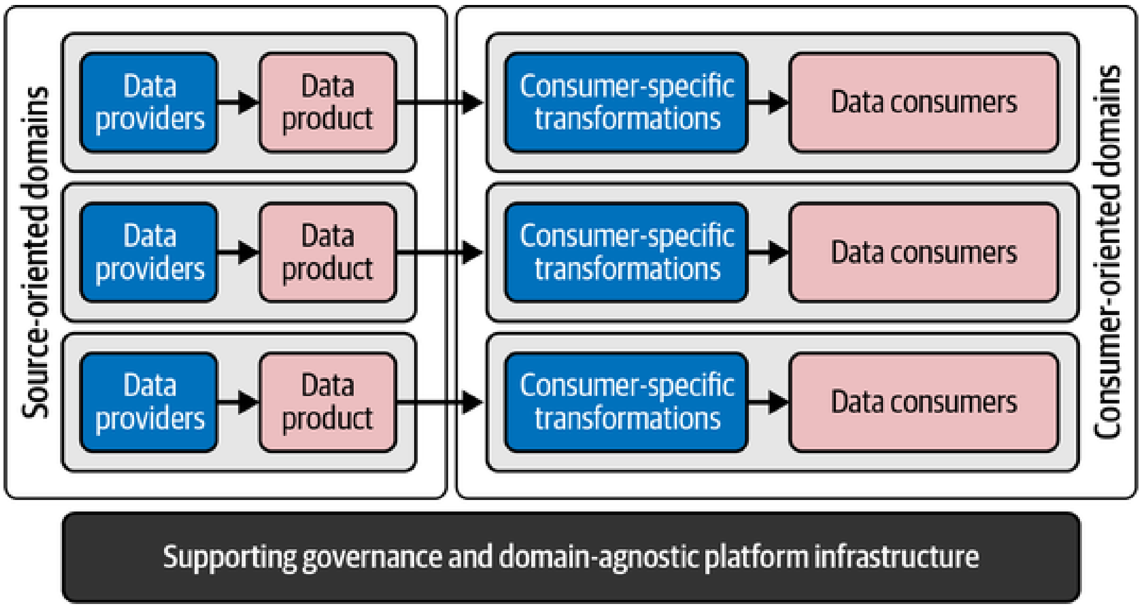

Source-oriented domains
Data providers
Data product
Data providers
Data product
Data providers
Data product
Consumer-specific transformations
Data consumers
Consumer-specific transformations
Data consumers
Consumer-specific transformations
Data consumers
Consumer-oriented domains
Supporting governance and domain-agnostic platform infrastructure

Figure 4-3. Decentralized collaboration of data providers and data consumers

Notice that data products are positioned on the left, near data providers, and transformation steps are near data consumers. This positioning originates from the unified approach of serving data to consumers. Instead of using a single unified data model, the design is changed to provide cleaned and readily consumable versions of the domain data to all consuming applications. This data isn't meant to deliver behavior or functionality. The nature of the data, compared to the operational data, is therefore different: it's optimized for intensive readability, empowering everyone to quickly turn data into value!

Design Principles for Data Products

Creating read-optimized and user-friendly data is the way forward. This sounds simple, but in practice it's often more difficult than expected. Specifically, you need to consider how the data should be captured, structured and modeled. The difficulties arise when you have to build data products repeatedly and in parallel with many teams. You want to do this efficiently, avoiding situations where each new data consumer leads to a newly developed data product, or a long cycle of studying complex data structures and working out data quality issues. You want maximum reusability and easy data to work with! This section presents a set of design principles that are helpful considerations when developing data products. Be warned, it's a long list.

Resource-Oriented Read-Optimized Design

Analytical models that are constantly retrained constantly read large volumes of data. This impacts data product designs because we need to optimize for data readability. A best practice is to look at resource orientation for designing APIs. A resource-oriented API is generally modeled as a resource hierarchy, where each node is either a simple resource or a collection resource. For convenience, they're often called resources and collections, respectively.

For data products, you can follow the same resource-orientation approach by logically grouping data and clustering it around subject areas, with each dataset representing a collection of homogeneous data. Such a design results in the data in data products being precomputed, denormalized, and materialized. On the consuming side, this leads to simpler and faster downstream consumption because there's no need to perform computationally expensive joins. It also reduces the time it takes to find the right data because data that belongs together is logically grouped together.

When applying a resource-oriented design to your data products, then, consequently, heavily normalized or too technical physical data models must be translated into more reusable and logically ordered datasets. The result is more denormalized and read-optimized data models, similar to a Kimball star schema or Inmon data mart. This also means that complex application logic must be abstracted away. Data must be served to other domains with an adequate level of granularity to satisfy as many consumers as possible. For any linked data products, this implies that cross-references and foreign key relationships must be consistent throughout the entire set of data products. Consuming domains, for example, shouldn't have to manipulate keys for joining different datasets! Building data products using this approach consequently means that domains own the semantic logic and are responsible for how data is transformed for improved readability. Enough about this, though—let's take a look at some other best practices and design principles.

Data Products and Data Vaults

You might be wondering whether you can design your data products using a Data Vault model. The Data Vault is a data modeling design pattern used for building a data warehouse.[3] It aims to provide flexibility, maintainability, and scalability by decoupling entities with additional relationships. Due to its considerable complexity, it makes data difficult to understand by "outsiders" who haven't been trained in the technique. Additionally, it will make your architecture slower and more expensive because the methodology of fine-grained data modeling often contradicts with the underlying distributed storage architectures of large cloud vendors. It requires many consumers to go through the iterations of joining and integrating data.

Figure 4-4 compares a Data Vault schema to a simple star schema. In short, the Data Vault methodology replaces entities with hubs, links, and satellites to support constantly evolving business requirements.

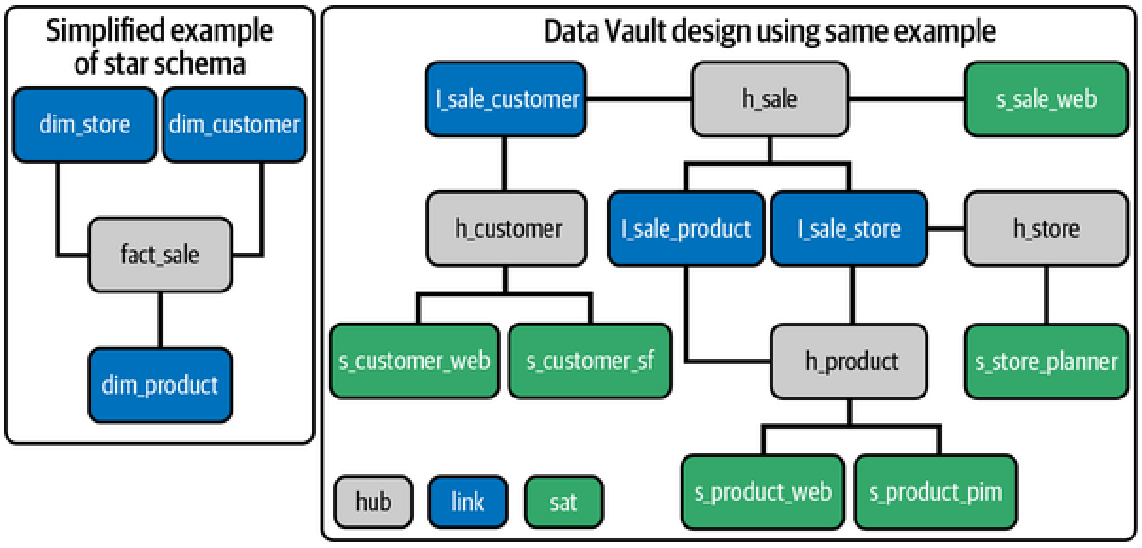


Figure 4-4. Data Vault methodology compared to simple star schema (source: PhData)

Data Product Data Is Immutable

Data product data is *immutable* and therefore *read-only*. Why must your read(-only) data stores be immutable? This property guarantees that we can regenerate the same data over and over again. If we use a read-only design, no different versions of the truth will exist for the same domain data. Data product architectures that follow this principle don't create new semantic data. The truth can be modified only in the golden source application.

Using the Ubiquitous Language

It's essential to understand the context of how the data has been created. This is where the ubiquitous language comes in: a constructed, formalized language, agreed upon by stakeholders and designers, to serve the needs of the design. The ubiquitous language should be aligned with the domain—that is, the business functions and goals. The context of the domain determines the design of the data products. A common implementation is to have a data catalog in which all of this information is stored, along with information from the domains. Publishing context and transparently making definitions clear to domains enables everybody to see and agree on the origin and meaning of data.

Some organizations require domains to use human-friendly column names in their data product physical datasets. This isn't essential, as long the mapping from the physical data model to the conceptual data model is provided. This linkage enables your domains to translate business concepts from the ubiquitous language to physical data.

Capture Directly from the Source

In my previous role, I learned that enterprises often have long data chains in which data is passed from system to system. For data management, this can be a problem because the origin isn't that clear. Therefore, you should always capture data from the system of origin. Using unique data from the golden sources guarantees that there's one source of truth for both data access and data management. This implies that data products are distinct and explicit: they belong to a single domain, which means they're isolated and can't have direct dependencies with applications from other domains. This design principle also means that domains aren't allowed to encapsulate data from other domains with different data owners, because that would obfuscate data ownership.

Clear Interoperability Standards

It's important to define how data products are served to other domains. A data product could be served as a database, a standardized file format, a graph, an API endpoint, an event stream, or something else. In all cases, you should standardize the interface specifications. If all of your domains define their own set of standards, data consumption becomes terribly difficult. Therefore, standardization is critical. This requires you to clearly define your different distribution types (e.g., batch-, API-, and event-oriented), metadata standards, and so on. Be specific about what a data product architecture looks like; for example, batch data products are published in Delta format and must be registered in a data catalog.

No Raw Data

A data product is the opposite of raw data because exposing raw data requires rework by all consuming domains. So, in every case, you should encapsulate legacy or complex systems and hide away technical application data. If you insist on making raw data available, for example for research analysis, make sure it's flagged and temporary by default. Raw or unaltered data comes without guarantees.

Can Systems Make Interfaces That Expose Raw Data?

Consider allowing applications to provide raw data and consumption-optimized data simultaneously. This approach benefits data scientists and consumers who want immediate access. Interfaces holding raw data should be marked as private and come without guarantees. They should never be used as production data pipelines.

The principle of no raw data, as discussed in Chapter 2, also applies for external data providers: external parties who operate outside the logical boundaries of your enterprise's ecosystem. They generally operate in separate, uncontrolled network locations. Ask your ecosystem partners to conform to your standard or apply mediation via an intermediate domain: a consuming domain that acts as an intermediate by abstracting complexity and guaranteeing stable and safe consumption.

Don't Conform to Consumers

It's likely that the same data products will be used repeatedly by different domain teams for a wide range of use cases. So your teams shouldn't conform their data products to specific needs of (individual) data consumers. The primary design principle should be to maximize domain productivity and promote consumption and reuse.

On the other hand, data products can and do evolve based on user feedback. You should be careful here! It can be tempting for teams to incorporate specific requirements from consumers. But if consumers push business logic into a producing domain's data products, it creates a strict dependency with the provider for making changes. This can trigger intense cross-coordination and backlog negotiation between domain teams. A recommendation here is to introduce a governance body that oversees that data products aren't created for specific consumers. This body can step in to guide domain teams, organize walk-in and knowledge-sharing sessions, provide practical feedback, and resolve issues between teams.

The Dangers of Outsourcing Business Logic

Let me share a personal story. In my previous role, the head of accounting proposed making the International Financial Reporting Standards (IFRS) accounting rules an integral part of all data products. He presumed that summing up calculated outcomes is easier than calculating everything yourself. There are a number of good reasons why you should never outsource your business logic to other domains. For one, it assumes that other domains have in-depth knowledge of your domain. It also requires all domains to simultaneously carry out changes when business logic must be updated. How would you manage this in an environment with tens or hundreds of domains? You'd probably need an army of consultants to orchestrate and manage all of these activities. The correct approach is to keep business logic close to where it belongs. For IFRS, this means asking other domains to deliver to the accounting department the information it needs to make its calculations correctly. That information should preserve the original context of the providing domain, allowing other domains to consume the data as well.

Missing Values, Defaults, and Data Types

I've experienced heated debates over how missing, low-quality, or default data should be interpreted. For example, suppose the web form of an operational system requires a value for the user's birth date, but the user doesn't supply this data. If no guidance is provided, employees might fill in a random value. This behavior could make it difficult for consumers to tell whether the data is accurate or contains default values. Providing clear guidance, such as always defaulting missing birth dates to absurd or future values, such as 9999-12-31, enables customers to identify that data is truly missing.

You may also want to introduce guidance on data types or data that must be formatted consistently throughout the entire dataset. Specifying the correct decimal precision, for example, ensures that data consumers don't have to apply complex application logic to use the data. You could set these principles at the system or domain level, but I also see large enterprises providing generic guidance on what data formats and types must be used across all data products.

Semantic Consistency

Data products must be semantically consistent across all delivery methods: batch, event-driven, and API-based. To me this sounds obvious, but I still see organizations today providing separate guidance for batch-, event-, and API-oriented data. Since the origin of the data is the same for all distribution patterns, I encourage you to make the guidance consistent for all patterns. I'll come back to this point in Chapter 7.

Atomicity

Data product attributes must be atomic. They represent the lowest level of granularity and have precise meaning or semantics. That is, it should not be possible to decompose them into meaningful other components. In an ideal state, data attributes are linked one-to-one with the business items within your data catalog. The benefit here is that data consumers aren't forced to split or concatenate data.[4] Another benefit of using atomic data is that any policy rules can be decoupled from the data. If regulation forces you to reconsider sensitive labels, for example, you don't have to relabel all your physical data. With a good metadata model and atomic data, any changes should be automatically inherited from your business metadata.

Compatibility

Data products should remain stable and be decoupled from the operational/transactional applications. This requires a mechanism for detecting schema drift, and avoiding disruptive changes. It also requires versioning and, in some cases, independent pipelines to run in parallel, giving your data consumers time to migrate from one version to another.

The process of keeping data products compatible isn't as simple as it may sound. It might involve moving historical data between old and new data products. This exercise can be a complex undertaking as it involves ETL tasks such as mapping, cleaning up, and providing defaulting logic.

Abstract Volatile Reference Data

You might want to provide guidance on how complex reference values are mapped to more abstract data product–friendly reference values. This requires nuance for agility mismatches: if the pace of change is high on the consuming side, your guiding principle must be that complex mapping tables are maintained on the consuming side; if the pace of change is faster on the providing side, the guiding principle is that data product owners should be asked to abstract or roll up detailed local reference values to more generic (less granular) consumer-agnostic reference values. This guidance also implies that the consumer might perform additional work, such as mapping the more generic reference values to consumer-specific reference values.

New Data Means New Ownership

Any data that is created because of a business transformation (a semantic change using business logic) and distributed is considered new, and falls under the ownership of the creating domain. The data distribution principles discussed in this chapter should be enforced for any newly created data that is shared.

Warning

It's fundamentally wrong to classify use case–based, integrated data as (consumer-aligned) data products. If you allow use case–specific data to be directly consumed by other domains, those domains will be highly dependent on the implementation details of the underlying consuming use case. So instead, always create an additional layer of abstraction and decouple your use case from other consumers. This approach to taking consumer-specific data and turning it into a new data product allows your domains to evolve independently from other domains.

A concern when distributing newly created data is traceability: knowing what happens with the data. To mitigate the risks of transparency, ask data consumers to catalog their acquisitions and the sequences of actions and transformations they apply to the data. This lineage metadata should be published centrally. I'll come back to this in Chapters 10 and 11.

Data Security Patterns

For data security, you need to define guidance for data providers. This should include the following:

- Guidance on encapsulating metadata for filtering and row-level access. By providing metadata along with the data, you can create policies or views to hide or display certain rows of data, depending on whether a consumer has permission to view those rows.
- Guidance on tags or classifications for column-level access and dynamic data masking. These tags or classifications are used as input for policies or views.
- Guidance on efficiently storing data in separate tables for providing coarse-grained security, enhancing performance, and facilitating maintenance.

We'll talk more about these topics in Chapter 8.

Establish a Metamodel

For properly managing data products and their corresponding metadata, I highly encourage you to create a consistent metamodel in which you define how entities (such as data domains, data products, data owners, business terms, physical data, and other metadata) relate to each other.[5] When working on your metamodel, it's best to use a data catalog to enforce capture of all the necessary metadata. For example, each time you instantiate a new data product in your catalog, it can trigger a workflow that begins by asking the data provider to provide a description and connect the data product to a data owner, originating application, data domain, and so on. Next, the catalog might ask the provider to link all of the data product elements to business terms and technical (physical) data attributes. It might then run automated tests to check whether these physical locations are addressable, and ask for classifications and usage restriction information.

Allow Self-Service

Data products are about data democratization. To improve the data product creation experience, consider building a data marketplace and offering self-service capabilities for discoverability, management, sharing, and observability. For example, allow engineers to use the data catalog's REST interface so data product registration can be automated. If done properly, engineers can register their data products as part of their continuous integration/continuous delivery (CI/CD) pipelines. More guidance on this subject will be provided in Chapter 9.

Cross-Domain Relationships

In the operational world, systems are often intertwined or strongly connected. Imagine an order system through which customers can place orders. It's likely that such a system will hold data from the customer database. Otherwise, it wouldn't be able to link orders to customers.

If domains develop data products independently from other domains, how do consumers recognize that certain data belongs together or is related? To manage these dependencies, consider using a catalog or knowledge graph that describes the relationships between the datasets and references the join keys or foreign keys across data products.

When providing guidance on managing cross-domain relationships, it's important that data always remain domain-oriented. The alignment between data and data ownership must be strong, so no cross-domain integration should be performed before any data is delivered to other domains.

Enterprise Consistency

Strong decentralization causes data to be siloed within individual domains. This causes concerns over the accessibility and usability of the data, because if all domains start to individually serve data, it becomes harder to integrate and combine data on the consuming side. To overcome these concerns, you may want to introduce some enterprise consistency by providing guidance for including enterprise reference values (currency codes, country codes, product codes, client segmentation codes, and so on). If applicable, you can ask your data product owners to map their local reference values to values from the enterprise lists.

Some data practitioners will argue that the use of enterprise reference values doesn't conform to a true data mesh implementation. This is true to some degree, but without any referential integrity you'll see duplicated efforts for harmonization and integration across all domains. Note that I'm not suggesting you build a canonical enterprise-wide data model, as seen in most enterprise data warehouse architectures, but simply provide some reference data as guidance. The resulting consistency can be helpful in a large-scale organization in which many domains rely on the same reference values.

The same guidance applies for master identification numbers, which link master data and data from the local systems together. These data elements are critical for tracking down what data has been mastered and what belongs together, so you might ask your local domains to include these master identifiers within their data products. More best practices in this area will be provided in Chapter 10.

Historization, Redeliveries, and Overwrites

To meet the needs of consumers, data providers often need to keep historical data in their original context. For example, consumers may need this data to perform retrospective trend analyses and predict what will happen next. To do this well, I recommend formulating guidance based on the design of the system, the type of data, and how it should be historized for later analysis. For example, datasets with master data should always be transformed into slowly changing dimensions. We'll look at processing historical data in more detail in "Data Historization".

Business Capabilities with Multiple Owners

When business capabilities are shared, I recommend defining a methodology for handling shared data. This may involve using reserved column names within your data products to define ownership, physically creating multiple data products, or embedding metadata within your data products.

Operating Model

Success in data product development requires an operating model that ensures effective data management, the establishment of standards and best practices, performance tracking, and quality assurance. A typical data product team usually comprises a supporting architect, data engineers, data modelers, and data scientists. This team must be adequately supported and funded to build and continually improve their data products. Additionally, there must be an organizational body that institutes and oversees standards and best practices for building data products across the entire organization. I find organizations are most successful when they develop playbooks: documents that describe all of the company's objectives, processes, workflows, trade-offs, checklists, roles, and responsibilities. Such playbooks are often hosted in open repositories, allowing everyone to contribute.

Establishing good standards and best practices includes defining how teams can measure performance and quality, as well as designing how the necessary services must fit together for each consumption pattern so they can be reused across all data products. To ensure that data products meet consumers' needs and are continually improving, teams should measure the value and success of their activities. Relevant metrics may include the number of consumers of a given data product, outstanding and open quality issues, satisfaction scores, return on investment, or use cases enabled.

Now that we've explored the foundations for the necessary mindset shift of managing data as products, let's tackle the questions that architects get most often: What is a good strategy for creating data products? Should it be facilitated with a centrally provided platform, or must domains cater to their own needs?

Data Product Architecture

To significantly reduce the complexity of your architecture, you need to find the right balance between centralization and decentralization. At one end of the spectrum is a fully federated and autonomous approach, allowing every domain to host its own architecture and choose its own technology stack. However, this can lead to a proliferation of data products, interface protocols, metadata information, and so on. It also makes data governance and control much harder because every domain must precisely implement central disciplines such as archiving, lineage, data quality, and data security. Working out all these competences in the domains themselves would be a challenge and would lead to fragmented, siloed complexity.

At the other end is a federated and centrally governed approach, not with a single silo, but with multiple platform instances or landing zone blueprints sharing the same standard infrastructure services. Standardized components and shared infrastructure reduce costs, lower the number of interfaces, and improve control. Ideally, multiple data product architectures would be hosted on the same platform infrastructure, but they're logically isolated from each other.

High-Level Platform Design

Let’s unpack the architecture and components that are needed for building data products. Exposing data as a product enables data consumers to easily discover, find, understand, and securely access data across many domains. Having said that, the main focus of the common platform capabilities for data product distribution should be on capturing data, transforming data into read-optimized versions, and securely serving data to other domains. Any analytical or reporting capabilities for developing use cases for data-driven decision making will be added to the architecture later; those are discussed in Chapter 11.

In a little more depth, at a minimum your data product architecture should have the following capabilities:

- Capabilities for capturing, ingesting, and onboarding data via your input ports
- Capabilities for serving or providing data to diverse domains via your output ports
- Metadata capabilities for documenting, discovering, monitoring, securing, and governing data
- Data engineering capabilities for developing, deploying, and running the data product’s code
- Data quality and master data management capabilities
- Data marketplace capabilities for delivering a unified data shopping experience

Figure 4-5 is a logical design of a data product architecture, demonstrating how data is captured, transformed, and served to other domains. We’ll go into more detail on each individual component, but first I’ll provide a high-level overview. At the top, there’s a metadata layer that provides insights into the available data, its properties, and relationships to other data management areas. Within this layer there are metadata repositories for orchestration, observability, contract management, and so on. These are the linking pins between your data marketplace and data product architecture.

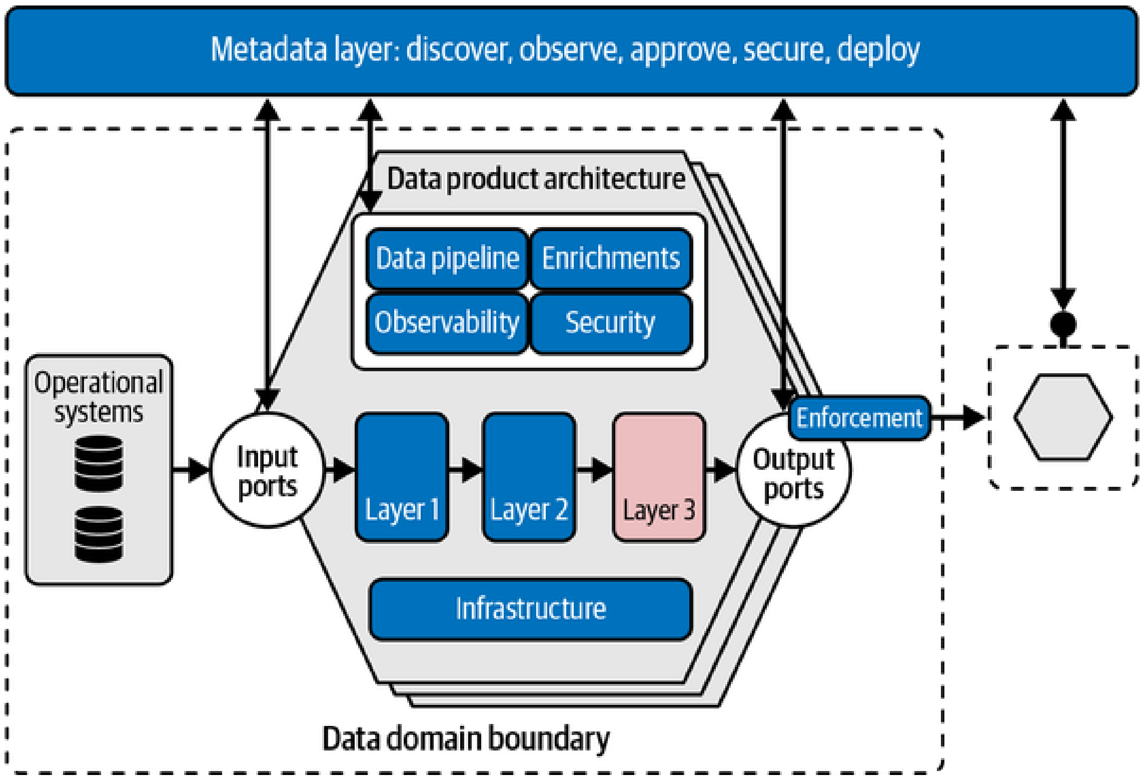


Figure 4-5. Example data product architecture

When you use shared infrastructure, different data product architectures are isolated and not directly shared between domains. Each data product architecture, and accountability, remains with the data provider. A data provider, and thus a domain, has its own data product architecture instance and its own pipeline and takes ownership of the quality and integrity of the data.

The key to engineering shared platforms and data product architectures is to provide a set of central foundational and domain-agnostic components and make various design decisions in order to abstract repetitive problems from the domains and make data management less of a challenge. In the next sections, I'll highlight the most important considerations, based on observations from the field.

Capabilities for Capturing and Onboarding Data

Data onboarding is the critical first step for successfully building a data architecture. However, the process of extracting source system data also remains one of the most challenging problems when using data at large. It requires several considerations.

Ingestion method

First, you should determine the speed at which data must be delivered or synchronized. Roughly speaking, there are two scenarios to choose from:

Batch processing

Consumers that require data only periodically can be facilitated with batch processing, which refers to the process of transferring a large volume of data at once. The batch is typically a dataset with millions of records stored as a file. Why do we still need batch processing? In many cases, this is the only way to collect our data. Most significant relational database management systems (RDBMSs) have batch components to support movement of data in batches. *Data reconciliation*, the process of verifying data during this movement, is often a concern.[6] Verifying that you have all of the data is essential, especially within financial processes. In addition, many systems are complex, with tremendous numbers of tables, and Structured Query Language (SQL) is the only proper way to select and extract their data.

Real-time data ingestion

Consumers that desire incremental or near-real-time data updates can be best facilitated with *event-driven* or *API-based ingestion*. Such processing allows us to collect and process data relatively quickly and works well for use cases where the amounts of data are relatively small, the computations performed on the data are relatively simple, and there's a need for near-real-time latencies. Many use cases can be handled with event-driven ingestion, but if completeness and large data are a major concern, most teams will fall back on the traditional batches. We'll pay more attention to this in Chapter 6.

Although real-time data ingestion has started to gain popularity, a streaming-only (Kappa) architecture will never fully replace batch processing for all use cases. Within most enterprises, I expect both ingestion methods to be used side by side.

Complex software packages

For all data onboarding, it's important to acknowledge the diversity of specialized software packages. Many are extremely difficult to interpret or access. Database schemas are often terribly complex, and referential integrity is typically maintained programmatically through the application instead of the database. Some vendors even protect their data, so it can only be extracted with support from a third-party solution.

In all of these complex situations, I recommend you evaluate for each source system whether you need to complement your architecture with additional services that allow extracting data. Such services might hide these complex source system structures and protect you from lengthy and expensive mapping exercises. They also protect you from tight coupling because typically the data schema of a complex vendor product isn't directly controlled by you. If the vendor releases a product version upgrade and the data structures change, then all your data pipelines break. Bottom line, a third-party component is worth the investment.

External APIs and SaaS providers

External API or SaaS providers typically require special attention too. I have examined situations where a full dataset was required, but the SaaS provider only provided a relatively small amount of data via an API. Other providers might apply throttling: a quota or a limit for the number of requests. There are also providers with expensive payment plans for every call you make. In all of these situations, I recommend either obtaining a tool that supports extraction from APIs or building a custom component that periodically fetches data.

A correctly designed custom implementation might route all API calls to your data product architecture first. If the same request has been made recently, the results are directly served from the data product architecture. If not, the SaaS provider's API is triggered and the results are returned and also stored in your data product architecture for any subsequent calls. With this pattern, a full collection of data can eventually be populated for serving consumers.

Lineage and metadata

For all of your data extraction and onboarding components, it's important to pay close attention to data lineage extraction. Organizations find it important to know what data has been extracted and what transformations were applied on the data. Standardization and determining how connectors would integrate with your data lineage repository is an important exercise. You should do this investigation up front, so you know what will integrate and what will not. Lineage can be hard to add retrospectively, given its tight dependence on transformation frameworks and ETL tools; it's best to select, test, validate, and standardize before starting the actual implementation.

Data Quality

Data quality is another important design consideration. When datasets are ingested into the data product architecture, their quality should be validated. There are two aspects to this.

First, you should validate the integrity of your data, comparing it to the published schemas. You do this because you want to ensure new data is of high quality and conforms to your standards and expectations, and take action otherwise. This first line of defense is the responsibility of the data owners, data stewards, and data engineers. These are the team members who create and source the data. It's up to them to ensure that the data meets the data quality requirements stipulated by regulators or by the data consumers.

Second, there are quality checks that are more federated by nature. This validation relies less on individual data product teams because other teams define what constitutes good data quality. These kinds of data quality checks are typically run asynchronously and flag or/and notify respective parties. The federated controls are usually functional checks: completeness, accuracy, consistency, plausibility, etc. Data integrity and federated data quality are two different disciplines that sometimes have their own metadata repository in which schema metadata or functional validation rules are stored. In this case, each has its own standard platform services.

For your platform design, a benefit when using shared infrastructure for data product architectures is that you can leverage the power of big data for data quality processing. For instance, Apache Spark provides the distributed processing power to process hundreds of millions of rows of data. To use it efficiently, you can use a framework to document, test, and profile data for data quality. Great Expectations and Soda, for example, are open standards for data quality that are used by many large organizations. Another advantage of managing data quality on shared infrastructure is that you can perform cross-domain referential integrity checks. Source systems often refer to data from other applications or systems. By cross-checking the integrity of the references and comparing and contrasting different datasets, you'll find errors and correlations you didn't know existed.

When data quality monitoring is implemented properly, it not only detects and observes data quality issues but becomes part of the overall data quality control framework that checks and adds new data to the existing data.[7] If for whatever reason quality drops below a specified threshold, the framework can park the data and ask the data owners to take a look and either vouch for, remove, or redeliver the data. Having a control framework for data quality means that there's a closed feedback loop that continuously corrects quality issues and prevents them from occurring again. Data quality is continuously monitored; variations in the level of quality are addressed immediately. Bear in mind that data quality issues must be resolved in the source systems, not within your data product architecture. Fixing data quality issues at the source ensures they won't pop up in other places.

The impact of data quality shouldn't be underestimated. If the quality of the data is bad, consumers will be confronted with the repetitive work of cleaning up and correcting the data. I also want to stress the importance of standardization. If individual teams decide on their own data quality frameworks, it will be impossible to compare metrics and results between domains.

Data Historization

Correctly managing historical data is critical for a business, because without this they're unable to see trends over time and make predictions for the future. Standards should also be considered for managing historical data. This section provides some background and explores a few different representations and approaches for managing this data.

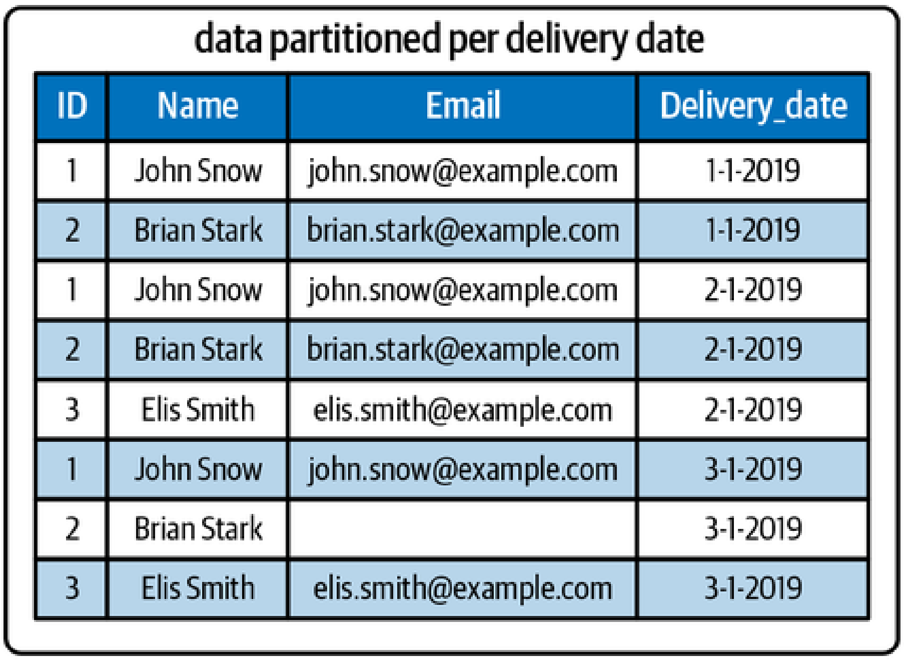
Organizing historical data is a key aspect of data warehousing. In this context, you often hear the terms *nonvolatile* and *time-variant*. Nonvolatile means the previous data isn't erased when new data is added to it. Time-variant implies that the data is consistent within a certain period; for example, data is loaded daily or on some other periodic basis, and doesn't change within that period.

Data products play an important role in storing and managing large quantities of historical data. But how would you address this in a decentralized architecture? A difference from data warehousing is that data products preserve data in its original context. No transformation to an enterprise data model is expected because data products are aligned with domains; the original (domain) context isn't lost. This is a major benefit: it means domains can "dogfood" their own data products for their own operational use cases, such as machine learning, while serving data to other domains at the same time. As a consequence, domains have an intrinsic need to care about the data they own. Ultimately, this practice of using one's own data products will improve both their quality and customer satisfaction.

Although a data product architecture is a technology-agnostic concept, it's likely that many of these architectures will be engineered with lower-cost distributed filesystems, such as data lake services. This means different ways of data processing are required for updating and overwriting master, transactional, and reference data. Therefore, I recommend considering one or a combination of the following approaches, each of which involves a trade-off between investing in and managing incoming data and the ease of data consumption. Each approach has pros and cons.

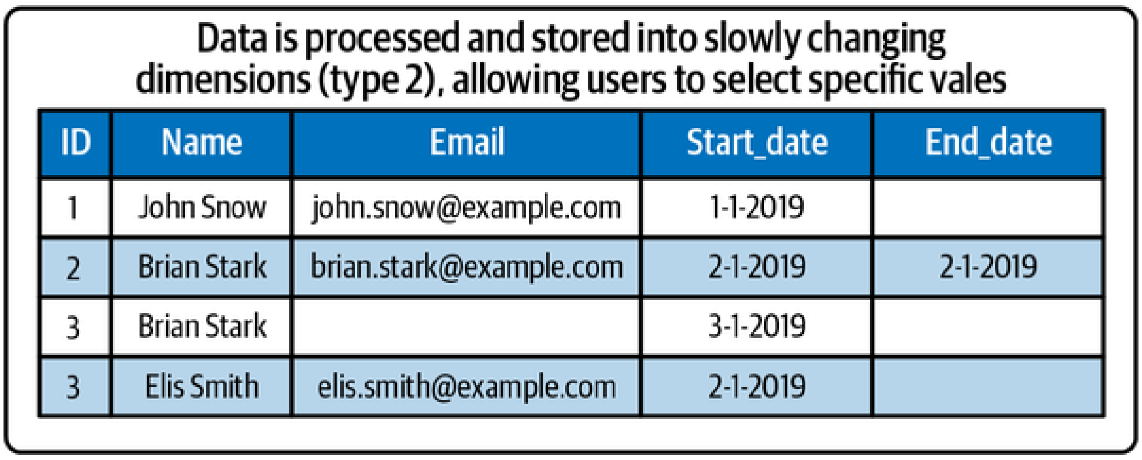
Point-in-time

The first approach for managing historical data is storing the original data as a historical reference, usually immutable. Think of a reservoir into which data is ingested using a copy activity. In this insert-only approach, it's recommended to partition the data logically, for example, using a date/time logical separation (see Figure 4-6).[8]



data partitioned per delivery date

ID	Name	Email	Delivery_date
1	John Snow	john.snow@example.com	1-1-2019
2	Brian Stark	brian.stark@example.com	1-1-2019
1	John Snow	john.snow@example.com	2-1-2019
2	Brian Stark	brian.stark@example.com	2-1-2019
3	Elis Smith	elis.smith@example.com	2-1-2019
1	John Snow	john.snow@example.com	3-1-2019
2	Brian Stark		3-1-2019
3	Elis Smith	elis.smith@example.com	3-1-2019



Data is processed and stored into slowly changing dimensions (type 2), allowing users to select specific vales

ID	Name	Email	Start_date	End_date
1	John Snow	john.snow@example.com	1-1-2019	
2	Brian Stark	brian.stark@example.com	2-1-2019	2-1-2019
3	Brian Stark		3-1-2019	
3	Elis Smith	elis.smith@example.com	2-1-2019	

Figure 4-6. Examples of how data looks when partitioned with full-dimensional snapshots or slowly changing dimensions (type 2)

Managing only full snapshots is easy to implement. At each ETL cycle, the entire output is stored as an immutable snapshot. After that, table schemas can be replaced using the location in which the new data is stored. The container is a collection of all of these point-in-time snapshots. Some practitioners might argue that this approach results in data duplication. I don't consider this a problem, because cloud storage is relatively cheap these days. Full snapshots also facilitate redelivery, or rebuilding an integrated dataset that holds all the changes made over time. If a source system discovers that incorrect data was delivered, the data can be submitted again, and it will overwrite the partition.

Note

Only using a point-in-time representation approach might be appropriate when the data that is provided already contains all the needed historical data—for example, if data is provided in such a way that it already contains start and end dates for all records.

A drawback of snapshotting full datasets is that (retrospective) data analysis is more difficult. Comparisons between specific time periods are problematic if start and end dates are not available, because consumers will need to process and store all the historical data. Processing three years' worth of historical data, for instance, might require processing over a thousand files in sequence. Data processing can take up a lot of time and compute capacity, depending on the size of the data. Therefore, it's recommended to also consider an interval approach.

Interval

Another approach for managing and presenting historical data is building up historical datasets in which all data is compared and processed. For example, you might create datasets using *slowly changing dimensions* (SCDs),[9] which show all changes over time. This process requires ETL because you need to take the existing data delivery and compare that data with any previous deliveries. After comparison, records are opened and/or closed. Within this process, typically an end date is assigned to values that are updated and a start date to the new values. For the comparison, it is recommended to exclude all nonfunctional data.[10]

Can I Design My Data Products Using Data Virtualization?

I waited until the historization discussion to bring up data virtualization. Data virtualization and data product creation aren't a great combination for several reasons:

- Data virtualization isn't complementary to data life cycle management. It can't move irrelevant data out of the underlying systems. It requires all historical data to stay in the OLTP systems, which in the long term makes data virtualization an expensive solution.
- The data virtualization layer leads to tighter coupling.[11] If source systems change, changes to the data virtualization layer are immediately required as well. Views or additional layers can help, but any change still requires coordination between the source system owners and engineers maintaining the data virtualization layer.
- Data virtualization relies on the network. In a distributed environment with a lot of networks and hops (passing through additional network devices), latency is expected to increase. Additionally, there's coupling, so if the network is down, the virtualization layer is broken.
- Data virtualization is limited by the underlying supportive technology, which is typically an RDBMS. Although data virtualization can read many database systems, in general it doesn't allow you to create document, key/value, columnar, and graph database endpoints.
- For intensive and repeatable queries data virtualization uses more computing power, because transformations are performed in real time when data is queried. Caching techniques can reduce this effect, but the amount of computing power required will always be greater when using data after it has been preprocessed for consumption.

Bottom line: access to data products can be virtualized, but you shouldn't design and develop your data products using only a data virtualization engine.

The approach of comparing intervals and building up historical data has the benefit of storing data more efficiently because it's processed, deduplicated, and combined. As you can see in Figure 4-6, the slowly changing dimension on the right side takes up half the number of rows. Querying the data, for example using a relational database, is consequently easier and faster. Cleaning the data or removing individual records, as you might need to for GDPR compliance, will be easier as well, since you won't have to crunch all the previously delivered datasets.

The downside, however, is that building up slowly changing dimensions requires more management. All the data needs to be processed. Changes in the underlying source data need to be detected as soon as they occur and then processed. This detection requires additional code and compute capacity. Another consideration is that data consumers typically have different historization requirements. For example, if data is processed on a daily basis, but a consumer requires representation by month, then they will still need to do an additional processing step.

In addition, the redelivery process can be painful and difficult to manage because incorrect data might be processed and become part of the dimensions. This can be fixed with reprocessing logic, additional versions, or validity dates, but additional management is still required. The pitfall here is that managing the data correctly can become the responsibility of a central team, so this scalability requires a great deal of intelligent self-service functionality.

Another drawback of building up historical data for generic consumption by all consumers is that consumers might still need to do some processing whenever they leave out columns in their selections. For instance, if a consumer makes a more narrow selection, they might end up with duplicate records and thus need to process the data again to deduplicate it. For scalability and to help consumers, you can also consider mixing the two approaches, keeping full-dimensional snapshots and providing "historical data as a service." In this scenario, a small computation framework is offered to all consumer domains, by which they can schedule historical data to be created based on the scope (daily, weekly, or monthly), time span, attributes, and datasets they need. With short-lived processing instances on the public cloud, you can even make this cost-effective. The big benefit with this approach is that you retain flexibility for redeliveries but don't require an engineering team to manage and prepare datasets. Another benefit is that you can tailor the time spans, scopes, and attributes to everyone's needs.

Append-only

Some data delivery styles can be best facilitated with an append-only approach. With this method, you load only new or changed records from the application database and append them to the existing set of records. This works for transactional data as well as event-based data, which we'll discuss in Chapter 6. The append-only approach is often combined with change data capture, where you set up a stream of changes by identifying and capturing changes made in standard tables, directory tables, and views.

Historization and Streaming Ingestion

Real-time data ingestion is typically handled in two different ways. In the first scenario, transformations happen on the fly as data comes in, and the result is directly appended in, for example, a NoSQL database. This whole process usually takes a couple of seconds or minutes. The main objective is to serve the data as soon as it arrives. In the second scenario, real-time data is processed periodically, for example as microbatches. The main goal here is to merge and historize all the data so data can be consumed downstream at a faster interval. A consideration when merging data is the latency of processing because processing takes time.

Defining your historization strategy

The right historization approach for a data product depends on the data types, consumer requirements, and regulations. All of the approaches for historizing and building up data products can be combined, and you may use different approaches for different kinds of data. Master data, for example, can be delivered from an application and built up using the slowly changing dimensions style. Reference data can be easily processed using snapshotting, while you might want to handle transactional data for the same system via append-only delivery. You can retain full snapshots of all deliveries while also building up slowly changed dimensions.

To be successful in data product development and design, a comprehensive strategy is the key. Best practices should focus on how to layer data and build up history within each of your domains. This layering strategy may include the development and maintenance of scripts and standard services, such as change data capture services.[12]

The best practices outlined here are meant to provide insights, but are also shared to show that data product management is a heavy undertaking. In the next section, we'll dive more deeply into some of the solution design aspects when developing an architecture. We'll do this using a real-world example while taking into consideration what we've already discussed.

Solution Design

Now that you're familiar with some best practices and design principles, it's time for something more practical: building read-optimized abstractions over your complex application data using solutions and services. The technologies and methodologies that organizations apply for developing data products typically differ significantly from enterprise to enterprise. This is also a difficult subject to address because there are no platforms or products to manage the full data product life cycle. It would take another book to describe all the different ways to implement an end-to-end data product architecture, so instead I'll focus on the key consideration points before discussing a concrete example:

- The cloud has become the default infrastructure for processing data at large because it offers significant advantages over on-premises environments. All the popular cloud platforms provide a set of self-serve data services that are sufficient to kick-start any implementation.
- Data lake services are a popular choice among all types of organizations. With data volumes increasing on a daily basis, placing data in, for example, an HDFS-compatible cloud object storage is a great way to make your architecture cost-effective. The benefits of these types of services include separation of storage and compute and the reduction of data duplication by using lightweight query services.[13]
- A popular stack is processing data with Spark, Python, and notebooks. Motivations for this pattern include a wide and active community, the benefits of open source and strong interoperability, and wide-ranging support for varying use cases, from

data engineering to data science. While Spark is an excellent choice for ETL and machine learning, it isn't optimized for performing low-latency and interactive queries. Another consideration when using notebooks is to pay attention to repetitive activities, such as data historization, technical data transformations, and schema validation. A common best practice when using notebooks is to design common procedures and a metadata-driven framework using configurable procedures for destinations, validations, security, and so on.[14]

- For data transformation and modeling, many organizations complement their architecture with extra services, like dbt. Although Spark with custom code can be used for data transformation, companies often feel larger projects can be better streamlined by templating and writing configurations. Both templating tools and notebooks are viable options, and can also complement each other. I see lots of companies that first handle data validation, technical transformation, and historization with notebooks, then integrate their data with tools like dbt.
- Orchestration and workflow automation are needed to manage the process from ingestion all the way to serving data. Although standardization is key for observability, it's a best practice to give each team freedom to develop its own local knowledge, and to pursue local priorities. Another best practice is to integrate your CI/CD and workflow processes, but keep your pipelines independent for each application or data product.
- When selecting any tools, I recommend searching for *modern data stack*. There are many popular options to choose from, varying from open source to closed source solutions.
- Metadata management services, such as data catalogs and data lineage tools, often sit outside data product architectures. These services are offered as generic services by a central authority. Considerations will be discussed in great depth in Chapter 9.

A common question with regard to data product design is how the concepts of a data lake and data mesh relate. While at first glance they might seem contradictory, the underlying technology of data lakes complements the vision of data product design. So, there's nothing wrong with using data lake technologies in a data mesh.

Another frequently asked question is whether each domain has autonomy to select its own modern data stack. Enthusiasts quickly draw a parallel between microservices and data mesh architectures, and argue that full autonomy enables each domain to make its own optimal technology choices. However, many organizations fail because domain teams make contrasting decisions. Imagine you have 25 "autonomous" teams, all implementing their data product architectures using their own tools and ways of working. If all the tools differ, how will you collect lineage, data quality metrics, monitoring statistics, and results from master data management? How can you achieve interoperability between platforms when each team uses its own standard? I'll address these questions at the end of this chapter, but it's important to note that identity, governance, and interoperability are the cornerstones of any federated pattern. Sacrifice any one of these pillars, and you quickly end up with many point-to-point solutions, which get architecturally expensive and difficult to govern. Autonomy starts with enterprise architecture, at a central level, and requires standardization. Flexibility is about loose coupling and removing dependencies. Autonomy shouldn't end up in organizational chaos, uncontrolled technology proliferation, an explosion of costs, or a combination of all these.

Real-World Example

In this section, I'll walk you through a real-world example of creating a data product architecture. This example won't be that complex, although it contains the essential ingredients for later discussion and evaluation. Let's assume the data product architecture is being designed for an organization where roughly half of the domains are operational and transactional by nature. These systems are marked as golden sources and are the starting point for data ingestion. The other domains are consumer-driven; data must be served to them.

Tip

For an extensive walkthrough of this example, including screenshots and step-by-step instructions, see my blog post "Using DBT for Building a Medallion Lakehouse Architecture".

If I were tasked to design a data product architecture for this organization on Azure, I might propose the solution seen in Figure 4-7. My starting point would be to provision a data landing zone and some resource groups with a standardized set of services for my first domain: Azure Data Factory (ADF), Azure Data Lake Storage (ALDS), and Azure Databricks. Subsequently, I would collect and harvest data using a service like ADF. Where possible, I would use prebuilt connectors. Alternatively, I would ask application owners to export and stage their data in intermediary locations. These could be, for example, Blob Storage accounts, from which the data will be picked up for later processing.

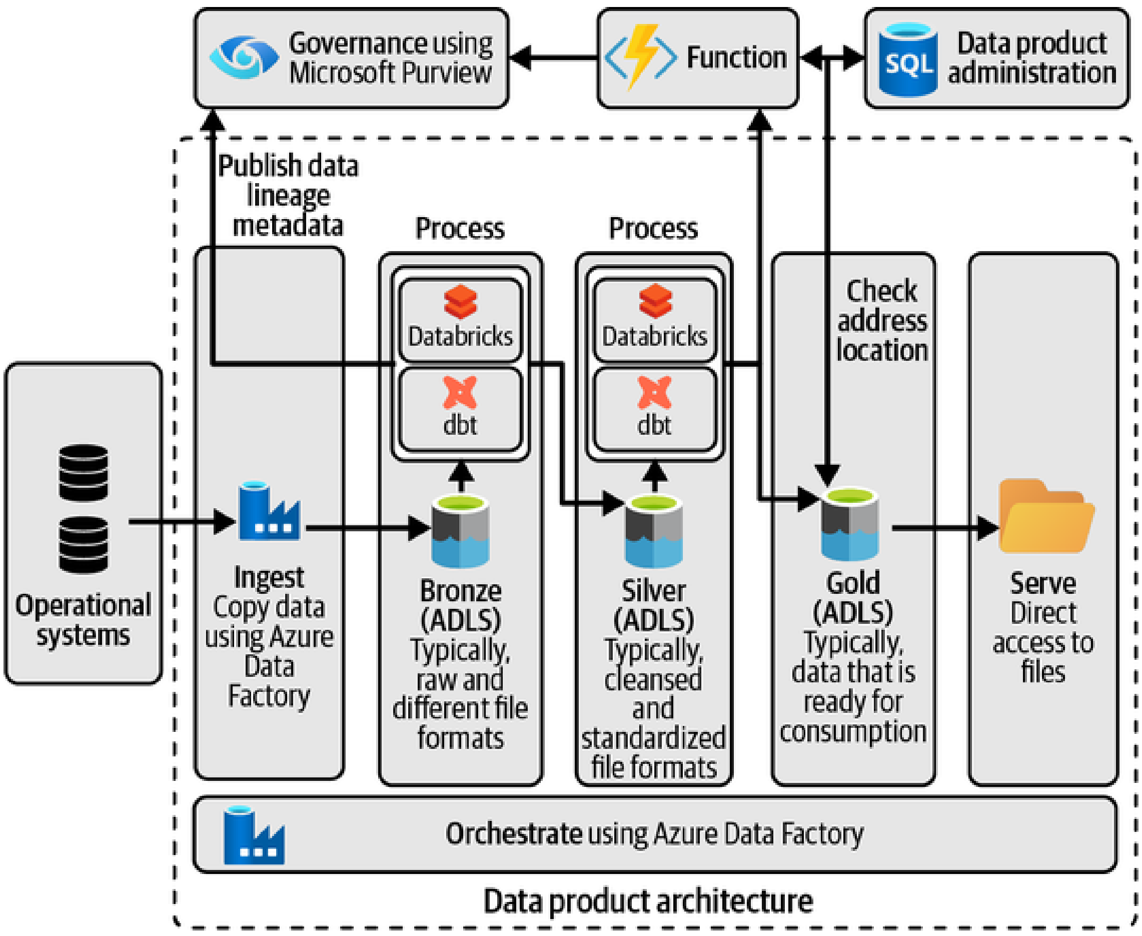

Governance using Microsoft Purview
Function
Data product administration
Publish data lineage metadata
Process
Process
Databricks
dbt
Databricks
dbt
Check address location
Operational systems
Ingest Copy data using Azure Data Factory
Bronze (ADLS) Typically, raw and different file formats
Silver (ADLS) Typically, cleansed and standardized file formats
Gold (ADLS) Typically, data that is ready for consumption
Serve Direct access to files
Orchestrate using Azure Data Factory
Data product architecture

Figure 4-7. A simple data product architecture using a lakehouse design: it features services for data ingestion, data product creation, and data governance

I would use ADLS to store data in my data product architecture. A common design pattern for grouping data is using three containers:**15**

Bronze

Raw data that might be in multiple formats and structures.

Silver

Filtered and cleaned data, stored using a standardized column-oriented data file format.

Gold

Read-optimized data that is ready for consumption by other domains. Some organizations call these read-optimized or consumer-friendly datasets data products.

After configuring my data lake, I would continue the exercise by creating my first data pipeline. First, I would use the copy activity for copying data into the Bronze layer. Data in this layer is stored using Parquet format because no versioning is required. For historizing data, a *YYYYMMDD* partitioning scheme is used. Older versions of data are kept for ad hoc analysis or debugging purposes.

Next, I would add another pipeline activity to execute notebooks and pass parameters to Databricks.**16** Databricks will use a dynamic notebook to validate data, then drop any existing schemas and create new ones using the *YYYYMMDD* partition locations.

For the Silver layer, my preferred file format is Delta, which offers fast performance and protection from corruption. Retention is enabled for the cases where you need to roll back changes and restore data to a previous version—for each operation that modifies data, Databricks creates a new table version, storing 30 days of history by default. For selecting data from the Bronze layer and transforming it using a type 2 SCD table design, I prefer dbt. I apply limited transformations when moving data to Silver, selecting only functional data. Before I select any data, I perform a set of tests to ensure the data is of high quality and meets integrity standards. One last note for this layer: if I have different sources, I treat these as separate data feeds. The integration and combination of all the data is handled in the next layers.

For moving data into the Gold layer, where it is integrated and made consumer-ready, I need to write business logic to map all the objects into user-friendly and consumable datasets. Again, I use dbt templates for this, creating a template and YML model for each logical data entity. Each template defines the selects, joins, and transformations that need to be applied, and writes to a unique folder (e.g., customer data, order data, sales data, and so on). Each output logically clusters data around a subject area. The data is materialized, and the file format is again Delta.

Finally, I would save and schedule my pipeline within ADF. For all individual steps, I would add additional steps for capturing the load date/time, process ID, number of records processed, error or success messages, and so on.

For each additional domain, I would repeat all of these steps. So, each domain would store its data products in its own domain-specific data product architecture, manage its own data pipeline, and use the same layering for building data products. Once I had the ability to do this in a controlled way at scale, I would start adding variations to this design. For example, I might have several output ports utilizing the same semantic data for different representations for different consumers.

For the governance, self-service, and security pieces of the architecture, I'd propose using a combination of an Azure SQL database, Azure Functions, and Microsoft Purview, a unified data governance service. These components can be deployed in the data management landing zone and used to catalog and oversee all data products.

For publishing data products, I would ask all domains to publish their data products via a declarative function, as part of their CI/CD pipelines. This function could, for example, take the dbt YML specifications from the Gold layer as input and store these specifications in a SQL database. Next, the function would validate that the provided location and schema information represent the universal resource identifier (URI) and structure of the data product. If so, it will make another call to publish all the information into the catalog. This final call makes the data publicly available for other consumers.

For providing data access to my consumers, I would propose using self-service data discovery and access. In the background, this will create access control lists (ACLs) for individual data products. ACLs aren't great for fine-grained column-level or row-level filtering, but for the purpose of this real-world example, they're fine. More complex scenarios will be discussed in Chapter 8.

Note

This real-world example has been simplified for teaching purposes. In reality, data product management is more complex because of the need to take into account factors like security, profiling, complex lookups, data ingestion and serving variations, and different enrichment scenarios. For example, there might be specific restrictions for archiving data, tokenization, or accessing data within a certain time frame.

Let's quickly recap the key points from this example. First, we quickly laid out the foundation for data product creation. We made the domain accountable for the life cycle of the data under its control, and for transforming data from the raw form captured by applications to a form that is suitable for consumption by external parties. Second, the architecture can be easily scaled by adding more domains with additional data product architectures. Third, we applied several data mesh principles, such as domain-oriented data ownership, data-as-a-product thinking, self-service data platforms, and federated computational governance. In the next section, we'll turn our attention to the numerous considerations you must make when scaling up.

Alignment with Storage Accounts

It's important to pay attention to the alignment of data product data and storage accounts. In the example design in the previous section, I decided to share a data product architecture with multiple data product datasets from a single domain. An alternative would have been to align each dataset with a dedicated storage account and Spark pool for data processing. Such a setup will tie together data, code, metadata, and the necessary infrastructure, and would be closer to the data product concept within a data mesh. It simplifies access management and enables automation so that you can provision processing clusters and storage accounts on the fly. If, for example, your Silver layer is temporal, you can wipe it out once it's no longer required. On the other hand, closely linking data products and infrastructure dramatically increases infrastructure overhead and the number of pipelines that you need to manage. For example, it may result in a large number of permissions and network and service endpoints that need to be managed, secured, scanned, and monitored, or generate a complex web of storage accounts because data products are often derived from the same underlying data. So, you'll need to weigh how closely you want to align your data products with your infrastructure.

What about having multiple domains share the same storage account? This is also possible, but having multiple domains share the underlying storage infrastructure requires a different container and folder structure. This is a pattern I often see at smaller organizations, because centrally managed storage eases the management burden and other issues commonly seen when moving and securing data. On the other hand, it requires all domains to share the same limits and configuration items, such as ACLs and IP address rules. It's probably also not the best approach if you need georedundancy and flexible performance.[17] Again, you'll need to consider the trade-offs.

Alternatively, you could strike a balance between locally and centrally aligned data by implementing a hybrid approach of using both domain-local and central storage accounts. In this case, each domain uses an internal, dedicated storage account in which all domain-specific processing is done (for example, in the Bronze and Silver layers). Such an account is meant for internal use only and won't be exposed to any other domains. Next to these domain-local storage accounts, there's a centrally shared storage account, which is meant for distributing the final data of the data products (created in the Gold layer) to other domains. This shared storage account is often managed by a central department. The rationale for such a design is to provide flexibility for domains while maintaining oversight of all the data products' data exchanges.

Alignment with Data Pipelines

Another important consideration is the alignment of data pipelines and data products. As you saw earlier, a data pipeline is a series of steps for moving and transforming data. It generally uses metadata and code, just like in a metadata-driven ingestion framework, to determine what operations to execute on what data. In the previous example, we used the same pipeline for all the data products, but some practitioners advocate using a dedicated pipeline for each one. So the question is, which approach is better for your situation?

To answer this question, consider the granularity of your data products and the corresponding data pipelines. Suppose you batch process the data from one application and build several coarse-grained data products (datasets) from it, each using its own data pipeline. It's likely that multiple pipelines will be processing the same underlying source system data because input data for data products often overlaps. This means some processing steps may be repeated, which makes maintenance harder. Additionally, moving specific parts of the data through different pipelines at different intervals causes integrity issues, which makes it hard for consumers to combine data later.

Having too fine-grained a setup for your data products and data pipelines is challenging as well. If an underlying source system generates hundreds of small data products each having their own data pipeline, it will be hard to troubleshoot issues and oversee dependencies: a complex web of repeated processing steps may be observed. Additionally, it will be hard for consumers to use the data, as they will have to integrate many small datasets. Referential integrity is also a big concern here because many pipelines are expected run at slightly different intervals.

To conclude, having a dedicated pipeline for each data product is usually not the ideal solution. The recommended best practice is to create one pipeline using the appropriate series of steps that takes the entire application state, copies it, and then transforms the data into user-friendly datasets.

Capabilities for Serving Data

Let's switch to the serving side of the data product architecture and see what components and services we need to provide there. For serving the data from the output ports,[18] there are different dimensions of data processing that impact how the data product architectures can be engineered. The data product architecture's design largely depends on the use cases and requirements of the consuming domains. In many cases, you're processing data that isn't time-critical. Sometimes answering the question or fulfilling the business need can wait a few hours or even a few days. However, there are also use cases in which the data needs to be delivered within a few seconds.

Warning

Data product management shouldn't be confused with data value creation, such as through business intelligence and machine learning applications. Despite the overlap, on the consumer side you generally see heavy diversification in the way data is used. Hence, data product creation must be managed apart from data value creation.

In recent years, the variety of storage and database services available have vastly increased. Traditionally, transactional and operational systems are designed for high integrity and thus use an ACID-compliant and relational database. But non-ACID-compliant schemaless databases (such as document or key/value stores) are also trending because they relax the stronger integrity controls in favor of faster performance and allow for more flexible data types, such as data originating from mobile devices, gaming, social media channels, sensors, etc. The size of the data can make a difference: some applications store relatively small amounts of data, while other applications might rely on terabytes of data.

Why Are There So Many Different Databases?

When choosing a database, there are many factors and trade-offs to consider. In addition to the structure of the data, you'll need to evaluate your needs with regard to consistency, availability, partition tolerance, and caching and indexing for better performance. There are different ways to store data and retrieve it: small chunks, big chunks, chunks that are sorted, etc. There are distributability and consistency trade-offs that can affect performance, as can features such as continued monitoring and analytics. Finally, there are nonfunctional requirements to consider, such as vendor lock-in, support, compatibility, and query languages. The bottom line is that no database can excel in all dimensions at the same time. Select the solution that best matches your requirements.

The advantage of using a data product architecture is that you can add multiple designs for the specific read patterns of the various use cases.[19] You can duplicate data to support different data structures, velocities, and volumes, or to serve different representations of the same data. You could offer a file-based endpoint for ad hoc, exploratory, or analytical use cases, a relational-based endpoint to facilitate the more complex and unpredictable queries, a columnar-based endpoint for data consumers requiring heavy aggregations, or a document-based endpoint for data consumers that require a higher velocity and a semistructured data format (JSON). You could even model your data products with nodes and relationships using a graph database.

When providing different endpoints or output ports, all must be managed under the same data governance umbrella. Accountability lies with the same providing domain. The principles for reusable and read-optimized data also apply to all data product variations. Additionally, the context and semantics are expected to be the same; for example, the notion of "customer" or "customer name" should be consistent for all data that is served. The same ubiquitous language must be used for all the data products that belong to the same data provider. We'll look at how semantic consistency is ensured across all endpoints and other integration patterns in Chapter 7.

Data Serving Services

When zooming in on the user and application interactions with the data product architectures, you'll typically find that there are a large variety of data access patterns to provide. These patterns might include ad hoc querying, direct reporting, building up semantic layers or cubes, providing Open Database Connectivity (ODBC) to other applications, processing with ETL tools, data wrangling, or data science processing. In order to support these, it's essential to give your data product architecture sufficient performance and security functions.

Note

For all data-consuming services, you want to be consistent on data access and life cycle management. This is what data contracts are for; we'll discuss these in Chapter 8.

Distributed filesystems, such as cloud-based object storage, are cost-effective when it comes to storing large quantities of data but aren't generally fast enough for ad hoc and interactive queries. Many vendors have recognized this problem and offer SQL query engines or lightweight virtualization layers to solve it. The benefit of allowing queries to be executed directly against data products is that data doesn't have to be duplicated, which makes the architecture cheaper. These tools also offer fine-grained access to data, and they make the operational data store obsolete. The data that sits in a product architecture becomes a candidate for operational reporting and ad hoc analysis. Data virtualization options are good solutions, but only fit for simple reporting use cases. If this isn't sufficient, consider alternatives such as duplicating data.

File Manipulation Service

While we're discussing data duplication and the consumption of data, some domains have use cases or applications that accept only flat files (for example, CSV files) as input. For these situations, consider building a file manipulation service that automatically masks or strips out sensitive data that consumers aren't allowed to see. This service, just like all other consumption endpoints, must be hooked up to the central security model, which dictates that all data consumption must respect the data contracts (see "Data Contracts"). It also prescribes that filters must be applied automatically and are always based on metadata classifications and attributes. We'll discuss this in more detail in Chapter 8.

De-Identification Service

When directly using data products for data exploration, data science, machine learning, and sharing data with third parties, it's important to protect sensitive data from consumers. Thanks to security methods like tokenization, hashing, scrambling, and encryption, you can use rules and classifications to protect your data. Depending on how you engineer the data product architecture, you can apply the following methods:

- Protect data at runtime during consumption. This technique protects sensitive data without making any changes to the data stored, but only when data is queried. Databricks, for example, uses a feature called *dynamic view functions* to hide or mask data when it's being accessed.
- Obfuscate or mask sensitive data before it ever lands in a data product architecture (for example, by using tokenization). The de-identified matching then happens at the stage of consumption.
- Duplicate and process data for every project. You can customize your protection by defining which classifications and masking and filtering rules you use.

Data security relies on data governance. These two disciplines will be discussed in depth in Chapter 8.

Distributed Orchestration

The last aspect that requires design considerations and standardization is supporting teams as they implement, test, and orchestrate their data pipelines. Building and automating these data pipelines is an iterative process that involves a series of steps such as data extraction, preparation, and transformation. You should allow teams to test and monitor the quality of all their data pipelines and artifacts, and support them with code changes during the deployment process. This end-to-end delivery approach is a lot like DataOps, in that it comes with plenty of automation, technical practices, workflows, architectural patterns, and tools.

When standardizing tools and best practices, I recommend that organizations set up a centralized or platform team that supports other teams by providing capabilities for tasks like scheduling, metadata curation, logging metrics, and so on. This team should also be responsible for overseeing end-to-end monitoring, and it should step in when pipelines are broken for too long. In order to abstract away the complexity of dependencies and time interval differences from different data providers, this team can also set up an extra processing framework that can, for example, inform consumers when multiple sources with dependency relationships can be combined.

Intelligent Consumption Services

Building on all of these different capabilities, you could also extend the data product architecture with a layer (fabric) for applying intelligent data transformations upon data consumption, such as automatic profiling, filtering, aligning schemas, combining, and so on. This would enable data consumers to define the data they need and receive it already modeled into the shape they want to have it in. This type of intelligent service typically is also used to deploy a semantic layer that hides complex logic and then presents user-friendly representations to business users. It heavily utilizes metadata, including many other services that are specific to data governance. A visionary approach would be to fuel your fabric with upcoming technologies such as semantic knowledge graphs and machine learning. Although this trend is relatively new, I give several recommendations for preparing to implement this approach in Chapter 9.

Direct Usage Considerations

A tough design consideration is whether data products should be able to act as direct sources for consuming domains, or if those domains need to extract and duplicate them. I'm talking here about persistent data transformations, which refer to the process by which data is written and stored in a new (data lake or database) location, outside the boundaries of the data product architecture. Avoiding the creation of uncontrolled data copies is preferred, but there may be situations where it's better to have a copy available nearby or inside the consuming application. For example:

- If the data product is used to create new data, logically this data needs to be stored in a new location and requires a different owner.
- If you want to reduce overall latency, it might be better to duplicate data so it's available locally, closer to the consuming application's destination. We'll take a closer look at this in Chapter 6, where you'll learn how to make fully controlled, distributed, materialized views.

- Direct and real-time data transformations might be so heavy that they degrade the user experience. For example, queries on data products might take so long that users become frustrated. Extracting, restructuring, and preprocessing data or using a faster database engine can alleviate this frustration.
- When the consuming application is of such high importance that decoupling is required to avoid data product architecture failure, it might make sense to duplicate the data.

In all of these cases, I recommend implementing a form of enterprise architecture control that dictates which approach is taken for each data product. Be aware that duplicated data is harder to clean, and it can be difficult to get insights on how the data is used. If the data can be copied without restrictions or management, it may be possible for it to be distributed further. Another potential problem is compliance with regulations such as the GDPR or CCPA; we'll discuss this in greater detail in Chapter 11.

Getting Started

A transition toward data ownership and data product development can be a cultural and technical challenge for many teams. For example, not all teams have the necessary knowledge, see the value of data, or are willing to invest in data. If you want to follow a natural evolution to data product development, consider the following best practices:

- Start small. Generate some excitement, and start with one or a few domains whose teams have bought in to the concept of data product development. Show success first, before scaling up.
- Create a consistent definition of what a data product means for your organization. Align this with the metamodel that is being used for metadata management.
- Governance is important, but take it slow. Keep a delicate balance to encourage productivity. Focus on foundational elements, such as ownership, metadata, and interoperability standards. Align with industry standards and best practices wherever possible.
- Your first data products should set an example for the rest of your domains. Put acceptance quality and design criteria in place and focus on dataset design. Resource-oriented API design is a great starting point for building high-quality data products.
- Don't try to build a platform for supporting all use cases at once. At the start, your platform will typically only support one type of data ingestion and consumption: for example, batch data movements and Parquet files as a consumption format.
- Allow centralization, next to decentralization. Your initial data product deployment doesn't have to immediately disrupt your organizational structure. Keep things central for a while before you align engineers with domains, or apply soft alignment to get engineers to work with subject matter experts. At the beginning, domain-specific datasets can be created with support from the central team.

When you're starting out, take it slow. Learn and iterate before moving on to the next stage. Centralization versus decentralization isn't all or nothing; instead, it is a constant calibration that includes nuances and periodic evaluations.

Wrapping Up

Data products are a big change in the way dependencies between domains and applications are managed. Anytime the boundaries of a functional concern or bounded context are crossed, teams should conform to the unified way of distributing data: serving data as data products to other teams. This way of working is important because fewer shared dependencies means less coordination between teams is needed.

At the beginning of this chapter, I drew a distinction between the concepts of "data as a product" and "data products." To recap, data as a product is the mindset that you apply when managing data at large. It's about creating trust, establishing governance, and applying product thinking; it's taking care of the data you value most and putting principles into practice. It's a mindset that your domain teams need to have. They must take ownership of their data, fix data-quality issues at the source, and serve data in such a way that it is useful for a large audience. Instead of a central team absorbing all the changes and integrating data, your domain teams should encapsulate the physical data model within their domain boundaries. This the opposite of throwing raw data over the fence and letting consumers work out data quality and modeling issues. Your domain teams should model and serve data in such a way that it offers a best-in-class user experience.

A data product is a concept you need to demystify for yourself. My recommendation is to begin by standardizing the way you talk about data product management in your organization, as there may be variations in the language and definitions used. Consider defining data products as logical constructs, rather than using a viewpoint that purely focuses on physical representations or the underlying technology architecture. At some point you may introduce a semantic model for your organization, so your data products become graphical representations from which you draw relationships to domains, data owners, originating applications, and physical representations. This may sound a bit too conceptual, but the benefits of managing your data products like this will become clearer when you read Chapter 9.

There's a caveat with data products. They are likely to suffer from similar issues to those seen in microservices architectures, where data is too fine-grained and disjointed, and must be pulled from hundreds of different locations. The solution to these problems is establishing good standards and a central authority that oversees all data product creation activities. Interoperability standards for data exchanges must be set. Coordination is required on changes and how data is modeled and served. Unfortunately, these activities aren't easy. As you learned, data product development has a strong relationship with data warehousing, which consists of many disciplines, such as data modeling (making data consistent, readable, discoverable, addressable, trustworthy, and secure). Some of these activities might be complex, like the historization aspects that we discussed in this chapter.

Data products and the data-as-a-product mindset are about *providing data*. *Consuming data* is the next stage. While certain activities and disciplines might overlap, as you'll learn in Chapter 11, the concerns of providing data and consuming data aren't the same. By managing data as products and providing the data products as domain interfaces, you eliminate dependencies and better isolate applications with contrasting concerns. This is important because fewer shared dependencies means less coordination between teams is needed.

The data product architecture should be designed collaboratively and based on the organization’s needs. Provide generic blueprints to domain teams. Allow other teams to contribute or temporarily join the platform team(s). If, for example, a new consumption pattern is required, deliver it sustainably as a reusable pattern or component. The same applies for capturing and onboarding data. When scaling up and building a modern data platform, many extra self-service application components will be required: these will likely include centrally managed ETL frameworks, change data capture services, and real-time ingestion tooling.

One final and important note: data products, as discussed in this chapter, are meant for making data available. We haven’t yet discussed how to make the data usable and turn it into real value. This aspect has different dynamics than data product creation and will have to wait until Chapter 11.

In the next chapter, we’ll look at API architecture, which is used for service communication and for distributing smaller amounts of data for real-time and low-latency use cases.

1 A drawback of CQRS is that it makes things more complex—you’ll need to keep the two models in sync with an extra layer. This could also generate some additional latency.

2 TechDifferences has an overview of the differences between a database view and materialized view.

3 Morris’s article on Data Vault 2.0 describes “the good, the bad and the downright confusing” about this design pattern.

4 Believe it or not, I once saw a legacy application in which multiple customer values were concatenated into a single field using double pipe operators, like Firstname||Middlename||Lastname.

5 A metamodel is a framework for understanding what metadata needs to be captured when describing data.

6 *Reconciliation* can be implemented in many different ways. For example, you can compare row counts, use column checksums, slice large tables and reprocess them, or use data comparisons tools such as Redgate.

7 For a discussion of how Azure Synapse Analytics and Microsoft Purview can be integrated together to intelligently process and validate data using Great Expectations, see my blog post “Modern Data Pipelines with Azure Synapse Analytics and Azure Purview”.

8 Partitioning is a common technique for organizing files or tables into distinct groups (partitions) to improve manageability, performance, or availability. It’s usually executed on data attributes, such as geographical (city, country), value-based (unique identifiers, segment codes), or time-based attributes (delivery date, time).

9 A slowly changing dimension is a dimension that stores and manages both current and historical data over time in a data warehouse. Dimensional tables in data management and data warehousing can be maintained via several methodologies, referred to as type 0 through 6. The different methodologies are described by the Kimball Group.

10 For example, within your processing framework you can use the exclude_from_delta_processing parameter on an attribute level to exclude data from being compared.

11 Some database vendors provide a database (virtual) query layer, which is also called a data virtualization layer. This layer abstracts the database and optimizes the data for better read performance. Another reason to abstract is to intercept queries for better security.

12 One of my blog posts has a simple sample script for processing SCD tables in Delta format using Spark pools.

13 By separating storage from compute, you can automatically scale infrastructure to match elastic workloads.

14 See my blog post "Designing a Metadata-Driven Processing Framework for Azure Synapse and Azure Purview" for details on this concept.

15 If you plan to set up a data lake, the "Hitchhiker's Guide to the Data Lake" is a great resource with guidance and best practices.

16 Microsoft has developed a solution accelerator together with the OpenLineage project. This open source connector transfers lineage metadata from Spark operations in Azure Databricks to Microsoft Purview, allowing you to see a table-level lineage graph. The project is hosted on GitHub.

17 Georedundancy is achieved by distributing critical components or infrastructures (e.g., servers) across multiple data centers in different geographic locations. For example, for cloud storage you can choose between conventional hard disks and faster solid state drives

18 The recommendation for a data mesh is to share data products using highly standardized interfaces, providing read-only and read-optimized access to data. These interfaces are also known as *output ports*.

19 When using an enterprise data warehouse for data distribution, it's often difficult to facilitate various forms and dimensions of the data. Enterprise data warehouses are generally engineered with a single type of technology (the RDBMS database engine) and thus don't allow much room for variations.