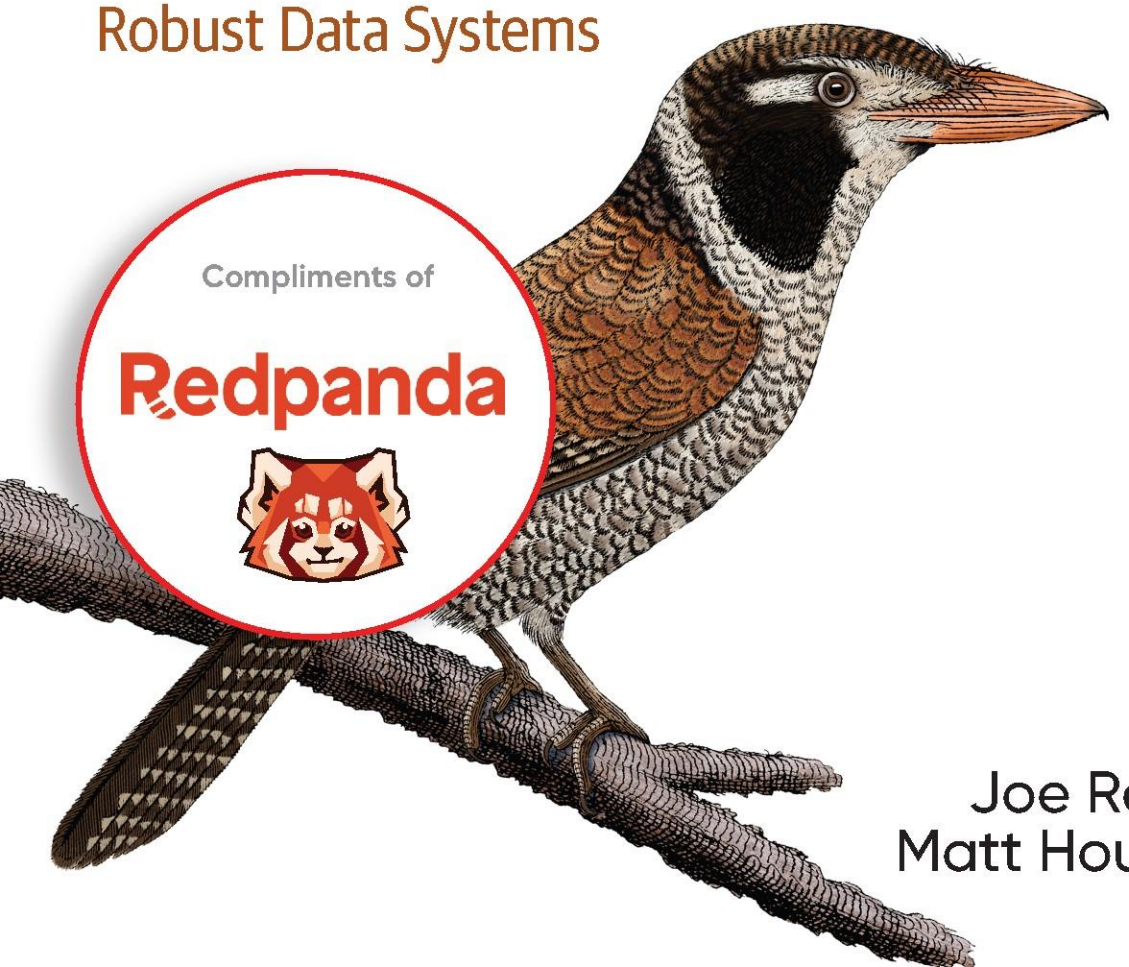
O'REILLY®

Fundamentals of Data Engineering

Plan and Build Robust Data Systems



Compliments of

Redpanda

Joe Reis &
Matt Housley

Fundamentals of Data Engineering

Plan and Build Robust Data Systems

Joe Reis and Matt Housley

Beijing . Boston . Farnham . Sebastopol . Tokyo

O'REILLY®

Table of Contents

Part II. The Data Engineering Lifecycle in Depth

PART I Foundation and Building Block

CHAPTE
R 1

Data Engineering Described

If you work in data or software, you may have noticed data engineering emerging from the shadows and now sharing the stage with data science. Data engineering is one of the hottest fields in data and technology, and for a good reason. It builds the foundation for data science and analytics in production. This chapter explores what data engineering is, how the field was born and its evolution, the skills of data engineers, and with whom they work.

What Is Data Engineering?

Despite the current popularity of data engineering, there's a lot of confusion about what data engineering means and what data engineers do. Data engineering has existed in some form since companies started doing things with data—such as predictive analysis, descriptive analytics, and reports—and came into sharp focus alongside the rise of data science in the 2010s. For the purpose of this book, it's critical to define what *data engineering* and *data engineer* mean.

First, let's look at the landscape of how data engineering is described and develop some terminology we can use throughout this book. Endless definitions of *data engineering* exist. In early 2022, a Google exact-match search for "what is data engineering?" returns over 91,000 unique results. Before we give our definition, here are a few examples of how some experts in the field define data engineering:

> Data engineering is a set of operations aimed at creating interfaces and mechanisms for the flow and access of information. It takes dedicated specialists—data engineers—to maintain data so that it remains available and usable by others. In short, data engineers set up and operate the organization's data infrastructure, preparing it for further analysis by data analysts and scientists.
>
> —From "Data Engineering and Its Main Concepts" by AlexSoft[1]

> The first type of data engineering is SQL-focused. The work and primary storage of the data is in relational databases. All of the data processing is done with SQL or a SQL-based language. Sometimes, this data processing is done with an ETL tool.[2] The second type of data engineering is Big Data–focused. The work and primary storage of the data is in Big Data technologies like Hadoop, Cassandra, and HBase. All of the data processing is done in Big Data frameworks like MapReduce, Spark, and Flink. While SQL is used, the primary processing is done with programming languages like Java, Scala, and Python.
>
> —Jesse Anderson[3]

> In relation to previously existing roles, the data engineering field could be thought of as a superset of business intelligence and data warehousing that brings more elements from software engineering. This discipline also integrates specialization around the operation of so-called "big data" distributed systems, along with concepts around the extended Hadoop ecosystem, stream processing, and in computation at scale.
>
> —Maxime Beauchemin[4]

> Data engineering is all about the movement, manipulation, and management of data.
>
> —Lewis Gavin[5]

Wow! It's entirely understandable if you've been confused about data engineering. That's only a handful of definitions, and they contain an enormous range of opinions about the meaning of *data engineering*.

Data Engineering Defined

[1] "Data Engineering and Its Main Concepts," AlexSoft, last updated August 26, 2021, *https://oreil.ly/e94py*.

[2] *ETL* stands for *extract, transform, load*, a common pattern we cover in the book.

[3] Jesse Anderson, "The Two Types of Data Engineering," June 27, 2018, *https://oreil.ly/dxDt6*.

[4] Maxime Beauchemin, "The Rise of the Data Engineer," January 20, 2017, *https://oreil.ly/kNDmd*.

[5] Lewis Gavin, *What Is Data Engineering?* (Sebastapol, CA: O'Reilly, 2020), *https://oreil.ly/ELxLi*.

When we unpack the common threads of how various people define data engineering, an obvious pattern emerges: a data engineer gets data, stores it, and prepares it for consumption by data scientists, analysts, and others. We define *data engineering* and *data engineer* as follows:

> *Data engineering* is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning. Data engineering is the intersection of security, data management, DataOps, data architecture, orchestration, and software engineering. A *data engineer* manages the data engineering lifecycle, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning.

4

The Data Engineering Lifecycle

It is all too easy to fixate on technology and miss the bigger picture myopically. This book centers around a big idea called the *data engineering lifecycle* (Figure 1-1), which we believe gives data engineers the holistic context to view their role.

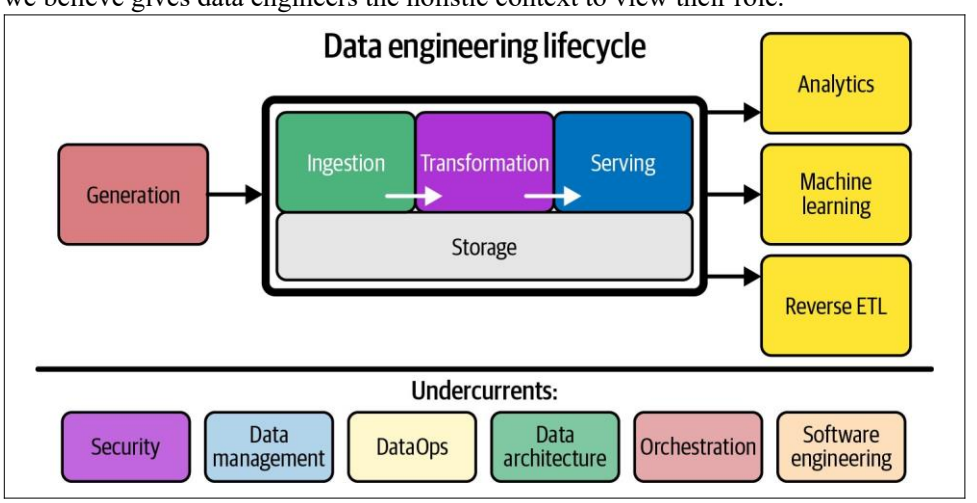


Figure 1-1. The data engineering lifecycle

The data engineering lifecycle shifts the conversation away from technology and toward the data itself and the end goals that it must serve. The stages of the data engineering lifecycle are as follows:

- Generation
- Storage
- Ingestion
- Transformation
- Serving

The data engineering lifecycle also has a notion of *undercurrents*—critical ideas across the entire lifecycle. These include security, data management, DataOps, data architecture, orchestration, and software engineering. We cover the data engineering lifecycle and its undercurrents more extensively in Chapter 2. Still, we introduce it here because it is essential to our definition of data engineering and the discussion that follows in this chapter.

Now that you have a working definition of data engineering and an introduction to its lifecycle, let's take a step back and look at a bit of history.

Evolution of the Data Engineer

> History doesn't repeat itself, but it rhymes.
>
> —A famous adage often attributed to Mark Twain

Understanding data engineering today and tomorrow requires a context of how the field evolved. This section is not a history lesson, but looking at the past is invaluable in understanding where we are today and where things are going. A common theme constantly reappears: what's old is new again.

The early days: 1980 to 2000, from data warehousing to the web

The birth of the data engineer arguably has its roots in data warehousing, dating as far back as the 1970s, with the *business data warehouse* taking shape in the 1980s and Bill Inmon officially coining the term *data warehouse* in 1989. After engineers at IBM developed the relational database and Structured Query Language (SQL), Oracle popularized the technology. As nascent data systems grew, businesses needed dedicated tools and data pipelines for reporting and business intelligence (BI). To help people correctly model their business logic in the data warehouse, Ralph Kimball and Inmon developed their respective eponymous data-modeling techniques and approaches, which are still widely used today.

Data warehousing ushered in the first age of scalable analytics, with new massively parallel processing (MPP) databases that use multiple processors to crunch large amounts of data coming on the market and supporting unprecedented volumes of data. Roles such as BI engineer, ETL developer, and data warehouse engineer addressed the various needs of the data warehouse. Data warehouse and BI engineering were a precursor to today's data engineering and still play a central role in the discipline.

The internet went mainstream around the mid-1990s, creating a whole new generation of web-first companies such as AOL, Yahoo, and Amazon. The dot-com boom spawned a ton of activity in web applications and the backend systems to support them—servers, databases, and storage. Much of the infrastructure was expensive, monolithic, and heavily licensed. The vendors selling these backend systems likely didn't foresee the sheer scale of the data that web applications would produce.

The early 2000s: The birth of contemporary data engineering

Fast-forward to the early 2000s, when the dot-com boom of the late '90s went bust, leaving behind a tiny cluster of survivors. Some of these companies, such as Yahoo, Google, and Amazon, would grow into powerhouse tech companies. Initially, these companies continued to rely on the traditional monolithic, relational databases and data warehouses of the 1990s, pushing these systems to the limit. As these systems buckled,

updated approaches were needed to handle data growth. The new generation of the systems must be cost-effective, scalable, available, and reliable.

Coinciding with the explosion of data, commodity hardware—such as servers, RAM, disks, and flash drives—also became cheap and ubiquitous. Several innovations allowed distributed computation and storage on massive computing clusters at a vast scale. These innovations started decentralizing and breaking apart traditionally monolithic services. The "big data" era had begun.

The *Oxford English Dictionary* defines big data as "extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions." Another famous and succinct description of big data is the three *V*s of data: velocity, variety, and volume.

In 2003, Google published a paper on the Google File System, and shortly after that, in 2004, a paper on MapReduce, an ultra-scalable data-processing paradigm. In truth, big data has earlier antecedents in MPP data warehouses and data management for experimental physics projects, but Google's publications constituted a "big bang" for data technologies and the cultural roots of data engineering as we know it today. You'll learn more about MPP systems and MapReduce in Chapters 3 and 8, respectively.

The Google papers inspired engineers at Yahoo to develop and later open source Apache Hadoop in 2006.[6] It's hard to overstate the impact of Hadoop. Software engineers interested in large-scale data problems were drawn to the possibilities of this new open source technology ecosystem. As companies of all sizes and types saw their data grow into many terabytes and even petabytes, the era of the big data engineer was born.

Around the same time, Amazon had to keep up with its own exploding data needs and created elastic computing environments (Amazon Elastic Compute Cloud, or EC2), infinitely scalable storage systems (Amazon Simple Storage Service, or S3), highly scalable NoSQL databases (Amazon DynamoDB), and many other core data building blocks.[7] Amazon elected to offer these services for internal and external consumption through *Amazon Web Services* (AWS), becoming the first popular public cloud. AWS created an ultra-flexible pay-as-you-go resource marketplace by virtualizing and reselling vast pools of commodity hardware. Instead of purchasing hardware for a data center, developers could simply rent compute and storage from AWS.

[6] Cade Metz, "How Yahoo Spawned Hadoop, the Future of Big Data," *Wired*, October 18, 2011, *https://oreil.ly/iaD9G*.

[7] Ron Miller, "How AWS Came to Be," *TechCrunch*, July 2, 2016, *https://oreil.ly/VJehv*.

As AWS became a highly profitable growth engine for Amazon, other public clouds would soon follow, such as Google Cloud, Microsoft Azure, and DigitalOcean. The public cloud is arguably one of the most significant innovations of the 21st century and spawned a revolution in the way software and data applications are developed and deployed.

The early big data tools and public cloud laid the foundation for today's data ecosystem. The modern data landscape—and data engineering as we know it now—would not exist without these innovations.

The 2000s and 2010s: Big data engineering

Open source big data tools in the Hadoop ecosystem rapidly matured and spread from Silicon Valley to tech-savvy companies worldwide. For the first time, any business had access to the same bleeding-edge data tools used by the top tech companies. Another revolution occurred with the transition from batch computing to event streaming, ushering in a new era of big "real-time" data. You'll learn about batch and event streaming throughout this book.

Engineers could choose the latest and greatest—Hadoop, Apache Pig, Apache Hive, Dremel, Apache HBase, Apache Storm, Apache Cassandra, Apache Spark, Presto, and numerous other new technologies that came on the scene. Traditional enterpriseoriented and GUI-based data tools suddenly felt outmoded, and code-first engineering was in vogue with the ascendance of MapReduce. We (the authors) were around during this time, and it felt like old dogmas died a sudden death upon the altar of big data.

The explosion of data tools in the late 2000s and 2010s ushered in the *big data engineer*. To effectively use these tools and techniques—namely, the Hadoop ecosystem including Hadoop, YARN, Hadoop Distributed File System (HDFS), and MapReduce—big data engineers had to be proficient in software development and low-level infrastructure hacking, but with a shifted emphasis. Big data engineers typically maintained massive clusters of commodity hardware to deliver data at scale. While they might occasionally submit pull requests to Hadoop core code, they shifted their focus from core technology development to data delivery.

Big data quickly became a victim of its own success. As a buzzword, *big data* gained popularity during the early 2000s through the mid-2010s. Big data captured the imagination of companies trying to make sense of the ever-growing volumes of data and the endless barrage of shameless marketing from companies selling big data tools and services. Because of the immense hype, it was common to see companies using big data tools for small data problems, sometimes standing up a Hadoop cluster to process just a few gigabytes. It seemed like everyone wanted in on the big data action.

Dan Ariely tweeted, "Big data is like teenage sex: everyone talks about it, nobody really knows how to do it, everyone thinks everyone else is doing it, so everyone claims they are doing it."

Figure 1-2 shows a snapshot of Google Trends for the search term "big data" to get an idea of the rise and fall of big data.

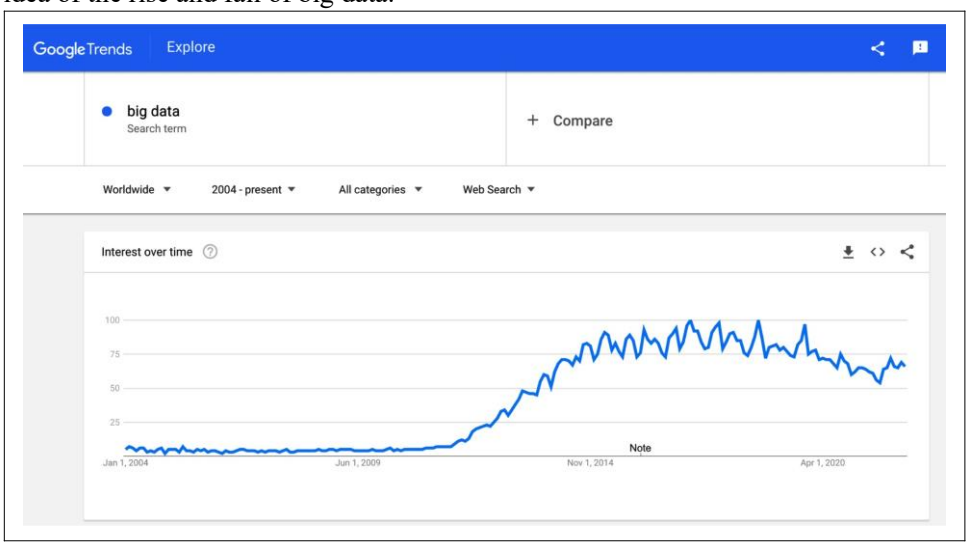


Figure 1-2. Google Trends for "big data" (March 2022)

Despite the term's popularity, big data has lost steam. What happened? One word: simplification. Despite the power and sophistication of open source big data tools, managing them was a lot of work and required constant attention. Often, companies employed entire teams of big data engineers, costing millions of dollars a year, to babysit these platforms. Big data engineers often spent excessive time maintaining complicated tooling and arguably not as much time delivering the business's insights and value.

Open source developers, clouds, and third parties started looking for ways to abstract, simplify, and make big data available without the high administrative overhead and cost of managing their clusters, and installing, configuring, and upgrading their open source code. The term *big data* is essentially a relic to describe a particular time and approach to handling large amounts of data.

Today, data is moving faster than ever and growing ever larger, but big data processing has become so accessible that it no longer merits a separate term; every company aims to solve its data problems, regardless of actual data size. Big data engineers are now simply *data engineers*.

The 2020s: Engineering for the data lifecycle

At the time of this writing, the data engineering role is evolving rapidly. We expect this evolution to continue at a rapid clip for the foreseeable future. Whereas data engineers historically tended to the low-level details of monolithic frameworks such as Hadoop, Spark, or Informatica, the trend is moving toward decentralized, modularized, managed, and highly abstracted tools.

Indeed, data tools have proliferated at an astonishing rate (see Figure 1-3). Popular trends in the early 2020s include the *modern data stack*, representing a collection of off-the-shelf open source and third-party products assembled to make analysts' lives easier. At the same time, data sources and data formats are growing both in variety and size. Data engineering is increasingly a discipline of interoperation, and connecting various technologies like LEGO bricks, to serve ultimate business goals.

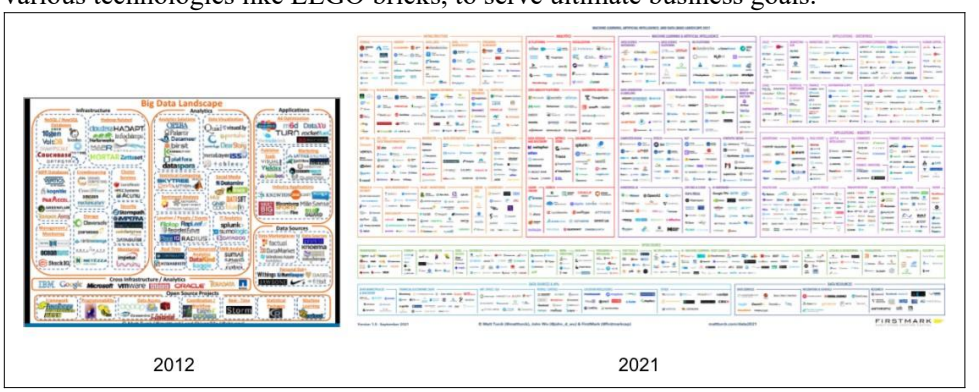


Figure 1-3. Matt Turck's Data Landscape in 2012 versus 2021

The data engineer we discuss in this book can be described more precisely as a *data lifecycle engineer*. With greater abstraction and simplification, a data lifecycle engineer is no longer encumbered by the gory details of yesterday's big data frameworks. While data engineers maintain skills in low-level data programming and use these as required, they increasingly find their role focused on things higher in the value chain: security, data management, DataOps, data architecture, orchestration, and general data lifecycle management.[8]

As tools and workflows simplify, we've seen a noticeable shift in the attitudes of data engineers. Instead of focusing on who has the "biggest data," open source projects and services are increasingly concerned with managing and governing data, making it easier to use and discover, and improving its quality. Data engineers are now

[8] *DataOps* is an abbreviation for *data operations*. We cover this topic in Chapter 2. For more information, read the DataOps Manifesto.

conversant in acronyms such as *CCPA* and *GDPR*;[9] as they engineer pipelines, they concern themselves with privacy, anonymization, data garbage collection, and compliance with regulations.

What's old is new again. While "enterprisey" stuff like data management (including data quality and governance) was common for large enterprises in the pre-big-data era, it wasn't widely adopted in smaller companies. Now that many of the challenging problems of yesterday's data systems are solved, neatly productized, and packaged, technologists and entrepreneurs have shifted focus back to the "enterprisey" stuff, but with an emphasis on decentralization and agility, which contrasts with the traditional enterprise command-and-control approach.

We view the present as a golden age of data lifecycle management. Data engineers managing the data engineering lifecycle have better tools and techniques than ever before. We discuss the data engineering lifecycle and its undercurrents in greater detail in the next chapter.

Data Engineering and Data Science

Where does data engineering fit in with data science? There's some debate, with some arguing data engineering is a subdiscipline of data science. We believe data engineering is *separate* from data science and analytics. They complement each other, but they are distinctly different. Data engineering sits upstream from data science (Figure 1-4), meaning data engineers provide the inputs used by data scientists (downstream from data engineering), who convert these inputs into something useful.

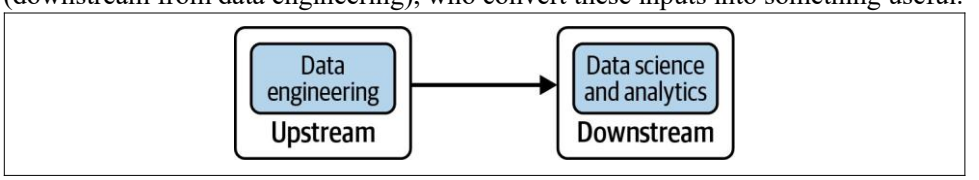


Figure 1-4. Data engineering sits upstream from data science

Consider the Data Science Hierarchy of Needs (Figure 1-5). In 2017, Monica Rogati published this hierarchy in an article that showed where AI and machine learning (ML) sat in proximity to more "mundane" areas such as data movement/storage, collection, and infrastructure.

[9] These acronyms stand for *California Consumer Privacy Act* and *General Data Protection Regulation*, respectively.

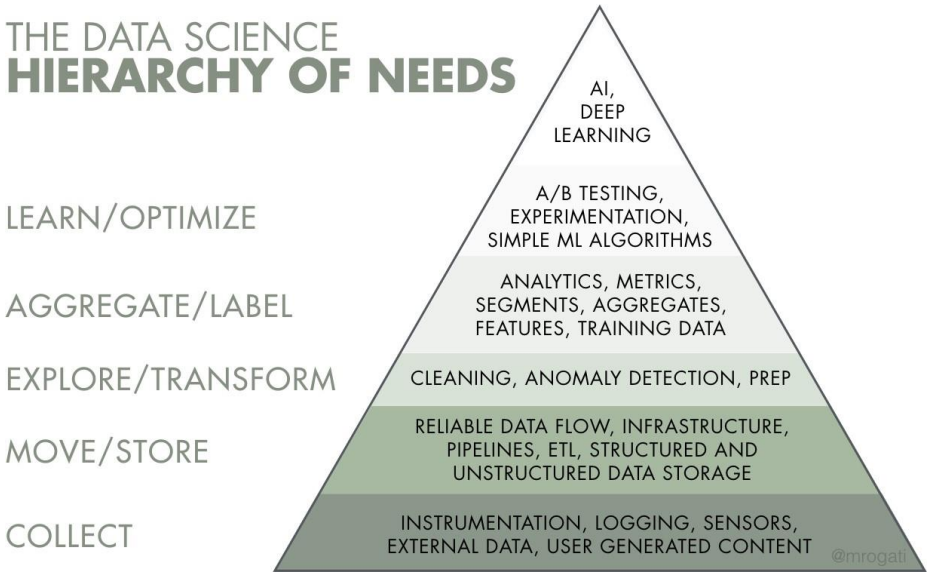


Figure 1-5. The Data Science Hierarchy of Needs

Although many data scientists are eager to build and tune ML models, the reality is an estimated 70% to 80% of their time is spent toiling in the bottom three parts of the hierarchy—gathering data, cleaning data, processing data—and only a tiny slice of their time on analysis and ML. Rogati argues that companies need to build a solid data foundation (the bottom three levels of the hierarchy) before tackling areas such as AI and ML.

Data scientists aren't typically trained to engineer production-grade data systems, and they end up doing this work haphazardly because they lack the support and resources of a data engineer. In an ideal world, data scientists should spend more than 90% of their time focused on the top layers of the pyramid: analytics, experimentation, and ML. When data engineers focus on these bottom parts of the hierarchy, they build a solid foundation for data scientists to succeed.

With data science driving advanced analytics and ML, data engineering straddles the divide between getting data and getting value from data (see Figure 1-6). We believe data engineering is of equal importance and visibility to data science, with data engineers playing a vital role in making data science successful in production.

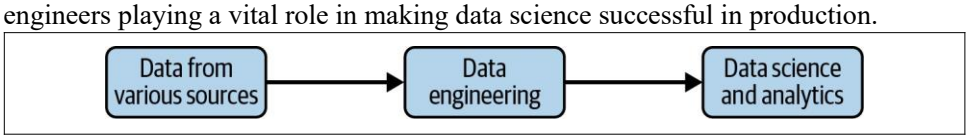


Figure 1-6. A data engineer gets data and provides value from the data

Data Engineering Skills and Activities

The skill set of a data engineer encompasses the "undercurrents" of data engineering: security, data management, DataOps, data architecture, and software engineering. This skill set requires an understanding of how to evaluate data tools and how they fit together across the data engineering lifecycle. It's also critical to know how data is produced in source systems and how analysts and data scientists will consume and create value after processing and curating data. Finally, a data engineer juggles a lot of complex moving parts and must constantly optimize along the axes of cost, agility, scalability, simplicity, reuse, and interoperability (Figure 1-7). We cover these topics in more detail in upcoming chapters.

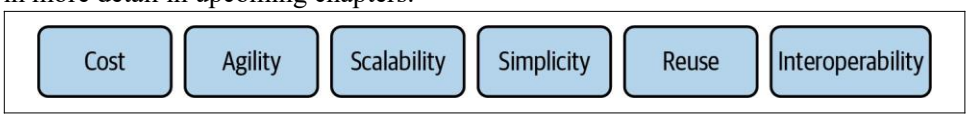


Figure 1-7. The balancing act of data engineering

As we discussed, in the recent past, a data engineer was expected to know and understand how to use a small handful of powerful and monolithic technologies (Hadoop, Spark, Teradata, Hive, and many others) to create a data solution. Utilizing these technologies often requires a sophisticated understanding of software engineering, networking, distributed computing, storage, or other low-level details. Their work would be devoted to cluster administration and maintenance, managing overhead, and writing pipeline and transformation jobs, among other tasks.

Nowadays, the data-tooling landscape is dramatically less complicated to manage and deploy. Modern data tools considerably abstract and simplify workflows. As a result, data engineers are now focused on balancing the simplest and most cost-effective, best-of-breed services that deliver value to the business. The data engineer is also expected to create agile data architectures that evolve as new trends emerge.

What are some things a data engineer does *not* do? A data engineer typically does not directly build ML models, create reports or dashboards, perform data analysis, build key performance indicators (KPIs), or develop software applications. A data engineer should have a good functioning understanding of these areas to serve stakeholders

best.

Data Maturity and the Data Engineer

The level of data engineering complexity within a company depends a great deal on the company's data maturity. This significantly impacts a data engineer's day-to-day job responsibilities and career progression. What is data maturity, exactly?

•

Data maturity is the progression toward higher data utilization, capabilities, and integration across the organization, but data maturity does not simply depend on the age or revenue of a company. An early-stage startup can have greater data maturity than a 100-year-old company with annual revenues in the billions. What matters is the way data is leveraged as a competitive advantage.

Data maturity models have many versions, such as Data Management Maturity (DMM) and others, and it's hard to pick one that is both simple and useful for data engineering. So, we'll create our own simplified data maturity model. Our data maturity model (Figure 1-8) has three stages: starting with data, scaling with data, and leading with data. Let's look at each of these stages and at what a data engineer typically does at each stage.

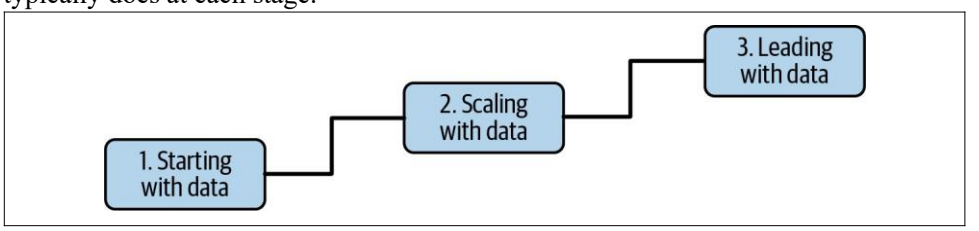


Figure 1-8. Our simplified data maturity model for a company

Stage 1: Starting with data

A company getting started with data is, by definition, in the very early stages of its data maturity. The company may have fuzzy, loosely defined goals or no goals. Data architecture and infrastructure are in the very early stages of planning and development. Adoption and utilization are likely low or nonexistent. The data team is small, often with a headcount in the single digits. At this stage, a data engineer is usually a generalist and will typically play several other roles, such as data scientist or software engineer. A data engineer's goal is to move fast, get traction, and add value.

The practicalities of getting value from data are typically poorly understood, but the desire exists. Reports or analyses lack formal structure, and most requests for data are ad hoc. While it's tempting to jump headfirst into ML at this stage, we don't recommend it. We've seen countless data teams get stuck and fall short when they try to jump to ML without building a solid data foundation.

That's not to say you can't get wins from ML at this stage—it is rare but possible. Without a solid data foundation, you likely won't have the data to train reliable ML models nor the means to deploy these models to production in a scalable and repeatable way. We half-jokingly call ourselves "recovering data scientists", mainly from personal experience with being involved in premature data science projects without adequate data maturity or data engineering support.

A data engineer should focus on the following in organizations getting started with data:

Get buy-in from key stakeholders, including executive management. Ideally, the data engineer should have a sponsor for critical initiatives to design and build a data architecture to support the company's goals.

- Define the right data architecture (usually solo, since a data architect likely isn't available). This means determining business goals and the competitive advantage you're aiming to achieve with your data initiative. Work toward a data architecture that supports these goals. See Chapter 3 for our advice on "good" data architecture.
- Identify and audit data that will support key initiatives and operate within the data architecture you designed.
- Build a solid data foundation for future data analysts and data scientists to generate reports and models that provide competitive value. In the meantime, you may also have to generate these reports and models until this team is hired.

This is a delicate stage with lots of pitfalls. Here are some tips for this stage:

- Organizational willpower may wane if a lot of visible successes don't occur with data. Getting quick wins will establish the importance of data within the organization. Just keep in mind that quick wins will likely create technical debt. Have a plan to reduce this debt, as it will otherwise add friction for future delivery.
- Get out and talk to people, and avoid working in silos. We often see the data team working in a bubble, not communicating with people outside their departments and getting perspectives and feedback from business stakeholders. The danger is you'll spend a lot of time working on things of little use to people.
- Avoid undifferentiated heavy lifting. Don't box yourself in with unnecessary technical complexity. Use off-the-shelf, turnkey solutions wherever possible.
- Build custom solutions and code only where this creates a competitive advantage.

Stage 2: Scaling with data

At this point, a company has moved away from ad hoc data requests and has formal data practices. Now the challenge is creating scalable data architectures and planning for a future where the company is genuinely data-driven. Data engineering roles move from generalists to specialists, with people focusing on particular aspects of the data engineering lifecycle.

In organizations that are in stage 2 of data maturity, a data engineer's goals are to do the following:

- Establish formal data practices

-
- Create scalable and robust data architectures

- Adopt DevOps and DataOps practices
- Build systems that support ML
- Continue to avoid undifferentiated heavy lifting and customize only when a competitive advantage results

We return to each of these goals later in the book. Issues

to watch out for include the following:

- As we grow more sophisticated with data, there's a temptation to adopt bleedingedge technologies based on social proof from Silicon Valley companies. This is rarely a good use of your time and energy. Any technology decisions should be driven by the value they'll deliver to your customers.
- The main bottleneck for scaling is not cluster nodes, storage, or technology but the data engineering team. Focus on solutions that are simple to deploy and manage to expand your team's throughput.
- You'll be tempted to frame yourself as a technologist, a data genius who can deliver magical products. Shift your focus instead to pragmatic leadership and begin transitioning to the next maturity stage; communicate with other teams about the practical utility of data. Teach the organization how to consume and leverage data.

Stage 3: Leading with data

At this stage, the company is data-driven. The automated pipelines and systems created by data engineers allow people within the company to do self-service analytics and ML. Introducing new data sources is seamless, and tangible value is derived. Data engineers implement proper controls and practices to ensure that data is always available to the people and systems. Data engineering roles continue to specialize more deeply than in stage 2.

In organizations in stage 3 of data maturity, a data engineer will continue building on prior stages, plus they will do the following:

- Create automation for the seamless introduction and usage of new data
- Focus on building custom tools and systems that leverage data as a competitive advantage
- Focus on the "enterprisey" aspects of data, such as data management (including data governance and quality) and DataOps
- Deploy tools that expose and disseminate data throughout the organization, including data catalogs, data lineage tools, and metadata management systems

- Collaborate efficiently with software engineers, ML engineers, analysts, and others
- Create a community and environment where people can collaborate and speak openly, no matter their role or position

Issues to watch out for include the following:

- At this stage, complacency is a significant danger. Once organizations reach stage 3, they must constantly focus on maintenance and improvement or risk falling back to a lower stage.
- Technology distractions are a more significant danger here than in the other stages. There's a temptation to pursue expensive hobby projects that don't deliver value to the business. Utilize custom-built technology only where it provides a competitive advantage.

The Background and Skills of a Data Engineer

Data engineering is a fast-growing field, and a lot of questions remain about how to become a data engineer. Because data engineering is a relatively new discipline, little formal training is available to enter the field. Universities don't have a standard data engineering path. Although a handful of data engineering boot camps and online tutorials cover random topics, a common curriculum for the subject doesn't yet exist.

People entering data engineering arrive with varying backgrounds in education, career, and skill set. Everyone entering the field should expect to invest a significant amount of time in self-study. Reading this book is a good starting point; one of the primary goals of this book is to give you a foundation for the knowledge and skills we think are necessary to succeed as a data engineer.

If you're pivoting your career into data engineering, we've found that the transition is easiest when moving from an adjacent field, such as software engineering, ETL development, database administration, data science, or data analysis. These disciplines tend to be "data aware" and provide good context for data roles in an organization. They also equip folks with the relevant technical skills and context to solve data engineering problems.

Despite the lack of a formalized path, a requisite body of knowledge exists that we believe a data engineer should know to be successful. By definition, a data engineer must understand both data and technology. With respect to data, this entails knowing about various best practices around data management. On the technology end, a data engineer must be aware of various options for tools, their interplay, and their trade-

offs. This requires a good understanding of software engineering, DataOps, and data architecture.

Zooming out, a data engineer must also understand the requirements of data consumers (data analysts and data scientists) and the broader implications of data across the organization. Data engineering is a holistic practice; the best data engineers view their responsibilities through business and technical lenses.

Business Responsibilities

The macro responsibilities we list in this section aren't exclusive to data engineers but are crucial for anyone working in a data or technology field. Because a simple Google search will yield tons of resources to learn about these areas, we will simply list them for brevity:

Know how to communicate with nontechnical and technical people.
: Communication is key, and you need to be able to establish rapport and trust with people across the organization. We suggest paying close attention to organizational hierarchies, who reports to whom, how people interact, and which silos exist. These observations will be invaluable to your success.

Understand how to scope and gather business and product requirements.
: You need to know what to build and ensure that your stakeholders agree with your assessment. In addition, develop a sense of how data and technology decisions impact the business.

Understand the cultural foundations of Agile, DevOps, and DataOps.
: Many technologists mistakenly believe these practices are solved through technology. We feel this is dangerously wrong. Agile, DevOps, and DataOps are fundamentally cultural, requiring buy-in across the organization.

Control costs.
: You'll be successful when you can keep costs low while providing outsized value. Know how to optimize for time to value, the total cost of ownership, and opportunity cost. Learn to monitor costs to avoid surprises.

Learn continuously.
: The data field feels like it's changing at light speed. People who succeed in it are great at picking up new things while sharpening their fundamental knowledge. They're also good at filtering, determining which new developments are most relevant to their work, which are still immature, and which are just fads. Stay abreast of the field and learn how to learn.

A successful data engineer always zooms out to understand the big picture and how to achieve outsized value for the business. Communication is vital, both for technical and

-

nontechnical people. We often see data teams succeed based on their communication with other stakeholders; success or failure is rarely a technology issue. Knowing how to navigate an organization, scope and gather requirements,

control costs, and continuously learn will set you apart from the data engineers who rely solely on their technical abilities to carry their career.

Technical Responsibilities

You must understand how to build architectures that optimize performance and cost at a high level, using prepackaged or homegrown components. Ultimately, architectures and constituent technologies are building blocks to serve the data engineering lifecycle. Recall the stages of the data engineering lifecycle:

- Generation
- Storage
- Ingestion
- Transformation
- Serving

The undercurrents of the data engineering lifecycle are the following:

- Security
- Data management
- DataOps
- Data architecture
- Orchestration
- Software engineering

Zooming in a bit, we discuss some of the tactical data and technology skills you'll need as a data engineer in this section; we discuss these in more detail in subsequent chapters.

People often ask, should a data engineer know how to code? Short answer: yes. A data engineer should have production-grade software engineering chops. We note that the nature of software development projects undertaken by data engineers has changed fundamentally in the last few years. Fully managed services now replace a great deal of low-level programming effort previously expected of engineers, who now use managed open source, and simple plug-and-play software-as-a-service (SaaS) offerings. For example, data engineers now focus on high-level abstractions or writing pipelines as code within an orchestration framework.

Even in a more abstract world, software engineering best practices provide a competitive advantage, and data engineers who can dive into the deep architectural details of a codebase give their companies an edge when specific technical needs arise. In short, a data engineer who can't write production-grade code will be severely

hindered, and we don't see this changing anytime soon. Data engineers remain software engineers, in addition to their many other roles.

What languages should a data engineer know? We divide data engineering programming languages into primary and secondary categories. At the time of this writing, the primary languages of data engineering are SQL, Python, a Java Virtual Machine (JVM) language (usually Java or Scala), and bash:

SQL
: The most common interface for databases and data lakes. After briefly being sidelined by the need to write custom MapReduce code for big data processing, SQL (in various forms) has reemerged as the lingua franca of data.

Python
: The bridge language between data engineering and data science. A growing number of data engineering tools are written in Python or have Python APIs. It's known as "the second-best language at everything." Python underlies popular data tools such as pandas, NumPy, Airflow, sci-kit learn, TensorFlow, PyTorch, and PySpark. Python is the glue between underlying components and is frequently a first-class API language for interfacing with a framework.

JVM languages such as Java and Scala
: Prevalent for Apache open source projects such as Spark, Hive, and Druid. The JVM is generally more performant than Python and may provide access to lower-level features than a Python API (for example, this is the case for Apache Spark and Beam). Understanding Java or Scala will be beneficial if you're using a popular open source data framework.

bash
: The command-line interface for Linux operating systems. Knowing bash commands and being comfortable using CLIs will significantly improve your productivity and workflow when you need to script or perform OS operations. Even today, data engineers frequently use command-line tools like awk or sed to process files in a data pipeline or call bash commands from orchestration frameworks. If you're using Windows, feel free to substitute PowerShell for bash.

The Unreasonable Effectiveness of SQL

The advent of MapReduce and the big data era relegated SQL to passé status. Since then, various developments have dramatically enhanced the utility of SQL in the data engineering lifecycle. Spark SQL, Google BigQuery, Snowflake, Hive, and many other data tools can process massive amounts of data by using declarative, set-theoretic SQL semantics. SQL is also supported by many streaming frameworks, such as Apache Flink,

Beam, and Kafka. We believe that competent data engineers should be highly proficient in SQL.

Are we saying that SQL is a be-all and end-all language? Not at all. SQL is a powerful tool that can quickly solve complex analytics and data transformation problems. Given that time is a primary constraint for data engineering team throughput, engineers should embrace tools that combine simplicity and high productivity. Data engineers also do well to develop expertise in composing SQL with other operations, either within frameworks such as Spark and Flink or by using orchestration to combine multiple tools. Data engineers should also learn modern SQL semantics for dealing with JavaScript Object Notation (JSON) parsing and nested data and consider leveraging a SQL management framework such as dbt (Data Build Tool).

A proficient data engineer also recognizes when SQL is not the right tool for the job and can choose and code in a suitable alternative. A SQL expert could likely write a query to stem and tokenize raw text in a natural language processing (NLP) pipeline but would also recognize that coding in native Spark is a far superior alternative to this masochistic exercise.

Data engineers may also need to develop proficiency in secondary programming languages, including R, JavaScript, Go, Rust, C/C++, C#, and Julia. Developing in these languages is often necessary when popular across the company or used with domain-specific data tools. For instance, JavaScript has proven popular as a language for user-defined functions in cloud data warehouses. At the same time, C# and PowerShell are essential in companies that leverage Azure and the Microsoft ecosystem.

Keeping Pace in a Fast-Moving Field

Once a new technology rolls over you, if you're not part of the steamroller, you're part of the road.

—Stewart Brand

How do you keep your skills sharp in a rapidly changing field like data engineering? Should you focus on the latest tools or deep dive into fundamentals? Here's our advice: focus on the fundamentals to understand what's not going to change; pay attention to ongoing developments to know where the field is going. New paradigms and practices are introduced all the time, and it's incumbent on you to stay current.
Strive to understand how new technologies will be helpful in the lifecycle.

The Continuum of Data Engineering Roles, from A to B

Although job descriptions paint a data engineer as a "unicorn" who must possess every data skill imaginable, data engineers don't all do the same type of work or have the

same skill set. Data maturity is a helpful guide to understanding the types of data challenges a company will face as it grows its data capability. It's beneficial to look at some critical distinctions in the kinds of work data engineers do. Though these distinctions are simplistic, they clarify what data scientists and data engineers do and avoid lumping either role into the unicorn bucket.

In data science, there's the notion of type A and type B data scientists.[10] *Type A data scientists*—where *A* stands for *analysis*—focus on understanding and deriving insight from data. *Type B data scientists*—where *B* stands for *building*—share similar backgrounds as type A data scientists and possess strong programming skills. The type B data scientist builds systems that make data science work in production. Borrowing from this data scientist continuum, we'll create a similar distinction for two types of data engineers:

Type A data engineers
: *A* stands for *abstraction*. In this case, the data engineer avoids undifferentiated heavy lifting, keeping data architecture as abstract and straightforward as possible and not reinventing the wheel. Type A data engineers manage the data engineering lifecycle mainly by using entirely off-the-shelf products, managed services, and tools. Type A data engineers work at companies across industries and at all levels of data maturity.

Type B data engineers
: *B* stands for *build*. Type B data engineers build data tools and systems that scale and leverage a company's core competency and competitive advantage. In the data maturity range, a type B data engineer is more commonly found at companies in stage 2 and 3 (scaling and leading with data), or when an initial data use case is so unique and mission-critical that custom data tools are required to get started.

Type A and type B data engineers may work in the same company and may even be the same person! More commonly, a type A data engineer is first hired to set the foundation, with type B data engineer skill sets either learned or hired as the need arises within a company.

Data Engineers Inside an Organization

Data engineers don't work in a vacuum. Depending on what they're working on, they will interact with technical and nontechnical people and face different directions (internal and external). Let's explore what data engineers do inside an organization and with whom they interact.

[10] Robert Chang, "Doing Data Science at Twitter," *Medium*, June 20, 2015, *https://oreil.ly/xqjAx*.

Internal-Facing Versus External-Facing Data Engineers

A data engineer serves several end users and faces many internal and external directions (Figure 1-9). Since not all data engineering workloads and responsibilities are the same, it's essential to understand whom the data engineer serves. Depending on the end-use cases, a data engineer's primary responsibilities are external facing, internal facing, or a blend of the two.

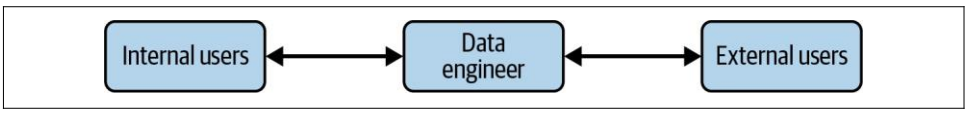


Figure 1-9. The directions a data engineer faces

An *external-facing* data engineer typically aligns with the users of external-facing applications, such as social media apps, Internet of Things (IoT) devices, and ecommerce platforms. This data engineer architects, builds, and manages the systems that collect, store, and process transactional and event data from these applications. The systems built by these data engineers have a feedback loop from the application to the data pipeline, and then back to the application (Figure 1-10).

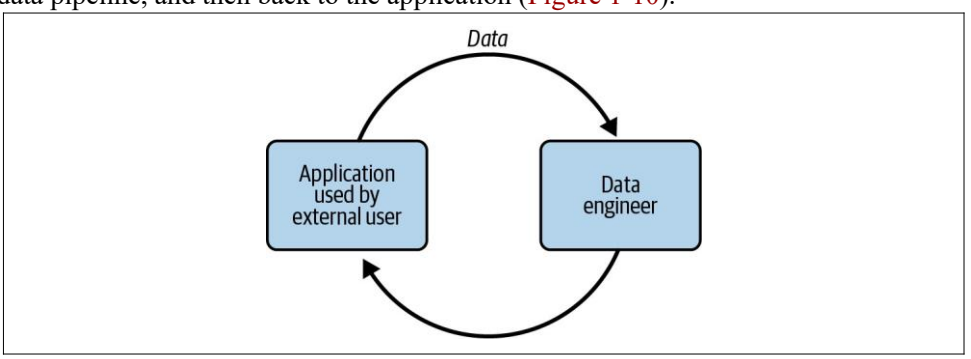


Figure 1-10. External-facing data engineer systems

External-facing data engineering comes with a unique set of problems. Externalfacing query engines often handle much larger concurrency loads than internalfacing systems. Engineers also need to consider putting tight limits on queries that users can run to limit the infrastructure impact of any single user. In addition, security is a much more complex and sensitive problem for external queries, especially if the data being queried is multitenant (data from many customers and housed in a single table).

An *internal-facing data engineer* typically focuses on activities crucial to the needs of the business and internal stakeholders (Figure 1-11). Examples include creating and maintaining data pipelines and data warehouses for BI dashboards, reports, business processes, data science, and ML models.

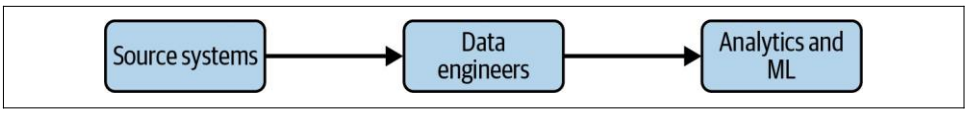


Figure 1-11. Internal-facing data engineer

External-facing and internal-facing responsibilities are often blended. In practice, internal-facing data is usually a prerequisite to external-facing data. The data engineer has two sets of users with very different requirements for query concurrency, security, and more.

Data Engineers and Other Technical Roles

In practice, the data engineering lifecycle cuts across many domains of responsibility. Data engineers sit at the nexus of various roles, directly or through managers, interacting with many organizational units.

Let's look at whom a data engineer may impact. In this section, we'll discuss technical roles connected to data engineering (Figure 1-12).

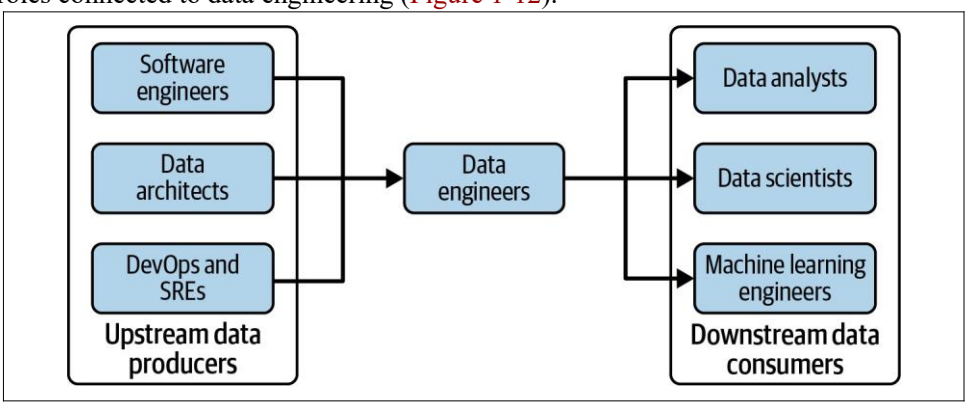


Figure 1-12. Key technical stakeholders of data engineering

The data engineer is a hub between *data producers*, such as software engineers, data architects, and DevOps or site-reliability engineers (SREs), and *data consumers*, such as data analysts, data scientists, and ML engineers. In addition, data engineers will interact with those in operational roles, such as DevOps engineers.

Given the pace at which new data roles come into vogue (analytics and ML engineers come to mind), this is by no means an exhaustive list.

Upstream stakeholders

To be successful as a data engineer, you need to understand the data architecture you're using or designing and the source systems producing the data you'll need. Next, we

discuss a few familiar upstream stakeholders: data architects, software engineers, and DevOps engineers.

Data architects. Data architects function at a level of abstraction one step removed from data engineers. Data architects design the blueprint for organizational data management, mapping out processes and overall data architecture and systems.[11]They also serve as a bridge between an organization's technical and nontechnical sides. Successful data architects generally have "battle scars" from extensive engineering experience, allowing them to guide and assist engineers while successfully communicating engineering challenges to nontechnical business stakeholders. Data architects implement policies for managing data across silos and business units, steer global strategies such as data management and data governance, and guide significant initiatives. Data architects often play a central role in cloud migrations and greenfield cloud design.

The advent of the cloud has shifted the boundary between data architecture and data engineering. Cloud data architectures are much more fluid than on-premises systems, so architecture decisions that traditionally involved extensive study, long lead times, purchase contracts, and hardware installation are now often made during the implementation process, just one step in a larger strategy. Nevertheless, data architects will remain influential visionaries in enterprises, working hand in hand with data engineers to determine the big picture of architecture practices and data strategies.

Depending on the company's data maturity and size, a data engineer may overlap with or assume the responsibilities of a data architect. Therefore, a data engineer should have a good understanding of architecture best practices and approaches.

Note that we have placed data architects in the *upstream stakeholders* section. Data architects often help design application data layers that are source systems for data engineers. Architects may also interact with data engineers at various other stages of the data engineering lifecycle. We cover "good" data architecture in Chapter 3.

Software engineers. Software engineers build the software and systems that run a business; they are largely responsible for generating the *internal data* that data engineers will consume and process. The systems built by software engineers typically generate application event data and logs, which are significant assets in their own right. This internal data contrasts with *external data* pulled from SaaS platforms or partner businesses. In well-run technical organizations, software engineers and data engineers

[11] Paramita (Guha) Ghosh, "Data Architect vs. Data Engineer," Dataversity, November 12, 2021, *https://oreil.ly/TlyZY*.

coordinate from the inception of a new project to design application data for consumption by analytics and ML applications.

A data engineer should work together with software engineers to understand the applications that generate data, the volume, frequency, and format of the generated data, and anything else that will impact the data engineering lifecycle, such as data security and regulatory compliance. For example, this might mean setting upstream expectations on what the data software engineers need to do their jobs. Data engineers must work closely with the software engineers.

DevOps engineers and site-reliability engineers. DevOps and SREs often produce data through operational monitoring. We classify them as upstream of data engineers, but they may also be downstream, consuming data through dashboards or interacting with data engineers directly in coordinating operations of data systems.

Downstream stakeholders

Data engineering exists to serve downstream data consumers and use cases. This section discusses how data engineers interact with various downstream roles. We'll also introduce a few service models, including centralized data engineering teams and cross-functional teams.

Data scientists. Data scientists build forward-looking models to make predictions and recommendations. These models are then evaluated on live data to provide value in various ways. For example, model scoring might determine automated actions in response to real-time conditions, recommend products to customers based on the browsing history in their current session, or make live economic predictions used by traders.

According to common industry folklore, data scientists spend 70% to 80% of their time collecting, cleaning, and preparing data.[12] In our experience, these numbers often reflect immature data science and data engineering practices. In particular, many popular data science frameworks can become bottlenecks if they are not scaled up appropriately. Data scientists who work exclusively on a single workstation force themselves to downsample data, making data preparation significantly more complicated and potentially compromising the quality of the models they produce. Furthermore, locally developed code and environments are often difficult to deploy in

[12] A variety of references exist for this notion. Although this cliche is widely known, a healthy debate has arisen around its validity in different practical settings. For more details, see Leigh Dodds, "Do Data Scientists Spend 80% of Their Time Cleaning Data? Turns Out, No?" Lost Boy blog, January 31, 2020, *https://oreil.ly/szFww*; and Alex Woodie, "Data Prep Still Dominates Data Scientists' Time, Survey Finds," *Datanami*, July 6, 2020, *https://oreil.ly/jDVWF*.

production, and a lack of automation significantly hampers data science workflows. If data engineers do their job and collaborate successfully, data scientists shouldn't spend their time collecting, cleaning, and preparing data after initial exploratory work. Data engineers should automate this work as much as possible.

The need for production-ready data science is a significant driver behind the emergence of the data engineering profession. Data engineers should help data scientists to enable a path to production. In fact, we (the authors) moved from data science to data engineering after recognizing this fundamental need. Data engineers work to provide the data automation and scale that make data science more efficient.

Data analysts. Data analysts (or business analysts) seek to understand business performance and trends. Whereas data scientists are forward-looking, a data analyst typically focuses on the past or present. Data analysts usually run SQL queries in a data warehouse or a data lake. They may also utilize spreadsheets for computation and analysis and various BI tools such as Microsoft Power BI, Looker, or Tableau. Data analysts are domain experts in the data they work with frequently and become intimately familiar with data definitions, characteristics, and quality problems. A data analyst's typical downstream customers are business users, management, and

executives.

Data engineers work with data analysts to build pipelines for new data sources required by the business. Data analysts' subject-matter expertise is invaluable in improving data quality, and they frequently collaborate with data engineers in this capacity.

Machine learning engineers and AI researchers. Machine learning engineers (ML engineers) overlap with data engineers and data scientists. ML engineers develop advanced ML techniques, train models, and design and maintain the infrastructure running ML processes in a scaled production environment. ML engineers often have advanced working knowledge of ML and deep learning techniques and frameworks such as PyTorch or TensorFlow.

ML engineers also understand the hardware, services, and systems required to run these frameworks, both for model training and model deployment at a production scale. It's common for ML flows to run in a cloud environment where ML engineers can spin up and scale infrastructure resources on demand or rely on managed services.

As we've mentioned, the boundaries between ML engineering, data engineering, and data science are blurry. Data engineers may have some operational responsibilities over ML systems, and data scientists may work closely with ML engineering in designing advanced ML processes.

The world of ML engineering is snowballing and parallels a lot of the same developments occurring in data engineering. Whereas several years ago, the attention of ML was focused on how to build models, ML engineering now increasingly emphasizes incorporating best practices of machine learning operations (MLOps) and other mature practices previously adopted in software engineering and DevOps. AI researchers work on new, advanced ML techniques. AI researchers may work inside large technology companies, specialized intellectual property startups (OpenAI, DeepMind), or academic institutions. Some practitioners are dedicated to part-time research in conjunction with ML engineering responsibilities inside a company. Those working inside specialized ML labs are often 100% dedicated to research. Research problems may target immediate practical applications or more abstract demonstrations of AI. DALL-E, Gato AI, AlphaGo, and GPT-3/GPT-4 are great examples of ML research projects. Given the pace of advancements in ML, these examples will very likely be quaint in a few years' time. We've provided some references in "Additional Resources" on page 32.

AI researchers in well-funded organizations are highly specialized and operate with supporting teams of engineers to facilitate their work. ML engineers in academia usually have fewer resources but rely on teams of graduate students, postdocs, and university staff to provide engineering support. ML engineers who are partially dedicated to research often rely on the same support teams for research and production.

Data Engineers and Business Leadership

We've discussed technical roles with which a data engineer interacts. But data engineers also operate more broadly as organizational connectors, often in a nontechnical capacity. Businesses have come to rely increasingly on data as a core part of many products or a product in itself. Data engineers now participate in strategic planning and lead key initiatives that extend beyond the boundaries of IT. Data engineers often support data architects by acting as the glue between the business and data science/analytics.

Data in the C-suite

C-level executives are increasingly involved in data and analytics, as these are recognized as significant assets for modern businesses. For example, CEOs now concern themselves with initiatives that were once the exclusive province of IT, such as cloud migrations or deployment of a new customer data platform.

Chief executive officer. Chief executive officers (CEOs) at nontech companies generally don't concern themselves with the nitty-gritty of data frameworks and software. Instead, they define a vision in collaboration with technical C-suite roles and company data leadership. Data engineers provide a window into what's possible with

data. Data engineers and their managers maintain a map of what data is available to the organization—both internally and from third parties—in what time frame. They are also tasked to study primary data architectural changes in collaboration with other engineering roles. For example, data engineers are often heavily involved in cloud migrations, migrations to new data systems, or deployment of streaming technologies.

Chief information officer. A chief information officer (CIO) is the senior C-suite executive responsible for information technology within an organization; it is an internal-facing role. A CIO must possess deep knowledge of information technology and business processes—either alone is insufficient. CIOs direct the information technology organization, setting ongoing policies while also defining and executing significant initiatives under the direction of the CEO.

CIOs often collaborate with data engineering leadership in organizations with a welldeveloped data culture. If an organization is not very high in its data maturity, a CIO will typically help shape its data culture. CIOs will work with engineers and architects to map out major initiatives and make strategic decisions on adopting major architectural elements, such as enterprise resource planning (ERP) and customer relationship management (CRM) systems, cloud migrations, data systems, and internal-facing IT.

Chief technology officer. A chief technology officer (CTO) is similar to a CIO but faces outward. A CTO owns the key technological strategy and architectures for externalfacing applications, such as mobile, web apps, and IoT—all critical data sources for data engineers. The CTO is likely a skilled technologist and has a good sense of software engineering fundamentals and system architecture. In some organizations without a CIO, the CTO or sometimes the chief operating officer (COO) plays the role of CIO. Data engineers often report directly or indirectly through a CTO.

Chief data officer. The chief data officer (CDO) was created in 2002 at Capital One to recognize the growing importance of data as a business asset. The CDO is responsible for a company's data assets and strategy. CDOs are focused on data's business utility but should have a strong technical grounding. CDOs oversee data products, strategy, initiatives, and core functions such as master data management and privacy. Occasionally, CDOs manage business analytics and data engineering.

Chief analytics officer. The chief analytics officer (CAO) is a variant of the CDO role. Where both roles exist, the CDO focuses on the technology and organization required to deliver data. The CAO is responsible for analytics, strategy, and decision making for the business. A CAO may oversee data science and ML, though this largely depends on whether the company has a CDO or CTO role.

Chief algorithms officer. A chief algorithms officer (CAO-2) is a recent innovation in the C-suite, a highly technical role focused specifically on data science and ML. CAO-2s typically have experience as individual contributors and team leads in data science or ML projects. Frequently, they have a background in ML research and a related advanced degree.

CAO-2s are expected to be conversant in current ML research and have deep technical knowledge of their company's ML initiatives. In addition to creating business initiatives, they provide technical leadership, set research and development agendas, and build research teams.

Data engineers and project managers

Data engineers often work on significant initiatives, potentially spanning many years. As we write this book, many data engineers are working on cloud migrations, migrating pipelines and warehouses to the next generation of data tools. Other data engineers are starting greenfield projects, assembling new data architectures from scratch by selecting from an astonishing number of best-of-breed architecture and tooling options.

These large initiatives often benefit from *project management* (in contrast to product management, discussed next). Whereas data engineers function in an infrastructure and service delivery capacity, project managers direct traffic and serve as gatekeepers. Most project managers operate according to some variation of Agile and Scrum, with Waterfall still appearing occasionally. Business never sleeps, and business stakeholders often have a significant backlog of things they want to address and new initiatives they want to launch. Project managers must filter a long list of requests and prioritize critical deliverables to keep projects on track and better serve the company.

Data engineers interact with project managers, often planning sprints for projects and ensuing standups related to the sprint. Feedback goes both ways, with data engineers informing project managers and other stakeholders about progress and blockers, and project managers balancing the cadence of technology teams against the ever-changing needs of the business.

Data engineers and product managers

Product managers oversee product development, often owning product lines. In the context of data engineers, these products are called *data products*. Data products are either built from the ground up or are incremental improvements upon existing products. Data engineers interact more frequently with *product managers* as the corporate world has adopted a data-centric focus. Like project managers, product managers balance the activity of technology teams against the needs of the customer and business.

Data engineers and other management roles

Data engineers interact with various managers beyond project and product managers. However, these interactions usually follow either the services or cross-functional models. Data engineers either serve a variety of incoming requests as a centralized team or work as a resource assigned to a particular manager, project, or product.

For more information on data teams and how to structure them, we recommend John Thompson's *Building Analytics Teams* (Packt) and Jesse Anderson's *Data Teams* (Apress). Both books provide strong frameworks and perspectives on the roles of executives with data, who to hire, and how to construct the most effective data team for your company.

Companies don't hire engineers simply to hack on code in isolation. To be worthy of their title, engineers should develop a deep understanding of the problems they're tasked with solving, the technology tools at their disposal, and the people they work with and serve.

Conclusion

This chapter provided you with a brief overview of the data engineering landscape, including the following:

- Defining data engineering and describing what data engineers do
- Describing the types of data maturity in a company
- Type A and type B data engineers
- Whom data engineers work with

We hope that this first chapter has whetted your appetite, whether you are a software development practitioner, data scientist, ML engineer, business stakeholder, entrepreneur, or venture capitalist. Of course, a great deal still remains to elucidate in subsequent chapters. Chapter 2 covers the data engineering lifecycle, followed by architecture in Chapter 3. The following chapters get into the nitty-gritty of technology decisions for each part of the lifecycle. The entire data field is in flux, and as much as possible, each chapter focuses on the *immutables*—perspectives that will be valid for many years amid relentless change.

Additional Resources

- "The AI Hierarchy of Needs" by Monica Rogati
- The AlphaGo research web page
- "Big Data Will Be Dead in Five Years" by Lewis Gavin
- *Building Analytics Teams* by John K. Thompson (Packt)
- Chapter 1 of *What Is Data Engineering?* by Lewis Gavin (O'Reilly)
- "Data as a Product vs. Data as a Service" by Justin Gage
- "Data Engineering: A Quick and Simple Definition" by James Furbush (O'Reilly)
- *Data Teams* by Jesse Anderson (Apress)
- "Doing Data Science at Twitter" by Robert Chang
- "The Downfall of the Data Engineer" by Maxime Beauchemin
- "The Future of Data Engineering Is the Convergence of Disciplines" by Liam Hausmann
- "How CEOs Can Lead a Data-Driven Culture" by Thomas H. Davenport and Nitin Mittal
- "How Creating a Data-Driven Culture Can Drive Success" by Frederik Bussler
- The Information Management Body of Knowledge website
- "Information Management Body of Knowledge" Wikipedia page
- "Information Management" Wikipedia page
- "On Complexity in Big Data" by Jesse Anderson (O'Reilly)
- "OpenAI's New Language Generator GPT-3 Is Shockingly Good—and Completely Mindless" by Will Douglas Heaven
- "The Rise of the Data Engineer" by Maxime Beauchemin
- "A Short History of Big Data" by Mark van Rijmenam
- "Skills of the Data Architect" by Bob Lambert
- "The Three Levels of Data Analysis: A Framework for Assessing Data Organization Maturity" by Emilie Schario
- "What Is a Data Architect? IT's Data Framework Visionary" by Thor Olavsrud
- "Which Profession Is More Complex to Become, a Data Engineer or a Data Scientist?" thread on Quora
- "Why CEOs Must Lead Big Data Initiatives" by John Weathington

The Data Engineering Lifecycle

The major goal of this book is to encourage you to move beyond viewing data engineering as a specific collection of data technologies. The data landscape is undergoing an explosion of new data technologies and practices, with ever-increasing levels of abstraction and ease of use. Because of increased technical abstraction, data engineers will increasingly become *data lifecycle engineers*, thinking and operating in terms of the *principles* of data lifecycle management.

In this chapter, you'll learn about the *data engineering lifecycle*, which is the central theme of this book. The data engineering lifecycle is our framework describing "cradle to grave" data engineering. You will also learn about the undercurrents of the data engineering lifecycle, which are key foundations that support all data engineering efforts.

What Is the Data Engineering Lifecycle?

The data engineering lifecycle comprises stages that turn raw data ingredients into a useful end product, ready for consumption by analysts, data scientists, ML engineers, and others. This chapter introduces the major stages of the data engineering lifecycle, focusing on each stage's core concepts and saving details for later chapters. We divide the data engineering lifecycle into five stages (Figure 2-1, top):

- Generation
- Storage
- Ingestion
- Transformation
- Serving data

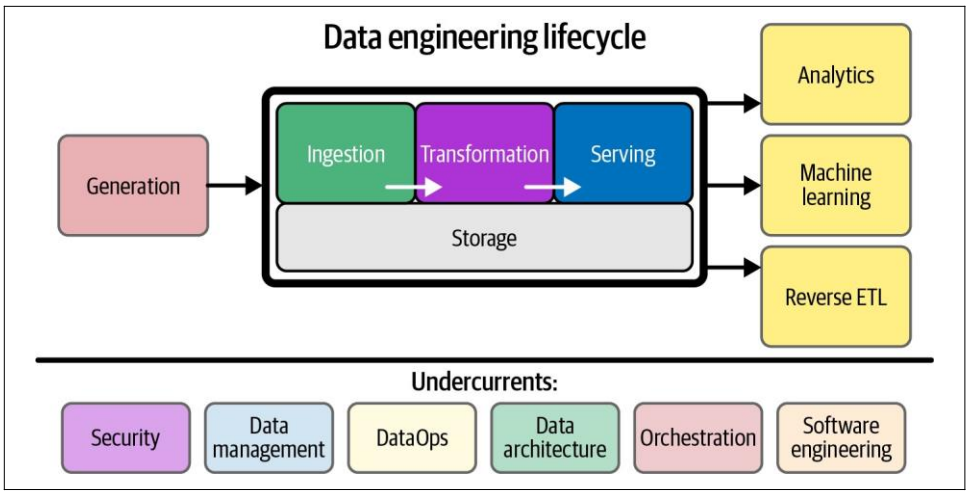


Figure 2-1. Components and undercurrents of the data engineering lifecycle

We begin the data engineering lifecycle by getting data from source systems and storing it. Next, we transform the data and then proceed to our central goal, serving data to analysts, data scientists, ML engineers, and others. In reality, storage occurs throughout the lifecycle as data flows from beginning to end—hence, the diagram shows the storage "stage" as a foundation that underpins other stages.

In general, the middle stages—storage, ingestion, transformation—can get a bit jumbled. And that's OK. Although we split out the distinct parts of the data engineering lifecycle, it's not always a neat, continuous flow. Various stages of the lifecycle may repeat themselves, occur out of order, overlap, or weave together in interesting and unexpected ways.

Acting as a bedrock are *undercurrents* (Figure 2-1, bottom) that cut across multiple stages of the data engineering lifecycle: security, data management, DataOps, data architecture, orchestration, and software engineering. No part of the data engineering lifecycle can adequately function without these undercurrents.

The Data Lifecycle Versus the Data Engineering Lifecycle

You may be wondering about the difference between the overall data lifecycle and the data engineering lifecycle. There's a subtle distinction between the two. The data engineering lifecycle is a subset of the whole data lifecycle (Figure 2-2). Whereas the full data lifecycle encompasses data across its entire lifespan, the data engineering lifecycle focuses on the stages a data engineer controls.

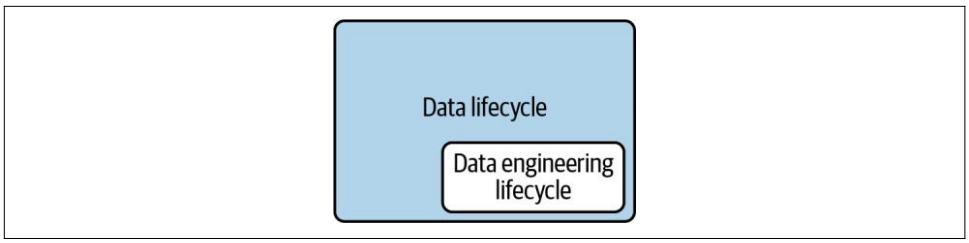


Figure 2-2. The data engineering lifecycle is a subset of the full data lifecycle

Generation: Source Systems

A *source system* is the origin of the data used in the data engineering lifecycle. For example, a source system could be an IoT device, an application message queue, or a transactional database. A data engineer consumes data from a source system but doesn't typically own or control the source system itself. The data engineer needs to have a working understanding of the way source systems work, the way they generate data, the frequency and velocity of the data, and the variety of data they generate.

Engineers also need to keep an open line of communication with source system owners on changes that could break pipelines and analytics. Application code might change the structure of data in a field, or the application team might even choose to migrate the backend to an entirely new database technology.

A major challenge in data engineering is the dizzying array of data source systems engineers must work with and understand. As an illustration, let's look at two common source systems, one very traditional (an application database) and the other a more recent example (IoT swarms).

Figure 2-3 illustrates a traditional source system with several application servers supported by a database. This source system pattern became popular in the 1980s with the explosive success of relational database management systems (RDBMSs). The application + database pattern remains popular today with various modern evolutions of software development practices. For example, applications often consist of many small service/database pairs with microservices rather than a single monolith.

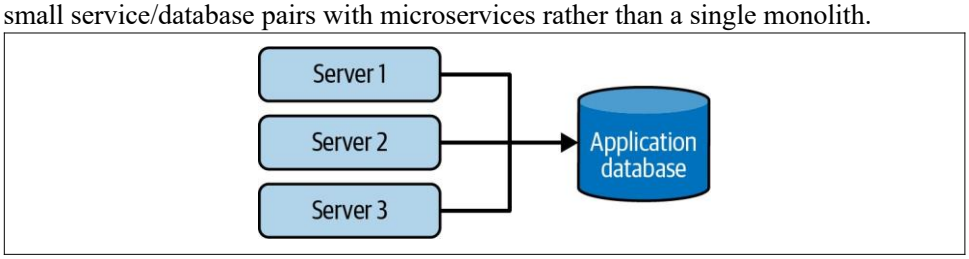


Figure 2-3. Source system example: an application database

Let's look at another example of a source system. Figure 2-4 illustrates an IoT swarm: a fleet of devices (circles) sends data messages (rectangles) to a central collection system. This IoT source system is increasingly common as IoT devices such as sensors, smart devices, and much more increase in the wild.

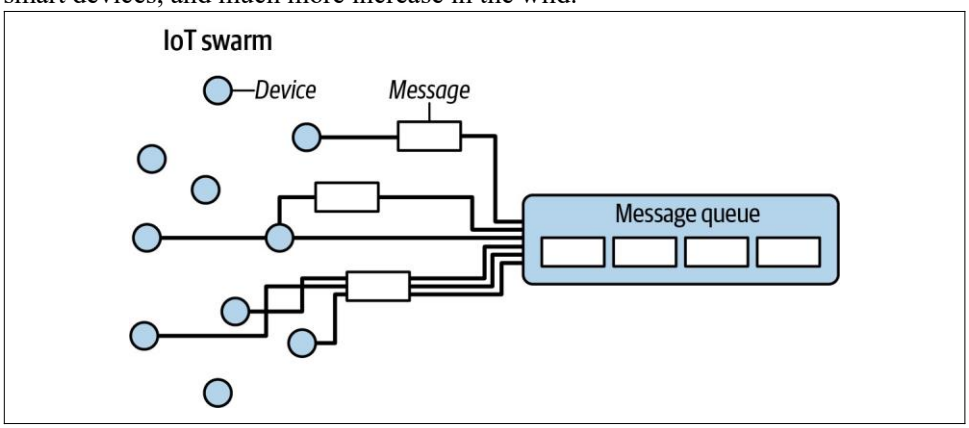


Figure 2-4. Source system example: an IoT swarm and message queue

Evaluating source systems: Key engineering considerations

There are many things to consider when assessing source systems, including how the system handles ingestion, state, and data generation. The following is a starting set of evaluation questions of source systems that data engineers must consider:

- What are the essential characteristics of the data source? Is it an application? A swarm of IoT devices?
- How is data persisted in the source system? Is data persisted long term, or is it temporary and quickly deleted?
- At what rate is data generated? How many events per second? How many gigabytes per hour?
- What level of consistency can data engineers expect from the output data? If you're running data-quality checks against the output data, how often do data inconsistencies occur—nulls where they aren't expected, lousy formatting, etc.?
- How often do errors occur?
- Will the data contain duplicates?
- Will some data values arrive late, possibly much later than other messages produced simultaneously?
- What is the schema of the ingested data? Will data engineers need to join across several tables or even several systems to get a complete picture of the data?

- If schema changes (say, a new column is added), how is this dealt with and communicated to downstream stakeholders?
- How frequently should data be pulled from the source system?
- For stateful systems (e.g., a database tracking customer account information), is data provided as periodic snapshots or update events from change data capture (CDC)? What's the logic for how changes are performed, and how are these tracked in the source database?
- Who/what is the data provider that will transmit the data for downstream consumption?
- Will reading from a data source impact its performance?
- Does the source system have upstream data dependencies? What are the characteristics of these upstream systems?
- Are data-quality checks in place to check for late or missing data?

Sources produce data consumed by downstream systems, including humangenerated spreadsheets, IoT sensors, and web and mobile applications. Each source has its unique volume and cadence of data generation. A data engineer should know how the source generates data, including relevant quirks or nuances. Data engineers also need to understand the limits of the source systems they interact with. For example, will analytical queries against a source application database cause resource contention and performance issues?

One of the most challenging nuances of source data is the schema. The *schema* defines the hierarchical organization of data. Logically, we can think of data at the level of a whole source system, drilling down into individual tables, all the way to the structure of respective fields. The schema of data shipped from source systems is handled in various ways. Two popular options are schemaless and fixed schema.

Schemaless doesn't mean the absence of schema. Rather, it means that the application defines the schema as data is written, whether to a message queue, a flat file, a blob, or a document database such as MongoDB. A more traditional model built on relational database storage uses a *fixed schema* enforced in the database, to which application writes must conform.

Either of these models presents challenges for data engineers. Schemas change over time; in fact, schema evolution is encouraged in the Agile approach to software development. A key part of the data engineer's job is taking raw data input in the source system schema and transforming this into valuable output for analytics. This job becomes more challenging as the source schema evolves.

We dive into source systems in greater detail in Chapter 5; we also cover schemas and data modeling in Chapters 6 and 8, respectively.

Storage

You need a place to store data. Choosing a storage solution is key to success in the rest of the data lifecycle, and it's also one of the most complicated stages of the data lifecycle for a variety of reasons. First, data architectures in the cloud often leverage *several* storage solutions. Second, few data storage solutions function purely as storage, with many supporting complex transformation queries; even object storage solutions may support powerful query capabilities—e.g., Amazon S3 Select. Third, while storage is a stage of the data engineering lifecycle, it frequently touches on other stages, such as ingestion, transformation, and serving.

Storage runs across the entire data engineering lifecycle, often occurring in multiple places in a data pipeline, with storage systems crossing over with source systems, ingestion, transformation, and serving. In many ways, the way data is stored impacts how it is used in all of the stages of the data engineering lifecycle. For example, cloud data warehouses can store data, process data in pipelines, and serve it to analysts. Streaming frameworks such as Apache Kafka and Pulsar can function simultaneously as ingestion, storage, and query systems for messages, with object storage being a standard layer for data transmission.

Evaluating storage systems: Key engineering considerations

Here are a few key engineering questions to ask when choosing a storage system for a data warehouse, data lakehouse, database, or object storage:

- Is this storage solution compatible with the architecture's required write and read speeds?
- Will storage create a bottleneck for downstream processes?
- Do you understand how this storage technology works? Are you utilizing the storage system optimally or committing unnatural acts? For instance, are you applying a high rate of random access updates in an object storage system? (This is an antipattern with significant performance overhead.)
- Will this storage system handle anticipated future scale? You should consider all capacity limits on the storage system: total available storage, read operation rate, write volume, etc.
- Will downstream users and processes be able to retrieve data in the required service-level agreement (SLA)?
- Are you capturing metadata about schema evolution, data flows, data lineage, and so forth? Metadata has a significant impact on the utility of data. Metadata

represents an investment in the future, dramatically enhancing discoverability and institutional knowledge to streamline future projects and architecture changes.

- Is this a pure storage solution (object storage), or does it support complex query patterns (i.e., a cloud data warehouse)?
- Is the storage system schema-agnostic (object storage)? Flexible schema (Cassandra)? Enforced schema (a cloud data warehouse)?
- How are you tracking master data, golden records data quality, and data lineage for data governance? (We have more to say on these in "Data Management" on page 50.)
- How are you handling regulatory compliance and data sovereignty? For example, can you store your data in certain geographical locations but not others?

Understanding data access frequency

Not all data is accessed in the same way. Retrieval patterns will greatly vary based on the data being stored and queried. This brings up the notion of the "temperatures" of data. Data access frequency will determine the temperature of your data.

Data that is most frequently accessed is called *hot data*. Hot data is commonly retrieved many times per day, perhaps even several times per second—for example, in systems that serve user requests. This data should be stored for fast retrieval, where "fast" is relative to the use case. *Lukewarm data* might be accessed every so often—say, every week or month.

Cold data is seldom queried and is appropriate for storing in an archival system. Cold data is often retained for compliance purposes or in case of a catastrophic failure in another system. In the "old days," cold data would be stored on tapes and shipped to remote archival facilities. In cloud environments, vendors offer specialized storage tiers with very cheap monthly storage costs but high prices for data retrieval.

Selecting a storage system

What type of storage solution should you use? This depends on your use cases, data volumes, frequency of ingestion, format, and size of the data being ingested—essentially, the key considerations listed in the preceding bulleted questions. There is no one-size-fits-all universal storage recommendation. Every storage technology has its trade-offs. Countless varieties of storage technologies exist, and it's easy to be overwhelmed when deciding the best option for your data architecture.

Chapter 6 covers storage best practices and approaches in greater detail, as well as the crossover between storage and other lifecycle stages.

Ingestion

After you understand the data source, the characteristics of the source system you're using, and how data is stored, you need to gather the data. The next stage of the data engineering lifecycle is data ingestion from source systems.

In our experience, source systems and ingestion represent the most significant bottlenecks of the data engineering lifecycle. The source systems are normally outside your direct control and might randomly become unresponsive or provide data of poor quality. Or, your data ingestion service might mysteriously stop working for many reasons. As a result, data flow stops or delivers insufficient data for storage, processing, and serving.

Unreliable source and ingestion systems have a ripple effect across the data engineering lifecycle. But you're in good shape, assuming you've answered the big questions about source systems.

Key engineering considerations for the ingestion phase

When preparing to architect or build a system, here are some primary questions about the ingestion stage:

- What are the use cases for the data I'm ingesting? Can I reuse this data rather than create multiple versions of the same dataset?
- Are the systems generating and ingesting this data reliably, and is the data available when I need it?
- What is the data destination after ingestion?
- How frequently will I need to access the data?
- In what volume will the data typically arrive?
- What format is the data in? Can my downstream storage and transformation systems handle this format?
- Is the source data in good shape for immediate downstream use? If so, for how long, and what may cause it to be unusable?
- If the data is from a streaming source, does it need to be transformed before reaching its destination? Would an in-flight transformation be appropriate, where the data is transformed within the stream itself?

These are just a sample of the factors you'll need to think about with ingestion, and we cover those questions and more in Chapter 7. Before we leave, let's briefly turn our attention to two major data ingestion concepts: batch versus streaming and push versus pull.

Batch versus streaming

Virtually all data we deal with is inherently *streaming*. Data is nearly always produced and updated continually at its source. *Batch ingestion* is simply a specialized and convenient way of processing this stream in large chunks—for example, handling a full day's worth of data in a single batch.

Streaming ingestion allows us to provide data to downstream systems—whether other applications, databases, or analytics systems—in a continuous, real-time fashion. Here, *real-time* (or *near real-time*) means that the data is available to a downstream system a short time after it is produced (e.g., less than one second later). The latency required to qualify as real-time varies by domain and requirements.

Batch data is ingested either on a predetermined time interval or as data reaches a preset size threshold. Batch ingestion is a one-way door: once data is broken into batches, the latency for downstream consumers is inherently constrained. Because of limitations of legacy systems, batch was for a long time the default way to ingest data. Batch processing remains an extremely popular way to ingest data for downstream consumption, particularly in analytics and ML.

However, the separation of storage and compute in many systems and the ubiquity of event-streaming and processing platforms make the continuous processing of data streams much more accessible and increasingly popular. The choice largely depends on the use case and expectations for data timeliness.

Key considerations for batch versus stream ingestion

Should you go streaming-first? Despite the attractiveness of a streaming-first approach, there are many trade-offs to understand and think about. The following are some questions to ask yourself when determining whether streaming ingestion is an appropriate choice over batch ingestion:

- If I ingest the data in real time, can downstream storage systems handle the rate of data flow?
- Do I need millisecond real-time data ingestion? Or would a micro-batch approach work, accumulating and ingesting data, say, every minute?
- What are my use cases for streaming ingestion? What specific benefits do I realize by implementing streaming? If I get data in real time, what actions can I take on that data that would be an improvement upon batch?
- Will my streaming-first approach cost more in terms of time, money, maintenance, downtime, and opportunity cost than simply doing batch?
- Are my streaming pipeline and system reliable and redundant if infrastructure fails?

- What tools are most appropriate for the use case? Should I use a managed service (Amazon Kinesis, Google Cloud Pub/Sub, Google Cloud Dataflow) or stand up my own instances of Kafka, Flink, Spark, Pulsar, etc.? If I do the latter, who will manage it? What are the costs and trade-offs?
- If I'm deploying an ML model, what benefits do I have with online predictions and possibly continuous training?
- Am I getting data from a live production instance? If so, what's the impact of my ingestion process on this source system?

As you can see, streaming-first might seem like a good idea, but it's not always straightforward; extra costs and complexities inherently occur. Many great ingestion frameworks do handle both batch and micro-batch ingestion styles. We think batch is an excellent approach for many common use cases, such as model training and weekly reporting. Adopt true real-time streaming only after identifying a business use case that justifies the trade-offs against using batch.

Push versus pull

In the *push* model of data ingestion, a source system writes data out to a target, whether a database, object store, or filesystem. In the *pull* model, data is retrieved from the source system. The line between the push and pull paradigms can be quite blurry; data is often pushed and pulled as it works its way through the various stages of a data pipeline.

Consider, for example, the extract, transform, load (ETL) process, commonly used in batch-oriented ingestion workflows. ETL's *extract* (*E*) part clarifies that we're dealing with a pull ingestion model. In traditional ETL, the ingestion system queries a current source table snapshot on a fixed schedule. You'll learn more about ETL and extract, load, transform (ELT) throughout this book.

In another example, consider continuous CDC, which is achieved in a few ways. One common method triggers a message every time a row is changed in the source database. This message is *pushed* to a queue, where the ingestion system picks it up. Another common CDC method uses binary logs, which record every commit to the database. The database *pushes* to its logs. The ingestion system reads the logs but doesn't directly interact with the database otherwise. This adds little to no additional load to the source database. Some versions of batch CDC use the *pull* pattern. For example, in timestamp-based CDC, an ingestion system queries the source database and pulls the rows that have changed since the previous update.

With streaming ingestion, data bypasses a backend database and is pushed directly to an endpoint, typically with data buffered by an event-streaming platform. This pattern

is useful with fleets of IoT sensors emitting sensor data. Rather than relying on a database to maintain the current state, we simply think of each recorded reading as an event. This pattern is also growing in popularity in software applications as it simplifies real-time processing, allows app developers to tailor their messages for downstream analytics, and greatly simplifies the lives of data engineers.

We discuss ingestion best practices and techniques in depth in Chapter 7. Next, let's turn to the transformation stage of the data engineering lifecycle.

Transformation

After you've ingested and stored data, you need to do something with it. The next stage of the data engineering lifecycle is *transformation*, meaning data needs to be changed from its original form into something useful for downstream use cases. Without proper transformations, data will sit inert, and not be in a useful form for reports, analysis, or ML. Typically, the transformation stage is where data begins to create value for downstream user consumption.

Immediately after ingestion, basic transformations map data into correct types (changing ingested string data into numeric and date types, for example), putting records into standard formats, and removing bad ones. Later stages of transformation may transform the data schema and apply normalization. Downstream, we can apply large-scale aggregation for reporting or featurize data for ML processes.

Key considerations for the transformation phase

When considering data transformations within the data engineering lifecycle, it helps to consider the following:

- What's the cost and return on investment (ROI) of the transformation? What is the associated business value?
- Is the transformation as simple and self-isolated as possible?
- What business rules do the transformations support?

You can transform data in batch or while streaming in flight. As mentioned in "Ingestion" on page 39, virtually all data starts life as a continuous stream; batch is just a specialized way of processing a data stream. Batch transformations are overwhelmingly popular, but given the growing popularity of stream-processing solutions and the general increase in the amount of streaming data, we expect the popularity of streaming transformations to continue growing, perhaps entirely replacing batch processing in certain domains soon.

Logically, we treat transformation as a standalone area of the data engineering lifecycle, but the realities of the lifecycle can be much more complicated in practice.

Transformation is often entangled in other phases of the lifecycle. Typically, data is transformed in source systems or in flight during ingestion. For example, a source system may add an event timestamp to a record before forwarding it to an ingestion process. Or a record within a streaming pipeline may be "enriched" with additional fields and calculations before it's sent to a data warehouse. Transformations are ubiquitous in various parts of the lifecycle. Data preparation, data wrangling, and cleaning—these transformative tasks add value for end consumers of data.

Business logic is a major driver of data transformation, often in data modeling. Data translates business logic into reusable elements (e.g., a sale means "somebody bought 12 picture frames from me for $30 each, or $360 in total"). In this case, somebody bought 12 picture frames for $30 each. Data modeling is critical for obtaining a clear and current picture of business processes. A simple view of raw retail transactions might not be useful without adding the logic of accounting rules so that the CFO has a clear picture of financial health. Ensure a standard approach for implementing business logic across your transformations.

Data featurization for ML is another data transformation process. Featurization intends to extract and enhance data features useful for training ML models. Featurization can be a dark art, combining domain expertise (to identify which features might be important for prediction) with extensive experience in data science. For this book, the main point is that once data scientists determine how to featurize data, featurization processes can be automated by data engineers in the transformation stage of a data pipeline.

Transformation is a profound subject, and we cannot do it justice in this brief introduction. Chapter 8 delves into queries, data modeling, and various transformation practices and nuances.

Serving Data

You've reached the last stage of the data engineering lifecycle. Now that the data has been ingested, stored, and transformed into coherent and useful structures, it's time to get value from your data. "Getting value" from data means different things to different users.

Data has *value* when it's used for practical purposes. Data that is not consumed or queried is simply inert. Data vanity projects are a major risk for companies. Many companies pursued vanity projects in the big data era, gathering massive datasets in data lakes that were never consumed in any useful way. The cloud era is triggering a new wave of vanity projects built on the latest data warehouses, object storage systems, and streaming technologies. Data projects must be intentional across the lifecycle. What is the ultimate business purpose of the data so carefully collected, cleaned, and stored?

Data serving is perhaps the most exciting part of the data engineering lifecycle. This is where the magic happens. This is where ML engineers can apply the most advanced techniques. Let's look at some of the popular uses of data: analytics, ML, and reverse ETL.

Analytics

Analytics is the core of most data endeavors. Once your data is stored and transformed, you're ready to generate reports or dashboards and do ad hoc analysis on the data. Whereas the bulk of analytics used to encompass BI, it now includes other facets such as operational analytics and embedded analytics (Figure 2-5). Let's briefly touch on these variations of analytics.

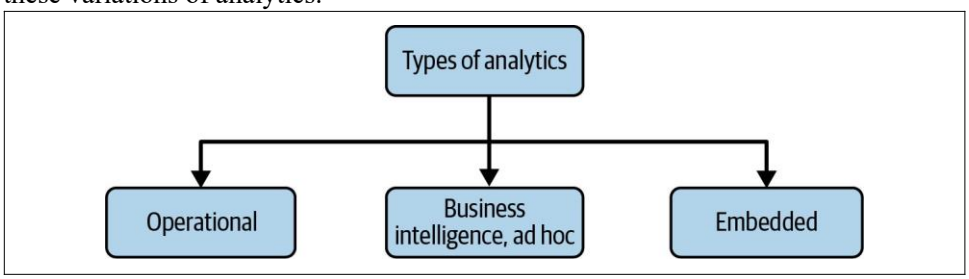


Figure 2-5. Types of analytics

Business intelligence. BI marshals collected data to describe a business's past and current state. BI requires using business logic to process raw data. Note that data serving for analytics is yet another area where the stages of the data engineering lifecycle can get tangled. As we mentioned earlier, business logic is often applied to data in the transformation stage of the data engineering lifecycle, but a logic-on-read approach has become increasingly popular. Data is stored in a clean but fairly raw form, with minimal postprocessing business logic. A BI system maintains a repository of business logic and definitions. This business logic is used to query the data warehouse so that reports and dashboards align with business definitions.

As a company grows its data maturity, it will move from ad hoc data analysis to self-service analytics, allowing democratized data access to business users without needing IT to intervene. The capability to do self-service analytics assumes that data is good enough that people across the organization can simply access it themselves, slice and dice it however they choose, and get immediate insights. Although self-service analytics is simple in theory, it's tough to pull off in practice. The main reason is that poor data quality, organizational silos, and a lack of adequate data skills often get in the way of allowing widespread use of analytics.

Operational analytics. Operational analytics focuses on the fine-grained details of operations, promoting actions that a user of the reports can act upon immediately. Operational analytics could be a live view of inventory or real-time dashboarding of website or application health. In this case, data is consumed in real time, either directly from a source system or from a streaming data pipeline. The types of insights in operational analytics differ from traditional BI since operational analytics is focused on the present and doesn't necessarily concern historical trends.

Embedded analytics. You may wonder why we've broken out embedded analytics (customer-facing analytics) separately from BI. In practice, analytics provided to customers on a SaaS platform come with a separate set of requirements and complications. Internal BI faces a limited audience and generally presents a limited number of unified views. Access controls are critical but not particularly complicated. Access is managed using a handful of roles and access tiers.

With embedded analytics, the request rate for reports, and the corresponding burden on analytics systems, goes up dramatically; access control is significantly more complicated and critical. Businesses may be serving separate analytics and data to thousands or more customers. Each customer must see their data and only their data. An internal data-access error at a company would likely lead to a procedural review. A data leak between customers would be considered a massive breach of trust, leading to media attention and a significant loss of customers. Minimize your blast radius related to data leaks and security vulnerabilities. Apply tenant- or data-level security within your storage and anywhere there's a possibility of data leakage.

Multitenancy

Many current storage and analytics systems support multitenancy in various ways. Data engineers may choose to house data for many customers in common tables to allow a unified view for internal analytics and ML. This data is presented externally to individual customers through logical views with appropriately defined controls and filters. It is incumbent on data engineers to understand the minutiae of multitenancy in the systems they deploy to ensure absolute data security and isolation.

Machine learning

The emergence and success of ML is one of the most exciting technology revolutions. Once organizations reach a high level of data maturity, they can begin to identify problems amenable to ML and start organizing a practice around it.

The responsibilities of data engineers overlap significantly in analytics and ML, and the boundaries between data engineering, ML engineering, and analytics engineering can be fuzzy. For example, a data engineer may need to support Spark clusters that facilitate analytics pipelines and ML model training. They may also need to provide a system that orchestrates tasks across teams and support metadata and cataloging systems that track data history and lineage. Setting these domains of responsibility and the relevant reporting structures is a critical organizational decision.

The feature store is a recently developed tool that combines data engineering and ML engineering. Feature stores are designed to reduce the operational burden for ML engineers by maintaining feature history and versions, supporting feature sharing among teams, and providing basic operational and orchestration capabilities, such as backfilling. In practice, data engineers are part of the core support team for feature stores to support ML engineering.

Should a data engineer be familiar with ML? It certainly helps. Regardless of the operational boundary between data engineering, ML engineering, business analytics, and so forth, data engineers should maintain operational knowledge about their teams. A good data engineer is conversant in the fundamental ML techniques and related data-processing requirements, the use cases for models within their company, and the responsibilities of the organization's various analytics teams. This helps maintain efficient communication and facilitate collaboration. Ideally, data engineers will build tools in partnership with other teams that neither team can make independently.

This book cannot possibly cover ML in depth. A growing ecosystem of books, videos, articles, and communities is available if you're interested in learning more; we include a few suggestions in "Additional Resources" on page 69.

The following are some considerations for the serving data phase specific to ML:

- Is the data of sufficient quality to perform reliable feature engineering? Quality requirements and assessments are developed in close collaboration with teams consuming the data.
- Is the data discoverable? Can data scientists and ML engineers easily find valuable data?
- Where are the technical and organizational boundaries between data engineering and ML engineering? This organizational question has significant architectural implications.
- Does the dataset properly represent ground truth? Is it unfairly biased?

While ML is exciting, our experience is that companies often prematurely dive into it. Before investing a ton of resources into ML, take the time to build a solid data

foundation. This means setting up the best systems and architecture across the data engineering and ML lifecycle. It's generally best to develop competence in analytics before moving to ML. Many companies have dashed their ML dreams because they undertook initiatives without appropriate foundations.

Reverse ETL

Reverse ETL has long been a practical reality in data, viewed as an antipattern that we didn't like to talk about or dignify with a name. *Reverse ETL* takes processed data from the output side of the data engineering lifecycle and feeds it back into source systems, as shown in Figure 2-6. In reality, this flow is beneficial and often necessary; reverse ETL allows us to take analytics, scored models, etc., and feed these back into production systems or SaaS platforms.

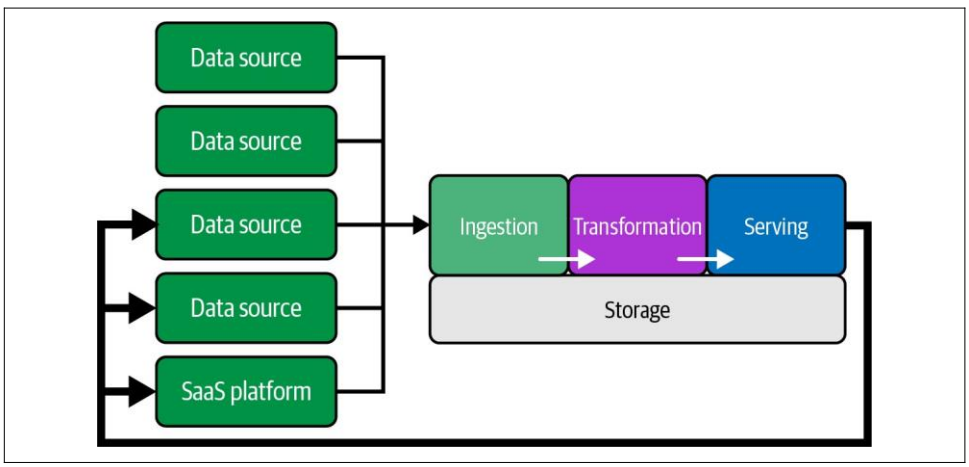


Figure 2-6. Reverse ETL

Marketing analysts might calculate bids in Microsoft Excel by using the data in their data warehouse, and then upload these bids to Google Ads. This process was often entirely manual and primitive.

As we've written this book, several vendors have embraced the concept of reverse ETL and built products around it, such as Hightouch and Census. Reverse ETL remains nascent as a practice, but we suspect that it is here to stay.

Reverse ETL has become especially important as businesses rely increasingly on SaaS and external platforms. For example, companies may want to push specific metrics from their data warehouse to a customer data platform or CRM system. Advertising platforms are another everyday use case, as in the Google Ads example. Expect to see more activity in reverse ETL, with an overlap in both data engineering and ML engineering.

The jury is out on whether the term *reverse ETL* will stick. And the practice may evolve. Some engineers claim that we can eliminate reverse ETL by handling data transformations in an event stream and sending those events back to source systems as needed. Realizing widespread adoption of this pattern across businesses is another matter. The gist is that transformed data will need to be returned to source systems in some manner, ideally with the correct lineage and business process associated with the source system.

Major Undercurrents Across the Data Engineering Lifecycle

Data engineering is rapidly maturing. Whereas prior cycles of data engineering simply focused on the technology layer, the continued abstraction and simplification

of tools and practices have shifted this focus. Data engineering now encompasses far more than tools and technology. The field is now moving up the value chain, incorporating traditional enterprise practices such as data management and cost optimization and newer practices like DataOps.

We've termed these practices *undercurrents*—security, data management, DataOps, data architecture, orchestration, and software engineering—that support every aspect of the data engineering lifecycle (Figure 2-7). In this section, we give a brief overview of these undercurrents and their major components, which you'll see in more detail throughout the book.

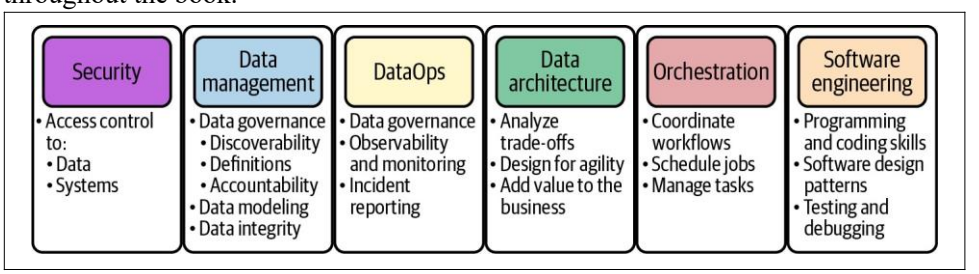


Figure 2-7. The major undercurrents of data engineering

Security

Security must be top of mind for data engineers, and those who ignore it do so at their peril. That's why security is the first undercurrent. Data engineers must understand both data and access security, exercising the principle of least privilege. The principle of least privilege means giving a user or system access to only the essential data and resources to perform an intended function. A common antipattern we see with data engineers with little security experience is to give admin access to all users. This is a catastrophe waiting to happen!

Give users only the access they need to do their jobs today, nothing more. Don't operate from a root shell when you're just looking for visible files with standard user access. When querying tables with a lesser role, don't use the superuser role in a database. Imposing the principle of least privilege on ourselves can prevent accidental damage and keep you in a security-first mindset.

People and organizational structure are always the biggest security vulnerabilities in any company. When we hear about major security breaches in the media, it often turns out that someone in the company ignored basic precautions, fell victim to a phishing attack, or otherwise acted irresponsibly. The first line of defense for data security is to create a culture of security that permeates the organization. All individuals who have access to data must understand their responsibility in protecting the company's sensitive data and its customers.

Data security is also about timing—providing data access to exactly the people and systems that need to access it and *only for the duration necessary to perform their work*. Data should be protected from unwanted visibility, both in flight and at rest, by using encryption, tokenization, data masking, obfuscation, and simple, robust access controls.

Data engineers must be competent security administrators, as security falls in their domain. A data engineer should understand security best practices for the cloud and on prem. Knowledge of user and identity access management (IAM) roles, policies, groups, network security, password policies, and encryption are good places to start.

Throughout the book, we highlight areas where security should be top of mind in the data engineering lifecycle. You can also gain more detailed insights into security in Chapter 10.

Data Management

You probably think that data management sounds very…corporate. "Old school" data management practices make their way into data and ML engineering. What's old is new again. Data management has been around for decades but didn't get a lot of traction in data engineering until recently. Data tools are becoming simpler, and there is less complexity for data engineers to manage. As a result, the data engineer moves up the value chain toward the next rung of best practices. Data best practices once reserved for huge companies—data governance, master data management, dataquality management, metadata management—are now filtering down to companies of all sizes and maturity levels. As we like to say, data engineering is becoming "enterprisey." This is ultimately a great thing!

The Data Management Association International (DAMA) *Data Management Body of Knowledge* (*DMBOK*), which we consider to be the definitive book for enterprise data management, offers this definition:

> Data management is the development, execution, and supervision of plans, policies, programs, and practices that deliver, control, protect, and enhance the value of data and information assets throughout their lifecycle.

That's a bit lengthy, so let's look at how it ties to data engineering. Data engineers manage the data lifecycle, and data management encompasses the set of best practices that data engineers will use to accomplish this task, both technically and strategically. Without a framework for managing data, data engineers are simply technicians operating in a vacuum. Data engineers need a broader perspective of data's utility across the organization, from the source systems to the C-suite, and everywhere in between.

Why is data management important? Data management demonstrates that data is vital to daily operations, just as businesses view financial resources, finished goods, or real estate as assets. Data management practices form a cohesive framework that everyone can adopt to ensure that the organization gets value from data and handles it appropriately.

Data management has quite a few facets, including the following:

- Data governance, including discoverability and accountability
- Data modeling and design
- Data lineage
- Storage and operations
- Data integration and interoperability
- Data lifecycle management
- Data systems for advanced analytics and ML
- Ethics and privacy

While this book is in no way an exhaustive resource on data management, let's briefly cover some salient points from each area as they relate to data engineering.

Data governance

According to *Data Governance: The Definitive Guide,* "Data governance is, first and foremost, a data management function to ensure the quality, integrity, security, and usability of the data collected by an organization."[13]

We can expand on that definition and say that data governance engages people, processes, and technologies to maximize data value across an organization while protecting data with appropriate security controls. Effective data governance is developed with intention and supported by the organization. When data governance is accidental and haphazard, the side effects can range from untrusted data to security breaches and everything in between. Being intentional about data governance will maximize the organization's data capabilities and the value generated from data. It will also (hopefully) keep a company out of the headlines for questionable or downright reckless data practices.

Think of the typical example of data governance being done poorly. A business analyst gets a request for a report but doesn't know what data to use to answer the question. They may spend hours digging through dozens of tables in a transactional database,

[13] Evren Eryurek et al., *Data Governance: The Definitive Guide* (Sebastopol, CA: O'Reilly, 2021), 1, *https://oreil.ly/LFT4d*.

wildly guessing at which fields might be useful. The analyst compiles a "directionally correct" report but isn't entirely sure that the report's underlying data is accurate or sound. The recipient of the report also questions the validity of the data. The integrity of the analyst—and of all data in the company's systems—is called into question. The company is confused about its performance, making business planning impossible.

Data governance is a foundation for data-driven business practices and a missioncritical part of the data engineering lifecycle. When data governance is practiced well, people, processes, and technologies align to treat data as a key business driver; if data issues occur, they are promptly handled.

The core categories of data governance are discoverability, security, and accountability.[14] Within these core categories are subcategories, such as data quality, metadata, and privacy. Let's look at each core category in turn.

Discoverability. In a data-driven company, data must be available and discoverable. End users should have quick and reliable access to the data they need to do their jobs. They should know where the data comes from, how it relates to other data, and what the data means.

Some key areas of data discoverability include metadata management and master data management. Let's briefly describe these areas.

Metadata. *Metadata* is "data about data," and it underpins every section of the data engineering lifecycle. Metadata is exactly the data needed to make data discoverable and governable.

We divide metadata into two major categories: autogenerated and human generated. Modern data engineering revolves around automation, but metadata collection is often manual and error prone.

Technology can assist with this process, removing much of the error-prone work of manual metadata collection. We're seeing a proliferation of data catalogs, data-lineage tracking systems, and metadata management tools. Tools can crawl databases to look for relationships and monitor data pipelines to track where data comes from and where it goes. A low-fidelity manual approach uses an internally led effort where various stakeholders crowdsource metadata collection within the organization. These data management tools are covered in depth throughout the book, as they undercut much of the data engineering lifecycle.

Metadata becomes a byproduct of data and data processes. However, key challenges remain. In particular, interoperability and standards are still lacking. Metadata tools are

[14] Eryurek, *Data Governance*, 5.

only as good as their connectors to data systems and their ability to share metadata. In addition, automated metadata tools should not entirely take humans out of the loop.

Data has a social element; each organization accumulates social capital and knowledge around processes, datasets, and pipelines. Human-oriented metadata systems focus on the social aspect of metadata. This is something that Airbnb has emphasized in its various blog posts on data tools, particularly its original Dataportal concept.[15]Such tools should provide a place to disclose data owners, data consumers, and domain experts. Documentation and internal wiki tools provide a key foundation for metadata management, but these tools should also integrate with automated data cataloging. For example, data-scanning tools can generate wiki pages with links to relevant data objects.

Once metadata systems and processes exist, data engineers can consume metadata in useful ways. Metadata becomes a foundation for designing pipelines and managing data throughout the lifecycle.

DMBOK identifies four main categories of metadata that are useful to data engineers:

- Business metadata
- Technical metadata
- Operational metadata
- Reference metadata

Let's briefly describe each category of metadata.

Business metadata relates to the way data is used in the business, including business and data definitions, data rules and logic, how and where data is used, and the data owner(s).

A data engineer uses business metadata to answer nontechnical questions about who, what, where, and how. For example, a data engineer may be tasked with creating a data pipeline for customer sales analysis. But what is a customer? Is it someone who's purchased in the last 90 days? Or someone who's purchased at any time the business has been open? A data engineer would use the correct data to refer to business metadata (data dictionary or data catalog) to look up how a "customer" is defined. Business metadata provides a data engineer with the right context and definitions to properly use data.

Technical metadata describes the data created and used by systems across the data engineering lifecycle. It includes the data model and schema, data lineage, field

[15] Chris Williams et al., "Democratizing Data at Airbnb," *The Airbnb Tech Blog*, May 12, 2017, *https://oreil.ly/dM332*.

mappings, and pipeline workflows. A data engineer uses technical metadata to create, connect, and monitor various systems across the data engineering lifecycle.

Here are some common types of technical metadata that a data engineer will use:

- Pipeline metadata (often produced in orchestration systems)
- Data lineage
- Schema

Orchestration is a central hub that coordinates workflow across various systems. *Pipeline metadata* captured in orchestration systems provides details of the workflow schedule, system and data dependencies, configurations, connection details, and much more.

Data-lineage metadata tracks the origin and changes to data, and its dependencies, over time. As data flows through the data engineering lifecycle, it evolves through transformations and combinations with other data. Data lineage provides an audit trail of data's evolution as it moves through various systems and workflows.

Schema metadata describes the structure of data stored in a system such as a database, a data warehouse, a data lake, or a filesystem; it is one of the key differentiators across different storage systems. Object stores, for example, don't manage schema metadata; instead, this must be managed in a *metastore*. On the other hand, cloud data warehouses manage schema metadata internally.

These are just a few examples of technical metadata that a data engineer should know about. This is not a complete list, and we cover additional aspects of technical metadata throughout the book.

Operational metadata describes the operational results of various systems and includes statistics about processes, job IDs, application runtime logs, data used in a process, and error logs. A data engineer uses operational metadata to determine whether a process succeeded or failed and the data involved in the process.

Orchestration systems can provide a limited picture of operational metadata, but the latter still tends to be scattered across many systems. A need for better-quality operational metadata, and better metadata management, is a major motivation for next-generation orchestration and metadata management systems.

Reference metadata is data used to classify other data. This is also referred to as *lookup data*. Standard examples of reference data are internal codes, geographic codes, units of measurement, and internal calendar standards. Note that much of reference data is fully managed internally, but items such as geographic codes might come from

standard external references. Reference data is essentially a standard for interpreting other data, so if it changes, this change happens slowly over time.

Data accountability. *Data accountability* means assigning an individual to govern a portion of data. The responsible person then coordinates the governance activities of other stakeholders. Managing data quality is tough if no one is accountable for the data in question.

Note that people accountable for data need not be data engineers. The accountable person might be a software engineer or product manager, or serve in another role. In addition, the responsible person generally doesn't have all the resources necessary to maintain data quality. Instead, they coordinate with all people who touch the data, including data engineers.

Data accountability can happen at various levels; accountability can happen at the level of a table or a log stream but could be as fine-grained as a single field entity that occurs across many tables. An individual may be accountable for managing a customer ID across many systems. For enterprise data management, a data domain is the set of all possible values that can occur for a given field type, such as in this ID example. This may seem excessively bureaucratic and meticulous, but it can significantly affect data quality.

Data quality.

> Can I trust this data?
>
> —Everyone in the business

Data quality is the optimization of data toward the desired state and orbits the question, "What do you get compared with what you expect?" Data should conform to the expectations in the business metadata. Does the data match the definition agreed upon by the business?

A data engineer ensures data quality across the entire data engineering lifecycle. This involves performing data-quality tests, and ensuring data conformance to schema expectations, data completeness, and precision.

According to *Data Governance: The Definitive Guide*, data quality is defined by three main characteristics:[16]

Accuracy
: Is the collected data factually correct? Are there duplicate values? Are the numeric values accurate?

[16] Eryurek, *Data Governance*, 113.

Completeness

Are the records complete? Do all required fields contain valid values?

Timeliness

Are records available in a timely fashion?

Each of these characteristics is quite nuanced. For example, how do we think about bots and web scrapers when dealing with web event data? If we intend to analyze the customer journey, we must have a process that lets us separate humans from machine-generated traffic. Any bot-generated events misclassified as *human* present data accuracy issues, and vice versa.

A variety of interesting problems arise concerning completeness and timeliness. In the Google paper introducing the Dataflow model, the authors give the example of an offline video platform that displays ads.[17] The platform downloads video and ads while a connection is present, allows the user to watch these while offline, and then uploads ad view data once a connection is present again. This data may arrive late, well after the ads are watched. How does the platform handle billing for the ads?

Fundamentally, this problem can't be solved by purely technical means. Rather, engineers will need to determine their standards for late-arriving data and enforce these uniformly, possibly with the help of various technology tools.

Master Data Management

Master data is data about business entities such as employees, customers, products, and locations. As organizations grow larger and more complex through organic growth and acquisitions, and collaborate with other businesses, maintaining a consistent picture of entities and identities becomes more and more challenging.

Master data management (MDM) is the practice of building consistent entity definitions known as *golden records*. Golden records harmonize entity data across an organization and with its partners. MDM is a business operations process facilitated by building and deploying technology tools. For example, an MDM team might determine a standard format for addresses, and then work with data engineers to build an API to return consistent addresses and a system that uses address data to match customer records across company divisions.

MDM reaches across the full data cycle into operational databases. It may fall directly under the purview of data engineering but is often the assigned responsibility of a

[17] Tyler Akidau et al., "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing," *Proceedings of the VLDB Endowment* 8 (2015): 1792–1803, *https://oreil.ly/Z6XYy*.

dedicated team that works across the organization. Even if they don't own MDM, data engineers must always be aware of it, as they might collaborate on MDM initiatives.

Data quality sits across the boundary of human and technology problems. Data engineers need robust processes to collect actionable human feedback on data quality and use technology tools to detect quality issues preemptively before downstream users ever see them. We cover these collection processes in the appropriate chapters throughout this book.

Data modeling and design

To derive business insights from data, through business analytics and data science, the data must be in a usable form. The process for converting data into a usable form is known as *data modeling and design*. Whereas we traditionally think of data modeling as a problem for database administrators (DBAs) and ETL developers, data modeling can happen almost anywhere in an organization. Firmware engineers develop the data format of a record for an IoT device, or web application developers design the JSON response to an API call or a MySQL table schema—these are all instances of data modeling and design.

Data modeling has become more challenging because of the variety of new data sources and use cases. For instance, strict normalization doesn't work well with event data. Fortunately, a new generation of data tools increases the flexibility of data models, while retaining logical separations of measures, dimensions, attributes, and hierarchies. Cloud data warehouses support the ingestion of enormous quantities of denormalized and semistructured data, while still supporting common data modeling patterns, such as Kimball, Inmon, and Data Vault. Data processing frameworks such as Spark can ingest a whole spectrum of data, from flat structured relational records to raw unstructured text. We discuss these data modeling and transformation patterns in greater detail in Chapter 8.

With the wide variety of data that engineers must cope with, there is a temptation to throw up our hands and give up on data modeling. This is a terrible idea with harrowing consequences, made evident when people murmur of the write once, read never (WORN) access pattern or refer to a *data swamp*. Data engineers need to understand modeling best practices as well as develop the flexibility to apply the appropriate level and type of modeling to the data source and use case.

Data lineage

As data moves through its lifecycle, how do you know what system affected the data or what the data is composed of as it gets passed around and transformed? *Data lineage* describes the recording of an audit trail of data through its lifecycle, tracking both the systems that process the data and the upstream data it depends on.

Data lineage helps with error tracking, accountability, and debugging of data and the systems that process it. It has the obvious benefit of giving an audit trail for the data lifecycle and helps with compliance. For example, if a user would like their data deleted from your systems, having lineage for that data lets you know where that data is stored and its dependencies.

Data lineage has been around for a long time in larger companies with strict compliance standards. However, it's now being more widely adopted in smaller companies as data management becomes mainstream. We also note that Andy Petrella's concept of Data Observability Driven Development (DODD) is closely related to data lineage. DODD observes data all along its lineage. This process is applied during development, testing, and finally production to deliver quality and conformity to expectations.

Data integration and interoperability

Data integration and interoperability is the process of integrating data across tools and processes. As we move away from a single-stack approach to analytics and toward a heterogeneous cloud environment in which various tools process data on demand, integration and interoperability occupy an ever-widening swath of the data engineer's job.

Increasingly, integration happens through general-purpose APIs rather than custom database connections. For example, a data pipeline might pull data from the Salesforce API, store it to Amazon S3, call the Snowflake API to load it into a table, call the API again to run a query, and then export the results to S3 where Spark can consume them.

All of this activity can be managed with relatively simple Python code that talks to data systems rather than handling data directly. While the complexity of interacting with data systems has decreased, the number of systems and the complexity of pipelines has dramatically increased. Engineers starting from scratch quickly outgrow the capabilities of bespoke scripting and stumble into the need for *orchestration.* Orchestration is one of our undercurrents, and we discuss it in detail in "Orchestration" on page 64.

Data lifecycle management

The advent of data lakes encouraged organizations to ignore data archival and destruction. Why discard data when you can simply add more storage ad infinitum? Two changes have encouraged engineers to pay more attention to what happens at the end of the data engineering lifecycle.

First, data is increasingly stored in the cloud. This means we have pay-as-you-go storage costs instead of large up-front capital expenditures for an on-premises data lake. When every byte shows up on a monthly AWS statement, CFOs see opportunities

for savings. Cloud environments make data archival a relatively straightforward process. Major cloud vendors offer archival-specific object storage classes that allow long-term data retention at an extremely low cost, assuming very infrequent access (it should be noted that data retrieval isn't so cheap, but that's for another conversation). These storage classes also support extra policy controls to prevent accidental or deliberate deletion of critical archives.

Second, privacy and data retention laws such as the GDPR and the CCPA require data engineers to actively manage data destruction to respect users' "right to be forgotten." Data engineers must know what consumer data they retain and must have procedures to destroy data in response to requests and compliance requirements.

Data destruction is straightforward in a cloud data warehouse. SQL semantics allow deletion of rows conforming to a `where` clause. Data destruction was more challenging in data lakes, where write-once, read-many was the default storage pattern. Tools such as Hive ACID and Delta Lake allow easy management of deletion transactions at scale. New generations of metadata management, data lineage, and cataloging tools will also streamline the end of the data engineering lifecycle.

Ethics and privacy

The last several years of data breaches, misinformation, and mishandling of data make one thing clear: data impacts people. Data used to live in the Wild West, freely collected and traded like baseball cards. Those days are long gone. Whereas data's ethical and privacy implications were once considered nice to have, like security, they're now central to the general data lifecycle. Data engineers need to do the right thing when no one else is watching, because everyone will be watching someday.[18]We hope that more organizations will encourage a culture of good data ethics and privacy.

How do ethics and privacy impact the data engineering lifecycle? Data engineers need to ensure that datasets mask personally identifiable information (PII) and other sensitive information; bias can be identified and tracked in datasets as they are transformed. Regulatory requirements and compliance penalties are only growing. Ensure that your data assets are compliant with a growing number of data regulations, such as GDPR and CCPA. Please take this seriously. We offer tips throughout the book to ensure that you're baking ethics and privacy into the data engineering lifecycle.

[18] We espouse the notion that ethical behavior is doing the right thing when no one is watching, an idea that occurs in the writings of C. S. Lewis, Charles Marshall, and many other authors.

DataOps

DataOps maps the best practices of Agile methodology, DevOps, and statistical process control (SPC) to data. Whereas DevOps aims to improve the release and quality of software products, DataOps does the same thing for data products.

Data products differ from software products because of the way data is used. A software product provides specific functionality and technical features for end users. By contrast, a data product is built around sound business logic and metrics, whose users make decisions or build models that perform automated actions. A data engineer must understand both the technical aspects of building software products and the business logic, quality, and metrics that will create excellent data products.

Like DevOps, DataOps borrows much from lean manufacturing and supply chain management, mixing people, processes, and technology to reduce time to value. As Data Kitchen (experts in DataOps) describes it:[19]

> DataOps is a collection of technical practices, workflows, cultural norms, and architectural patterns that enable:
>
> - Rapid innovation and experimentation delivering new insights to customers with increasing velocity
> - Extremely high data quality and very low error rates
> - Collaboration across complex arrays of people, technology, and environments
> - Clear measurement, monitoring, and transparency of results

Lean practices (such as lead time reduction and minimizing defects) and the resulting improvements to quality and productivity are things we are glad to see gaining momentum both in software and data operations.

First and foremost, DataOps is a set of cultural habits; the data engineering team needs to adopt a cycle of communicating and collaborating with the business, breaking down silos, continuously learning from successes and mistakes, and rapid iteration. Only when these cultural habits are set in place can the team get the best results from technology and tools.

Depending on a company's data maturity, a data engineer has some options to build DataOps into the fabric of the overall data engineering lifecycle. If the company has no preexisting data infrastructure or practices, DataOps is very much a greenfield opportunity that can be baked in from day one. With an existing project or infrastructure that lacks DataOps, a data engineer can begin adding DataOps into workflows. We suggest first starting with observability and monitoring to get a window

[19] "What Is DataOps," DataKitchen FAQ page, accessed May 5, 2022, *https://oreil.ly/Ns06w*.

into the performance of a system, then adding in automation and incident response. A data engineer may work alongside an existing DataOps team to improve the data engineering lifecycle in a data-mature company. In all cases, a data engineer must be aware of the philosophy and technical aspects of DataOps.

DataOps has three core technical elements: automation, monitoring and observability, and incident response (Figure 2-8). Let's look at each of these pieces and how they relate to the data engineering lifecycle.

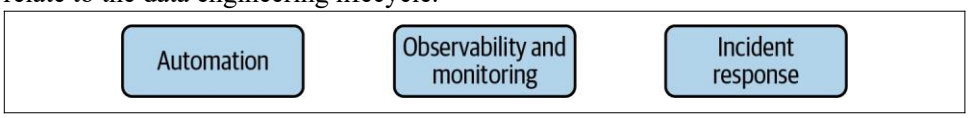


Figure 2-8. The three pillars of DataOps

Automation

Automation enables reliability and consistency in the DataOps process and allows data engineers to quickly deploy new product features and improvements to existing workflows. DataOps automation has a similar framework and workflow to DevOps, consisting of change management (environment, code, and data version control), continuous integration/continuous deployment (CI/CD), and configuration as code. Like DevOps, DataOps practices monitor and maintain the reliability of technology and systems (data pipelines, orchestration, etc.), with the added dimension of checking for data quality, data/model drift, metadata integrity, and more.

Let's briefly discuss the evolution of DataOps automation within a hypothetical organization. An organization with a low level of DataOps maturity often attempts to schedule multiple stages of data transformation processes using cron jobs. This works well for a while. As data pipelines become more complicated, several things are likely to happen. If the cron jobs are hosted on a cloud instance, the instance may have an operational problem, causing the jobs to stop running unexpectedly. As the spacing between jobs becomes tighter, a job will eventually run long, causing a subsequent job to fail or produce stale data. Engineers may not be aware of job failures until they hear from analysts that their reports are out-of-date.

As the organization's data maturity grows, data engineers will typically adopt an orchestration framework, perhaps Airflow or Dagster. Data engineers are aware that Airflow presents an operational burden, but the benefits of orchestration eventually outweigh the complexity. Engineers will gradually migrate their cron jobs to Airflow jobs. Now, dependencies are checked before jobs run. More transformation jobs can be packed into a given time because each job can start as soon as upstream data is ready rather than at a fixed, predetermined time.

The data engineering team still has room for operational improvements. A data scientist eventually deploys a broken DAG, bringing down the Airflow web server and leaving the data team operationally blind. After enough such headaches, the data engineering team members realize that they need to stop allowing manual DAG deployments. In their next phase of operational maturity, they adopt automated DAG deployment. DAGs are tested before deployment, and monitoring processes ensure that the new DAGs start running properly. In addition, data engineers block the deployment of new Python dependencies until installation is validated. After automation is adopted, the data team is much happier and experiences far fewer headaches.

One of the tenets of the DataOps Manifesto is "Embrace change." This does not mean change for the sake of change but rather goal-oriented change. At each stage of our automation journey, opportunities exist for operational improvement. Even at the high level of maturity that we've described here, further room for improvement remains. Engineers might embrace a next-generation orchestration framework that builds in better metadata capabilities. Or they might try to develop a framework that builds DAGs automatically based on data-lineage specifications. The main point is that engineers constantly seek to implement improvements in automation that will reduce their workload and increase the value that they deliver to the business.

Observability and monitoring

As we tell our clients, "Data is a silent killer." We've seen countless examples of bad data lingering in reports for months or years. Executives may make key decisions from this bad data, discovering the error only much later. The outcomes are usually bad and sometimes catastrophic for the business. Initiatives are undermined and destroyed, years of work wasted. In some of the worst cases, bad data may lead companies to financial ruin.

Another horror story occurs when the systems that create the data for reports randomly stop working, resulting in reports being delayed by several days. The data team doesn't know until they're asked by stakeholders why reports are late or producing stale information. Eventually, various stakeholders lose trust in the capabilities of the core data team and start their own splinter teams. The result is many different unstable systems, inconsistent reports, and silos.

If you're not observing and monitoring your data and the systems that produce the data, you're inevitably going to experience your own data horror story. Observability, monitoring, logging, alerting, and tracing are all critical to getting ahead of any problems along the data engineering lifecycle. We recommend you incorporate SPC to understand whether events being monitored are out of line and which incidents are worth responding to.

Petrella's DODD method mentioned previously in this chapter provides an excellent framework for thinking about data observability. DODD is much like test-driven development (TDD) in software engineering:[20]

> The purpose of DODD is to give everyone involved in the data chain visibility into the data and data applications so that everyone involved in the data value chain has the ability to identify changes to the data or data applications at every step—from ingestion to transformation to analysis—to help troubleshoot or prevent data issues. DODD focuses on making data observability a first-class consideration in the data engineering lifecycle.

We cover many aspects of monitoring and observability throughout the data engineering lifecycle in later chapters.

Incident response

A high-functioning data team using DataOps will be able to ship new data products quickly. But mistakes will inevitably happen. A system may have downtime, a new data model may break downstream reports, an ML model may become stale and provide bad predictions—countless problems can interrupt the data engineering lifecycle. *Incident response* is about using the automation and observability capabilities mentioned previously to rapidly identify root causes of an incident and resolve it as reliably and quickly as possible.

Incident response isn't just about technology and tools, though these are beneficial; it's also about open and blameless communication, both on the data engineering team and across the organization. As Werner Vogels, CTO of Amazon Web Services, is famous for saying, "Everything breaks all the time." Data engineers must be prepared for a disaster and ready to respond as swiftly and efficiently as possible.

Data engineers should proactively find issues before the business reports them. Failure happens, and when the stakeholders or end users see problems, they will present them. They will be unhappy to do so. The feeling is different when they go to raise those issues to a team and see that they are actively being worked on to resolve already. Which team's state would you trust more as an end user? Trust takes a long time to build and can be lost in minutes. Incident response is as much about retroactively responding to incidents as proactively addressing them before they happen.

[20] Andy Petrella, "Data Observability Driven Development: The Perfect Analogy for Beginners," Kensu, accessed May 5, 2022, *https://oreil.ly/MxvSX*.

DataOps summary

At this point, DataOps is still a work in progress. Practitioners have done a good job of adapting DevOps principles to the data domain and mapping out an initial vision through the DataOps Manifesto and other resources. Data engineers would do well to make DataOps practices a high priority in all of their work. The up-front effort will see a significant long-term payoff through faster delivery of products, better reliability and accuracy of data, and greater overall value for the business.

The state of operations in data engineering is still quite immature compared with software engineering. Many data engineering tools, especially legacy monoliths, are not automation-first. A recent movement has arisen to adopt automation best practices across the data engineering lifecycle. Tools like Airflow have paved the way for a new generation of automation and data management tools. The general practices we describe for DataOps are aspirational, and we suggest companies try to adopt them to the fullest extent possible, given the tools and knowledge available today.

Data Architecture

A data architecture reflects the current and future state of data systems that support an organization's long-term data needs and strategy. Because an organization's data requirements will likely change rapidly, and new tools and practices seem to arrive on a near-daily basis, data engineers must understand good data architecture. Chapter 3 covers data architecture in depth, but we want to highlight here that data architecture is an undercurrent of the data engineering lifecycle.

A data engineer should first understand the needs of the business and gather requirements for new use cases. Next, a data engineer needs to translate those requirements to design new ways to capture and serve data, balanced for cost and operational simplicity. This means knowing the trade-offs with design patterns, technologies, and tools in source systems, ingestion, storage, transformation, and serving data.

This doesn't imply that a data engineer is a data architect, as these are typically two separate roles. If a data engineer works alongside a data architect, the data engineer should be able to deliver on the data architect's designs and provide architectural feedback.

Orchestration

> We think that orchestration matters because we view it as really the center of gravity of both the data platform as well as the data lifecycle, the software development lifecycle as it comes to data.

—Nick Schrock, founder of Elementl[21]

Orchestration is not only a central DataOps process, but also a critical part of the engineering and deployment flow for data jobs. So, what is orchestration?

Orchestration is the process of coordinating many jobs to run as quickly and efficiently as possible on a scheduled cadence. For instance, people often refer to orchestration tools like Apache Airflow as *schedulers*. This isn't quite accurate. A pure scheduler, such as cron, is aware only of time; an orchestration engine builds in metadata on job dependencies, generally in the form of a directed acyclic graph (DAG). The DAG can be run once or scheduled to run at a fixed interval of daily, weekly, every hour, every five minutes, etc.

As we discuss orchestration throughout this book, we assume that an orchestration system stays online with high availability. This allows the orchestration system to sense and monitor constantly without human intervention and run new jobs anytime they are deployed. An orchestration system monitors jobs that it manages and kicks off new tasks as internal DAG dependencies are completed. It can also monitor external systems and tools to watch for data to arrive and criteria to be met. When certain conditions go out of bounds, the system also sets error conditions and sends alerts through email or other channels. You might set an expected completion time of 10 a.m. for overnight daily data pipelines. If jobs are not done by this time, alerts go out to data engineers and consumers.

Orchestration systems also build job history capabilities, visualization, and alerting. Advanced orchestration engines can backfill new DAGs or individual tasks as they are added to a DAG. They also support dependencies over a time range. For example, a monthly reporting job might check that an ETL job has been completed for the full month before starting.

Orchestration has long been a key capability for data processing but was not often top of mind nor accessible to anyone except the largest companies. Enterprises used various tools to manage job flows, but these were expensive, out of reach of small startups, and generally not extensible. Apache Oozie was extremely popular in the 2010s, but it was designed to work within a Hadoop cluster and was difficult to use in a more heterogeneous environment. Facebook developed Dataswarm for internal use in the late 2000s; this inspired popular tools such as Airflow, introduced by Airbnb in 2014.

Airflow was open source from its inception and was widely adopted. It was written in Python, making it highly extensible to almost any use case imaginable. While many

[21] Ternary Data, "An Introduction to Dagster: The Orchestrator for the Full Data Lifecycle - UDEM June 2021," YouTube video, 1:09:40, *https://oreil.ly/HyGMh*.

other interesting open source orchestration projects exist, such as Luigi and Conductor, Airflow is arguably the mindshare leader for the time being. Airflow arrived just as data processing was becoming more abstract and accessible, and engineers were increasingly interested in coordinating complex flows across multiple processors and storage systems, especially in cloud environments.

At this writing, several nascent open source projects aim to mimic the best elements of Airflow's core design while improving on it in key areas. Some of the most interesting examples are Prefect and Dagster, which aim to improve the portability and testability of DAGs to allow engineers to move from local development to production more easily. Argo is an orchestration engine built around Kubernetes primitives; Metaflow is an open source project out of Netflix that aims to improve data science orchestration.

We must point out that orchestration is strictly a batch concept. The streaming alternative to orchestrated task DAGs is the streaming DAG. Streaming DAGs remain challenging to build and maintain, but next-generation streaming platforms such as Pulsar aim to dramatically reduce the engineering and operational burden. We talk more about these developments in Chapter 8.

Software Engineering

Software engineering has always been a central skill for data engineers. In the early days of contemporary data engineering (2000–2010), data engineers worked on lowlevel frameworks and wrote MapReduce jobs in C, C++, and Java. At the peak of the big data era (the mid-2010s), engineers started using frameworks that abstracted away these low-level details.

This abstraction continues today. Cloud data warehouses support powerful transformations using SQL semantics; tools like Spark have become more user-friendly, transitioning away from low-level coding details and toward easy-to-use dataframes. Despite this abstraction, software engineering is still critical to data engineering. We want to briefly discuss a few common areas of software engineering that apply to the data engineering lifecycle.

Core data processing code

Though it has become more abstract and easier to manage, core data processing code still needs to be written, and it appears throughout the data engineering lifecycle. Whether in ingestion, transformation, or data serving, data engineers need to be highly proficient and productive in frameworks and languages such as Spark, SQL, or Beam; we reject the notion that SQL is not code.

It's also imperative that a data engineer understand proper code-testing methodologies, such as unit, regression, integration, end-to-end, and smoke.

Development of open source frameworks

Many data engineers are heavily involved in developing open source frameworks. They adopt these frameworks to solve specific problems in the data engineering lifecycle, and then continue developing the framework code to improve the tools for their use cases and contribute back to the community.

In the big data era, we saw a Cambrian explosion of data-processing frameworks inside the Hadoop ecosystem. These tools primarily focused on transforming and serving parts of the data engineering lifecycle. Data engineering tool speciation has not ceased or slowed down, but the emphasis has shifted up the ladder of abstraction, away from direct data processing. This new generation of open source tools assists engineers in managing, enhancing, connecting, optimizing, and monitoring data.

For example, Airflow dominated the orchestration space from 2015 until the early 2020s. Now, a new batch of open source competitors (including Prefect, Dagster, and Metaflow) has sprung up to fix perceived limitations of Airflow, providing better metadata handling, portability, and dependency management. Where the future of orchestration goes is anyone's guess.

Before data engineers begin engineering new internal tools, they would do well to survey the landscape of publicly available tools. Keep an eye on the total cost of ownership (TCO) and opportunity cost associated with implementing a tool. There is a good chance that an open source project already exists to address the problem they're looking to solve, and they would do well to collaborate rather than reinventing the wheel.

Streaming

Streaming data processing is inherently more complicated than batch, and the tools and paradigms are arguably less mature. As streaming data becomes more pervasive in every stage of the data engineering lifecycle, data engineers face interesting software engineering problems.

For instance, data processing tasks such as joins that we take for granted in the batch processing world often become more complicated in real time, requiring more complex software engineering. Engineers must also write code to apply a variety of *windowing* methods. Windowing allows real-time systems to calculate valuable metrics such as trailing statistics. Engineers have many frameworks to choose from, including various function platforms (OpenFaaS, AWS Lambda, Google Cloud Functions) for handling individual events or dedicated stream processors (Spark, Beam, Flink, or Pulsar) for analyzing streams to support reporting and real-time actions.

Infrastructure as code

Infrastructure as code (IaC) applies software engineering practices to the configuration and management of infrastructure. The infrastructure management burden of the big data era has decreased as companies have migrated to managed big data systems—such as Databricks and Amazon Elastic MapReduce (EMR)—and cloud data warehouses. When data engineers have to manage their infrastructure in a cloud environment, they increasingly do this through IaC frameworks rather than manually spinning up instances and installing software. Several general-purpose and cloudplatform-specific frameworks allow automated infrastructure deployment based on a set of specifications. Many of these frameworks can manage cloud services as well as infrastructure. There is also a notion of IaC with containers and Kubernetes, using tools like Helm.

These practices are a vital part of DevOps, allowing version control and repeatability of deployments. Naturally, these capabilities are vital throughout the data engineering lifecycle, especially as we adopt DataOps practices.

Pipelines as code

Pipelines as code is the core concept of present-day orchestration systems, which touch every stage of the data engineering lifecycle. Data engineers use code (typically Python) to declare data tasks and dependencies among them. The orchestration engine interprets these instructions to run steps using available resources.

General-purpose problem solving

In practice, regardless of which high-level tools they adopt, data engineers will run into corner cases throughout the data engineering lifecycle that require them to solve problems outside the boundaries of their chosen tools and to write custom code. When using frameworks like Fivetran, Airbyte, or Matillion, data engineers will encounter data sources without existing connectors and need to write something custom. They should be proficient in software engineering to understand APIs, pull and transform data, handle exceptions, and so forth.

Conclusion

Most discussions we've seen in the past about data engineering involve technologies but miss the bigger picture of data lifecycle management. As technologies become more abstract and do more heavy lifting, a data engineer has the opportunity to think and act on a higher level. The data engineering lifecycle, supported by its undercurrents, is an extremely useful mental model for organizing the work of data engineering.

We break the data engineering lifecycle into the following stages:

- Generation
- Storage
- Ingestion
- Transformation
- Serving data

Several themes cut across the data engineering lifecycle as well. These are the undercurrents of the data engineering lifecycle. At a high level, the undercurrents are as follows:

- Security
- Data management
- DataOps
- Data architecture
- Orchestration
- Software engineering

A data engineer has several top-level goals across the data lifecycle: produce optimum ROI and reduce costs (financial and opportunity), reduce risk (security, data quality), and maximize data value and utility.

The next two chapters discuss how these elements impact good architecture design, along with choosing the right technologies. If you feel comfortable with these two topics, feel free to skip ahead to Part II, where we cover each of the stages of the data engineering lifecycle.

Additional Resources

- "A Comparison of Data Processing Frameworks" by Ludovic Santos
- DAMA International website
- "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing" by Tyler Akidau et al.
- "Data Processing" Wikipedia page
- "Data Transformation" Wikipedia page
- "Democratizing Data at Airbnb" by Chris Williams et al.
- "Five Steps to Begin Collecting the Value of Your Data" Lean-Data web page
- "Getting Started with DevOps Automation" by Jared Murrell
- "Incident Management in the Age of DevOps" Atlassian web page
- "An Introduction to Dagster: The Orchestrator for the Full Data Lifecycle" video by Nick Schrock
- "Is DevOps Related to DataOps?" by Carol Jang and Jove Kuang
- "The Seven Stages of Effective Incident Response" Atlassian web page
- "Staying Ahead of Debt" by Etai Mizrahi
- "What Is Metadata" by Michelle Knight

CHAPTER 3

Designing Good Data Architecture

Good data architecture provides seamless capabilities across every step of the data lifecycle and undercurrent. We'll begin by defining *data architecture* and then discuss components and considerations. We'll then touch on specific batch patterns (data warehouses, data lakes), streaming patterns, and patterns that unify batch and streaming. Throughout, we'll emphasize leveraging the capabilities of the cloud to deliver scalability, availability, and reliability.

What Is Data Architecture?

Successful data engineering is built upon rock-solid data architecture. This chapter aims to review a few popular architecture approaches and frameworks, and then craft our opinionated definition of what makes "good" data architecture. Indeed, we won't make everyone happy. Still, we will lay out a pragmatic, domain-specific, working definition for *data architecture* that we think will work for companies of vastly different scales, business processes, and needs.

What is data architecture? When you stop to unpack it, the topic becomes a bit murky; researching data architecture yields many inconsistent and often outdated definitions. It's a lot like when we defined *data engineering* in Chapter 1—there's no consensus. In a field that is constantly changing, this is to be expected. So what do we mean by *data architecture* for the purposes of this book? Before defining the term, it's essential to understand the context in which it sits. Let's briefly cover enterprise architecture, which will frame our definition of data architecture.

Enterprise Architecture Defined

Enterprise architecture has many subsets, including business, technical, application, and data (Figure 3-1). As such, many frameworks and resources are devoted to enterprise architecture. In truth, architecture is a surprisingly controversial topic.

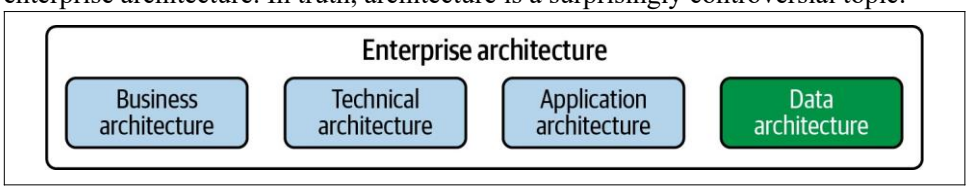


Figure 3-1. Data architecture is a subset of enterprise architecture

The term *enterprise* gets mixed reactions. It brings to mind sterile corporate offices, command-and-control/waterfall planning, stagnant business cultures, and empty catchphrases. Even so, we can learn some things here.

Before we define and describe *enterprise architecture*, let's unpack this term. Let's look at how enterprise architecture is defined by some significant thought leaders: TOGAF, Gartner, and EABOK.

TOGAF's definition

TOGAF is *The Open Group Architecture Framework*, a standard of The Open Group. It's touted as the most widely used architecture framework today. Here's the TOGAF definition:[22]

> The term "enterprise" in the context of "enterprise architecture" can denote an entire enterprise—encompassing all of its information and technology services, processes, and infrastructure—or a specific domain within the enterprise. In both cases, the architecture crosses multiple systems, and multiple functional groups within the enterprise.

[22] The Open Group, *TOGAF Version 9.1*, *https://oreil.ly/A1H67*.

Gartner's definition

Gartner is a global research and advisory company that produces research articles and reports on trends related to enterprises. Among other things, it is responsible for the (in)famous Gartner Hype Cycle. Gartner's definition is as follows:[23]

> Enterprise architecture (EA) is a discipline for proactively and holistically leading enterprise responses to disruptive forces by identifying and analyzing the execution of change toward desired business vision and outcomes. EA delivers value by presenting business and IT leaders with signature-ready recommendations for adjusting policies and projects to achieve targeted business outcomes that capitalize on relevant business disruptions.

EABOK's definition

EABOK is the *Enterprise Architecture Book of Knowledge*, an enterprise architecture reference produced by the MITRE Corporation. EABOK was released as an incomplete draft in 2004 and has not been updated since. Though seemingly obsolete, EABOK is frequently referenced in descriptions of enterprise architecture; we found many of its ideas helpful while writing this book. Here's the EABOK definition:[24]

> Enterprise Architecture (EA) is an organizational model; an abstract representation of an Enterprise that aligns strategy, operations, and technology to create a roadmap for success.

Our definition

We extract a few common threads in these definitions of enterprise architecture: change, alignment, organization, opportunities, problem-solving, and migration. Here is our definition of *enterprise architecture*, one that we feel is more relevant to today's fast-moving data landscape:

> Enterprise architecture is the design of systems to support *change in the enterprise*, achieved by *flexible and reversible decisions* reached through careful *evaluation of trade-offs*.

Here, we touch on some key areas we'll return to throughout the book: flexible and reversible decisions, change management, and evaluation of trade-offs. We discuss each theme at length in this section and then make the definition more concrete in the latter part of the chapter by giving various examples of data architecture. Flexible and reversible decisions are essential for two reasons. First, the world is constantly changing, and predicting the future is impossible. Reversible decisions allow you to adjust course as the world changes and you gather new information. Second, there is a

[23] Gartner Glossary, s.v. "Enterprise Architecture (EA)," *https://oreil.ly/SWwQF*.
[24] EABOK Consortium website, *https://eabok.org*.

natural tendency toward enterprise ossification as organizations grow. Adopting a culture of reversible decisions helps overcome this tendency by reducing the risk attached to a decision.

Jeff Bezos is credited with the idea of one-way and two-way doors.[25] A *one-way door* is a decision that is almost impossible to reverse. For example, Amazon could have decided to sell AWS or shut it down. It would be nearly impossible for Amazon to rebuild a public cloud with the same market position after such an action.

On the other hand, a *two-way door* is an easily reversible decision: you walk through and proceed if you like what you see in the room or step back through the door if you don't. Amazon might decide to require the use of DynamoDB for a new microservices database. If this policy doesn't work, Amazon has the option of reversing it and refactoring some services to use other databases. Since the stakes attached to each reversible decision (two-way door) are low, organizations can make more decisions, iterating, improving, and collecting data rapidly.

Change management is closely related to reversible decisions and is a central theme of enterprise architecture frameworks. Even with an emphasis on reversible decisions, enterprises often need to undertake large initiatives. These are ideally broken into smaller changes, each one a reversible decision in itself. Returning to Amazon, we note a five-year gap (2007 to 2012) from the publication of a paper on the DynamoDB concept to Werner Vogels's announcement of the DynamoDB service on AWS. Behind the scenes, teams took numerous small actions to make DynamoDB a concrete reality for AWS customers. Managing such small actions is at the heart of change management.

Architects are not simply mapping out IT processes and vaguely looking toward a distant, utopian future; they actively solve business problems and create new opportunities. Technical solutions exist not for their own sake but in support of business goals. Architects identify problems in the current state (poor data quality, scalability limits, money-losing lines of business), define desired future states (agile data-quality improvement, scalable cloud data solutions, improved business processes), and realize initiatives through execution of small, concrete steps. It bears repeating:

> *Technical solutions exist not for their own sake but in support of business goals.*

We found significant inspiration in *Fundamentals of Software Architecture* by Mark Richards and Neal Ford (O'Reilly). They emphasize that trade-offs are inevitable and

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

ubiquitous in the engineering space. Sometimes the relatively fluid nature of software and data leads us to believe that we are freed from the constraints that engineers face in the hard, cold physical world. Indeed, this is partially true; patching a software bug is much easier than redesigning and replacing an airplane wing. However, digital systems are ultimately constrained by physical limits such as latency, reliability, density, and energy consumption. Engineers also confront various nonphysical limits, such as characteristics of programming languages and frameworks, and practical constraints in managing complexity, budgets, etc. Magical thinking culminates in poor engineering. Data engineers must account for trade-offs at every step to design an optimal system while minimizing high-interest technical debt.

Let's reiterate one central point in our enterprise architecture definition: enterprise architecture balances flexibility and trade-offs. This isn't always an easy balance, and architects must constantly assess and reevaluate with the recognition that the world is dynamic. Given the pace of change that enterprises are faced with, organizations—and their architecture—cannot afford to stand still.

Data Architecture Defined

Now that you understand enterprise architecture, let's dive into data architecture by establishing a working definition that will set the stage for the rest of the book. *Data architecture* is a subset of enterprise architecture, inheriting its properties: processes, strategy, change management, and evaluating trade-offs. Here are a couple of definitions of data architecture that influence our definition.

TOGAF's definition

TOGAF defines data architecture as follows:[26]

> A description of the structure and interaction of the enterprise's major types and sources of data, logical data assets, physical data assets, and data management resources.

DAMA's definition

The DAMA *DMBOK* defines data architecture as follows:[27]

> Identifying the data needs of the enterprise (regardless of structure) and designing and maintaining the master blueprints to meet those needs. Using master blueprints to

[26] The Open Group, *TOGAF Version 9.1*, *https://oreil.ly/A1H67*.

[27] *DAMA - DMBOK: Data Management Body of Knowledge*, 2nd ed. (Technics Publications, 2017).

guide data integration, control data assets, and align data investments with business strategy.

Our definition

Considering the preceding two definitions and our experience, we have crafted our definition of *data architecture*:

> Data architecture is the design of systems to support the evolving data needs of an enterprise, achieved by flexible and reversible decisions reached through a careful evaluation of trade-offs.

How does data architecture fit into data engineering? Just as the data engineering lifecycle is a subset of the data lifecycle, data engineering architecture is a subset of general data architecture. *Data engineering architecture* is the systems and frameworks that make up the key sections of the data engineering lifecycle. We'll use *data architecture* interchangeably with *data engineering architecture* throughout this book.

Other aspects of data architecture that you should be aware of are operational and technical (Figure 3-2). *Operational architecture* encompasses the functional requirements of what needs to happen related to people, processes, and technology. For example, what business processes does the data serve? How does the organization manage data quality? What is the latency requirement from when the data is produced to when it becomes available to query? *Technical architecture* outlines how data is ingested, stored, transformed, and served along the data engineering lifecycle. For instance, how will you move 10 TB of data every hour from a source database to your data lake? In short, operational architecture describes *what* needs to be done, and technical architecture details *how* it will happen.

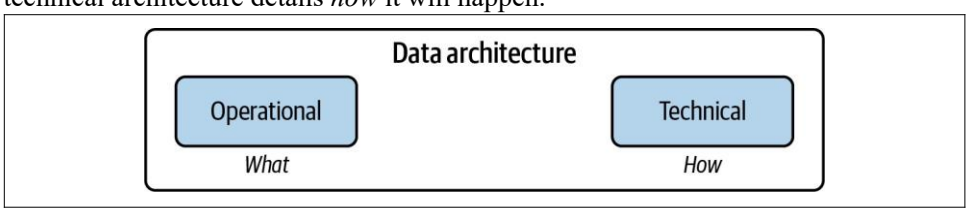


Figure 3-2. Operational and technical data architecture

Now that we have a working definition of data architecture, let's cover the elements of "good" data architecture.

"Good" Data Architecture

> Never shoot for the best architecture, but rather the least worst architecture.

—Mark Richards and Neal Ford[28]

According to Grady Booch, "Architecture represents the significant design decisions that shape a system, where *significant* is measured by cost of change." Data architects aim to make significant decisions that will lead to good architecture at a basic level.

What do we mean by "good" data architecture? To paraphrase an old cliche, you know good when you see it. *Good data architecture* serves business requirements with a common, widely reusable set of building blocks while maintaining flexibility and making appropriate trade-offs. Bad architecture is authoritarian and tries to cram a bunch of one-size-fits-all decisions into a big ball of mud.

[28] Mark Richards and Neal Ford, *Fundamentals of Software Architecture* (Sebastopol, CA: O'Reilly, 2020), *https://oreil.ly/hpCp0*.

Agility is the foundation for good data architecture; it acknowledges that the world is fluid. *Good data architecture is flexible and easily maintainable*. It evolves in response to changes within the business and new technologies and practices that may unlock even more value in the future. Businesses and their use cases for data are always evolving. The world is dynamic, and the pace of change in the data space is accelerating. Last year's data architecture that served you well might not be sufficient for today, let alone next year.

Bad data architecture is tightly coupled, rigid, overly centralized, or uses the wrong tools for the job, hampering development and change management. Ideally, by designing architecture with reversibility in mind, changes will be less costly.

The undercurrents of the data engineering lifecycle form the foundation of good data architecture for companies at any stage of data maturity. Again, these undercurrents are security, data management, DataOps, data architecture, orchestration, and software engineering.

Good data architecture is a living, breathing thing. It's never finished. In fact, per our definition, change and evolution are central to the meaning and purpose of data architecture. Let's now look at the principles of good data architecture.

Principles of Good Data Architecture

This section takes a 10,000-foot view of good architecture by focusing on principles—key ideas useful in evaluating major architectural decisions and practices. We borrow inspiration for our architecture principles from several sources, especially the AWS Well-Architected Framework and Google Cloud's Five Principles for Cloud-Native Architecture.

The AWS Well-Architected Framework consists of six pillars:

- Operational excellence
- Security
- Reliability
- Performance efficiency
- Cost optimization
- Sustainability

Google Cloud's Five Principles for Cloud-Native Architecture are as follows:

- Design for automation.
- Be smart with state.

- Favor managed services.
- Practice defense in depth.
- Always be architecting.

We advise you to carefully study both frameworks, identify valuable ideas, and determine points of disagreement. We'd like to expand or elaborate on these pillars with these principles of data engineering architecture:

1. Choose common components wisely.
2. Plan for failure.
3. Architect for scalability.
4. Architecture is leadership.
5. Always be architecting.
6. Build loosely coupled systems.
7. Make reversible decisions.
8. Prioritize security.
9. Embrace FinOps.

Principle 1: Choose Common Components Wisely

One of the primary jobs of a data engineer is to choose common components and practices that can be used widely across an organization. When architects choose well and lead effectively, common components become a fabric facilitating team collaboration and breaking down silos. Common components enable agility within and across teams in conjunction with shared knowledge and skills.

Common components can be anything that has broad applicability within an organization. Common components include object storage, version-control systems, observability, monitoring and orchestration systems, and processing engines. Common components should be accessible to everyone with an appropriate use case, and teams are encouraged to rely on common components already in use rather than reinventing the wheel. Common components must support robust permissions and security to enable sharing of assets among teams while preventing unauthorized access.

Cloud platforms are an ideal place to adopt common components. For example, compute and storage separation in cloud data systems allows users to access a shared storage layer (most commonly object storage) using specialized tools to access and query the data needed for specific use cases.

Choosing common components is a balancing act. On the one hand, you need to focus on needs across the data engineering lifecycle and teams, utilize common components that will be useful for individual projects, and simultaneously facilitate interoperation and collaboration. On the other hand, architects should avoid decisions that will hamper the productivity of engineers working on domain-specific problems by forcing them into one-size-fits-all technology solutions. Chapter 4 provides additional details.

Principle 2: Plan for Failure

> Everything fails, all the time.
>
> —Werner Vogels, CTO of Amazon Web Services[29]

Modern hardware is highly robust and durable. Even so, any hardware component will fail, given enough time. To build highly robust data systems, you must consider failures in your designs. Here are a few key terms for evaluating failure scenarios; we describe these in greater detail in this chapter and throughout the book:

Availability
: The percentage of time an IT service or component is in an operable state.

Reliability
: The system's probability of meeting defined standards in performing its intended function during a specified interval.

Recovery time objective
: The maximum acceptable time for a service or system outage. The recovery time objective (RTO) is generally set by determining the business impact of an outage. An RTO of one day might be fine for an internal reporting system. A website outage of just five minutes could have a significant adverse business impact on an online retailer.

Recovery point objective
: The acceptable state after recovery. In data systems, data is often lost during an outage. In this setting, the recovery point objective (RPO) refers to the maximum acceptable data loss.

Engineers need to consider acceptable reliability, availability, RTO, and RPO in designing for failure. These will guide their architecture decisions as they assess possible failure scenarios.

[29] UberPulse, "Amazon.com CTO: Everything Fails," YouTube video, 3:03, *https://oreil.ly/vDVlX*.

Principle 3: Architect for Scalability

Scalability in data systems encompasses two main capabilities. First, scalable systems can *scale up* to handle significant quantities of data. We might need to spin up a large cluster to train a model on a petabyte of customer data or scale out a streaming ingestion system to handle a transient load spike. Our ability to scale up allows us to handle extreme loads temporarily. Second, scalable systems can *scale down*. Once the load spike ebbs, we should automatically remove capacity to cut costs. (This is related to principle 9.) An *elastic system* can scale dynamically in response to load, ideally in an automated fashion.

Some scalable systems can also *scale to zero*: they shut down completely when not in use. Once the large model-training job completes, we can delete the cluster. Many serverless systems (e.g., serverless functions and serverless online analytical processing, or OLAP, databases) can automatically scale to zero.

Note that deploying inappropriate scaling strategies can result in overcomplicated systems and high costs. A straightforward relational database with one failover node may be appropriate for an application instead of a complex cluster arrangement. Measure your current load, approximate load spikes, and estimate load over the next several years to determine if your database architecture is appropriate. If your startup grows much faster than anticipated, this growth should also lead to more available resources to rearchitect for scalability.

Principle 4: Architecture Is Leadership

Data architects are responsible for technology decisions and architecture descriptions and disseminating these choices through effective leadership and training. Data architects should be highly technically competent but delegate most individual contributor work to others. Strong leadership skills combined with high technical competence are rare and extremely valuable. The best data architects take this duality seriously.

Note that leadership does not imply a command-and-control approach to technology. It was not uncommon in the past for architects to choose one proprietary database technology and force every team to house their data there. We oppose this approach because it can significantly hinder current data projects. Cloud environments allow architects to balance common component choices with flexibility that enables innovation within projects.

Returning to the notion of technical leadership, Martin Fowler describes a specific archetype of an ideal software architect, well embodied in his colleague Dave Rice:[30]

> In many ways, the most important activity of *Architectus Oryzus* is to mentor the development team, to raise their level so they can take on more complex issues. Improving the development team's ability gives an architect much greater leverage than being the sole decision-maker and thus running the risk of being an architectural bottleneck.

An ideal data architect manifests similar characteristics. They possess the technical skills of a data engineer but no longer practice data engineering day to day; they mentor current data engineers, make careful technology choices in consultation with their organization, and disseminate expertise through training and leadership. They train engineers in best practices and bring the company's engineering resources together to pursue common goals in both technology and business.

As a data engineer, you should practice architecture leadership and seek mentorship from architects. Eventually, you may well occupy the architect role yourself.

Principle 5: Always Be Architecting

We borrow this principle directly from Google Cloud's Five Principles for CloudNative Architecture. Data architects don't serve in their role simply to maintain the existing state; instead, they constantly design new and exciting things in response to changes in business and technology. Per the EABOK, an architect's job is to develop deep knowledge of the *baseline architecture* (current state), develop a *target architecture*, and map out a *sequencing plan* to determine priorities and the order of architecture changes.

We add that modern architecture should not be command-and-control or waterfall but collaborative and agile. The data architect maintains a target architecture and sequencing plans that change over time. The target architecture becomes a moving target, adjusted in response to business and technology changes internally and worldwide. The sequencing plan determines immediate priorities for delivery.

Principle 6: Build Loosely Coupled Systems

> When the architecture of the system is designed to enable teams to test, deploy, and change systems without dependencies on other teams, teams require little communication to get work done. In other words, both the architecture and the teams are loosely coupled.
>
> —Google DevOps tech architecture guide[31]

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

[31] Google Cloud, "DevOps Tech: Architecture," Cloud Architecture Center, *https://oreil.ly/j4MT1*.

In 2002, Bezos wrote an email to Amazon employees that became known as the Bezos API Mandate:[32]

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

The advent of Bezos's API Mandate is widely viewed as a watershed moment for Amazon. Putting data and services behind APIs enabled the loose coupling and eventually resulted in AWS as we know it now. Google's pursuit of loose coupling allowed it to grow its systems to an extraordinary scale.

For software architecture, a loosely coupled system has the following properties:

1. Systems are broken into many small components.
2. These systems interface with other services through abstraction layers, such as a messaging bus or an API. These abstraction layers hide and protect internal details of the service, such as a database backend or internal classes and method

calls.

3. As a consequence of property 2, internal changes to a system component don't require changes in other parts. Details of code updates are hidden behind stable APIs. Each piece can evolve and improve separately.
4. As a consequence of property 3, there is no waterfall, global release cycle for the whole system. Instead, each component is updated separately as changes and improvements are made.

Notice that we are talking about *technical systems*. We need to think bigger. Let's translate these technical characteristics into organizational characteristics:

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

1. Many small teams engineer a large, complex system. Each team is tasked with engineering, maintaining, and improving some system components.
2. These teams publish the abstract details of their components to other teams via API definitions, message schemas, etc. Teams need not concern themselves with other teams' components; they simply use the published API or message specifications to call these components. They iterate their part to improve their performance and capabilities over time. They might also publish new capabilities as they are added or request new stuff from other teams. Again, the latter happens without teams needing to worry about the internal technical details of the requested features. Teams work together through *loosely coupled communication*.
3. As a consequence of characteristic 2, each team can rapidly evolve and improve its component independently of the work of other teams.
4. Specifically, characteristic 3 implies that teams can release updates to their components with minimal downtime. Teams release continuously during regular working hours to make code changes and test them.

Loose coupling of both technology and human systems will allow your data engineering teams to more efficiently collaborate with one another and with other parts of the company. This principle also directly facilitates principle 7.

Principle 7: Make Reversible Decisions

The data landscape is changing rapidly. Today's hot technology or stack is tomorrow's afterthought. Popular opinion shifts quickly. You should aim for reversible decisions, as these tend to simplify your architecture and keep it agile.

As Fowler wrote, "One of an architect's most important tasks is to remove architecture by finding ways to eliminate irreversibility in software designs."[33] What was true when Fowler wrote this in 2003 is just as accurate today.

As we said previously, Bezos refers to reversible decisions as "two-way doors." As he says, "If you walk through and don't like what you see on the other side, you can't get back to before. We can call these Type 1 decisions. But most decisions aren't like that—they are changeable, reversible—they're two-way doors." Aim for two-way doors whenever possible.

Given the pace of change—and the decoupling/modularization of technologies across your data architecture—always strive to pick the best-of-breed solutions that work for today. Also, be prepared to upgrade or adopt better practices as the landscape evolves.

[33] Fowler, "Who Needs an Architect?"

Principle 8: Prioritize Security

Every data engineer must assume responsibility for the security of the systems they build and maintain. We focus now on two main ideas: zero-trust security and the shared responsibility security model. These align closely to a cloud-native architecture.

Hardened-perimeter and zero-trust security models

To define *zero-trust security*, it's helpful to start by understanding the traditional hard-perimeter security model and its limitations, as detailed in Google Cloud's Five Principles:[34]

> Traditional architectures place a lot of faith in perimeter security, crudely a hardened network perimeter with "trusted things" inside and "untrusted things" outside. Unfortunately, this approach has always been vulnerable to insider attacks, as well as external threats such as spear phishing.

The 1996 film *Mission Impossible* presents a perfect example of the hard-perimeter security model and its limitations. In the movie, the CIA hosts highly sensitive data on a storage system inside a room with extremely tight physical security. Ethan Hunt infiltrates CIA headquarters and exploits a human target to gain physical access to the storage system. Once inside the secure room, he can exfiltrate data with relative ease.

For at least a decade, alarming media reports have made us aware of the growing menace of security breaches that exploit human targets inside hardened organizational security perimeters. Even as employees work on highly secure corporate networks, they remain connected to the outside world through email and mobile devices. External threats effectively become internal threats.

In a cloud-native environment, the notion of a hardened perimeter erodes further. All assets are connected to the outside world to some degree. While virtual private cloud (VPC) networks can be defined with no external connectivity, the API control plane that engineers use to define these networks still faces the internet.

The shared responsibility model

Amazon emphasizes the shared responsibility model, which divides security into the security *of* the cloud and security *in* the cloud. AWS is responsible for the security of the cloud:[35]

---

[34] Tom Grey, "5 Principles for Cloud-Native Architecture—What It Is and How to Master It," Google Cloud blog, June 19, 2019, *https://oreil.ly/4NkGf*.

[35] Amazon Web Services, "Security in AWS WAF," AWS WAF documentation, *https://oreil.ly/rEFoU*.

> AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely.

AWS users are responsible for security in the cloud:

> Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

In general, all cloud providers operate on some form of this shared responsibility model. They secure their services according to published specifications. Still, it is ultimately the user's responsibility to design a security model for their applications and data and leverage cloud capabilities to realize this model.

Data engineers as security engineers

In the corporate world today, a command-and-control approach to security is quite common, wherein security and networking teams manage perimeters and general security practices. The cloud pushes this responsibility out to engineers who are not explicitly in security roles. Because of this responsibility, in conjunction with more general erosion of the hard security perimeter, all data engineers should consider themselves security engineers.

Failure to assume these new implicit responsibilities can lead to dire consequences. Numerous data breaches have resulted from the simple error of configuring Amazon S3 buckets with public access.[36] Those who handle data must assume that they are ultimately responsible for securing it.

Principle 9: Embrace FinOps

Let's start by considering a couple of definitions of FinOps. First, the FinOps Foundation offers this:[37]

> FinOps is an evolving cloud financial management discipline and cultural practice that enables organizations to get maximum business value by helping engineering, finance, technology, and business teams to collaborate on data-driven spending decisions. In addition, J. R. Sorment and Mike Fuller provide the following definition in *Cloud FinOps*:[38]

> The term "FinOps" typically refers to the emerging professional movement that advocates a collaborative working relationship between DevOps and Finance, resulting

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

[37] FinOps Foundation, "What Is FinOps," *https://oreil.ly/wJFVn*.

[38] J. R. Storment and Mike Fuller, *Cloud FinOps* (Sebastapol, CA: O'Reilly, 2019), *https://oreil.ly/QV6vF*.

in an iterative, data-driven management of infrastructure spending (i.e., lowering the unit economics of cloud) while simultaneously increasing the cost efficiency and, ultimately, the profitability of the cloud environment.

The cost structure of data has evolved dramatically during the cloud era. In an on-premises setting, data systems are generally acquired with a capital expenditure (described more in Chapter 4) for a new system every few years in an on-premises setting. Responsible parties have to balance their budget against desired compute and storage capacity. Overbuying entails wasted money, while underbuying means hampering future data projects and driving significant personnel time to control system load and data size; underbuying may require faster technology refresh cycles, with associated extra costs.

In the cloud era, most data systems are pay-as-you-go and readily scalable. Systems can run on a cost-per-query model, cost-per-processing-capacity model, or another variant of a pay-as-you-go model. This approach can be far more efficient than the capital expenditure approach. It is now possible to scale up for high performance, and then scale down to save money. However, the pay-as-you-go approach makes spending far more dynamic. The new challenge for data leaders is to manage budgets, priorities, and efficiency.

Cloud tooling necessitates a set of processes for managing spending and resources. In the past, data engineers thought in terms of performance engineering—maximizing the performance for data processes on a fixed set of resources and buying adequate resources for future needs. With FinOps, engineers need to learn to think about the cost structures of cloud systems. For example, what is the appropriate mix of AWS spot instances when running a distributed cluster? What is the most appropriate approach for running a sizable daily job in terms of cost-effectiveness and performance? When should the company switch from a pay-per-query model to reserved capacity?

FinOps evolves the operational monitoring model to monitor spending on an ongoing basis. Rather than simply monitor requests and CPU utilization for a web server, FinOps might monitor the ongoing cost of serverless functions handling traffic, as well as spikes in spending trigger alerts. Just as systems are designed to fail gracefully in excessive traffic, companies may consider adopting hard limits for spending, with graceful failure modes in response to spending spikes.

Ops teams should also think in terms of cost attacks. Just as a distributed denial-ofservice (DDoS) attack can block access to a web server, many companies have discovered to their chagrin that excessive downloads from S3 buckets can drive spending through the roof and threaten a small startup with bankruptcy. When sharing data publicly, data teams can address these issues by setting requester-pays policies, or

simply monitoring for excessive data access spending and quickly removing access if spending begins to rise to unacceptable levels.

As of this writing, FinOps is a recently formalized practice. The FinOps Foundation was started only in 2019.[39] However, we highly recommend you start thinking about FinOps early, before you encounter high cloud bills. Start your journey with the FinOps Foundation and O'Reilly's *Cloud FinOps*. We also suggest that data engineers involve themselves in the community process of creating FinOps practices for data engineering— in such a new practice area, a good deal of territory is yet to be mapped out.

Now that you have a high-level understanding of good data architecture principles, let's dive a bit deeper into the major concepts you'll need to design and build good data architecture.

Major Architecture Concepts

If you follow the current trends in data, it seems like new types of data tools and architectures are arriving on the scene every week. Amidst this flurry of activity, we must not lose sight of the main goal of all of these architectures: to take data and transform it into something useful for downstream consumption.

Domains and Services

> Domain: A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.
>
> —Eric Evans[40]

Before diving into the components of the architecture, let's briefly cover two terms you'll see come up very often: domain and services. A *domain* is the real-world subject area for which you're architecting. A *service* is a set of functionality whose goal is to accomplish a task. For example, you might have a sales order-processing service whose task is to process orders as they are created. The sales order-processing service's only job is to process orders; it doesn't provide other functionality, such as inventory management or updating user profiles.

A domain can contain multiple services. For example, you might have a sales domain with three services: orders, invoicing, and products. Each service has particular tasks that support the sales domain. Other domains may also share services (Figure 3-3). In this case, the accounting domain is responsible for basic accounting functions: invoicing, payroll, and accounts receivable (AR). Notice the accounting domain shares

[39] "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*.

[40] Eric Evans, *Domain-Driven Design Reference: Definitions and Pattern Summaries* (March 2015), *https://oreil.ly/pQ9oq*.

the invoice service with the sales domain since a sale generates an invoice, and accounting must keep track of invoices to ensure that payment is received. Sales and accounting own their respective domains.

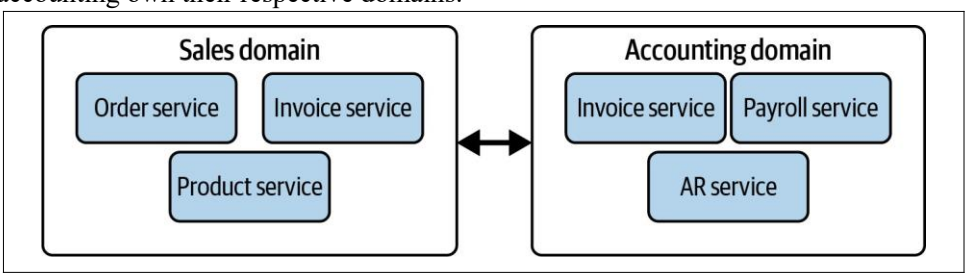


Figure 3-3. Two domains (sales and accounting) share a common service (invoices), and sales and accounting own their respective domains

When thinking about what constitutes a domain, focus on what the domain represents in the real world and work backward. In the preceding example, the sales domain should represent what happens with the sales function in your company. When architecting the sales domain, avoid cookie-cutter copying and pasting from what other companies do. Your company's sales function likely has unique aspects that require specific services to make it work the way your sales team expects. Identify what should go in the domain. When determining what the domain should encompass and what services to include, the best advice is to simply go and talk with users and stakeholders, listen to what they're saying, and build the services that will help them do their job. Avoid the classic trap of architecting in a vacuum.

Distributed Systems, Scalability, and Designing for Failure

The discussion in this section is related to our second and third principles of data engineering architecture discussed previously: plan for failure and architect for scalability. As data engineers, we're interested in four closely related characteristics of data systems (availability and reliability were mentioned previously, but we reiterate them here for completeness):

Scalability
: Allows us to increase the capacity of a system to improve performance and handle the demand. For example, we might want to scale a system to handle a high rate of queries or process a huge data set.

Elasticity
: The ability of a scalable system to scale dynamically; a highly elastic system can automatically scale up and down based on the current workload. Scaling up is critical as demand increases, while scaling down saves money in a cloud

environment. Modern systems sometimes scale to zero, meaning they can automatically shut down when idle.

Availability
: The percentage of time an IT service or component is in an operable state.

Reliability
: The system's probability of meeting defined standards in performing its intended function during a specified interval.

See PagerDuty's "Why Are Availability and Reliability Crucial?" web page for definitions and background on availability and reliability.

How are these characteristics related? If a system fails to meet performance requirements during a specified interval, it may become unresponsive. Thus low reliability can lead to low availability. On the other hand, dynamic scaling helps ensure adequate performance without manual intervention from engineers—elasticity improves reliability.

Scalability can be realized in a variety of ways. For your services and domains, does a single machine handle everything? A single machine can be scaled vertically; you can increase resources (CPU, disk, memory, I/O). But there are hard limits to possible resources on a single machine. Also, what happens if this machine dies? Given enough time, some components will eventually fail. What's your plan for backup and failover? Single machines generally can't offer high availability and reliability. We utilize a distributed system to realize higher overall scaling capacity and increased availability and reliability. *Horizontal scaling* allows you to add more machines to satisfy load and resource requirements (Figure 3-4). Common horizontally scaled systems have a leader node that acts as the main point of contact for the instantiation, progress, and completion of workloads. When a workload is started, the leader node distributes tasks to the worker nodes within its system, completing the tasks and returning the results to the leader node. Typical modern distributed architectures also build in redundancy. Data is replicated so that if a machine dies, the other machines can pick up where the missing server left off; the cluster may add more machines to restore capacity.

Distributed systems are widespread in the various data technologies you'll use across your architecture. Almost every cloud data warehouse object storage system you use has some notion of distribution under the hood. Management details of the distributed system are typically abstracted away, allowing you to focus on high-level architecture instead of low-level plumbing. However, we highly recommend that you learn more about distributed systems because these details can be extremely helpful in

understanding and improving the performance of your pipelines; Martin Kleppmann's *Designing Data-Intensive Applications* (O'Reilly) is an excellent resource.

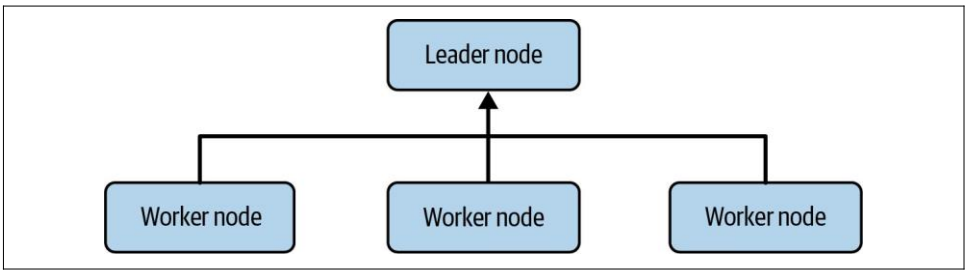


Figure 3-4. A simple horizontal distributed system utilizing a leader-follower architecture, with one leader node and three worker nodes

Tight Versus Loose Coupling: Tiers, Monoliths, and Microservices

When designing a data architecture, you choose how much interdependence you want to include within your various domains, services, and resources. On one end of the spectrum, you can choose to have extremely centralized dependencies and workflows. Every part of a domain and service is vitally dependent upon every other domain and service. This pattern is known as *tightly coupled*.

On the other end of the spectrum, you have decentralized domains and services that do not have strict dependence on each other, in a pattern known as *loose coupling*. In a loosely coupled scenario, it's easy for decentralized teams to build systems whose data may not be usable by their peers. Be sure to assign common standards, ownership, responsibility, and accountability to the teams owning their respective domains and services. Designing "good" data architecture relies on trade-offs between the tight and loose coupling of domains and services.

It's worth noting that many of the ideas in this section originate in software development. We'll try to retain the context of these big ideas' original intent and spirit— keeping them agnostic of data—while later explaining some differences you should be aware of when applying these concepts to data specifically.

Architecture tiers

As you develop your architecture, it helps to be aware of architecture tiers. Your architecture has layers—data, application, business logic, presentation, and so forth — and you need to know how to decouple these layers. Because tight coupling of modalities presents obvious vulnerabilities, keep in mind how you structure the layers

of your architecture to achieve maximum reliability and flexibility. Let's look at single-tier and multitier architecture.

Single tier. In a *single-tier architecture*, your database and application are tightly coupled, residing on a single server (Figure 3-5). This server could be your laptop or a single virtual machine (VM) in the cloud. The tightly coupled nature means if the server, the database, or the application fails, the entire architecture fails. While single-tier architectures are good for prototyping and development, they are not advised for production environments because of the obvious failure risks.

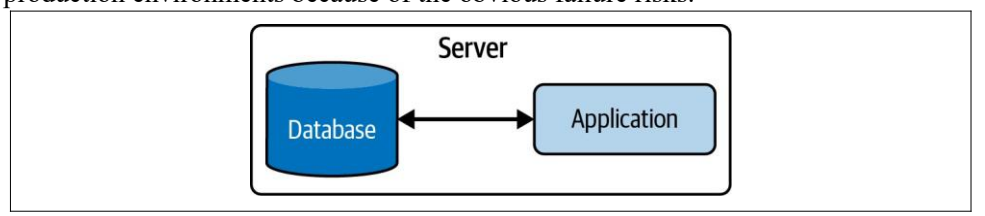


Figure 3-5. Single-tier architecture

Even when single-tier architectures build in redundancy (for example, a failover replica), they present significant limitations in other ways. For instance, it is often impractical (and not advisable) to run analytics queries against production application databases. Doing so risks overwhelming the database and causing the application to become unavailable. A single-tier architecture is fine for testing systems on a local machine but is not advised for production uses.

Multitier. The challenges of a tightly coupled single-tier architecture are solved by decoupling the data and application. A *multitier* (also known as *n-tier*) architecture is composed of separate layers: data, application, business logic, presentation, etc. These layers are bottom-up and hierarchical, meaning the lower layer isn't necessarily dependent on the upper layers; the upper layers depend on the lower layers. The notion is to separate data from the application, and application from the presentation.

A common multitier architecture is a three-tier architecture, a widely used clientserver design. A *three-tier architecture* consists of data, application logic, and presentation tiers (Figure 3-6). Each tier is isolated from the other, allowing for separation of concerns. With a three-tier architecture, you're free to use whatever technologies you prefer within each tier without the need to be monolithically focused.

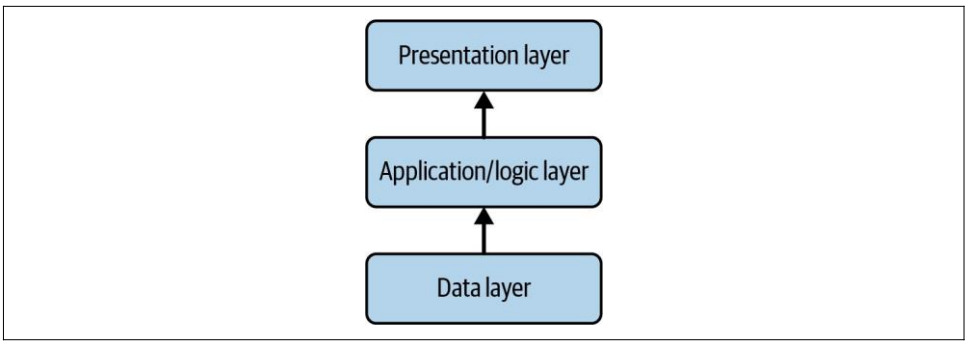


Figure 3-6. A three-tier architecture

We've seen many single-tier architectures in production. Single-tier architectures offer simplicity but also severe limitations. Eventually, an organization or application outgrows this arrangement; it works well until it doesn't. For instance, in a single-tier architecture, the data and logic layers share and compete for resources (disk, CPU, and memory) in ways that are simply avoided in a multitier architecture. Resources are spread across various tiers. Data engineers should use tiers to evaluate their layered architecture and the way dependencies are handled. Again, start simple and bake in evolution to additional tiers as your architecture becomes more complex.

In a multitier architecture, you need to consider separating your layers and the way resources are shared within layers when working with a distributed system. Distributed systems under the hood power many technologies 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.

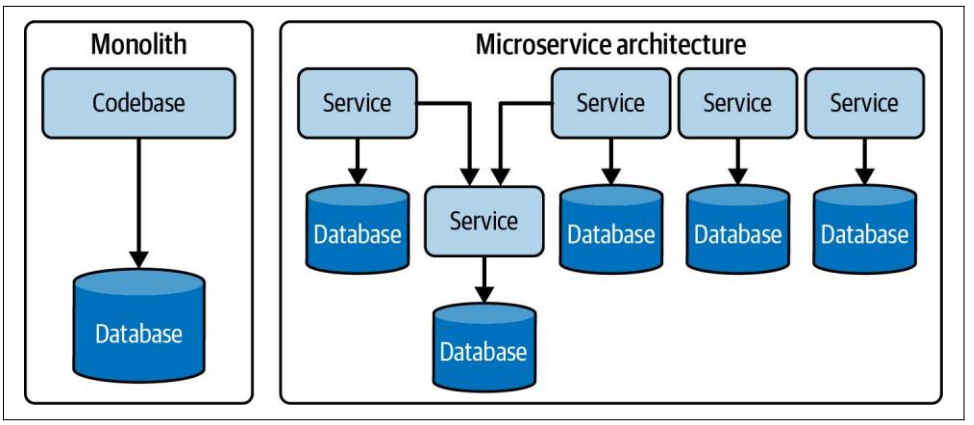


Figure 3-7. An extremely monolithic architecture runs all functionality inside a single codebase, potentially colocating a database on the same host server

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.

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.

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

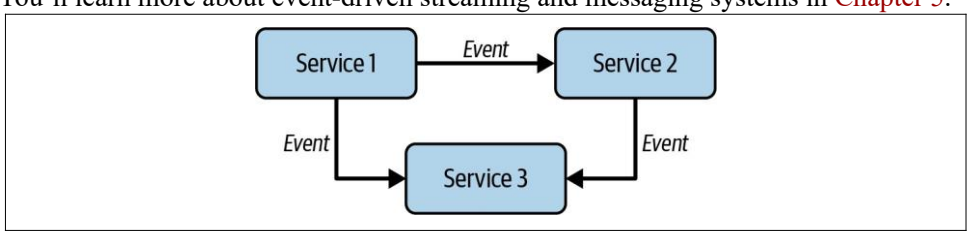


Figure 3-9. In an event-driven architecture, events are passed between loosely coupled services

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.[41] 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

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

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.[42]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.

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

In 1989, 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."[43] 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.

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

The organizational data warehouse architecture has two main characteristics:

Separates online analytical processing (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.

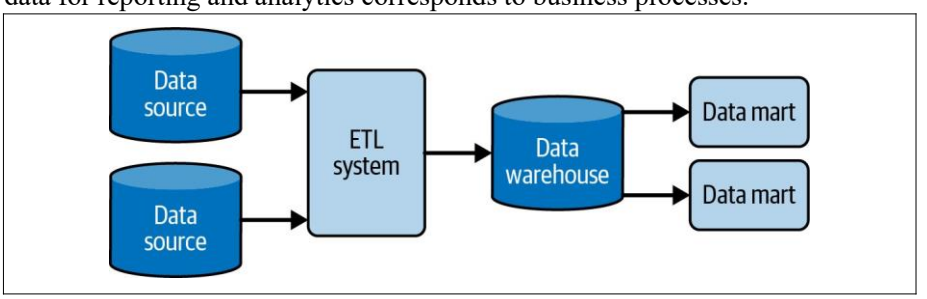


Figure 3-10. Basic data warehouse with ETL

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.

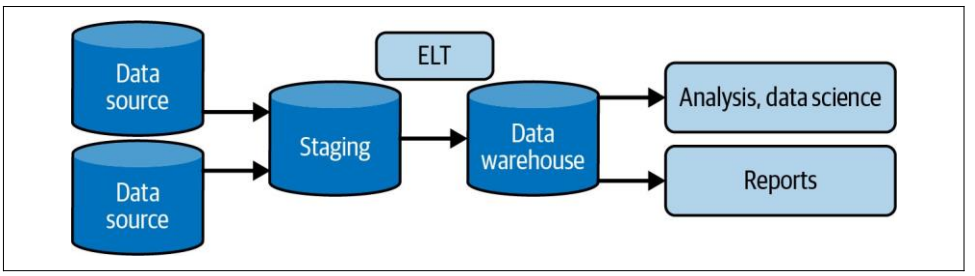


Figure 3-11. ELT—extract, load, and transform

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" on page 101.

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.

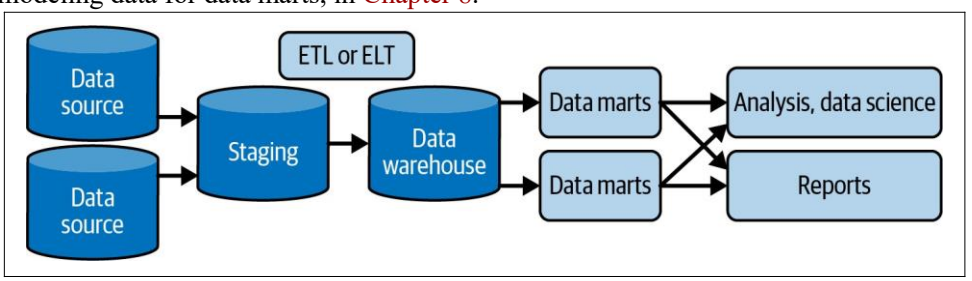


Figure 3-12. ETL or ELT plus data marts

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 a variety of unstructured data 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-today 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.

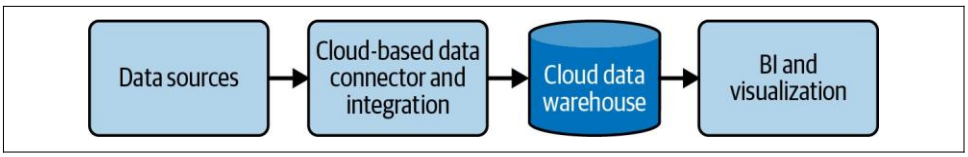


Figure 3-13. Basic components of the modern data stack

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.

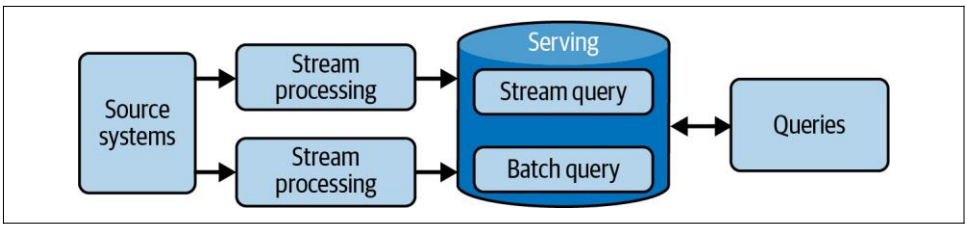


Figure 3-14. Lambda architecture

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).[44] 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.

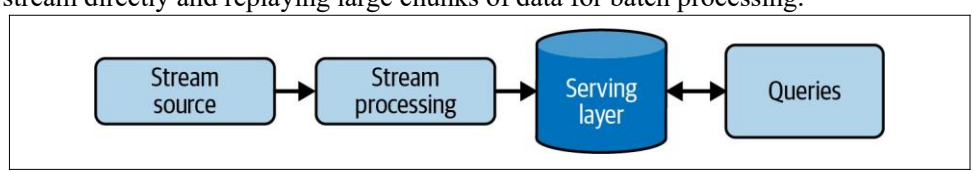


Figure 3-15. Kappa architecture

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; it's easy to talk about, but harder than expected to execute. Second, Kappa architecture turns out to be complicated and expensive in practice. While some streaming systems can scale to huge data volumes, they are

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

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).

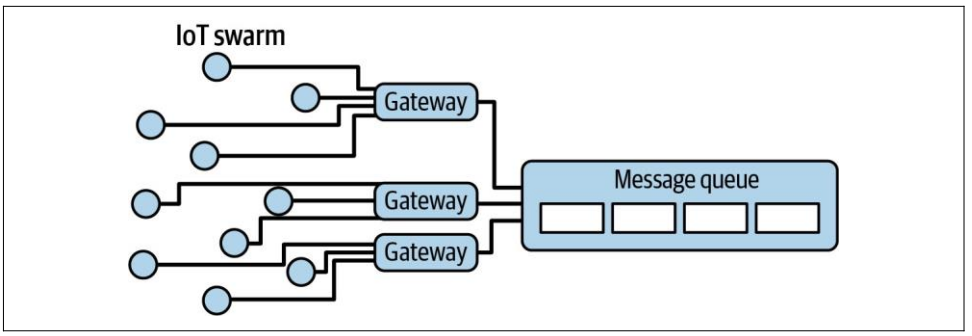


Figure 3-16. A device swarm (circles), IoT gateways, and message queue with messages (rectangles within the queue)

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.

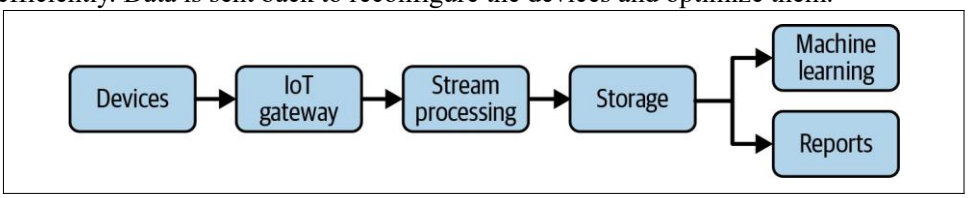


Figure 3-17. IoT serving pattern for downstream use cases

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.[45] 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:[46]

> 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.

[45] Zhamak Dehghani, "Data Mesh Principles and Logical Architecture," MartinFowler.com, December 3, 2020, *https://oreil.ly/ezWE7*.

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

Dehghani later identified four key components of the data mesh:[47]

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

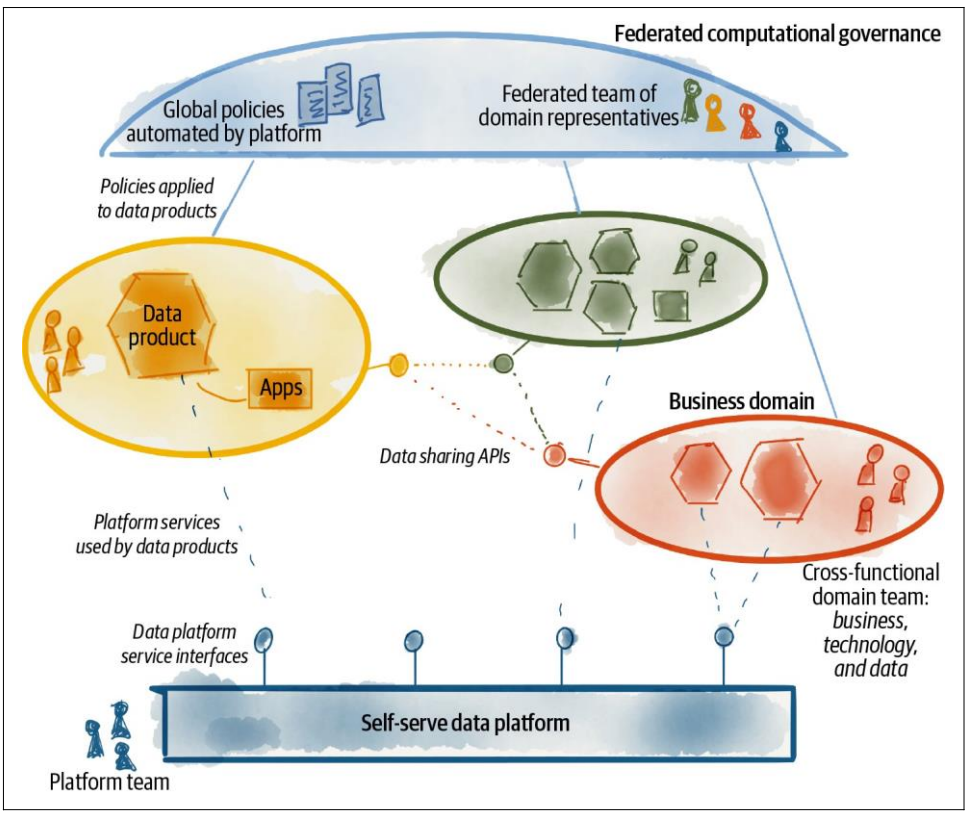


Figure 3-18. Simplified example of a data mesh architecture. Source: From Data Mesh*, by Zhamak Dehghani. Copyright © 2022 Zhamak Dehghani. Published by O'Reilly Media, Inc. Used with permission.*

[47] Zhamak Dehghani, "Data Mesh Principles and Logical Architecture."

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 work 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

- "AnemicDomainModel" by Martin Fowler
- "Big Data Architectures" Azure documentation
- "BoundedContext" by Martin Fowler

-
- "A Brief Introduction to Two Data Processing Architectures—Lambda and Kappa for Big Data" by Iman Samizadeh
- "The Building Blocks of a Modern Data Platform" by Prukalpa • "Choosing Open Wisely" by Benoit Dageville et al.

"Choosing the Right Architecture for Global Data Distribution" Google Cloud Architecture web page

- "Column-Oriented DBMS" Wikipedia page
- "A Comparison of Data Processing Frameworks" by Ludovik Santos
- "The Cost of Cloud, a Trillion Dollar Paradox" by Sarah Wang and Martin Casado
- "The Curse of the Data Lake Monster" by Kiran Prakash and Lucy Chambers
- *Data Architecture: A Primer for the Data Scientist* by W. H. Inmon et al. (Academic Press)
- "Data Architecture: Complex vs. Complicated" by Dave Wells
- "Data as a Product vs. Data as a Service" by Justin Gage
- "The Data Dichotomy: Rethinking the Way We Treat Data and Services" by Ben Stopford
- "Data Fabric Architecture Is Key to Modernizing Data Management and Integration" by Ashutosh Gupta
- "Data Fabric Defined" by James Serra
- "Data Team Platform" by GitLab Data
- "Data Warehouse Architecture: Overview" by Roelant Vos
- "Data Warehouse Architecture" tutorial at Javatpoint
- "Defining Architecture" ISO/IEC/IEEE 42010 web page
- "The Design and Implementation of Modern Column-Oriented Database Systems" by Daniel Abadi et al.
- "Disasters I've Seen in a Microservices World" by Joao Alves
- "DomainDrivenDesign" by Martin Fowler
- "Down with Pipeline Debt: Introducing Great Expectations" by the Great Expectations project
- *EABOK* draft, edited by Paula Hagan
- EABOK website
- "EagerReadDerivation" by Martin Fowler

-
- “End-to-End Serverless ETL Orchestration in AWS: A Guide” by Rittika Jindal
- “Enterprise Architecture” Gartner Glossary definition
- “Enterprise Architecture’s Role in Building a Data-Driven Organization” by Ashutosh Gupta
- “Event Sourcing” by Martin Fowler
  “Falling Back in Love with Data Pipelines” by Sean Knapp
- “Five Principles for Cloud-Native Architecture: What It Is and How to Master It” by Tom Grey
- “Focusing on Events” by Martin Fowler
- “Functional Data Engineering: A Modern Paradigm for Batch Data Processing” by Maxime Beauchemin
- “Google Cloud Architecture Framework” Google Cloud Architecture web page
- “How to Beat the Cap Theorem” by Nathan Marz
- “How to Build a Data Architecture to Drive Innovation—Today and Tomorrow” by Antonio Castro et al.
- “How TOGAF Defines Enterprise Architecture (EA)” by Avancier Limited
- The Information Management Body of Knowledge website
- “Introducing Dagster: An Open Source Python Library for Building Data Applications” by Nick Schrock
- “The Log: What Every Software Engineer Should Know About Real-Time Data’s Unifying Abstraction” by Jay Kreps
- “Microsoft Azure IoT Reference Architecture” documentation
- Microsoft’s “Azure Architecture Center”
- “Modern CI Is Too Complex and Misdirected” by Gregory Szorc
- “The Modern Data Stack: Past, Present, and Future” by Tristan Handy
- “Moving Beyond Batch vs. Streaming” by David Yaffe
- “A Personal Implementation of Modern Data Architecture: Getting Strava Data into Google Cloud Platform” by Matthew Reeve
- “Polyglot Persistence” by Martin Fowler
- “Potemkin Data Science” by Michael Correll
- “Principled Data Engineering, Part I: Architectural Overview” by Hussein Danish
- “Questioning the Lambda Architecture” by Jay Kreps
- “Reliable Microservices Data Exchange with the Outbox Pattern” by Gunnar Morling
- “ReportingDatabase” by Martin Fowler

-
- “The Rise of the Metadata Lake” by Prukalpa
- “Run Your Data Team Like a Product Team” by Emilie Schario and Taylor A. Murphy

Additional Resources

“Separating Utility from Value Add” by Ross Pettit

- “The Six Principles of Modern Data Architecture” by Joshua Klahr
- Snowflake’s “What Is Data Warehouse Architecture” web page
- “Software Infrastructure 2.0: A Wishlist” by Erik Bernhardsson
- “Staying Ahead of Data Debt” by Etai Mizrahi
- “Tactics vs. Strategy: SOA and the Tarpit of Irrelevancy” by Neal Ford
- “Test Data Quality at Scale with Deequ” by Dustin Lange et al.
- “Three-Tier Architecture” by IBM Education
- TOGAF framework website
- “The Top 5 Data Trends for CDOs to Watch Out for in 2021” by Prukalpa
- “240 Tables and No Documentation?” by Alexey Makhotkin
- “The Ultimate Data Observability Checklist” by Molly Vorwerck
- “Unified Analytics: Where Batch and Streaming Come Together; SQL and Beyond” Apache Flink Roadmap
- “UtilityVsStrategicDichotomy” by Martin Fowler
- “What Is a Data Lakehouse?” by Ben Lorica et al.
- “What Is Data Architecture? A Framework for Managing Data” by Thor Olavsrud
- “What Is the Open Data Ecosystem and Why It’s Here to Stay” by Casber Wang
- “What’s Wrong with MLOps?” by Laszlo Sragner
- “What the Heck Is Data Mesh” by Chris Riccomini
- “Who Needs an Architect” by Martin Fowler
- “Zachman Framework” Wikipedia page

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 high-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. Seuss 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 technology, language, or tool, 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.[48] 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

[48] For more details, see "Total Opportunity Cost of Ownership" by Joseph Reis in *97 Things Every Data Engineer Should Know* (O'Reilly).

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 other data stacks?

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" on page 85. 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:[49]

> 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.

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

[49] J. R. Storment and Mike Fuller, *Cloud FinOps* (Sebastopol, CA: O'Reilly, 2019), 6, *https://oreil.ly/RvRvX*.

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 as a whole.

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, and React and Vue.js have massive mindshare today. What's the popular frontend JavaScript 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. If VCs pouring millions (or billions) into data-tooling investments can't get it right, 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).

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. Almost every technology follows this inevitable path of decline.

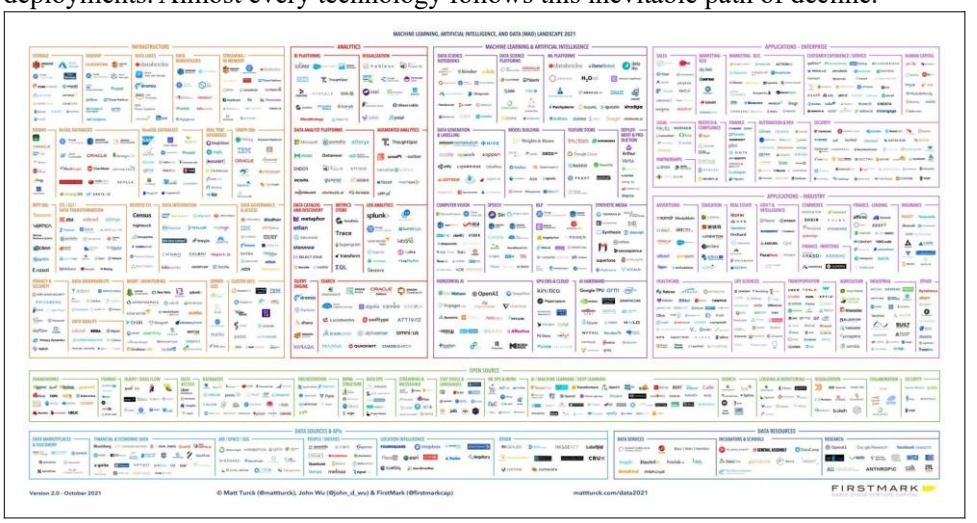


Figure 4-1. Matt Turck's 2021 MAD data landscape

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.

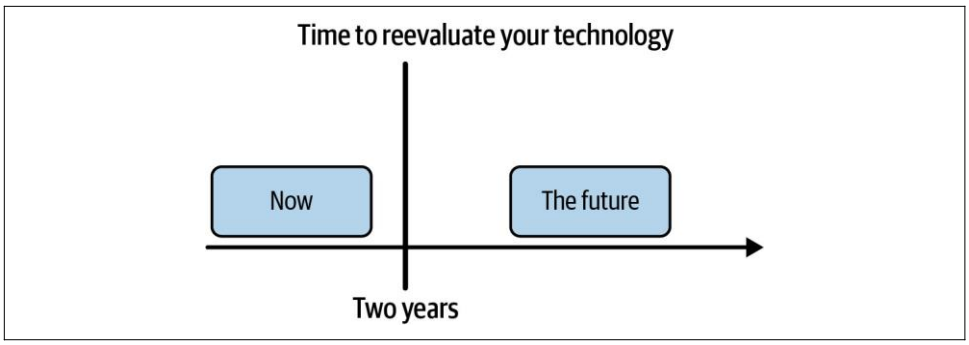


Figure 4-2. Use a two-year time horizon to reevaluate your technology choices

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, often 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" on page 120. Enterprises that migrate to the cloud often make major deployment errors by not appropriately adapting their practices to the cloud pricing model.

A Brief Detour on Cloud Economics

To understand how to use cloud services efficiently through cloud native architecture, you need to know how clouds make money. This is an extremely complex concept and one on which cloud providers offer little transparency. Consider this sidebar a starting point for your research, discovery, and process development.

Cloud Services and Credit Default Swaps

Let's go on a little tangent about credit default swaps. Don't worry, this will make sense in a bit. Recall that credit default swaps rose to infamy after the 2007 global financial crisis. A credit default swap was a mechanism for selling different tiers of risk attached to an asset (e.g., a mortgage). It is not our intention to present this idea in any detail, but rather to offer an analogy wherein many cloud services are similar to financial derivatives; cloud providers not only slice hardware assets into small pieces through virtualization, but also sell these pieces with varying technical characteristics and risks attached. While providers are extremely tight-lipped about details of their internal systems, there are massive opportunities for optimization and scaling by understanding cloud pricing and exchanging notes with other users.

Look at the example of archival cloud storage. At the time of this writing, GCP openly

admits that its archival class storage runs on the same clusters as standard cloud storage, yet the price per gigabyte per month of archival storage is roughly 1/17 that of standard storage. How is this possible?

Here's our educated guess. When purchasing cloud storage, each disk in a storage cluster has three assets that cloud providers and consumers use. First, it has a certain storage capacity—say, 10 TB. Second, it supports a certain number of input/output operations (IOPs) per second—say, 100. Third, disks support a certain maximum bandwidth, the maximum read speed for optimally organized files. A magnetic drive might be capable of reading at 200 MB/s.

Any of these limits (IOPs, storage capacity, bandwidth) is a potential bottleneck for a cloud provider. For instance, the cloud provider might have a disk storing 3 TB of data but hitting maximum IOPs. An alternative to leaving the remaining 7 TB empty is to sell the empty space without selling IOPs. Or, more specifically, sell cheap storage space and expensive IOPs to discourage reads.

Much like traders of financial derivatives, cloud vendors also deal in risk. In the case of archival storage, vendors are selling a type of insurance, but one that pays out for the insurer rather than the policy buyer in the event of a catastrophe. While data storage costs per month are extremely cheap, I risk paying a high price if I ever need to retrieve data. But this is a price that I will happily pay in a true emergency.

Similar considerations apply to nearly any cloud service. While on-premises servers are essentially sold as commodity hardware, the cost model in the cloud is more subtle. Rather than just charging for CPU cores, memory, and features, cloud vendors monetize characteristics such as durability, reliability, longevity, and predictability; a variety of compute platforms discount their offerings for workloads that are ephemeral or can be arbitrarily interrupted when capacity is needed elsewhere.

Cloud ≠ On Premises

This heading may seem like a silly tautology, but the belief that cloud services are just like familiar on-premises servers is a widespread cognitive error that plagues cloud migrations and leads to horrifying bills. This demonstrates a broader issue in tech that we refer to as *the curse of familiarity*. Many new technology products are intentionally designed to look like something familiar to facilitate ease of use and accelerate adoption. But, any new technology product has subtleties and wrinkles that users must learn to identify, accommodate, and optimize.

Moving on-premises servers one by one to VMs in the cloud—known as simple *lift and shift*—is a perfectly reasonable strategy for the initial phase of cloud migration, especially when a company is facing some kind of financial cliff, such as the need to sign a significant new lease or hardware contract if existing hardware is not shut down. However, companies that leave their cloud assets in this initial state are in for a rude shock. On a direct comparison basis, long-running servers in the cloud are significantly more expensive than their on-premises counterparts.

The key to finding value in the cloud is understanding and optimizing the cloud pricing model. Rather than deploying a set of long-running servers capable of handling full peak load, use autoscaling to allow workloads to scale down to minimal infrastructure when loads are light and up to massive clusters during peak times. To realize discounts through more ephemeral, less durable workloads, use reserved or spot instances, or use serverless functions in place of servers.

We often think of this optimization as leading to lower costs, but we should also strive to *increase business value* by exploiting the dynamic nature of the cloud.[50] Data engineers can create new value in the cloud by accomplishing things that were impossible in their on-premises environment. For example, it is possible to quickly spin up massive compute clusters to run complex transformations at scales that were unaffordable for on-premises hardware.

Data Gravity

In addition to basic errors such as following on-premises operational practices in the cloud, data engineers need to watch out for other aspects of cloud pricing and incentives that frequently catch users unawares.

Vendors want to lock you into their offerings. Getting data onto the platform is cheap or free on most cloud platforms, but getting data out can be extremely expensive. Be aware of data egress fees and their long-term impacts on your business before getting blindsided by a large bill. *Data gravity* is real: once data lands in a cloud, the cost to extract it and migrate processes can be very high.

Hybrid Cloud

As more established businesses migrate into the cloud, the hybrid cloud model is growing in importance. Virtually no business can migrate all of its workloads overnight. The hybrid cloud model assumes that an organization will indefinitely maintain some workloads outside the cloud.

There are many reasons to consider a hybrid cloud model. Organizations may believe that they have achieved operational excellence in certain areas, such as their application stack and associated hardware. Thus, they may migrate only to specific workloads where they see immediate benefits in the cloud environment. For example, an on-premises Spark stack is migrated to ephemeral cloud clusters, reducing the operational burden of managing software and hardware for the data engineering team and allowing rapid scaling for large data jobs.

[50] This is a major point of emphasis in Storment and Fuller, *Cloud FinOps*.

This pattern of putting analytics in the cloud is beautiful because data flows primarily in one direction, minimizing data egress costs (Figure 4-3). That is, on-premises applications generate event data that can be pushed to the cloud essentially for free. The bulk of data remains in the cloud where it is analyzed, while smaller amounts of data are pushed back to on premises for deploying models to applications, reverse ETL, etc.

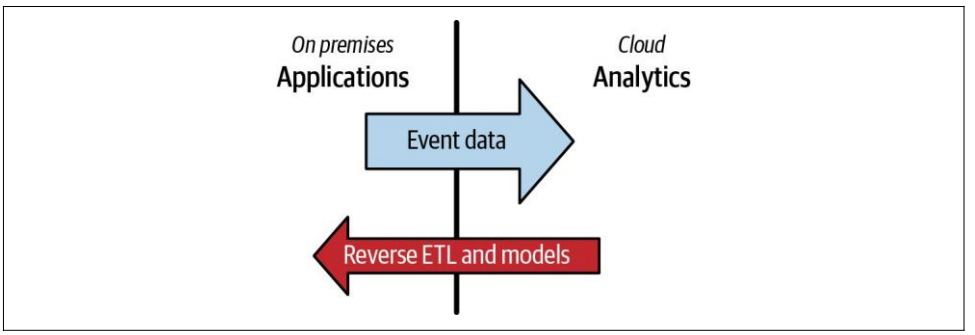


Figure 4-3. A hybrid cloud data flow model minimizing egress costs

A new generation of managed hybrid cloud service offerings also allows customers to locate cloud-managed servers in their data centers.[51] This gives users the ability to incorporate the best features in each cloud alongside on-premises infrastructure.

Multicloud

Multicloud simply refers to deploying workloads to multiple public clouds. Companies may have several motivations for multicloud deployments. SaaS platforms often wish to offer services close to existing customer cloud workloads. Snowflake and Databricks provide their SaaS offerings across multiple clouds for this reason. This is especially critical for data-intensive applications, where network latency and bandwidth limitations hamper performance, and data egress costs can be prohibitive.

Another common motivation for employing a multicloud approach is to take advantage of the best services across several clouds. For example, a company might want to handle its Google Ads and Analytics data on Google Cloud and deploy Kubernetes through GKE. And the company might also adopt Azure specifically for Microsoft workloads. Also, the company may like AWS because it has several best-in-class services (e.g., AWS Lambda) and enjoys huge mindshare, making it relatively easy to hire AWS-proficient engineers. Any mix of various cloud provider services is possible.

[51] Examples include Google Cloud Anthos and AWS Outposts.

Given the intense competition among the major cloud providers, expect them to offer more best-of-breed services, making multicloud more compelling.

A multicloud methodology has several disadvantages. As we just mentioned, data egress costs and networking bottlenecks are critical. Going multicloud can introduce significant complexity. Companies must now manage a dizzying array of services across several clouds; cross-cloud integration and security present a considerable challenge; multicloud networking can be diabolically complicated.

A new generation of "cloud of clouds" services aims to facilitate multicloud with reduced complexity by offering services across clouds and seamlessly replicating data between clouds or managing workloads on several clouds through a single pane of glass. To cite one example, a Snowflake account runs in a single cloud region, but customers can readily spin up other accounts in GCP, AWS, or Azure. Snowflake provides simple scheduled data replication between these various cloud accounts. The Snowflake interface is essentially the same in all of these accounts, removing the training burden of switching between cloud-native data services.

The "cloud of clouds" space is evolving quickly; within a few years of this book's publication, many more of these services will be available. Data engineers and architects would do well to maintain awareness of this quickly changing cloud landscape.

Decentralized: Blockchain and the Edge

Though not widely used now, it's worth briefly mentioning a new trend that might become popular over the next decade: decentralized computing. Whereas today's applications mainly run on premises and in the cloud, the rise of blockchain, Web 3.0, and edge computing may invert this paradigm. For the moment, decentralized platforms have proven extremely popular but have not had a significant impact in the data space; even so, keeping an eye on these platforms is worthwhile as you assess technology decisions.

Our Advice

From our perspective, we are still at the beginning of the transition to the cloud. Thus the evidence and arguments around workload placement and migration are in flux. The cloud itself is changing, with a shift from the IaaS model built around Amazon EC2 that drove the early growth of AWS and more generally toward more managed service offerings such as AWS Glue, Google BigQuery, and Snowflake.

We've also seen the emergence of new workload placement abstractions. On-premises services are becoming more cloud-like and abstracted. Hybrid cloud services allow customers to run fully managed services within their walls while facilitating tight

integration between local and remote environments. Further, the "cloud of clouds" is beginning to take shape, fueled by third-party services and public cloud vendors.

Choose technologies for the present, but look toward the future

As we mentioned in "Today Versus the Future: Immutable Versus Transitory Technologies" on page 120, you need to keep one eye on the present while planning for unknowns. Right now is a tough time to plan workload placements and migrations. Because of the fast pace of competition and change in the cloud industry, the decision space will look very different in five to ten years. It is tempting to take into account every possible future architecture permutation.

We believe that it is critical to avoid this endless trap of analysis. Instead, plan for the present. Choose the best technologies for your current needs and concrete plans for the near future. Choose your deployment platform based on real business needs while focusing on simplicity and flexibility.

In particular, don't choose a complex multicloud or hybrid-cloud strategy unless there's a compelling reason. Do you need to serve data near customers on multiple clouds? Do industry regulations require you to house certain data in your data centers? Do you have a compelling technology need for specific services on two different clouds? Choose a single-cloud deployment strategy if these scenarios don't apply to you.

On the other hand, have an escape plan. As we've emphasized before, every technology—even open source software—comes with some degree of lock-in. A single-cloud strategy has significant advantages of simplicity and integration but comes with significant lock-in attached. In this instance, we're talking about mental flexibility, the flexibility to evaluate the current state of the world and imagine alternatives. Ideally, your escape plan will remain locked behind glass, but preparing this plan will help you to make better decisions in the present and give you a way out if things go wrong in the future.

Cloud Repatriation Arguments

As we wrote this book, Sarah Wang and Martin Casado published "The Cost of Cloud, A Trillion Dollar Paradox", an article that generated significant sound and fury in the tech space. Readers widely interpreted the article as a call for the repatriation of cloud workloads to on-premises servers. They make a somewhat more subtle argument that companies should expend significant resources to control cloud spending and should consider repatriation as a possible option.

We want to take a moment to dissect one part of their discussion. Wang and Casado cite Dropbox's repatriation of significant workloads from AWS to Dropbox-owned servers as a case study for companies considering similar repatriation moves.

You are not Dropbox, nor are you Cloudflare

We believe that this case study is frequently used without appropriate context and is a compelling example of the *false equivalence* logical fallacy. Dropbox provides particular services where ownership of hardware and data centers can offer a competitive advantage. Companies should not rely excessively on Dropbox's example when assessing cloud and on-premises deployment options.

First, it's important to understand that Dropbox stores enormous amounts of data. The company is tight-lipped about exactly how much data it hosts but says it is many exabytes and continues to grow.

Second, Dropbox handles a vast amount of network traffic. We know that its bandwidth consumption in 2017 was significant enough for the company to add "hundreds of gigabits of internet connectivity with transit providers (regional and global ISPs), and hundreds of new peering partners (where we exchange traffic directly rather than through an ISP)."[52] The data egress costs would be extremely high in a public cloud environment.

Third, Dropbox is essentially a cloud storage vendor, but one with a highly specialized storage product that combines object and block storage characteristics. Dropbox's core competence is a differential file-update system that can efficiently synchronize actively edited files among users while minimizing network and CPU usage. The product is not a good fit for object storage, block storage, or other standard cloud offerings. Dropbox has instead benefited from building a custom, highly integrated software and hardware stack.[53]

Fourth, while Dropbox moved its core product to its hardware, it continued building out other AWS workloads. This allows Dropbox to focus on building one highly tuned cloud service at an extraordinary scale rather than trying to replace multiple services. Dropbox can focus on its core competence in cloud storage and data synchronization while offloading software and hardware management in other areas, such as data analytics.[54]

Other frequently cited success stories that companies have built outside the cloud include Backblaze and Cloudflare, but these offer similar lessons. Backblaze began life as a personal cloud data backup product but has since begun to offer B2, an object storage service similar to Amazon S3. Backblaze currently stores over an exabyte of

---

[52] Raghav Bhargava, "Evolution of Dropbox's Edge Network," Dropbox.Tech, June 19, 2017, *https://oreil.ly/RAwPf*.

[53] Akhil Gupta, "Scaling to Exabytes and Beyond," Dropbox.Tech, March 14, 2016, *https://oreil.ly/5XPKv*.

[54] "Dropbox Migrates 34 PB of Data to an Amazon S3 Data Lake for Analytics," AWS website, 2020, *https://oreil.ly/wpVoM*.

data. Cloudflare claims to provide services for over 25 million internet properties, with points of presence in over 200 cities and 51 terabits per second (Tbps) of total network capacity.

Netflix offers yet another useful example. The company is famous for running its tech stack on AWS, but this is only partially true. Netflix does run video transcoding on AWS, accounting for roughly 70% of its compute needs in 2017.[55] Netflix also runs its application backend and data analytics on AWS. However, rather than using the AWS content distribution network, Netflix has built a custom CDN in collaboration with internet service providers, utilizing a highly specialized combination of software and hardware. For a company that consumes a substantial slice of all internet traffic,[56]building out this critical infrastructure allowed it to deliver high-quality video to a huge customer base cost-effectively.

These case studies suggest that it makes sense for companies to manage their own hardware and network connections in particular circumstances. The biggest modern success stories of companies building and maintaining hardware involve extraordinary scale (exabytes of data, terabits per second of bandwidth, etc.) and limited use cases where companies can realize a competitive advantage by engineering highly integrated hardware and software stacks. In addition, all of these companies consume massive network bandwidth, suggesting that data egress charges would be a major cost if they chose to operate fully from a public cloud.

Consider continuing to run workloads on premises or repatriating cloud workloads if you run a truly cloud-scale service. What is cloud scale? You might be at cloud scale if you are storing an exabyte of data or handling terabits per second of traffic *to and from the internet*. (Achieving a terabit per second of *internal* network traffic is fairly easy.) In addition, consider owning your servers if data egress costs are a major factor for your business. To give a concrete example of cloud scale workloads that could benefit from repatriation, Apple might gain a significant financial and performance advantage by migrating iCloud storage to its own servers.[57]

Build Versus Buy

[55] Todd Hoff, "The Eternal Cost Savings of Netflix's Internal Spot Market," High Scalability, December 4, 2017, *https://oreil.ly/LLoFt*.

[56] Todd Spangler, "Netflix Bandwidth Consumption Eclipsed by Web Media Streaming Applications," *Variety*, September 10, 2019, *https://oreil.ly/tTm3k*.

[57] Amir Efrati and Kevin McLaughlin, "Apple's Spending on Google Cloud Storage on Track to Soar 50% This Year," *The Information*, June 29, 2021, *https://oreil.ly/OlFyR*.

Build versus buy is an age-old debate in technology. The argument for building is that you have end-to-end control over the solution and are not at the mercy of a vendor or open source community. The argument supporting buying comes down to resource constraints and expertise; do you have the expertise to build a

better solution than something already available? Either decision comes down to TCO, TOCO, and whether the solution provides a competitive advantage to your organization.

If you've caught on to a theme in the book so far, it's that we suggest investing in building and customizing *when doing so will provide a competitive advantage* for your business. Otherwise, stand on the shoulders of giants and *use what's already available* in the market. Given the number of open source and paid services—both of which may have communities of volunteers or highly paid teams of amazing engineers— you're foolish to build everything yourself.

As we often ask, "When you need new tires for your car, do you get the raw materials, create the tires from scratch, and install them yourself?" Like most people, you're probably buying tires and having someone install them. The same argument applies to build versus buy. We've seen teams that have built their databases from scratch. A simple open source RDBMS would have served their needs much better upon closer inspection. Imagine the amount of time and money invested in this homegrown database. Talk about low ROI for TCO and opportunity cost.

This is where the distinction between the type A and type B data engineer comes in handy. As we pointed out earlier, type A and type B roles are often embodied in the same engineer, especially in a small organization. Whenever possible, lean toward type A behavior; avoid undifferentiated heavy lifting and embrace abstraction. Use open source frameworks, or if this is too much trouble, look at buying a suitable managed or proprietary solution. Plenty of great modular services are available to choose from in either case.

The shifting reality of how companies adopt software is worth mentioning. Whereas in the past, IT used to make most of the software purchase and adoption decisions in a top-down manner, these days, the trend is for bottom-up software adoption in a company, driven by developers, data engineers, data scientists, and other technical roles. Technology adoption within companies is becoming an organic, continuous

process.

Let's look at some options for open source and proprietary solutions.

Open Source Software

Open source software (OSS) is a software distribution model in which software, and the underlying codebase, is made available for general use, typically under specific licensing terms. Often OSS is created and maintained by a distributed team of collaborators. OSS is free to use, change, and distribute most of the time, but with

specific caveats. For example, many licenses require that the source code of open source–derived software be included when the software is distributed.

The motivations for creating and maintaining OSS vary. Sometimes OSS is organic, springing from the mind of an individual or a small team that creates a novel solution and chooses to release it into the wild for public use. Other times, a company may make a specific tool or technology available to the public under an OSS license.

OSS has two main flavors: community managed and commercial OSS.

Community-managed OSS

OSS projects succeed with a strong community and vibrant user base. *Communitymanaged OSS* is a prevalent path for OSS projects. The community opens up high rates of innovations and contributions from developers worldwide with popular OSS projects.

The following are factors to consider with a community-managed OSS project:

Mindshare
: Avoid adopting OSS projects that don't have traction and popularity. Look at the number of GitHub stars, forks, and commit volume and recency. Another thing to pay attention to is community activity on related chat groups and forums. Does the project have a strong sense of community? A strong community creates a virtuous cycle of strong adoption. It also means that you'll have an easier time getting technical assistance and finding talent qualified to work with the framework.

Maturity
: How long has the project been around, how active is it today, and how usable are people finding it in production? A project's maturity indicates that people find it useful and are willing to incorporate it into their production workflows.

Troubleshooting
: How will you have to handle problems if they arise? Are you on your own to troubleshoot issues, or can the community help you solve your problem?

Project management
: Look at Git issues and the way they're addressed. Are they addressed quickly? If so, what's the process to submit an issue and get it resolved?

Team
: Is a company sponsoring the OSS project? Who are the core contributors?

Developer relations and community management

What is the project doing to encourage uptake and adoption? Is there a vibrant chat community (e.g., in Slack) that provides encouragement and support?

Contributing
: Does the project encourage and accept pull requests? What are the process and timelines for pull requests to be accepted and included in main codebase?

Roadmap
: Is there a project roadmap? If so, is it clear and transparent?

Self-hosting and maintenance
: Do you have the resources to host and maintain the OSS solution? If so, what's the TCO and TOCO versus buying a managed service from the OSS vendor?

Giving back to the community
: If you like the project and are actively using it, consider investing in it. You can contribute to the codebase, help fix issues, and give advice in the community forums and chats. If the project allows donations, consider making one. Many OSS projects are essentially community-service projects, and the maintainers often have full-time jobs in addition to helping with the OSS project. Sadly, it's often a labor of love that doesn't afford the maintainer a living wage. If you can afford to donate, please do so.

Commercial OSS

Sometimes OSS has some drawbacks. Namely, you have to host and maintain the solution in your environment. This may be trivial or extremely complicated and cumbersome, depending on the OSS application. Commercial vendors try to solve this management headache by hosting and managing the OSS solution for you, typically as a cloud SaaS offering. Examples of such vendors include Databricks (Spark), Confluent (Kafka), DBT Labs (dbt), and there are many, many others.

This model is called *commercial OSS* (COSS). Typically, a vendor will offer the "core" of the OSS for free while charging for enhancements, curated code distributions, or fully managed services.

A vendor is often affiliated with the community OSS project. As an OSS project becomes more popular, the maintainers may create a separate business for a managed version of the OSS. This typically becomes a cloud SaaS platform built around a managed version of the open source code. This is a widespread trend: an OSS project becomes popular, an affiliated company raises truckloads of venture capital (VC) money to commercialize the OSS project, and the company scales as a fast-moving rocket ship.

At this point, the data engineer has two options. You can continue using the community-managed OSS version, which you need to continue maintaining on your own (updates, server/container maintenance, pull requests for bug fixes, etc.). Or, you can pay the vendor and let it take care of the administrative management of the COSS product.

The following are factors to consider with a commercial OSS project:

Value
: Is the vendor offering a better value than if you managed the OSS technology yourself? Some vendors will add many bells and whistles to their managed offerings that aren't available in the community OSS version. Are these additions compelling to you?

Delivery model
: How do you access the service? Is the product available via download, API, or web/mobile UI? Be sure you can easily access the initial version and subsequent releases.

Support
: Support cannot be understated, and it's often opaque to the buyer. What is the support model for the product, and is there an extra cost for support? Frequently, vendors will sell support for an additional fee. Be sure you clearly understand the costs of obtaining support. Also, understand what is covered in support, and what is not covered. Anything that's not covered by support will be your responsibility to own and manage.

Releases and bug fixes
: Is the vendor transparent about the release schedule, improvements, and bug fixes? Are these updates easily available to you?

Sales cycle and pricing
: Often a vendor will offer on-demand pricing, especially for a SaaS product, and offer you a discount if you commit to an extended agreement. Be sure to understand the trade-offs of paying as you go versus paying up front. Is it worth paying a lump sum, or is your money better spent elsewhere?

Company finances
: Is the company viable? If the company has raised VC funds, you can check their funding on sites like Crunchbase. How much runway does the company have, and will it still be in business in a couple of years?

Logos versus revenue
: Is the company focused on growing the number of customers (logos), or is it trying to grow revenue? You may be surprised by the number of companies primarily

concerned with growing their customer count, GitHub stars, or Slack channel membership without the revenue to establish sound finances.

Community support
: Is the company truly supporting the community version of the OSS project? How much is the company contributing to the community OSS codebase? Controversies have arisen with certain vendors co-opting OSS projects and subsequently providing little value back to the community. How likely will the product remain viable as a community-supported open source if the company shuts down?

Note also that clouds offer their own managed open source products. If a cloud vendor sees traction with a particular product or project, expect that vendor to offer its version. This can range from simple examples (open source Linux offered on VMs) to extremely complex managed services (fully managed Kafka). The motivation for these offerings is simple: clouds make their money through consumption. More offerings in a cloud ecosystem mean a greater chance of "stickiness" and increased customer spending.

Proprietary Walled Gardens

While OSS is ubiquitous, a big market also exists for non-OSS technologies. Some of the biggest companies in the data industry sell closed source products. Let's look at two major types of *proprietary walled gardens*, independent companies and cloudplatform offerings.

Independent offerings

The data-tool landscape has seen exponential growth over the last several years. Every day, new independent offerings arise for data tools. With the ability to raise funds from VCs flush with capital, these data companies can scale and hire great engineering, sales, and marketing teams. This presents a situation where users have some great product choices in the marketplace while having to wade through endless sales and marketing clutter. At the time of this writing, the good times of freely available capital for data companies are coming to an end, but that's another long story whose consequences are still unfolding.

Often a company selling a data tool will not release it as OSS, instead offering a proprietary solution. Although you won't have the transparency of a pure OSS solution, a proprietary independent solution can work quite well, especially as a fully managed service in the cloud.

The following are things to consider with an independent offering:

Interoperability
: Make sure that the tool interoperates with other tools you've chosen (OSS, other independents, cloud offerings, etc.). Interoperability is key, so make sure you can try it before you buy.

Mindshare and market share
: Is the solution popular? Does it command a presence in the marketplace? Does it enjoy positive customer reviews?

Documentation and support
: Problems and questions will inevitably arise. Is it clear how to solve your problem, either through documentation or support?

Pricing
: Is the pricing understandable? Map out low-, medium-, and high-probability usage scenarios, with respective costs. Are you able to negotiate a contract, along with a discount? Is it worth it? How much flexibility do you lose if you sign a contract, both in negotiation and the ability to try new options? Are you able to obtain contractual commitments on future pricing?

Longevity
: Will the company survive long enough for you to get value from its product? If the company has raised money, search around for its funding situation. Look at user reviews. Ask friends and post questions on social networks about users' experiences with the product. Make sure you know what you're getting into.

Cloud platform proprietary service offerings

Cloud vendors develop and sell their proprietary services for storage, databases, and more. Many of these solutions are internal tools used by respective sibling companies. For example, Amazon created the database DynamoDB to overcome the limitations of traditional relational databases and handle the large amounts of user and order data as Amazon.com grew into a behemoth. Amazon later offered the DynamoDB service solely on AWS; it's now a top-rated product used by companies of all sizes and maturity levels. Cloud vendors will often bundle their products to work well together. Each cloud can create stickiness with its user base by creating a strong integrated ecosystem.

The following are factors to consider with a proprietary cloud offering:

Performance versus price comparisons
: Is the cloud offering substantially better than an independent or OSS version? What's the TCO of choosing a cloud's offering?

Purchase considerations

On-demand pricing can be expensive. Can you lower your cost by purchasing reserved capacity or entering into a long-term commitment agreement?

Our Advice

Build versus buy comes back to knowing your competitive advantage and where it makes sense to invest resources toward customization. In general, we favor OSS and COSS by default, which frees you to focus on improving those areas where these options are insufficient. Focus on a few areas where building something will add significant value or reduce friction substantially.

Don't treat internal operational overhead as a sunk cost. There's excellent value in upskilling your existing data team to build sophisticated systems on managed platforms rather than babysitting on-premises servers. In addition, think about how a company makes money, especially its sales and customer experience teams, which will generally indicate how you're treated during the sales cycle and when you're a paying customer.

Finally, who is responsible for the budget at your company? How does this person decide the projects and technologies that get funded? Before making the business case for COSS or managed services, does it make sense to try to use OSS first? The last thing you want is for your technology choice to be stuck in limbo while waiting for budget approval. As the old saying goes, *time kills deals*. In your case, more time spent in limbo means a higher likelihood your budget approval will die. Know beforehand who controls the budget and what will successfully get approved.

Monolith Versus Modular

Monoliths versus modular systems is another longtime debate in the software architecture space. Monolithic systems are self-contained, often performing multiple functions under a single system. The monolith camp favors the simplicity of having everything in one place. It's easier to reason about a single entity, and you can move faster because there are fewer moving parts. The *modular* camp leans toward decoupled, best-of-breed technologies performing tasks at which they are uniquely great. Especially given the rate of change in products in the data world, the argument is you should aim for interoperability among an ever-changing array of solutions.

What approach should you take in your data engineering stack? Let's explore the trade-offs.

Monolith

The *monolith* (Figure 4-4) has been a technology mainstay for decades. The old days of waterfall meant that software releases were huge, tightly coupled, and moved at a slow cadence. Large teams worked together to deliver a single working codebase. Monolithic data systems continue to this day, with older software vendors such as Informatica and open source frameworks such as Spark.

The pros of the monolith are it's easy to reason about, and it requires a lower cognitive burden and context switching since everything is self-contained. Instead of dealing with dozens of technologies, you deal with "one" technology and typically one principal programming language. Monoliths are an excellent option if you want simplicity in reasoning about your architecture and processes.

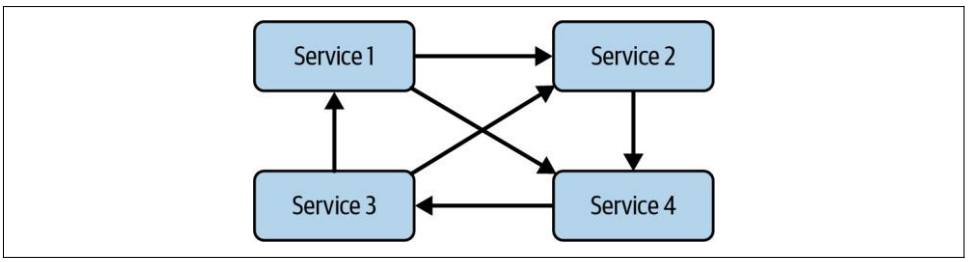


Figure 4-4. The monolith tightly couples its services

Of course, the monolith has cons. For one thing, monoliths are brittle. Because of the vast number of moving parts, updates and releases take longer and tend to bake in "the kitchen sink." If the system has a bug—hopefully, the software's been thoroughly tested before release!—it can harm the entire system.

User-induced problems also happen with monoliths. For example, we saw a monolithic ETL pipeline that took 48 hours to run. If anything broke anywhere in the pipeline, the entire process had to restart. Meanwhile, anxious business users were waiting for their reports, which were already two days late by default and usually arrived much later. Breakages were common enough that the monolithic system was eventually thrown out.

Multitenancy in a monolithic system can also be a significant problem. It can be challenging to isolate the workloads of multiple users. In an on-prem data warehouse, one user-defined function might consume enough CPU to slow the system for other users. Conflicts between dependencies and resource contention are frequent sources of headaches.

Another con of monoliths is that switching to a new system will be painful if the vendor or open source project dies. Because all of your processes are contained in the monolith, extracting yourself out of that system, and onto a new platform, will be costly in both time and money.

Modularity

Modularity (Figure 4-5) is an old concept in software engineering, but modular distributed systems truly came into vogue with the rise of microservices. Instead of relying on a massive monolith to handle your needs, why not break apart systems and processes into their self-contained areas of concern? Microservices can communicate via APIs, allowing developers to focus on their domains while making their applications accessible to other microservices. This is the trend in software engineering and is increasingly seen in modern data systems.

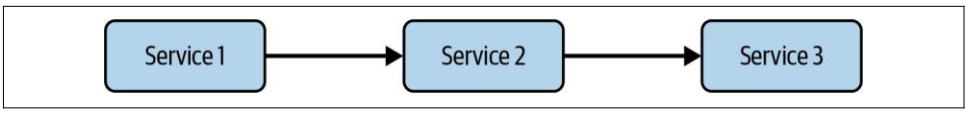


Figure 4-5. With modularity, each service is decoupled from another

Major tech companies have been key drivers in the microservices movement. The famous Bezos API mandate decreases coupling between applications, allowing refactoring and decomposition. Bezos also imposed the two-pizza rule (no team should be so large that two pizzas can't feed the whole group). Effectively, this means that a team will have at most five members. This cap also limits the complexity of a team's domain of responsibility—in particular, the codebase that it can manage. Whereas an extensive monolithic application might entail a group of one hundred people, dividing developers into small groups of five requires that this application be broken into small, manageable, loosely coupled pieces.

In a modular microservice environment, components are swappable, and it's possible to create a *polyglot* (multiprogramming language) application; a Java service can replace a service written in Python. Service customers need worry only about the technical specifications of the service API, not behind-the-scenes details of

implementation.

Data-processing technologies have shifted toward a modular model by providing strong support for interoperability. Data is stored in object storage in a standard format such as Parquet in data lakes and lakehouses. Any processing tool that supports the format can read the data and write processed results back into the lake for processing by another tool. Cloud data warehouses support interoperation with object storage through import/export using standard formats and external tables— i.e., queries run directly on data in a data lake.

New technologies arrive on the scene at a dizzying rate in today's data ecosystem, and most get stale and outmoded quickly. Rinse and repeat. The ability to swap out tools as technology changes is invaluable. We view data modularity as a more powerful paradigm than monolithic data engineering. Modularity allows engineers to choose the best technology for each job or step along the pipeline.

The cons of modularity are that there's more to reason about. Instead of handling a single system of concern, now you potentially have countless systems to understand and operate. Interoperability is a potential headache; hopefully, these systems all play nicely together.

This very problem led us to break out orchestration as a separate undercurrent instead of placing it under data management. Orchestration is also important for monolithic data architectures; witness the success of tools like BMC Software's Control-M in the traditional data warehousing space. But orchestrating five or ten

tools is dramatically more complex than orchestrating one. Orchestration becomes the glue that binds data stack modules together.

The Distributed Monolith Pattern

The *distributed monolith pattern* is a distributed architecture that still suffers from many of the limitations of monolithic architecture. The basic idea is that one runs a distributed system with different services to perform different tasks. Still, services and nodes share a common set of dependencies or a common codebase.

One standard example is a traditional Hadoop cluster. A Hadoop cluster can simultaneously host several frameworks, such as Hive, Pig, or Spark. The cluster also has many internal dependencies. In addition, the cluster runs core Hadoop components: Hadoop common libraries, HDFS, YARN, and Java. In practice, a cluster often has one version of each component installed.

A standard on-prem Hadoop system entails managing a common environment that works for all users and all jobs. Managing upgrades and installations is a significant challenge. Forcing jobs to upgrade dependencies risks breaking them; maintaining two versions of a framework entails extra complexity.

Some modern Python-based orchestration technologies—e.g., Apache Airflow—also suffer from this problem. While they utilize a highly decoupled and asynchronous architecture, every service runs the same codebase with the same dependencies. Any executor can execute any task, so a client library for a single task run in one DAG must be installed on the whole cluster. Orchestrating many tools entails installing client libraries for a host of APIs. Dependency conflicts are a constant problem.

One solution to the problems of the distributed monolith is ephemeral infrastructure in a cloud setting. Each job gets its own temporary server or cluster installed with dependencies. Each cluster remains highly monolithic, but separating jobs dramatically reduces conflicts. For example, this pattern is now quite common for Spark with services like Amazon EMR and Google Cloud Dataproc.

A second solution is to properly decompose the distributed monolith into multiple software environments using containers. We have more to say on containers in "Serverless Versus Servers" on page 143.

Our Advice

While monoliths are attractive because of ease of understanding and reduced complexity, this comes at a high cost. The cost is the potential loss of flexibility, opportunity cost, and high-friction development cycles.

Here are some things to consider when evaluating monoliths versus modular options:

Interoperability
: Architect for sharing and interoperability.

Avoiding the "bear trap"
: Something that is easy to get into might be painful or impossible to escape.

Flexibility
: Things are moving so fast in the data space right now. Committing to a monolith reduces flexibility and reversible decisions.

Serverless Versus Servers

A big trend for cloud providers is *serverless*, allowing developers and data engineers to run applications without managing servers behind the scenes. Serverless provides a quick time to value for the right use cases. For other cases, it might not be a good fit. Let's look at how to evaluate whether serverless is right for you.

Serverless

Though serverless has been around for quite some time, the serverless trend kicked off in full force with AWS Lambda in 2014. With the promise of executing small chunks of code on an as-needed basis without having to manage a server, serverless exploded in popularity. The main reasons for its popularity are cost and convenience. Instead of paying the cost of a server, why not just pay when your code is evoked?

Serverless has many flavors. Though function as a service (FaaS) is wildly popular, serverless systems predate the advent of AWS Lambda. For example, Google Cloud's BigQuery is serverless in that data engineers don't need to manage backend infrastructure, and the system scales to zero and scales up automatically to handle large queries. Just load data into the system and start querying. You pay for the amount of data your query consumes and a small cost to store your data. This payment model—paying for consumption and storage—is becoming more prevalent. When does serverless make sense? As with many other cloud services, it depends; and data engineers would do well to understand the details of cloud pricing to predict when serverless deployments will become expensive. Looking specifically at the case of AWS Lambda, various engineers have found hacks to run batch workloads at meager costs.[58] On the other hand, serverless functions suffer from an inherent overhead inefficiency. Handling one event per function call at a high event rate can be

[58] Evan Sangaline, "Running FFmpeg on AWS Lambda for 1.9% the Cost of AWS Elastic Transcoder," Intoli blog, May 2, 2018, *https://oreil.ly/myzOv*.

catastrophically expensive, especially when simpler approaches like multithreading or multiprocessing are great alternatives.

As with other areas of ops, it's critical to monitor and model. *Monitor* to determine cost per event in a real-world environment and maximum length of serverless execution, and *model* using this cost per event to determine overall costs as event rates grow. Modeling should also include worst-case scenarios—what happens if my site gets hit by a bot swarm or DDoS attack?

Containers

In conjunction with serverless and microservices, *containers* are one of the most powerful trending operational technologies as of this writing. Containers play a role in both serverless and microservices.

Containers are often referred to as *lightweight virtual machines*. Whereas a traditional VM wraps up an entire operating system, a container packages an isolated user space (such as a filesystem and a few processes); many such containers can coexist on a single host operating system. This provides some of the principal benefits of virtualization (i.e., dependency and code isolation) without the overhead of carrying around an entire operating system kernel.

A single hardware node can host numerous containers with fine-grained resource allocations. At the time of this writing, containers continue to grow in popularity, along with Kubernetes, a container management system. Serverless environments typically run on containers behind the scenes. Indeed, Kubernetes is a kind of serverless environment because it allows developers and ops teams to deploy microservices without worrying about the details of the machines where they are deployed. Containers provide a partial solution to problems of the distributed monolith mentioned earlier in this chapter. For example, Hadoop now supports containers, allowing each job to have its own isolated dependencies.

Container clusters do not provide the same security and isolation that full VMs offer. *Container escape*—broadly, a class of exploits whereby code in a container gains privileges outside the container at the OS level—is common enough to be considered a risk for multitenancy. While Amazon EC2 is a truly multitenant environment with VMs from many customers hosted on the same hardware, a Kubernetes cluster should host code only within an environment of mutual trust (e.g., inside the walls of a single company). In addition, code review processes and vulnerability scanning are critical to ensure that a developer doesn't introduce a security hole.

Various flavors of container platforms add additional serverless features. Containerized function platforms run containers as ephemeral units triggered by events rather than

persistent services.[59] This gives users the simplicity of AWS Lambda with the full flexibility of a container environment instead of the highly restrictive Lambda runtime. And services such as AWS Fargate and Google App Engine run containers without managing a compute cluster required for Kubernetes. These services also fully isolate containers, preventing the security issues associated with multitenancy.

Abstraction will continue working its way across the data stack. Consider the impact of Kubernetes on cluster management. While you can manage your Kubernetes cluster—and many engineering teams do so—even Kubernetes is widely available as a managed service. What comes after Kubernetes? We're as excited as you to find out.

How to Evaluate Server Versus Serverless

Why would you want to run your own servers instead of using serverless? There are a few reasons. Cost is a big factor. Serverless makes less sense when the usage and cost exceed the ongoing cost of running and maintaining a server (Figure 4-6). However, at a certain scale, the economic benefits of serverless may diminish, and running servers becomes more attractive.

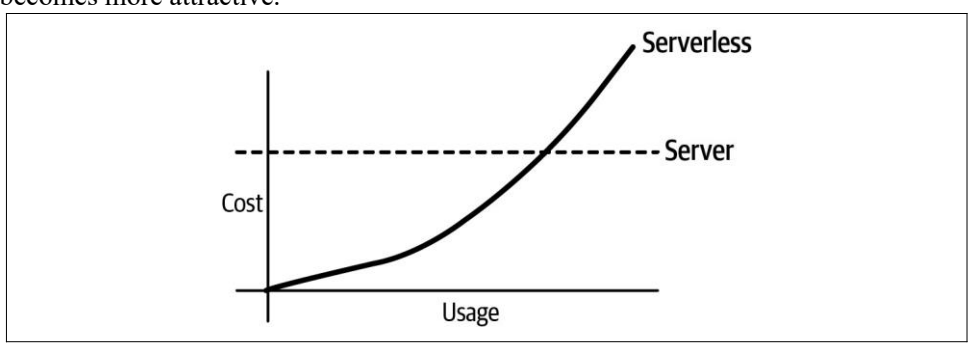


Figure 4-6. Cost of serverless versus utilizing a server

Customization, power, and control are other major reasons to favor servers over serverless. Some serverless frameworks can be underpowered or limited for certain use cases. Here are some things to consider when using servers, particularly in the cloud, where server resources are ephemeral:

Expect servers to fail.

Server failure will happen. Avoid using a "special snowflake" server that is overly customized and brittle, as this introduces a glaring vulnerability in your architecture. Instead, treat servers as ephemeral resources that you can create as needed and then delete. If your application requires specific code to be installed

[59] Examples include OpenFaaS, Knative, and Google Cloud Run.

on the server, use a boot script or build an image. Deploy code to the server through a CI/CD pipeline.

Use clusters and autoscaling.
: Take advantage of the cloud's ability to grow and shrink compute resources on demand. As your application increases its usage, cluster your application servers, and use autoscaling capabilities to automatically horizontally scale your application as demand grows.

Treat your infrastructure as code.
: Automation doesn't apply to just servers and should extend to your infrastructure whenever possible. Deploy your infrastructure (servers or otherwise) using deployment managers such as Terraform, AWS CloudFormation, and Google Cloud Deployment Manager.

Use containers.
: For more sophisticated or heavy-duty workloads with complex installed dependencies, consider using containers on either a single server or Kubernetes.

Our Advice

Here are some key considerations to help you determine whether serverless is right for you:

Workload size and complexity
: Serverless works best for simple, discrete tasks and workloads. It's not as suitable if you have many moving parts or require a lot of compute or memory horsepower. In that case, consider using containers and a container workflow orchestration framework like Kubernetes.

Execution frequency and duration
: How many requests per second will your serverless application process? How long will each request take to process? Cloud serverless platforms have limits on execution frequency, concurrency, and duration. If your application can't function neatly within these limits, it is time to consider a container-oriented approach.

Requests and networking
: Serverless platforms often utilize some form of simplified networking and don't support all cloud virtual networking features, such as VPCs and firewalls.

Language
: What language do you typically use? If it's not one of the officially supported languages supported by the serverless platform, you should consider containers instead.

Runtime limitations

Serverless platforms don't give you complete operating system abstractions. Instead, you're limited to a specific runtime image.

Cost

Serverless functions are incredibly convenient but potentially expensive. When your serverless function processes only a few events, your costs are low; costs rise rapidly as the event count increases. This scenario is a frequent source of surprise cloud bills.

In the end, abstraction tends to win. We suggest looking at using serverless first and then servers—with containers and orchestration if possible—once you've outgrown serverless options.

Optimization, Performance, and the Benchmark Wars

Imagine that you are a billionaire shopping for new transportation. You've narrowed your choice to two options:

- 787 Business Jet
  - — Range: 9,945 nautical miles (with 25 passengers)
  - — Maximum speed: 0.90 Mach
  - — Cruise speed: 0.85 Mach
  - — Fuel capacity: 101,323 kilograms
  - — Maximum takeoff weight: 227,930 kilograms
  - — Maximum thrust: 128,000 pounds
- Tesla Model S Plaid
  - — Range: 560 kilometers
  - — Maximum speed: 322 kilometers/hour
  - — 0–100 kilometers/hour: 2.1 seconds
  - — Battery capacity: 100 kilowatt hours
  - — Nurburgring lap time: 7 minutes, 30.9 seconds
  - — Horsepower: 1020
  - — Torque: 1050 lb-ft

Which of these options offers better performance? You don't have to know much about cars or aircraft to recognize that this is an idiotic comparison. One option is a wide-

body private jet designed for intercontinental operation, while the other is an electric supercar.

We see such apples-to-oranges comparisons made all the time in the database space. Benchmarks either compare databases that are optimized for completely different use cases, or use test scenarios that bear no resemblance to real-world needs.

Recently, we saw a new round of benchmark wars flare up among major vendors in the data space. We applaud benchmarks and are glad to see many database vendors finally dropping DeWitt clauses from their customer contracts.[60] Even so, let the buyer beware: the data space is full of nonsensical benchmarks.[61] Here are a few common tricks used to place a thumb on the benchmark scale.

Big Data...for the 1990s

Products that claim to support "big data" at petabyte scale will often use benchmark datasets small enough to easily fit in the storage on your smartphone. For systems that rely on caching layers to deliver performance, test datasets fully reside in solidstate drive (SSD) or memory, and benchmarks can show ultra-high performance by repeatedly querying the same data. A small test dataset also minimizes RAM and SSD costs when comparing pricing.

To benchmark for real-world use cases, you must simulate anticipated real-world data and query size. Evaluate query performance and resource costs based on a detailed evaluation of your needs.

Nonsensical Cost Comparisons

Nonsensical cost comparisons are a standard trick when analyzing a price/performance or TCO. For instance, many MPP systems can't be readily created and deleted even when they reside in a cloud environment; these systems run for years on end once they've been configured. Other databases support a dynamic compute model and charge either per query or per second of use. Comparing ephemeral and nonephemeral systems on a cost-per-second basis is nonsensical, but we see this all the time in benchmarks.

[60] Justin Olsson and Reynold Xin, "Eliminating the Anti-competitive DeWitt Clause for Database Benchmarking," Databricks, November 8, 2021, *https://oreil.ly/3iFOE*.

[61] For a classic of the genre, see William McKnight and Jake Dolezal, "Data Warehouse in the Cloud Benchmark," GigaOm, February 7, 2019, *https://oreil.ly/QjCmA*.

Asymmetric Optimization

The deceit of asymmetric optimization appears in many guises, but here's one example. Often a vendor will compare a row-based MPP system against a columnar database by using a benchmark that runs complex join queries on highly normalized data. The normalized data model is optimal for the row-based system, but the columnar system would realize its full potential only with some schema changes. To make matters worse, vendors juice their systems with an extra shot of join optimization (e.g., preindexing joins) without applying comparable tuning in the competing database (e.g., putting joins in a materialized view).

Caveat Emptor

As with all things in data technology, let the buyer beware. Do your homework before blindly relying on vendor benchmarks to evaluate and choose technology.

Undercurrents and Their Impacts on Choosing Technologies

As seen in this chapter, a data engineer has a lot to consider when evaluating technologies. Whatever technology you choose, be sure to understand how it supports the undercurrents of the data engineering lifecycle. Let's briefly review them again.

Data Management

Data management is a broad area, and concerning technologies, it isn't always apparent whether a technology adopts data management as a principal concern. For example, behind the scenes, a third-party vendor may use data management best practices—regulatory compliance, security, privacy, data quality, and governance— but hide these details behind a limited UI layer. In this case, while evaluating the product, it helps to ask the company about its data management practices. Here are some sample questions you should ask:

- How are you protecting data against breaches, both from the outside and within?
- What is your product's compliance with GDPR, CCPA, and other data privacy regulations?
- Do you allow me to host my data to comply with these regulations?
- How do you ensure data quality and that I'm viewing the correct data in your solution?

There are many other questions to ask, and these are just a few of the ways to think about data management as it relates to choosing the right technologies. These same questions should also apply to the OSS solutions you're considering.

DataOps

Problems will happen. They just will. A server or database may die, a cloud's region may have an outage, you might deploy buggy code, bad data might be introduced into your data warehouse, and other unforeseen problems may occur.

Undercurrents and Their Impacts on Choosing Technologies

When evaluating a new technology, how much control do you have over deploying new code, how will you be alerted if there's a problem, and how will you respond when there's a problem? The answer largely depends on the type of technology you're considering. If the technology is OSS, you're likely responsible for setting up monitoring, hosting, and code deployment. How will you handle issues? What's your incident response?

Much of the operations are outside your control if you're using a managed offering. Consider the vendor's SLA, the way they alert you to issues, and whether they're transparent about how they're addressing the case, including providing an ETA to a fix.

Data Architecture

As discussed in Chapter 3, good data architecture means assessing trade-offs and choosing the best tools for the job while keeping your decisions reversible. With the data landscape morphing at warp speed, the *best tool* for the job is a moving target. The main goals are to avoid unnecessary lock-in, ensure interoperability across the data stack, and produce high ROI. Choose your technologies accordingly.

Orchestration Example: Airflow

Throughout most of this chapter, we have actively avoided discussing any particular technology too extensively. We make an exception for orchestration because the space is currently dominated by one open source technology, Apache Airflow.

Maxime Beauchemin kicked off the Airflow project at Airbnb in 2014. Airflow was developed from the beginning as a noncommercial open source project. The framework quickly grew significant mindshare outside Airbnb, becoming an Apache Incubator project in 2016 and a full Apache-sponsored project in 2019.

Airflow enjoys many advantages, largely because of its dominant position in the open source marketplace. First, the Airflow open source project is extremely active, with a high rate of commits and a quick response time for bugs and security issues, and the

project recently released Airflow 2, a major refactor of the codebase. Second, Airflow enjoys massive mindshare. Airflow has a vibrant, active community on many communications platforms, including Slack, Stack Overflow, and GitHub. Users can easily find answers to questions and problems. Third, Airflow is available commercially as a managed service or software distribution through many vendors, including GCP, AWS, and Astronomer.io.

Airflow also has some downsides. Airflow relies on a few core nonscalable components (the scheduler and backend database) that can become bottlenecks for performance, scale, and reliability; the scalable parts of Airflow still follow a distributed monolith pattern. (See "Monolith Versus Modular" on page 139.) Finally, Airflow lacks support for many data-native constructs, such as schema management, lineage, and cataloging; and it is challenging to develop and test Airflow workflows.

We do not attempt an exhaustive discussion of Airflow alternatives here but just mention a couple of the key orchestration contenders at the time of writing. Prefect and Dagster aim to solve some of the problems discussed previously by rethinking components of the Airflow architecture. Will there be other orchestration frameworks and technologies not discussed here? Plan on it.

We highly recommend that anyone choosing an orchestration technology study the options discussed here. They should also acquaint themselves with activity in the space, as new developments will certainly occur by the time you read this.

Software Engineering

As a data engineer, you should strive for simplification and abstraction across the data stack. Buy or use prebuilt open source solutions whenever possible. Eliminating undifferentiated heavy lifting should be your big goal. Focus your resources—custom coding and tooling—on areas that give you a solid competitive advantage. For example, is hand-coding a database connection between your production database and your cloud data warehouse a competitive advantage for you? Probably not. This is very much a solved problem. Pick an off-the-shelf solution (open source or managed SaaS) instead. The world doesn't need the millionth +1 database-to-cloud data warehouse connector.

On the other hand, why do customers buy from you? Your business likely has something special about the way it does things. Maybe it's a particular algorithm that powers your fintech platform. By abstracting away a lot of the redundant workflows and processes, you can continue chipping away, refining, and customizing the things that move the needle for the business.

Conclusion

Choosing the right technologies is no easy task, especially when new technologies and patterns emerge daily. Today is possibly the most confusing time in history for evaluating and selecting technologies. Choosing technologies is a balance of use case, cost, build versus buy, and modularization. Always approach technology the same way as architecture: assess trade-offs and aim for reversible decisions.

Additional Resources

- *Cloud FinOps* by J. R. Storment and Mike Fuller (O'Reilly)
- "Cloud Infrastructure: The Definitive Guide for Beginners" by Matthew Smith

- "The Cost of Cloud, a Trillion Dollar Paradox" by Sarah Wang and Martin Casado
- FinOps Foundation's "What Is FinOps" web page
- "Red Hot: The 2021 Machine Learning, AI and Data (MAD) Landscape" by Matt Turck
- Ternary Data's "What's Next for Analytical Databases? w/ Jordan Tigani (MotherDuck)" video
- "The Unfulfilled Promise of Serverless" by Corey Quinn
- "What Is the Modern Data Stack?" by Charles Wang

PART II

The Data Engineering Lifecycle in Depth

CHAPTER 5

Data Generation in Source Systems

Welcome to the first stage of the data engineering lifecycle: data generation in source systems. As we described earlier, the job of a data engineer is to take data from source systems, do something with it, and make it helpful in serving downstream use cases. But before you get raw data, you must understand where the data exists, how it is generated, and its characteristics and quirks.

This chapter covers some popular operational source system patterns and the significant types of source systems. Many source systems exist for data generation, and we're not exhaustively covering them all. We'll consider the data these systems generate and things you should consider when working with source systems. We also discuss how the undercurrents of data engineering apply to this first phase of the data engineering lifecycle (Figure 5-1).

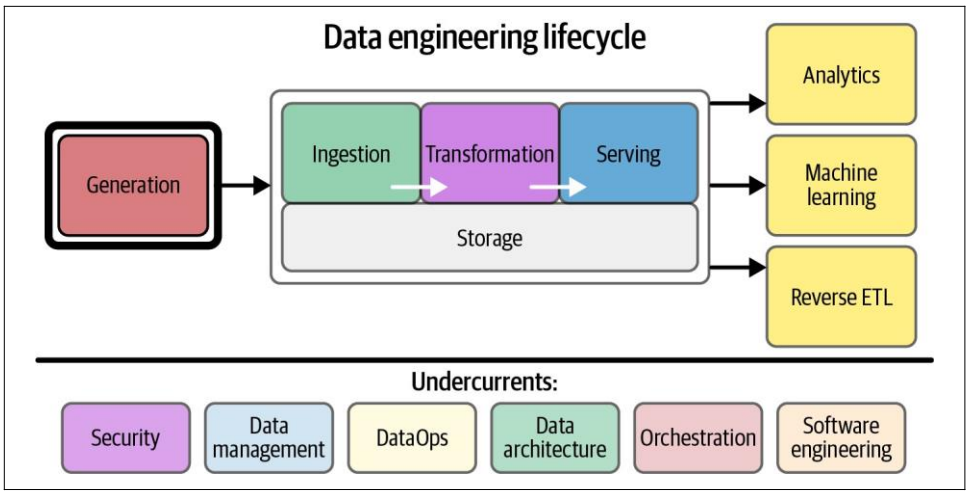


Figure 5-1. Source systems generate the data for the rest of the data engineering lifecycle As data proliferates, especially with the rise of data sharing (discussed next), we expect that a data engineer's role will shift heavily toward understanding the interplay between data sources and destinations. The basic plumbing tasks of data engineering—moving data from A to B—will simplify dramatically. On the other hand, it will remain critical to understand the nature of data as it's created in source systems.

Sources of Data: How Is Data Created?

As you learn about the various underlying operational patterns of the systems that generate data, it's essential to understand how data is created. Data is an unorganized, context-less collection of facts and figures. It can be created in many ways, both analog and digital.

Analog data creation occurs in the real world, such as vocal speech, sign language, writing on paper, or playing an instrument. This analog data is often transient; how often have you had a verbal conversation whose contents are lost to the ether after the conversation ends?

Digital data is either created by converting analog data to digital form or is the native product of a digital system. An example of analog to digital is a mobile texting app that converts analog speech into digital text. An example of digital data creation is a credit card transaction on an ecommerce platform. A customer places an order, the transaction is charged to their credit card, and the information for the transaction is saved to various databases.

We'll utilize a few common examples in this chapter, such as data created when interacting with a website or mobile application. But in truth, data is everywhere in the world around us. We capture data from IoT devices, credit card terminals, telescope sensors, stock trades, and more.

Get familiar with your source system and how it generates data. Put in the effort to read the source system documentation and understand its patterns and quirks. If your source system is an RDBMS, learn how it operates (writes, commits, queries, etc.); learn the ins and outs of the source system that might affect your ability to ingest from it.

Source Systems: Main Ideas

Source systems produce data in various ways. This section discusses the main ideas you'll frequently encounter as you work with source systems.

Files and Unstructured Data

A *file* is a sequence of bytes, typically stored on a disk. Applications often write data to files. Files may store local parameters, events, logs, images, and audio.

In addition, files are a universal medium of data exchange. As much as data engineers wish that they could get data programmatically, much of the world still sends and receives files. For example, if you're getting data from a government agency, there's an excellent chance you'll download the data as an Excel or CSV file or receive the file in an email.

The main types of source file formats you'll run into as a data engineer—files that originate either manually or as an output from a source system process—are Excel, CSV, TXT, JSON, and XML. These files have their quirks and can be structured (Excel, CSV), semistructured (JSON, XML, CSV), or unstructured (TXT, CSV). Although you'll use certain formats heavily as a data engineer (such as Parquet, ORC, and Avro), we'll cover these later and put the spotlight here on source system files. Chapter 6 covers the technical details of files.

APIs

Application programming interfaces (APIs) are a standard way of exchanging data between systems. In theory, APIs simplify the data ingestion task for data engineers. In practice, many APIs still expose a good deal of data complexity for engineers to manage. Even with the rise of various services and frameworks, and services for automating API data ingestion, data engineers must often invest a good deal of energy

into maintaining custom API connections. We discuss APIs in greater detail later in this chapter.

Application Databases (OLTP Systems)

An *application database* stores the state of an application. A standard example is a database that stores account balances for bank accounts. As customer transactions and payments happen, the application updates bank account balances.

Typically, an application database is an *online transaction processing* (OLTP) system— a database that reads and writes individual data records at a high rate. OLTP systems are often referred to as *transactional databases*, but this does not necessarily imply that the system in question supports *atomic transactions*.

More generally, OLTP databases support low latency and high concurrency. An RDBMS database can select or update a row in less than a millisecond (not accounting for network latency) and handle thousands of reads and writes per second. A document database cluster can manage even higher document commit rates at the expense of potential inconsistency. Some graph databases can also handle transactional use cases.

Fundamentally, OLTP databases work well as application backends when thousands or even millions of users might be interacting with the application simultaneously, updating and writing data concurrently. OLTP systems are less suited to use cases driven by analytics at scale, where a single query must scan a vast amount of data.

ACID

Support for atomic transactions is one of a critical set of database characteristics known together as ACID (as you may recall from Chapter 3, this stands for *atomicity, consistency, isolation, durability*). *Consistency* means that any database read will return the last written version of the retrieved item. *Isolation* entails that if two updates are in flight concurrently for the same thing, the end database state will be consistent with the sequential execution of these updates in the order they were submitted. *Durability* indicates that committed data will never be lost, even in the event of power loss.

Note that ACID characteristics are not required to support application backends, and relaxing these constraints can be a considerable boon to performance and scale. However, ACID characteristics guarantee that the database will maintain a consistent picture of the world, dramatically simplifying the app developer's task.

All engineers (data or otherwise) must understand operating with and without ACID. For instance, to improve performance, some distributed databases use relaxed consistency constraints, such as *eventual consistency*, to improve performance.

Understanding the consistency model you're working with helps you prevent disasters.

Atomic transactions

An *atomic transaction* is a set of several changes that are committed as a unit. In the example in Figure 5-2, a traditional banking application running on an RDBMS executes a SQL statement that checks two account balances, one in Account A (the source) and another in Account B (the destination). Money is then moved from Account A to Account B if sufficient funds are in Account A. The entire transaction should run with updates to both account balances or fail without updating either account balance. That is, the whole operation should happen as a *transaction*.

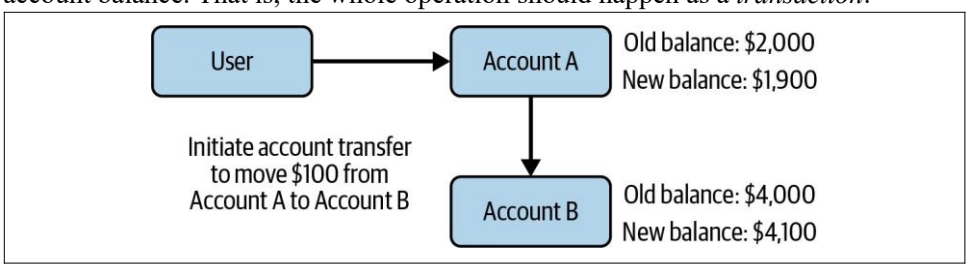


Figure 5-2. Example of an atomic transaction: a bank account transfer using OLTP

OLTP and analytics

Often, small companies run analytics directly on an OLTP. This pattern works in the short term but is ultimately not scalable. At some point, running analytical queries on OLTP runs into performance issues due to structural limitations of OLTP or resource contention with competing transactional workloads. Data engineers must understand the inner workings of OLTP and application backends to set up appropriate integrations with analytics systems without degrading production application performance.

As companies offer more analytics capabilities in SaaS applications, the need for hybrid capabilities—quick updates with combined analytics capabilities—has created new challenges for data engineers. We'll use the term *data application* to refer to applications that hybridize transactional and analytics workloads.

Online Analytical Processing System

In contrast to an OLTP system, an *online analytical processing* (OLAP) system is built to run large analytics queries and is typically inefficient at handling lookups of individual records. For example, modern column databases are optimized to scan large volumes of data, dispensing with indexes to improve scalability and scan performance. Any query typically involves scanning a minimal data block, often 100 MB or more in size. Trying to look up thousands of individual items per second in such a system will

bring it to its knees unless it is combined with a caching layer designed for this use case.

Note that we're using the term *OLAP* to refer to any database system that supports high-scale interactive analytics queries; we are not limiting ourselves to systems that support OLAP cubes (multidimensional arrays of data). The *online* part of OLAP implies that the system constantly listens for incoming queries, making OLAP systems suitable for interactive analytics.

Although this chapter covers source systems, OLAPs are typically storage and query systems for analytics. Why are we talking about them in our chapter on source systems? In practical use cases, engineers often need to read data from an OLAP system. For example, a data warehouse might serve data used to train an ML model. Or, an OLAP system might serve a reverse ETL workflow, where derived data in an analytics system is sent back to a source system, such as a CRM, SaaS platform, or transactional application.

Change Data Capture

Change data capture (CDC) is a method for extracting each change event (insert, update, delete) that occurs in a database. CDC is frequently leveraged to replicate between databases in near real time or create an event stream for downstream processing.

CDC is handled differently depending on the database technology. Relational databases often generate an event log stored directly on the database server that can be processed to create a stream. (See "Database Logs" on page 161.) Many cloud NoSQL databases can send a log or event stream to a target storage location.

Logs

A *log* captures information about events that occur in systems. For example, a log may capture traffic and usage patterns on a web server. Your desktop computer's operating system (Windows, macOS, Linux) logs events as the system boots and when applications start or crash, for example.

Logs are a rich data source, potentially valuable for downstream data analysis, ML, and automation. Here are a few familiar sources of logs:

- Operating systems
- Applications
- Servers
- Containers

- Networks
- IoT devices

All logs track events and event metadata. At a minimum, a log should capture who, what, and when:

Who
: The human, system, or service account associated with the event (e.g., a web browser user agent or a user ID)

What happened
: The event and related metadata

When
: The timestamp of the event

Log encoding

Logs are encoded in a few ways:

Binary-encoded logs
: These encode data in a custom compact format for space efficiency and fast I/O. Database logs, discussed in "Database Logs" on page 161, are a standard example.

Semistructured logs
: These are encoded as text in an object serialization format (JSON, more often than not). Semistructured logs are machine-readable and portable. However, they are much less efficient than binary logs. And though they are nominally machine-readable, extracting value from them often requires significant custom code.

Plain-text (unstructured) logs
: These essentially store the console output from software. As such, no generalpurpose standards exist. These logs can provide helpful information for data scientists and ML engineers, though extracting useful information from the raw text data might be complicated.

Log resolution

Logs are created at various resolutions and log levels. The log *resolution* refers to the amount of event data captured in a log. For example, database logs capture enough information from database events to allow reconstructing the database state at any point in time.

On the other hand, capturing all data changes in logs for a big data system often isn't practical. Instead, these logs may note only that a particular type of commit event has occurred. The *log level* refers to the conditions required to record a log entry,

specifically concerning errors and debugging. Software is often configurable to log every event or to log only errors, for example.

Log latency: Batch or real time

Batch logs are often written continuously to a file. Individual log entries can be written to a messaging system such as Kafka or Pulsar for real-time applications.

Database Logs

Database logs are essential enough that they deserve more detailed coverage. Writeahead logs—typically, binary files stored in a specific database-native format—play a crucial role in database guarantees and recoverability. The database server receives write and update requests to a database table (see Figure 5-3), storing each operation in the log before acknowledging the request. The acknowledgment comes with a log-associated guarantee: even if the server fails, it can recover its state on reboot by completing the unfinished work from the logs.

Database logs are extremely useful in data engineering, especially for CDC to generate event streams from database changes.

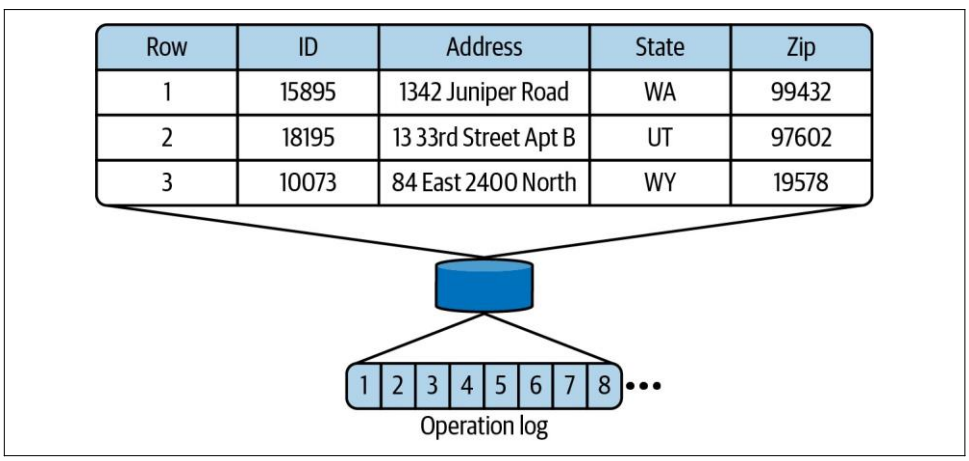


Figure 5-3. Database logs record operations on a table

CRUD

CRUD, which stands for *create*, *read*, *update*, and *delete*, is a transactional pattern commonly used in programming and represents the four basic operations of persistent storage. CRUD is the most common pattern for storing application state in a database. A basic tenet of CRUD is that data must be created before being used. After the data has been created, the data can be read and updated. Finally, the data may need to be

destroyed. CRUD guarantees these four operations will occur on data, regardless of its storage.

CRUD is a widely used pattern in software applications, and you'll commonly find CRUD used in APIs and databases. For example, a web application will make heavy use of CRUD for RESTful HTTP requests and storing and retrieving data from a database.

As with any database, we can use snapshot-based extraction to get data from a database where our application applies CRUD operations. On the other hand, event extraction with CDC gives us a complete history of operations and potentially allows for near real-time analytics.

Insert-Only

The *insert-only pattern* retains history directly in a table containing data. Rather than updating records, new records get inserted with a timestamp indicating when they were created (Table 5-1). For example, suppose you have a table of customer addresses. Following a CRUD pattern, you would simply update the record if the customer changed their address. With the insert-only pattern, a new address record is inserted with the same customer ID. To read the current customer address by customer ID, you would look up the latest record under that ID.

Table 5-1. An insert-only pattern produces multiple versions of a record

Record ID	Value	Timestamp
1	40	2021-09-19T00:10:23+00:00
1	51	2021-09-30T00:12:00+00:00

In a sense, the insert-only pattern maintains a database log directly in the table itself, making it especially useful if the application needs access to history. For example, the insert-only pattern would work well for a banking application designed to present customer address history.

A separate analytics insert-only pattern is often used with regular CRUD application tables. In the insert-only ETL pattern, data pipelines insert a new record in the target analytics table anytime an update occurs in the CRUD table.

Insert-only has a couple of disadvantages. First, tables can grow quite large, especially if data frequently changes, since each change is inserted into the table. Sometimes records are purged based on a record sunset date or a maximum number of record versions to keep table size reasonable. The second disadvantage is that record lookups incur extra overhead because looking up the current state involves running `MAX`

`(created_timestamp)`. If hundreds or thousands of records are under a single ID, this lookup operation is expensive to run.

Messages and Streams

Related to event-driven architecture, two terms that you'll often see used interchangeably are *message queue* and *streaming platform*, but a subtle but essential difference exists between the two. Defining and contrasting these terms is worthwhile since they encompass many big ideas related to source systems and practices and technologies spanning the entire data engineering lifecycle.

A *message* is raw data communicated across two or more systems (Figure 5-4). For example, we have System 1 and System 2, where System 1 sends a message to System 2. These systems could be different microservices, a server sending a message to a serverless function, etc. A message is typically sent through a *message queue* from a publisher to a consumer, and once the message is delivered, it is removed from the queue.

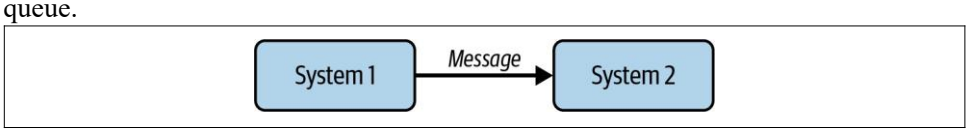


Figure 5-4. A message passed between two systems

Messages are discrete and singular signals in an event-driven system. For example, an IoT device might send a message with the latest temperature reading to a message queue. This message is then ingested by a service that determines whether the furnace should be turned on or off. This service sends a message to a furnace controller that takes the appropriate action. Once the message is received, and the action is taken, the message is removed from the message queue.

By contrast, a *stream* is an append-only log of event records. (Streams are ingested and stored in *event-streaming platforms*, which we discuss at greater length in "Messages and Streams" on page 163.) As events occur, they are accumulated in an ordered sequence (Figure 5-5); a timestamp or an ID might order events. (Note that events aren't always delivered in exact order because of the subtleties of distributed systems.)

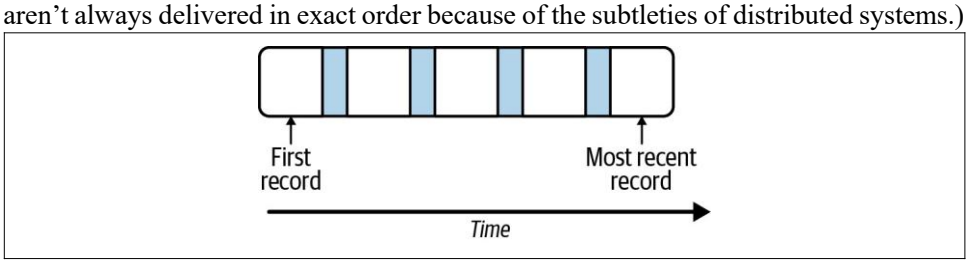


Figure 5-5. A stream, which is an ordered append-only log of records

You'll use streams when you care about what happened over many events. Because of the append-only nature of streams, records in a stream are persisted over a long retention window—often weeks or months—allowing for complex operations on records such as aggregations on multiple records or the ability to rewind to a point in time within the stream.

It's worth noting that systems that process streams can process messages, and streaming platforms are frequently used for message passing. We often accumulate messages in streams when we want to perform message analytics. In our IoT example, the temperature readings that trigger the furnace to turn on or off might also be later analyzed to determine temperature trends and statistics.

Types of Time

While time is an essential consideration for all data ingestion, it becomes that much more critical and subtle in the context of streaming, where we view data as continuous and expect to consume it shortly after it is produced. Let's look at the key types of time you'll run into when ingesting data: the time that the event is generated, when it's ingested and processed, and how long processing took (Figure 5-6).

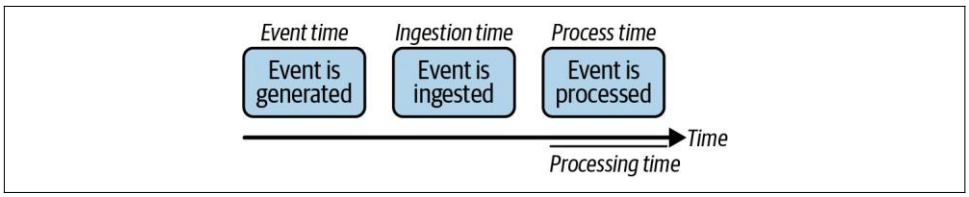


Figure 5-6. Event, ingestion, process, and processing time

Event time indicates when an event is generated in a source system, including the timestamp of the original event itself. An undetermined time lag will occur upon event creation, before the event is ingested and processed downstream. Always include timestamps for each phase through which an event travels. Log events as they occur and at each stage of time—when they're created, ingested, and processed. Use these timestamp logs to accurately track the movement of your data through your data pipelines.

After data is created, it is ingested somewhere. *Ingestion time* indicates when an event is ingested from source systems into a message queue, cache, memory, object storage, a database, or any place else that data is stored (see Chapter 6). After ingestion, data may be processed immediately; or within minutes, hours, or days; or simply persist in storage indefinitely.

Process time occurs after ingestion time, when the data is processed (typically, a transformation). *Processing time* is how long the data took to process, measured in seconds, minutes, hours, etc.

You'll want to record these various times, preferably in an automated way. Set up monitoring along your data workflows to capture when events occur, when they're ingested and processed, and how long it took to process events.

Source System Practical Details

This section discusses the practical details of interacting with modern source systems. We'll dig into the details of commonly encountered databases, APIs, and other aspects. This information will have a shorter shelf life than the main ideas discussed previously; popular API frameworks, databases, and other details will continue to change rapidly.

Nevertheless, these details are critical knowledge for working data engineers. We suggest that you study this information as baseline knowledge but read extensively to stay abreast of ongoing developments.

Databases

In this section, we'll look at common source system database technologies that you'll encounter as a data engineer and high-level considerations for working with these systems. There are as many types of databases as there are use cases for data.

Major considerations for understanding database technologies

Here, we introduce major ideas that occur across a variety of database technologies, including those that back software applications and those that support analytics use cases:

Database management system
: A database system used to store and serve data. Abbreviated as DBMS, it consists of a storage engine, query optimizer, disaster recovery, and other key components for managing the database system.

Lookups
: How does the database find and retrieve data? Indexes can help speed up lookups, but not all databases have indexes. Know whether your database uses indexes; if so, what are the best patterns for designing and maintaining them? Understand how to leverage for efficient extraction. It also helps to have a basic knowledge of the major types of indexes, including B-tree and log-structured merge-trees (LSM).

Query optimizer
: Does the database utilize an optimizer? What are its characteristics?

Scaling and distribution
: Does the database scale with demand? What scaling strategy does it deploy? Does it scale horizontally (more database nodes) or vertically (more resources on a single machine)?

Modeling patterns
: What modeling patterns work best with the database (e.g., data normalization or wide tables)? (See Chapter 8 for our discussion of data modeling.)

CRUD
: How is data queried, created, updated, and deleted in the database? Every type of database handles CRUD operations differently.

Consistency
: Is the database fully consistent, or does it support a relaxed consistency model (e.g., eventual consistency)? Does the database support optional consistency modes for reads and writes (e.g., strongly consistent reads)?

We divide databases into relational and nonrelational categories. In truth, the nonrelational category is far more diverse, but relational databases still occupy significant space in application backends.

Relational databases

A *relational database management system* (RDBMS) is one of the most common application backends. Relational databases were developed at IBM in the 1970s and popularized by Oracle in the 1980s. The growth of the internet saw the rise of the LAMP stack (Linux, Apache web server, MySQL, PHP) and an explosion of vendor and open source RDBMS options. Even with the rise of NoSQL databases (described in the following section), relational databases have remained extremely popular.

Data is stored in a table of *relations* (rows), and each relation contains multiple *fields* (columns); see Figure 5-7. Note that we use the terms *column* and *field* interchangeably throughout this book. Each relation in the table has the same *schema* (a sequence of columns with assigned static types such as string, integer, or float). Rows are typically stored as a contiguous sequence of bytes on disk.

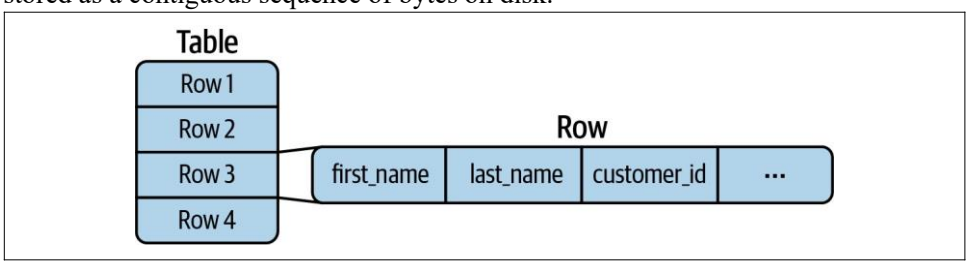


Figure 5-7. RDBMS stores and retrieves data at a row level

Tables are typically indexed by a *primary key*, a unique field for each row in the table. The indexing strategy for the primary key is closely connected with the layout of the table on disk.

Tables can also have various *foreign keys*—fields with values connected with the values of primary keys in other tables, facilitating joins, and allowing for complex schemas that spread data across multiple tables. In particular, it is possible to design a *normalized schema*. Normalization is a strategy for ensuring that data in records is not duplicated in multiple places, thus avoiding the need to update states in multiple locations at once and preventing inconsistencies (see Chapter 8).

RDBMS systems are typically ACID compliant. Combining a normalized schema, ACID compliance, and support for high transaction rates makes relational database systems ideal for storing rapidly changing application states. The challenge for data engineers is to determine how to capture state information over time.

A full discussion of the theory, history, and technology of RDBMS is beyond the scope of this book. We encourage you to study RDBMS systems, relational algebra, and strategies for normalization because they're widespread, and you'll encounter them frequently. See "Additional Resources" on page 188 for suggested books.

Nonrelational databases: NoSQL

While relational databases are terrific for many use cases, they're not a one-size-fitsall solution. We often see that people start with a relational database under the impression it's a universal appliance and shoehorn in a ton of use cases and workloads. As data and query requirements morph, the relational database collapses under its weight. At that point, you'll want to use a database that's appropriate for the specific workload under pressure. Enter nonrelational or NoSQL databases. *NoSQL*, which stands for *not only SQL*, refers to a whole class of databases that abandon the relational paradigm.

On the one hand, dropping relational constraints can improve performance, scalability, and schema flexibility. But as always in architecture, trade-offs exist. NoSQL databases also typically abandon various RDBMS characteristics, such as strong consistency, joins, or a fixed schema.

A big theme of this book is that data innovation is constant. Let's take a quick look at the history of NoSQL, as it's helpful to gain a perspective on why and how data innovations impact your work as a data engineer. In the early 2000s, tech companies such as Google and Amazon began to outgrow their relational databases and pioneered new distributed, nonrelational databases to scale their web platforms.

While the term *NoSQL* first appeared in 1998, the modern version was coined by Eric Evans in the 2000s.[62] He tells the story in a 2009 blog post:

> I've spent the last couple of days at nosqleast and one of the hot topics here is the name "nosql." Understandably, there are a lot of people who worry that the name is Bad, that it sends an inappropriate or inaccurate message. While I make no claims to the idea, I do have to accept some blame for what it is now being called. How's that? Johan Oskarsson was organizing the first meetup and asked the question "What's a good name?" on IRC; it was one of three or four suggestions that I spouted off in the span of like 45 seconds, without thinking.
>
> My regret, however, isn't about what the name says; it's about what it doesn't. When Johan originally had the idea for the first meetup, he seemed to be thinking Big Data and linearly scalable distributed systems, but the name is so vague that it opened the door to talk submissions for literally anything that stored data, and wasn't an RDBMS.

[62] Keith D. Foote, "A Brief History of Non-Relational Databases," Dataversity, June 19, 2018, *https://oreil.ly/5Ukg2*.

NoSQL remains vague in 2022, but it's been widely adopted to describe a universe of "new school" databases, alternatives to relational databases.

There are numerous flavors of NoSQL database designed for almost any imaginable use case. Because there are far too many NoSQL databases to cover exhaustively in this section, we consider the following database types: key-value, document, wide-column, graph, search, and time series. These databases are all wildly popular and enjoy widespread adoption. A data engineer should understand these types of databases, including usage considerations, the structure of the data they store, and how to leverage each in the data engineering lifecycle.

Key-value stores. A *key-value database* is a nonrelational database that retrieves records using a key that uniquely identifies each record. This is similar to hash map or dictionary data structures presented in many programming languages but potentially more scalable. Key-value stores encompass several NoSQL database types —for example, document stores and wide column databases (discussed next). Different types of key-value databases offer a variety of performance characteristics to serve various application needs. For example, in-memory key-value databases are popular for caching session data for web and mobile applications, where ultra-fast lookup and high concurrency are required. Storage in these systems is typically temporary; if the database shuts down, the data disappears. Such caches can reduce pressure on the main application database and serve speedy responses.

Of course, key-value stores can also serve applications requiring high-durability persistence. An ecommerce application may need to save and update massive amounts of event state changes for a user and their orders. A user logs into the ecommerce application, clicks around various screens, adds items to a shopping cart, and then checks out. Each event must be durably stored for retrieval. Key-value stores often persist data to disk and across multiple nodes to support such use cases.

Document stores. As mentioned previously, a *document store* is a specialized key-value store. In this context, a *document* is a nested object; we can usually think of each document as a JSON object for practical purposes. Documents are stored in collections and retrieved by key. A *collection* is roughly equivalent to a table in a relational database (see Table 5-2).

Table 5-2. Comparison of RDBMS and document terminology

RDBMS	Document database
Table	Collection
Row	Document, items, entity

One key difference between relational databases and document stores is that the latter does not support joins. This means that data cannot be easily *normalized*, i.e., split across multiple tables. (Applications can still join manually. Code can look up a document, extract a property, and then retrieve another document.) Ideally, all related data can be stored in the same document.

In many cases, the same data must be stored in multiple documents spread across numerous collections; software engineers must be careful to update a property everywhere it is stored. (Many document stores support a notion of transactions to facilitate this.)

Document databases generally embrace all the flexibility of JSON and don't enforce schema or types; this is a blessing and a curse. On the one hand, this allows the schema to be highly flexible and expressive. The schema can also evolve as an application grows. On the flip side, we've seen document databases become absolute nightmares to manage and query. If developers are not careful in managing schema evolution, data may become inconsistent and bloated over time. Schema evolution can also break downstream ingestion and cause headaches for data engineers if it's not communicated in a timely fashion (before deployment).

The following is an example of data that is stored in a collection called `users`. The collection key is the `id`. We also have a `name` (along with `first` and `last` as child elements) and an array of the user's favorite bands within each document:

```
{
  "users":[
      {
      "id":1234,
      "name":{
      "first":"Joe",
      "last":"Reis"
      },
      "favorite_bands":[
      "AC/DC",
      "Slayer",
      "WuTang Clan",
      "Action Bronson"
      ]
      },
      {
      "id":1235,
      "name":{
      "first":"Matt",
      "last":"Housley"
      },
      "favorite_bands":[
"Dave Matthews Band",
```

```
            "Creed",
            "Nickelback"
            ]
            }
        ]
    }
```

To query the data in this example, you can retrieve records by key. Note that most document databases also support the creation of indexes and lookup tables to allow retrieval of documents by specific properties. This is often invaluable in application development when you need to search for documents in various ways. For example, you could set an index on `name`.

Another critical technical detail for data engineers is that document stores are generally not ACID compliant, unlike relational databases. Technical expertise in a particular document store is essential to understanding performance, tuning, configuration, related effects on writes, consistency, durability, etc. For example, many document stores are *eventually consistent*. Allowing data distribution across a cluster is a boon for scaling and performance but can lead to catastrophes when engineers and developers don't understand the implications.[63]

To run analytics on document stores, engineers generally must run a full scan to extract all data from a collection or employ a CDC strategy to send events to a target stream. The full scan approach can have both performance and cost implications. The scan often slows the database as it runs, and many serverless cloud offerings charge a significant fee for each full scan. In document databases, it's often helpful to create an index to help speed up queries. We discuss indexes and query patterns in Chapter 8.

Wide-column. A *wide-column database* is optimized for storing massive amounts of data with high transaction rates and extremely low latency. These databases can scale to extremely high write rates and vast amounts of data. Specifically, widecolumn databases can support petabytes of data, millions of requests per second, and sub-10ms latency. These characteristics have made wide-column databases popular in ecommerce, fintech, ad tech, IoT, and real-time personalization applications. Data engineers must be aware of the operational characteristics of the wide-column databases they work with to set up a suitable configuration, design the schema, and choose an appropriate row key to optimize performance and avoid common operational issues.

These databases support rapid scans of massive amounts of data, but they do not support complex queries. They have only a single index (the row key) for lookups.

[63] Nemil Dalal's excellent series on the history of MongoDB recounts some harrowing tales of database abuse and its consequences for fledgling startups.

Data engineers must generally extract data and send it to a secondary analytics system to run complex queries to deal with these limitations. This can be accomplished by running large scans for the extraction or employing CDC to capture an event stream.

Graph databases. *Graph databases* explicitly store data with a mathematical graph structure (as a set of nodes and edges).[64] Neo4j has proven extremely popular, while Amazon, Oracle, and other vendors offer their graph database products. Roughly speaking, graph databases are a good fit when you want to analyze the connectivity between elements.

For example, you could use a document database to store one document for each user describing their properties. You could add an array element for *connections* that contains directly connected users' IDs in a social media context. It's pretty easy to determine the number of direct connections a user has, but suppose you want to know how many users can be reached by traversing two direct connections. You could answer this question by writing complex code, but each query would run slowly and consume significant resources. The document store is simply not optimized for this use case.

Graph databases are designed for precisely this type of query. Their data structures allow for queries based on the connectivity between elements; graph databases are indicated when we care about understanding complex traversals between elements. In the parlance of graphs, we store *nodes* (users in the preceding example) and *edges* (connections between users). Graph databases support rich data models for both nodes and edges. Depending on the underlying graph database engine, graph databases utilize specialized query languages such as SPARQL, Resource Description Framework (RDF), Graph Query Language (GQL), and Cypher.

As an example of a graph, consider a network of four users. User 1 follows User 2, who follows User 3 and User 4; User 3 also follows User 4 (Figure 5-8).

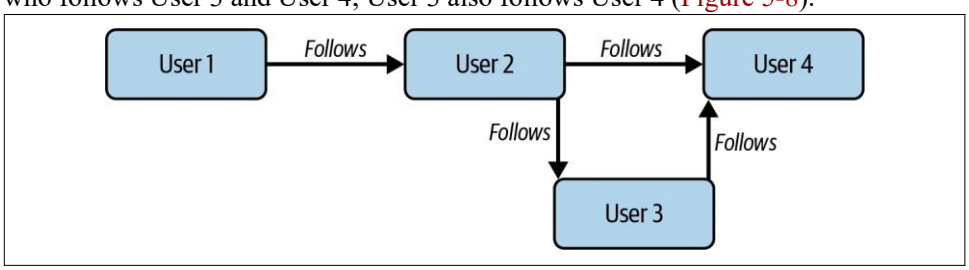


Figure 5-8. A social network graph

[64] Martin Kleppmann, *Designing Data-Intensive Applications* (Sebastopol, CA: O'Reilly, 2017), 49, *https://oreil.ly/v1NhG*.

We anticipate that graph database applications will grow dramatically outside of tech companies; market analyses also predict rapid growth.[65] Of course, graph databases are beneficial from an operational perspective and support the kinds of complex social relationships critical to modern applications. Graph structures are also fascinating from the perspective of data science and ML, potentially revealing deep insights into human interactions and behavior.

This introduces unique challenges for data engineers who may be more accustomed to dealing with structured relations, documents, or unstructured data. Engineers must choose whether to do the following:

- Map source system graph data into one of their existing preferred paradigms
- Analyze graph data within the source system itself
- Adopt graph-specific analytics tools

Graph data can be reencoded into rows in a relational database, which may be a suitable solution depending on the analytics use case. Transactional graph databases are also designed for analytics, although large queries may overload production systems. Contemporary cloud-based graph databases support read-heavy graph analytics on massive quantities of data.

Search. A *search database* is a nonrelational database used to search your data's complex and straightforward semantic and structural characteristics. Two prominent use cases exist for a search database: text search and log analysis. Let's cover each of these separately.

Text search involves searching a body of text for keywords or phrases, matching on exact, fuzzy, or semantically similar matches. *Log analysis* is typically used for anomaly detection, real-time monitoring, security analytics, and operational analytics. Queries can be optimized and sped up with the use of indexes.

Depending on the type of company you work at, you may use search databases either regularly or not at all. Regardless, it's good to be aware they exist in case you come across them in the wild. Search databases are popular for fast search and retrieval and can be found in various applications; an ecommerce site may power its product search using a search database. As a data engineer, you might be expected to bring data from a search database (such as Elasticsearch, Apache Solr or Lucene, or Algolia) into downstream KPI reports or something similar.

[65] Aashish Mehra, "Graph Database Market Worth $5.1 Billion by 2026: Exclusive Report by MarketsandMarkets," Cision PR Newswire, July 30, 2021, *https://oreil.ly/mGVkY*.

Time series. A *time series* is a series of values organized by time. For example, stock prices might move as trades are executed throughout the day, or a weather sensor will take atmospheric temperatures every minute. Any events that are recorded over time—either regularly or sporadically—are time-series data. A *time-series database* is optimized for retrieving and statistical processing of time-series data.

While time-series data such as orders, shipments, logs, and so forth have been stored in relational databases for ages, these data sizes and volumes were often tiny. As data grew faster and bigger, new special-purpose databases were needed. Time-series databases address the needs of growing, high-velocity data volumes from IoT, event and application logs, ad tech, and fintech, among many other use cases. Often these workloads are write-heavy. As a result, time-series databases often utilize memory buffering to support fast writes and reads.

We should distinguish between measurement and event-based data, common in time-series databases. *Measurement data* is generated regularly, such as temperature or air-quality sensors. *Event-based data* is irregular and created every time an event occurs—for instance, when a motion sensor detects movement.

The schema for a time series typically contains a timestamp and a small set of fields. Because the data is time-dependent, the data is ordered by the timestamp. This makes time-series databases suitable for operational analytics but not great for BI use cases. Joins are not common, though some quasi time-series databases such as Apache Druid support joins. Many time-series databases are available, both as open source and paid options.

APIs

APIs are now a standard and pervasive way of exchanging data in the cloud, for SaaS platforms, and between internal company systems. Many types of API interfaces exist across the web, but we are principally interested in those built around HTTP, the most popular type on the web and in the cloud.

REST

We'll first talk about REST, currently the dominant API paradigm. As noted in Chapter 4, *REST* stands for *representational state transfer*. This set of practices and philosophies for building HTTP web APIs was laid out by Roy Fielding in 2000 in a PhD dissertation. REST is built around HTTP verbs, such as GET and PUT; in practice, modern REST uses only a handful of the verb mappings outlined in the original dissertation.

One of the principal ideas of REST is that interactions are stateless. Unlike in a Linux terminal session, there is no notion of a session with associated state variables such as a working directory; each REST call is independent. REST calls can change the

system's state, but these changes are global, applying to the full system rather than a current session.

Critics point out that REST is in no way a full specification.[66] REST stipulates basic properties of interactions, but developers utilizing an API must gain a significant amount of domain knowledge to build applications or pull data effectively.

We see great variation in levels of API abstraction. In some cases, APIs are merely a thin wrapper over internals that provides the minimum functionality required to protect the system from user requests. In other examples, a REST data API is a masterpiece of engineering that prepares data for analytics applications and supports advanced reporting.

A couple of developments have simplified setting up data-ingestion pipelines from REST APIs. First, data providers frequently supply client libraries in various languages, especially in Python. Client libraries remove much of the boilerplate labor of building API interaction code. Client libraries handle critical details such as authentication and map fundamental methods into accessible classes.

Second, various services and open source libraries have emerged to interact with APIs and manage data synchronization. Many SaaS and open source vendors provide off-the-shelf connectors for common APIs. Platforms also simplify the process of building custom connectors as required.

There are numerous data APIs without client libraries or out-of-the-box connector support. As we emphasize throughout the book, engineers would do well to reduce undifferentiated heavy lifting by using off-the-shelf tools. However, low-level *plumbing* tasks still consume many resources. At virtually any large company, data engineers will need to deal with the problem of writing and maintaining custom code to pull data from APIs, which requires understanding the structure of the data as provided, developing appropriate data-extraction code, and determining a suitable data synchronization strategy.

GraphQL

GraphQL was created at Facebook as a query language for application data and an alternative to generic REST APIs. Whereas REST APIs generally restrict your queries to a specific data model, GraphQL opens up the possibility of retrieving multiple data models in a single request. This allows for more flexible and expressive queries than with REST. GraphQL is built around JSON and returns data in a shape resembling the JSON query.

[66] For one example, see Michael S. Mikowski, "RESTful APIs: The Big Lie," August 10, 2015, *https://oreil.ly/rqja3*.

There's something of a holy war between REST and GraphQL, with some engineering teams partisans of one or the other and some using both. In reality, engineers will encounter both as they interact with source systems.

Webhooks

Webhooks are a simple event-based data-transmission pattern. The data source can be an application backend, a web page, or a mobile app. When specified events happen in the source system, this triggers a call to an HTTP endpoint hosted by the data consumer. Notice that the connection goes from the source system to the data sink, the opposite of typical APIs. For this reason, webhooks are often called *reverse APIs*.

The endpoint can do various things with the POST event data, potentially triggering a downstream process or storing the data for future use. For analytics purposes, we're interested in collecting these events. Engineers commonly use message queues to ingest data at high velocity and volume. We will talk about message queues and event streams later in this chapter.

RPC and gRPC

A *remote procedure call* (RPC) is commonly used in distributed computing. It allows you to run a procedure on a remote system.

gRPC is a remote procedure call library developed internally at Google in 2015 and later released as an open standard. Its use at Google alone would be enough to merit inclusion in our discussion. Many Google services, such as Google Ads and GCP, offer gRPC APIs. gRPC is built around the Protocol Buffers open data serialization standard, also developed by Google.

gRPC emphasizes the efficient bidirectional exchange of data over HTTP/2. *Efficiency* refers to aspects such as CPU utilization, power consumption, battery life, and bandwidth. Like GraphQL, gRPC imposes much more specific technical standards than REST, thus allowing the use of common client libraries and allowing engineers to develop a skill set that will apply to any gRPC interaction code.

Data Sharing

The core concept of cloud data sharing is that a multitenant system supports security policies for sharing data among tenants. Concretely, any public cloud object storage system with a fine-grained permission system can be a platform for data sharing. Popular cloud data-warehouse platforms also support data-sharing capabilities. Of course, data can also be shared through download or exchange over email, but a multitenant system makes the process much easier.

Many modern sharing platforms (especially cloud data warehouses) support row, column, and sensitive data filtering. Data sharing also streamlines the notion of the *data marketplace*, available on several popular clouds and data platforms. Data marketplaces provide a centralized location for data commerce, where data providers can advertise their offerings and sell them without worrying about the details of managing network access to data systems.

Data sharing can also streamline data pipelines within an organization. Data sharing allows units of an organization to manage their data and selectively share it with other units while still allowing individual units to manage their compute and query costs separately, facilitating data decentralization. This facilitates decentralized data management patterns such as data mesh.[67]

Data sharing and data mesh align closely with our philosophy of common architecture components. Choose common components (see Chapter 3) that allow the simple and efficient interchange of data and expertise rather than embracing the most exciting and sophisticated technology.

Third-Party Data Sources

The consumerization of technology means every company is essentially now a technology company. The consequence is that these companies—and increasingly government agencies—want to make their data available to their customers and users, either as part of their service or as a separate subscription. For example, the US Bureau of Labor Statistics publishes various statistics about the US labor market. The National Aeronautics and Space Administration (NASA) publishes various data from its research initiatives. Facebook shares data with businesses that advertise on its platform.

Why would companies want to make their data available? Data is sticky, and a flywheel is created by allowing users to integrate and extend their application into a user's application. Greater user adoption and usage means more data, which means users can integrate more data into their applications and data systems. The side effect is there are now almost infinite sources of third-party data.

Direct third-party data access is commonly done via APIs, through data sharing on a cloud platform, or through data download. APIs often provide deep integration capabilities, allowing customers to pull and push data. For example, many CRMs offer APIs that their users can integrate into their systems and applications. We see a common workflow to get data from a CRM, blend the CRM data through the customer

[67] Martin Fowler, "How to Move Beyond a Monolithic Data Lake to a Distributed Data Mesh," MartinFowler.com, May 20, 2019, *https://oreil.ly/TEdJF*.

scoring model, and then use reverse ETL to send that data back into CRM for salespeople to contact better-qualified leads.

Message Queues and Event-Streaming Platforms

Event-driven architectures are pervasive in software applications and are poised to grow their popularity even further. First, message queues and event-streaming platforms—critical layers in event-driven architectures—are easier to set up and manage in a cloud environment. Second, the rise of data apps—applications that directly integrate real-time analytics—are growing from strength to strength. Event-driven architectures are ideal in this setting because events can both trigger work in the application and feed near real-time analytics.

Please note that streaming data (in this case, messages and streams) cuts across many data engineering lifecycle stages. Unlike an RDBMS, which is often directly attached to an application, the lines of streaming data are sometimes less clear-cut. These systems are used as source systems, but they will often cut across the data engineering lifecycle because of their transient nature. For example, you can use an event-streaming platform for message passing in an event-driven application, a source system. The same event-streaming platform can be used in the ingestion and transformation stage to process data for real-time analytics.

As source systems, message queues and event-streaming platforms are used in numerous ways, from routing messages between microservices ingesting millions of events per second of event data from web, mobile, and IoT applications. Let's look at message queues and event-streaming platforms a bit more closely.

Message queues

A *message queue* is a mechanism to asynchronously send data (usually as small individual messages, in the kilobytes) between discrete systems using a publish and subscribe model. Data is published to a message queue and is delivered to one or more subscribers (Figure 5-9). The subscriber acknowledges receipt of the message, removing it from the queue.

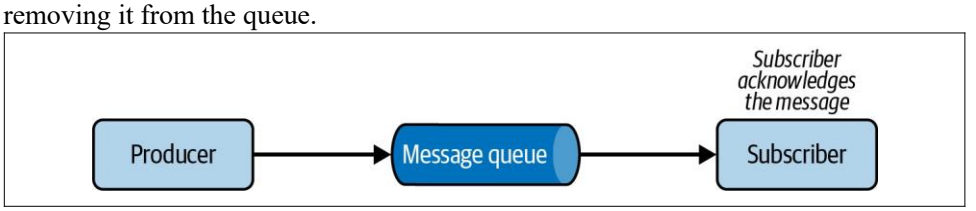


Figure 5-9. A simple message queue

Message queues allow applications and systems to be decoupled from each other and are widely used in microservices architectures. The message queue buffers messages

to handle transient load spikes and makes messages durable through a distributed architecture with replication.

Message queues are a critical ingredient for decoupled microservices and eventdriven architectures. Some things to keep in mind with message queues are frequency of delivery, message ordering, and scalability.

Message ordering and delivery. The order in which messages are created, sent, and received can significantly impact downstream subscribers. In general, order in distributed message queues is a tricky problem. Message queues often apply a fuzzy notion of order and first in, first out (FIFO). Strict FIFO means that if message A is ingested before message B, message A will always be delivered before message B. In practice, messages might be published and received out of order, especially in highly

distributed message systems.

For example, Amazon SQS standard queues make the best effort to preserve message order. SQS also offers FIFO queues, which offer much stronger guarantees at the cost of extra overhead.

In general, don't assume that your messages will be delivered in order unless your message queue technology guarantees it. You typically need to design for out-oforder message delivery.

Delivery frequency. Messages can be sent exactly once or at least once. If a message is sent *exactly once*, then after the subscriber acknowledges the message, the message disappears and won't be delivered again.[68] Messages sent *at least once* can be consumed by multiple subscribers or by the same subscriber more than once. This is great when duplications or redundancy don't matter.

Ideally, systems should be *idempotent*. In an idempotent system, the outcome of processing a message once is identical to the outcome of processing it multiple times. This helps to account for a variety of subtle scenarios. For example, even if our system can guarantee exactly-once delivery, a consumer might fully process a message but fail right before acknowledging processing. The message will effectively be processed twice, but an idempotent system handles this scenario gracefully.

Scalability. The most popular message queues utilized in event-driven applications are horizontally scalable, running across multiple servers. This allows these queues to scale up and down dynamically, buffer messages when systems fall behind, and

[68] Whether *exactly once* is possible is a semantical debate. Technically, exactly once delivery is impossible to guarantee, as illustrated by the Two Generals Problem.

durably store messages for resilience against failure. However, this can create a variety of complications, as mentioned previously (multiple deliveries and fuzzy ordering).

Event-streaming platforms

In some ways, an *event-streaming platform* is a continuation of a message queue in that messages are passed from producers to consumers. As discussed previously in this chapter, the big difference between messages and streams is that a message queue is primarily used to route messages with certain delivery guarantees. In contrast, an event-streaming platform is used to ingest and process data in an ordered log of records. In an event-streaming platform, data is retained for a while, and it is possible to replay messages from a past point in time.

Let's describe an event related to an event-streaming platform. As mentioned in Chapter 3, an event is "something that happened, typically a change in the *state* of something." An event has the following features: a key, a value, and a timestamp. Multiple key-value timestamps might be contained in a single event. For example, an event for an ecommerce order might look like this:

```
{
  "Key":"Order # 12345",
  "Value":"SKU 123, purchase price of $100",
  "Timestamp":"2023-01-02 06:01:00"
}
```

Let's look at some of the critical characteristics of an event-streaming platform that you should be aware of as a data engineer.

Topics. In an event-streaming platform, a producer streams events to a topic, a collection of related events. A topic might contain fraud alerts, customer orders, or temperature readings from IoT devices, for example. A topic can have zero, one, or multiple producers and customers on most event-streaming platforms.

Using the preceding event example, a topic might be `web orders`. Also, let's send this topic to a couple of consumers, such as `fulfillment` and `marketing`. This is an excellent example of blurred lines between analytics and an event-driven system. The `fulfillment` subscriber will use events to trigger a fulfillment process, while `marketing` runs real-time analytics or trains and runs ML models to tune marketing campaigns (Figure 5-10).

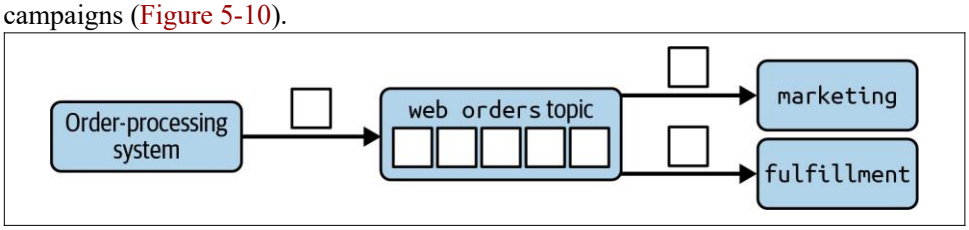


Figure 5-10. An order-processing system generates events (small squares) and publishes them to the `web orders` topic. Two subscribers—`marketing` and `fulfillment`—pull events from the topic.

Stream partitions. *Stream partitions* are subdivisions of a stream into multiple streams. A good analogy is a multilane freeway. Having multiple lanes allows for parallelism and higher throughput. Messages are distributed across partitions by *partition key*. Messages with the same partition key will always end up in the same partition.

In Figure 5-11, for example, each message has a numeric ID— shown inside the circle representing the message—that we use as a partition key. To determine the partition, we divide by 3 and take the remainder. Going from bottom to top, the partitions have remainder 0, 1, and 2, respectively.

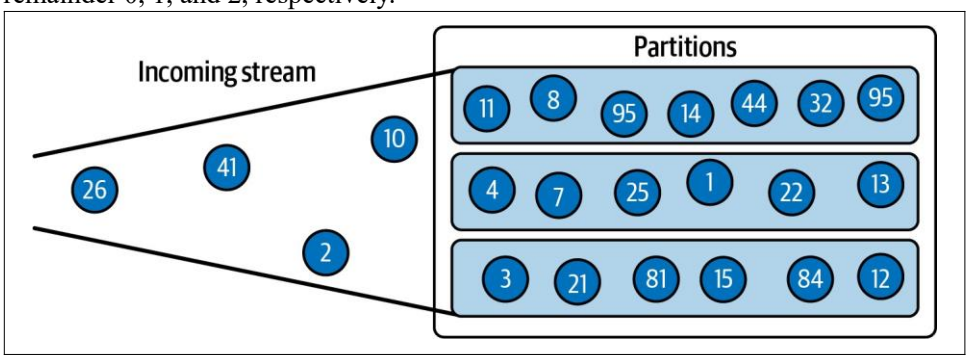


Figure 5-11. An incoming message stream broken into three partitions

Set a partition key so that messages that should be processed together have the same partition key. For example, it is common in IoT settings to want to send all messages from a particular device to the same processing server. We can achieve this by using a device ID as the partition key, and then setting up one server to consume from each partition.

A key concern with stream partitioning is ensuring that your partition key does not generate *hotspotting*—a disproportionate number of messages delivered to one partition. For example, if each IoT device were known to be located in a particular US state, we might use the state as the partition key. Given a device distribution proportional to state population, the partitions containing California, Texas, Florida, and New York might be overwhelmed, with other partitions relatively underutilized. Ensure that your partition key will distribute messages evenly across partitions.

Fault tolerance and resilience. Event-streaming platforms are typically distributed systems, with streams stored on various nodes. If a node goes down, another node replaces it, and the stream is still accessible. This means records aren't lost; you may choose to delete records, but that's another story. This fault tolerance and resilience make streaming platforms a good choice when you need a system that can reliably produce, store, and ingest event data.

Whom You'll Work With

When accessing source systems, it's essential to understand the people with whom you'll work. In our experience, good diplomacy and relationships with the

stakeholders of source systems are an underrated and crucial part of successful data engineering.

Who are these stakeholders? Typically, you'll deal with two categories of stakeholders: systems and data stakeholders (Figure 5-12). A *systems stakeholder* builds and maintains the source systems; these might be software engineers, application developers, and third parties. Data stakeholders own and control access to the data you want, generally handled by IT, a data governance group, or third parties. The systems and data stakeholders are often different people or teams; sometimes, they are the same.

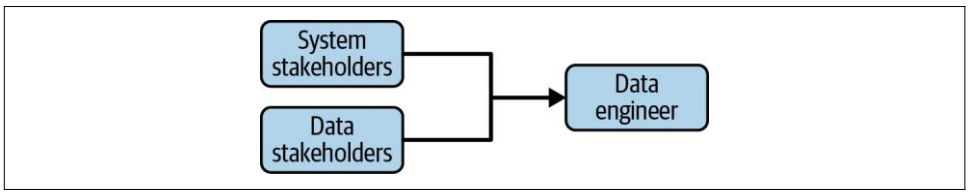


Figure 5-12. The data engineer's upstream stakeholders

You're often at the mercy of the stakeholder's ability to follow correct software engineering, database management, and development practices. Ideally, the stakeholders are doing DevOps and working in an agile manner. We suggest creating a feedback loop between data engineers and stakeholders of the source systems to create awareness of how data is consumed and used. This is among the single most overlooked areas where data engineers can get a lot of value. When something happens to the upstream source data—and something will happen, whether it's a schema or data change, a failed server or database, or other important events—you want to make sure that you're made aware of the impact these issues will have on your data engineering systems.

It might help to have a data contract in place with your upstream source system owners. What is a data contract? James Denmore offers this definition:[69]

> A data contract is a written agreement between the owner of a source system and the team ingesting data from that system for use in a data pipeline. The contract should state what data is being extracted, via what method (full, incremental), how often, as well as who (person, team) are the contacts for both the source system and the ingestion. Data contracts should be stored in a well-known and easy-to-find location such as a GitHub repo or internal documentation site. If possible, format data contracts

[69] James Denmore, *Data Pipelines Pocket Reference* (Sebastopol, CA: O'Reilly), *https://oreil.ly/8QdkJ*. Read the book for more information on how a data contract should be written.

in a standardized form so they can be integrated into the development process or queried programmatically.

In addition, consider establishing an SLA with upstream providers. An SLA provides expectations of what you can expect from the source systems you rely upon. An example of an SLA might be "data from source systems will be reliably available and of high quality." A service-level objective (SLO) measures performance against what you've agreed to in the SLA. For example, given your example SLA, an SLO might be "source systems will have 99% uptime." If a data contract or SLA/SLO seems too formal, at least verbally set expectations for source system guarantees for uptime, data quality, and anything else of importance to you. Upstream owners of source systems need to understand your requirements so they can provide you with the data you need.

Undercurrents and Their Impact on Source Systems

Unlike other parts of the data engineering lifecycle, source systems are generally outside the control of the data engineer. There's an implicit assumption (some might call it *hope*) that the stakeholders and owners of the source systems—and the data they produce—are following best practices concerning data management, DataOps (and DevOps), DODD (mentioned in Chapter 2) data architecture, orchestration, and software engineering. The data engineer should get as much upstream support as possible to ensure that the undercurrents are applied when data is generated in source systems. Doing so will make the rest of the steps in the data engineering lifecycle proceed a lot more smoothly.

How do the undercurrents impact source systems? Let's have a look.

Security

Security is critical, and the last thing you want is to accidentally create a point of vulnerability in a source system. Here are some areas to consider:

- Is the source system architected so data is secure and encrypted, both with data at rest and while data is transmitted?
- Do you have to access the source system over the public internet, or are you using a virtual private network (VPN)?
- Keep passwords, tokens, and credentials to the source system securely locked away. For example, if you're using Secure Shell (SSH) keys, use a key manager to protect your keys; the same rule applies to passwords—use a password manager or a single sign-on (SSO) provider.

- Do you trust the source system? Always be sure to trust but verify that the source system is legitimate. You don't want to be on the receiving end of data from a malicious actor.

Data Management

Data management of source systems is challenging for data engineers. In most cases, you will have only peripheral control—if any control at all—over source systems and the data they produce. To the extent possible, you should understand the way data is managed in source systems since this will directly influence how you ingest, store, and transform the data.

Here are some areas to consider:

Data governance
: Are upstream data and systems governed in a reliable, easy-to-understand fashion? Who manages the data?

Data quality
: How do you ensure data quality and integrity in upstream systems? Work with source system teams to set expectations on data and communication.

Schema
: Expect that upstream schemas will change. Where possible, collaborate with source system teams to be notified of looming schema changes.

Master data management
: Is the creation of upstream records controlled by a master data management practice or system?

Privacy and ethics
: Do you have access to raw data, or will the data be obfuscated? What are the implications of the source data? How long is it retained? Does it shift locations based on retention policies?

Regulatory
: Based upon regulations, are you supposed to access the data?

DataOps

Operational excellence—DevOps, DataOps, MLOps, *X*Ops—should extend up and down the entire stack and support the data engineering and lifecycle. While this is ideal, it's often not fully realized.

Because you're working with stakeholders who control both the source systems and the data they produce, you need to ensure that you can observe and monitor the uptime and usage of the source systems and respond when incidents occur. For example, when the application database you depend on for CDC exceeds its I/O capacity and needs to be rescaled, how will that affect your ability to receive data from this system? Will you be able to access the data, or will it be unavailable until the database is rescaled? How will this affect reports? In another example, if the software engineering team is continuously deploying, a code change may cause unanticipated failures in the application itself. How will the failure impact your ability to access the databases powering the application? Will the data be up-to-date?

Set up a clear communication chain between data engineering and the teams supporting the source systems. Ideally, these stakeholder teams have incorporated DevOps into their workflow and culture. This will go a long way to accomplishing the goals of DataOps (a sibling of DevOps), to address and reduce errors quickly. As we mentioned earlier, data engineers need to weave themselves into the DevOps practices of stakeholders, and vice versa. Successful DataOps works when all people are on board and focus on making systems holistically work.

A few DataOps considerations are as follows:

Automation
: There's the automation impacting the source system, such as code updates and new features. Then there's the DataOps automation that you've set up for your data workflows. Does an issue in the source system's automation impact your data workflow automation? If so, consider decoupling these systems so they can perform automation independently.

Observability
: How will you know when there's an issue with a source system, such as an outage or a data-quality issue? Set up monitoring for source system uptime (or use the monitoring created by the team that owns the source system). Set up checks to ensure that data from the source system conforms with expectations for downstream usage. For example, is the data of good quality? Is the schema conformant? Are customer records consistent? Is data hashed as stipulated by the internal policy?

Incident response
: What's your plan if something bad happens? For example, how will your data pipeline behave if a source system goes offline? What's your plan to backfill the "lost" data once the source system is back online?

Data Architecture

Similar to data management, unless you're involved in the design and maintenance of the source system architecture, you'll have little impact on the upstream source system architecture. You should also understand how the upstream architecture is designed and its strengths and weaknesses. Talk often with the teams responsible for the source systems to understand the factors discussed in this section and ensure that their systems can meet your expectations. Knowing where the architecture performs well and where it doesn't will impact how you design your data pipeline.

Here are some things to consider regarding source system architectures:

Reliability
: All systems suffer from entropy at some point, and outputs will drift from what's expected. Bugs are introduced, and random glitches happen. Does the system produce predictable outputs? How often can we expect the system to fail? What's the mean time to repair to get the system back to sufficient reliability?

Durability
: Everything fails. A server might die, a cloud's zone or region could go offline, or other issues may arise. You need to account for how an inevitable failure or outage will affect your managed data systems. How does the source system handle data loss from hardware failures or network outages? What's the plan for handling outages for an extended period and limiting the blast radius of an outage?

Availability
: What guarantees that the source system is up, running, and available when it's supposed to be?

People
: Who's in charge of the source system's design, and how will you know if breaking changes are made in the architecture? A data engineer needs to work with the teams who maintain the source systems and ensure that these systems are architected reliably. Create an SLA with the source system team to set expectations about potential system failure.

Orchestration

When orchestrating within your data engineering workflow, you'll primarily be concerned with making sure your orchestration can access the source system, which requires the correct network access, authentication, and authorization.

Here are some things to think about concerning orchestration for source systems:

Cadence and frequency
: Is the data available on a fixed schedule, or can you access new data whenever you want?

Common frameworks
: Do the software and data engineers use the same container manager, such as Kubernetes? Would it make sense to integrate application and data workloads into the same Kubernetes cluster? If you're using an orchestration framework like Airflow, does it make sense to integrate it with the upstream application team? There's no correct answer here, but you need to balance the benefits of integration with the risks of tight coupling.

Software Engineering

As the data landscape shifts to tools that simplify and automate access to source systems, you'll likely need to write code. Here are a few considerations when writing code to access a source system:

Networking
: Make sure your code will be able to access the network where the source system resides. Also, always think about secure networking. Are you accessing an HTTPS URL over the public internet, SSH, or a VPN?

Authentication and authorization
: Do you have the proper credentials (tokens, username/passwords) to access the source system? Where will you store these credentials so they don't appear in your code or version control? Do you have the correct IAM roles to perform the coded tasks?

Access patterns
: How are you accessing the data? Are you using an API, and how are you handling REST/GraphQL requests, response data volumes, and pagination? If you're accessing data via a database driver, is the driver compatible with the database you're accessing? For either access pattern, how are things like retries and timeouts handled?

Orchestration
: Does your code integrate with an orchestration framework, and can it be executed as an orchestrated workflow?

Parallelization
: How are you managing and scaling parallel access to source systems?

Deployment
: How are you handling the deployment of source code changes?

Conclusion

Source systems and their data are vital in the data engineering lifecycle. Data engineers tend to treat source systems as "someone else's problem"—do this at your peril! Data engineers who abuse source systems may need to look for another job when production goes down.

If there's a stick, there's also a carrot. Better collaboration with source system teams can lead to higher-quality data, more successful outcomes, and better data products. Create a bidirectional flow of communications with your counterparts on these teams; set up processes to notify of schema and application changes that affect

analytics and ML. Communicate your data needs proactively to assist application teams in the data engineering process.

Be aware that the integration between data engineers and source system teams is growing. One example is reverse ETL, which has long lived in the shadows but has recently risen into prominence. We also discussed that the event-streaming platform could serve a role in event-driven architectures and analytics; a source system can also be a data engineering system. Build shared systems where it makes sense to do so.

Look for opportunities to build user-facing data products. Talk to application teams about analytics they would like to present to their users or places where ML could improve the user experience. Make application teams stakeholders in data engineering, and find ways to share your successes.

Now that you understand the types of source systems and the data they generate, we'll next look at ways to store this data.

Additional Resources

- Confluent's "Schema Evolution and Compatibility" documentation
- *Database Internals* by Alex Petrov (O'Reilly)
- *Database System Concepts* by Abraham (Avi) Silberschatz et al. (McGraw Hill)
- "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction" by Jay Kreps
- "Modernizing Business Data Indexing" by Benjamin Douglas and Mohammad Mohtasham
- "NoSQL: What's in a Name" by Eric Evans

- “Test Data Quality at Scale with Deequ” by Dustin Lange et al.
- “The What, Why, and When of Single-Table Design with DynamoDB” by Alex DeBrie

CHAPTER 6

Storage

Storage is the cornerstone of the data engineering lifecycle (Figure 6-1) and underlies its major stages—ingestion, transformation, and serving. Data gets stored many times as it moves through the lifecycle. To paraphrase an old saying, it's storage all the way down. Whether data is needed seconds, minutes, days, months, or years later, it must persist in storage until systems are ready to consume it for further processing and transmission. Knowing the use case of the data and the way you will retrieve it in the future is the first step to choosing the proper storage solutions for your data architecture.

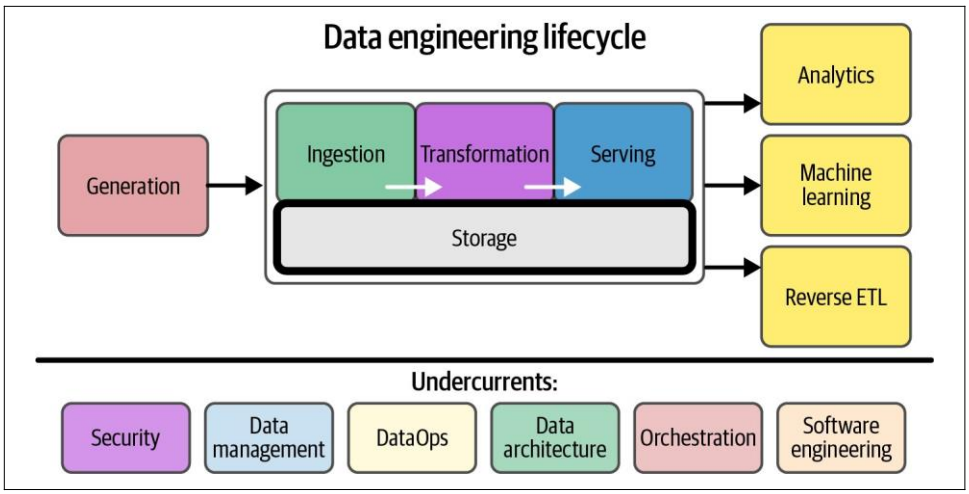


Figure 6-1. Storage plays a central role in the data engineering lifecycle

We also discussed storage in Chapter 5, but with a difference in focus and domain of control. Source systems are generally not maintained or controlled by data engineers. The storage that data engineers handle directly, which we'll focus on in this chapter, encompasses the data engineering lifecycle stages of ingesting data from source systems to serving data to deliver value with analytics, data science, etc. Many forms of storage undercut the entire data engineering lifecycle in some fashion.

To understand storage, we're going to start by studying the *raw ingredients* that compose storage systems, including hard drives, solid state drives, and system memory (see Figure 6-2). It's essential to understand the basic characteristics of physical storage technologies to assess the trade-offs inherent in any storage architecture. This section also discusses serialization and compression, key software elements of practical storage. (We defer a deeper technical discussion of serialization and compression to Appendix A.) We also discuss *caching*, which is critical in assembling storage systems.

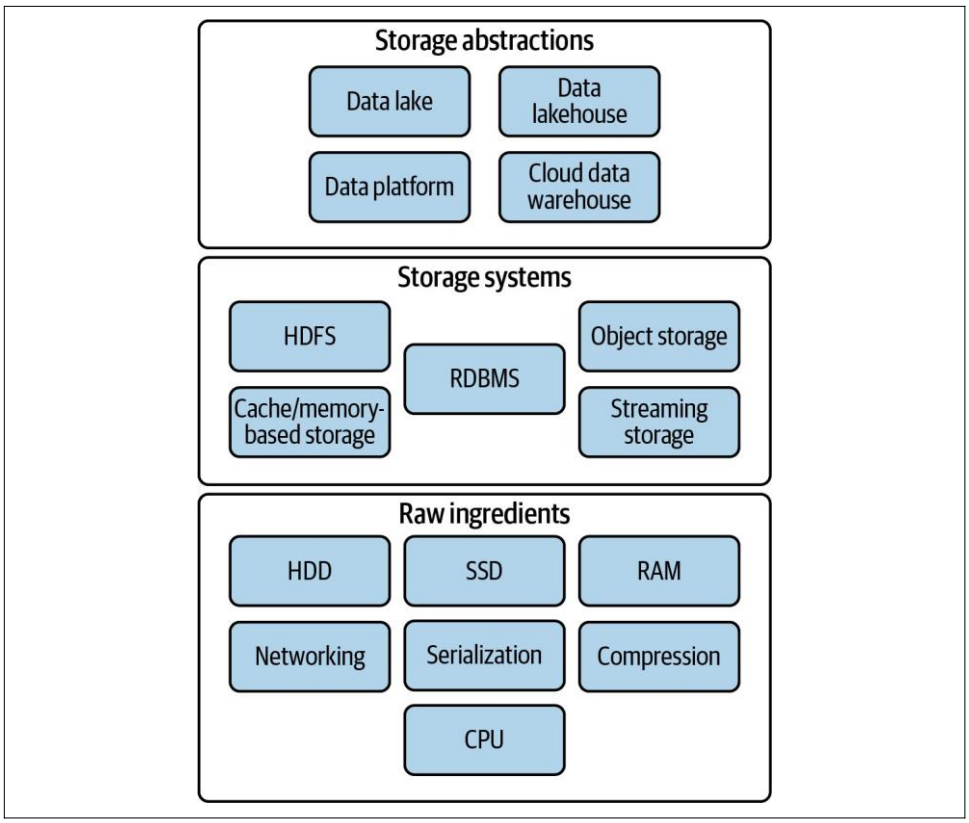


Figure 6-2. Raw ingredients, storage systems, and storage abstractions

Next, we'll look at *storage systems*. In practice, we don't directly access system memory or hard disks. These physical storage components exist inside servers and clusters that can ingest and retrieve data using various access paradigms.

Finally, we'll look at *storage abstractions*. Storage systems are assembled into a cloud data warehouse, a data lake, etc. When building data pipelines, engineers choose the appropriate abstractions for storing their data as it moves through the ingestion, transformation, and serving stages.

Raw Ingredients of Data Storage

Storage is so common that it's easy to take it for granted. We're often surprised by the number of software and data engineers who use storage every day but have little idea how it works behind the scenes or the trade-offs inherent in various storage media. As a result, we see storage used in some pretty...interesting ways. Though current managed services potentially free data engineers from the complexities of managing servers,

data engineers still need to be aware of underlying components' essential characteristics, performance considerations, durability, and costs.

In most data architectures, data frequently passes through magnetic storage, SSDs, and memory as it works its way through the various processing phases of a data pipeline. Data storage and query systems generally follow complex recipes involving distributed systems, numerous services, and multiple hardware storage layers. These systems require the right raw ingredients to function correctly.

Let's look at some of the raw ingredients of data storage: disk drives, memory, networking and CPU, serialization, compression, and caching.

Magnetic Disk Drive

Magnetic disks utilize spinning platters coated with a ferromagnetic film (Figure 6-3). This film is magnetized by a read/write head during write operations to physically encode binary data. The read/write head detects the magnetic field and outputs a bitstream during read operations. Magnetic disk drives have been around for ages. They still form the backbone of bulk data storage systems because they are significantly cheaper than SSDs per gigabyte of stored data.

On the one hand, these disks have seen extraordinary improvements in performance, storage density, and cost.[70] On the other hand, SSDs dramatically outperform magnetic disks on various metrics. Currently, commercial magnetic disk drives cost roughly 3 cents per gigabyte of capacity. (Note that we'll frequently use the abbreviations *HDD* and *SSD* to denote rotating magnetic disk and solid-state drives, respectively.)

[70] Andy Klein, "Hard Disk Drive (HDD) vs. Solid-State Drive (SSD): What's the Diff?," Backblaze blog, October 5, 2021, *https://oreil.ly/XBps8*.

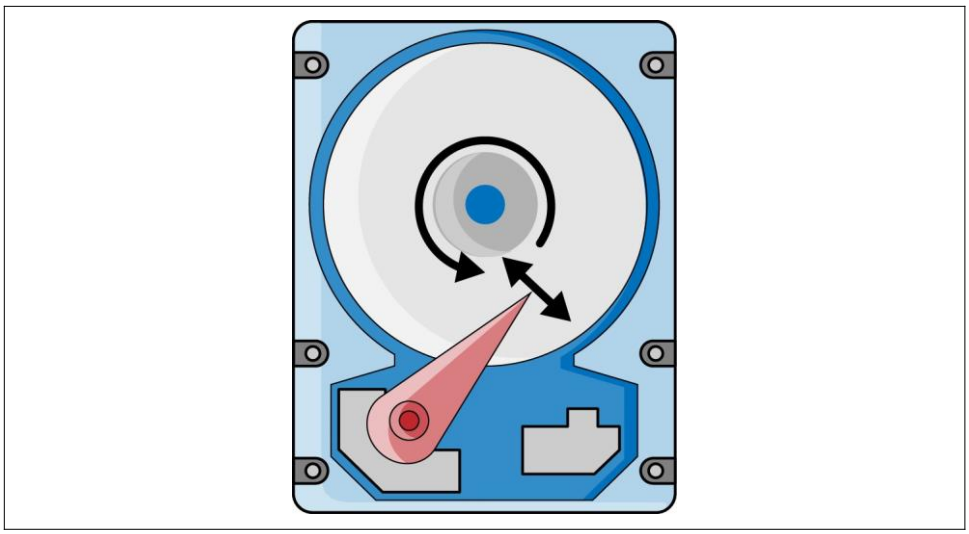


Figure 6-3. Magnetic disk head movement and rotation are essential in random access latency

IBM developed magnetic disk drive technology in the 1950s. Since then, magnetic disk capacities have grown steadily. The first commercial magnetic disk drive, the IBM 350, had a capacity of 3.75 megabytes. As of this writing, magnetic drives storing 20 TB are commercially available. In fact, magnetic disks continue to see rapid innovation, with methods such as heat-assisted magnetic recording (HAMR), shingled magnetic recording (SMR), and helium-filled disk enclosures being used to realize ever greater storage densities. In spite of the continuing improvements in drive capacity, other aspects of HDD performance are hampered by physics.

First, *disk transfer speed*, the rate at which data can be read and written, does not scale in proportion with disk capacity. Disk capacity scales with *areal density* (gigabits stored per square inch), whereas transfer speed scales with *linear density* (bits per inch). This means that if disk capacity grows by a factor of 4, transfer speed increases by only a factor of 2. Consequently, current data center drives support maximum data transfer speeds of 200–300 MB/s. To frame this another way, it takes more than 20 hours to read the entire contents of a 30 TB magnetic drive, assuming a transfer speed of 300 MB/s.

A second major limitation is seek time. To access data, the drive must physically relocate the read/write heads to the appropriate track on the disk. Third, in order to find a particular piece of data on the disk, the disk controller must wait for that data to rotate under the read/write heads. This leads to *rotational latency*. Typical commercial drives spinning at 7,200 revolutions per minute (RPM) seek time, and rotational latency, leads to over four milliseconds of overall average latency (time to access a selected piece of

data). A fourth limitation is input/output operations per second (IOPS), critical for transactional databases. A magnetic drive ranges from 50 to 500 IOPS.

Various tricks can improve latency and transfer speed. Using a higher rotational speed can increase transfer rate and decrease rotational latency. Limiting the radius of the disk platter or writing data into only a narrow band on the disk reduces seek time. However, none of these techniques makes magnetic drives remotely competitive with SSDs for random access lookups. SSDs can deliver data with significantly lower latency, higher IOPS, and higher transfer speeds, partially because there is no physically rotating disk or magnetic head to wait for.

As mentioned earlier, magnetic disks are still prized in data centers for their low datastorage costs. In addition, magnetic drives can sustain extraordinarily high transfer rates through parallelism. This is the critical idea behind cloud object storage: data can be distributed across thousands of disks in clusters. Data-transfer rates go up dramatically by reading from numerous disks simultaneously, limited primarily by network performance rather than disk transfer rate. Thus, network components and CPUs are also key raw ingredients in storage systems, and we will return to these topics shortly.

Solid-State Drive

Solid-state drives (SSDs) store data as charges in flash memory cells. SSDs eliminate the mechanical components of magnetic drives; the data is read by purely electronic means. SSDs can look up random data in less than 0.1 ms (100 microseconds). In addition, SSDs can scale both data-transfer speeds and IOPS by slicing storage into partitions with numerous storage controllers running in parallel. Commercial SSDs can support transfer speeds of many gigabytes per second and tens of thousands of IOPS.

Because of these exceptional performance characteristics, SSDs have revolutionized transactional databases and are the accepted standard for commercial deployments of OLTP systems. SSDs allow relational databases such as PostgreSQL, MySQL, and SQL Server to handle thousands of transactions per second.

However, SSDs are not currently the default option for high-scale analytics data storage. Again, this comes down to cost. Commercial SSDs typically cost 20–30 cents (USD) per gigabyte of capacity, nearly 10 times the cost per capacity of a magnetic drive. Thus, object storage on magnetic disks has emerged as the leading option for large-scale data storage in data lakes and cloud data warehouses.

SSDs still play a significant role in OLAP systems. Some OLAP databases leverage SSD caching to support high-performance queries on frequently accessed data. As low-latency OLAP becomes more popular, we expect SSD usage in these systems to follow suit.

Random Access Memory

We commonly use the terms *random access memory* (RAM) and *memory* interchangeably. Strictly speaking, magnetic drives and SSDs also serve as memory that stores data for later random access retrieval, but RAM has several specific characteristics:

- It is attached to a CPU and mapped into CPU address space.
- It stores the code that CPUs execute and the data that this code directly processes.
- It is *volatile*, while magnetic drives and SSDs are *nonvolatile*. Though they may occasionally fail and corrupt or lose data, drives generally retain data when powered off. RAM loses data in less than a second when it is unpowered.
- It offers significantly higher transfer speeds and faster retrieval times than SSD storage. DDR5 memory—the latest widely used standard for RAM—offers data retrieval latency on the order of 100 ns, roughly 1,000 times faster than SSD. A typical CPU can support 100 GB/s bandwidth to attached memory and millions of IOPS. (Statistics vary dramatically depending on the number of memory channels and other configuration details.)
- It is significantly more expensive than SSD storage, at roughly $10/GB (at the time of this writing).
- It is limited in the amount of RAM attached to an individual CPU and memory controller. This adds further to complexity and cost. High-memory servers typically utilize many interconnected CPUs on one board, each with a block of attached RAM.
- It is still significantly slower than CPU cache, a type of memory located directly on the CPU die or in the same package. Cache stores frequently and recently accessed data for ultrafast retrieval during processing. CPU designs incorporate several layers of cache of varying size and performance characteristics.

When we talk about system memory, we almost always mean *dynamic RAM*, a high-density, low-cost form of memory. Dynamic RAM stores data as charges in capacitors. These capacitors leak over time, so the data must be frequently *refreshed* (read and rewritten) to prevent data loss. The hardware memory controller handles these technical details; data engineers simply need to worry about bandwidth and retrieval latency characteristics. Other forms of memory, such as *static RAM*, are used in specialized applications such as CPU caches.

Current CPUs virtually always employ the *von Neumann architecture*, with code and data stored together in the same memory space. However, CPUs typically also support

the option to disable code execution in specific pages of memory for enhanced security. This feature is reminiscent of the *Harvard architecture*, which separates code and data.

RAM is used in various storage and processing systems and can be used for caching, data processing, or indexes. Several databases treat RAM as a primary storage layer, allowing ultra-fast read and write performance. In these applications, data engineers must always keep in mind the volatility of RAM. Even if data stored in memory is replicated across a cluster, a power outage that brings down several nodes could cause data loss. Architectures intended to durably store data may use battery backups and automatically dump all data to disk in the event of power loss.

Networking and CPU

Why are we mentioning networking and CPU as raw ingredients for storing data? Increasingly, storage systems are distributed to enhance performance, durability, and availability. We mentioned specifically that individual magnetic disks offer relatively low-transfer performance, but a cluster of disks parallelizes reads for significant performance scaling. While storage standards such as redundant arrays of independent disks (RAID) parallelize on a single server, cloud object storage clusters operate at a much larger scale, with disks distributed across a network and even multiple data centers and availability zones.

Availability zones are a standard cloud construct consisting of compute environments with independent power, water, and other resources. Multizonal storage enhances both the availability and durability of data.

CPUs handle the details of servicing requests, aggregating reads, and distributing writes. Storage becomes a web application with an API, backend service components, and load balancing. Network device performance and network topology are key factors in realizing high performance.

Data engineers need to understand how networking will affect the systems they build and use. Engineers constantly balance the durability and availability achieved by spreading out data geographically versus the performance and cost benefits of keeping storage in a small geographic area and close to data consumers or writers. Appendix B covers cloud networking and major relevant ideas.

Serialization

Serialization is another raw storage ingredient and a critical element of database design. The decisions around serialization will inform how well queries perform across a network, CPU overhead, query latency, and more. Designing a data lake, for example,

involves choosing a base storage system (e.g., Amazon S3) and standards for serialization that balance interoperability with performance considerations.

What is serialization, exactly? Data stored in system memory by software is generally not in a format suitable for storage on disk or transmission over a network. Serialization is the process of flattening and packing data into a standard format that a reader will be able to decode. Serialization formats provide a standard of data exchange. We might encode data in a row-based manner as an XML, JSON, or CSV file and pass it to another user who can then decode it using a standard library. A serialization algorithm has logic for handling types, imposes rules on data structure, and allows exchange between programming languages and CPUs. The serialization algorithm also has rules for handling exceptions. For instance, Python objects can contain cyclic references; the serialization algorithm might throw an error or limit nesting depth on encountering a cycle.

Low-level database storage is also a form of serialization. Row-oriented relational databases organize data as rows on disk to support speedy lookups and in-place updates. Columnar databases organize data into column files to optimize for highly efficient compression and support fast scans of large data volumes. Each serialization choice comes with a set of trade-offs, and data engineers tune these choices to optimize performance to requirements.

We provide a more detailed catalog of common data serialization techniques and formats in Appendix A. We suggest that data engineers become familiar with common serialization practices and formats, especially the most popular current formats (e.g., Apache Parquet), hybrid serialization (e.g., Apache Hudi), and in-memory serialization (e.g., Apache Arrow).

Compression

Compression is another critical component of storage engineering. On a basic level, compression makes data smaller, but compression algorithms interact with other details of storage systems in complex ways.

Highly efficient compression has three main advantages in storage systems. First, the data is smaller and thus takes up less space on the disk. Second, compression increases the practical scan speed per disk. With a 10:1 compression ratio, we go from scanning 200 MB/s per magnetic disk to an effective rate of 2 GB/s per disk.

The third advantage is in network performance. Given that a network connection between an Amazon EC2 instance and S3 provides 10 gigabits per second (Gbps) of bandwidth, a 10:1 compression ratio increases effective network bandwidth to 100 Gbps.

Compression also comes with disadvantages. Compressing and decompressing data entails extra time and resource consumption to read or write data. We undertake a more detailed discussion of compression algorithms and trade-offs in Appendix A.

Caching

We've already mentioned caching in our discussion of RAM. The core idea of caching is to store frequently or recently accessed data in a fast access layer. The faster the cache, the higher the cost and the less storage space available. Less frequently accessed data is stored in cheaper, slower storage. Caches are critical for data serving, processing, and transformation.

As we analyze storage systems, it is helpful to put every type of storage we utilize inside a *cache hierarchy* (Table 6-1). Most practical data systems rely on many cache layers assembled from storage with varying performance characteristics. This starts inside CPUs; processors may deploy up to four cache tiers. We move down the hierarchy to RAM and SSDs. Cloud object storage is a lower tier that supports long-term data retention and durability while allowing for data serving and dynamic data movement in pipelines.

Table 6-1. A heuristic cache hierarchy displaying storage types with approximate pricing and performance characteristics

Storage type	Data fetch latency[a]	Bandwidth	Price
CPU cache	1 nanosecond	1 TB/s	N/A
RAM	0.1 microseconds	100 GB/s	$10/GB
SSD	0.1 milliseconds	4 GB/s	$0.20/GB
HDD	4 milliseconds	300 MB/s	$0.03/GB
Object storage	100 milliseconds	10 GB/s	$0.02/GB per month
Archival storage	12 hours	Same as object storage once data is available	$0.004/GB per month

[a] A microsecond is 1,000 nanoseconds, and a millisecond is 1,000 microseconds.

We can think of archival storage as a *reverse cache*. Archival storage provides inferior access characteristics for low costs. Archival storage is generally used for data backups and to meet data-retention compliance requirements. In typical scenarios, this data will be accessed only in an emergency (e.g., data in a database might be lost and need to be recovered, or a company might need to look back at historical data for legal discovery).

Data Storage Systems

This section covers the major data storage systems you'll encounter as a data engineer. Storage systems exist at a level of abstraction above raw ingredients. For example, magnetic disks are a raw storage ingredient, while major cloud object storage platforms and HDFS are storage systems that utilize magnetic disks. Still higher levels of storage abstraction exist, such as data lakes and lakehouses (which we cover in "Data Engineering Storage Abstractions" on page 215).

Single Machine Versus Distributed Storage

As data storage and access patterns become more complex and outgrow the usefulness of a single server, distributing data to more than one server becomes necessary. Data can be stored on multiple servers, known as *distributed storage*. This is a distributed system whose purpose is to store data in a distributed fashion (Figure 6-4).

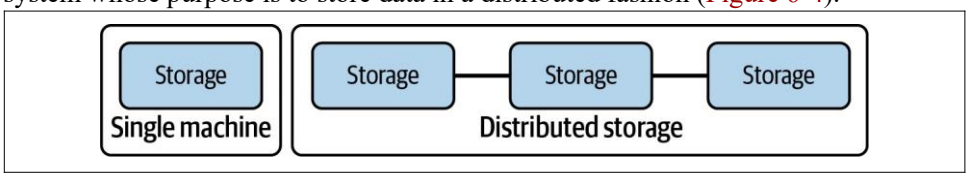


Figure 6-4. Single machine versus distributed storage on multiple servers

Distributed storage coordinates the activities of multiple servers to store, retrieve, and process data faster and at a larger scale, all while providing redundancy in case a server becomes unavailable. Distributed storage is common in architectures where you want built-in redundancy and scalability for large amounts of data. For example, object storage, Apache Spark, and cloud data warehouses rely on distributed storage architectures.

Data engineers must always be aware of the consistency paradigms of the distributed systems, which we'll explore next.

Eventual Versus Strong Consistency

A challenge with distributed systems is that your data is spread across multiple servers. How does this system keep the data consistent? Unfortunately, distributed systems pose a dilemma for storage and query accuracy. It takes time to replicate changes across the nodes of a system; often a balance exists between getting current data and getting "sorta" current data in a distributed database. Let's look at two common consistency patterns in distributed systems: eventual and strong.

We've covered ACID compliance throughout this book, starting in Chapter 5. Another acronym is *BASE*, which stands for *basically available, soft-state, eventual*

consistency. Think of it as the opposite of ACID. BASE is the basis of eventual consistency. Let's briefly explore its components:

Basically available
: Consistency is not guaranteed, but database reads and writes are made on a best-effort basis, meaning consistent data is available most of the time.

Soft-state
: The state of the transaction is fuzzy, and it's uncertain whether the transaction is committed or uncommitted.

Eventual consistency
: At *some* point, reading data will return consistent values.

If reading data in an eventually consistent system is unreliable, why use it? Eventual consistency is a common trade-off in large-scale, distributed systems. If you want to scale horizontally (across multiple nodes) to process data in high volumes, then eventually, consistency is often the price you'll pay. Eventual consistency allows you to retrieve data quickly without verifying that you have the latest version across all nodes.

The opposite of eventual consistency is *strong consistency*. With strong consistency, the distributed database ensures that writes to any node are first distributed with a consensus and that any reads against the database return consistent values. You'll use strong consistency when you can tolerate higher query latency and require correct data every time you read from the database.

Generally, data engineers make decisions about consistency in three places. First, the database technology itself sets the stage for a certain level of consistency. Second, configuration parameters for the database will have an impact on consistency. Third, databases often support some consistency configuration at an individual query level. For example, DynamoDB supports eventually consistent reads and strongly consistent reads. Strongly consistent reads are slower and consume more resources, so it is best to use them sparingly, but they are available when consistency is required. You should understand how your database handles consistency. Again, data engineers are tasked with understanding technology deeply and using it to solve problems appropriately. A data engineer might need to negotiate consistency requirements with other technical and business stakeholders. Note that this is both a technology and organizational problem; ensure that you have gathered requirements from your stakeholders and choose your technologies appropriately.

File Storage

We deal with files every day, but the notion of a file is somewhat subtle. A *file* is a data entity with specific read, write, and reference characteristics used by software and operating systems. We define a file to have the following characteristics:

Finite length
: A file is a finite-length stream of bytes.

Append operations
: We can append bytes to the file up to the limits of the host storage system.

Random access
: We can read from any location in the file or write updates to any location. *Object storage* behaves much like file storage but with key differences. While we set the stage for object storage by discussing file storage first, object storage is arguably much more important for the type of data engineering you'll do today. We will forward-reference the object storage discussion extensively over the next few pages.

File storage systems organize files into a directory tree. The directory reference for a file might look like this:

```
/Users/matthewhousley/output.txt
```

When this file reference is passed to the operating system, it starts at the root directory `/`, finds `Users`, `matthewhousley`, and finally `output.txt`. Working from the left, each directory is contained inside a parent directory, until we finally reach the file `output.txt`. This example uses Unix semantics, but Windows file reference semantics are similar. The filesystem stores each directory as metadata about the files and directories that it contains. This metadata consists of the name of each entity, relevant permission details, and a pointer to the actual entity. To find a file on disk, the operating system looks at the metadata at each hierarchy level and follows the pointer to the next subdirectory entity until finally reaching the file itself.

Note that other file-like data entities generally don't necessarily have all these properties. For example, *objects* in object storage support only the first characteristic, finite length, but are still extremely useful. We discuss this in "Object Storage" on page 205.

In cases where file storage paradigms are necessary for a pipeline, be careful with state and try to use ephemeral environments as much as possible. Even if you must process files on a server with an attached disk, use object storage for intermediate storage between processing steps. Try to reserve manual, low-level file processing for one-time ingestion steps or the exploratory stages of pipeline development.

Local disk storage

The most familiar type of file storage is an operating system–managed filesystem on a local disk partition of SSD or magnetic disk. New Technology File System (NTFS) and ext4 are popular filesystems on Windows and Linux, respectively. The operating system handles the details of storing directory entities, files, and metadata. Filesystems are designed to write data to allow for easy recovery in the event of power loss during a write, though any unwritten data will still be lost.

Local filesystems generally support full read after write consistency; reading immediately after a write will return the written data. Operating systems also employ various locking strategies to manage concurrent writing attempts to a file.

Local disk filesystems may also support advanced features such as journaling, snapshots, redundancy, the extension of the filesystem across multiple disks, full disk encryption, and compression. In "Block Storage" on page 202, we also discuss RAID.

Network-attached storage

Network-attached storage (NAS) systems provide a file storage system to clients over a network. NAS is a prevalent solution for servers; they quite often ship with built-in dedicated NAS interface hardware. While there are performance penalties to accessing the filesystem over a network, significant advantages to storage virtualization also exist, including redundancy and reliability, fine-grained control of resources, storage pooling across multiple disks for large virtual volumes, and file sharing across multiple machines. Engineers should be aware of the consistency model provided by their NAS solution, especially when multiple clients will potentially access the same data.

A popular alternative to NAS is a storage area network (SAN), but SAN systems provide block-level access without the filesystem abstraction. We cover SAN systems in "Block Storage" on page 202.

Cloud filesystem services

Cloud filesystem services provide a fully managed filesystem for use with multiple cloud VMs and applications, potentially including clients outside the cloud environment. Cloud filesystems should not be confused with standard storage attached to VMs—generally, block storage with a filesystem managed by the VM operating system. Cloud filesystems behave much like NAS solutions, but the details of networking, managing disk clusters, failures, and configuration are fully handled by the cloud vendor.

For example, Amazon Elastic File System (EFS) is an extremely popular example of a cloud filesystem service. Storage is exposed through the NFS 4 protocol, which is also used by NAS systems. EFS provides automatic scaling and pay-per-storage pricing

with no advanced storage reservation required. The service also provides *local* read-after-write consistency (when reading from the machine that performed the write). It also offers open-after-close consistency across the full filesystem. In other words, once an application closes a file, subsequent readers will see changes saved to the closed file.

Block Storage

Fundamentally, *block storage* is the type of raw storage provided by SSDs and magnetic disks. In the cloud, virtualized block storage is the standard for VMs. These block storage abstractions allow fine control of storage size, scalability, and data durability beyond that offered by raw disks.

In our earlier discussion of SSDs and magnetic disks, we mentioned that with these random-access devices, the operating system can seek, read, and write any data on the disk. A *block* is the smallest addressable unit of data supported by a disk. This was often 512 bytes of usable data on older disks, but it has now grown to 4,096 bytes for most current disks, making writes less fine-grained but dramatically reducing the overhead of managing blocks. Blocks typically contain extra bits for error detection/correction and other metadata.

Blocks on magnetic disks are geometrically arranged on a physical platter. Two blocks on the same track can be read without moving the head, while reading two blocks on separate tracks requires a seek. Seek time can occur between blocks on an SSD, but this is infinitesimal compared to the seek time for magnetic disk tracks.

Block storage applications

Transactional database systems generally access disks at a block level to lay out data for optimal performance. For row-oriented databases, this originally meant that rows of data were written as continuous streams; the situation has grown more complicated with the arrival of SSDs and their associated seek-time performance improvements, but transactional databases still rely on the high random access performance offered by direct access to a block storage device.

Block storage also remains the default option for operating system boot disks on cloud VMs. The block device is formatted much as it would be directly on a physical disk, but the storage is usually virtualized. (See "Cloud virtualized block storage" on page 203.)

RAID

RAID stands for *redundant array of independent disks*, as noted previously. RAID simultaneously controls multiple disks to improve data durability, enhance performance, and combine capacity from multiple drives. An array can appear to the

operating system as a single block device. Many encoding and parity schemes are available, depending on the desired balance between enhanced effective bandwidth and higher fault tolerance (tolerance for many disk failures).

Storage area network

Storage area network (SAN) systems provide virtualized block storage devices over a network, typically from a storage pool. SAN abstraction can allow fine-grained storage scaling and enhance performance, availability, and durability. You might encounter SAN systems if you're working with on-premises storage systems; you might also encounter a cloud version of SAN, as in the next subsection.

Cloud virtualized block storage

Cloud virtualized block storage solutions are similar to SAN but free engineers from dealing with SAN clusters and networking details. We'll look at Amazon Elastic Block Store (EBS) as a standard example; other public clouds have similar offerings. EBS is the default storage for Amazon EC2 virtual machines; other cloud providers also treat virtualized object storage as a key component of their VM offerings.

EBS offers several tiers of service with different performance characteristics. Generally, EBS performance metrics are given in IOPS and throughput (transfer speed). The higher performance tiers of EBS storage are backed by SSD disks, while magnetic disk-backed storage offers lower IOPS but costs less per gigabyte.

EBS volumes store data separate from the instance host server but in the same zone to support high performance and low latency (Figure 6-5). This allows EBS volumes to persist when an EC2 instance shuts down, when a host server fails, or even when the instance is deleted. EBS storage is suitable for applications such as databases, where data durability is a high priority. In addition, EBS replicates all data to at least two separate host machines, protecting data if a disk fails.

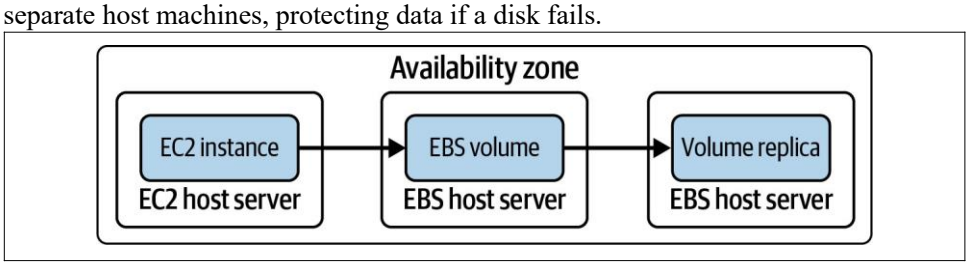


Figure 6-5. EBS volumes replicate data to multiple hosts and disks for high durability and availability, but are not resilient to the failure of an availability zone

EBS storage virtualization also supports several advanced features. For example, EBS volumes allow instantaneous point-in-time snapshots while the drive is used. Although it still takes some time for the snapshot to be replicated to S3, EBS can effectively

freeze the state of data blocks when the snapshot is taken, while allowing the client machine to continue using the disk. In addition, snapshots after the initial full backup are differential; only changed blocks are written to S3 to minimize storage costs and backup time.

EBS volumes are also highly scalable. At the time of this writing, some EBS volume classes can scale up to 64 TiB, 256,000 IOPS, and 4,000 MiB/s.

Local instance volumes

Cloud providers also offer block storage volumes that are physically attached to the host server running a virtual machine. These storage volumes are generally very low cost (included with the price of the VM in the case of Amazon's EC2 instance store) and provide low latency and high IOPS.

Instance store volumes (Figure 6-6) behave essentially like a disk physically attached to a server in a data center. One key difference is that when a VM shuts down or is deleted, the contents of the locally attached disk are lost, whether or not this event was caused by intentional user action. This ensures that a new virtual machine cannot read disk contents belonging to a different customer.

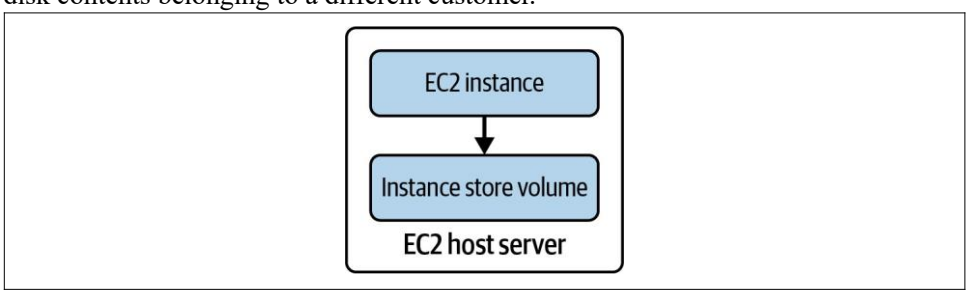


Figure 6-6. Instance store volumes offer high performance and low cost but do not protect data in the event of disk failure or VM shutdown

Locally attached disks support none of the advanced virtualization features offered by virtualized storage services like EBS. The locally attached disk is not replicated, so a physical disk failure can lose or corrupt data even if the host VM continues running. Furthermore, locally attached volumes do not support snapshots or other backup features.

Despite these limitations, locally attached disks are extremely useful. In many cases, we use disks as a local cache and hence don't need all the advanced virtualization features of a service like EBS. For example, suppose we're running AWS EMR on EC2 instances. We may be running an ephemeral job that consumes data from S3, stores it temporarily in the distributed filesystem running across the instances, processes the data, and writes the results back to S3. The EMR filesystem builds in

replication and redundancy and is serving as a cache rather than permanent storage. The EC2 instance store is a perfectly suitable solution in this case and can enhance performance since data can be read and processed locally without flowing over a network (see Figure 6-7).

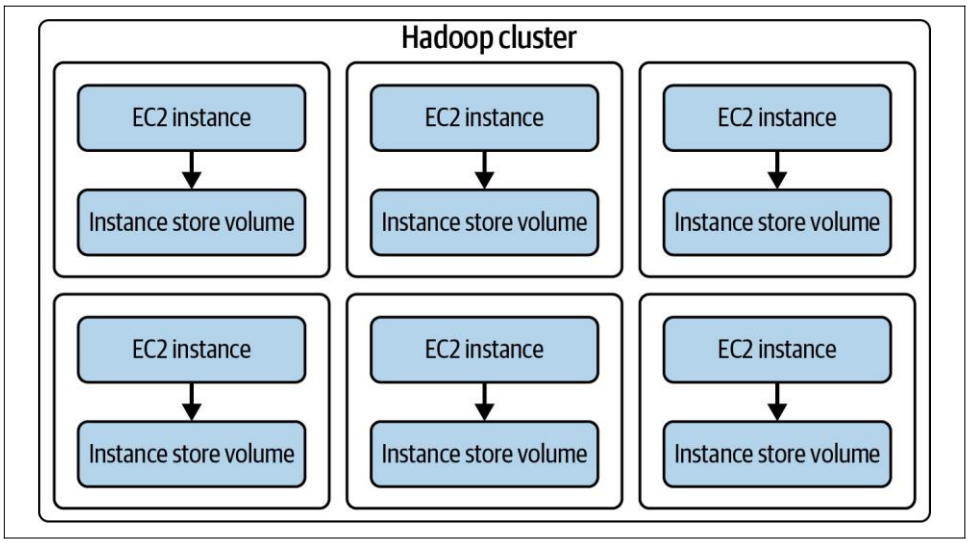


Figure 6-7. Instance store volumes can be used as a processing cache in an ephemeral Hadoop cluster

We recommend that engineers think about locally attached storage in worst-case scenarios. What are the consequences of a local disk failure? Of an accidental VM or cluster shutdown? Of a zonal or regional cloud outage? If none of these scenarios will have catastrophic consequences when data on locally attached volumes is lost, local storage may be a cost-effective and performant option. In addition, simple mitigation strategies (periodic checkpoint backups to S3) can prevent data loss.

Object Storage

Object storage contains *objects* of all shapes and sizes (Figure 6-8). The term *object storage* is somewhat confusing because *object* has several meanings in computer science. In this context, we're talking about a specialized file-like construct. It could be any type of file—TXT, CSV, JSON, images, videos, or audio.

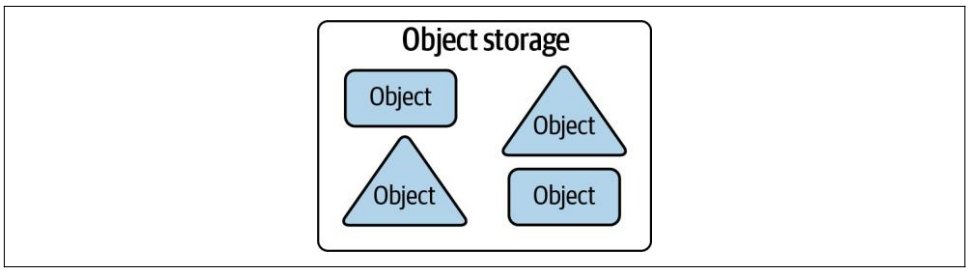


Figure 6-8. Object storage contains immutable objects of all shapes and sizes. Unlike files on a local disk, objects cannot be modified in place.

Object stores have grown in importance and popularity with the rise of big data and the cloud. Amazon S3, Azure Blob Storage, and Google Cloud Storage (GCS) are widely used object stores. In addition, many cloud data warehouses (and a growing number of databases) utilize object storage as their storage layer, and cloud data lakes generally sit on object stores.

Although many on-premises object storage systems can be installed on server clusters, we'll focus mostly on fully managed cloud object stores. From an operational perspective, one of the most attractive characteristics of cloud object storage is that it is straightforward to manage and use. Object storage was arguably one of the first "serverless" services; engineers don't need to consider the characteristics of underlying server clusters or disks.

An object store is a key-value store for immutable data objects. We lose much of the writing flexibility we expect with file storage on a local disk in an object store. Objects don't support random writes or append operations; instead, they are written once as a stream of bytes. After this initial write, objects become immutable. To change data in an object or append data to it, we must rewrite the full object. Object stores generally support random reads through range requests, but these lookups may perform much worse than random reads from data stored on an SSD.

For a software developer used to leveraging local random access file storage, the characteristics of objects might seem like constraints, but less is more; object stores don't need to support locks or change synchronization, allowing data storage across massive disk clusters. Object stores support extremely performant parallel stream writes and reads across many disks, and this parallelism is hidden from engineers, who can simply deal with the stream rather than communicating with individual disks. In a cloud environment, write speed scales with the number of streams being written up to quota limits set by the vendor. Read bandwidth can scale with the number of parallel requests, the number of virtual machines employed to read data, and the number of CPU cores. These characteristics make object storage ideal for serving high-volume web traffic or delivering data to highly parallel distributed query engines.

Typical cloud object stores save data in several availability zones, dramatically reducing the odds that storage will go fully offline or be lost in an unrecoverable way. This durability and availability are built into the cost; cloud storage vendors offer other storage classes at discounted prices in exchange for reduced durability or availability. We'll discuss this in "Storage classes and tiers" on page 210.

Cloud object storage is a key ingredient in separating compute and storage, allowing engineers to process data with ephemeral clusters and scale these clusters up and down on demand. This is a key factor in making big data available to smaller organizations that can't afford to own hardware for data jobs that they'll run only occasionally. Some major tech companies will continue to run permanent Hadoop clusters on their hardware. Still, the general trend is that most organizations will move data processing to the cloud, using an object store as essential storage and serving layer while processing data on ephemeral clusters.

In object storage, available storage space is also highly scalable, an ideal characteristic for big data systems. Storage space is constrained by the number of disks the storage provider owns, but these providers handle exabytes of data. In a cloud environment, available storage space is virtually limitless; in practice, the primary limit on storage space for public cloud customers is budget. From a practical standpoint, engineers can quickly store massive quantities of data for projects without planning months in advance for necessary servers and disks.

Object stores for data engineering applications

From the standpoint of data engineering, object stores provide excellent performance for large batch reads and batch writes. This corresponds well to the use case for massive OLAP systems. A bit of data engineering folklore says that object stores are not good for updates, but this is only partially true. Object stores are an inferior fit for transactional workloads with many small updates every second; these use cases are much better served by transactional databases or block storage systems. Object stores work well for a low rate of update operations, where each operation updates a large volume of data.

Object stores are now the gold standard of storage for data lakes. In the early days of data lakes, write once, read many (WORM) was the operational standard, but this had more to do with the complexities of managing data versions and files than the limitations of HDFS and object stores. Since then, systems such as Apache Hudi and Delta Lake have emerged to manage this complexity, and privacy regulations such as GDPR and CCPA have made deletion and update capabilities imperative. Update management for object storage is the central idea behind the data lakehouse concept, which we introduced in Chapter 3.

Object storage is an ideal repository for unstructured data in any format beyond these structured data applications. Object storage can house any binary data with no constraints on type or structure and frequently plays a role in ML pipelines for raw text, images, video, and audio.

Object lookup

As we mentioned, object stores are key-value stores. What does this mean for engineers? It's critical to understand that, unlike file stores, object stores do not utilize a directory tree to find objects. The object store uses a top-level logical container (a bucket in S3 and GCS) and references objects by key. A simple example in S3 might look like this:

```
S3://oreilly-data-engineering-book/data-example.json
```

In this case, `S3://oreilly-data-engineering-book/` is the bucket name, and `dataexample.json` is the key pointing to a particular object. S3 bucket names must be unique across all of AWS. Keys are unique within a bucket. Although cloud object stores may appear to support directory tree semantics, no true directory hierarchy exists. We might store an object with the following full path:

```
S3://oreilly-data-engineering-book/project-data/11/23/2021/data.txt
```

On the surface, this looks like subdirectories you might find in a regular file folder system: `project-data`, `11`, `23`, and `2021`. Many cloud console interfaces allow users to view the objects inside a "directory," and cloud command-line tools often support Unix-style commands such as `ls` inside an object store directory. However, behind the scenes, the object system does not traverse a directory tree to reach the object. Instead, it simply sees a key (`project-data/11/23/2021/data.txt`) that happens to match directory semantics. This might seem like a minor technical detail, but engineers need to understand that certain "directory"-level operations are costly in an object store. To run `aws ls S3://oreilly-data-engineering-book/projectdata/11/` the object store must filter keys on the key prefix `project-data/11/`. If the bucket contains millions of objects, this operation might take some time, even if the "subdirectory" houses only a few objects.

Object consistency and versioning

As mentioned, object stores don't support in-place updates or appends as a general rule. We write a new object under the same key to update an object. When data engineers utilize updates in data processes, they must be aware of the consistency model for the object store they're using. Object stores may be eventually consistent or strongly consistent. For example, until recently, S3 was *eventually consistent*; after a new version of an object was written under the same key, the object store might sometimes return the old version of the object. The *eventual* part of *eventual*

consistency means that after enough time has passed, the storage cluster reaches a state such that only the latest version of the object will be returned. This contrasts with the *strong consistency* model we expect of local disks attached to a server: reading after a write will return the most recently written data.

It might be desirable to impose strong consistency on an object store for various reasons, and standard methods are used to achieve this. One approach is to add a strongly consistent database (e.g., PostgreSQL) to the mix. Writing an object is now a two-step process:

1. Write the object.
2. Write the returned metadata for the object version to the strongly consistent database.

The version metadata (an object hash or an object timestamp) can uniquely identify an object version in conjunction with the object key. To read an object, a reader undertakes the following steps:

1. Fetch the latest object metadata from the strongly consistent database.
2. Query object metadata using the object key. Read the object data if it matches the metadata fetched from the consistent database.
3. If the object metadata does not match, repeat step 2 until the latest version of the object is returned.

A practical implementation has exceptions and edge cases to consider, such as when the object gets rewritten during this querying process. These steps can be managed behind an API so that an object reader sees a strongly consistent object store at the cost of higher latency for object access.

Object versioning is closely related to object consistency. When we rewrite an object under an existing key in an object store, we're essentially writing a brand-new object, setting references from the existing key to the object, and deleting the old object references. Updating all references across the cluster takes time, hence the potential for stale reads. Eventually, the storage cluster garbage collector deallocates the space dedicated to the dereferenced data, recycling disk capacity for use by new objects.

With object versioning turned on, we add additional metadata to the object that stipulates a version. While the default key reference gets updated to point to the new object, we retain other pointers to previous versions. We also maintain a version list so that clients can get a list of all object versions, and then pull a specific version. Because old versions of the object are still referenced, they aren't cleaned up by the garbage collector.

If we reference an object with a version, the consistency issue with some object storage systems disappears: the key and version metadata together form a unique reference to a particular, immutable data object. We will always get the same object back when we use this pair, provided that we haven't deleted it. The consistency issue still exists when a client requests the "default" or "latest" version of an object.

The principal overhead that engineers need to consider with object versioning is the cost of storage. Historical versions of objects generally have the same associated storage costs as current versions. Object version costs may be nearly insignificant or catastrophically expensive, depending on various factors. The data size is an issue, as is update frequency; more object versions can lead to significantly larger data size. Keep in mind that we're talking about brute-force object versioning. Object storage systems generally store full object data for each version, not differential snapshots.

Engineers also have the option of deploying storage lifecycle policies. Lifecycle policies allow automatic deletion of old object versions when certain conditions are met (e.g., when an object version reaches a certain age or many newer versions exist). Cloud vendors also offer various archival data tiers at heavily discounted prices, and the archival process can be managed using lifecycle policies.

Storage classes and tiers

Cloud vendors now offer storage classes that discount data storage pricing in exchange for reduced access or reduced durability. We use the term *reduced access* here because many of these storage tiers still make data highly available, but with high retrieval costs in exchange for reduced storage costs.

Let's look at a couple of examples in S3 since Amazon is a benchmark for cloud service standards. The S3 Standard-Infrequent Access storage class discounts monthly storage costs for increased data retrieval costs. (See "A Brief Detour on Cloud Economics" on page 125 for a theoretical discussion of the economics of cloud storage tiers.) Amazon also offers the Amazon S3 One Zone-Infrequent Access tier, replicating only to a single zone. Projected availability drops from 99.9% to 99.5% to account for the possibility of a zonal outage. Amazon still claims extremely high data durability, with the caveat that data will be lost if an availability zone is destroyed.

Further down the tiers of reduced access are the archival tiers in S3 Glacier. S3 Glacier promises a dramatic reduction in long-term storage costs for much higher access costs. Users have various retrieval speed options, from minutes to hours, with higher retrieval costs for faster access. For example, at the time of this writing, S3 Glacier Deep Archive discounts storage costs even further; Amazon advertises that storage costs start at $1 per terabyte per month. In exchange, data restoration takes 12 hours. In addition,

this storage class is designed for data that will be stored 7–10 years and be accessed only one to two times per year.

Be aware of how you plan to utilize archival storage, as it's easy to get into and often costly to access data, especially if you need it more often than expected. See Chapter 4 for a more extensive discussion of archival storage economics.

Object store–backed filesystems

Object store synchronization solutions have become increasingly popular. Tools like s3fs and Amazon S3 File Gateway allow users to mount an S3 bucket as local storage. Users of these tools should be aware of the characteristics of writes to the filesystem and how these will interact with the characteristics and pricing of object storage. File Gateway, for example, handles changes to files fairly efficiently by combining portions of objects into a new object using the advanced capabilities of S3. However, high-speed transactional writing will overwhelm the update capabilities of an object store. Mounting object storage as a local filesystem works well for files that are updated infrequently.

Cache and Memory-Based Storage Systems

As discussed in "Raw Ingredients of Data Storage" on page 191, RAM offers excellent latency and transfer speeds. However, traditional RAM is extremely vulnerable to data loss because a power outage lasting even a second can erase data. RAM-based storage systems are generally focused on caching applications, presenting data for quick access and high bandwidth. Data should generally be written to a more durable medium for retention purposes.

These ultra-fast cache systems are useful when data engineers need to serve data with ultra-fast retrieval latency.

Example: Memcached and lightweight object caching

Memcached is a key-value store designed for caching database query results, API call responses, and more. Memcached uses simple data structures, supporting either string or integer types. Memcached can deliver results with very low latency while also taking the load off backend systems.

Example: Redis, memory caching with optional persistence

Like Memcached, *Redis* is a key-value store, but it supports somewhat more complex data types (such as lists or sets). Redis also builds in multiple persistence mechanisms, including snapshotting and journaling. With a typical configuration, Redis writes data

roughly every two seconds. Redis is thus suitable for extremely high-performance applications but can tolerate a small amount of data loss.

The Hadoop Distributed File System

In the recent past, "Hadoop" was virtually synonymous with "big data." The Hadoop Distributed File System is based on Google File System (GFS) and was initially engineered to process data with the MapReduce programming model. Hadoop is similar to object storage but with a key difference: Hadoop combines compute and storage on the same nodes, where object stores typically have limited support for internal processing.

Hadoop breaks large files into *blocks*, chunks of data less than a few hundred megabytes in size. The filesystem is managed by the *NameNode*, which maintains directories, file metadata, and a detailed catalog describing the location of file blocks in the cluster. In a typical configuration, each block of data is replicated to three nodes. This increases both the durability and availability of data. If a disk or node fails, the replication factor for some file blocks will fall below 3. The NameNode will instruct other nodes to replicate these file blocks so that they again reach the correct replication factor. Thus, the probability of losing data is very low, barring a *correlated failure* (e.g., an asteroid hitting the data center).

Hadoop is not simply a storage system. Hadoop combines compute resources with storage nodes to allow in-place data processing. This was originally achieved using the MapReduce programming model, which we discuss in Chapter 8.

Hadoop is dead. Long live Hadoop!

We often see claims that Hadoop is dead. This is only partially true. Hadoop is no longer a hot, bleeding-edge technology. Many Hadoop ecosystem tools such as Apache Pig are now on life support and primarily used to run legacy jobs. The pure MapReduce programming model has fallen by the wayside. HDFS remains widely used in various applications and organizations.

Hadoop still appears in many legacy installations. Many organizations that adopted Hadoop during the peak of the big data craze have no immediate plans to migrate to newer technologies. This is a good choice for companies that run massive (thousandnode) Hadoop clusters and have the resources to maintain on-premises systems effectively. Smaller companies may want to reconsider the cost overhead and scale limitations of running a small Hadoop cluster against migrating to cloud solutions.

In addition, HDFS is a key ingredient of many current big data engines, such as Amazon EMR. In fact, Apache Spark is still commonly run on HDFS clusters. We discuss this in more detail in "Separation of Compute from Storage" on page 220.

Streaming Storage

Streaming data has different storage requirements than nonstreaming data. In the case of message queues, stored data is temporal and expected to disappear after a certain duration. However, distributed, scalable streaming frameworks like Apache Kafka now allow extremely long-duration streaming data retention. Kafka supports indefinite data retention by pushing old, infrequently accessed messages down to object storage. Kafka competitors (including Amazon Kinesis, Apache Pulsar, and Google Cloud Pub/Sub) also support long data retention.

Closely related to data retention in these systems is the notion of replay. *Replay* allows a streaming system to return a range of historical stored data. Replay is the standard data-retrieval mechanism for streaming storage systems. Replay can be used to run batch queries over a time range or to reprocess data in a streaming pipeline. Chapter 7 covers replay in more depth.

Other storage engines have emerged for real-time analytics applications. In some sense, transactional databases emerged as the first real-time query engines; data becomes visible to queries as soon as it is written. However, these databases have well-known scaling and locking limitations, especially for analytics queries that run across large volumes of data. While scalable versions of row-oriented transactional databases have overcome some of these limitations, they are still not truly optimized for analytics at scale.

Indexes, Partitioning, and Clustering

Indexes provide a map of the table for particular fields and allow extremely fast lookup of individual records. Without indexes, a database would need to scan an entire table to find the records satisfying a `WHERE` condition.

In most RDBMSs, indexes are used for primary table keys (allowing unique identification of rows) and foreign keys (allowing joins with other tables). Indexes can also be applied to other columns to serve the needs of specific applications. Using indexes, an RDBMS can look up and update thousands of rows per second.

We do not cover transactional database records in depth in this book; numerous technical resources are available on this topic. Rather, we are interested in the evolution away from indexes in analytics-oriented storage systems and some new developments in indexes for analytics use cases.

The evolution from rows to columns

An early data warehouse was typically built on the same type of RDBMS used for transactional applications. The growing popularity of MPP systems meant a shift

toward parallel processing for significant improvements in scan performance across large quantities of data for analytics purposes. However, these row-oriented MPPs still used indexes to support joins and condition checking.

In "Raw Ingredients of Data Storage" on page 191, we discuss columnar serialization. *Columnar serialization* allows a database to scan only the columns required for a particular query, sometimes dramatically reducing the amount of data read from the disk. In addition, arranging data by column packs similar values next to each other, yielding high-compression ratios with minimal compression overhead. This allows data to be scanned more quickly from disk and over a network.

Columnar databases perform poorly for transactional use cases—i.e., when we try to look up large numbers of individual rows asynchronously. However, they perform extremely well when large quantities of data must be scanned—e.g., for complex data transformations, aggregations, statistical calculations, or evaluation of complex conditions on large datasets.

In the past, columnar databases performed poorly on joins, so the advice for data engineers was to denormalize data, using wide schemas, arrays, and nested data wherever possible. Join performance for columnar databases has improved dramatically in recent years, so while there can still be performance advantages in denormalization, this is no longer a necessity. You'll learn more about normalization and denormalization in Chapter 8.

From indexes to partitions and clustering

While columnar databases allow for fast scan speeds, it's still helpful to reduce the amount of data scanned as much as possible. In addition to scanning only data in columns relevant to a query, we can partition a table into multiple subtables by splitting it on a field. It is quite common in analytics and data science use cases to scan over a time range, so date- and time-based partitioning is extremely common. Columnar databases generally support a variety of other partition schemes as well.

Clusters allow finer-grained organization of data within partitions. A clustering scheme applied within a columnar database sorts data by one or a few fields, colocating similar values. This improves performance for filtering, sorting, and joining these values.

Example: Snowflake micro-partitioning

We mention Snowflake micro-partitioning because it's a good example of recent developments and evolution in approaches to columnar storage. *Micro partitions* are sets of rows between 50 and 500 megabytes in uncompressed size. Snowflake uses an algorithmic approach that attempts to cluster together similar rows. This contrasts the traditional naive approach to partitioning on a single designated field, such as a date.

Snowflake specifically looks for values that are repeated in a field across many rows. This allows aggressive *pruning* of queries based on predicates. For example, a `WHERE` clause might stipulate the following:

```
WHERE created_date='2022-01-02'
```

In such a query, Snowflake excludes any micro-partitions that don't include this date, effectively pruning this data. Snowflake also allows overlapping micro-partitions, potentially partitioning on multiple fields showing significant repeats.

Efficient pruning is facilitated by Snowflake's metadata database, which stores a description of each micro-partition, including the number of rows and value ranges for fields. At each query stage, Snowflake analyzes micro-partitions to determine which ones need to be scanned. Snowflake uses the term *hybrid columnar storage*,[71] partially referring to the fact that its tables are broken into small groups of rows, even though storage is fundamentally columnar. The metadata database plays a role similar to an index in a traditional relational database.

[71] Benoit Dageville, "The Snowflake Elastic Data Warehouse," *SIGMOD '16: Proceedings of the 2016 International Conference on Management of Data* (June 2016): 215–226, *https://oreil.ly/Tc1su*.

Data Engineering Storage Abstractions

Data engineering storage abstractions are data organization and query patterns that sit at the heart of the data engineering lifecycle and are built atop the data storage systems discussed previously (see Figure 6-3). We introduced many of these abstractions in Chapter 3, and we will revisit them here.

The main types of abstractions we'll concern ourselves with are those that support data science, analytics, and reporting use cases. These include data warehouse, data lake, data lakehouse, data platforms, and data catalogs. We won't cover source systems, as they are discussed in Chapter 5.

The storage abstraction you require as a data engineer boils down to a few key considerations:

Purpose and use case
: You must first identify the purpose of storing the data. What is it used for?

Update patterns
: Is the abstraction optimized for bulk updates, streaming inserts, or upserts?

Cost
: What are the direct and indirect financial costs? The time to value? The opportunity costs?

Separate storage and compute
: The trend is toward separating storage and compute, but most systems hybridize separation and colocation. We cover this in "Separation of Compute from Storage" on page 220 since it affects purpose, speed, and cost.

You should know that the popularity of separating storage from compute means the lines between OLAP databases and data lakes are increasingly blurring. Major cloud data warehouses and data lakes are on a collision course. In the future, the differences between these two may be in name only since they might functionally and technically be very similar under the hood.

The Data Warehouse

Data warehouses are a standard OLAP data architecture. As discussed in Chapter 3, the term *data warehouse* refers to technology platforms (e.g., Google BigQuery and Teradata), an architecture for data centralization, and an organizational pattern within a company. In terms of storage trends, we've evolved from building data warehouses atop conventional transactional databases, row-based MPP systems (e.g., Teradata and IBM Netezza), and columnar MPP systems (e.g., Vertica and Teradata Columnar) to cloud data warehouses and data platforms. (See our data warehousing discussion in Chapter 3 for more details on MPP systems.)

In practice, cloud data warehouses are often used to organize data into a data lake, a storage area for massive amounts of unprocessed raw data, as originally conceived by James Dixon.[72] Cloud data warehouses can handle massive amounts of raw text and complex JSON documents. The limitation is that cloud data warehouses cannot handle truly unstructured data, such as images, video, or audio, unlike a true data lake. Cloud data warehouses can be coupled with object storage to provide a complete data-lake solution.

The Data Lake

The *data lake* was originally conceived as a massive store where data was retained in raw, unprocessed form. Initially, data lakes were built primarily on Hadoop systems, where cheap storage allowed for retention of massive amounts of data without the cost overhead of a proprietary MPP system.

The last five years have seen two major developments in the evolution of data lake storage. First, a major migration toward *separation of compute and storage* has occurred. In practice, this means a move away from Hadoop toward cloud object storage for long-term retention of data. Second, data engineers discovered that much of the functionality offered by MPP systems (schema management; update, merge and delete capabilities) and initially dismissed in the rush to data lakes was, in fact, extremely useful. This led to the notion of the data lakehouse.

The Data Lakehouse

The *data lakehouse* is an architecture that combines aspects of the data warehouse and the data lake. As it is generally conceived, the lakehouse stores data in object storage just like a lake. However, the lakehouse adds to this arrangement features designed to streamline data management and create an engineering experience similar to a data warehouse. This means robust table and schema support and features for managing incremental updates and deletes. Lakehouses typically also support table history and rollback; this is accomplished by retaining old versions of files and metadata.

A lakehouse system is a metadata and file-management layer deployed with data management and transformation tools. Databricks has heavily promoted the lakehouse concept with Delta Lake, an open source storage management system. We would be remiss not to point out that the architecture of the data lakehouse is similar to the architecture used by various commercial data platforms, including BigQuery and Snowflake. These systems store data in object storage and provide automated metadata

[72] James Dixon, "Data Lakes Revisited," *James Dixon's Blog*, September 25, 2014, *https://oreil.ly/FH25v*.

management, table history, and update/delete capabilities. The complexities of managing underlying files and storage are fully hidden from the user.

The key advantage of the data lakehouse over proprietary tools is interoperability. It's much easier to exchange data between tools when stored in an open file format. Reserializing data from a proprietary database format incurs overhead in processing, time, and cost. In a data lakehouse architecture, various tools can connect to the metadata layer and read data directly from object storage.

It is important to emphasize that much of the data in a data lakehouse may not have a table structure imposed. We can impose data warehouse features where we need them in a lakehouse, leaving other data in a raw or even unstructured format.

The data lakehouse technology is evolving rapidly. A variety of new competitors to Delta Lake have emerged, including Apache Hudi and Apache Iceberg. See Appendix A for more details.

Data Platforms

Increasingly, vendors are styling their products as *data platforms*. These vendors have created their ecosystems of interoperable tools with tight integration into the core data storage layer. In evaluating platforms, engineers must ensure that the tools offered meet their needs. Tools not directly provided in the platform can still interoperate, with extra data overhead for data interchange. Platforms also emphasize close integration with object storage for unstructured use cases, as mentioned in our discussion of cloud data warehouses.

At this point, the notion of the data platform frankly has yet to be fully fleshed out. However, the race is on to create a walled garden of data tools, both simplifying the work of data engineering and generating significant vendor lock-in.

Stream-to-Batch Storage Architecture

The stream-to-batch storage architecture has many similarities to the Lambda architecture, though some might quibble over the technical details. Essentially, data flowing through a topic in the streaming storage system is written out to multiple consumers.

Some of these consumers might be real-time processing systems that generate statistics on the stream. In addition, a batch storage consumer writes data for long-term retention and batch queries. The batch consumer could be AWS Kinesis Firehose, which can generate S3 objects based on configurable triggers (e.g., time and batch size). Systems such as BigQuery ingest streaming data into a streaming buffer. This streaming buffer is automatically reserialized into columnar object storage. The query engine supports seamless querying of both the streaming buffer and the object data to provide users a current, nearly real-time view of the table.

Big Ideas and Trends in Storage

In this section, we'll discuss some big ideas in storage—key considerations that you need to keep in mind as you build out your storage architecture. Many of these considerations are part of larger trends. For example, data catalogs fit under the trend toward "enterprisey" data engineering and data management. Separation of compute from storage is now largely an accomplished fact in cloud data systems. And data sharing is an increasingly important consideration as businesses adopt data technology.

Data Catalog

A *data catalog* is a centralized metadata store for all data across an organization. Strictly speaking, a data catalog is not a top-level data storage abstraction, but it integrates with various systems and abstractions. Data catalogs typically work across operational and analytics data sources, integrate data lineage and presentation of data relationships, and allow user editing of data descriptions.

Data catalogs are often used to provide a central place where people can view their data, queries, and data storage. As a data engineer, you'll likely be responsible for setting up and maintaining the various data integrations of data pipeline and storage systems that will integrate with the data catalog and the integrity of the data catalog itself.

Catalog application integration

Ideally, data applications are designed to integrate with catalog APIs to handle their metadata and updates directly. As catalogs are more widely used in an organization, it becomes easier to approach this ideal.

Automated scanning

In practice, cataloging systems typically need to rely on an automated scanning layer that collects metadata from various systems such as data lakes, data warehouses, and operational databases. Data catalogs can collect existing metadata and may also use scanning tools to infer metadata such as key relationships or the presence of sensitive data.

Data portal and social layer

Data catalogs also typically provide a human access layer through a web interface, where users can search for data and view data relationships. Data catalogs can be enhanced with a social layer offering Wiki functionality. This allows users to provide

information on their datasets, request information from other users, and post updates as they become available.

Data catalog use cases

Data catalogs have both organizational and technical use cases. Data catalogs make metadata easily available to systems. For instance, a data catalog is a key ingredient of the data lakehouse, allowing table discoverability for queries.

Organizationally, data catalogs allow business users, analysts, data scientists, and engineers to search for data to answer questions. Data catalogs streamline crossorganizational communications and collaboration.

Data Sharing

Data sharing allows organizations and individuals to share specific data and carefully defined permissions with specific entities. Data sharing allows data scientists to share data from a sandbox with their collaborators within an organization. Across organizations, data sharing facilitates collaboration between partner businesses. For example, an ad tech company can share advertising data with its customers.

A cloud multitenant environment makes interorganizational collaboration much easier. However, it also presents new security challenges. Organizations must carefully control policies that govern who can share data with whom to prevent accidental exposure or deliberate exfiltration.

Data sharing is a core feature of many cloud data platforms. See Chapter 5 for a more extensive discussion of data sharing.

Schema

What is the expected form of the data? What is the file format? Is it structured, semistructured, or unstructured? What data types are expected? How does the data fit into a larger hierarchy? Is it connected to other data through shared keys or other relationships?

Note that schema need not be *relational*. Rather, data becomes more useful when we have as much information about its structure and organization. For images stored in a data lake, this schema information might explain the image format, resolution, and the way the images fit into a larger hierarchy.

Schema can function as a sort of Rosetta stone, instructions that tell us how to read the data. Two major schema patterns exist: schema on write and schema on read. *Schema on write* is essentially the traditional data warehouse pattern: a table has an integrated schema; any writes to the table must conform. To support schema on write, a data lake must integrate a schema metastore.

With *schema on read*, the schema is dynamically created when data is written, and a reader must determine the schema when reading the data. Ideally, schema on read is

implemented using file formats that implement built-in schema information, such as Parquet or JSON. CSV files are notorious for schema inconsistency and are not recommended in this setting.

The principal advantage of schema on write is that it enforces data standards, making data easier to consume and utilize in the future. Schema on read emphasizes flexibility, allowing virtually any data to be written. This comes at the cost of greater difficulty consuming data in the future.

Separation of Compute from Storage

A key idea we revisit throughout this book is the separation of compute from storage. This has emerged as a standard data access and query pattern in today's cloud era. Data lakes, as we discussed, store data in object stores and spin up temporary compute capacity to read and process it. Most fully managed OLAP products now rely on object storage behind the scenes. To understand the motivations for separating compute and storage, we should first look at the colocation of compute and storage.

Colocation of compute and storage

Colocation of compute and storage has long been a standard method to improve database performance. For transactional databases, data colocation allows fast, lowlatency disk reads and high bandwidth. Even when we virtualize storage (e.g., using Amazon EBS), data is located relatively close to the host machine.

The same basic idea applies for analytics query systems running across a cluster of machines. For example, with HDFS and MapReduce, the standard approach is to locate data blocks that need to be scanned in the cluster, and then push individual *map* jobs out to these blocks. The data scan and processing for the map step are strictly local. The *reduce* step involves shuffling data across the cluster, but keeping map steps local effectively preserves more bandwidth for shuffling, delivering better overall performance; map steps that filter heavily also dramatically reduce the amount of data to be shuffled.

Separation of compute and storage

If colocation of compute and storage delivers high performance, why the shift toward separation of compute and storage? Several motivations exist.

Ephemerality and scalability. In the cloud, we've seen a dramatic shift toward ephemerality. In general, it's cheaper to buy and host a server than to rent it from a cloud provider, *provided that you're running it 24 hours a day nonstop for years on end*. In practice, workloads vary dramatically, and significant efficiencies are realized

with a pay-as-you-go model if servers can scale up and down. This is true for web servers in online retail, and it is also true for big data batch jobs that may run only periodically.

Ephemeral compute resources allow engineers to spin up massive clusters to complete jobs on time and then delete clusters when these jobs are done. The performance benefits of temporarily operating at ultra-high scale can outweigh the bandwidth limitations of object storage.

Data durability and availability. Cloud object stores significantly mitigate the risk of data loss and generally provide extremely high uptime (availability). For example, S3 stores data across multiple zones; if a natural disaster destroys a zone, data is still available from the remaining zones. Having multiple zones available also reduces the odds of a data outage. If resources in one zone go down, engineers can spin up the same resources in a different zone.

The potential for a misconfiguration that destroys data in object storage is still somewhat scary, but simple-to-deploy mitigations are available. Copying data to multiple cloud regions reduces this risk since configuration changes are generally deployed to only one region at a time. Replicating data to multiple storage providers can further reduce the risk.

Hybrid separation and colocation

The practical realities of separating compute from storage are more complicated than we've implied. In practice, we constantly hybridize colocation and separation to realize the benefits of both approaches. This hybridization is typically done in two ways: multitier caching and hybrid object storage.

With *multitier caching*, we utilize object storage for long-term data retention and access but spin up local storage to be used during queries and various stages of data pipelines. Both Google and Amazon offer versions of hybrid object storage (object storage that is tightly integrated with compute).

Let's look at examples of how some popular processing engines hybridize separation and colocation of storage and compute.

Example: AWS EMR with S3 and HDFS. Big data services like Amazon EMR spin up temporary HDFS clusters to process data. Engineers have the option of referencing both S3 and HDFS as a filesystem. A common pattern is to stand up HDFS on SSD drives, pull from S3, and save data from intermediate processing steps on local HDFS. Doing so can realize significant performance gains over processing directly from S3.

Full results are written back to S3 once the cluster completes its steps, and the cluster and HDFS are deleted. Other consumers read the output data directly from S3.

Example: Apache Spark. In practice, Spark generally runs jobs on HDFS or some other ephemeral distributed filesystem to support performant storage of data between processing steps. In addition, Spark relies heavily on in-memory storage of data to improve processing. The problem with owning the infrastructure for running Spark is that dynamic RAM (DRAM) is extremely expensive; by separating compute and storage in the cloud, we can rent large quantities of memory and then release that memory when the job completes.

Example: Apache Druid. Apache Druid relies heavily on SSDs to realize high performance. Since SSDs are significantly more expensive than magnetic disks, Druid keeps only one copy of data in its cluster, reducing "live" storage costs by a factor of three.

Of course, maintaining data durability is still critical, so Druid uses an object store as its durability layer. When data is ingested, it's processed, serialized into compressed columns, and written to cluster SSDs and object storage. In the event of node failure or cluster data corruption, data can be automatically recovered to new nodes. In addition, the cluster can be shut down and then fully recovered from SSD storage.

Example: Hybrid object storage. Google's Colossus file storage system supports finegrained control of data block location, although this functionality is not exposed directly to the public. BigQuery uses this feature to colocate customer tables in a single location, allowing ultra-high bandwidth for queries in that location.[73] We refer to this as *hybrid object storage* because it combines the clean abstractions of object storage with some advantages of colocating compute and storage. Amazon also offers some notion of hybrid object storage through S3 Select, a feature that allows users to filter S3 data directly in S3 clusters before data is returned across the network. We speculate that public clouds will adopt hybrid object storage more widely to improve the performance of their offerings and make more efficient use of available network resources. Some may be already doing so without disclosing this publicly.

The concept of hybrid object storage underscores that there can still be advantages to having low-level access to hardware rather than relying on someone else's public cloud. Public cloud services do not expose low-level details of hardware and systems (e.g., data block locations for Colossus), but these details can be extremely useful in

[73] Valliappa Lakshmanan and Jordan Tigani, *Google BigQuery: The Definitive Guide* (Sebastopol, CA: O'Reilly, 2019), 16–17, 188, *https://oreil.ly/5aXXu*.

performance optimization and enhancement. See our discussion of cloud economics in Chapter 4.

While we're now seeing a mass migration of data to public clouds, we believe that many hyper-scale data service vendors that currently run on public clouds provided by other vendors may build their data centers in the future, albeit with deep network integration into public clouds.

Zero-copy cloning

Cloud-based systems based around object storage support *zero-copy cloning*. This typically means that a new virtual copy of an object is created (e.g., a new table) without necessarily physically copying the underlying data. Typically, new pointers are created to the raw data files, and future changes to these tables will not be recorded in the old table. For those familiar with the inner workings of object-oriented languages such as Python, this type of "shallow" copying is familiar from other contexts.

Zero-copy cloning is a compelling feature, but engineers must understand its strengths and limitations. For example, cloning an object in a data lake environment and then deleting the files in the original object might also wipe out the new object.

For fully managed object-store-based systems (e.g., Snowflake and BigQuery), engineers need to be extremely familiar with the exact limits of shallow copying. Engineers have more access to underlying object storage in data lake systems such as Databricks—a blessing and a curse. Data engineers should exercise great caution before deleting any raw files in the underlying object store. Databricks and other data lake management technologies sometimes also support a notion of *deep copying*, whereby all underlying data objects are copied. This is a more expensive process, but also more robust in the event that files are unintentionally lost or deleted.

Data Storage Lifecycle and Data Retention

Storing data isn't as simple as just saving it to object storage or disk and forgetting about it. You need to think about the data storage lifecycle and data retention. When you think about access frequency and use cases, ask, "How important is the data to downstream users, and how often do they need to access it?" This is the data storage lifecycle. Another question you should ask is, "How long should I keep this data?" Do you need to retain data indefinitely, or are you fine discarding it past a certain time frame? This is data retention. Let's dive into each of these.

Hot, warm, and cold data

Did you know that data has a temperature? Depending on how frequently data is accessed, we can roughly bucket the way it is stored into three categories of

persistence: hot, warm, and cold. Query access patterns differ for each dataset (Figure 6-9). Typically, newer data is queried more often than older data. Let's look at hot, cold, and warm data in that order.

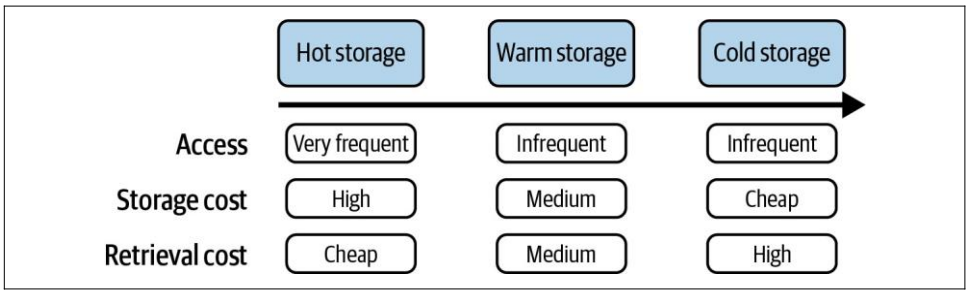


Figure 6-9. Hot, warm, and cold data costs associated with access frequency

Hot data. *Hot data* has instant or frequent access requirements. The underlying storage for hot data is suited for fast access and reads, such as SSD or memory. Because of the type of hardware involved with hot data, storing hot data is often the most expensive form of storage. Example use cases for hot data include retrieving product recommendations and product page results. The cost of storing hot data is the highest of these three storage tiers, but retrieval is often inexpensive.

Query results cache is another example of hot data. When a query is run, some query engines will persist the query results in the cache. For a limited time, when the same query is run, instead of rerunning the same query against storage, the query results cache serves the cached results. This allows for much faster query response times versus redundantly issuing the same query repeatedly. In upcoming chapters, we cover query results caches in more detail.

Warm data. *Warm data* is accessed semi-regularly, say, once per month. No hard and fast rules indicate how often warm data is accessed, but it's less than hot data and more than cold data. The major cloud providers offer object storage tiers that accommodate warm data. For example, S3 offers an Infrequently Accessed Tier, and Google Cloud has a similar storage tier called Nearline. Vendors give their models of recommended access frequency, and engineers can also do their cost modeling and monitoring. Storage of warm data is cheaper than hot data, with slightly more expensive retrieval costs.

Cold data. On the other extreme, *cold data* is infrequently accessed data. The hardware used to archive cold data is typically cheap and durable, such as HDD, tape storage, and cloud-based archival systems. Cold data is mainly meant for long-term

archival, when there's little to no intention to access the data. Though storing cold data is cheap, retrieving cold data is often expensive.

Storage tier considerations. When considering the storage tier for your data, consider the costs of each tier. If you store all of your data in hot storage, all of the data can be accessed quickly. But this comes at a tremendous price! Conversely, if you store all data in cold storage to save on costs, you'll certainly lower your storage costs, but at the expense of prolonged retrieval times and high retrieval costs if you need to access data. The storage price goes down from faster/higher performing storage to lower storage.

Cold storage is popular for archiving data. Historically, cold storage involved physical backups and often mailing this data to a third party that would archive it in a literal vault. Cold storage is increasingly popular in the cloud. Every cloud vendor offers a cold data solution, and you should weigh the cost of pushing data into cold storage versus the cost and time to retrieve the data.

Data engineers need to account for spillover from hot to warm/cold storage. Memory is expensive and finite. For example, if hot data is stored in memory, it can be spilled to disk when there's too much new data to store and not enough memory. Some databases may move infrequently accessed data to warm or cold tiers, offloading the data to either HDD or object storage. The latter is increasingly more common because of the cost-effectiveness of object storage. If you're in the cloud and using managed services, disk spillover will happen automatically.

If you're using cloud-based object storage, create automated lifecycle policies for your data. This will drastically reduce your storage costs. For example, if your data needs to be accessed only once a month, move the data to an infrequent access storage tier. If your data is 180 days old and not accessed for current queries, move it to an archival storage tier. In both cases, you can automate the migration of data away from regular object storage, and you'll save money. That said, consider the retrieval costs—both in time and money—using infrequent or archival style storage tiers. Access and retrieval times and costs may vary depending on the cloud provider. Some cloud providers make it simple and cheap to migrate data into archive storage, but it is costly and slow to retrieve your data.

Data retention

Back in the early days of "big data," there was a tendency to err on the side of accumulating every piece of data possible, regardless of its usefulness. The expectation was, "we might need this data in the future." This data hoarding inevitably became unwieldy and dirty, giving rise to data swamps and regulatory crackdowns on data retention, among other consequences and nightmares. Nowadays, data engineers need

to consider data retention: what data do you *need* to keep, and how *long* should you keep it? Here are some things to think about with data retention.

Value. Data is an asset, so you should know the value of the data you're storing. Of course, value is subjective and depends on what it's worth to your immediate use case and your broader organization. Is this data impossible to re-create, or can it easily be re-created by querying upstream systems? What's the impact to downstream users if this data is available versus if it is not?

Time. The value to downstream users also depends upon the age of the data. New data is typically more valuable and frequently accessed than older data. Technical limitations may determine how long you can store data in certain storage tiers. For example, if you store hot data in cache or memory, you'll likely need to set a time to live (TTL), so you can expire data after a certain point or persist it to warm or cold storage. Otherwise, your hot storage will become full, and queries against the hot data will suffer from performance lags.

Compliance. Certain regulations (e.g., HIPAA and Payment Card Industry, or PCI) might require you to keep data for a certain time. In these situations, the data simply needs to be accessible upon request, even if the likelihood of an access request is low. Other regulations might require you to hold data for only a limited period of time, and you'll need to have the ability to delete specific information on time and within compliance guidelines. You'll need a storage and archival data process—along with the ability to search the data—that fits the retention requirements of the particular regulation with which you need to comply. Of course, you'll want to balance compliance against cost.

Cost. Data is an asset that (hopefully) has an ROI. On the cost side of ROI, an obvious storage expense is associated with data. Consider the timeline in which you need to retain data. Given our discussion about hot, warm, and cold data, implement automatic data lifecycle management practices and move the data to cold storage if you don't need the data past the required retention date. Or delete data if it's truly not needed.

Single-Tenant Versus Multitenant Storage

In Chapter 3, we covered the trade-offs between single-tenant and multitenant architecture. To recap, with *single-tenant* architecture, each group of tenants (e.g., individual users, groups of users, accounts, or customers) gets its own dedicated set of resources such as networking, compute, and storage. A *multitenant* architecture inverts this and shares these resources among groups of users. Both architectures are widely used. This section looks at the implications of single-tenant and multitenant storage.

Adopting single-tenant storage means that every tenant gets their dedicated storage. In the example in Figure 6-10, each tenant gets a database. No data is shared among these databases, and storage is totally isolated. An example of using single-tenant storage is that each customer's data must be stored in isolation and cannot be blended with any other customer's data. In this case, each customer gets their own database.

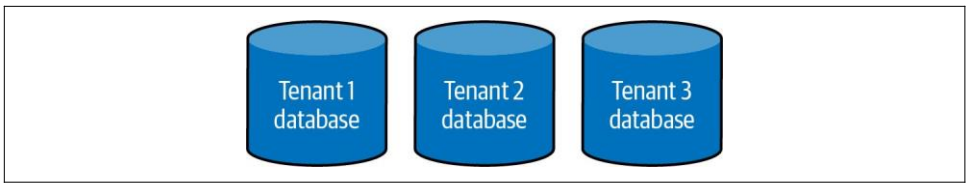


Figure 6-10. In single-tenant storage, each tenant gets their own database

Separate data storage implies separate and independent schemas, bucket structures, and everything related to storage. This means you have the liberty of designing each tenant's storage environment to be uniform or let them evolve however they may. Schema variation across customers can be an advantage and a complication; as always, consider the trade-offs. If each tenant's schema isn't uniform across all tenants, this has major consequences if you need to query multiple tenants' tables to create a unified view of all tenant data.

Multitenant storage allows for the storage of multiple tenants within a single database. For example, instead of the single-tenant scenario where customers get their own database, multiple customers may reside in the same database schemas or tables in a multitenant database. Storing multitenant data means each tenant's data is stored in the same place (Figure 6-11).

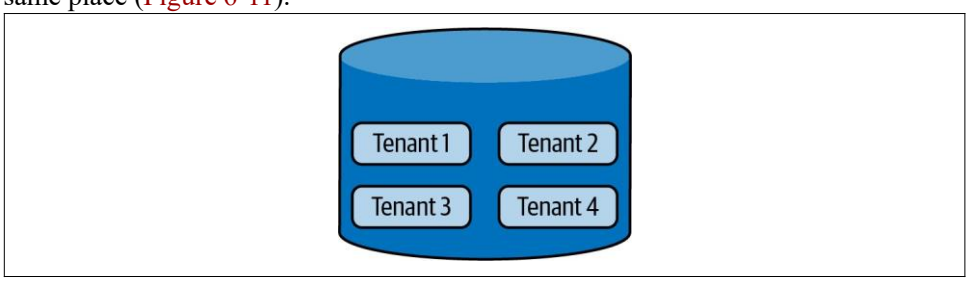


Figure 6-11. In this multitenant storage, four tenants occupy the same database

You need to be aware of querying both single and multitenant storage, which we cover in more detail in Chapter 8.

Whom You'll Work With

Storage is at the heart of data engineering infrastructure. You'll interact with the people who own your IT infrastructure—typically, DevOps, security, and cloud architects. Defining domains of responsibility between data engineering and other teams is critical. Do data engineers have the authority to deploy their infrastructure in an AWS account, or must another team handle these changes? Work with other teams to define streamlined processes so that teams can work together efficiently and quickly.

The division of responsibilities for data storage will depend significantly on the maturity of the organization involved. The data engineer will likely manage the storage systems and workflow if the company is early in its data maturity. If the company is later in its data maturity, the data engineer will probably manage a section of the storage system. This data engineer will also likely interact with engineers on either side of storage—ingestion and transformation.

The data engineer needs to ensure that the storage systems used by downstream users are securely available, contain high-quality data, have ample storage capacity, and perform when queries and transformations are run.

Undercurrents

The undercurrents for storage are significant because storage is a critical hub for all stages of the data engineering lifecycle. Unlike other undercurrents for which data might be in motion (ingestion) or queried and transformed, the undercurrents for storage differ because storage is so ubiquitous.

Security

While engineers often view security as an impediment to their work, they should embrace the idea that security is a key enabler. Robust security at rest and in motion with fine-grained data access control allows data to be shared and consumed more widely within a business. The value of data goes up significantly when this is possible.

As always, exercise the principle of least privilege. Don't give full database access to anyone unless required. This means most data engineers don't need full database access in practice. Also, pay attention to the column, row, and cell-level access controls in your database. Give users only the information they need and no more.

Data Management

Data management is critical as we read and write data with storage systems.

Data catalogs and metadata management

Data is enhanced by robust metadata. Cataloging enables data scientists, analysts, and ML engineers by enabling data discovery. Data lineage accelerates the time to track down data problems and allows consumers to locate upstream raw sources. As you build out your storage systems, invest in your metadata. Integration of a data dictionary with these other tools allows users to share and record institutional knowledge robustly.

Metadata management also significantly enhances data governance. Beyond simply enabling passive data cataloging and lineage, consider implementing analytics over these systems to get a clear, active picture of what's happening with your data.

Data versioning in object storage

Major cloud object storage systems enable data versioning. Data versioning can help with error recovery when processes fail, and data becomes corrupted. Versioning is also beneficial for tracking the history of datasets used to build models. Just as code version control allows developers to track down commits that cause bugs, data version control can aid ML engineers in tracking changes that lead to model performance degradation.

Privacy

GDPR and other privacy regulations have significantly impacted storage system design. Any data with privacy implications has a lifecycle that data engineers must manage. Data engineers must be prepared to respond to data deletion requests and selectively remove data as required. In addition, engineers can accommodate privacy and security through anonymization and masking.

DataOps

DataOps is not orthogonal to data management, and a significant area of overlap exists. DataOps concerns itself with traditional operational monitoring of storage systems and monitoring the data itself, inseparable from metadata and quality.

Systems monitoring

Data engineers must monitor storage in a variety of ways. This includes monitoring infrastructure storage components, where they exist, but also monitoring object storage and other "serverless" systems. Data engineers should take the lead on FinOps (cost management), security monitoring, and access monitoring.

Observing and monitoring data

While metadata systems as we've described are critical, good engineering must consider the entropic nature of data by actively seeking to understand its characteristics and watching for major changes. Engineers can monitor data statistics, apply anomaly detection methods or simple rules, and actively test and validate for logical inconsistencies.

Data Architecture

Chapter 3 covers the basics of data architecture, as storage is the critical underbelly of the data engineering lifecycle.

Consider the following data architecture tips. Design for required reliability and durability. Understand the upstream source systems and how that data, once ingested, will be stored and accessed. Understand the types of data models and queries that will occur downstream.

If data is expected to grow, can you negotiate storage with your cloud provider? Take an active approach to FinOps, and treat it as a central part of architecture conversations. Don't prematurely optimize, but prepare for scale if business opportunities exist in operating on large data volumes.

Lean toward fully managed systems, and understand provider SLAs. Fully managed systems are generally far more robust and scalable than systems you have to babysit.

Orchestration

Orchestration is highly entangled with storage. Storage allows data to flow through pipelines, and orchestration is the pump. Orchestration also helps engineers cope with the complexity of data systems, potentially combining many storage systems and query engines.

Software Engineering

We can think about software engineering in the context of storage in two ways. First, the code you write should perform well with your storage system. Make sure the code you write stores the data correctly and doesn't accidentally cause data, memory leaks, or performance issues. Second, define your storage infrastructure as code and use ephemeral compute resources when it's time to process your data. Because storage is increasingly distinct from compute, you can automatically spin resources up and down while keeping your data in object storage. This keeps your infrastructure clean and avoids coupling your storage and query layers.

Conclusion

Storage is everywhere and underlays many stages of the data engineering lifecycle. In this chapter, you learned about the raw ingredients, types, abstractions, and big ideas around storage systems. Gain deep knowledge of the inner workings and limitations of the storage systems you'll use. Know the types of data, activities, and workloads appropriate for your storage.

Additional Resources

- "Column-Oriented DBMS" Wikipedia page
- "The Design and Implementation of Modern Column-Oriented Database Systems" by Daniel Abadi et al.
- *Designing Data-Intensive Applications* by Martin Kleppmann (O'Reilly)
- "Diving Into Delta Lake: Schema Enforcement and Evolution" by Burak Yavuz et al.
- "Hot Data vs. Cold Data: Why It Matters" by Afzaal Ahmad Zeeshan
- IDC's "Data Creation and Replication Will Grow at a Faster Rate than Installed Storage Capacity, According to the IDC Global DataSphere and StorageSphere Forecasts" press release
- "Rowise vs. Columnar Database? Theory and in Practice" by Mangat Rai Modi
- "Snowflake Solution Anti-Patterns: The Probable Data Scientist" by John Aven
- "What Is a Vector Database?" by Bryan Turriff
- "What Is Object Storage? A Definition and Overview" by Alex Chan
- "The What, When, Why, and How of Incremental Loads" by Tim Mitchell

CHAPTER 7

Ingestion

You've learned about the various source systems you'll likely encounter as a data engineer and about ways to store data. Let's now turn our attention to the patterns and choices that apply to ingesting data from various source systems. In this chapter, we discuss data ingestion (see Figure 7-1), the key engineering considerations for the ingestion phase, the major patterns for batch and streaming ingestion, technologies you'll encounter, whom you'll work with as you develop your data ingestion pipeline, and how the undercurrents feature in the ingestion phase.

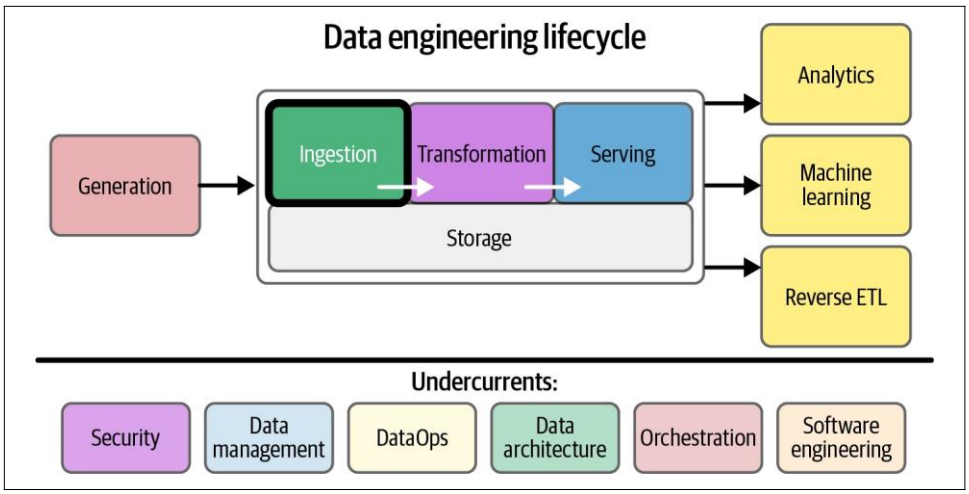


Figure 7-1. To begin processing data, we must ingest it

What Is Data Ingestion?

Data ingestion is the process of moving data from one place to another. Data ingestion implies data movement from source systems into storage in the data engineering lifecycle, with ingestion as an intermediate step (Figure 7-2).

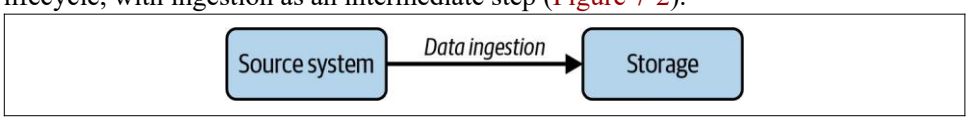


Figure 7-2. Data from system 1 is ingested into system 2

It's worth quickly contrasting data ingestion with data integration. Whereas *data ingestion* is data movement from point A to B, *data integration* combines data from disparate sources into a new dataset. For example, you can use data integration to combine data from a CRM system, advertising analytics data, and web analytics to create a user profile, which is saved to your data warehouse. Furthermore, using reverse ETL, you can send this newly created user profile *back* to your CRM so salespeople can use the data for prioritizing leads. We describe data integration more fully in Chapter 8, where we discuss data transformations; reverse ETL is covered in Chapter 9.

We also point out that data ingestion is different from *internal ingestion* within a system. Data stored in a database is copied from one table to another, or data in a stream is temporarily cached. We consider this another part of the general data transformation process covered in Chapter 8.

Data Pipelines Defined

Data pipelines begin in source systems, but ingestion is the stage where data engineers begin actively designing data pipeline activities. In the data engineering space, a good deal of ceremony occurs around data movement and processing patterns, with established patterns such as ETL, newer patterns such as ELT, and new names for long-established practices (reverse ETL) and data sharing.

All of these concepts are encompassed in the idea of a *data pipeline*. It is essential to understand the details of these various patterns and know that a modern data pipeline includes all of them. As the world moves away from a traditional monolithic approach with rigid constraints on data movement, and toward an open ecosystem of cloud services that are assembled like LEGO bricks to realize products, data engineers prioritize using the right tools to accomplish the desired outcome over adhering to a narrow philosophy of data movement.

In general, here's our definition of a data pipeline:

> A data pipeline is the combination of architecture, systems, and processes that move data through the stages of the data engineering lifecycle.

Our definition is deliberately fluid—and intentionally vague—to allow data engineers to plug in whatever they need to accomplish the task at hand. A data pipeline could be a traditional ETL system, where data is ingested from an on-premises transactional system, passed through a monolithic processor, and written into a data warehouse. Or it could be a cloud-based data pipeline that pulls data from 100 sources, combines it into 20 wide tables, trains five other ML models, deploys them into production, and monitors ongoing performance. A data pipeline should be flexible enough to fit any needs along the data engineering lifecycle.

Let's keep this notion of data pipelines in mind as we proceed through this chapter.

Key Engineering Considerations for the Ingestion Phase

When preparing to architect or build an ingestion system, here are some primary considerations and questions to ask yourself related to data ingestion:

- What's the use case for the data I'm ingesting?
- Can I reuse this data and avoid ingesting multiple versions of the same dataset?
- Where is the data going? What's the destination?
- How often should the data be updated from the source?
- What is the expected data volume?

- What format is the data in? Can downstream storage and transformation accept this format?
- Is the source data in good shape for immediate downstream use? That is, is the data of good quality? What post-processing is required to serve it? What are data-quality risks (e.g., could bot traffic to a website contaminate the data)?
- Does the data require in-flight processing for downstream ingestion if the data is from a streaming source?

These questions undercut batch and streaming ingestion and apply to the underlying architecture you'll create, build, and maintain. Regardless of how often the data is ingested, you'll want to consider these factors when designing your ingestion architecture:

- Bounded versus unbounded
- Frequency
- Synchronous versus asynchronous
- Serialization and deserialization
- Throughput and scalability
- Reliability and durability
- Payload
- Push versus pull versus poll patterns Let's look at each of these.

Bounded Versus Unbounded Data

As you might recall from Chapter 3, data comes in two forms: bounded and unbounded (Figure 7-3). *Unbounded data* is data as it exists in reality, as events happen, either sporadically or continuously, ongoing and flowing. *Bounded data* is a convenient way of bucketing data across some sort of boundary, such as time.

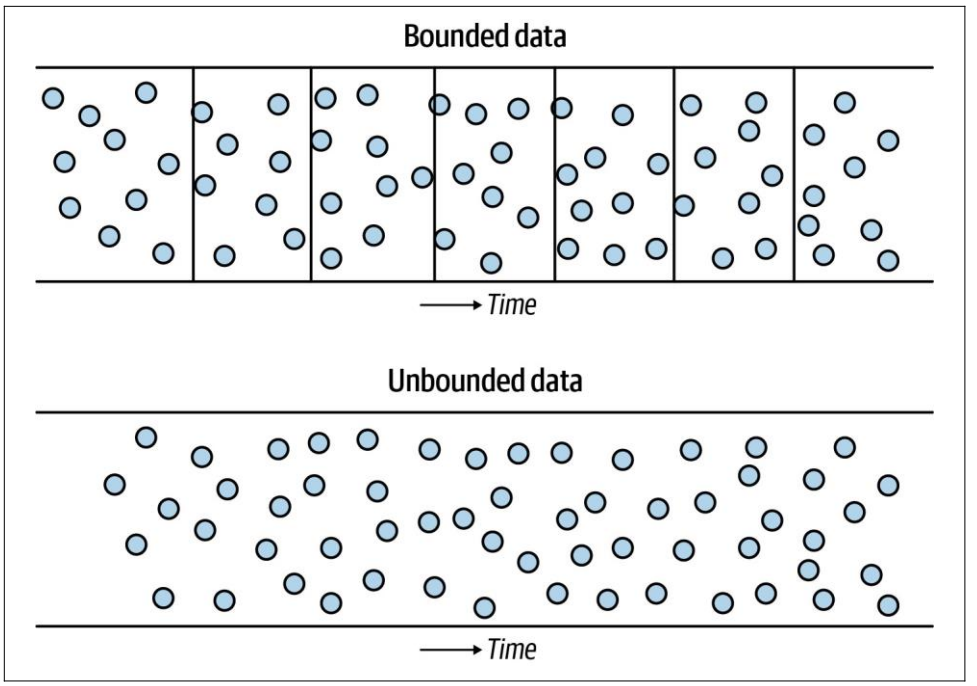


Figure 7-3. Bounded versus unbounded data

Let us adopt this mantra: *All data is unbounded until it's bounded.* Like many mantras, this one is not precisely accurate 100% of the time. The grocery list that I scribbled this afternoon is bounded data. I wrote it as a stream of consciousness (unbounded data) onto a piece of scrap paper, where the thoughts now exist as a list of things (bounded data) I need to buy at the grocery store. However, the idea is correct for practical purposes for the vast majority of data you'll handle in a business context. For example, an online retailer will process customer transactions 24 hours a day until the business fails, the economy grinds to a halt, or the sun explodes.

Business processes have long imposed artificial bounds on data by cutting discrete batches. Keep in mind the true unboundedness of your data; streaming ingestion systems are simply a tool for preserving the unbounded nature of data so that subsequent steps in the lifecycle can also process it continuously.

Frequency

One of the critical decisions that data engineers must make in designing dataingestion processes is the data-ingestion frequency. Ingestion processes can be batch, micro-batch, or real-time.

Ingestion frequencies vary dramatically from slow to fast (Figure 7-4). On the slow end, a business might ship its tax data to an accounting firm once a year. On the faster side, a CDC system could retrieve new log updates from a source database once a minute. Even faster, a system might continuously ingest events from IoT sensors and process these within seconds. Data-ingestion frequencies are often mixed in a company, depending on the use case and technologies.

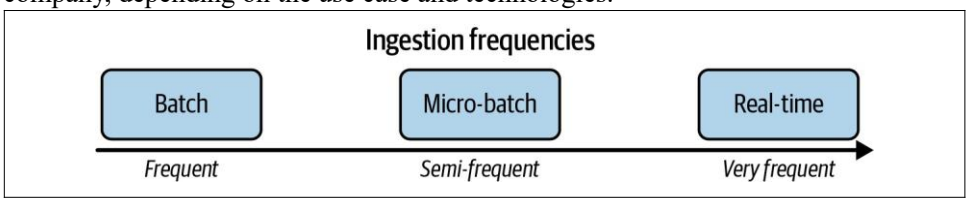


Figure 7-4. The spectrum batch to real-time ingestion frequencies

We note that "real-time" ingestion patterns are becoming increasingly common. We put "real-time" in quotation marks because no ingestion system is genuinely real-time. Any database, queue or pipeline has inherent latency in delivering data to a target system. It is more accurate to speak of *near real-time*, but we often use *real-time* for brevity. The near real-time pattern generally does away with an explicit update frequency; events are processed in the pipeline either one by one as they arrive or in micro-batches (i.e., batches over concise time intervals). For this book, we will use *real-time* and *streaming* interchangeably.

Even with a streaming data-ingestion process, batch processing downstream is relatively standard. At the time of this writing, ML models are typically trained on a batch basis, although continuous online training is becoming more prevalent. Rarely do data engineers have the option to build a purely near real-time pipeline with no batch components. Instead, they choose where batch boundaries will occur—i.e., the data engineering lifecycle data will be broken into batches. Once data reaches a batch process, the batch frequency becomes a bottleneck for all downstream processing.

In addition, streaming systems are the best fit for many data source types. In IoT applications, the typical pattern is for each sensor to write events or measurements to streaming systems as they happen. While this data can be written directly into a database, a streaming ingestion platform such as Amazon Kinesis or Apache Kafka is a better fit for the application. Software applications can adopt similar patterns by writing events to a message queue as they happen rather than waiting for an extraction process to pull events and state information from a backend database. This pattern works exceptionally well for event-driven architectures already exchanging messages through queues. And again, streaming architectures generally coexist with batch processing.

Synchronous Versus Asynchronous Ingestion

With *synchronous ingestion*, the source, ingestion, and destination have complex dependencies and are tightly coupled. As you can see in Figure 7-5, each stage of the data engineering lifecycle has processes A, B, and C directly dependent upon one another. If process A fails, processes B and C cannot start; if process B fails, process C doesn't start. This type of synchronous workflow is common in older ETL systems, where data extracted from a source system must then be transformed before being loaded into a data warehouse. Processes downstream of ingestion can't start until all data in the batch has been ingested. If the ingestion or transformation process fails, the entire process must be rerun.

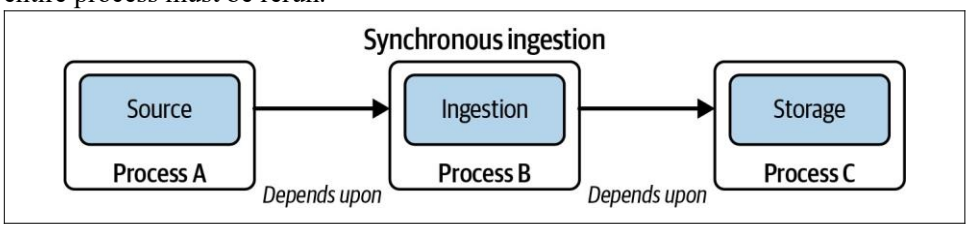


Figure 7-5. A synchronous ingestion process runs as discrete batch steps

Here's a mini case study of how *not* to design your data pipelines. At one company, the transformation process itself was a series of dozens of tightly coupled synchronous workflows, with the entire process taking over 24 hours to finish. If any step of that transformation pipeline failed, the whole transformation process had to be restarted from the beginning! In this instance, we saw process after process fail, and because of nonexistent or cryptic error messages, fixing the pipeline was a game of whack-a-mole that took over a week to diagnose and cure. Meanwhile, the business didn't have updated reports during that time. People weren't happy.

With *asynchronous ingestion*, dependencies can now operate at the level of individual events, much as they would in a software backend built from microservices (Figure 7-6). Individual events become available in storage as soon as they are ingested individually. Take the example of a web application on AWS that emits events into Amazon Kinesis Data Streams (here acting as a buffer). The stream is read by Apache Beam, which parses and enriches events, and then forwards them to a second Kinesis stream; Kinesis Data Firehose rolls up events and writes objects to Amazon S3.

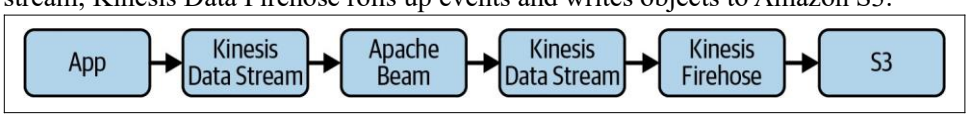


Figure 7-6. Asynchronous processing of an event stream in AWS

The big idea is that rather than relying on asynchronous processing, where a batch process runs for each stage as the input batch closes and certain time conditions are met, each stage of the asynchronous pipeline can process data items as they become available in parallel across the Beam cluster. The processing rate depends on available resources. The Kinesis Data Stream acts as the shock absorber, moderating the load so that event rate spikes will not overwhelm downstream processing. Events will move through the pipeline quickly when the event rate is low, and any backlog has cleared. Note that we could modify the scenario and use a Kinesis Data Stream for storage, eventually extracting events to S3 before they expire out of the stream.

Serialization and Deserialization

Moving data from source to destination involves serialization and deserialization. As a reminder, *serialization* means encoding the data from a source and preparing data structures for transmission and intermediate storage stages.

When ingesting data, ensure that your destination can deserialize the data it receives. We've seen data ingested from a source but then sitting inert and unusable in the destination because the data cannot be properly deserialized. See the more extensive discussion of serialization in Appendix A.

Throughput and Scalability

In theory, your ingestion should never be a bottleneck. In practice, ingestion bottlenecks are pretty standard. Data throughput and system scalability become critical as your data volumes grow and requirements change. Design your systems to scale and shrink to flexibly match the desired data throughput.

Where you're ingesting data from matters a lot. If you're receiving data as it's generated, will the upstream system have any issues that might impact your downstream ingestion pipelines? For example, suppose a source database goes down. When it comes back online and attempts to backfill the lapsed data loads, will your ingestion be able to keep up with this sudden influx of backlogged data?

Another thing to consider is your ability to handle bursty data ingestion. Data generation rarely happens at a constant rate and often ebbs and flows. Built-in buffering is required to collect events during rate spikes to prevent data from getting lost. Buffering bridges the time while the system scales and allows storage systems to accommodate bursts even in a dynamically scalable system.

Whenever possible, use managed services that handle the throughput scaling for you. While you can manually accomplish these tasks by adding more servers, shards, or workers, often this isn't value-added work, and there's a good chance you'll miss something. Much of this heavy lifting is now automated. Don't reinvent the data ingestion wheel if you don't have to.

Reliability and Durability

Reliability and durability are vital in the ingestion stages of data pipelines. *Reliability* entails high uptime and proper failover for ingestion systems. *Durability* entails making sure that data isn't lost or corrupted.

Some data sources (e.g., IoT devices and caches) may not retain data if it is not correctly ingested. Once lost, it is gone for good. In this sense, the *reliability* of ingestion systems leads directly to the *durability* of generated data. If data is ingested, downstream processes can theoretically run late if they break temporarily.

Our advice is to evaluate the risks and build an appropriate level of redundancy and self-healing based on the impact and cost of losing data. Reliability and durability have both direct and indirect costs. For example, will your ingestion process continue if an AWS zone goes down? How about a whole region? How about the power grid or the internet? Of course, nothing is free. How much will this cost you? You might be able to build a highly redundant system and have a team on call 24 hours a day to handle outages. This also means your cloud and labor costs become prohibitive (direct costs), and the ongoing work takes a significant toll on your team (indirect costs). There's no single correct answer, and you need to evaluate the costs and benefits of your reliability and durability decisions.

Don't assume that you can build a system that will reliably and durably ingest data in every possible scenario. Even the nearly infinite budget of the US federal government can't guarantee this. In many extreme scenarios, ingesting data actually won't matter. There will be little to ingest if the internet goes down, even if you build multiple airgapped data centers in underground bunkers with independent power. Continually evaluate the trade-offs and costs of reliability and durability.

Payload

A *payload* is the dataset you're ingesting and has characteristics such as kind, shape, size, schema and data types, and metadata. Let's look at some of these characteristics to understand why this matters.

Kind

The *kind* of data you handle directly impacts how it's dealt with downstream in the data engineering lifecycle. Kind consists of type and format. Data has a type—tabular, image, video, text, etc. The type directly influences the data format or the way it is expressed in bytes, names, and file extensions. For example, a tabular kind of data may be in formats such as CSV or Parquet, with each of these formats having different byte patterns for serialization and deserialization. Another kind of data is an image, which has a format of JPG or PNG and is inherently unstructured.

Shape

Every payload has a *shape* that describes its dimensions. Data shape is critical across the data engineering lifecycle. For instance, an image's pixel and red, green, blue (RGB) dimensions are necessary for training deep learning models. As another example, if you're trying to import a CSV file into a database table, and your CSV has more columns than the database table, you'll likely get an error during the import process. Here are some examples of the shapes of various kinds of data:

Tabular
: The number of rows and columns in the dataset, commonly expressed as *M* rows and *N* columns

Semistructured JSON
: The key-value pairs and nesting depth occur with subelements

Unstructured text
: Number of words, characters, or bytes in the text body

Images
: The width, height, and RGB color depth (e.g., 8 bits per pixel)

Uncompressed audio
: Number of channels (e.g., two for stereo), sample depth (e.g., 16 bits per sample), sample rate (e.g., 48 kHz), and length (e.g., 10,003 seconds)

Size

The *size* of the data describes the number of bytes of a payload. A payload may range in size from single bytes to terabytes and larger. To reduce the size of a payload, it may be compressed into various formats such as ZIP and TAR (see the discussion of compression in Appendix A).

A massive payload can also be split into chunks, which effectively reduces the size of the payload into smaller subsections. When loading a huge file into a cloud object storage or data warehouse, this is a common practice as the small individual files are easier to transmit over a network (especially if they're compressed). The smaller chunked files are sent to their destination and then reassembled after all data has arrived.

Schema and data types

Many data payloads have a schema, such as tabular and semistructured data. As mentioned earlier in this book, a schema describes the fields and types of data within those fields. Other data, such as unstructured text, images, and audio, will not have an

explicit schema or data types. However, they might come with technical file descriptions on shape, data and file format, encoding, size, etc.

Although you can connect to databases in various ways (such as file export, CDC, JDBC/ODBC), the connection is easy. The great engineering challenge is understanding the underlying schema. Applications organize data in various ways, and engineers need to be intimately familiar with the organization of the data and relevant update patterns to make sense of it. The problem has been somewhat exacerbated by the popularity of object-relational mapping (ORM), which automatically generates schemas based on object structure in languages such as Java or Python. Natural structures in an object-oriented language often map to something messy in an operational database. Data engineers may need to familiarize themselves with the class structure of application code.

Schema is not only for databases. As we've discussed, APIs present their schema complications. Many vendor APIs have friendly reporting methods that prepare data for analytics. In other cases, engineers are not so lucky. The API is a thin wrapper around underlying systems, requiring engineers to understand application internals to use the data.

Much of the work associated with ingesting from source schemas happens in the data engineering lifecycle transformation stage, which we discuss in Chapter 8. We've placed this discussion here because data engineers need to begin studying source schemas as soon they plan to ingest data from a new source.

Communication is critical for understanding source data, and engineers also have the opportunity to reverse the flow of communication and help software engineers improve data where it is produced. Later in this chapter, we'll return to this topic in "Whom You'll Work With" on page 262.

Detecting and handling upstream and downstream schema changes. Changes in schema frequently occur in source systems and are often well out of data engineers' control. Examples of schema changes include the following:

- Adding a new column
- Changing a column type
- Creating a new table
- Renaming a column

It's becoming increasingly common for ingestion tools to automate the detection of schema changes and even auto-update target tables. Ultimately, this is something of a mixed blessing. Schema changes can still break pipelines downstream of staging and ingestion.

Engineers must still implement strategies to respond to changes automatically and alert on changes that cannot be accommodated automatically. Automation is excellent, but the analysts and data scientists who rely on this data should be informed of the schema changes that violate existing assumptions. Even if automation can accommodate a change, the new schema may adversely affect the performance of reports and models. Communication between those making schema changes and those impacted by these changes is as important as reliable automation that checks for schema changes.

Schema registries. In streaming data, every message has a schema, and these schemas may evolve between producers and consumers. A *schema registry* is a metadata repository used to maintain schema and data type integrity in the face of constantly changing schemas. Schema registries can also track schema versions and history. It describes the data model for messages, allowing consistent serialization and deserialization between producers and consumers. Schema registries are used in most major data tools and clouds.

Metadata

In addition to the apparent characteristics we've just covered, a payload often contains metadata, which we first discussed in Chapter 2. Metadata is data about data. Metadata can be as critical as the data itself. One of the significant limitations of the early approach to the data lake—or data swamp, which could become a data superfund site—was a complete lack of attention to metadata. Without a detailed description of the data, it may be of little value. We've already discussed some types of metadata (e.g., schema) and will address them many times throughout this chapter.

Push Versus Pull Versus Poll Patterns

We mentioned push versus pull when we introduced the data engineering lifecycle in Chapter 2. A *push* strategy (Figure 7-7) involves a source system sending data to a target, while a *pull* strategy (Figure 7-8) entails a target reading data directly from a source. As we mentioned in that discussion, the lines between these strategies are blurry.

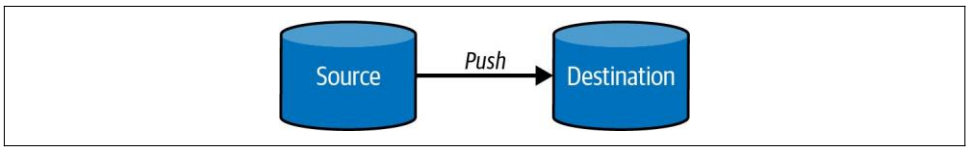


Figure 7-7. Pushing data from source to destination

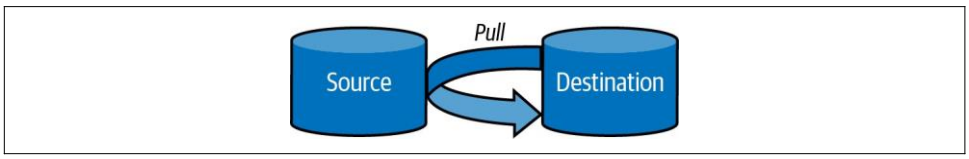


Figure 7-8. A destination pulling data from a source

Another pattern related to pulling is *polling* for data (Figure 7-9). Polling involves periodically checking a data source for any changes. When changes are detected, the destination pulls the data as it would in a regular pull situation.

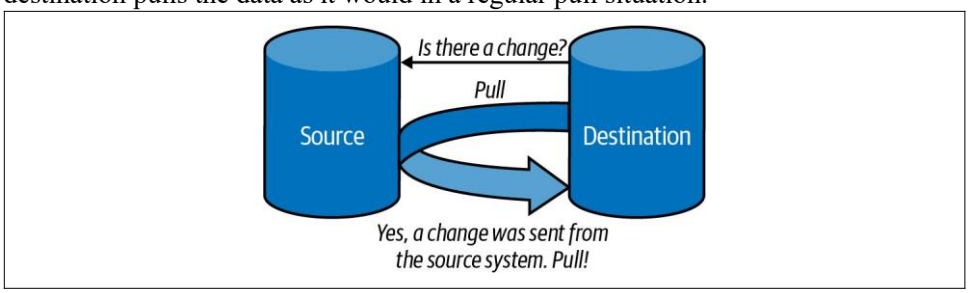


Figure 7-9. Polling for changes in a source system

Batch Ingestion Considerations

Batch ingestion, which involves processing data in bulk, is often a convenient way to ingest data. This means that data is ingested by taking a subset of data from a source system, based either on a time interval or the size of accumulated data (Figure 7-10).

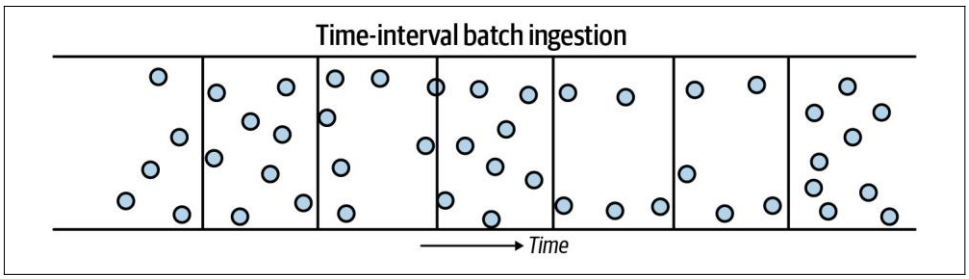


Figure 7-10. Time-interval batch ingestion

Time-interval batch ingestion is widespread in traditional business ETL for data warehousing. This pattern is often used to process data once a day, overnight during off-hours, to provide daily reporting, but other frequencies can also be used. *Size-based batch ingestion* (Figure 7-11) is quite common when data is moved from a streaming-based system into object storage; ultimately, you must cut the data into discrete blocks for future processing in a data lake. Some size-based ingestion systems can break data into objects based on various criteria, such as the size in bytes of the total number of events.

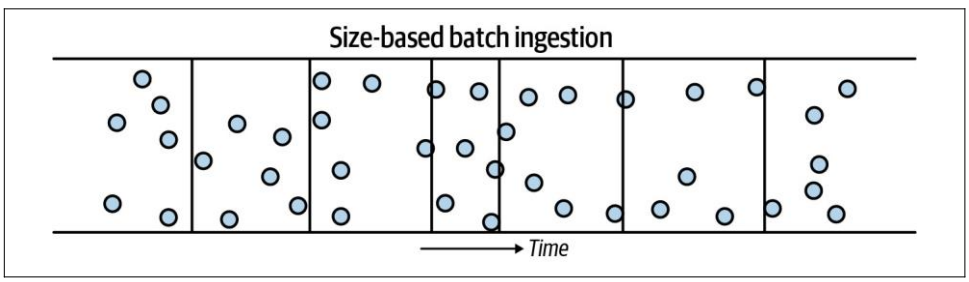


Figure 7-11. Size-based batch ingestion

Some commonly used batch ingestion patterns, which we discuss in this section, include the following:

- Snapshot or differential extraction
- File-based export and ingestion
- ETL versus ELT
- Inserts, updates, and batch size
- Data migration

Snapshot or Differential Extraction

Data engineers must choose whether to capture full snapshots of a source system or differential (sometimes called *incremental*) updates. With *full snapshots*, engineers grab the entire current state of the source system on each update read. With the *differential update* pattern, engineers can pull only the updates and changes since the last read from the source system. While differential updates are ideal for minimizing network traffic and target storage usage, full snapshot reads remain extremely common because of their simplicity.

File-Based Export and Ingestion

Data is quite often moved between databases and systems using files. Data is serialized into files in an exchangeable format, and these files are provided to an ingestion system. We consider file-based export to be a *push-based* ingestion pattern. This is because data export and preparation work is done on the source system side.

File-based ingestion has several potential advantages over a direct database connection approach. It is often undesirable to allow direct access to backend systems for security reasons. With file-based ingestion, export processes are run on the data-source side, giving source system engineers complete control over what data gets exported and how the data is preprocessed. Once files are done, they can be provided to the target system in various ways. Common file-exchange methods are object storage, secure file transfer protocol (SFTP), electronic data interchange (EDI),
or secure copy (SCP).

ETL Versus ELT

Chapter 3 introduced ETL and ELT, both extremely common ingestion, storage, and transformation patterns you'll encounter in batch workloads. The following are brief definitions of the extract and load parts of ETL and ELT:

Extract
: This means getting data from a source system. While *extract* seems to imply *pulling* data, it can also be push based. Extraction may also require reading metadata and schema changes.

Load
: Once data is extracted, it can either be transformed (ETL) before loading it into a storage destination or simply loaded into storage for future transformation. When loading data, you should be mindful of the type of system you're loading, the schema of the data, and the performance impact of loading.

We cover ETL and ELT in greater detail in Chapter 8.

Inserts, Updates, and Batch Size

Batch-oriented systems often perform poorly when users attempt to perform many small-batch operations rather than a smaller number of large operations. For example, while it is common to insert one row at a time in a transactional database, this is a bad pattern for many columnar databases, as it forces the creation of many small, suboptimal files and forces the system to run a high number of *create object* operations. Running many small in-place update operations is an even bigger problem because it causes the database to scan each existing column file to run the update.

Understand the appropriate update patterns for the database or data store you're working with. Also, understand that certain technologies are purpose-built for high insert rates. For example, Apache Druid and Apache Pinot can handle high insert rates. SingleStore can manage hybrid workloads that combine OLAP and OLTP characteristics. BigQuery performs poorly on a high rate of vanilla SQL single-row inserts but extremely well if data is fed in through its stream buffer. Know the limits and characteristics of your tools.

Data Migration

Migrating data to a new database or environment is not usually trivial, and data needs to be moved in bulk. Sometimes this means moving data sizes that are hundreds of terabytes or much larger, often involving the migration of specific tables and moving entire databases and systems.

Data migrations probably aren't a regular occurrence as a data engineer, but you should be familiar with them. As is often the case for data ingestion, schema management is a crucial consideration. Suppose you're migrating data from one database system to a different one (say, SQL Server to Snowflake). No matter how closely the two databases resemble each other, subtle differences almost always exist in the way they handle schema. Fortunately, it is generally easy to test ingestion of a sample of data and find schema issues before undertaking a complete table migration.

Most data systems perform best when data is moved in bulk rather than as individual rows or events. File or object storage is often an excellent intermediate stage for transferring data. Also, one of the biggest challenges of database migration is not the movement of the data itself but the movement of data pipeline connections from the old system to the new one.

Be aware that many tools are available to automate various types of data migrations. Especially for large and complex migrations, we suggest looking at these options before doing this manually or writing your own migration solution.

Message and Stream Ingestion Considerations

Ingesting event data is common. This section covers issues you should consider when ingesting events, drawing on topics covered in Chapters 5 and 6.

Schema Evolution

Schema evolution is common when handling event data; fields may be added or removed, or value types might change (say, a string to an integer). Schema evolution can have unintended impacts on your data pipelines and destinations. For example, an IoT device gets a firmware update that adds a new field to the event it transmits, or a third-party API introduces changes to its event payload or countless other scenarios. All of these potentially impact your downstream capabilities.

To alleviate issues related to schema evolution, here are a few suggestions. First, if your event-processing framework has a schema registry (discussed earlier in this chapter), use it to version your schema changes. Next, a dead-letter queue (described in "Error Handling and Dead-Letter Queues" on page 249) can help you investigate issues with events that are not properly handled. Finally, the low-fidelity route (and the most effective) is regularly communicating with upstream stakeholders about potential schema changes and proactively addressing schema changes with the teams introducing these changes instead of reacting to the receiving end of breaking changes.

Late-Arriving Data

Though you probably prefer all event data to arrive on time, event data might arrive late. A group of events might occur around the same time frame (similar event times), but some might arrive later than others (late ingestion times) because of various circumstances.

For example, an IoT device might be late sending a message because of internet latency issues. This is common when ingesting data. You should be aware of latearriving data and the impact on downstream systems and uses. Suppose you assume that ingestion or process time is the same as the event time. You may get some strange results if your reports or analysis depend on an accurate portrayal of when events occur. To handle late-arriving data, you need to set a cutoff time for when late-arriving data will no longer be processed.

Ordering and Multiple Delivery

Streaming platforms are generally built out of distributed systems, which can cause some complications. Specifically, messages may be delivered out of order and more than once (at-least-once delivery). See the event-streaming platforms discussion in Chapter 5 for more details.

Replay

Replay allows readers to request a range of messages from the history, allowing you to rewind your event history to a particular point in time. Replay is a key capability in many streaming ingestion platforms and is particularly useful when you need to reingest and reprocess data for a specific time range. For example, RabbitMQ typically deletes messages after all subscribers consume them. Kafka, Kinesis, and Pub/Sub all support event retention and replay.

Time to Live

How long will you preserve your event record? A key parameter is *maximum message retention time*, also known as the *time to live* (TTL). TTL is usually a configuration you'll set for how long you want events to live before they are acknowledged and ingested. Any unacknowledged event that's not ingested after its TTL expires automatically disappears. This is helpful to reduce backpressure and unnecessary event volume in your event-ingestion pipeline.

Find the right balance of TTL impact on our data pipeline. An extremely short TTL (milliseconds or seconds) might cause most messages to disappear before processing. A very long TTL (several weeks or months) will create a backlog of many unprocessed messages, resulting in long wait times.

Let's look at how some popular platforms handle TTL at the time of this writing. Google Cloud Pub/Sub supports retention periods of up to 7 days. Amazon Kinesis Data Streams retention can be turned up to 365 days. Kafka can be configured for indefinite retention, limited by available disk space. (Kafka also supports the option to write older messages to cloud object storage, unlocking virtually unlimited storage space and retention.)

Message Size

Message size is an easily overlooked issue: you must ensure that the streaming framework in question can handle the maximum expected message size. Amazon Kinesis supports a maximum message size of 1 MB. Kafka defaults to this maximum size but can be configured for a maximum of 20 MB or more. (Configurability may vary on managed service platforms.)

Error Handling and Dead-Letter Queues

Sometimes events aren't successfully ingested. Perhaps an event is sent to a nonexistent topic or message queue, the message size may be too large, or the event has expired past its TTL. Events that cannot be ingested need to be rerouted and stored in a separate location called a *dead-letter queue*.

A dead-letter queue segregates problematic events from events that can be accepted by the consumer (Figure 7-12). If events are not rerouted to a dead-letter queue, these erroneous events risk blocking other messages from being ingested. Data engineers can use a dead-letter queue to diagnose why event ingestions errors occur and solve data pipeline problems, and might be able to reprocess some messages in the queue after fixing the underlying cause of errors.

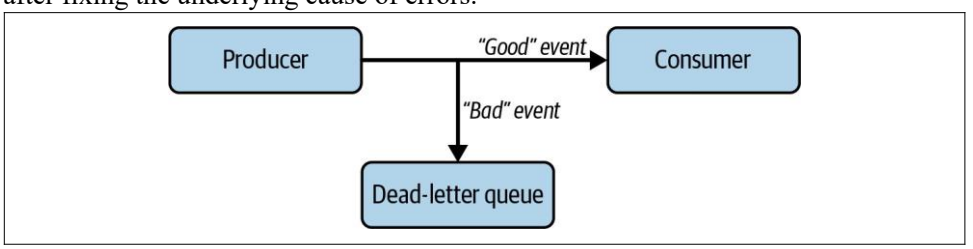


Figure 7-12. "Good" events are passed to the consumer, whereas "bad" events are stored in a dead-letter queue

Consumer Pull and Push

A consumer subscribing to a topic can get events in two ways: push and pull. Let's look at the ways some streaming technologies pull and push data. Kafka and Kinesis support only pull subscriptions. Subscribers read messages from a topic and confirm when they have been processed. In addition to pull subscriptions, Pub/Sub and RabbitMQ support push subscriptions, allowing these services to write messages to a listener.

Pull subscriptions are the default choice for most data engineering applications, but you may want to consider push capabilities for specialized applications. Note that pull-only message ingestion systems can still push if you add an extra layer to handle this.

Location

It is often desirable to integrate streaming across several locations for enhanced redundancy and to consume data close to where it is generated. As a general rule, the

closer your ingestion is to where data originates, the better your bandwidth and latency. However, you need to balance this against the costs of moving data between regions to run analytics on a combined dataset. As always, data egress costs can spiral quickly. Do a careful evaluation of the trade-offs as you build out your architecture.

Ways to Ingest Data

Now that we've described some of the significant patterns underlying batch and streaming ingestion, let's focus on ways you can ingest data. Although we will cite

some common ways, keep in mind that the universe of data ingestion practices and technologies is vast and growing daily.

Direct Database Connection

Data can be pulled from databases for ingestion by querying and reading over a network connection. Most commonly, this connection is made using ODBC or JDBC.

ODBC uses a driver hosted by a client accessing the database to translate commands issued to the standard ODBC API into commands issued to the database. The database returns query results over the wire, where the driver receives them and translates them back into a standard form to be read by the client. For ingestion, the application utilizing the ODBC driver is an ingestion tool. The ingestion tool may pull data through many small queries or a single large query.

JDBC is conceptually remarkably similar to ODBC. A Java driver connects to a remote database and serves as a translation layer between the standard JDBC API and the native network interface of the target database. It might seem strange to have a database API dedicated to a single programming language, but there are strong motivations for this. The Java Virtual Machine (JVM) is standard, portable across hardware architectures and operating systems, and provides the performance of compiled code through a just-in-time (JIT) compiler. The JVM is an extremely popular compiling VM for running code in a portable manner.

JDBC provides extraordinary database driver portability. ODBC drivers are shipped as OS and architecture native binaries; database vendors must maintain versions for each architecture/OS version that they wish to support. On the other hand, vendors can ship a single JDBC driver that is compatible with any JVM language (e.g., Java, Scala, Clojure, or Kotlin) and JVM data framework (i.e., Spark.) JDBC has become so popular that it is also used as an interface for non-JVM languages such as Python; the Python ecosystem provides translation tools that allow Python code to talk to a JDBC driver running on a local JVM.

JDBC and ODBC are used extensively for data ingestion from relational databases, returning to the general concept of direct database connections. Various enhancements are used to accelerate data ingestion. Many data frameworks can parallelize several simultaneous connections and partition queries to pull data in parallel. On the other hand, nothing is free; using parallel connections also increases the load on the source database.

JDBC and ODBC were long the gold standards for data ingestion from databases, but these connection standards are beginning to show their age for many data engineering applications. These connection standards struggle with nested data, and they send data

as rows. This means that native nested data must be reencoded as string data to be sent over the wire, and columns from columnar databases must be reserialized as rows.

As discussed in "File-Based Export and Ingestion" on page 246, many databases now support native file export that bypasses JDBC/ODBC and exports data directly in formats such as Parquet, ORC, and Avro. Alternatively, many cloud data warehouses provide direct REST APIs.

JDBC connections should generally be integrated with other ingestion technologies. For example, we commonly use a reader process to connect to a database with JDBC, write the extracted data into multiple objects, and then orchestrate ingestion into a downstream system (see Figure 7-13). The reader process can run in a wholly ephemeral cloud instance or in an orchestration system.

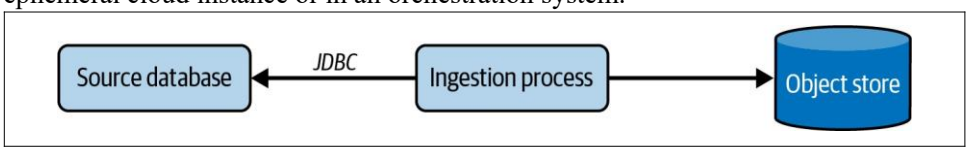


Figure 7-13. An ingestion process reads from a source database using JDBC, and then writes objects into object storage. A target database (not shown) can be triggered to ingest the data with an API call from an orchestration system.

Change Data Capture

Change data capture (CDC), introduced in Chapter 2, is the process of ingesting changes from a source database system. For example, we might have a source PostgreSQL system that supports an application and periodically or continuously ingests table changes for analytics.

Note that our discussion here is by no means exhaustive. We introduce you to common patterns but suggest that you read the documentation on a particular database to handle the details of CDC strategies.

Batch-oriented CDC

If the database table in question has an `updated_at` field containing the last time a record was written or updated, we can query the table to find all updated rows since a specified time. We set the filter timestamp based on when we last captured changed rows from the tables. This process allows us to pull changes and differentially update a target table.

This form of batch-oriented CDC has a key limitation: while we can easily determine which rows have changed since a point in time, we don't necessarily obtain all changes that were applied to these rows. Consider the example of running batch CDC on a bank account table every 24 hours. This operational table shows the current account balance

for each account. When money is moved in and out of accounts, the banking application runs a transaction to update the balance.

When we run a query to return all rows in the account table that changed in the last 24 hours, we'll see records for each account that recorded a transaction. Suppose that a certain customer withdrew money five times using a debit card in the last 24 hours. Our query will return only the last account balance recorded in the 24 hour period; other records over the period won't appear. This issue can be mitigated by utilizing an insert-only schema, where each account transaction is recorded as a new record in the table (see "Insert-Only" on page 162).

Continuous CDC

Continuous CDC captures all table history and can support near real-time data ingestion, either for real-time database replication or to feed real-time streaming analytics. Rather than running periodic queries to get a batch of table changes, continuous CDC treats each write to the database as an event.

We can capture an event stream for continuous CDC in a couple of ways. One of the most common approaches with a transactional database such as PostgreSQL is *log-based CDC*. The database binary log records every change to the database sequentially (see "Database Logs" on page 161). A CDC tool can read this log and send the events to a target, such as the Apache Kafka Debezium streaming platform.

Some databases support a simplified, managed CDC paradigm. For instance, many cloud-hosted databases can be configured to directly trigger a serverless function or write to an event stream every time a change happens in the database. This completely frees engineers from worrying about the details of how events are captured in the database and forwarded.

CDC and database replication

CDC can be used to replicate between databases: events are buffered into a stream and *asynchronously* written into a second database. However, many databases natively support a tightly coupled version of replication (synchronous replication) that keeps the replica fully in sync with the primary database. Synchronous replication typically requires that the primary database and the replica are of the same type (e.g., PostgreSQL to PostgreSQL). The advantage of synchronous replication is that the secondary database can offload work from the primary database by acting as a read replica; read queries can be redirected to the replica. The query will return the same results that would be returned from the primary database.

Read replicas are often used in batch data ingestion patterns to allow large scans to run without overloading the primary production database. In addition, an application can

be configured to fail over to the replica if the primary database becomes unavailable. No data will be lost in the failover because the replica is entirely in sync with the primary database.

The advantage of asynchronous CDC replication is a loosely coupled architecture pattern. While the replica might be slightly delayed from the primary database, this is often not a problem for analytics applications, and events can now be directed to a variety of targets; we might run CDC replication while simultaneously directing events to object storage and a streaming analytics processor.

CDC considerations

Like anything in technology, CDC is not free. CDC consumes various database resources, such as memory, disk bandwidth, storage, CPU time, and network bandwidth. Engineers should work with production teams and run tests before turning on CDC on production systems to avoid operational problems. Similar considerations apply to synchronous replication.

For batch CDC, be aware that running any large batch query against a transactional production system can cause excessive load. Either run such queries only at off-hours or use a read replica to avoid burdening the primary database.

APIs

> The bulk of software engineering is just plumbing.
>
> —Karl Hughes[74]

As we mentioned in Chapter 5, APIs are a data source that continues to grow in importance and popularity. A typical organization may have hundreds of external data sources such as SaaS platforms or partner companies. The hard reality is that no proper standard exists for data exchange over APIs. Data engineers can spend a significant amount of time reading documentation, communicating with external data owners, and writing and maintaining API connection code.

Three trends are slowly changing this situation. First, many vendors provide API client libraries for various programming languages that remove much of the complexity of API access.

Second, numerous data connector platforms are available now as SaaS, open source, or managed open source. These platforms provide turnkey data connectivity to many

[74] Karl Hughes, "The Bulk of Software Engineering Is Just Plumbing," Karl Hughes website, July 8, 2018, *https://oreil.ly/uIuqJ*.

data sources; they offer frameworks for writing custom connectors for unsupported data sources. See "Managed Data Connectors" on page 256.

The third trend is the emergence of data sharing (discussed in Chapter 5)—i.e., the ability to exchange data through a standard platform such as BigQuery, Snowflake, Redshift, or S3. Once data lands on one of these platforms, it is straightforward to store it, process it, or move it somewhere else. Data sharing has had a large and rapid impact in the data engineering space.

Don't reinvent the wheel when data sharing is not an option and direct API access is necessary. While a managed service might look like an expensive option, consider the value of your time and the opportunity cost of building API connectors when you could be spending your time on higher-value work.

In addition, many managed services now support building custom API connectors. This may provide API technical specifications in a standard format or writing connector code that runs in a serverless function framework (e.g., AWS Lambda) while letting the managed service handle the details of scheduling and synchronization. Again, these services can be a huge time-saver for engineers, both for development and ongoing maintenance.

Reserve your custom connection work for APIs that aren't well supported by existing frameworks; you will find that there are still plenty of these to work on. Handling custom API connections has two main aspects: software development and ops. Follow software development best practices; you should use version control, continuous delivery, and automated testing. In addition to following DevOps best practices, consider an orchestration framework, which can dramatically streamline the operational burden of data ingestion.

Message Queues and Event-Streaming Platforms

Message queues and event-streaming platforms are widespread ways to ingest realtime data from web and mobile applications, IoT sensors, and smart devices. As real-time data becomes more ubiquitous, you'll often find yourself either introducing or retrofitting ways to handle real-time data in your ingestion workflows. As such, it's essential to know how to ingest real-time data. Popular real-time data ingestion includes message queues or event-streaming platforms, which we covered in Chapter 5. Though these are both source systems, they also act as ways to ingest data. In both cases, you consume events from the publisher you subscribe to.

Recall the differences between messages and streams. A *message* is handled at the individual event level and is meant to be transient. Once a message is consumed, it is acknowledged and removed from the queue. On the other hand, a *stream* ingests events into an ordered log. The log persists for as long as you wish, allowing events to be

queried over various ranges, aggregated, and combined with other streams to create new transformations published to downstream consumers. In Figure 7-14, we have two producers (producers 1 and 2) sending events to two consumers (consumers 1 and 2). These events are combined into a new dataset and sent to a producer for downstream consumption.

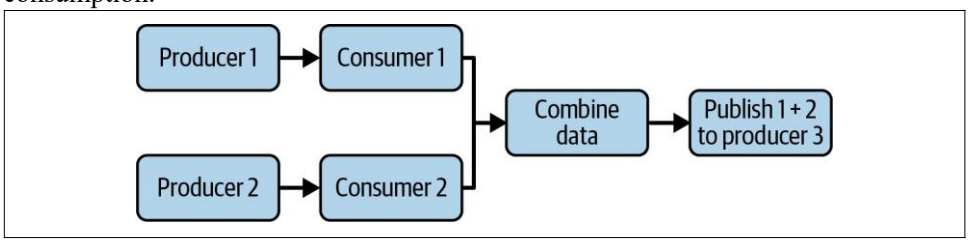


Figure 7-14. Two datasets are produced and consumed (producers 1 and 2) and then combined, with the combined data published to a new producer (producer 3)

The last point is an essential difference between batch and streaming ingestion. Whereas batch usually involves static workflows (ingest data, store it, transform it, and serve it), messages and streams are fluid. Ingestion can be nonlinear, with data being published, consumed, republished, and reconsumed. When designing your real-time ingestion workflows, keep in mind how data will flow.

Another consideration is the throughput of your real-time data pipelines. Messages and events should flow with as little latency as possible, meaning you should provision adequate partition (or shard) bandwidth and throughput. Provide sufficient memory, disk, and CPU resources for event processing, and if you're managing your real-time pipelines, incorporate autoscaling to handle spikes and save money as load decreases. For these reasons, managing your streaming platform can entail significant overhead. Consider managed services for your real-time ingestion pipelines, and focus your attention on ways to get value from your real-time data.

Managed Data Connectors

These days, if you're considering writing a data ingestion connector to a database or API, ask yourself: has this already been created? Furthermore, is there a service that will manage the nitty-gritty details of this connection for me? "APIs" on page 254 mentions the popularity of managed data connector platforms and frameworks. These tools aim to provide a standard set of connectors available out of the box to spare data engineers building complicated plumbing to connect to a particular source. Instead of creating and managing a data connector, you outsource this service to a third party.

Generally, options in the space allow users to set a target and source, ingest in various ways (e.g., CDC, replication, truncate and reload), set permissions and credentials,

configure an update frequency, and begin syncing data. The vendor or cloud behind the scenes fully manages and monitors data syncs. If data synchronization fails, you'll receive an alert with logged information on the cause of the error.

We suggest using managed connector platforms instead of creating and managing your connectors. Vendors and OSS projects each typically have hundreds of prebuilt connector options and can easily create custom connectors. The creation and management of data connectors is largely undifferentiated heavy lifting these days and should be outsourced whenever possible.

Moving Data with Object Storage

Object storage is a multitenant system in public clouds, and it supports storing massive amounts of data. This makes object storage ideal for moving data in and out of data lakes, between teams, and transferring data between organizations. You can even provide short-term access to an object with a signed URL, giving a user temporary permission.

In our view, object storage is the most optimal and secure way to handle file exchange. Public cloud storage implements the latest security standards, has a robust track record of scalability and reliability, accepts files of arbitrary types and sizes, and provides high-performance data movement. We discussed object storage much more extensively in Chapter 6.

EDI

Another practical reality for data engineers is *electronic data interchange* (EDI). The term is vague enough to refer to any data movement method. It usually refers to somewhat archaic means of file exchange, such as by email or flash drive. Data engineers will find that some data sources do not support more modern means of data transport, often because of archaic IT systems or human process limitations.

Engineers can at least enhance EDI through automation. For example, they can set up a cloud-based email server that saves files onto company object storage as soon as they are received. This can trigger orchestration processes to ingest and process data. This is much more robust than an employee downloading the attached file and manually uploading it to an internal system, which we still frequently see.

Databases and File Export

Engineers should be aware of how the source database systems handle file export. Export involves large data scans that significantly load the database for many transactional systems. Source system engineers must assess when these scans can be run without affecting application performance and might opt for a strategy to mitigate the load. Export queries can be broken into smaller exports by querying over key

ranges or one partition at a time. Alternatively, a read replica can reduce load. Read replicas are especially appropriate if exports happen many times a day and coincide with a high source system load.

Major cloud data warehouses are highly optimized for direct file export. For example, Snowflake, BigQuery, Redshift, and others support direct export to object storage in various formats.

Practical Issues with Common File Formats

Engineers should also be aware of the file formats to export. CSV is still ubiquitous and highly error prone at the time of this writing. Namely, CSV's default delimiter is also one of the most familiar characters in the English language—the comma! But it gets worse.

CSV is by no means a uniform format. Engineers must stipulate the delimiter, quote characters, and escaping to appropriately handle the export of string data. CSV also doesn't natively encode schema information or directly support nested structures. CSV file encoding and schema information must be configured in the target system to ensure appropriate ingestion. Autodetection is a convenience feature provided in many cloud environments but is inappropriate for production ingestion. As a best practice, engineers should record CSV encoding and schema details in file metadata.

More robust and expressive export formats include Parquet, Avro, Arrow, and ORC or JSON. These formats natively encode schema information and handle arbitrary string data with no particular intervention. Many of them also handle nested data structures natively so that JSON fields are stored using internal nested structures rather than simple strings. For columnar databases, columnar formats (Parquet, Arrow, ORC) allow more efficient data export because columns can be directly transcoded between formats. These formats are also generally more optimized for query engines. The Arrow file format is designed to map data directly into processing engine memory, providing high performance in data lake environments.

The disadvantage of these newer formats is that many of them are not natively supported by source systems. Data engineers are often forced to work with CSV data and then build robust exception handling and error detection to ensure data quality on ingestion. See Appendix A for a more extensive discussion of file formats.

Shell

The *shell* is an interface by which you may execute commands to ingest data. The shell can be used to script workflows for virtually any software tool, and shell scripting is still used extensively in ingestion processes. A shell script might read data from a database, reserialize it into a different file format, upload it to object storage, and

trigger an ingestion process in a target database. While storing data on a single instance or server is not highly scalable, many of our data sources are not particularly large, and such approaches work just fine.

In addition, cloud vendors generally provide robust CLI-based tools. It is possible to run complex ingestion processes simply by issuing commands to the AWS CLI. As ingestion processes grow more complicated and the SLA grows more stringent, engineers should consider moving to a proper orchestration system.

SSH

SSH is not an ingestion strategy but a protocol used with other ingestion strategies. We use SSH in a few ways. First, SSH can be used for file transfer with SCP, as mentioned earlier. Second, SSH tunnels are used to allow secure, isolated connections to databases.

Application databases should never be directly exposed on the internet. Instead, engineers can set up a bastion host—i.e., an intermediate host instance that can connect to the database in question. This host machine is exposed on the internet, although locked down for minimal access from only specified IP addresses to specified ports. To connect to the database, a remote machine first opens an SSH tunnel connection to the bastion host and then connects from the host machine to the database.

SFTP and SCP

Accessing and sending data both from secure FTP (SFTP) and secure copy (SCP) are techniques you should be familiar with, even if data engineers do not typically use these regularly (IT or security/secOps will handle this).

Engineers rightfully cringe at the mention of SFTP (occasionally, we even hear instances of FTP being used in production). Regardless, SFTP is still a practical reality for many businesses. They work with partner businesses that consume or provide data using SFTP and are unwilling to rely on other standards. To avoid data leaks, security analysis is critical in these situations.

SCP is a file-exchange protocol that runs over an SSH connection. SCP can be a secure file-transfer option if it is configured correctly. Again, adding additional network access control (defense in depth) to enhance SCP security is highly recommended.

Webhooks

Webhooks, as we discussed in Chapter 5, are often referred to as *reverse APIs*. For a typical REST data API, the data provider gives engineers API specifications that they use to write their data ingestion code. The code makes requests and receives data in responses.

With a webhook (Figure 7-15), the data provider defines an API request specification, but the data provider *makes API calls* rather than receiving them; it's the data consumer's responsibility to provide an API endpoint for the provider to call. The consumer is responsible for ingesting each request and handling data aggregation, storage, and processing.

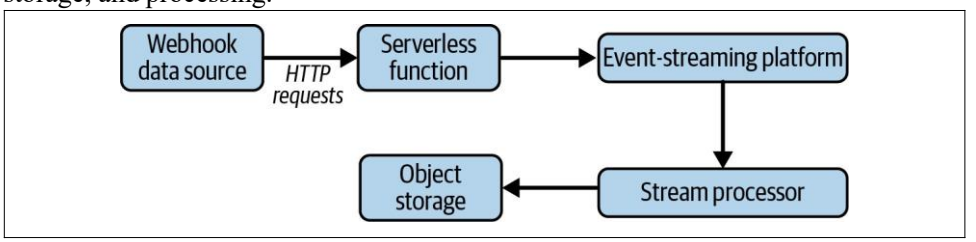


Figure 7-15. A basic webhook ingestion architecture built from cloud services

Webhook-based data ingestion architectures can be brittle, difficult to maintain, and inefficient. Using appropriate off-the-shelf tools, data engineers can build more robust webhook architectures with lower maintenance and infrastructure costs. For example, a webhook pattern in AWS might use a serverless function framework (Lambda) to receive incoming events, a managed event-streaming platform to store and buffer messages (Kinesis), a stream-processing framework to handle real-time analytics (Flink), and an object store for long-term storage (S3).

You'll notice that this architecture does much more than simply ingest the data. This underscores ingestion's entanglement with the other stages of the data engineering lifecycle; it is often impossible to define your ingestion architecture without making decisions about storage and processing.

Web Interface

Web interfaces for data access remain a practical reality for data engineers. We frequently run into situations where not all data and functionality in a SaaS platform is exposed through automated interfaces such as APIs and file drops. Instead, someone must manually access a web interface, generate a report, and download a file to a local machine. This has obvious drawbacks, such as people forgetting to run the report or having their laptop die. Where possible, choose tools and workflows that allow for automated access to data.

Web Scraping

Web scraping automatically extracts data from web pages, often by combing the web page's various HTML elements. You might scrape ecommerce sites to extract product pricing information or scrape multiple news sites for your news aggregator. Web

scraping is widespread, and you may encounter it as a data engineer. It's also a murky area where ethical and legal lines are blurry.

Here is some top-level advice to be aware of before undertaking any web-scraping project. First, ask yourself if you should be web scraping or if data is available from a third party. If your decision is to web scrape, be a good citizen. Don't inadvertently create a denial-of-service (DoS) attack, and don't get your IP address blocked. Understand how much traffic you generate and pace your web-crawling activities appropriately. Just because you can spin up thousands of simultaneous Lambda functions to scrape doesn't mean you should; excessive web scraping could lead to the disabling of your AWS account.

Second, be aware of the legal implications of your activities. Again, generating DoS attacks can entail legal consequences. Actions that violate terms of service may cause headaches for your employer or you personally.

Third, web pages constantly change their HTML element structure, making it tricky to keep your web scraper updated. Ask yourself, is the headache of maintaining these systems worth the effort?

Web scraping has interesting implications for the data engineering lifecycle processing stage; engineers should think about various factors at the beginning of a webscraping project. What do you intend to do with the data? Are you just pulling required fields from the scraped HTML by using Python code and then writing these values to a database? Do you intend to maintain the complete HTML code of the scraped websites and process this data using a framework like Spark? These decisions may lead to very different architectures downstream of ingestion.

Transfer Appliances for Data Migration

For massive quantities of data (100 TB or more), transferring data directly over the internet may be a slow and costly process. At this scale, the fastest, most efficient way to move data is not over the wire but by truck. Cloud vendors offer the ability to send your data via a physical "box of hard drives." Simply order a storage device, called a *transfer appliance*, load your data from your servers, and then send it back to the cloud vendor, which will upload your data.

The suggestion is to consider using a transfer appliance if your data size hovers around 100 TB. On the extreme end, AWS even offers Snowmobile, a transfer appliance sent to you in a semitrailer! Snowmobile is intended to lift and shift an entire data center, in which data sizes are in the petabytes or greater.

Transfer appliances are handy for creating hybrid-cloud or multicloud setups. For example, Amazon's data transfer appliance (AWS Snowball) supports import and export. To migrate into a second cloud, users can export their data into a Snowball

device and then import it into a second transfer appliance to move data into GCP or Azure. This might sound awkward, but even when it's feasible to push data over the internet between clouds, data egress fees make this a costly proposition. Physical transfer appliances are a cheaper alternative when the data volumes are significant.

Remember that transfer appliances and data migration services are one-time data ingestion events and are not suggested for ongoing workloads. Suppose you have workloads requiring constant data movement in either a hybrid or multicloud scenario. In that case, your data sizes are presumably batching or streaming much smaller data sizes on an ongoing basis.

Data Sharing

Data sharing is growing as a popular option for consuming data (see Chapters 5 and 6). Data providers will offer datasets to third-party subscribers, either for free or at a cost. These datasets are often shared in a read-only fashion, meaning you can integrate these datasets with your own data (and other third-party datasets), but you do not own the shared dataset. In the strict sense, this isn't ingestion, where you get physical possession of the dataset. If the data provider decides to remove your access to a dataset, you'll no longer have access to it.

Many cloud platforms offer data sharing, allowing you to share your data and consume data from various providers. Some of these platforms also provide data marketplaces where companies and organizations can offer their data for sale.

Whom You'll Work With

Data ingestion sits at several organizational boundaries. In developing and managing data ingestion pipelines, data engineers will work with both people and systems sitting upstream (data producers) and downstream (data consumers).

Upstream Stakeholders

A significant disconnect often exists between those responsible for *generating data* — typically, software engineers—and the data engineers who will prepare this data for analytics and data science. Software engineers and data engineers usually sit in separate organizational silos; if they think about data engineers, they typically see them simply as downstream consumers of the data exhaust from their application, not as stakeholders.

We see this current state of affairs as a problem and a significant opportunity. Data engineers can improve the quality of their data by inviting software engineers to be stakeholders in data engineering outcomes. The vast majority of software engineers