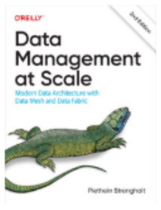

Data Management at Scale, 2nd Edition

Piethein Strengholt

Published by O'Reilly Media, Inc.

Chapter 11. Turning Data into Value

In this chapter, we'll discuss most people's favorite part of data management: turning data into value with intelligent services, such as business intelligence and advanced analytics. We'll do this by focusing more on the consuming side of the architecture, while keeping in mind everything we've already covered about the data-providing side of the architecture. You learned that it takes a village to make data available in a safe and controlled way, with proper governance and security.

Let me immediately put my cards on the table: on the consuming side, use cases often need data to be combined from different domains and data products. This interaction is complex on both the technical and organizational levels. Therefore, I advocate for managing consumer-focused data differently from data products. The core concern here is bringing data together and combining it for different business use cases. It's about turning data into value and using services that may be specific to the use case at hand. So, a large part of this chapter will be devoted to this topic.

The data-consuming side is also the most complex part of the architecture because each business problem has its own unique context and requirements. A large variety of tools, disciplines, roles, and activities are expected on the consuming side, which makes standardization difficult.

Business requirements always come first. Turning data into insights or actions requires understanding how information flows and using that knowledge to identify business opportunities. Use cases may start as one-off projects, but ideally you'll develop maintainable solutions based on your key data that deliver constant value for the organization. Depending on your business requirements, you might use different techniques, tools, and frameworks.

You'll also need to consider nonfunctional requirements. It takes many different database technologies to accommodate a large variety of complex use cases. Depending on the use case in question, you might pick one or several of these. Additionally, there are variations in the types and velocities of transformations you make to your data, optimizations for parallelization, and consumption patterns, all of which affect your end result. Finally, there are different business intelligence and machine learning patterns for delivering business value (actionable insights), which are complex and consist of many components. We'll look at all of these topics in this chapter.

To address all of these challenges, I suggest creating standardized, reusable patterns and building blocks. This approach is similar to how data product architectures are managed. However, for consuming data, you need to define different patterns, set principles for managing self-service data and managed data, and get clarity on considerations for selecting data stores, building pipelines, and using business intelligence and machine learning services. The consuming architecture we unpack in this chapter is built upon the data foundation discussed in Chapter 2 and works with the governance model discussed in Chapter 8.

The Challenges of Turning Data into Value

I've seen it happen all too many times: in the absence of specific analytical capabilities and with the need for greater speed, business users start to purchase and deploy their own tools themselves. They create point-to-point interfaces and hook analytical tools up directly to data warehouses and operational systems. They also begin to extract, transform, and load data directly into their self-managed business environments. As new use cases and opportunities pop up, more and more of these tools are brought in to solve just one problem, without regard to related issues. The inevitable result: the architecture becomes complex, expensive, and difficult to manage.

To avoid business users steering your data management architecture into chaos, you need to create controlled environments that integrate deeply with the underlying data management services and support the federated model of many business domains, while also addressing the needs of different audiences and user groups in a *self-supported* manner. We'll explore how this works—but first, let's recap what we've covered and the part of the architecture we'll be concentrating on here.

So far, we've mostly been focusing on making data ready and available for consumption. In Chapter 4, we discussed the design and management of data products. In the chapters that followed, we continued to explore the distribution and management of data by delving into areas such as API and event management, metadata, master data management, governance, and security. All of these topics fall into the realm of the data providers, on the left side of the architecture depicted in Figure 11-1.

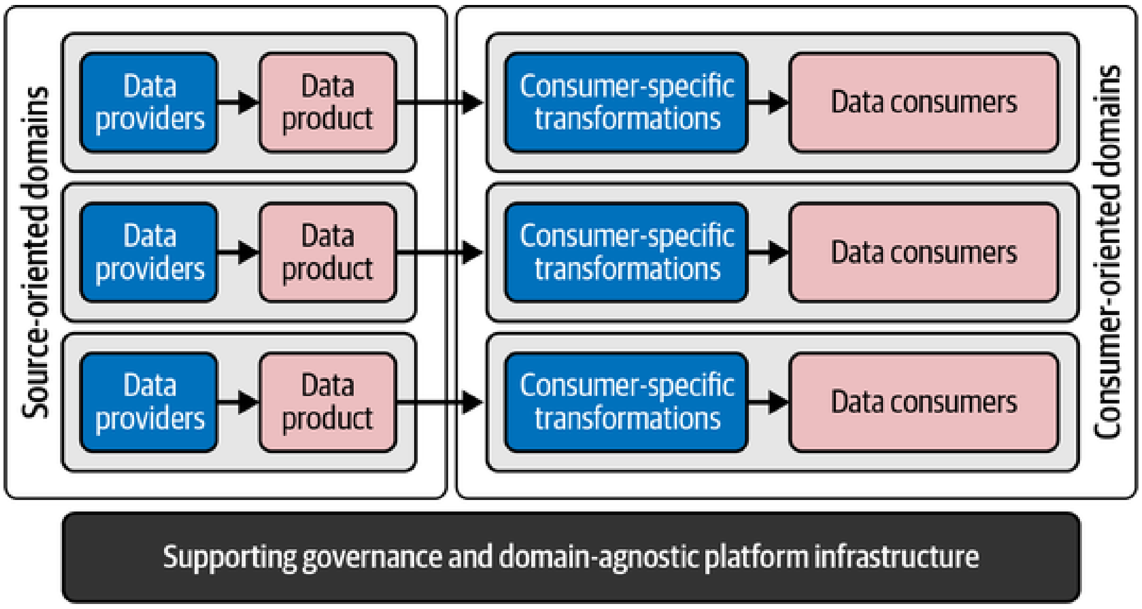
Source-oriented domains
Data providers
Data product
Data providers
Data product
Data providers
Data product
Consumer-specific transformations
Data consumers
Consumer-specific transformations
Data consumers
Consumer-specific transformations
Data consumers
Consumer-oriented domains
Supporting governance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

Figure 11-1. Decentralized collaboration of data providers and data consumers

On the right side of the architecture in Figure 11-1 are the consumer-aligned domains. As you learned in Chapter 2, these domains consume the data supplied by the providing domains, transforming it as needed to fit their business requirements. Consumer-aligned domains are unique contexts. Therefore, this consumer-specific transformation step is *always* required. The data consumers are the experts on their own use cases, so they're responsible for this step; they set the requirements and know what data must be integrated for what business (analytical) purpose. How do they go about this?

Data consumption, and the transformation that comes with it, often starts with duplication of the data. However, this isn't always necessary. Indeed, in a large architecture, you want to limit the number of copies of the same data because maintaining them can be costly and difficult. Security is a concern, and data may change or become outdated. Therefore, as a consumer, your first consideration should be whether you can use data products from data providers directly, without copying or persisting them.

Data products, as described in Chapter 4, are designed to serve out large volumes of immutable data repeatedly to consumers. Because of their underlying performance and access patterns, they can be used for directly retrieving data at the query time (e.g., for reporting, ad hoc analysis, analytics, or self-service activities). In a direct data consumption pattern, data is read but no new data is created. Consuming applications use the data products directly as their sources, perhaps performing some lightweight integration tasks based on mappings between similar data elements. They may change the context and temporarily create new data, but they don't require these results to be stored elsewhere. The benefit of this direct usage model is that it doesn't require creating new (complex) data models. You don't extract, transform, and load data into a new database. Transformations happen on the fly, but these results don't need a permanent new home.

To support this direct read consumption pattern, the underlying data product architecture must deliver sufficient performance. The problem, however, is that the consumers' needs may surpass what your data product architecture can offer. For example, the amount of data that needs to be processed may exceed what the data product platform can handle. In such a case, one solution might be to incrementally bring the (historical) data over to a new location, process it, and preoptimize it for later consumption. You may also run into problems when multiple data products need to be combined and harmonized. This typically requires many structural changes, orchestrating tasks, or first bringing the data closer together. Making users wait until all of these tasks are finished before presenting the data to them will negatively affect the user experience. In other cases, there's a clear need for the creation of new data; for example, when complex business logic is used to generate new business insights. To preserve these results for later analysis, you need to retain this information somewhere. Rather than using data products directly, in such cases you will need to use a different data consumption pattern: creating domain data stores.

Domain Data Stores

Newly created data should be managed the same way as data products are managed by domains, which raises two questions. First, how should we refer to the architecture in which the newly created data is managed? Some practitioners use the terms *data applications* and *analytical applications*, but to me, these terms could mean anything; they merely describe the architecture in which data products are consumed and integrated. For example, an analytical application could be any type of business application.

The second question is how to refer to this newly created data. Some people use the term *data products* for datasets created on the consuming side of the architecture as well, but I find that usage misleading. As we've seen, data products have certain universal characteristics: they inherit the ubiquitous language from the underlying providing domain; they remain stable and compatible after creation; they're decoupled from the application; they're designed for intensive readability. These characteristics contrast with how consumer-aligned domains in general operate. Data that has been transformed for a specific use case is tightly coupled to that use case: it doesn't necessarily have to be stable, and it should take on the structure and shape that best complement the requirements of the use case. Its design may vary depending on the specifics of the supporting system(s) and database technology in use (row-oriented database, time series database, columnar store, etc.).[1]

Note

Some organizations use the term *consumer-aligned data products* for data that specifically targets unique business needs. This term might work, as long as you strictly define the scope and purpose of this data. However, in casual usage it's likely to be shortened to just "data products," leading to confusion as the distinction is lost.

In conclusion, building and serving data products is unlike consuming and integrating data for business use cases. There might be some overlap in the services and techniques used, but the underlying concerns are different. Providing data—creating data products—is about serving data to others. It requires adhering to set principles to guarantee stable, easy, and safe consumption. Consuming data is about utilizing data for value creation. It involves data-driven decision making, which may require many extra tools and services. The consumed and transformed data might become a candidate for data product creation, but this doesn't mean that turning data into value is the same as managing data as a product.

To facilitate the management of that data and the related services and business use case application(s), I propose a new building block: *domain data stores* (DDSs). As with data products in Chapter 4, I advocate for approaching the management of the data and corresponding architecture (technology services) from two perspectives. The technology viewpoint addresses the underlying solution architecture needed for consuming, integrating, and using data. At this level, you'll find ETL, BI, ML, and other services, as well as products such as databases. Second, the data viewpoint addresses the consumed and newly created data. On this level, you may classify data as one of the following:

Copied data

Data with the same semantics as the original data product data, although it might have gone through some technical or structural changes.[2]

Integrated data

Data that has been consumed, combined, and transformed into a new context.[3] It has been specifically designed for the consuming use case.

(Newly created) data product data

Integrated data that will be made available to other domains to consume. This data is similar to data product data, and therefore follows all the principles discussed in Chapter 4 for managing data as a product. The difference with data products from the consumer side is that this data has already gone through the structural changes we described, and it is being shared after going through all of those steps.

Tip

You could use your metamodel to classify data and set principles for each type. For example, for newly created data, lineage always must be present.

Correctly classifying data is important because different principles apply to each type. For example, integrated data should never be directly shared with other domains as it is tightly coupled to the underlying use case. Needless to say, any disruptive change will directly impact data consumers. As a result, when such data must be shared with other domains, it first must become a new data product.

For managing the consumption side as a whole—both technology and data—I propose using DDSs. As shown in Figure 11-2, DDSs are similar to data product stores. Their role is to store and manage the newly created data that is needed for facilitating the consumer's use case.

DDSs importing and storing data, as domain C in Figure 11-2 shows, is necessary only in cases where consumers can't use data products from other domains directly. The underlying technology for domains is selected by use case, based on cost, agility, knowledge, and skills, as well as nontechnical requirements such as performance, reliability, data structure, data access, and integration patterns.

DDSs are architectures on their own. Just like data product architectures, DDSs are typically offered as blueprints that you can use to quickly spin up environments for domain teams. The blueprint may deploy a set of services you can use for data analytics and data science. It might also include ETL services, scheduling tools, CI/CD services, and so on. Any services your teams don't require should be explicitly disabled.

It is important to understand that a DDS can play the roles of data consumer and data provider at the same time. In such a situation, the DDS, in the role of consumer, consumes, integrates, and transforms the consumed data into newly integrated data. As provider, it provides the newly created data products to other data consumers. This process is shown in Figure 11-3.

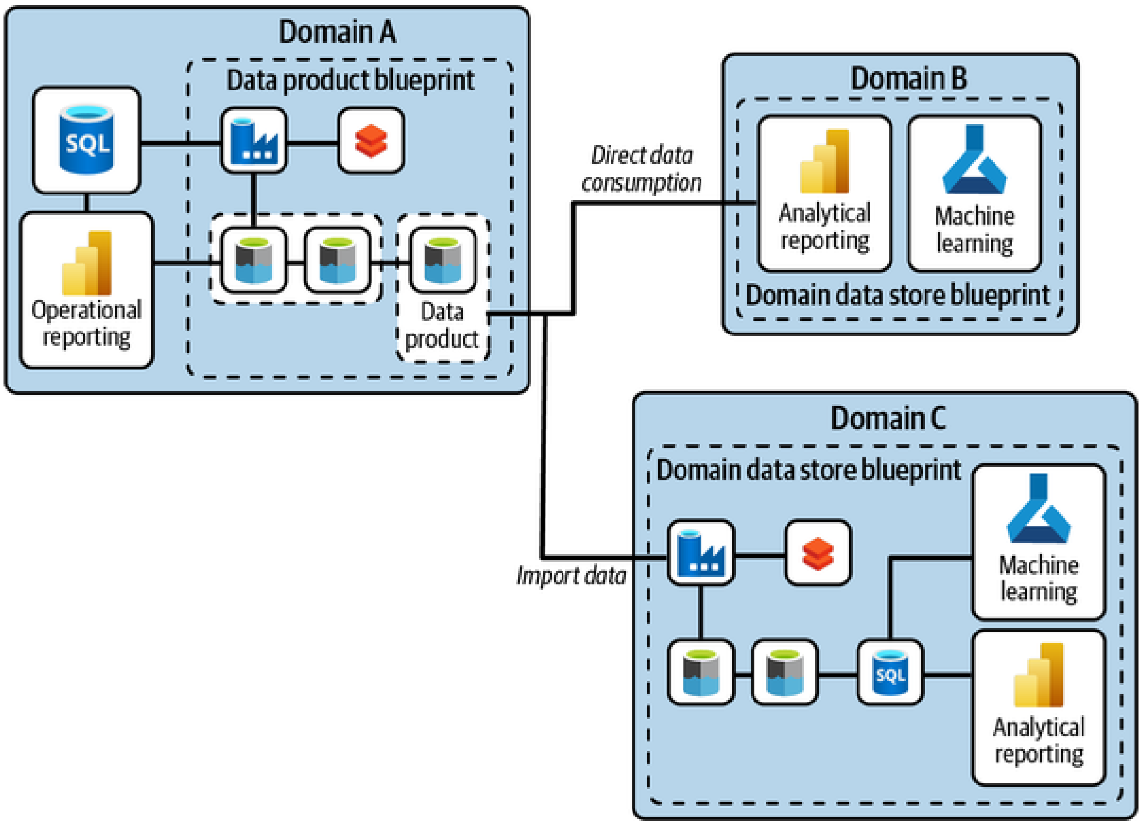

Figure 11-2. Domain data stores

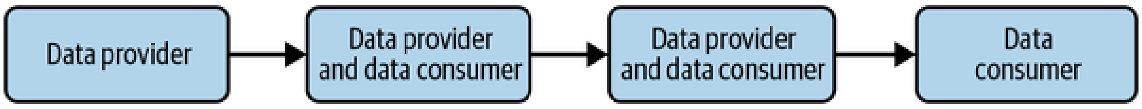
Data provider
Data provider and data consumer
Data provider and data consumer
Data consumer

Figure 11-3. A consumer can become a provider, which can result in a chain of data consumption and further distribution

It is essential to classify applications as either a golden source (see "Golden Sources") or a DDS, because doing so makes accountability for and the origins of unique and authentic data explicit. It also guarantees that data remains unique and unchanged throughout the entire chain of distribution. Note that an application can sometimes be both, depending on the role it is playing. For example, you can classify an application as a DDS when it consumes data from other domains, but if that same application generates new data, then it will play the role of a golden source for the newly created data.

When consuming, combining, integrating, and distributing data, you need to make certain decisions about the size and scope of your DDS. We'll explore these aspects in the next section.

Granularity of Consumer-Aligned Use Cases

To avoid having too many or too few DDSs, you need to ensure that demarcation lines for DDSs are clearly set and each use case or set of use cases consumes and integrates only the data it requires. Determining the appropriate scope, size, and placement of logical DDS boundaries is difficult and causes challenges when distributing data between domains. Typically, the bounded contexts are subject-oriented and aligned with both business capabilities and value streams. Preferably, the boundaries of a domain and DDS would be aligned, but this is not always the case: a very large DDS might be used for several use cases (i.e., domains), or a large domain might encompass multiple DDSs for different use cases of that same domain.

Thus, when defining the logical boundaries of a domain, there could be value in grouping subdomains into a larger domain, and then decomposing the DDS design to simplify data modeling activities and internal data distribution within a larger context. For example, if multiple domains within the boundaries of a value stream work closely together, it may help to pool them together logically within the boundaries of a DDS architecture. The same principle might apply for use cases that require data to be consistent between themselves. Decomposing domains when possible is equally important, especially when the domain is large or when subdomains require generic—repeatable—integration logic. In such situations, it could help to create a generic subdomain that provides integration logic in a way that allows other subdomains to standardize and benefit from it.[4] A ground rule is to keep the shared model between subdomains small and always aligned on the ubiquitous language.

When designing analytical architectures, the important task is to think carefully about the logical role of your DDSs. This involves consideration of both business and technical granularity:

- Start with a top-down decomposition of the business concerns: an analysis of the highest-level functional context, scope (that is, boundary context), and activities. These must be divided into smaller functional areas, use cases, and business objectives. This exercise requires good business knowledge and expertise on how to efficiently divide business processes, domains, functions, etc. The best practice is to use your business capabilities as a reference model and look for common terminology (ubiquitous language) and overlapping data requirements, such as shared integration logic.
- Next, consider your technical goals and how to achieve them. These include reusability, flexibility (easy adaptation to frequent functional changes), performance, security, and scalability. The key point is making the right trade-offs. Two business domains might use the same data, but if their technical requirements conflict, it might be better to separate the concerns. For example, if one business task needs to intensively aggregate data and another only quickly selects individual records, it may be preferable to separate them, even if they use the same data. Similarly, if one use case requires updating the data structures daily while the other expects them to remain stable for at least a quarter, you should consider separating them.

The story doesn't end there, however. For example, as mentioned previously, you'll also need to think about data reusability concerns. The same integrated data might be needed for multiple use cases. Organizing data internally within a business boundary can become more complex when a domain is larger and composed of several (overlapping) subdomains. The DDS in this situation can become a conglomerate of different domain zones: some zones are shared between multiple subdomains, while other zones are exclusively mapped to one domain. Let me try to make this concrete with an example using the medallion architecture. For a large domain, you could plot a boundary around the various zones of one DDS. Within this DDS, the first two zones (Bronze and Silver) can be shared between multiple subdomains, so tasks such as cleansing, correcting, and building up historical data are performed in common for all subdomains. For the transformations in the third zone (Gold), the story becomes more complex because data is required to be specific for each subdomain or use case. So, there will be pipelines that are shared and pipelines that are specific to one use case. This entire chain of data, including all of the pipelines, belongs together and thus can be seen as one giant DDS implementation.[5]

Note

The idea of building a common integration model and then introducing specific selections and designs for consumers isn't new. Layering is a core concept of data warehousing. The difference with DDSs, when applying layering for a larger domain, is the clear scope. In DDSs, the boundaries are clearly set: rather than integrating all data, you only select and consume data for a specific business purpose.

Properly guiding the consuming side is one of the key requirements for successfully implementing a federated architecture. If no architecture guidance is provided to your domain teams, you risk seeing overlapping and repeated activities across multiple domains, with different solutions to the same integration challenges that occur when combining data. Therefore, I recommend managing the consuming side jointly with your data product and master data management activities.

DDSs Versus Data Products

The goal of data products is providing data. DDSs have the opposite goal: consuming data and turning it into value. They're about making business decisions, supported by functions such as reporting and machine learning services. The data that is consumed by DDSs also differs from the data that is typically managed by data product architectures. Data in DDSs has been structured and optimized for the use case at hand. Each business problem is unique and may involve different target user groups, different business requirements, and different nonfunctional requirements.

DDSs may have some overlap with data products when it comes to distributing newly created data, however. We've already discussed that DDSs can play the role of data consumer, provider, or sometimes both. That is, DDSs may share newly created insights and data with other applications. To facilitate this, you need to carefully consider how to align the data providing and consuming sides of your architecture. As you can see in Figure 11-4, there are two design patterns you can choose from:

- Domain A uses one architecture to solely consume and use data and another for data product development.
- Domain B uses a combined architecture that supports both data value creation and data product development.

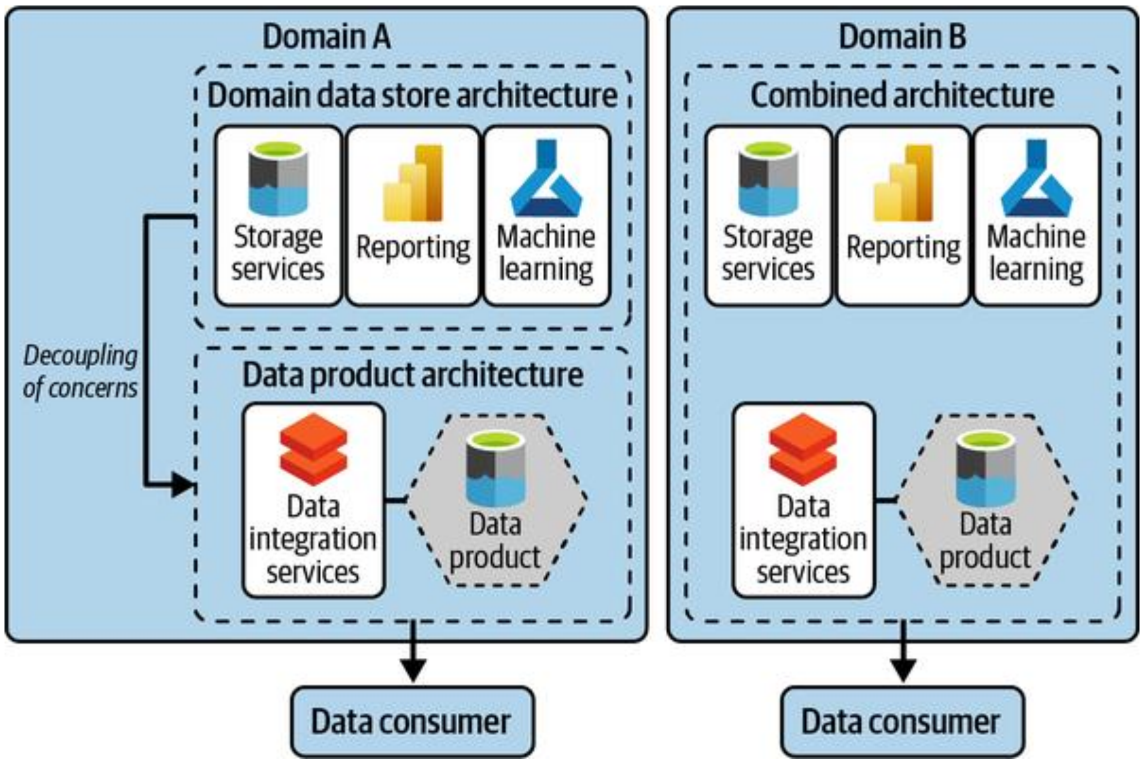

Figure 11-4. DDSs and data products can be managed together or independently

If DDSs are designed to solely consume and use data for analytical use cases, a separate architecture must be used for sharing newly created data (i.e., when building new data products). The DDS architecture, in this scenario, requires another blueprint with a new set of services. The motivation for managing your DDS with two separate blueprints is to decouple the concerns of using and sharing data. With this option, the inner architecture of the analytical use case is isolated from the architecture in which data products are being created and shared.

If DDSs are designed to include data product architectures, the same architecture can be used for consuming and sharing newly created data directly with other consumers. If you design your blueprints like this, there will be synergy in terms of the underlying services that are used for data consumption and sharing. For example, you can use the same ETL framework or share compute resources for transforming data for the two objectives of consuming and sharing newly created data. On the other hand, it will be harder to recognize data copies and newly created data because both are managed within the same architecture. So, there's a coupling risk that needs to be mitigated, for example, by using a metamodel (discussed in Chapter 9).

Your blueprint designs will thus be heavily influenced by the choice you make about whether DDSs and data products are managed together or independently from one another. In addition, your blueprints may differ based on what you need to offer for the activities involved in turning data into value. Analytical use cases are unique and may require different services. For example, a use case that only utilizes reporting services requires a different blueprint than a use case that utilizes machine learning capabilities.

For the data that is managed inside a DDS itself, I recommend distinguishing between data product data and integrated data with a few basic principles. Integrated data, as you've learned, is data that has been consumed, combined, and transformed into a new context. The input and original data originated elsewhere, so lineage is a bigger concern because it shows all the dependencies with other domains. Data ownership works differently, too, because the accountability for data creation and quality lies outside the domain of integration. Thus, distributing integrated data requires approval from the owner(s) of the original data. The same applies for copied data: the origins are elsewhere, so you'll need to track its lineage. You will also need to manage overlapping integration concerns with care: introduce master data management, or consider using any of the aggregation patterns discussed in Chapter 10.

Note

The principles of scoping and decoupling apply to BI and analytical services too. If, for example, one domain wants to expose its analytical models to another domain, they must be decoupled using proper integration patterns. The same coupling concerns apply to newly created data. Distributing integrated data creates new dependencies, so consider breaking down the data into small, highly focused pieces that each solve only one specific problem. Limit the shared domain model to a bare minimum. Don't create a distributed monolith that all teams rely on; require the scope to be set clearly. If data is subject to intensive reuse, it has to become part of the master data management discipline, as discussed in Chapter 10.

Best Practices

The biggest problem with delivering value on an enterprise scale is that use cases are highly diverse and scattered throughout the organization. They all try to address various aspects of solving a business problem, targeting different audiences. Use case scenarios are wide-ranging, and a large variety of data stores and services are needed to facilitate them. For example, one use case may require data that arrives at a fast pace for stream analytics, while another use case may require slowly arriving historical data for reporting. Others may focus on process mining, image recognition, or artificial intelligence. Some use cases are descriptive or diagnostic and only look back to the past, while others are predictive or prescriptive and peek into the future. As the demands of data consumers continue to diversify, it's impossible to enforce a common way of working and a generic solution design for all use cases. However, there are some best practices that can always be followed with regard to business requirements, the target audience and operating model, nonfunctional requirements such as databases and tools, and data pipelines and data models. We'll examine each in this section.

Business Requirements

Your starting point should always be your business requirements. Your use cases will mostly depend on the concerns and the opportunities that business teams see. There could be, for example, projects that add new revenue to the organization by monetizing insights, increasing customer satisfaction, or gaining competitive advantage by getting insights into market trends. Other use cases, for example, might be centered around risk, finance, cost reduction, sentiment analysis, cohort and tracking analysis, and increased operational efficiency with process mining and better insights. Some other use cases just want more accurate data to make better and more predictable business decisions. For all of your business use cases, the recommended approach is to make a long-term business plan. So, look at the long-term value-add. Don't end up in a situation where you're solving ad hoc problems with one-off solutions. Carefully consider the need for any new services;

they might be expensive and hard to maintain, control, secure, etc. If a new service is truly needed, enable it and evaluate the usage before enabling any additional services. In all cases, pay attention to overlapping services.

Implementing use cases is a challenging task in terms of analysis and required resources. To prepare, have brainstorming meetings with your business teams, then organize and cluster all of the output. Creating a business model canvas can be a good way to highlight what business use cases potentially unlock what value using what tools and services. You might want to extend this information with the required stakeholders, needed data sources, connections to any existing use cases, actors and interactions involved, and timelines and success criteria for implementation.

Your business requirements will also determine the amount of data you need to obtain. For example, if you want to develop a machine learning algorithm based on data with many fluctuations and high variability, you'll probably need much more data than you would for a simple model with low variability.[6]

After you've analyzed your business requirements, you'll need to ensure that your objectives and goals are well defined, detailed, and complete. Understanding them is the foundation for your solution and requires you to clarify what business problems need to be solved, what data sources are required, what solutions need to be operational, what data processing must be performed (in real time or offline), what the integrity and criticality requirements are, and whether the outcome is subject to reuse by other domains.

Target Audience and Operating Model

An important next step for working out your analytical use case is to determine who you need and what their responsibilities should be. There are many roles related to turning data into value. These can be defined in many ways, and also can be easily misunderstood. Therefore, it's a general best practice to make these roles part of your data governance framework (see "The Governance Framework"). Here are some suggestions for how you might define the different personas:

Domain owner

Higher-level decision maker, typically a product owner or business representative, who collaborates with business users on requirements and governance teams on data management policies, processes, and requirements, as described in Chapter 8.

Application owner

Responsible for the application or technical solution design, including all development, security, and application and data maintenance activities.

Analysts or subject matter experts (SMEs)

Responsible for defining business requirements, what data is used for, who may access it, and how the data is shared with others. Collaborate with the domain owner and other data owners as needed and support the business in its use of data. Data analysts or SMEs are typically referred to as "power users" because they know how to analyze data with tools but aren't necessarily data scientists or business intelligence experts.

Data engineers

Responsible for collecting, integrating, and enriching data. Data engineers have an important role, enabling the data analysts and data scientists to do their work properly. They maintain data pipelines and often work with languages like SQL and Python.

Data scientists

Statisticians, usually with a background in computer science or mathematics. Data scientists are experienced with algorithms, machine learning, and deep learning. They are usually familiar with the business and follow market trends. Their end goal is to find patterns and correlations and make predictions based on the data. They often work with languages like Python, Scala, R, and Java.

Report builder

Responsible for designing and building reports and dashboards for other users. A report builder typically has superior business intelligence skills.

Depending on the size and complexity of the business problem, some or all of these roles will be working together in a cross-functional (domain) team. The optimal size of such a team is often considered to be the size of a DevOps team: between 5 and 10 people. Creating small, independent teams doesn't mean that they can't share talent, but it should be clearly defined where responsibilities lie. Having small teams focusing on clearly defined and scoped targets enables scaling, as many teams can advance different aspects of the business simultaneously. All team members must have a good understanding of the business challenge, corresponding solution, and required data.

Nonfunctional Requirements

Nonfunctional requirements can make or break the success of a software system or product. These include cost, scalability and performance, latency, ease of maintenance, consistency, security and governance, write and read characteristics, data volume, variety, and velocity, and so on. Each will guide you in a certain direction.

The central question is often what type of database technology or data store should be used. Selecting the optimal data store depends on many criteria. There's no silver bullet. It's sometimes also a matter of taste and experience. The following questions are worth considering:

- How is your data structured? Structure influences preprocessing steps, type of cleansing performed, modeling steps, type of storage, and more. Failing to understand this from the beginning will result in wasted time and an incorrect outcome.
- Are your queries predictable? This can make a significant difference to your implementation. If the queries are predicted, you can optimize with caches, indexes, or preoptimized data, for example.
- What types of queries are you using? Simple lookups, aggregations, joins, mathematical operations, text search, geographic search, other complex operations? Relational databases, for example, are usually better at joins.
- What are the requirements for current, historical, and archived data? Does all data need to be retained?

- How do you want to balance between optimizing for integrity versus performance? For example, must your application design enforce strong consistency, or is eventual consistency good enough? If integrity is important, RDBMSs typically enforce consistency better.
- What trade-offs can you make with regard to prioritizing reads, ingestion, or integrity? Different data models have different characteristics. A Data Vault, for example, is flexible in its design and can handle parallel loads; however, reading the data is more performance-intensive.
- What kinds of operations will your database be doing? Certain types of operations are likely to be more dominant. Distributed filesystems are well suited for appends but not for (random access) inserts. Relational databases, on the other hand, excel at these.
- What volumes of data will be coming in and going out, and at what velocities? Some NoSQL databases perform badly with batch data ingestion. For high-speed data ingestion (messaging and streaming), normalized data models and ACID consistency models may decrease performance because of integrity and locking controls.
- What data access protocols do you need? Some databases are only accessible via SQL, ODBC/JDBC, or native drivers, while others only allow access via RESTful APIs.
- How much flexibility will you need to adapt or change the data structure? NoSQL databases can handle changing data structures well since they can be schemaless.
- How much scalability and elasticity will you require? Some systems support dynamic horizontal scaling.
- What are the database dialects and languages that are supported? Are your engineers already familiar with these? How difficult are they to learn?
- Does your vendor require specific storage solutions? The vendor's requirements might limit your choices.

Other considerations include costs, open source standards, features, integration with other components, security (such as access control), privacy, data governance, and ease of development and maintenance.

Consider how many and what types of data stores you want to offer to your organization. You might want to define a common set of reusable database technologies and patterns, as part of your blueprints, to ensure you best use the strengths of each data store. For example, mission-critical and transitional applications might only be allowed to go with strong consistency models, or business intelligence and reporting might only be allowed with stores that provide fast SQL access. Strive to offer a comprehensive list of choices, but limit the overlap! You don't want to end up with dozens of different database technologies.

Additionally, align your database choices with other tools and services. For example, some ETL frameworks work better with some types of databases than others. Also take these considerations into account when developing blueprints. Finally, you might want to vary between different cloud environments. You'll probably end up with a list of several technology services that will facilitate most use cases. Consider developing a flowchart or decision tree for supporting your teams in making the right decisions.

Data Pipelines and Data Models

Another important step in turning data into value is engineering a data pipeline that selects data and brings it all together in the target solution. The key point here is that all the data is already immutably persisted in data product stores, so there's not always the need to make an extra copy. This means you either obtain data directly on your regular schedule or wait for the data provider to tell you when you can start processing. When data is pulled over, a data transformation is typically required because you're moving data from one context into another. For this context transformation, you need ETL tooling (which we'll discuss in a moment).

For real-time ingestion, the pipeline works differently because you need to include a way to capture and store real-time messages. You can choose to store incoming messages in a folder or database for further processing. Another option is to use a buffer or extra application component, or use the integration capabilities of the streaming platform, which allow you to analyze, manipulate, and distribute messages. For more on this topic, revisit Chapter 6.

Note that many variations are possible, depending on your use case requirements. For example, for later analysis it's better to retain data by storing it as it's ingested, before transforming it or applying target semantics. For faster processing, you might combine ETL batch and stream processing. For handling different users' needs, you can also use a polyglot design that incorporates different data storage technologies. Again, look at your use case requirements, and make conscious choices.

Building a data pipeline is the most complex part of the entire project. Start small and keep it flexible. Most project failures happen when engineers try to "boil the ocean" and model all of the data at once. Another common mistake is immediately jumping into the technology and starting to write code before thinking through your requirements. Before you start building, carefully consider all of the functional and nonfunctional requirements, and address the fundamental questions. Are there reusable tasks? What is the best order for processing all the data? Are there dependencies, or can some processing steps be completed at the same time? How will the data be queried?

Note

Data scientists like to refer to what they call the "80/20 rule": 80% of a data scientist's valuable time is spent on finding, cleansing, and organizing data, leaving only 20% for actual development. There's no single algorithm that uses raw data and gives you the best model.

I recommend assembling your data pipelines as a series of isolated, immutable transformations that can be reused and combined easily. Input, transformation logic, and output in this case must be clearly separated from each other. For reproducible output data, I recommend versioning all pipelines. For better performance that takes advantage of the elasticity of modern infrastructure, you might want to engineer your pipelines to run in parallel. When it comes to complex data transformation and processing, it’s sometimes a better option to use in-memory distributed processing engines like Apache Spark. Another consideration is to write your transformation logic in templates or a high-level programming language within a development-friendly ecosystem. This approach can be supported with best practices for writing and sharing good code.

SQL Versus NoSQL Pipelines

SQL pipelines, in general, handle and execute complex queries much better, but also require a better understanding of the relationships between tables. A SQL pipeline, therefore, might require more validation steps and transformation and update logic, because data must be parsed into the right (restrictive) structure.

NoSQL pipelines, on the other hand, can be simpler and more flexible. They can create data immediately and dynamically without defining structures. They’re also more easily scalable, and queries often run faster since they don’t involve joining many tables. Data can be more easily replicated horizontally, but querying and integrating it can be more difficult.

For data quality you can rely in part on the validations in the data products, but this doesn’t mean there are no extra validations required in the pipelines. Consumers’ viewpoints on data quality vary, so pipelines need to be extended to address common ETL issues, which typically involve constraints, completeness, correctness, and cleanliness. Additionally, security and privacy concerns must be handled with care. So, extra steps are expected before importing and processing your data.

Additionally, I recommend categorizing all your ETL tools and frameworks in a unified portfolio, coupled with best practices and considerations. Provide flexibility, but standardize by limiting the options. For example, GUI- or template-based frameworks can be best used when data movements and transformations are somewhat straightforward. Programming-oriented languages, on the other hand, may be best suited for situations that require complex transformations or machine learning.

Finally, the entire pipeline must be set up to provide metadata for lineage, so it’s possible to trace and understand what data is mutated in what way at each specific step in the pipeline. This requires transparent back-pointers to, for example, your data products. File and event names, business keys, source system identifiers, and the like must be made available in central tools to ensure the correctness and completeness of all transformation steps. You might want to use a data catalog to stitch all of the integration architectures together. I also recommend formulating strict principles when teams build data pipelines, for example, determining under what conditions the lineage must be delivered centrally. These lineage requirements may also put constraints on the ETL tools you offer, because not all ETL tools perfectly integrate with all lineage solutions.

Scoping the Role Your DDSs Play

Persisting data for downstream consumption can happen for various reasons. Let me first provide some examples, and then I’ll share some best practices for DDSs:

Analytical data stores

Analytical data stores, or file stores, should be used for integrating and combining data for analytical workloads, such as machine learning. Their purpose is to provide high-quality data for accurate model development. Data in analytical data stores is typically flattened and denormalized.

Reporting stores

Reporting stores are used for business intelligence or dimensional reporting. Data is typically modeled into facts (measurements) and dimensions for another context. The most popular and easiest design choice is a star or 3NF schema.

Aggregate stores

As discussed in Chapter 10, when domains have overlapping data requirements it can be preferable to cluster some of the business logic and create a new, aggregate data product for the domains to use. Aggregate data stores are also a good design option when a domain can be decomposed into several subdomains and the data requirements overlap heavily.

Third-party or SaaS solutions

Some of your use cases might sit outside the logical boundaries of your architecture. For these situations, you will need connectors and a store for extracting and prepping the data before serving it to the external consumers

Operational consumers

Operational consumers are like OLTP systems, with a subtle difference: they don't just create data, but also consume and integrate data from other domains.

Data warehouses or data lakes

Consumers might require data warehouses or data lake services when they need to combine and integrate larger amounts of (historical) data from other domains into a common format that is easier to work with.

For all your use cases, standardize on what tools and services you would like to offer to your different domain teams. Consider aligning your blueprints for supporting different scenarios. For example, an aggregate store that only distributes newly created data doesn't require any business intelligence or machine learning services. Such a scenario can be best facilitated with a simpler blueprint that only contains some lightweight integration services. Scoping data stores and aligning on what blueprints your teams can choose from helps to reduce the overall complexity. For a federated approach, aim to always use a generic blueprint as a starting point. That is, each time a domain wants to consume data or build a use case, choose a generic blueprint, adjust it (as minimally as possible) to suit the domain's requirements, and deploy it.

For your blueprints, also consider aligning with your data product architectures, because the overlap is significant. For example, you might want to standardize on the services you choose from the modern data stack or the layering applied when using data lake services. Additionally, standardize on the software development aspects, including automation of code deployment, orchestration, and what frameworks, libraries, and tools users can deploy. Finally, you'll want to standardize on different aspects of data modeling and design, such as grains,[7] naming standards, materialization, permissions, data normalization techniques, and so on. A typical approach for organizing all this is to use inner sourcing or set up a central center of expertise.[8]

Note

Have you noticed how many times I've used the word *standardize*? A transition toward a federated architecture starts with setting standards.

Different data stores manage and organize their data internally in different ways. One common strategy is to separate (either logically or physically) the concerns of ingesting, cleansing, curating, harmonizing, serving, and so on. Within modern data stores or data lake services, this could involve various zones using different storage techniques, such as folders, buckets, databases, and the like. Zones also allow you to segregate concerns or purposes, so one store can be used for use case consumption and another can be used for data product maintenance and distribution. For all stores and zones, the scope must be very clear. They shouldn't span across multiple bounded contexts. Each domain operates within its own boundary, within which it will consume, integrate, and create new data. Again, consider making the zoning part of your blueprint designs.

Inmon Versus Kimball Versus Data Vault

What data modeling technique should you use on the consuming side when designing data warehouses or data lakes? Well, it very much depends on the characteristics and requirements of your solution. For cloud-based data warehouses, I generally prefer wide, nested, denormalized tables because such a design better utilizes the infrastructure provided by the cloud vendors. Furthermore, with such tables you avoid expensive computational joins and shuffling of data between compute nodes,[9] which can make your queries run slowly. On the flip side, if you have complex integration challenges, then a design such as a Data Vault could be more appropriate. For example, if many of your sources use the same business concepts, which are related and need to be integrated together, then you can separate concerns more easily with hubs, links, and satellites. In addition, a Data Vault has the benefit of decreasing dependencies between tables during the load process and simplifying the ingestion process by applying only inserts, which load faster than updates or merges. Bottom line: each domain should select a data modeling methodology that best matches its use case requirements.[10]

Although all data-driven decision solutions are expected to be designed specifically for each use case, you may notice that BI tools and ML services are common denominators. In the next sections, we'll take a closer look at each of these.

Business Intelligence

Business intelligence tools help businesses take a more structured look at data while providing deep interpretations. They allow for decision making via interactive access to and analysis of data. For BI tools such as reporting and dashboarding services, there are some extra considerations to weigh to ensure you're delivering the right value to the business.

Semantic Layers

The most fundamental question is whether you want to abstract your DDS with a semantic layer to present the available data to users in an understandable way for easy and consistent use. *Semantic layers* are business representations of data that help end users access data autonomously using common business terms. They are typically part of a reporting service and built and maintained by BI developers and business users. Using a semantic layer for business intelligence and other forms of data consumption has many benefits:

- Semantic layers make it easy for users to query the data and make the right combinations. Typically users don't have to worry about making the joins. The relationships are predefined and help everyone make the right combinations, avoiding incorrect calculations or interpretations. Fields are consistently named, formatted, and configured. For example, an amount can be automatically summed, or a date field can be used for time-oriented calculations.
- Data in semantic layers is optimized (preintegrated, aggregated, and cached) for consumption to improve overall performance. This results in a much better and quicker user experience.
- The underlying data store is decoupled, which means that changes to its data structure won't necessarily affect all reports and dashboards immediately.
- Semantic layers can have extra features, such as versioning, row-level security, and monitoring.
- A semantic layer can be built using a technology called *data virtualization*. This hides the technical implementation details from the consumers.

Semantic layers can be used to facilitate different kinds of data integration scenarios. They can be directly used by domains for data models that already align to specific analytical use cases. Domains can also use them as input for other semantic models: so one domain uses a semantic layer from another domain as input for its own semantic layer. In this case, the semantic models often need some tweaking, or are combined with data from another domain. To conclude: a semantic layer can be used to generate curated data, which means the semantic context of the data could change. This can mean that the ownership of data changes as well.

Most BI tools allow you to build semantic layers in two different ways. The first option is to use the proprietary in-memory or caching model. With this model, data from the underlying data store is transferred to the caching layer (refreshed) on a schedule. This often results in the data being duplicated; there's extra data latency because the underlying data store can be more up-to-date than the caching layer.

The second option is to utilize the underlying data store. In this model, queries are passed on to the underlying data source, so the semantic layer is only a metadata layer that directly federates the queries. This option is better for real-time and operational reporting because the database is updated within seconds, meaning there's less latency. The drawback is that data must be modeled and optimized for the reporting structure, typically a dimensional data model.

Semantic layers do have drawbacks and must be used with caution. For simple reports and predictable queries, it isn't essential to add the overhead of an extra layer. Another problem is that advanced tools can easily pick up and hold lots of data, which eventually makes the semantic layer costly. Another complicating factor is managing the overlap between reports and underlying (shared) data models that may combine the same data from different sources. As your landscape grows, so does the complexity of your reports, and the number of relationships to shared datasets grows. Set clear boundaries and align them with your capabilities. Only use shared (or composite) models if they are stable and truly matter.

The challenge of building semantic layers and reporting solutions is that you need to be clear about what insights you want to extract from the data. You need to know your business questions, organizational goals, and data sources up front. In many cases you'll first want to evaluate, explore, and discover what actionable patterns can be extracted from the data. This is why self-service data discovery and data preparation tools are so popular.

Self-Service Tools and Data

Self-service capabilities such as data discovery, preparation, and visualization tools aim to accelerate the process of deriving value from data by providing easy-to-use, intuitive self-service functions for exploring, combining, cleaning, transforming, and visualizing data. The rationale behind data preparation is that the data becomes more user-friendly and can be more easily used for reporting and analytics—i.e., it's more suitable for further processing and analysis. Data analysts and scientists can perform these activities themselves without requiring any programming or complex IT skills.[11] Some BI and data analysis tools come with machine learning algorithms to recommend or even automate and accelerate data preparation.

Note

Some people use the terms *data discovery* and *data exploration* interchangeably, but they aren't identical. *Data discovery* is the broader term used to describe the iterative process of searching for data in a data catalog, and then finding patterns and trends in data. It's usually the first step before performing a data analysis. *Data exploration* is about getting a further-reaching, deeper understanding of what's inside the data and what its characteristics are.

Empowering data professionals to turn data into value themselves influences your architecture design and blueprints because data management and end-user tools need to be integrated and accessible. Data professionals are segmented based on their skills and required tools. This segmentation and focus on self-service affects standardization. If dozens of tools are available, it will be hard to integrate and apply automation. Successfully implementing self-service capabilities thus requires trimming down the number of business intelligence and analytical tools with overlapping features and aiming for standardized solutions that meet the enterprise's specific needs. It also requires you to clearly distinguish between managed and self-service data:

- Managed data is for ongoing business needs, and thus by nature stable, automated, and standardized. It must meet all your data governance criteria.
- Self-service data is for ad hoc and one-off analyses and thus temporary by nature.

Self-service processes typically start with trying to understand what is in the data, what possible relationships there are, and how the data may be used for different business purposes. By manually analyzing, combining, and adjusting the data, you can validate a general hypothesis. These activities can also include experimenting with the data. Once the outcomes of your discovery and exploration activities are clear, it's time to operationalize the value, which means building a properly managed data pipeline (model and ETL) in a managed (production) environment with real data. Self-service can also be used to help business users looking to answer a single, specific business question quickly. This is known as an ad hoc analysis.

Between self-service activities and managed production workloads there must always be a proper handover from business users to data engineers. Self-service is often achieved through the creation of a semantic layer or a project that is run by only business users. This means IT is removed as the middleman; it allows business users to drive their own analysis. In that respect, IT's role should be focused on delivering the self-service capabilities as a platform. The data in this environment could be temporary and should be allowed to leave only under strict conditions. Self-service activities can also be managed with policies. I see a lot of organizations that provision lightweight, trimmed-down discovery environments, within which end users cannot orchestrate data pipelines or use Spark.

Tip

To prevent users from copying data and storing results locally, provision self-service capabilities into a (temporary) workspace layer (in addition to Bronze, Silver, and Gold) for the purpose of self-service discovery. Some organizations even allow data in workspace layers to be discovered by other domains; however, by virtue of being in a workspace layer, it is marked as noncertified. This approach enables power users in the organization to quickly discover and share insights without having to go through all the validations, formal approval, documentation, etc.

Managed data, unlike self-service data, can be offloaded directly into self-service environments. The opposite (moving self-service data into managed environments) is forbidden, because managed data is never allowed to depend on human intervention. The transition between self-service and IT-managed environments should be smooth. Users who maintain IT should work closely with those using the self-service exploration environment. An engineer, for example, might examine the self-service effort and even help build some sample ETL scripts, which are first tested in the self-service environment and eventually find their way to the managed environment.

Business intelligence, which includes managed and self-service data, powers decision making; however, it's important to find the right balance for your organization's needs. There's no one-size-fits-all solution, so consider following some generic best practices.

Best Practices

To ensure you get the maximum value out of your business intelligence investments, consider the following key actions:

- Make purposeful decisions about your BI strategy. Decide on the ideal balance of self-service BI, managed self-service BI, and enterprise BI.[12] Align the ownership and management of these models with your domains.
- Develop playbooks and design patterns for helping your users. For example, what is the best approach for loading data into BI tools? Should reports directly load data into a reporting environment, or should a database be created first, and all reports connect to that? As there are no fixed answers and many design decisions to make, consider writing playbooks: sets of rules or suggestions that are considered to be most suitable given a particular situation.
- Consider setting up a center of excellence (COE): a central team that's responsible for defining company-wide standard processes, training, guidelines, best practices, support, and much more. Talk to stakeholders in different business domains to understand which practices are currently working well and which practices aren't working well for data-driven decision making.
- Use different maturity levels for guiding the conditions in which business intelligence can be used. Level 1, for example, focuses on use cases that are new, undocumented, and without any process discipline. For this level, there can be few formal processes in place. Level 2 focuses on use cases that are repeatable or managed. A governance model must be in place and all users should be certified. Level 3 is for critical use cases that deliver much business value. On this level, automation and monitoring must be in place.
- Develop a champions network of recognized experts that continually build and share their knowledge with other people within the organization.
- Require domains to provide a sample report of the data that they're providing. Not only does this mean the datasets and connections to use cases are made available within the BI layer so consumers can discover and request access to it, but by seeing a dashboard, users can quickly get an idea of what the data means, how it's joined, etc.

Business intelligence is a way to create a holistic view of all your relevant business data. At the same time, business intelligence is often about looking backward, attempting to figure out why something has happened. It may involve analytics, but for really predicting what will happen next and making automated decisions, we need to look at advanced analytics.

Advanced Analytics (MLOps)

Machine learning, *artificial intelligence*, and *cognitive computing* are trending buzzwords. They all overlap and complement each other. Artificial intelligence is the umbrella term for when machines work "intelligently" and perform tasks that are normally done by humans. Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to learn to mimic intelligent behavior. For example, a machine learning model can predict the probability of you clicking on a link or buying something based on past behavior. Cognitive computing is another subfield of artificial intelligence; it places more emphasis on how the human brain works and can be used, for example, to translate speech to text or recognize and classify items in images.

While these services are getting easier to use for training and developing accurate models, deploying them into production—especially at scale—is a major challenge.[13] The reasons for this include:

- Models strongly depend on data pipelines. Using offline data from the data preparation environment is easy, but in production everything must be automated: data quality must be guaranteed, models must be automatically retrained and deployed, and human approvals must be required to validate accuracy after using fresh data.
- Many models are built in an isolated data science sandbox environment, without scalability in mind. Different frameworks, languages, libraries pulled from the internet, and custom code are often all mixed and combined. With an organizational structure with different teams and no proper handover, it's difficult to integrate everything in production.
- Managing models in production is a different ball game because in production everything needs to be continuously monitored, evaluated, and audited. When making real-time decisions, you need to guarantee the scoring efficiency, accuracy, and precision. You don't want to end up in a situation where two days later you find out that all your customers got a free promotion.

To overcome the challenges of managing analytical services, new practices for collaboration and communication have emerged.

Machine learning is difficult to manage at scale. The ML life cycle consists of many complex tasks, including data collection, data preparation, model training, model deployment, model monitoring, and explainability. It also requires collaboration between various user groups: business users, data engineers, application developers, machine learning experts, and platform engineers. A framework for a way of working, often called *MLOps* (a compound of "machine learning" and "operations"), can help organizations to streamline the process of deploying machine learning models to production and then maintaining and monitoring them.

MLOps tackles the collaboration and communication between data scientists and operations professionals to help manage the production ML life cycle. It overlaps with DevOps, although deploying software in production is fundamentally different from deploying analytical models into production. This is because software, in general, is static, whereas ML data is constantly refreshed. This means analytical models must be constantly retrained, recalibrated, and redeployed using the latest data.

A good MLOps process can be perfectly aligned with a federated way of working and thus nicely positioned within a data mesh architecture. It enables you to operate at scale by using a standard process for managing the activities end-to-end. Such a framework is also helpful because each phase within the process may involve several complex stages. To help you manage ML end-to-end, I'm going to lay out an MLOps reference process, which I'll discuss more intensively over the following sections. Figure 11-5 shows the entire life cycle of a machine learning project. Each phase has different stages, and each stage has multiple steps.

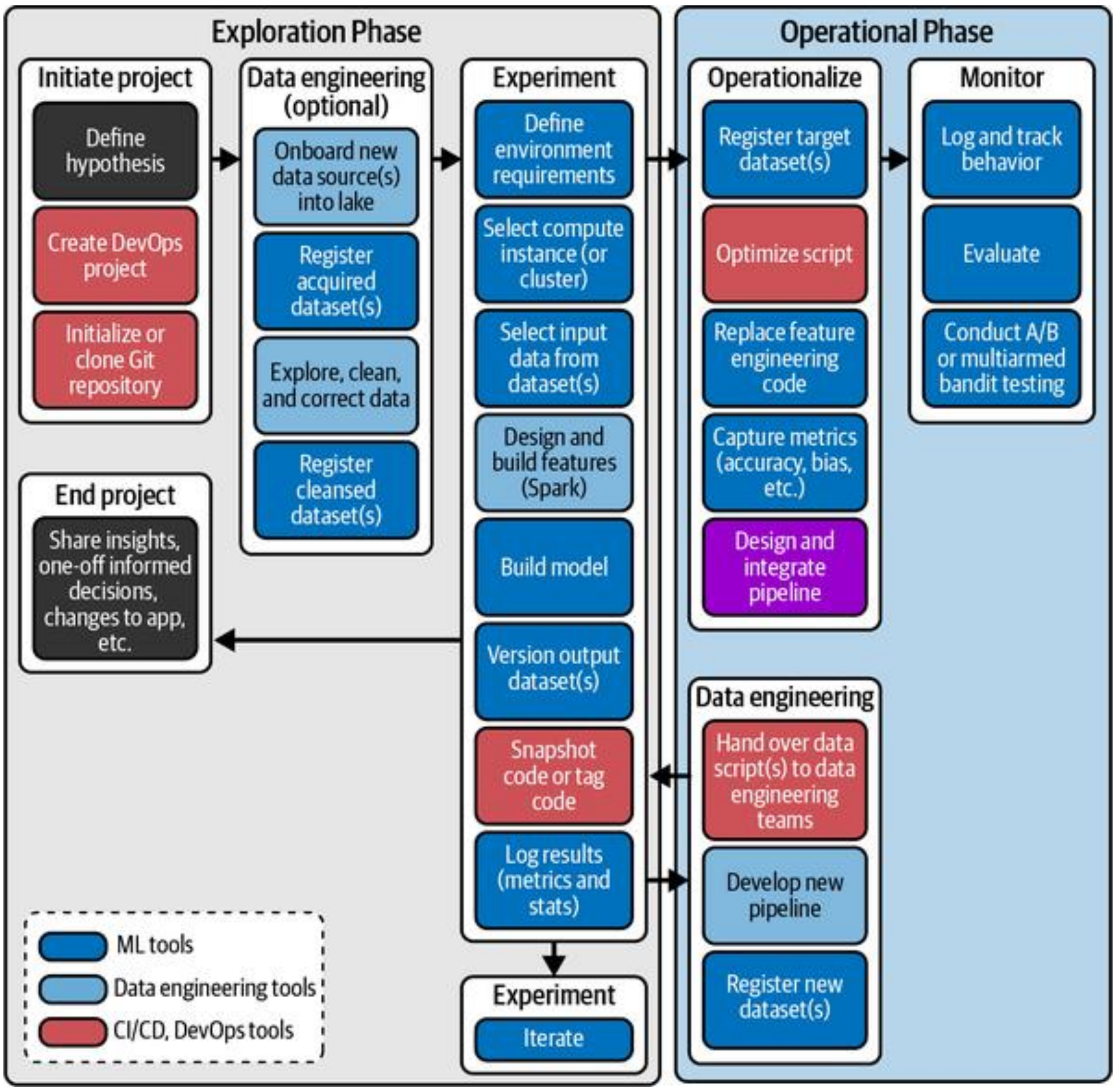
Exploration Phase
Initiate project
Define hypothesis
Create DevOps project
Initialize or clone Git repository
Data engineering (optional)
Onboard new data source(s) into lake
Register acquired dataset(s)
Explore, clean, and correct data
Register cleansed dataset(s)
Experiment
Define environment requirements
Select compute instance (or cluster)
Select input data from dataset(s)
Design and build features (Spark)
Build model
Version output dataset(s)
Snapshot code or tag code
Log results (metrics and stats)
End project
Share insights, one-off informed decisions, changes to app, etc.
Experiment
Iterate
ML tools
Data engineering tools
CI/CD, DevOps tools
Operational Phase
Operationalize
Register target dataset(s)
Optimize script
Replace feature engineering code
Capture metrics (accuracy, bias, etc.)
Design and integrate pipeline
Monitor
Log and track behavior
Evaluate
Conduct A/B or multiarmed bandit testing
Data engineering
Hand over data script(s) to data engineering teams
Develop new pipeline
Register new dataset(s)

Figure 11-5. Recommended MLOps process for managing machine learning projects end-to-end

Starting at the left of Figure 11-5 with the exploration phase, we initiate a project and get clarity on the business requirements. Next, we iterate, testing different algorithms and parameters. This often requires data engineering activities for feature creation.14 After we've demonstrated the project's value, we might enter a new phase: deploying the model into a production system. This phase is sometimes called *productionizing* or *operationalization*.15 This is the most complex part of the process; it involves many activities and requires an automated machine learning pipeline for managing all the steps, from data collection to data validation, model training, model validation, and model deployment.

Depending on your specific use case, your machine learning project might look different. For example, you might train, evaluate, and deploy multiple models in the same pipeline, or your project might be a one-off analysis, where analytical insights are shared with the business but no models are deployed in production. Your starting point may vary too; for example, if data isn't already available, you first must go through the steps of data collection and onboarding.

To help you better understand what MLOps is about, I'm going to discuss each step in detail. While doing so, I'll share best practices and describe the deliverables from each stage. At the end, I'll discuss some variations and exceptions.

Initiating a Project

The first phase of the MLOps project (the "initiate process" step in Figure 11-5) is devoted to understanding your business requirements and data needs, and planning the implementation and development work to be carried out in subsequent stages. It sets the foundation for the later stages. You start by defining your business objectives and formulating your success criteria, which define under what conditions you will move on to the next phase. For collaboration between team members, I recommend using DevOps tooling. For example, you could use a kanban board to manage and track your activities throughout the life cycle of your project. One of these activities might be to identify your data sources and determine whether data products already have been developed.

A good practice when starting a machine learning project is to create a new code repository in which all (future) artifacts will be stored and managed. A general best practice is to segment use cases from one another, so each time you create a new ML project, you should create a new repository. To do this efficiently, consider standardizing on a code template—this will allow for code reuse, and decrease the ramp-up time at project start or when a new team member joins the project.

When you initiate a new project, you typically produce several artifacts:

- A project document or wiki covering the business requirements, success criteria, ethical dilemmas, etc.
- A list of data sources and additional requirements that must be applied
- A team board, which holds the initial activities
- A new code repository using templates to ramp up the project development

These artifacts help your project team to track progress. They also help to share insights for other future projects. Once the business objectives are clear, you enter the next phase: experimentation and tracking.

Experimentation and Tracking

After you've defined your goals and ambitions and set up your project, it's time for experimentation. During this process, I recommend tracking all of your experiment results. Thus, each time you train and run a model, you should capture all the parameters, metrics, algorithms, and other artifacts, as well as the output. This enables you to do the following:

- Gather and organize all the elements needed to conduct the experiment.
- Reproduce any results using saved experiment data.
- Log iterative improvements across time, data, frameworks, models, users, etc.
- Prove to regulators how your models were developed, what algorithms were chosen, and what datasets were used as input.

For experiment tracking, there are many tools to choose from. A popular framework is MLFlow; it's free and open source and is used by many large vendors, including AWS, Databricks, Google, and Microsoft.

When performing experiments, the first step is to determine what compute infrastructure and environment you need.[16] A general best practice is to start fresh, using a clean development environment. Keep track of everything you do in each experiment, versioning and capturing all your inputs and outputs to ensure reproducibility. Pay close attention to all data engineering activities. Some of these may be generic steps and will also apply for other use cases. Finally, you'll need to determine the implementation integration pattern to use for your project in the production environment. This should be one of the following:

Model as an API

In this approach, the model is deployed as a web service so it can be used by other applications and processes. An API call has to be made in order to get predictions from the model.

Model as batches in/out

In this approach, the model works with batches of input and outputs mini-batches or batches of predictions. Chunks or subsets of the data with labels are used for training, predicting, and making recommendations. The input and output typically consists of a set of files (e.g., CSVs or Parquet files).

Model as stream

In this approach, the model interacts reactively with a data stream, where the data segments arrive in increments. If extra data is needed, the model can be allowed to read the DDS's database directly. The model can generate and publish new events as well.

During the experimentation phase of your project, it's common to have manual processes. Your goal is to confirm your hypothesis by validating your assumptions. At the end of this stage, the project might end as a one-off exercise, where the learnings are shared. Alternatively, you might see sustainable business value. In this case, your next step will be to continuously deliver value. You'll do this by automating the entire process of data collection, data validation, model training, model validation, and deployment. You'll learn more about this after we've discussed the data engineering activities.

By the end of the experimentation stage, you'll have several new deliverables. These may include:

- A project document covering the outcome and criteria for model operationalization. This document also contains a model report: for each model that's tried, a standard, template-based report that provides details on the experiment is produced.
- New objects in the analytical workspace:[17] environments, models, datasets, and experiments, including all logging and metrics.
- A feature report: a document that contains pointers to the code used to generate new features. You may want to classify what code is generic and can be used for

other use cases, what features are trained on potentially personal sensitive data, etc.[18]

- An updated code repository or code templates for new projects.
- A list of other improvements that can be made (e.g., adjustments to environments or new compute infrastructure options).

Data Engineering

During experimentation and model development, you may have encountered data quality issues or missing data. It's important to report any such issues back to the source system or data owners, because you don't want to see these problems pop up in other places. Another recommendation is to develop data pipelines that automate the process of using data products to retrain models in production. For this, you may want to implement some controls for data drift detection (decreasing model performance) and schema changes. Once you're done, and your data pipeline is built, you should register your newly created dataset as an official artifact in your machine learning environment. This accelerates model development and takes away the pain of dealing with configuration, such as folder locations and user credentials.

Another part of the data engineering phase is developing a feature store. A *feature store* is a centralized repository where you standardize the definition, storage, and features for model training and deployment.[19] This has a few important benefits:

- It makes the process of creating features much more streamlined and efficient, because developers can discover and reuse features, instead of re-creating the same or similar ones.
- It reduces costs by lowering complexity and improving performance. For example, suppose you have a feature that is computationally expensive and is used in multiple machine learning projects. Rather than using a transform function and storing the transformed feature in multiple training datasets, it's much more efficient and maintainable to store it in a shared repository. The same applies when preprocessing or building features takes a long time, or when features are time-dependent or depend on other features.

New deliverables added to the list of artifacts after you've finished the data engineering phase may include:

- An updated solution design. This is a diagram or description of your architecture, including sources, data pipeline(s), integration patterns, and so on.
- New or updated data pipelines for data collection and cleansing.
- An updated code repository that contains all the production-ready artifacts.
- A record of data quality issues or gaps that are reported back to data owners.

Model Operationalization

For rapid and reliable deployments in production, you need to automate the entire end-to-end process using a machine learning pipeline. This part is often the hardest, as it involves many additional steps and adds constraints for integrating with your production application(s). Let's look at a few of those steps:

- Replace all hardcoded command-line statements, file locations, user credentials, and so on with parameters and arguments.
- Remove all generic data engineering steps, handing them off to the data engineering team to embed into the data product pipelines.
- Implement continuous and automatic testing, training, and deployment for both data and models.
- Create unit and integration tests for your inference endpoints for enhanced debugging and accelerated time to deployment.
- Add statements for collecting additional metrics: features, model accuracy, visuals, record counts, and so on.
- Implement A/B or shadow testing, and validate bias and accuracy before deploying into production.
- Share lessons learned and best practices with other teams or team members.
- Integrate the model with the rest of your applications. For example, you may need to create a composite service for combining and integrating data from different places.
- Consider using frameworks for interoperability and environment migration, for example, using standards such as ONNX. This allows developers to host the output in a number of environments, stacks, and programming languages. It even allows groups of developers to combine ML models using different ML frameworks.
- Add information about the model itself to a central model catalog. For example, you should note whether any discriminating features are used or if personally identifiable data was used for training.

Rather than having data scientists spend a lot of their time and effort setting up environments, I recommend defining standard environments, languages, and libraries that allow data scientists to work efficiently. Another recommendation is to design your runtime platforms to always be stateless. They preferably won't persist data over the long term, although they can create temporary data or hold reference data. Any data that they must use should come from input folders and be written back to output folders. This is important because it means you can easily replace models without moving or migrating any data.

At the end of this stage, your list of artifacts will be even longer. New deliverables may include:

- Updated workspace objects: models, pipelines, metrics, new datasets, and so on
- An updated code repository that contains all production artifacts
- A status dashboard that displays model and system health and key metrics

- A final modeling report with deployment details
- A final solution architecture document
- New or updated data pipelines for data collection and cleansing
- Adjustments to other applications for integrating with the machine learning application
- Updated code templates
- A final project document

Exceptions

As you can see, MLOps is a complex process. To further complicate matters, there are several factors that might cause variations to your generic architecture. For example:

- Pipelines can be engineered to require manual approval after retraining or to stop automatically when data quality reaches a specific threshold.
- Model retraining can be triggered in several ways: a new release process for an updated model, a business event, new data coming in, by hand, etc. The model deployment pipeline can also be tightly coupled with the data engineering pipeline, so the model is automatically retrained and deployed when new data comes in.
- The design and structure of the DDS can vary based on your requirements. You might want to use a polyglot database design to validate different data structures and read patterns.
- To monitor bias, fairness, and the quality of your models, you can capture the output and automatically check for anomalous results. Multiple models can run simultaneously in this situation.
- Explainability might require you to version and tag all training data and models automatically.
- The output of models can be served back in many different ways, including via batches that include multiple scoring results in one set, via events that are generated by analyzing other events, or via API endpoints (i.e., request/response web services).
- Unstructured data can make the architecture look different. You might want to use external or cognitive services, or share data with external parties.
- Models may be deployed and running in different environments. To have an end-to-end oversight, you may need to add an extra central metadata repository to log, store, display, organize, compare, and query all metadata generated during the machine learning processes.
- Not all cognitive services can be trained or easily customized. For example, OpenAI’s ChatGPT comes as is; there’s no way to change how the service behaves. This means that it might be difficult to detect bias or monitor the service over time.

Consider putting principles in place about what types of cognitive services are allowed or require approval.

- You might need to implement complex composite services or read stores for first collecting and aggregating data, before making any API calls to the model itself.

As you can imagine, these factors largely depend on the business requirements of the models and how they are consumed by or integrated into other downstream processes. There will always be exceptions and differences in the details of how analytical models are implemented, but the generic architecture, with all of its best practices, will help you to be much more scalable.

Wrapping Up

Turning data into value is difficult. However, you've seen that by taking data management and automation seriously, you can democratize data usage at large. In this chapter, we focused on the data-consuming side of the architecture. We looked at different patterns of consumption (direct, or through domain data stores), and we walked through best practices for building DDSs. We learned to smash silos and keep dependencies between DDSs to a minimum by analyzing both functional and nonfunctional requirements. We also touched upon self-service and managed data activities and the principles that support them. Finally, we examined business intelligence and machine learning, the common denominators in most data-driven decision solutions. As you saw, with good processes in place, you can manage these activities end-to-end.

You also learned that to achieve a faster time to value, it's important to remove manual effort for tasks like versioning, analytics monitoring, and deploying. This requires a different culture that goes way beyond the traditional data modeling. Your data architects will need to become well-trained and experienced software engineers. It also requires that automation is embraced broadly. The following best practices will help:

- Guide your teams in how to manage overlapping concerns. Pay attention to setting boundaries: if they're too fine-grained, this can lead to overlap and repeated activities; if they're too coarse-grained, you can end up with too many coupling points.
- Align analytical consumption patterns with your landing zones. Develop blueprints for implementing standard and reusable functionality. Align these blueprints with your playbooks and best practices. Make concise choices about aligning your data-providing capabilities with your data-consuming capabilities.
- Set up a data modeling community within your organization for guiding your teams and ensuring data is transformed in line with your company standards.
- Pay attention to the providing side and the consuming side. On the consumer side, you generally see heavy diversification in the way data is used. This contrasts with the providing side, which can be more easily standardized. When combining these two concerns in situations where data consumers are also data providers, you risk tight coupling when analytical solutions are classified as data products. Therefore, I advocate to strictly separate data usage from data distribution by using two type of blueprints: one for data product creation and another for turning data into value.

- Plan the alignment of your configurations, workspaces, and domains. A workspace is the top-level resource in which analytical services, such as machine learning, are managed. From a manageability standpoint, it's important to map out how these workspaces are aligned with your activities. For example, you may have a workspace per team, with each team getting its own workspace instance. The benefit of such an alignment is that everything is managed within one place. Team members can easily access, explore, and reuse results. Alternatively, you might have one workspace per project. The benefit of this approach is that you manage costs at the project level and segregate conflicting (security) concerns. However, the discovery and reuse of assets might be more difficult because assets are spread across multiple workspace instances. Another consideration is that the management overhead becomes larger when more workspaces are used.
- Work closely with your infrastructure and networking team on how linked services and networks are integrated. For example, certain endpoints or linked services may require the network or DNS records to be changed by the central infrastructure team. A recommendation for this is to apply automation rather than using a ticket system: listen for cloud-based events, trigger workflows, invoke functions, and apply policies.
- Use code repositories to decouple generic blueprint templates from actual implementations. For example, each team might have its own repository in which they manage parameters for different environments, customizations, and so on. At a higher level, implement a policy-driven governance framework to prevent noncompliance by either restricting resources from being created or modifying settings to make resources compliant.

Following these recommendations will make your architecture more cost-efficient and interoperable. It will also help you manage your data in a more consistent way. We're almost at the end of our trip, but now we're going to get back in our helicopter one last time and pick up some speed. In this final part of the journey, we'll combine everything we've learned so far by putting theory into practice. May the skies treat you well!

1 *Polyglot persistence* is the term used to describe solutions that use a mix of data storage technologies.

2 In my previous role, I used the terms *syntactic* and *semantical* data changes. Syntactic data changes don't change the meaning of the data, although it might be presented differently. Semantical data changes change the context and thus the meaning.

3 Some engineers in the field use the term *cooked data* to make it clear that the data has been processed.

4 Domain-driven design uses the term *shared kernel* to indicate that part of the domain model is shared between different teams or subdomains. The shared kernel integration strategy reduces duplication and overhead.

5 A similar layering can be observed in data product development and the establishment of (providing) domain boundaries, as discussed in Chapter 4. However, there is a clear difference between data product architecture and DDS architecture. Within data product

architectures, there's a strong source system alignment. Within DDSs, you don't see this alignment, as the integration activities typically span across multiple sources.

6 Within machine learning, there's a common principle that more data beats clever algorithms.

7 The *grain* of a relation defines what each row represents in the relation. In a table like products, the grain might be a single product, so every product is in its own row, with exactly one row per product. By ensuring that your grains are clear, you specify exactly what a table record contains. This gives users insight into what data can be combined.

8 Nick Tune talks about inner sourcing in his blog post on how to better manage and eliminate cross-team dependencies.

9 A *shuffle* occurs when a part of a distributed table is moved to a different node during query execution. More information on this can be found in the Azure docs.

10 Christian Kaul maintains an extensive list of resources on data modeling.

11 Sean Kandel et al. wrote an interesting scientific paper on a data preparation tool called Wrangler.

12 Microsoft describes the three primary strategies for how business intelligence content can be owned and managed in the Power BI documentation.

13 Google researchers D. Sculley et al. have written a paper that describes the hidden technical debt in machine learning systems. Typically, only a small fraction of a real-world ML system is composed of ML code. The required surrounding infrastructure is often vast and complex.

14 *Feature creation* or *feature engineering* is a machine learning technique that leverages data to create new variables that aren't in the training set.

15 Some data scientists abbreviate operationalization as "o16n."

16 *Environments* are an encapsulation of the environment where your machine learning training happens. They typically specify the Python packages, environment variables, and software settings around your training and scoring scripts. They also specify runtimes for languages and tools like Python, Spark, and Docker.

17 A *workspace* is the top-level resource in which your artifacts are managed.

18 *Features* are the columns of data in your input data. For example, if you're trying to predict someone's age, your input features might be things like height and hair color. The *label* is the final output: 12 years, 78 years, and so on.

19 As described in my blog post "Feature Stores and Data Mesh", I'm no big fan of centralized or shared stores: they add unnecessary complexity to your architecture.