you'll encounter across the data engineering lifecycle. First, think about whether you want resource contention with your nodes. If not, exercise a *shared-nothing architecture*: a single node handles each request, meaning other nodes do not share resources such as memory, disk, or CPU with this node or with each other. Data and resources are isolated to the node. Alternatively, various nodes can handle multiple requests and share resources but at the risk of resource contention. Another consideration is whether nodes should share the same disk and memory accessible by all nodes. This is called a *shared disk architecture* and is common when you want shared resources if a random node failure occurs.

Monoliths

The general notion of a monolith includes as much as possible under one roof; in its most extreme version, a monolith consists of a single codebase running on a single machine that provides both the application logic and user interface.

Coupling within monoliths can be viewed in two ways: technical coupling and domain coupling. *Technical coupling* refers to architectural tiers, while *domain coupling* refers to the way domains are coupled together. A monolith has varying degrees of coupling among technologies and domains. You could have an application with various layers decoupled in a multitier architecture but still share multiple domains. Or, you could have a single-tier architecture serving a single domain.

The tight coupling of a monolith implies a lack of modularity of its components. Swapping out or upgrading components in a monolith is often an exercise in trading one pain for another. Because of the tightly coupled nature, reusing components across the architecture is difficult or impossible. When evaluating how to improve a monolithic architecture, it's often a game of whack-a-mole: one component is improved, often at the expense of unknown consequences with other areas of the monolith.

Data teams will often ignore solving the growing complexity of their monolith, letting it devolve into a big ball of mud.

Chapter 4 provides a more extensive discussion comparing monoliths to distributed technologies. We also discuss the *distributed monolith*, a strange hybrid that emerges when engineers build distributed systems with excessive tight coupling.

Microservices

Compared with the attributes of a monolith—interwoven services, centralization, and tight coupling among services—microservices are the polar opposite. *Microservices architecture* comprises separate, decentralized, and loosely coupled services. Each service has a specific function and is decoupled from other services operating within its domain. If one service temporarily goes down, it won't affect the ability of other services to continue functioning.

A question that comes up often is how to convert your monolith into many microservices (Figure 3-7). This completely depends on how complex your monolith is and how much effort it will be to start extracting services out of it. It's entirely possible that your monolith cannot be broken apart, in which case, you'll want to start creating a new parallel architecture that has the services decoupled in a microservices-friendly manner. We don't suggest an entire refactor but instead break out services. The monolith didn't arrive overnight and is a technology issue as an organizational one. Be sure you get buy-in from stakeholders of the monolith if you plan to break it apart.

If you'd like to learn more about breaking apart a monolith, we suggest reading the fantastic, pragmatic guide *Software Architecture: The Hard Parts* by Neal Ford et al. (O'Reilly).

Considerations for data architecture

As we mentioned at the start of this section, the concepts of tight versus loose coupling stem from software development, with some of these concepts dating back over 20 years. Though architectural practices in data are now adopting those from software development, it's still common to see very monolithic, tightly coupled data architectures. Some of this is due to the nature of existing data technologies and the way they integrate.

For example, data pipelines might consume data from many sources ingested into a central data warehouse. The central data warehouse is inherently monolithic. A move toward a microservices equivalent with a data warehouse is to decouple the workflow with domain-specific data pipelines connecting to corresponding domain-specific data warehouses. For example, the sales data pipeline connects to the sales-specific data warehouse, and the inventory and product domains follow a similar pattern.

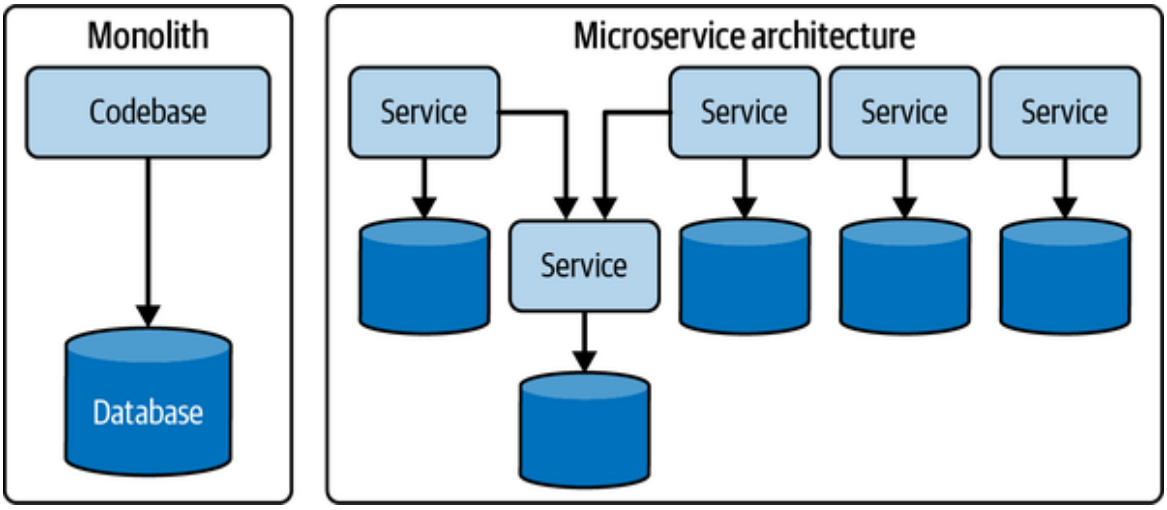

Figure 3-7. An extremely monolithic architecture runs all functionality inside a single codebase, potentially colocating a database on the same host server

Rather than dogmatically preach microservices over monoliths (among other arguments), we suggest you pragmatically use loose coupling as an ideal, while recognizing the state and limitations of the data technologies you're using within your data architecture. Incorporate reversible technology choices that allow for modularity and loose coupling whenever possible.

As you can see in Figure 3-7, you separate the components of your architecture into different layers of concern in a vertical fashion. While a multitier architecture solves the technical challenges of decoupling shared resources, it does not address the complexity of sharing domains. Along the lines of single versus multitiered architecture, you should also consider how you separate the domains of your data architecture. For example, your analyst team might rely on data from sales and inventory. The sales and inventory domains are different and should be viewed as separate.

One approach to this problem is centralization: a single team is responsible for gathering data from all domains and reconciling it for consumption across the organization. (This is a common approach in traditional data warehousing.) Another approach is the *data mesh*. With the data mesh, each software team is responsible for preparing its data for consumption across the rest of the organization. We'll say more about the data mesh later in this chapter.

Our advice: monoliths aren't necessarily bad, and it might make sense to start with one under certain conditions. Sometimes you need to move fast, and it's much simpler to start with a monolith. Just be prepared to break it into smaller pieces eventually; don't get too comfortable.

User Access: Single Versus Multitenant

As a data engineer, you have to make decisions about sharing systems across multiple teams, organizations, and customers. In some sense, all cloud services are multitenant, although this multitenancy occurs at various grains. For example, a cloud compute instance is usually on a shared server, but the VM itself provides some degree of isolation. Object storage is a multitenant system, but cloud vendors guarantee security and isolation so long as customers configure their permissions correctly.

Engineers frequently need to make decisions about multitenancy at a much smaller scale. For example, do multiple departments in a large company share the same data warehouse? Does the organization share data for multiple large customers within the same table?

We have two factors to consider in multitenancy: performance and security. With multiple large tenants within a cloud system, will the system support consistent performance for all tenants, or will there be a noisy neighbor problem? (That is, will high usage from one tenant degrade performance for other tenants?) Regarding security, data from different tenants must be properly isolated. When a company has multiple external customer tenants, these tenants should not be aware of one another, and engineers must prevent data leakage. Strategies for data isolation vary by system. For

instance, it is often perfectly acceptable to use multitenant tables and isolate data through views. However, you must make certain that these views cannot leak data. Read vendor or project documentation to understand appropriate strategies and risks.

Event-Driven Architecture

Your business is rarely static. Things often happen in your business, such as getting a new customer, a new order from a customer, or an order for a product or service. These are all examples of *events* that are broadly defined as something that happened, typically a change in the *state* of something. For example, a new order might be created by a customer, or a customer might later make an update to this order.

An event-driven workflow (Figure 3-8) encompasses the ability to create, update, and asynchronously move events across various parts of the data engineering lifecycle. This workflow boils down to three main areas: event production, routing, and consumption. An event must be produced and routed to something that consumes it without tightly coupled dependencies among the producer, event router, and consumer.

Figure 3-8. In an event-driven workflow, an event is produced, routed, and then consumed

An event-driven architecture (Figure 3-9) embraces the event-driven workflow and uses this to communicate across various services. The advantage of an event-driven architecture is that it distributes the state of an event across multiple services. This is helpful if a service goes offline, a node fails in a distributed system, or you'd like multiple consumers or services to access the same events. Anytime you have loosely coupled services, this is a candidate for event-driven architecture. Many of the examples we describe later in this chapter incorporate some form of event-driven architecture.

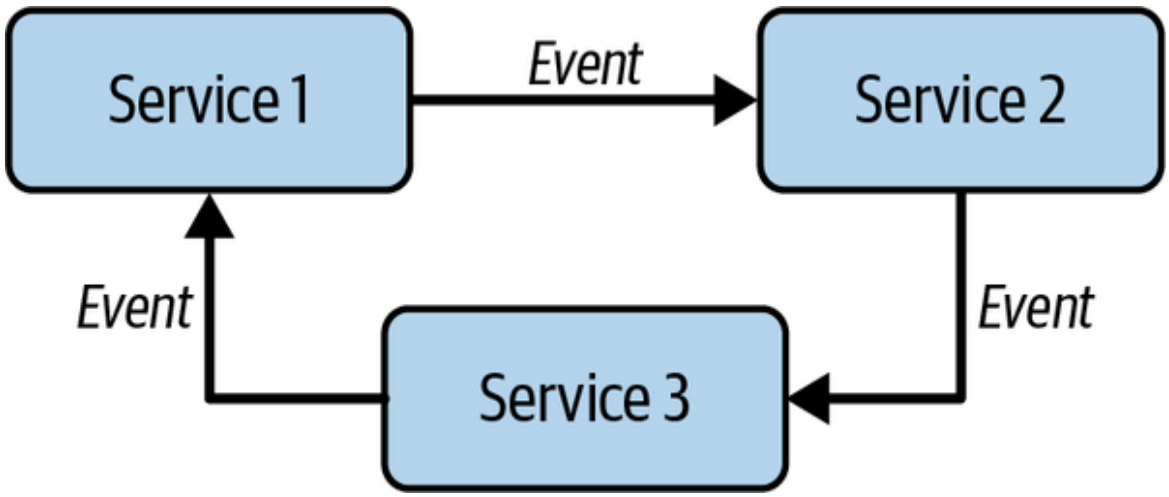

Figure 3-9. In an event-driven architecture, events are passed between loosely coupled services

You'll learn more about event-driven streaming and messaging systems in Chapter 5.

Brownfield Versus Greenfield Projects

Before you design your data architecture project, you need to know whether you're starting with a clean slate or redesigning an existing architecture. Each type of project requires assessing trade-offs, albeit with different considerations and approaches. Projects roughly fall into two buckets: brownfield and greenfield.

Brownfield projects

Brownfield projects often involve refactoring and reorganizing an existing architecture and are constrained by the choices of the present and past. Because a key part of architecture is change management, you must figure out a way around these limitations and design a path forward to achieve your new business and technical objectives. Brownfield projects require a thorough understanding of the legacy architecture and the interplay of various old and new technologies. All too often, it's easy to criticize a prior team's work and decisions, but it is far better to dig deep, ask questions, and understand why decisions were made. Empathy and context go a long way in helping you diagnose problems with the existing architecture, identify opportunities, and recognize pitfalls.

You'll need to introduce your new architecture and technologies and deprecate the old stuff at some point. Let's look at a couple of popular approaches. Many teams jump headfirst into an all-at-once or big-bang overhaul of the old architecture, often figuring out deprecation as they go. Though popular, we don't advise this approach because of the associated risks and lack of a plan. This path often leads to disaster, with many irreversible and costly decisions. Your job is to make reversible, high-ROI decisions.

A popular alternative to a direct rewrite is the strangler pattern: new systems slowly and incrementally replace a legacy architecture's components.[7] Eventually, the legacy architecture is completely replaced. The attraction to the strangler pattern is its targeted and surgical approach of deprecating one piece of a system at a time. This allows for flexible and reversible decisions while assessing the impact of the deprecation on dependent systems.

It's important to note that deprecation might be "ivory tower" advice and not practical or achievable. Eradicating legacy technology or architecture might be impossible if you're at a large organization. Someone, somewhere, is using these legacy components. As someone once said, "Legacy is a condescending way to describe something that makes money."

If you can deprecate, understand there are numerous ways to deprecate your old architecture. It is critical to demonstrate value on the new platform by gradually increasing its maturity to show evidence of success and then follow an exit plan to shut down old systems.

Greenfield projects

On the opposite end of the spectrum, a *greenfield project* allows you to pioneer a fresh start, unconstrained by the history or legacy of a prior architecture. Greenfield projects tend to be easier than brownfield projects, and many data architects and engineers find them more fun! You have the opportunity to try the newest and coolest tools and architectural patterns. What could be more exciting?

You should watch out for some things before getting too carried away. We see teams get overly exuberant with shiny object syndrome. They feel compelled to reach for the latest and greatest technology fad without understanding how it will impact the value of the project. There's also a temptation to do *resume-driven development*, stacking up impressive new technologies without prioritizing the project's ultimate goals.[8] Always prioritize requirements over building something cool.

Whether you're working on a brownfield or greenfield project, always focus on the tenets of "good" data architecture. Assess trade-offs, make flexible and reversible decisions, and strive for positive ROI.

Now, we'll look at examples and types of architectures—some established for decades (the data warehouse), some brand-new (the data lakehouse), and some that quickly came and went but still influence current architecture patterns (Lambda architecture).

Examples and Types of Data Architecture

Because data architecture is an abstract discipline, it helps to reason by example. In this section, we outline prominent examples and types of data architecture that are popular today. Though this set of examples is by no means exhaustive, the intention is to expose you to some of the most common data architecture patterns and to get you thinking about the requisite flexibility and trade-off analysis needed when designing a good architecture for your use case.

Data Warehouse

A *data warehouse* is a central data hub used for reporting and analysis. Data in a data warehouse is typically highly formatted and structured for analytics use cases. It's among the oldest and most well-established data architectures.

In 1990, Bill Inmon originated the notion of the data warehouse, which he described as "a subject-oriented, integrated, nonvolatile, and time-variant

collection of data in support of management's decisions."[9] Though technical aspects of the data warehouse have evolved significantly, we feel this original definition still holds its weight today.

In the past, data warehouses were widely used at enterprises with significant budgets (often in the millions of dollars) to acquire data systems and pay internal teams to provide ongoing support to maintain the data warehouse. This was expensive and labor-intensive. Since then, the scalable, pay-as-you-go model has made cloud data warehouses accessible even to tiny companies. Because a third-party provider manages the data warehouse infrastructure, companies can do a lot more with fewer people, even as the complexity of their data grows.

It's worth noting two types of data warehouse architecture: organizational and technical. The *organizational data warehouse architecture* organizes data associated with certain business team structures and processes. The *technical data warehouse architecture* reflects the technical nature of the data warehouse, such as MPP. A company can have a data warehouse without an MPP system or run an MPP system that is not organized as a data warehouse. However, the technical and organizational architectures have existed in a virtuous cycle and are frequently identified with each other.

The organizational data warehouse architecture has two main characteristics:

Separates analytics processes (OLAP) from production databases (online transaction processing)

> This separation is critical as businesses grow. Moving data into a separate physical system directs load away from production systems and improves analytics performance.

Centralizes and organizes data

> Traditionally, a data warehouse pulls data from application systems by using ETL. The extract phase pulls data from source systems. The transformation phase cleans and standardizes data, organizing and

imposing business logic in a highly modeled form. (Chapter 8 covers transformations and data models.) The load phase pushes data into the data warehouse target database system. Data is loaded into multiple data marts that serve the analytical needs for specific lines or business and departments. Figure 3-10 shows the general workflow. The data warehouse and ETL go hand in hand with specific business structures, including DBA and ETL developer teams that implement the direction of business leaders to ensure that data for reporting and analytics corresponds to business processes. An ETL system is not a prerequisite for a data warehouse, as you will learn when we discuss another load and transformation pattern, ELT.

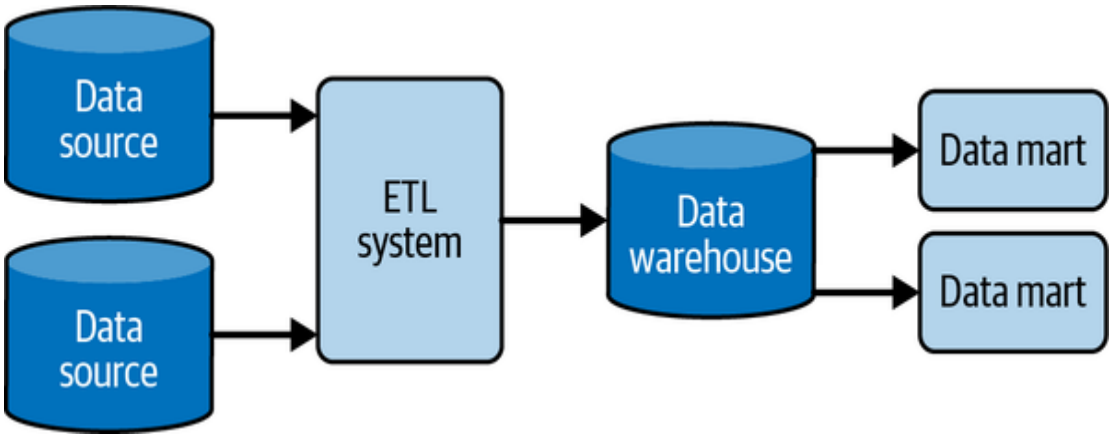

Figure 3-10. Basic data warehouse with ETL

Regarding the technical data warehouse architecture, the first MPP systems in the late 1970s became popular in the 1980s. MPPs support essentially the same SQL semantics used in relational application databases. Still, they are optimized to scan massive amounts of data in parallel and thus allow high-performance aggregation and statistical calculations. In recent years, MPP systems have increasingly shifted from a row-based to a columnar architecture to facilitate even larger data and queries, especially in cloud data warehouses. MPPs are indispensable for running performant queries for large enterprises as data and reporting needs grow.

One variation on ETL is ELT. With the ELT data warehouse architecture, data gets moved more or less directly from production systems into a

staging area in the data warehouse. Staging in this setting indicates that the data is in a raw form. Rather than using an external system, transformations are handled directly in the data warehouse. The intention is to take advantage of the massive computational power of cloud data warehouses and data processing tools. Data is processed in batches, and transformed output is written into tables and views for analytics. Figure 3-11 shows the general process. ELT is also popular in a streaming arrangement, as events are streamed from a CDC process, stored in a staging area, and then subsequently transformed within the data warehouse.

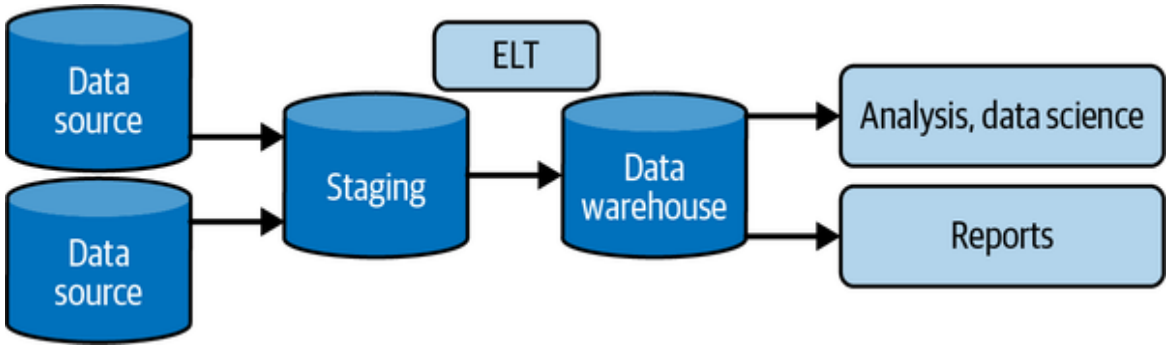

Figure 3-11. ELT—extract, load, and transform

A second version of ELT was popularized during big data growth in the Hadoop ecosystem. This is *transform-on-read ELT*, which we discuss in "Data Lake".

The cloud data warehouse

Cloud data warehouses represent a significant evolution of the on-premises data warehouse architecture and have thus led to significant changes to the organizational architecture. Amazon Redshift kicked off the cloud data warehouse revolution. Instead of needing to appropriately size an MPP system for the next several years and sign a multimillion-dollar contract to procure the system, companies had the option of spinning up a Redshift cluster on demand, scaling it up over time as data and analytics demand grew. They could even spin up new Redshift clusters on demand to serve specific workloads and quickly delete clusters when they were no longer needed.

Google BigQuery, Snowflake, and other competitors popularized the idea of separating compute from storage. In this architecture, data is housed in object storage, allowing virtually limitless storage. This also gives users the option to spin up computing power on demand, providing ad hoc big data capabilities without the long-term cost of thousands of nodes.

Cloud data warehouses expand the capabilities of MPP systems to cover many big data use cases that required a Hadoop cluster in the very recent past. They can readily process petabytes of data in a single query. They typically support data structures that allow the storage of tens of megabytes of raw text data per row or extremely rich and complex JSON documents. As cloud data warehouses (and data lakes) mature, the line between the data warehouse and the data lake will continue to blur.

So significant is the impact of the new capabilities offered by cloud data warehouses that we might consider jettisoning the term *data warehouse* altogether. Instead, these services are evolving into a new data platform with much broader capabilities than those offered by a traditional MPP system.

Data marts

A *data mart* is a more refined subset of a warehouse designed to serve analytics and reporting, focused on a single suborganization, department, or line of business; every department has its own data mart, specific to its needs. This is in contrast to the full data warehouse that serves the broader organization or business.

Data marts exist for two reasons. First, a data mart makes data more easily accessible to analysts and report developers. Second, data marts provide an additional stage of transformation beyond that provided by the initial ETL or ELT pipelines. This can significantly improve performance if reports or analytics queries require complex joins and aggregations of data, especially when the raw data is large. Transform processes can populate the data mart with joined and aggregated data to improve performance for live queries. Figure 3-12 shows the general workflow. We discuss data marts, and modeling data for data marts, in Chapter 8.

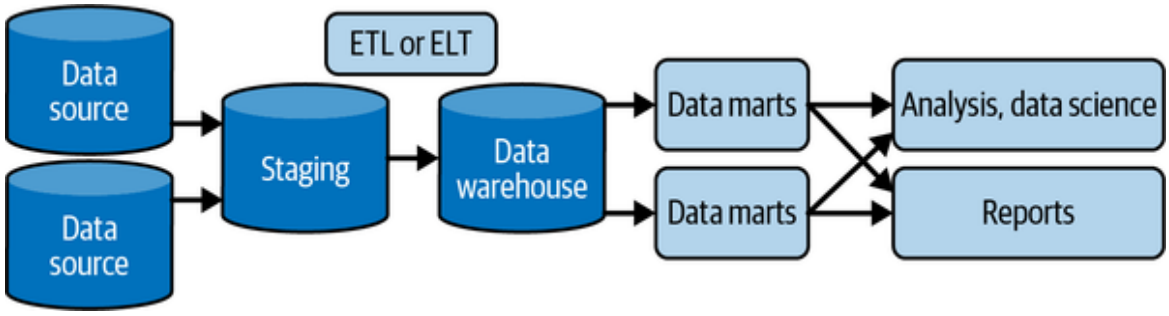

Figure 3-12. ETL or ELT plus data marts

Data Lake

Among the most popular architectures that appeared during the big data era is the *data lake*. Instead of imposing tight structural limitations on data, why not simply dump all of your data—structured and unstructured—into a central location? The data lake promised to be a democratizing force, liberating the business to drink from a fountain of limitless data. The first-generation data lake, "data lake 1.0," made solid contributions but generally failed to deliver on its promise.

Data lake 1.0 started with HDFS. As the cloud grew in popularity, these data lakes moved to cloud-based object storage, with extremely cheap storage costs and virtually limitless storage capacity. Instead of relying on a monolithic data warehouse where storage and compute are tightly coupled, the data lake allows an immense amount of data of any size and type to be stored. When this data needs to be queried or transformed, you have access to nearly unlimited computing power by spinning up a cluster on demand, and you can pick your favorite data-processing technology for the task at hand—MapReduce, Spark, Ray, Presto, Hive, etc.

Despite the promise and hype, data lake 1.0 had serious shortcomings. The data lake became a dumping ground; terms such as *data swamp*, *dark data*, and *WORN* were coined as once-promising data projects failed. Data grew to unmanageable sizes, with little in the way of schema management, data cataloging, and discovery tools. In addition, the original data lake concept was essentially write-only, creating huge headaches with the arrival of regulations such as GDPR that required targeted deletion of user records.

Processing data was also challenging. Relatively banal data transformations such as joins were a huge headache to code as MapReduce jobs. Later frameworks such as Pig and Hive somewhat improved the situation for data processing but did little to address the basic problems of data management. Simple data manipulation language (DML) operations common in SQL—deleting or updating rows—were painful to implement, generally achieved by creating entirely new tables. While big data engineers radiated a particular disdain for their counterparts in data warehousing, the latter could point out that data warehouses provided basic data management capabilities out of the box, and that SQL was an efficient tool for writing complex, performant queries and transformations.

Data lake 1.0 also failed to deliver on another core promise of the big data movement. Open source software in the Apache ecosystem was touted as a means to avoid multimillion-dollar contracts for proprietary MPP systems. Cheap, off-the-shelf hardware would replace custom vendor solutions. In reality, big data costs ballooned as the complexities of managing Hadoop clusters forced companies to hire large teams of engineers at high salaries. Companies often chose to purchase licensed, customized versions of Hadoop from vendors to avoid the exposed wires and sharp edges of the raw Apache codebase and acquire a set of scaffolding tools to make Hadoop more user-friendly. Even companies that avoided managing Hadoop clusters using cloud storage had to spend big on talent to write MapReduce jobs.

We should be careful not to understate the utility and power of first-generation data lakes. Many organizations found significant value in data lakes—especially huge, heavily data-focused Silicon Valley tech companies like Netflix and Facebook. These companies had the resources to build successful data practices and create their custom Hadoop-based tools and enhancements. But for many organizations, data lakes turned into an internal superfund site of waste, disappointment, and spiraling costs.

Convergence, Next-Generation Data Lakes, and the Data Platform

In response to the limitations of first-generation data lakes, various players have sought to enhance the concept to fully realize its promise. For example, Databricks introduced the notion of a *data lakehouse*. The lakehouse incorporates the controls, data management, and data structures found in a data warehouse while still housing data in object storage and supporting a variety of query and transformation engines. In particular, the data lakehouse supports atomicity, consistency, isolation, and durability (ACID) transactions, a big departure from the original data lake, where you simply pour in data and never update or delete it. The term *data lakehouse* suggests a convergence between data lakes and data warehouses.

The technical architecture of cloud data warehouses has evolved to be very similar to a data lake architecture. Cloud data warehouses separate compute from storage, support petabyte-scale queries, store unstructured text and semistructured objects, and integrate with advanced processing technologies such as Spark or Beam.

We believe that the trend of convergence will only continue. The data lake and the data warehouse will still exist as different architectures. In practice, their capabilities will converge so that few users will notice a boundary between them in their day-to-day work. We now see several vendors offering *data platforms* that combine data lake and data warehouse capabilities. From our perspective, AWS, Azure, Google Cloud, Snowflake, and Databricks are class leaders, each offering a constellation of tightly integrated tools for working with data, running the gamut from relational to completely unstructured. Instead of choosing between a data lake or data warehouse architecture, future data engineers will have the option to choose a converged data platform based on a variety of factors, including vendor, ecosystem, and relative openness.

Modern Data Stack

The *modern data stack* (Figure 3-13) is currently a trendy analytics architecture that highlights the type of abstraction we expect to see more widely used over the next several years. Whereas past data stacks relied on expensive, monolithic toolsets, the main objective of the modern data stack

is to use cloud-based, plug-and-play, easy-to-use, off-the-shelf components to create a modular and cost-effective data architecture. These components include data pipelines, storage, transformation, data management/governance, monitoring, visualization, and exploration. The domain is still in flux, and the specific tools are changing and evolving rapidly, but the core aim will remain the same: to reduce complexity and increase modularization. Note that the notion of a modern data stack integrates nicely with the converged data platform idea from the previous section.

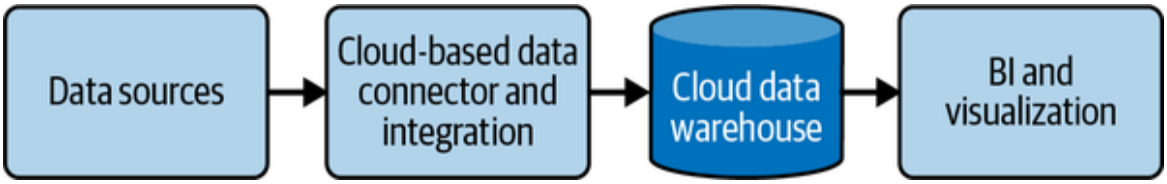

Figure 3-13. Basic components of the modern data stack

Key outcomes of the modern data stack are self-service (analytics and pipelines), agile data management, and using open source tools or simple proprietary tools with clear pricing structures. Community is a central aspect of the modern data stack as well. Unlike products of the past that had releases and roadmaps largely hidden from users, projects and companies operating in the modern data stack space typically have strong user bases and active communities that participate in the development by using the product early, suggesting features, and submitting pull requests to improve the code.

Regardless of where "modern" goes (we share our ideas in Chapter 11), we think the key concept of plug-and-play modularity with easy-to-understand pricing and implementation is the way of the future. Especially in analytics engineering, the modern data stack is and will continue to be the default choice of data architecture. Throughout the book, the architecture we reference contains pieces of the modern data stack, such as cloud-based and plug-and-play modular components.

Lambda Architecture

In the "old days" (the early to mid-2010s), the popularity of working with streaming data exploded with the emergence of Kafka as a highly scalable message queue and frameworks such as Apache Storm and Samza for streaming/real-time analytics. These technologies allowed companies to perform new types of analytics and modeling on large amounts of data, user aggregation and ranking, and product recommendations. Data engineers needed to figure out how to reconcile batch and streaming data into a single architecture. The Lambda architecture was one of the early popular responses to this problem.

In a *Lambda architecture* (Figure 3-14), you have systems operating independently of each other—batch, streaming, and serving. The source system is ideally immutable and append-only, sending data to two destinations for processing: stream, and batch. In-stream processing intends to serve the data with the lowest possible latency in a "speed" layer, usually a NoSQL database. In the batch layer, data is processed and transformed in a system such as a data warehouse, creating precomputed and aggregated views of the data. The serving layer provides a combined view by aggregating query results from the two layers.

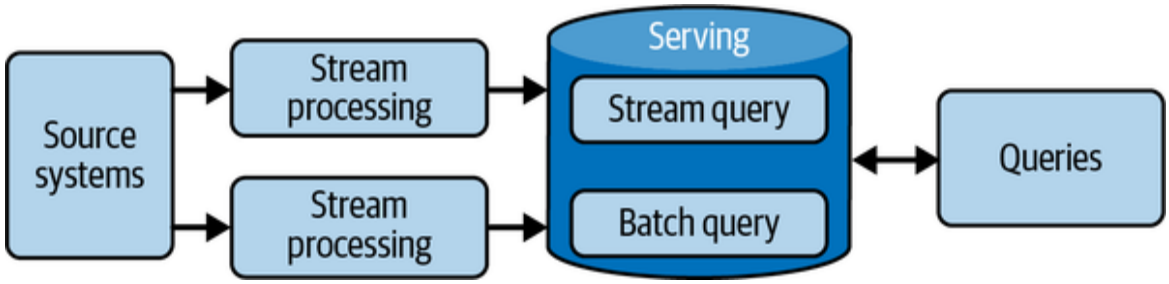

Figure 3-14. Lambda architecture

Lambda architecture has its share of challenges and criticisms. Managing multiple systems with different codebases is as difficult as it sounds, creating error-prone systems with code and data that are extremely difficult to reconcile.

We mention Lambda architecture because it still gets attention and is popular in search-engine results for data architecture. Lambda isn't our first

recommendation if you're trying to combine streaming and batch data for analytics. Technology and practices have moved on.

Next, let's look at a reaction to Lambda architecture, the Kappa architecture.

Kappa Architecture

As a response to the shortcomings of Lambda architecture, Jay Kreps proposed an alternative called *Kappa architecture* (Figure 3-15).[10] The central thesis is this: why not just use a stream-processing platform as the backbone for all data handling—ingestion, storage, and serving? This facilitates a true event-based architecture. Real-time and batch processing can be applied seamlessly to the same data by reading the live event stream directly and replaying large chunks of data for batch processing.

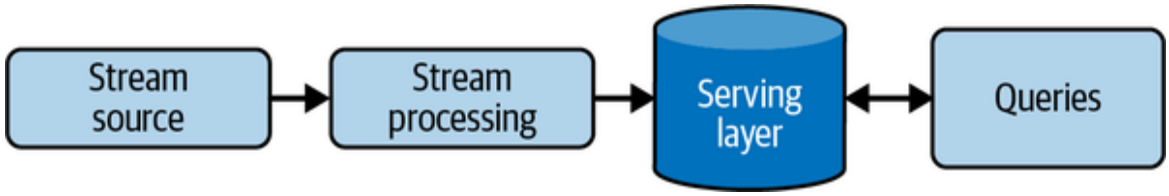

Figure 3-15. Kappa architecture

Though the original Kappa architecture article came out in 2014, we haven't seen it widely adopted. There may be a couple of reasons for this. First, streaming itself is still a bit of a mystery for many companies. Second, Kappa architecture turns out to be complicated and expensive in practice. While some streaming systems can scale to huge data volumes, they are complex and expensive; batch storage and processing remain much more efficient and cost-effective for enormous historical datasets.

The Dataflow Model and Unified Batch and Streaming

Both Lambda and Kappa sought to address limitations of the Hadoop ecosystem of the 2010s by trying to duct-tape together complicated tools that were likely not natural fits in the first place. The central challenge of

unifying batch and streaming data remained, and Lambda and Kappa both provided inspiration and groundwork for continued progress in this pursuit.

One of the central problems of managing batch and stream processing is unifying multiple code paths. While the Kappa architecture relies on a unified queuing and storage layer, one still has to confront using different tools for collecting real-time statistics or running batch aggregation jobs. Today, engineers seek to solve this in several ways. Google made its mark by developing the Dataflow model and the Apache Beam framework that implements this model.

The core idea in the Dataflow model is to view all data as events, as the aggregation is performed over various types of windows. Ongoing real-time event streams are *unbounded data*. Data batches are simply bounded event streams, and the boundaries provide a natural window. Engineers can choose from various windows for real-time aggregation, such as sliding or tumbling. Real-time and batch processing happens in the same system using nearly identical code.

The philosophy of "batch as a special case of streaming" is now more pervasive. Various frameworks such as Flink and Spark have adopted a similar approach.

Architecture for IoT

The *Internet of Things* (IoT) is the distributed collection of devices, aka *things*—computers, sensors, mobile devices, smart home devices, and anything else with an internet connection. Rather than generating data from direct human input (think data entry from a keyboard), IoT data is generated from devices that collect data periodically or continuously from the surrounding environment and transmit it to a destination. IoT devices are often low-powered and operate in low-resource/low bandwidth environments.

While the concept of IoT devices dates back at least a few decades, the smartphone revolution created a massive IoT swarm virtually overnight. Since then, numerous new IoT categories have emerged, such as smart

thermostats, car entertainment systems, smart TVs, and smart speakers. The IoT has evolved from a futurist fantasy to a massive data engineering domain. We expect IoT to become one of the dominant ways data is generated and consumed, and this section goes a bit deeper than the others you've read.

Having a cursory understanding of IoT architecture will help you understand broader data architecture trends. Let's briefly look at some IoT architecture concepts.

Devices

Devices (also known as *things*) are the physical hardware connected to the internet, sensing the environment around them and collecting and transmitting data to a downstream destination. These devices might be used in consumer applications like a doorbell camera, smartwatch, or thermostat. The device might be an AI-powered camera that monitors an assembly line for defective components, a GPS tracker to record vehicle locations, or a Raspberry Pi programmed to download the latest tweets and brew your coffee. Any device capable of collecting data from its environment is an IoT device.

Devices should be minimally capable of collecting and transmitting data. However, the device might also crunch data or run ML on the data it collects before sending it downstream—edge computing and edge machine learning, respectively.

A data engineer doesn't necessarily need to know the inner details of IoT devices but should know what the device does, the data it collects, any edge computations or ML it runs before transmitting the data, and how often it sends data. It also helps to know the consequences of a device or internet outage, environmental or other external factors affecting data collection, and how these may impact the downstream collection of data from the device.

Interfacing with devices

A device isn't beneficial unless you can get its data. This section covers some of the key components necessary to interface with IoT devices in the wild.

IoT gateway

An *IoT gateway* is a hub for connecting devices and securely routing devices to the appropriate destinations on the internet. While you can connect a device directly to the internet without an IoT gateway, the gateway allows devices to connect using extremely little power. It acts as a way station for data retention and manages an internet connection to the final data destination.

New low-power WiFi standards are designed to make IoT gateways less critical in the future, but these are just rolling out now. Typically, a swarm of devices will utilize many IoT gateways, one at each physical location where devices are present (Figure 3-16).

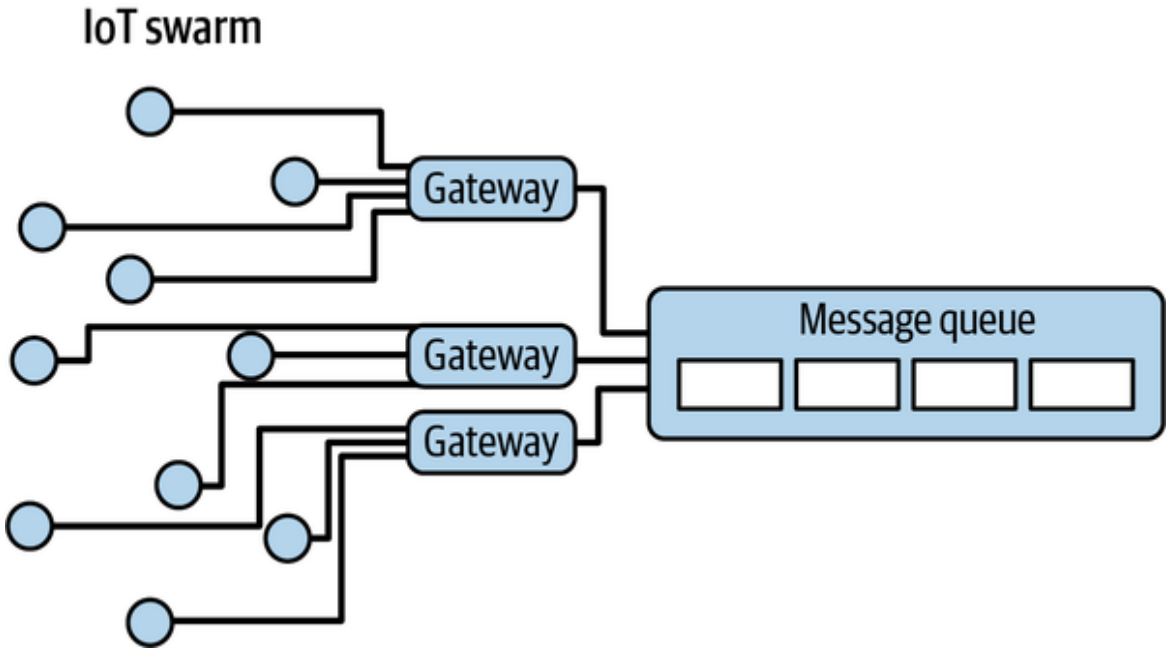

Figure 3-16. A device swarm (circles), IoT gateways, and message queue with messages (rectangles within the queue)

Ingestion

Ingestion begins with an IoT gateway, as discussed previously. From there, events and measurements can flow into an event ingestion architecture.

Of course, other patterns are possible. For instance, the gateway may accumulate data and upload it in batches for later analytics processing. In remote physical environments, gateways may not have connectivity to a network much of the time. They may upload all data only when they are brought into the range of a cellular or WiFi network. The point is that the diversity of IoT systems and environments presents complications—e.g., late-arriving data, data structure and schema disparities, data corruption, and connection disruption—that engineers must account for in their architectures and downstream analytics.

Storage

Storage requirements will depend a great deal on the latency requirement for the IoT devices in the system. For example, for remote sensors collecting scientific data for analysis at a later time, batch object storage may be perfectly acceptable. However, near real-time responses may be expected from a system backend that constantly analyzes data in a home monitoring and automation solution. In this case, a message queue or time-series database is more appropriate. We discuss storage systems in more detail in Chapter 6.

Serving

Serving patterns are incredibly diverse. In a batch scientific application, data might be analyzed using a cloud data warehouse and then served in a report. Data will be presented and served in numerous ways in a home-monitoring application. Data will be analyzed in the near time using a stream-processing engine or queries in a time-series database to look for critical events such as a fire, electrical outage, or break-in. Detection of an anomaly will trigger alerts to the homeowner, the fire department, or other entity. A batch analytics component also exists—for example, a monthly report on the state of the home.

One significant serving pattern for IoT looks like reverse ETL (Figure 3-17), although we tend not to use this term in the IoT context. Think of this scenario: data from sensors on manufacturing devices is collected and analyzed. The results of these measurements are processed to look for optimizations that will allow equipment to operate more efficiently. Data is sent back to reconfigure the devices and optimize them.

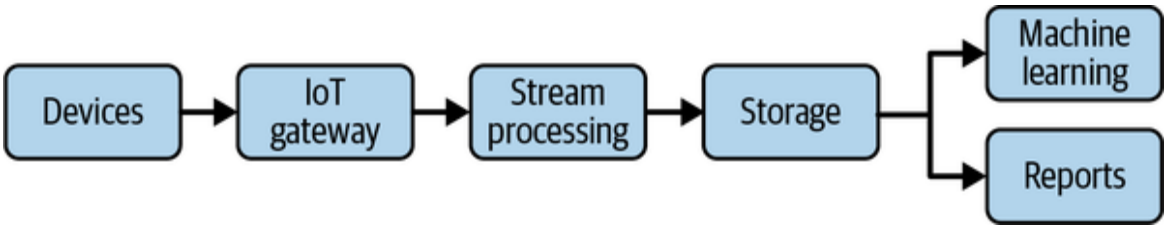

Figure 3-17. IoT serving pattern for downstream use cases

Scratching the surface of the IoT

IoT scenarios are incredibly complex, and IoT architecture and systems are also less familiar to data engineers who may have spent their careers working with business data. We hope that this introduction will encourage interested data engineers to learn more about this fascinating and rapidly evolving specialization.

Data Mesh

The *data mesh* is a recent response to sprawling monolithic data platforms, such as centralized data lakes and data warehouses, and "the great divide of data," wherein the landscape is divided between operational data and analytical data.[11] The data mesh attempts to invert the challenges of centralized data architecture, taking the concepts of domain-driven design (commonly used in software architectures) and applying them to data architecture. Because the data mesh has captured much recent attention, you should be aware of it.

A big part of the data mesh is decentralization, as Zhamak Dehghani noted in her groundbreaking article on the topic:

> *In order to decentralize the monolithic data platform, we need to reverse how we think about data, its locality, and ownership. Instead of flowing the data from domains into a centrally owned data lake or platform, domains need to host and serve their domain datasets in an easily consumable way.*[12]

Dehghani later identified four key components of the data mesh:[13]

- Domain-oriented decentralized data ownership and architecture
- Data as a product
- Self-serve data infrastructure as a platform
- Federated computational governance

Figure 3-18 shows a simplified version of a data mesh architecture, with the three domains interoperating.

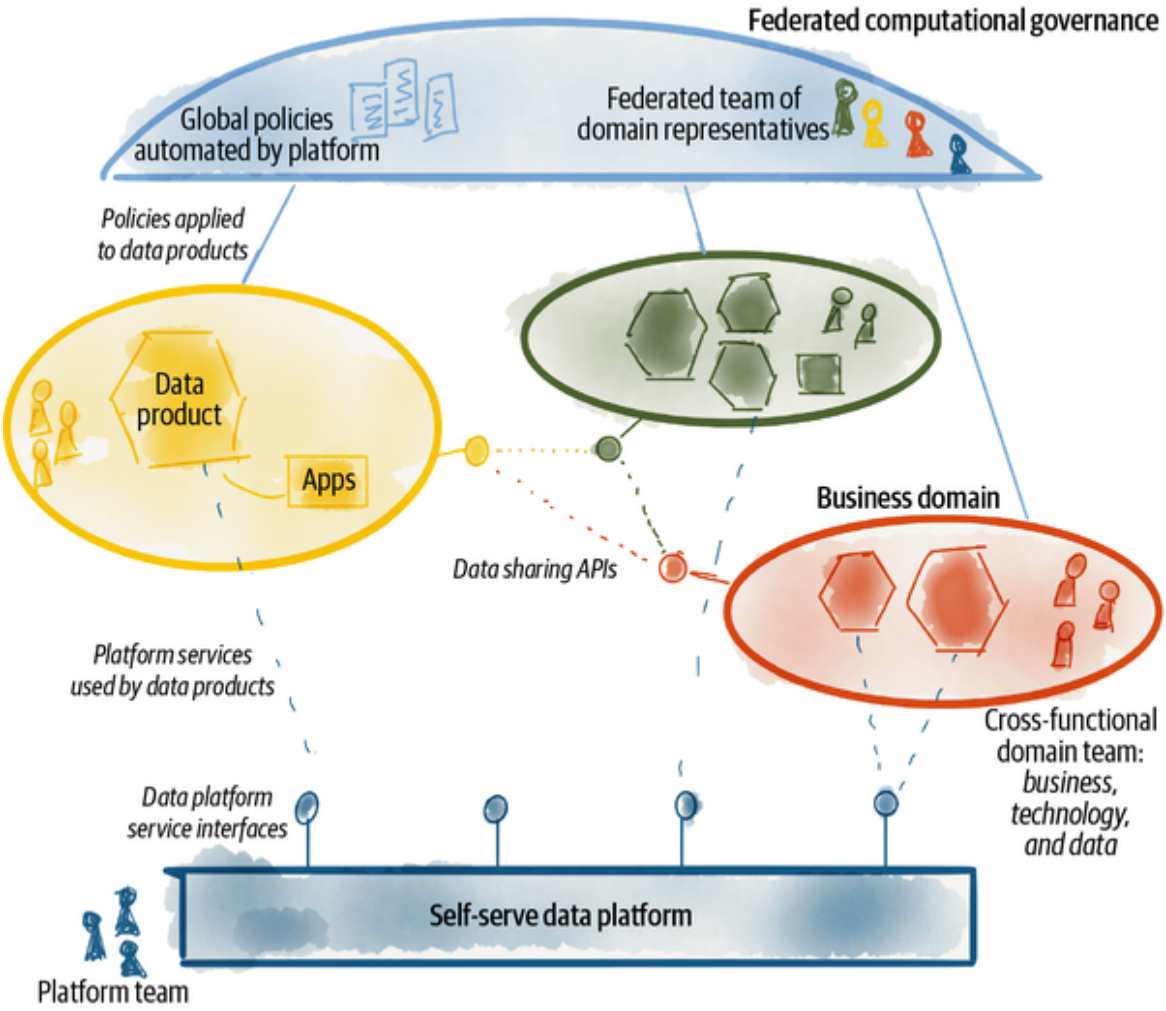

Figure 3-18. Simplified example of a data mesh architecture. Source: From Data Mesh, by Zhamak Dehghani. Copyright © 2022 Zhamak Dehghani. Published by O'Reilly Media, Inc. Used with permission.

You can learn more about data mesh in Dehghani's book *Data Mesh* (O'Reilly).

Other Data Architecture Examples

Data architectures have countless other variations, such as data fabric, data hub, scaled architecture, metadata-first architecture, event-driven architecture, live data stack (Chapter 11), and many more. And new architectures will continue to emerge as practices consolidate and mature, and tooling simplifies and improves. We've focused on a handful of the

most critical data architecture patterns that are extremely well established, evolving rapidly, or both.

As a data engineer, pay attention to how new architectures may help your organization. Stay abreast of new developments by cultivating a high-level awareness of the data engineering ecosystem developments. Be open-minded and don't get emotionally attached to one approach. Once you've identified potential value, deepen your learning and make concrete decisions. When done right, minor tweaks—or major overhauls—in your data architecture can positively impact the business.

Who's Involved with Designing a Data Architecture?

Data architecture isn't designed in a vacuum. Bigger companies may still employ data architects, but those architects will need to be heavily in tune and current with the state of technology and data. Gone are the days of ivory tower data architecture. In the past, architecture was largely orthogonal to engineering. We expect this distinction will disappear as data engineering, and engineering in general, quickly evolves, becoming more agile, with less separation between engineering and architecture.

Ideally, a data engineer will work alongside a dedicated data architect. However, if a company is small or low in its level of data maturity, a data engineer might do double duty as an architect. Because data architecture is an undercurrent of the data engineering lifecycle, a data engineer should understand "good" architecture and the various types of data architecture.

When designing architecture, you'll work alongside business stakeholders to evaluate trade-offs. What are the trade-offs inherent in adopting a cloud data warehouse versus a data lake? What are the trade-offs of various cloud platforms? When might a unified batch/streaming framework (Beam, Flink) be an appropriate choice? Studying these choices in the abstract will prepare you to make concrete, valuable decisions.

Conclusion

You've learned how data architecture fits into the data engineering lifecycle and what makes for "good" data architecture, and you've seen several examples of data architectures. Because architecture is such a key foundation for success, we encourage you to invest the time to study it deeply and understand the trade-offs inherent in any architecture. You will be prepared to map out architecture that corresponds to your organization's unique requirements.

Next up, let's look at some approaches to choosing the right technologies to be used in data architecture and across the data engineering lifecycle.

Additional Resources

- "Separating Utility from Value Add" by Ross Pettit
- "Tactics vs. Strategy: SOA and the Tarpit of Irrelevancy" by Neal Ford
- The Information Management Body of Knowledge website
- "The Modern Data Stack: Past, Present, and Future" by Tristan Handy
- "Potemkin Data Science" by Michael Correll
- "A Comparison of Data Processing Frameworks" by Ludovik Santos
- "Modern CI Is Too Complex and Misdirected" by Gregory Szorc
- "Questioning the Lambda Architecture" by Jay Kreps
- "End-to-End Serverless ETL Orchestration in AWS: A Guide" by Rittika Jindal
- "A Brief Introduction to Two Data Processing Architectures—Lambda and Kappa for Big Data" by Iman Samizadeh
- "How to Beat the Cap Theorem" by Nathan Marz

- "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction" by Jay Kreps
- "Run Your Data Team Like a Product Team" by Emilie Schario and Taylor A. Murphy
- "Data as a Product vs. Data as a Service" by Justin Gage

Martin Fowler articles:

- "EagerReadDerivation"
- "AnemicDomainModel"
- "DomainDrivenDesign"
- "Event Sourcing"
- "ReportingDatabase"
- "BoundedContext"
- "Focusing on Events"
- "Polyglot Persistence"
- "UtilityVsStrategicDichotomy"

Data applications:

- "Introducing Dagster: An Open Source Python Library for Building Data Applications" by Nick Schrock
- "Down with Pipeline Debt: Introducing Great Expectations," by the Great Expectations project
- "Functional Data Engineering: A Modern Paradigm for Batch Data Processing" by Maxime Beauchemin
- "What Is the Open Data Ecosystem and Why It's Here to Stay" by Casber Wang

- “Staying Ahead of Data Debt” by Etai Mizrahi
- “Choosing Open Wisely” by Benoit Dageville et al.
- “The Ultimate Data Observability Checklist” by Molly Vorwerck
- “Moving Beyond Batch vs. Streaming” by David Yaffe
- “Disasters I’ve Seen in a Microservices World” by Joao Alves

Naming conventions:

- “240 Tables and No Documentation?” by Alexey Makhotkin
- “Data Warehouse Architecture: Overview” by Roelant Vos
- “The Design and Implementation of Modern Column-Oriented Database Systems” by Daniel Abadi et al.
- “Column-oriented DBMS” Wikipedia page
- “The Data Dichotomy: Rethinking the Way We Treat Data and Services” by Ben Stopford
- “The Curse of the Data Lake Monster” by Kiran Prakash and Lucy Chambers
- “Software Infrastructure 2.0: A Wishlist” by Erik Bernhardsson
- “Data Team Platform” by GitLab Data
- “Falling Back in Love with Data Pipelines” by Sean Knapp

Build versus buy fallacy:

- “What’s Wrong with MLOps?” by Laszlo Sragner
- “Zachman Framework” Wikipedia page
- “How TOGAF Defines Enterprise Architecture (EA)” by Avancier Limited
- EABOK Draft, edited by Paula Hagan

- “What Is Data Architecture? A Framework for Managing Data” by Thor Olavsrud
- “Google Cloud Architecture Framework” Google Cloud Architecture web page
- Microsoft’s “Azure Architecture Center”
- “Five Principles for Cloud-Native Architecture: What It Is and How to Master It” by Tom Grey
- “Choosing the Right Architecture for Global Data Distribution” Google Cloud Architecture web page
- “Principled Data Engineering, Part I: Architectural Overview” by Hussein Danish
- “A Personal Implementation of Modern Data Architecture: Getting Strava Data into Google Cloud Platform” by Matthew Reeve
- “The Cost of Cloud, a Trillion Dollar Paradox” by Sarah Wang and Martin Casado
- *Data Architecture: A Primer for the Data Scientist* by W.H. Inmon et al. (Academic Press)
- “Enterprise Architecture” Gartner Glossary definition
- EABOK website
- TOGAF framework website
- “Defining Architecture” ISO/IEC/IEEE 42010 web page
- “The Six Principles of Modern Data Architecture” by Joshua Klahr
- “The Rise of the Metadata Lake” by Prukalpa
- “The Top 5 Data Trends for CDOs to Watch Out for in 2021” by Prukalpa
- “Test Data Quality at Scale with Deequ” by Dustin Lange et al.

- "What the Heck Is Data Mesh" by Chris Riccomini
- "Reliable Microservices Data Exchange with the Outbox Pattern" by Gunnar Morling
- "Who Needs an Architect" by Martin Fowler
- "Enterprise Architecture's Role in Building a Data-Driven Organization" by Ashutosh Gupta
- "How to Build a Data Architecture to Drive Innovation—Today and Tomorrow" by Antonio Castro et al.
- "Data Warehouse Architecture" tutorial at Javatpoint
- Snowflake's "What Is Data Warehouse Architecture" web page
- "Three-Tier Architecture" by IBM Education
- "The Building Blocks of a Modern Data Platform" by Prukalpa
- "Data Architecture: Complex vs. Complicated" by Dave Wells
- "Data Fabric Defined" by James Serra
- "Data Fabric Architecture Is Key to Modernizing Data Management and Integration" by Ashutosh Gupta
- "Unified Analytics: Where Batch and Streaming Come Together; SQL and Beyond" Apache Flink Roadmap
- "What Is a Data Lakehouse?" by Ben Lorica et al.
- "Big Data Architectures" Azure documentation
- "Microsoft Azure IoT Reference Architecture" documentation

---

1 Jeff Haden, "Amazon Founder Jeff Bezos: This Is How Successful People Make Such Smart Decisions," *Inc.*, December 3, 2018, *https://oreil.ly/QwIm0*.

2 Martin Fowler, "Who Needs an Architect?" *IEEE Software*, July/August 2003, *https://oreil.ly/wAMmZ*.

3 "The Bezos API Mandate: Amazon's Manifesto for Externalization," Nordic APIs, January 19, 2021, *https://oreil.ly/vIs8m*.

4 Fowler, "Who Needs an Architect?"

5 Ericka Chickowski, "Leaky Buckets: 10 Worst Amazon S3 Breaches," Bitdefender *Business Insights* blog, Jan 24, 2018, *https://oreil.ly/pFEFO*.

6 "FinOps Foundation Soars to 300 Members and Introduces New Partner Tiers for Cloud Service Providers and Vendors," Business Wire, June 17, 2019, *https://oreil.ly/XcwYO*.

7 Martin Fowler, "StranglerFigApplication," June 29, 2004, *https://oreil.ly/PmqxB*.

8 Mike Loukides, "Resume Driven Development," *O'Reilly Radar*, October 13, 2004, *https://oreil.ly/BUHa8*.

9 H.W. Inmon, *Building the Data Warehouse* (Hoboken: Wiley, 2005).

10 Jay Kreps, "Questioning the Lambda Architecture," O'Reilly, July 2, 2014, *https://oreil.ly/wWR3n*.

11 Martin Fowler, "Data Mesh Principles and Logical Architecture," MartinFowler.com, December 3, 2020, *https://oreil.ly/ezWE7*.

12 Zhamak Dehghani, "How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh," MartinFowler.com, May 20, 2019, *https://oreil.ly/SqMe8*.

13 Zhamak Dehghani, "Data Mesh Principles," MartinFowler.com, December 3, 2020, *https://oreil.ly/RymDi*.

Chapter 4. Choosing Technologies Across the Data Engineering Lifecycle

Data engineering nowadays suffers from an embarrassment of riches. We have no shortage of technologies to solve various types of data problems. Data technologies are available as turnkey offerings consumable in almost every way—open source, managed open source, proprietary software, proprietary service, and more. However, it's easy to get caught up in chasing bleeding-edge technology while losing sight of the core purpose of data engineering: designing robust and reliable systems to carry data through the full lifecycle and serve it according to the needs of end users. Just as structural engineers carefully choose technologies and materials to realize an architect's vision for a building, data engineers are tasked with making appropriate technology choices to shepherd data through the lifecycle to serve data applications and users.

Chapter 3 discussed "good" data architecture and why it matters. We now explain how to choose the right technologies to serve this architecture. Data engineers must choose good technologies to make the best possible data product. We feel the criteria to choose a good data technology is simple: does it add value to a data product and the broader business?

A lot of people confuse architecture and tools. Architecture is *strategic*; tools are *tactical*. We sometimes hear, "Our data architecture are tools *X*, *Y*, and *Z*." This is the wrong way to think about architecture. Architecture is the top-level design, roadmap, and blueprint of data systems that satisfy the strategic aims for the business. Architecture is the *what*, *why*, and *when*. Tools are used to make the architecture a reality; tools are the *how*.

We often see teams going "off the rails" and choosing technologies before mapping out an architecture. The reasons vary: shiny object syndrome,

resume-driven development, and a lack of expertise in architecture. In practice, this prioritization of technology often means they cobble together a kind of Dr. Suess fantasy machine rather than a true data architecture. We strongly advise against choosing technology before getting your architecture right. Architecture first, technology second.

This chapter discusses our tactical plan for making technology choices once we have a strategic architecture blueprint. The following are some considerations for choosing data technologies across the data engineering lifecycle:

- Team size and capabilities
- Speed to market
- Interoperability
- Cost optimization and business value
- Today versus the future: immutable versus transitory technologies
- Location (cloud, on prem, hybrid cloud, multicloud)
- Build versus buy
- Monolith versus modular
- Serverless versus servers
- Optimization, performance and the benchmark wars.
- The undercurrents of the data engineering lifecycle

Team Size and Capabilities

The first thing you need to assess is your team's size and its capabilities with technology. Are you on a small team (perhaps a team of one) of people who are expected to wear many hats, or is the team large enough that people work in specialized roles? Will a handful of people be responsible for multiple stages of the data engineering lifecycle, or do people cover

particular niches? Your team's size will influence the types of technologies you adopt.

There is a continuum of simple to complex technologies, and a team's size roughly determines the amount of bandwidth your team can dedicate to complex solutions. We sometimes see small data teams read blog posts about a new cutting-edge technology at a giant tech company and then try to emulate these same extremely complex technologies and practices. We call this *cargo-cult engineering*, and it's generally a big mistake that consumes a lot of valuable time and money, often with little to nothing to show in return. Especially for small teams or teams with weaker technical chops, use as many managed and SaaS tools as possible, and dedicate your limited bandwidth to solving the complex problems that directly add value to the business.

Take an inventory of your team's skills. Do people lean toward low-code tools, or do they favor code-first approaches? Are people strong in certain languages like Java, Python, or Go? Technologies are available to cater to every preference on the low-code to code-heavy spectrum. Again, we suggest sticking with technologies and workflows with which the team is familiar. We've seen data teams invest a lot of time in learning the shiny new data framework, only to never use it in production. Learning new technologies, languages, and tools is a considerable time investment, so make these investments wisely.

Speed to Market

In technology, speed to market wins. This means choosing the right technologies that help you deliver features and data faster while maintaining high-quality standards and security. It also means working in a tight feedback loop of launching, learning, iterating, and making improvements.

Perfect is the enemy of good. Some data teams will deliberate on technology choices for months or years without reaching any decisions. Slow decisions and output are the kiss of death to data teams. We've seen

more than a few data teams dissolve for moving too slow and failing to deliver the value they were hired to produce.

Deliver value early and often. As we've mentioned, use what works. Your team members will likely get better leverage with tools they already know. Avoid undifferentiated heavy lifting that engages your team in unnecessarily complex work that adds little to no value. Choose tools that help you move quickly, reliably, safely, and securely.

Interoperability

Rarely will you use only one technology or system. When choosing a technology or system, you'll need to ensure that it interacts and operates with other technologies. *Interoperability* describes how various technologies or systems connect, exchange information, and interact.

Let's say you're evaluating two technologies, A and B. How easily does technology A integrate with technology B when thinking about interoperability? This is often a spectrum of difficulty, ranging from seamless to time-intensive. Is seamless integration already baked into each product, making setup a breeze? Or do you need to do a lot of manual configuration to integrate these technologies?

Often, vendors and open source projects will target specific platforms and systems to interoperate. Most data ingestion and visualization tools have built-in integrations with popular data warehouses and data lakes. Furthermore, popular data-ingestion tools will integrate with common APIs and services, such as CRMs, accounting software, and more.

Sometimes standards are in place for interoperability. Almost all databases allow connections via Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC), meaning that you can easily connect to a database by using these standards. In other cases, interoperability occurs in the absence of standards. Representational state transfer (REST) is not truly a standard for APIs; every REST API has its quirks. In these cases, it's up

to the vendor or open source software (OSS) project to ensure smooth integration with other technologies and systems.

Always be aware of how simple it will be to connect your various technologies across the data engineering lifecycle. As mentioned in other chapters, we suggest designing for modularity and giving yourself the ability to easily swap out technologies as new practices and alternatives become available.

Cost Optimization and Business Value

In a perfect world, you'd get to experiment with all the latest, coolest technologies without considering cost, time investment, or value added to the business. In reality, budgets and time are finite, and the cost is a major constraint for choosing the right data architectures and technologies. Your organization expects a positive ROI from your data projects, so you must understand the basic costs you can control. Technology is a major cost driver, so your technology choices and management strategies will significantly impact your budget. We look at costs through three main lenses: total cost of ownership, opportunity cost, and FinOps.

Total Cost of Ownership

Total cost of ownership (TCO) is the total estimated cost of an initiative, including the direct and indirect costs of products and services utilized. *Direct costs* can be directly attributed to an initiative. Examples are the salaries of a team working on the initiative or the AWS bill for all services consumed. *Indirect costs*, also known as *overhead*, are independent of the initiative and must be paid regardless of where they're attributed.

Apart from direct and indirect costs, *how something is purchased* impacts the way costs are accounted for. Expenses fall into two big groups: capital expenses and operational expenses.

Capital expenses, also known as *capex*, require an up-front investment. Payment is required *today*. Before the cloud existed, companies would

typically purchase hardware and software up front through large acquisition contracts. In addition, significant investments were required to host hardware in server rooms, data centers, and colocation facilities. These up-front investments—commonly hundreds of thousands to millions of dollars or more—would be treated as assets and slowly depreciate over time. From a budget perspective, capital was required to fund the entire purchase. This is capex, a significant capital outlay with a long-term plan to achieve a positive ROI on the effort and expense put forth.

Operational expenses, also known as *opex*, are the opposite of capex in certain respects. Opex is gradual and spread out over time. Whereas capex is long-term focused, opex is short-term. Opex can be pay-as-you-go or similar and allows a lot of flexibility. Opex is closer to a direct cost, making it easier to attribute to a data project.

Until recently, opex wasn't an option for large data projects. Data systems often required multimillion-dollar contracts. This has changed with the advent of the cloud, as data platform services allow engineers to pay on a consumption-based model. In general, opex allows for a far greater ability for engineering teams to choose their software and hardware. Cloud-based services let data engineers iterate quickly with various software and technology configurations, often inexpensively.

Data engineers need to be pragmatic about flexibility. The data landscape is changing too quickly to invest in long-term hardware that inevitably goes stale, can't easily scale, and potentially hampers a data engineer's flexibility to try new things. Given the upside for flexibility and low initial costs, we urge data engineers to take an opex-first approach centered on the cloud and flexible, pay-as-you-go technologies.

Total Opportunity Cost of Ownership

Any choice inherently excludes other possibilities. *Total opportunity cost of ownership* (TOCO) is the cost of lost opportunities that we incur in choosing a technology, an architecture, or a process.[1] Note that ownership in this setting doesn't require long-term purchases of hardware or licenses.

Even in a cloud environment, we effectively own a technology, a stack, or a pipeline once it becomes a core part of our production data processes and is difficult to move away from. Data engineers often fail to evaluate TOCO when undertaking a new project; in our opinion, this is a massive blind spot.

If you choose data stack A, you've chosen the benefits of data stack A over all other options, effectively excluding data stacks B, C, and D. You're committed to data stack A and everything it entails—the team to support it, training, setup, and maintenance. What happens if data stack A was a poor choice? What happens when data stack A becomes obsolete? Can you still move to data stack B?

How quickly and cheaply can you move to something newer and better? This is a critical question in the data space, where new technologies and products seem to appear at an ever-faster rate. Does the expertise you've built up on data stack A translate to the next wave? Or are you able to swap out components of data stack A and buy yourself some time and options?

The first step to minimizing opportunity cost is evaluating it with eyes wide open. We've seen countless data teams get stuck with technologies that seemed good at the time and are either not flexible for future growth or simply obsolete. Inflexible data technologies are a lot like bear traps. They're easy to get into and extremely painful to escape.

FinOps

We already touched on FinOps in "Principle 9: Embrace FinOps". As we've discussed, typical cloud spending is inherently opex: companies pay for services to run critical data processes rather than making up-front purchases and clawing back value over time. The goal of FinOps is to fully operationalize financial accountability and business value by applying the DevOps-like practices of monitoring and dynamically adjusting systems.

In this chapter, we want to emphasize one thing about FinOps that is well embodied in this quote:

> *If it seems that FinOps is about saving money, then think again. FinOps is about making money. Cloud spend can drive more revenue, signal customer base growth, enable more product and feature release velocity, or even help shut down a data center.*[2]

In our setting of data engineering, the ability to iterate quickly and scale dynamically is invaluable for creating business value. This is one the major motivations for shifting data workloads to the cloud.

Today Versus the Future: Immutable Versus Transitory Technologies

In an exciting domain like data engineering, it's all too easy to focus on a rapidly evolving future while ignoring the concrete needs of the present. The intention to build a better future is noble but often leads to overarchitecting and overengineering. Tooling chosen for the future may be stale and out-of-date when this future arrives; the future frequently looks little like what we envisioned years before.

As many life coaches would tell you, focus on the present. You should choose the best technology for the moment and near future, but in a way that supports future unknowns and evolution. Ask yourself: where are you today, and what are your goals for the future? Your answers to these questions should inform your decisions about your architecture and thus the technologies used within that architecture. This is done by understanding what is likely to change and what tends to stay the same.

We have two classes of tools to consider: immutable and transitory. *Immutable technologies* might be components that underpin the cloud or languages and paradigms that have stood the test of time. In the cloud, examples of immutable technologies are object storage, networking, servers, and security. Object storage such as Amazon S3 and Azure Blob Storage will be around from today until the end of the decade, and probably much longer. Storing your data in object storage is a wise choice. Object storage continues to improve in various ways and constantly offers new

options, but your data will be safe and usable in object storage regardless of the rapid evolution of technology.

For languages, SQL and bash have been around for many decades, and we don't see them disappearing anytime soon. Immutable technologies benefit from the Lindy effect: the longer a technology has been established, the longer it will be used. Think of the power grid, relational databases, the C programming language, or the x86 processor architecture. We suggest applying the Lindy effect as a litmus test to determine whether a technology is potentially immutable.

Transitory technologies are those that come and go. The typical trajectory begins with a lot of hype, followed by meteoric growth in popularity, then a slow descent into obscurity. The JavaScript frontend landscape is a classic example. How many JavaScript frontend frameworks have come and gone between 2010 and 2020? Backbone.js, Ember.js, and Knockout were popular in the early 2010s, AngularJS in the mid-2010s, and React and Vue.js have massive mindshare today. What's the popular frontend framework three years from now? Who knows.

New well-funded entrants and open source projects arrive on the data front every day. Every vendor will say their product will change the industry and "make the world a better place". Most of these companies and projects don't get long-term traction and fade slowly into obscurity. Top VCs are making big-money bets, knowing that most of their data-tooling investments will fail. How can you possibly know which technologies to invest in for your data architecture? It's hard. Just consider the number of technologies in Matt Turck's (in)famous depictions of the ML, AI, and data (MAD) landscape that we introduced in Chapter 1 (Figure 4-1).

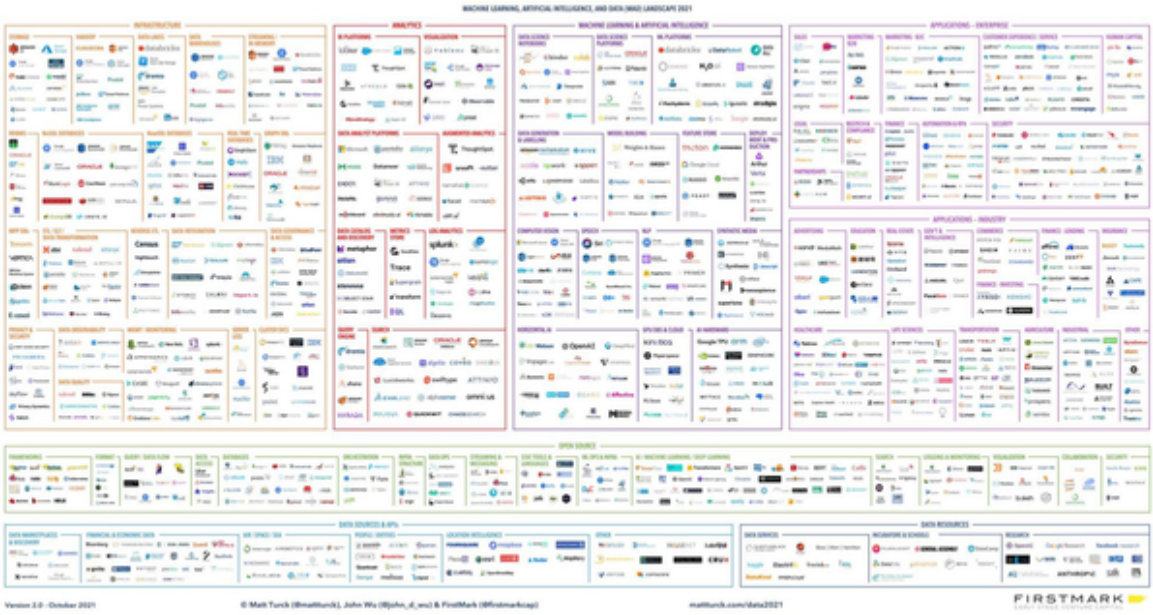

Figure 4-1. Matt Turck's 2021 MAD data landscape

Even relatively successful technologies often fade into obscurity quickly, after a few years of rapid adoption, a victim of their success. For instance, in the early 2010s, Hive was met with rapid uptake because it allowed both analysts and engineers to query massive datasets without coding complex MapReduce jobs by hand. Inspired by the success of Hive but wishing to improve on its shortcomings, engineers developed Presto and other technologies. Hive now appears primarily in legacy deployments.

Our Advice

Given the rapid pace of tooling and best-practice changes, we suggest evaluating tools every two years (Figure 4-2). Whenever possible, find the immutable technologies along the data engineering lifecycle, and use those as your base. Build transitory tools around the immutables.

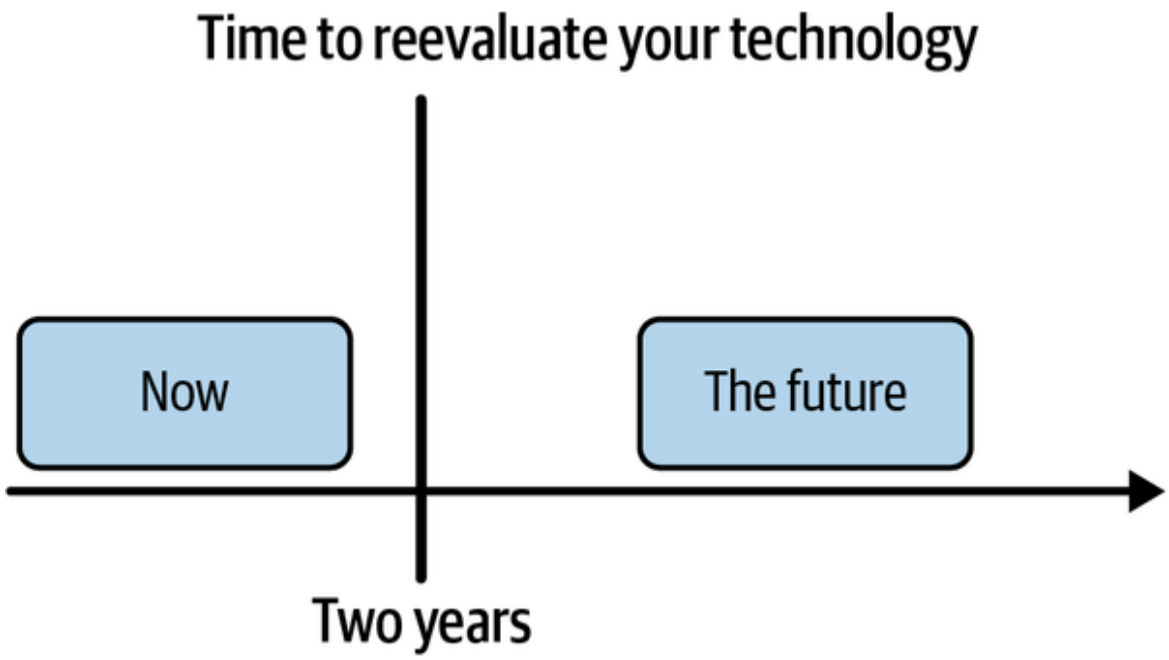

Figure 4-2. Use a two-year time horizon to reevaluate your technology choices

Given the reasonable probability of failure for many data technologies, you need to consider how easy it is to transition from a chosen technology. What are the barriers to leaving? As mentioned previously in our discussion about opportunity cost, avoid "bear traps." Go into a new technology with eyes wide open, knowing the project might get abandoned, the company may not be viable, or the technology simply isn't a good fit any longer.

Location

Companies now have numerous options when deciding where to run their technology stacks. A slow shift toward the cloud culminates in a veritable stampede of companies spinning up workloads on AWS, Azure, and Google Cloud Platform (GCP). In the last decade, many CTOs have come to view their decisions around technology hosting as having existential significance for their organizations. If they move too slowly, they risk being left behind by their more agile competition; on the other hand, a poorly planned cloud migration could lead to technological failure and catastrophic costs.

Let's look at the principal places to run your technology stack: on premises, the cloud, hybrid cloud, and multicloud.

On Premises

While new startups are increasingly born in the cloud, on-premises systems are still the default for established companies. Essentially, these companies own their hardware, which may live in data centers they own or in leased colocation space. In either case, companies are operationally responsible for their hardware and the software that runs on it. If hardware fails, they have to repair or replace it. They also have to manage upgrade cycles every few years as new, updated hardware is released and older hardware ages and becomes less reliable. They must ensure that they have enough hardware to handle peaks; for an online retailer, this means hosting enough capacity to handle the load spikes of Black Friday. For data engineers in charge of on-premises systems, this means buying large-enough systems to allow good performance for peak load and large jobs without overbuying and overspending.

On the one hand, established companies have established operational practices that have served them well. Suppose a company that relies on information technology has been in business for some time. This means it has managed to juggle the cost and personnel requirements of running its hardware, managing software environments, deploying code from dev teams, and running databases and big data systems.

On the other hand, established companies see their younger, more agile competition scaling rapidly and taking advantage of cloud-managed services. They also see established competitors making forays into the cloud, allowing them to temporarily scale up enormous computing power for massive data jobs or the Black Friday shopping spike.

Companies in competitive sectors generally don't have the option to stand still. Competition is fierce, and there's always the threat of being "disrupted" by more agile competition, backed by a large pile of venture capital dollars. Every company must keep its existing systems running efficiently while deciding what moves to make next. This could involve adopting newer DevOps practices, such as containers, Kubernetes, microservices, and continuous deployment while keeping their hardware

running on premises. It could involve a complete migration to the cloud, as discussed next.

Cloud

The cloud flips the on-premises model on its head. Instead of purchasing hardware, you simply rent hardware and managed services from a cloud provider (such as AWS, Azure, or Google Cloud). These resources can often be reserved on an extremely short-term basis; VMs spin up in less than a minute, and subsequent usage is billed in per-second increments. This allows cloud users to dynamically scale resources that were inconceivable with on-premises servers.

In a cloud environment, engineers can quickly launch projects and experiment without worrying about long lead time hardware planning. They can begin running servers as soon as their code is ready to deploy. This makes the cloud model extremely appealing to startups that are tight on budget and time.

The early cloud era was dominated by infrastructure as a service (IaaS) offerings—products such as VMs and virtual disks that are essentially rented slices of hardware. Slowly, we've seen a shift toward platform as a service (PaaS), while SaaS products continue to grow at a rapid clip.

PaaS includes IaaS products but adds more sophisticated managed services to support applications. Examples are managed databases such as Amazon Relational Database Service (RDS) and Google Cloud SQL, managed streaming platforms such as Amazon Kinesis and Simple Queue Service (SQS), and managed Kubernetes such as Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS). PaaS services allow engineers to ignore the operational details of managing individual machines and deploying frameworks across distributed systems. They provide turnkey access to complex, autoscaling systems with minimal operational overhead.

SaaS offerings move one additional step up the ladder of abstraction. SaaS typically provides a fully functioning enterprise software platform with little operational management. Examples of SaaS include Salesforce,

Google Workspace, Microsoft 365, Zoom, and Fivetran. Both the major public clouds and third parties offer SaaS platforms. SaaS covers a whole spectrum of enterprise domains, including video conferencing, data management, ad tech, office applications, and CRM systems.

This chapter also discusses serverless, increasingly important in PaaS and SaaS offerings. Serverless products generally offer automated scaling from zero to extremely high usage rates. They are billed on a pay-as-you-go basis and allow engineers to operate without operational awareness of underlying servers. Many people quibble with the term *serverless*; after all, the code must run somewhere. In practice, serverless usually means *many invisible servers*.

Cloud services have become increasingly appealing to established businesses with existing data centers and IT infrastructure. Dynamic, seamless scaling is extremely valuable to businesses that deal with seasonality (e.g., retail businesses coping with Black Friday load) and web traffic load spikes. The advent of COVID-19 in 2020 was a major driver of cloud adoption, as companies recognized the value of rapidly scaling up data processes to gain insights in a highly uncertain business climate; businesses also had to cope with substantially increased load due to a spike in online shopping, web app usage, and remote work.

Before we discuss the nuances of choosing technologies in the cloud, let's first discuss why migration to the cloud requires a dramatic shift in thinking, specifically on the pricing front; this is closely related to FinOps, introduced in "FinOps". Enterprises that migrate to the cloud often make major deployment errors by not appropriately adapting their practices to the cloud pricing model.