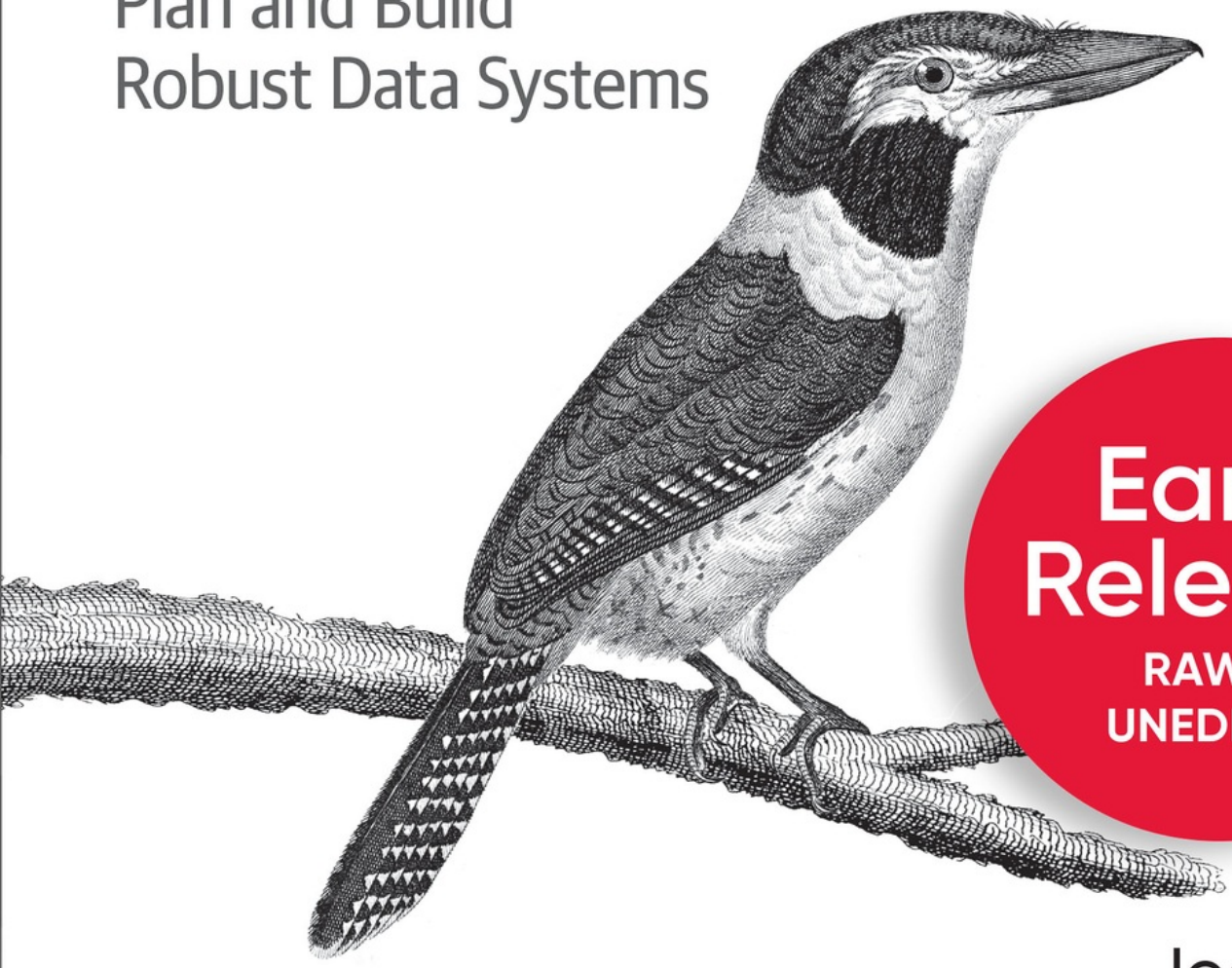
O'REILLY®

Fundamentals of Data Engineering

Plan and Build Robust Data Systems



Early Release

RAW & UNEDITED

Joe Reis &
Matt Housley

Fundamentals of Data Engineering

Plan and Build Robust Data Systems

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Joe Reis and Matt Housley

Fundamentals of Data Engineering

by Joe Reis and Matt Housley

Copyright © 2022 Joseph Reis and Matthew Housley. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jessica Haberman

Development Editor: Nicole Tache

Production Editor: Gregory Hyman

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

September 2022: First Edition

Revision History for the Early Release

- 2021-10-08: First Release
- 2021-11-12: Second Release
- 2022-03-02: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098108304 for release details.

The O’Reilly logo is a registered trademark of O’Reilly Media, Inc. *Fundamentals of Data Engineering*, the cover image, and related trade dress are trademarks of O’Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-10830-4

Chapter 1. Data Engineering Described

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the first chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at book_feedback@ternarydata.com.

Since this book is called *Fundamentals of Data Engineering*, it's critical that we clearly define what we mean by data engineering and data engineer. Terms matter, and there's a lot of confusion about what data engineering means, and what data engineers do. Let's look at the landscape of how data engineering is described, and develop some nomenclature we can use throughout this book.

What Is Data Engineering?

So what is data engineering? This is a great question. Though data engineering has existed in some form since companies have done things with data—such as data analysis and reports—it came into sharp focus alongside the rise of data science in the 2010s. But what is data engineering, exactly? Let's start with some real talk—there are endless definitions of

data engineering. A Google exact-match search for “what is data engineering?” returns over 91,000 unique results.

Let’s look at a few examples to illustrate the different ways data engineering is defined:

> *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.*[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. 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.*[2]

> *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.*[3]

> *Data engineering is all about the movement, manipulation, and management of data.*[4]

Wow! That’s only a handful of definitions, and you can see there’s a lot of variety. Clearly, there is not yet a consensus around what data engineering means. If you’ve been wondering about the term “data engineering”, your confusion is understandable.

This book provides a snapshot of data engineering today. To the fullest extent, we're focusing on the "immutables" of data engineering—what hasn't changed, and what won't likely change in the future. There are no guarantees, as this field is rapidly evolving. That said, we think we've captured a great framework to guide you on your data engineering journey for many years to come.

Before we define data engineering, you should understand a brief history of data engineering, how it's evolved, and where it fits into the general backdrop of data and technology.

Evolution of the Data Engineer

To understand data engineering today and tomorrow, it helps to have the context of how the field evolved. This is not a history book, but looking to the past is invaluable in understanding where we are today, and where things are going. There's a common theme that recurs constantly in data engineering: 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 1990. After the relational database and SQL were developed at IBM, 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 properly model their business logic in the data warehouse, Ralph Kimball and Bill Inmon developed their respective data modeling techniques and approaches, which are still widely used today.

Data warehousing ushered in the first age of scalable analytics, with new MPP systems (massively parallel processing databases) 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 was a precursor to today's data engineering, and still play a central role in the discipline.

Around the mid to late 1990s, the Internet went mainstream, creating a whole new generation of web-first companies such as AOL, Altavista, Yahoo, Amazon. The dot-com boom spawned a ton of activity in web applications, as well as the backend systems to support them—servers, databases, storage. Much of the infrastructure was expensive, monolithic, and heavily licensed. The vendors selling these backend systems 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; the dot-com bust of the late 90s left behind a small cluster of survivors. Some of these companies, such as Yahoo, Google, 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, new solutions and approaches were needed to handle the growing volume, variety, and velocity of data in a cost-effective, scalable, reliable, and fault-tolerant manner.

Coinciding with the explosion of data, commodity hardware—servers, ram, disks, flash drives, etc.—also became cheap and ubiquitous. To handle the explosion of data, several innovations allowed distributed computation and storage on massive computing clusters. Without realizing the future implications, these innovations started decentralizing and breaking apart traditionally monolithic services. This opened an era of "big data", which Oxford Dictionary defines as "extremely large data sets that may be analyzed computationally to reveal patterns, trends, and associations, especially relating to human behavior and interactions."[5] Another popular and succinct description of big data are the 3 V's of data—velocity, variety, and volume.

In 2003, Google published a paper on the *Google File System* and shortly thereafter 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.

The Google papers inspired engineers at Yahoo to develop and later open source Hadoop in 2006.[6] The impact of Hadoop cannot be overstated. Software engineers interested in large scale data problems were drawn to the possibilities of Hadoop and its ecosystem of technologies. 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[7] had to keep up with their own exploding data needs, and created elastic computing environments (Amazon EC2), infinitely scalable storage systems (Amazon S3), highly scalable NoSQL databases (DynamoDB), streaming pipelines (Kinesis) and many other core data building blocks. Amazon elected to offer these services for both internal and external consumption through *Amazon Web Services (AWS)*, which would go on to become 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. Turns out, AWS became a very profitable driver for Amazon's growth! Other public clouds would soon follow, such as Google Cloud, Microsoft Azure, OVH, Digital Ocean. The public cloud is arguably one of the biggest 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 upon which today's data ecosystem is built. The modern data landscape—and data engineering as we know it today—would not exist without these innovations.

The 2000s and 2010s: Big Data Engineering

Open source big data tools in the Hadoop ecosystem quickly matured and spread from Silicon Valley companies to tech-savvy companies all over the

world. For the first time, businesses anywhere in the world could use the same bleeding edge big data tools developed by the top tech companies. You could choose the latest and greatest—Hadoop, Pig, Hive, Dremel, HBase, Cassandra, Spark, Presto, and numerous other new technologies that came on the scene. Traditional enterprise-oriented and GUI-based data tools suddenly felt outmoded, and code was in vogue with the ascendance of MapReduce. 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 (Hadoop, YARN, HDFS, Map Reduce, etc)—big data engineers had to be proficient in software development and low-level infrastructure hacking, but with a shifted focus. Big data engineers were interested in maintaining huge 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. They also had the responsibility of working with data scientists to write low-level map-reduce jobs in a variety of languages, but especially in Java.

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, peaking around 2014. Big data captured the imagination of companies trying to make sense of the ever-growing volumes of data, as well as the endless barrage of shameless marketing from companies selling big data tools and services. Due to 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. As 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."[8]

To get an idea of the rise and fall of big data, see Figure 1-1 for a snapshot of Google Trends for the search term 'big data'.

Google Trends | Explore

big data
Search term

+ Compare

United States ▼ | 2004 - present ▼ | All categories ▼ | Web Search ▼

Interest over time

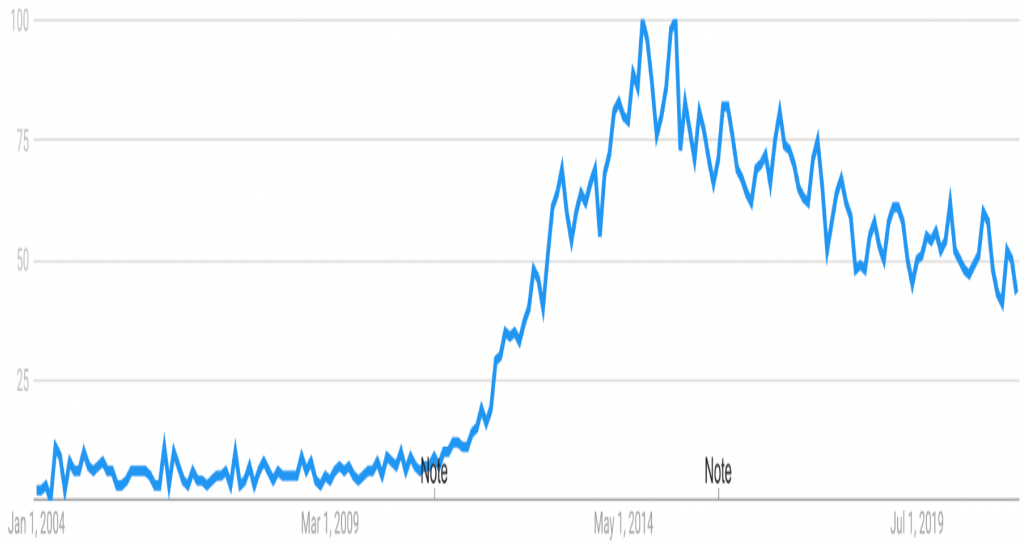


Figure 1-1. Google Trends for "big data"

Despite the popularity of the term, big data has lost steam since the mid 2010's. 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 a lot of attention. Oftentimes, companies employed entire teams of "big data engineers"—costing millions of dollars a year—to babysit these platforms. In addition, this new breed of data engineer had not been taught to work closely with business stakeholders, so it was a challenge to deliver insight the business needed.

Open-source developers, clouds, and 3rd parties started looking for ways to abstract, simplify, and make big data available to all, without the high administrative overhead and cost of managing their own clusters, installing, configuring, and upgrading their own open-source code, etc. The term "big data" is essentially a relic to describe a certain time and approach to handling large amounts of data. In truth, data is moving at a faster rate 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 Life Cycle

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 spent their time tending to the low-level details of monolithic frameworks such as Hadoop, Spark, or Informatica, the trend is moving toward the use of decentralized, modularized, managed, and highly abstracted tools. Indeed, data tools have proliferated at an astonishing rate (see Figure 1-2). New popular trends in the early 2020's include "The Modern Data Stack", which represents a collection of off-the-shelf open source and 3rd party products assembled to make the lives of analysts 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, of interconnecting numerous technologies like lego bricks to serve ultimate business goals.

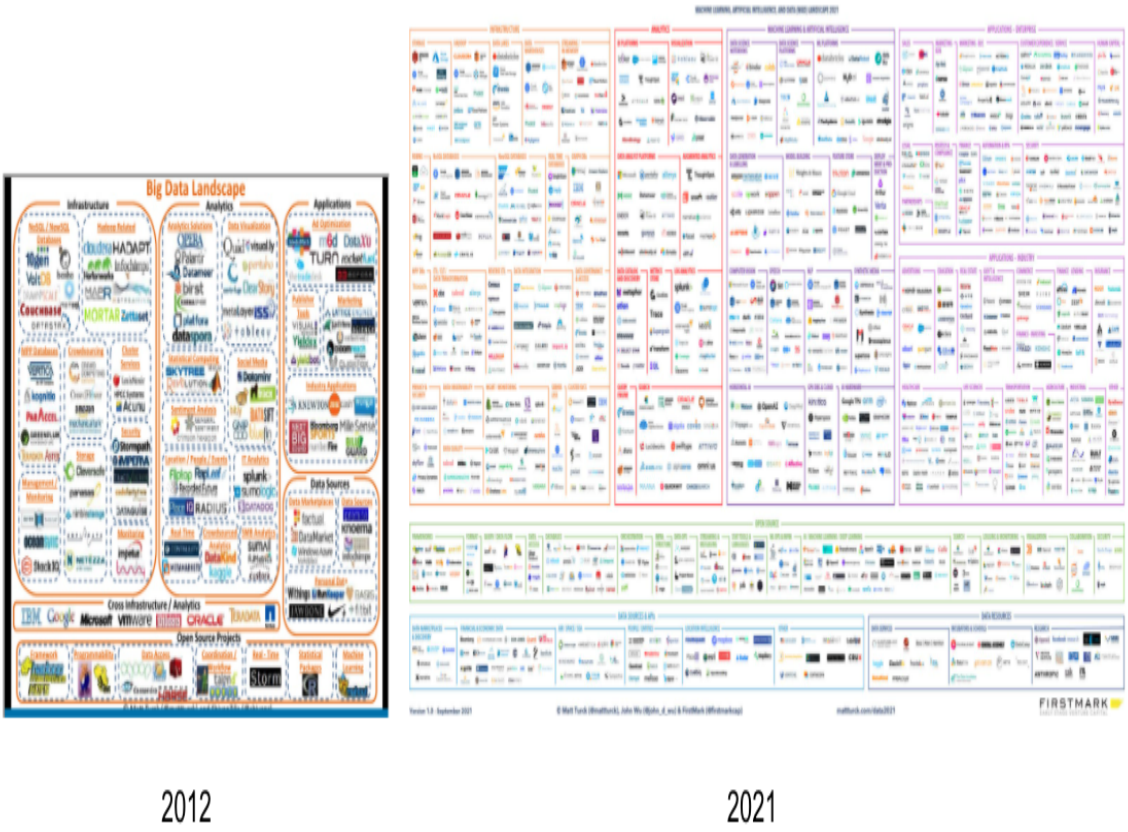


Figure 1-2. Data tools in 2012 vs 2021

In fact, the authors believe that data engineers should focus on the abstraction of data processing stages in what we term the *data engineering lifecycle* (see Figure 1-3).

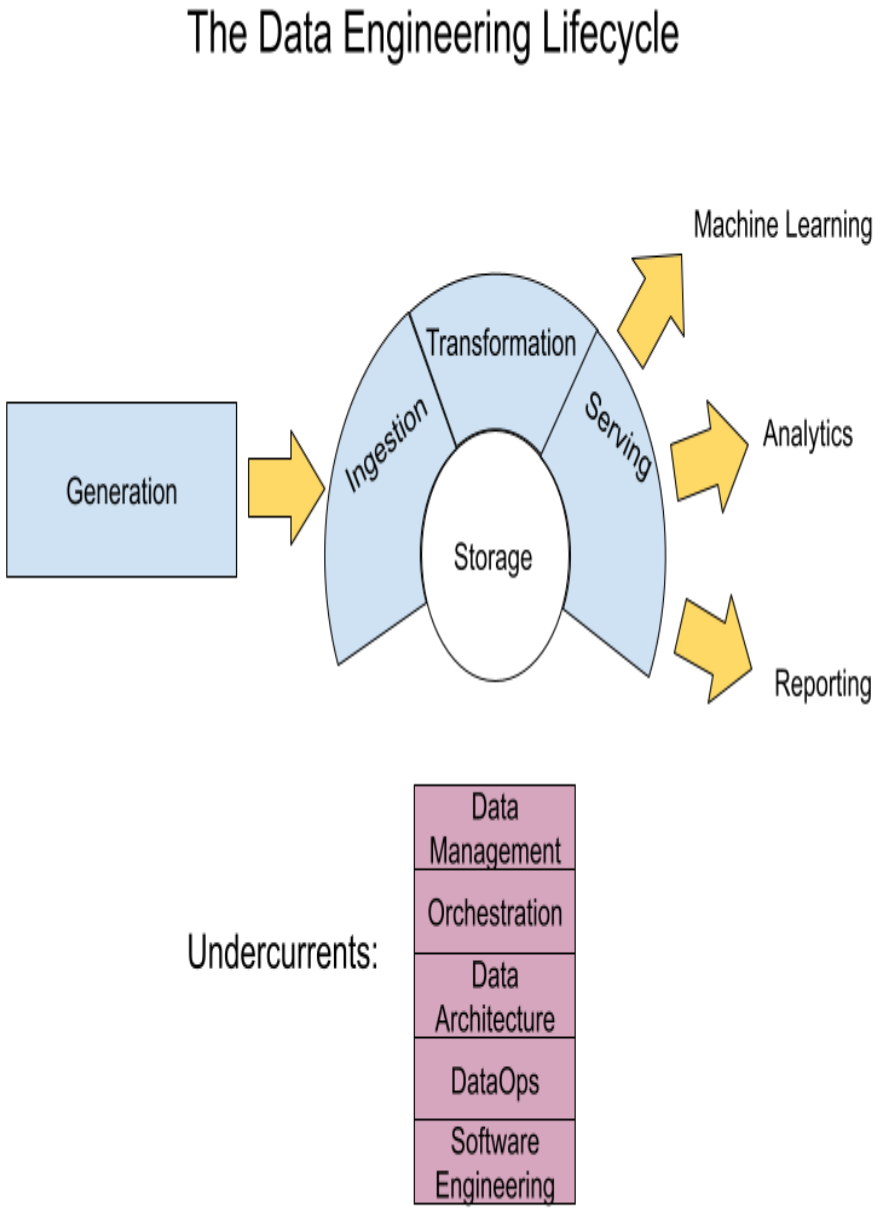


Figure 1-3. The Data Engineering Lifecycle

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

- Generation
- Ingestion
- Transformation
- Serving
- Storage

We put storage last because it plays a role in all the other stages. We also have a notion of *undercurrents,* i.e., ideas that are critical across the full lifecycle. We will cover the lifecycle more extensively in Chapter 2, but we outline it here because it is critical to our definition of data engineering.

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—data management, DataOps,[9] data architecture, orchestration, and general data lifecycle management.

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 conversant in acronyms such as CCPA and GDPR;[10] as they engineer pipelines, they concern themselves with privacy, anonymization, data garbage collection and compliance with regulations.

What's old is new again. Now that many of the hard 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 that contrasts with the traditional enterprise command-and-control approach. We view the

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

Data Engineering Defined

Okay, now that we've surveyed the recent past, we're ready to lay out our definition of data engineering. Let's unpack the common threads. In general, a data engineer gets data, stores it, and prepares it for consumption by data scientists and analysts.

For the purpose of this book, we define *data engineering* as follows:

Data engineering

> 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 data management, DataOps, data architecture, orchestration, and software engineering.

Data engineers

> Manage the data engineering lifecycle (see Figure 1-3) beginning with ingestion and ending with serving data for use cases, such as analysis or machine learning.

Put another way, data engineers produce reliable data that serves the business with predictable quality and meaning. The data engineering workflow is built atop systems supporting the data model, underpinned by the management, and configuration of storage and infrastructure. Data engineering systems and outputs are the backbones of successful data analytics and data science.

The data engineering workflow is roughly as follows (see Figure 1-4). This is the *general* process through which all data in an organization flows. Raw data is ingested from source systems, stored and processed, and served as conformed data models and datasets to consumers such as data scientists and data analysts. We'll return to the data engineering workflow throughout the book.

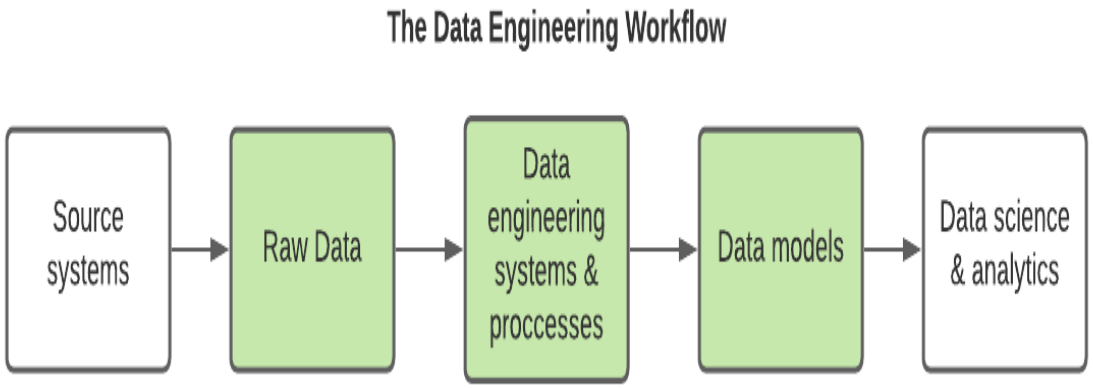


Figure 1-4. The Data Engineering Workflow

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. Data engineers serve data scientists and other data customers (Figure 1-5).

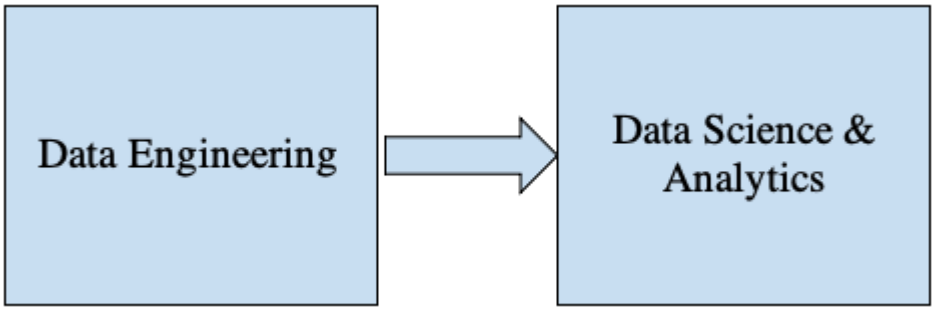
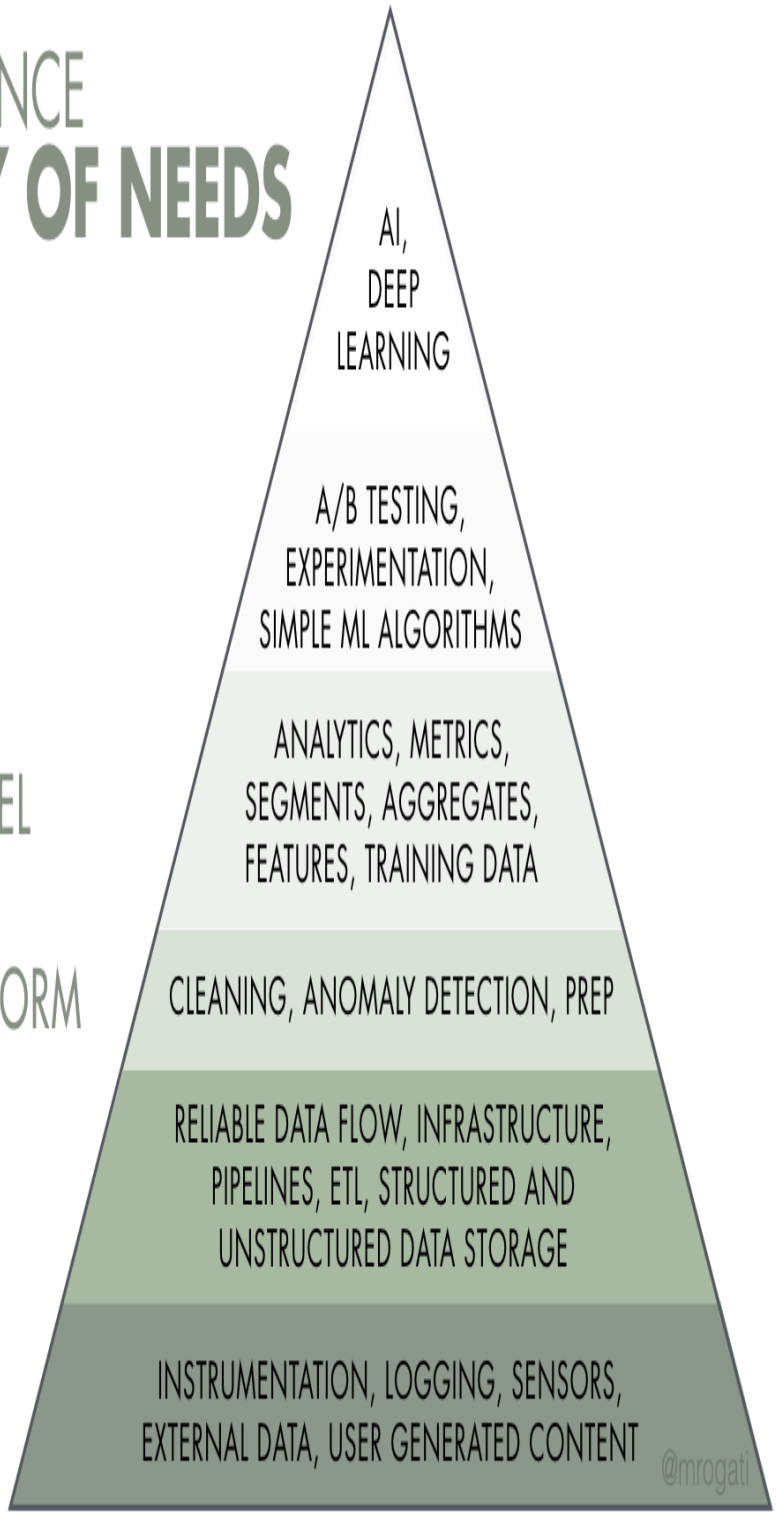


Figure 1-5. Data engineering sits upstream from data science

To justify this position, let's consider the Data Science Hierarchy of Needs[11] (see Figure 1-6). In 2017, Monica Rogati wrote an article by this title that showed where AI/ML sat in proximity to less "sexy" areas such as data movement/storage, collection, and infrastructure.

THE DATA SCIENCE HIERARCHY OF NEEDS



LEARN/OPTIMIZE

AGGREGATE/LABEL

EXPLORE/TRANSFORM

MOVE/STORE

COLLECT

Figure 1-6. The Data Science Hierarchy of Needs (Source: Rogati)

It is now widely recognized that data scientists spend an estimated 70% to 90% of their time on the bottom 3 parts of the hierarchy—gathering data, cleaning data, processing data—and only a small slice of their time on analysis and machine learning. Rogati argues that companies need to build a solid data foundation—the bottom 3 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 90%+ of their time focused on the top layers of the pyramid—analytics, experimentation, and machine learning. When data engineers focus on these bottom parts of the hierarchy, they build a solid foundation upon which data scientists can succeed.

With data science driving advanced analytics and machine learning, data engineering straddles the divide between getting data and getting value from data (see Figure 1-7). We believe data engineering will rise to be equal in importance and visibility to data science, with data engineers playing a vital role in making data science successful in production.

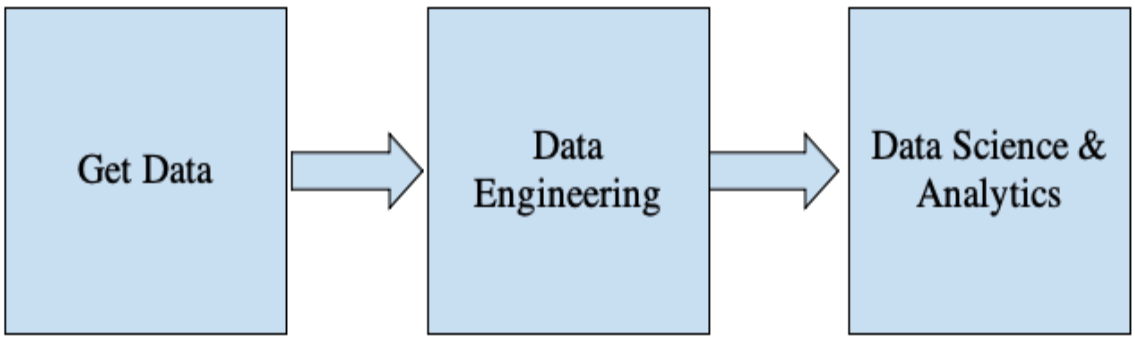


Figure 1-7. A data engineer gets data and provides value from data

Data Engineering Skills and Activities

The skillset of a data engineer encompasses what we call the "undercurrents" of data engineering—data management, data ops, data

architecture, and software engineering. It requires an understanding of how to evaluate data tools, and how they fit together across the data engineering lifecycle. It's also necessary to know how data is produced in source systems, as well as how analysts and data scientists will consume and create value from data after it is processed and curated. Finally, a data engineer juggles a lot of complex moving parts, and must constantly optimize along the axes of cost, agility, simplicity, reuse, and interoperability (Figure 1-8). We'll cover these topics in more detail in upcoming chapters.

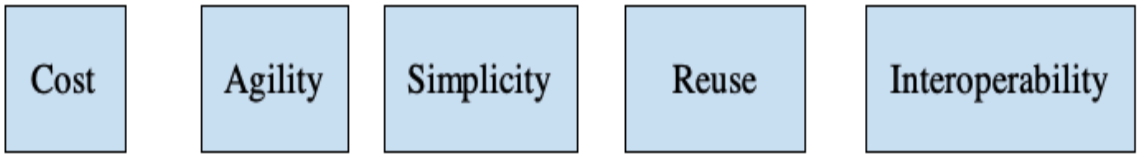


Figure 1-8. The balancing act of data engineering

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 to create a data solution. Utilizing these technologies (Hadoop, Spark, Teradata, and many others) often required 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 greatly 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 machine learning models, create reports or dashboards, perform data analysis, build KPIs, or develop software applications. That said, a data engineer should have a good functioning understanding of all of these areas, in order to best serve stakeholders.

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, in turn, has a significant impact on a data engineer's day-to-day job responsibilities, and potentially on their career progression as well. 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 how data is leveraged as a competitive advantage.

There are many versions of data maturity models, such as Data Management Maturity (DMM)[12] 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. In our data maturity model (Figure 1-9), we have three stages—starting with data, scaling with data, and leading with data. Let's look at each of these stages, and what a data engineer typically does at each stage.

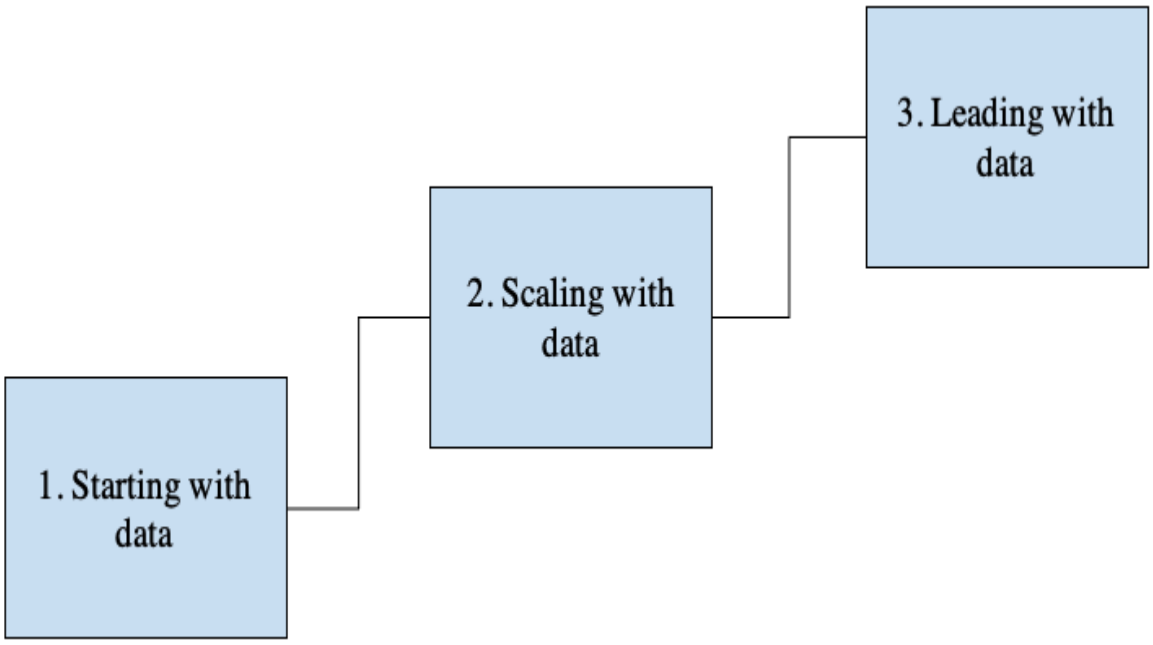


Figure 1-9. Our simplified data maturity model for a company

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 at all. Data architecture is 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 often 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. Machine learning is rarely successful at this stage, and so such projects are not recommended. We've seen countless data teams get stuck and fall short when they try to jump to machine learning without building a solid data foundation. The authors half-jokingly call themselves "recovering data scientists", largely from personal experience with being involved in premature data science projects without adequate data maturity or data engineering support.[13]

In organizations getting started with data, a data engineer should focus on the following:

- Get buy-in from key stakeholders, including executive management. Ideally, the data engineer should have a sponsor for key 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 what 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 that you designed.
- Build a solid data foundation upon which future data analysts and data scientists can 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.

Things to watch out for:

- This is a delicate stage with lots of pitfalls.
- Organizational willpower may wane.
- 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.
- Be visible and continue getting support.
- Avoid undifferentiated heavy lifting. Don't get caught in the trap of boxing yourself in with unnecessary technical complexity. Use off-the-shelf, turnkey solutions wherever possible.
- Build custom solutions and code only where you're creating 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 truly 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:

- Establish formal data practices

- Create scalable and robust data architectures
- Adopt DevOps and DataOps practices
- Build systems that support machine learning
- Continue to avoid undifferentiated heavy lifting and customizing only where there's a competitive advantage

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

Things to watch out for:

- As we grow more sophisticated with data, there's a temptation to adopt bleeding-edge technologies based on social proof from Silicon Valley companies. This is rarely the best use of your time and energy. Any technology decisions should be driven by the value that they'll deliver to your customers.
- The main bottleneck for scaling is not cluster nodes, storage, or technology, but the data engineering team itself. 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, as a data genius who can deliver magical products. Shift your focus instead to pragmatic leadership—begin transitioning to the next maturity stage now. 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 leads with data and is data-driven. The automated pipelines and systems created by data engineers allow people within the company to do self-service analytics and machine learning. Introducing new data sources is seamless and tangible value is derived. Data engineers implement formal controls and practices to ensure data is

always available to the people and systems that need it. Data engineering roles continue to specialize more deeply than in Stage 2.

In organizations that are in Stage 3 of data maturity, a data engineer will continue building on prior stages, plus:

- 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. Data management, data serving, DataOps, etc. Deploy tools that expose and disseminate data throughout the organization, including data catalogs, data lineage tools, metadata management systems, etc.
- Collaborate efficiently with software engineers, machine learning engineers, analysts, etc. Create a community and environment where people can collaborate and speak openly, no matter what role or position you're in.

Things to watch out for:

- At this stage, complacency is a significant danger. Once organizations reach Stage 3, they must focus constantly on maintenance and improvement or they risk falling back to a lower stage.
- Technology distractions are a bigger 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

Because data engineering is a relatively new discipline, there is little available in the way of formal training to enter the field. Universities don't

have a common data engineering path. Although there are a handful of data engineering boot camps and online tutorials covering random topics, a common curriculum for the subject doesn't yet exist. Those entering data engineering arrive with varying backgrounds in education, career, and skillset. Everyone entering the field should expect to invest a significant amount of time in self-study (including reading this book).

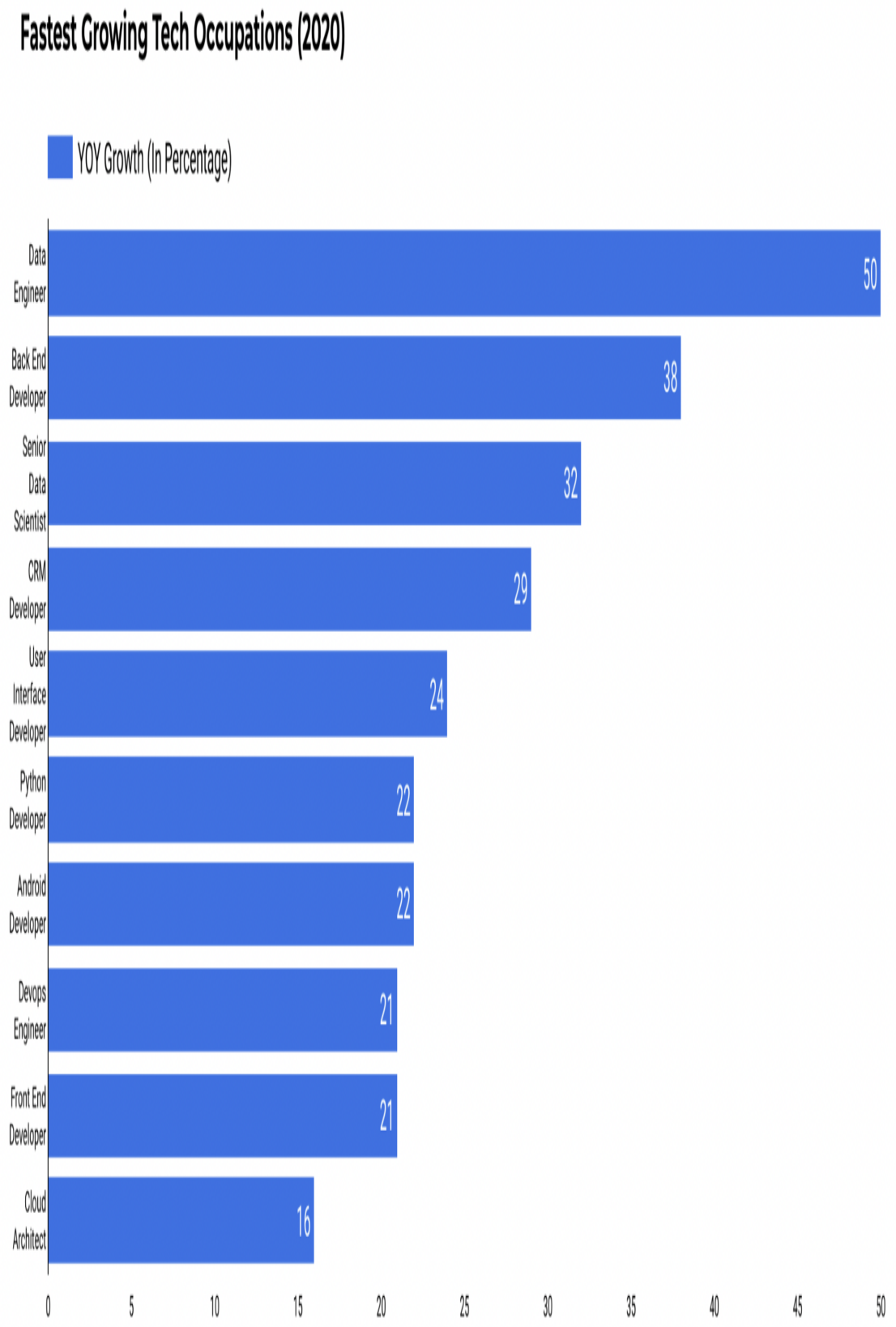

Fastest Growing Tech Occupations (2020)
YOY Growth (In Percentage)
Data Engineer
50
Back End Developer
38
Senior Data Scientist
32
CRM Developer
29
User Interface Developer
24
Python Developer
22
Android Developer
22
Devops Engineer
21
Front End Developer
21
Cloud Architect
16
0
5
10
15
20
25
30
35
40
45
50

Figure 1-10. Data engineering is the fastest-growing tech occupation (2020)[14]

If you're pivoting your career into data engineering (see Figure 1-10 for recent growth of the field), we've found that the transition is easiest when moving from an adjacent field, such as software engineering, ETL development, database administration, data science, and data analysis. These disciplines tend to be "data aware" and provide good context for data roles in an organization. They also tend to equip folks with the relevant technical skills and context to solve data engineering problems.

Despite the lack of a formalized path, there is a requisite body of knowledge that we believe a data engineer should know in order 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 tradeoffs. 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—as well as the broader implications of data across the organization. Data engineering is a holistic practice—the best data engineers view their responsibilities through both business and technical lenses.

Business Responsibilities

The macro responsibilities we list below aren't just important for a data engineer, but 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 with each other, and which silos exist. These observations will be invaluable to your success.

Understand how to scope, and how to gather business and product requirements.

Simply put, you need to know what to build and make sure 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. This is dangerously wrong; in fact, they are fundamentally cultural, requiring buy-in across the organization.

Cost control.

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.

Continuously learn.

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 key, both for technical and non-technical people. We often see data teams succeed or fail based upon their communication with other stakeholders; success or failure is very 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

At a high level, you must understand how to build architectures that optimize performance and cost, whether the components are prepackaged or homegrown. Ultimately, architectures and constituent technologies are building blocks to serve the data engineering lifecycle. Recall the stages of the data engineering lifecycle (Figure 1-3):

- Data generation
- Data ingestion
- Data storage
- Data transformation
- Serving data

The undercurrents of the data engineering lifecycle are:

- Data management
- DataOps
- Data architecture
- Software engineering

Zooming in a bit, we discuss some of the tactical data and technology skills you'll need as a data engineer here; we will discuss these in much 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. A great

deal of low-level programming effort previously expected of engineers is now replaced by fully managed services, managed open-source, and simple plug-and-play software-as-a-service (SAAS) offerings. For example, data engineers now focus on high-level abstractions, writing pipelines as code within an orchestration framework, or using dataframes inside Spark instead of having to worry about the nitty-gritty of Spark internals.

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 handicapped, 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 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 map-reduce 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. Further, a growing number of data engineer 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, SKLearn, Tensorflow, Pytorch, PySpark, and countless others. Python acts as 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

Popular for Apache open source projects such as Spark, Hive, Druid, and many more. 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). If you're using a popular open-source data framework, understanding Java or Scala will be very helpful.

Bash

The command line interface for Linux operating systems. Knowing Bash commands and being comfortable using CLI's will greatly 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 map reduce and the "Big Data" era relegated SQL to passé status. Since then, a variety of developments have dramatically enhanced the utility of SQL in the data engineering life cycle. Spark SQL, BigQuery, Snowflake, and many other data tools can process massive amounts of data using declarative, set-theoretic SQL semantics; SQL is also supported by many streaming frameworks, such as Flink, Beam, and Kafka. We believe that competent data engineers should be highly proficient in SQL.

Are we saying that SQL is an end-all-be-all language? Not at all. SQL is a powerful tool that can quickly solve many complex analytics and data transformation problems. Given that time is a primary constraint for data engineering team throughput, a tool that combines simplicity and high productivity should be embraced. 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 JSON parsing and nested data, and consider leveraging a SQL management framework such as DBT (Data Build Tool).[15]

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; while a SQL expert could likely write a query to stem and tokenize raw text in an NLP (natural language processing) pipeline, 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#, Julia, etc. Developing in these languages is often necessary when they are popular across the company, or to use domain-specific data tools. For instance, Javascript has proven popular as a language for user-defined

functions in cloud data warehouses, while C# and Powershell are important 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 understand 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 useful in the lifecycle.

The Continuum of Data Engineering Roles, from A to B

Despite job descriptions that 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 skillset. Just as data maturity is a helpful guide to understand the types of data challenges a company will face as it grows its data capability, it's helpful to look at some key distinctions in the types of work data engineers do. Though these distinctions are simplistic, it clarifies what data scientists and data engineers do, and avoids lumping either role into the "unicorn" bucket.

In data science, there's the notion of Type A and Type B data scientists.[16] 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 also possess strong programming skills. The Type B Data Scientist builds systems that make data science work in production.

Borrowing from the Type A and Type B 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 simple and abstract as possible and not re-inventing the wheel. Type A data engineers manage the data engineering lifecycle using mostly or completely off-the-shelf products, managed services, and tools. Type A data engineers work at companies across industries, and across 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 they 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 both with technical and non-technical people, as well as face different directions (internal and external). Let's explore what data engineers do inside an organization, and who they interact with.

Internal-Facing Versus External-Facing Data Engineers

A data engineer serves several end-users and faces many internal and external directions (Figure 1-11). Since not all data engineering workloads and responsibilities are the same, it's important to understand who 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.

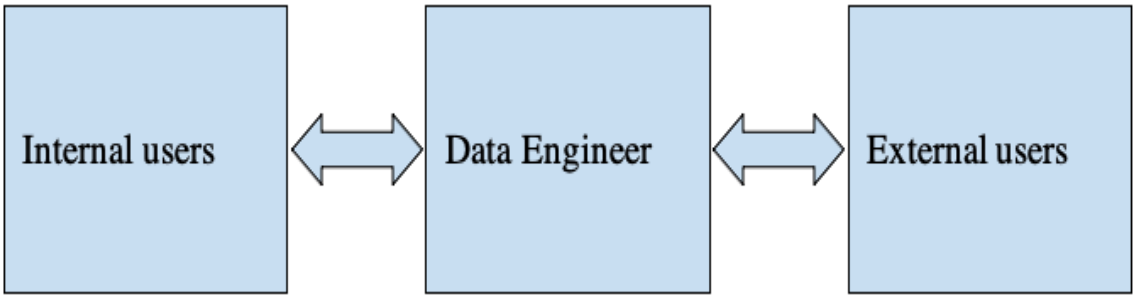


Figure 1-11. The directions a data engineer faces

An external-facing data engineer normally aligns with the users of external-facing applications, such as social media apps, IoT devices, and e-commerce 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, then back to the application (see Figure 1-12).

We note that external-facing data engineering comes with a special set of problems. External facing query engines often handle much larger concurrency loads than internal-facing systems; engineers also need to consider putting tight limits on queries that users can run in order to limit the infrastructure impact of any single user; and security is a much more complex and sensitive problem for external queries, especially if the data being queried is multi-tenant, i.e., data from many customers is housed in a single table.

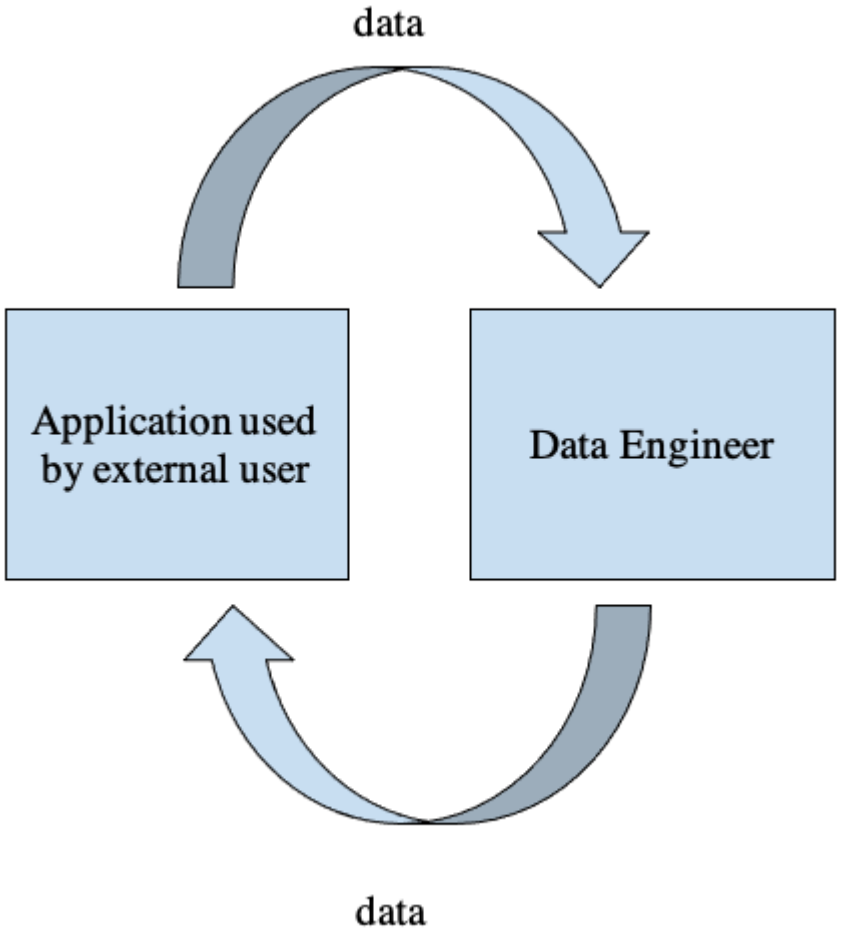


Figure 1-12. External-facing data engineer systems

An internal-facing data engineer typically focuses on activities crucial to the needs of the business and internal stakeholders (Figure 1-13). Examples include the creation and maintenance of data pipelines and data warehouses for BI dashboards, reports, business processes, data science, and machine learning models.

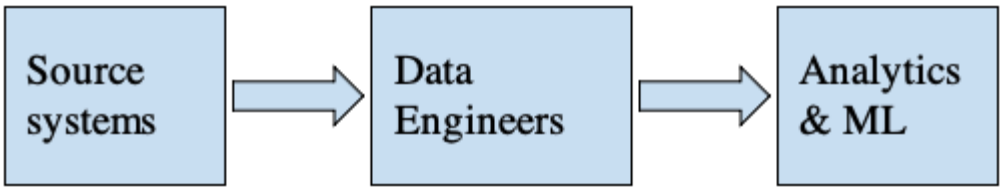


Figure 1-13. Internal-facing data engineer

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, etc., as mentioned above.

Data Engineers and Other Technical Roles

In practice, the data engineering lifecycle cuts across many different domains of responsibility. Data engineers sit at the nexus of a variety of roles, interacting with many organizational units, either directly or through managers. Let's look at who a data engineer may impact. We'll start with a discussion of technical roles connected to data engineering (Figure 1-14).

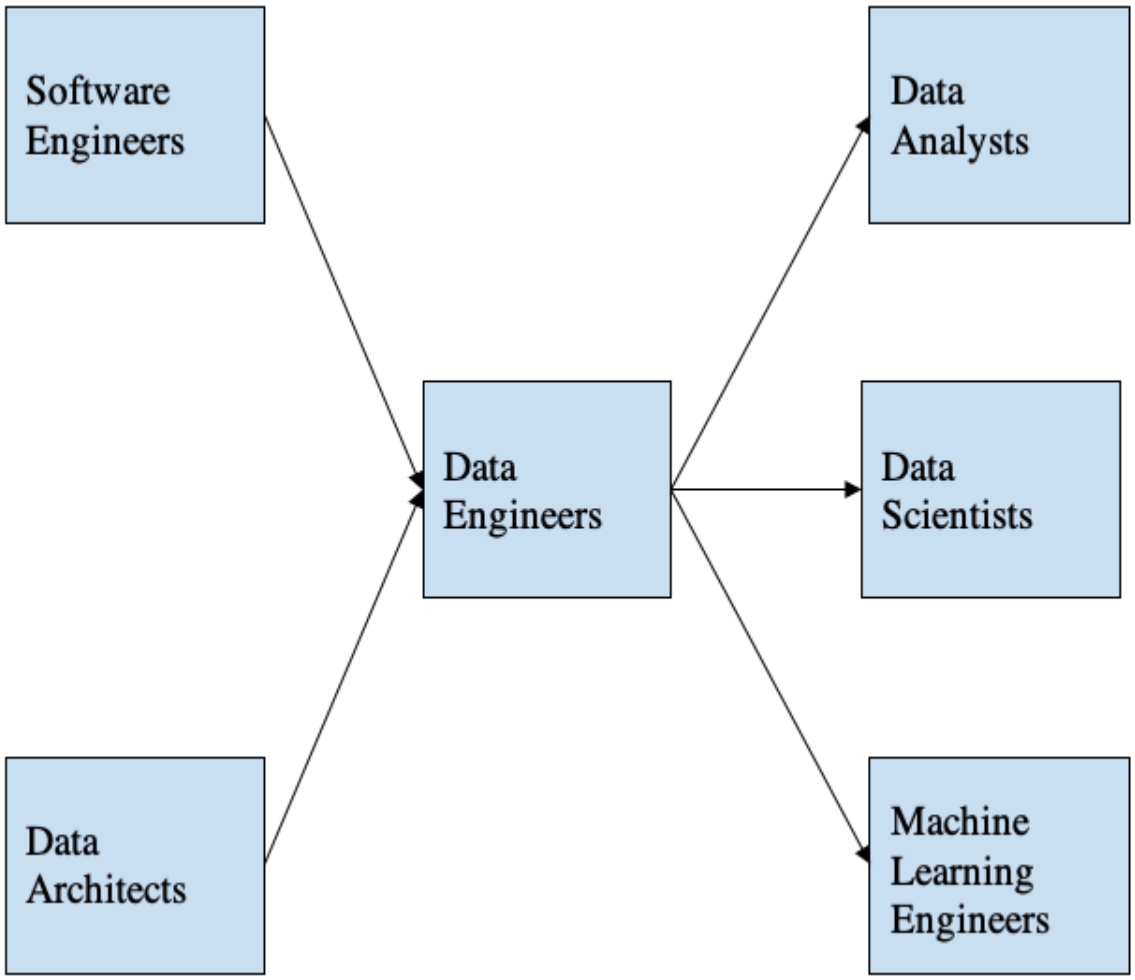


Figure 1-14. Key technical stakeholders of data engineering

In a nutshell, the data engineer is a hub between data producers, such as software engineers and data architects, and data consumers such as data analysts, data scientists, and machine learning 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 machine learning 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, as well as the source systems producing the data you'll need. Below we discuss a few common upstream stakeholders you'll encounter, such as 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,[17] mapping out processes, and overall data architecture and systems. They also serve as a bridge between the technical and non-technical sides of an organization. Successful data architects generally have 'battle scars' from extensive engineering experience, allowing them to guide and assist engineers, while successfully communicating the challenges of engineering to non-technical 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 major initiatives. Data architects often play a central role in cloud migrations and greenfield cloud design, which at the time of this writing are either inflight initiatives or on a roadmap for many companies.

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 important visionaries in enterprises, working hand in hand with data engineers to determine the big picture of architecture practices and data strategies.

Depending on the data maturity and size of a company, 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.

Readers will note that we have placed data architects in the *upstream stakeholder's* section. This is because data architects often help to design application data layers that are source systems for data engineers. In general, architects may also interact with data engineers at various other stages of the data engineering lifecycle.

We will 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 a major asset in their own right. This internal data is in contrast to *external data*, pulled from SAAS platforms or partner businesses. In well-run technical organizations, software engineers and data engineers coordinate from the inception of a new project to design application data for consumption by analytics and machine learning applications.

A data engineer should work together with software engineers to understand the applications that generate data, the volume, frequency, and format of the data being generated, and anything else that will impact the data engineering lifecycle, such as data security and regulatory compliance. In addition, data engineers and architects can coordinate on designing the

data bus that captures data from an application, whether this is a messaging queue, a log sink, or a batch extraction process.

DevOps engineers

DevOps engineers often produce data through operational monitoring; we classify them as upstream of data engineers for this reason, but they may also be downstream, consuming data through dashboards; or they may interact with data engineers directly in coordinating operations of data systems.

Downstream stakeholders

The modern data engineering profession exists to serve downstream data consumers and use cases. In this section, we'll discuss how data engineers interact with a variety of downstream roles. We'll also introduce a few different service models, including centralized data engineering teams and cross-functional teams. Chapter 8 will provide an in-depth discussion of these topics.

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 between 40 and 80% of their time collecting, cleaning, and preparing data.[18] In the authors' experience, these numbers are representative of poor data science and engineering practices. In particular, many popular and useful 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 that they produce. Furthermore, locally developed code and environments

are often difficult to deploy in production, and a lack of automation significantly hampers data science workflows. If data engineers are doing their job and collaborating successfully, data scientists shouldn't spend their time collecting, cleaning, and preparing data after initial exploratory work; data engineers should do this work.

The need for production-ready data science is a major driver behind the emergence of the data engineering profession. Data Engineers should help data scientists to enable a path to production. In fact, the authors moved from data science to data engineering in recognition of 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 business intelligence tools such as PowerBI, Looker, or Tableau. Data analysts are domain experts in the data they work with frequently, and therefore 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 both data engineers and data scientists. Machine learning engineers develop advanced machine learning techniques, train models, and design and maintain the infrastructure running machine learning processes in a scaled production

environment. ML engineers often have advanced working knowledge of machine learning and deep learning techniques, as well as frameworks such as PyTorch or Tensorflow.

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

As mentioned above, the boundaries between ML engineering, data engineering, and data science are fuzzy. Data engineers may have some DevOps responsibilities over ML systems, and data scientists may work closely with ML engineering in designing advanced machining learning processes.

The world of ML engineering is growing at a rapid pace and parallels a lot of the same developments occurring in data engineering. Whereas several years ago the attention of machine learning was focused on how to build models, ML engineering now increasingly emphasizes incorporating best practices of ML Operations (MLOps) and other mature practices previously adopted in software engineering and DevOps.

AI researchers work on new, advanced machine learning techniques. AI researchers may work inside large technology companies, specialized intellectual property startups (OpenAI, DeepMind), or academic institutions. Some practitioners are partially dedicated to research in conjunction with ML engineering responsibilities inside a company. Those working inside specialized machine learning labs are often 100% dedicated to research. Research problems may target immediate practical applications or more abstract demonstrations of artificial intelligence. AlphaGo and GPT-3 are great examples of machine learning research projects. We've provided some references in the *further reading* section at the end of the chapter.

AI researchers in well-funded organizations are highly specialized and operate with supporting teams of engineers to facilitate their work. For

example, see job listings for OpenAI or DeepMind. Machine learning engineers in academia usually have fewer resources, but rely on teams of graduate students, postdocs, and university staff to provide engineering support. Machine learning engineers who are partially dedicated to research often rely on the same support teams for research and production.

Data Engineers and Business Leadership

So far, we've discussed technical roles a data engineer interacts with. But data engineers also operate more broadly as organizational connectors, often in a non-technical capacity. Businesses have come to rely increasingly on data as a core part of many products, or as a product in itself. Data professionals, such as 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 central assets for modern businesses. CEOs now concern themselves with initiatives that were once the exclusive province of IT, such as major cloud migrations or deployment of a new customer data platform. This is certainly the case at tech companies such as Google, Amazon, and Facebook, but also in the wider business world.

Chief executive officer

CEOs at non-tech companies generally don't concern themselves with the nitty-gritty of data frameworks and software. Rather, 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. That is, data engineers and their managers maintain a map of what data is available to the organization -- both internally and from third parties -- in what timeframe. They are also tasked to study major 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 both information technology and business processes—either alone is insufficient. CIOs direct the information technology organization, setting ongoing policies while also defining and executing major initiatives under the direction of the CEO.

In organizations with well-developed data culture, CIOs often collaborate with data engineering leadership. 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 the adoption of major architectural elements, such as ERP and 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 external-facing 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, system architecture, and much more. 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 in recognition of 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 the business utility of data but should have a strong technical grounding. CDOs oversee data products, strategy, and initiatives, as well as core

functions such as master data management and data privacy. Occasionally, CDOs oversee 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, where the chief analytics officer is responsible for analytics, strategy, and decision making for the business. A CAO may oversee data science and machine learning as well, though this largely depends on whether or not the company has a CDO or CTO role.

Chief algorithms officer

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

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

Data engineers and project managers

Data engineers often work on large 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, creating new data architectures from scratch, able to choose from a breathtaking number of best-of-breed architecture and tooling options.

These large initiatives often benefit from *project management* (in contrast to product management, which we discuss below). Where 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 be addressed, and new initiatives they want to launch. Project managers must filter a long list of requests and prioritize key deliverables to keep projects on track and better serve the company as a whole.

Data engineers interact with project managers, often in planning sprints for projects, as well as 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. As the broader corporate world has adopted a data-centric focus, data engineers interact more frequently with product managers. Similar to 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 a variety of managers beyond project and product managers. However, these interactions usually follow either the services model or the cross-functional model. That is, 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" and Jesse

Anderson's "Data Teams" (see the *further reading* section at the end of the chapter). 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.

Conclusion

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

- Defining data engineering, and describing what data engineers do
- Describing the types of data maturity a company may
- Type A and Type B data engineers
- Who data engineers work with
- A stab at the future of data engineering

We hope that the first chapter has whetted the reader's appetite, whether they are practitioners of software development, data science, machine learning engineering, or business stakeholders, entrepreneurs, venture capitalists, etc. Of course, there is a great deal still to elucidate in subsequent chapters. In Chapter 2, we will cover the data engineering lifecycle in detail, followed by architecture in Chapter 3. In subsequent chapters, we will get into the nitty-gritty of technology decisions for each part of the lifecycle. The entire data field is in flux, and the aim in each chapter is to focus on the "immutables," or at least perspectives that will be valid for many years to come in the midst of relentless change.

---

1 https://www.altexsoft.com/blog/datascience/what-is-data-engineering-explaining-data-pipeline-data-warehouse-and-data-engineer-role/

2 Jesse Anderson, https://www.jesse-anderson.com/2018/06/the-two-types-of-data-engineering/

3 Maxime Beauchemin, https://medium.com/free-code-camp/the-rise-of-the-data-engineer-91be18f1e603

4 *What Is Data Engineering?* (O'Reilly 2020), https://learning.oreilly.com/library/view/what-is-data/9781492075578/ch01.html

5 https://www.lexico.com/en/definition/big_data

6 https://www.wired.com/2011/10/how-yahoo-spawned-hadoop/

7 https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/

8 https://twitter.com/danariely/status/287952257926971392?lang=en

9 DataOps (aka Data Operations). We will return to this topic throughout the book, starting in Chapter 2. For more information, read the DataOps Manifesto.

10 California Consumer Privacy Act and General Data Protection Regulation.

11 https://hackernoon.com/the-ai-hierarchy-of-needs-18f111fcc007

12 https://cmmiinstitute.com/dmm

13 https://www.linkedin.com/pulse/what-recovering-data-scientist-joe-reis/

14 https://towardsdatascience.com/should-you-become-a-data-engineer-in-2021-4db57b6cce35

15 https://www.getdbt.com/

16 https://medium.com/@rchang/my-two-year-journey-as-a-data-scientist-at-twitter-f0c13298aee6

17 https://www.dataversity.net/data-architect-vs-data-engineer/

18 There are a variety of references for this notion: https://blog.ldodds.com/2020/01/31/do-data-scientists-spend-80-of-their-time-cleaning-data-turns-out-no/; https://www.datanami.com/2020/07/06/data-prep-still-dominates-data-scientists-time-survey-finds/. This cliche is widely known, but there's healthy debate around its validity in different practical settings.

Chapter 2. The Data Engineering Lifecycle

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the second chapter of the final book.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at book_feedback@ternarydata.com.

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 seeing an explosion of new data technologies and practices, with higher levels of abstraction and ease of use. Some of these technologies will succeed, and most will fade into obscurity at an ever-faster rate. 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?

In simple terms, the data engineering lifecycle is the series of stages that turn raw data ingredients into a useful end product, ready for consumption by analysts, machine learning engineers, etc. In this chapter, we introduce the major stages of the data engineering lifecycle, focusing on core concepts and saving details of each stage to later chapters.

We divide the data engineering lifecycle into the following five stages (Figure 2-1):

- Generation: source systems
- Ingestion
- Storage
- Transformation
- Serving data

In general, the data engineering lifecycle starts by getting data from source systems, storing it, and transforming and serving data to analysts, data scientists, machine learning engineers, and others. The stages in the middle —ingestion, storage, 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 step, or weave together in interesting ways.

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

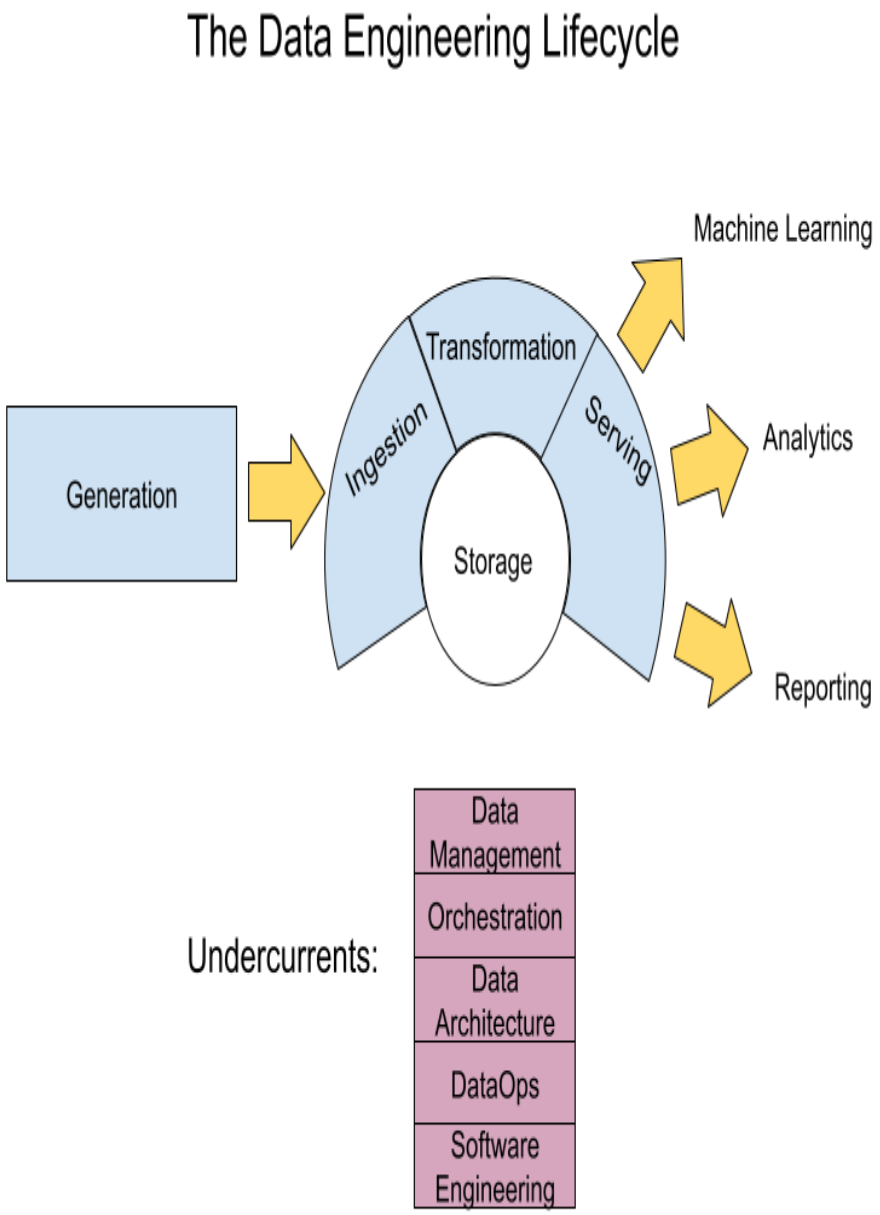


Figure 2-1. Components and undercurrents of the data engineering lifecycle

The Data Lifecycle Versus the Data Engineering Lifecycle

You may be wondering what the difference is 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 full data lifecycle that is owned by data engineers (see 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, namely data generation in source systems to serving the data for analytics and machine learning use-cases.

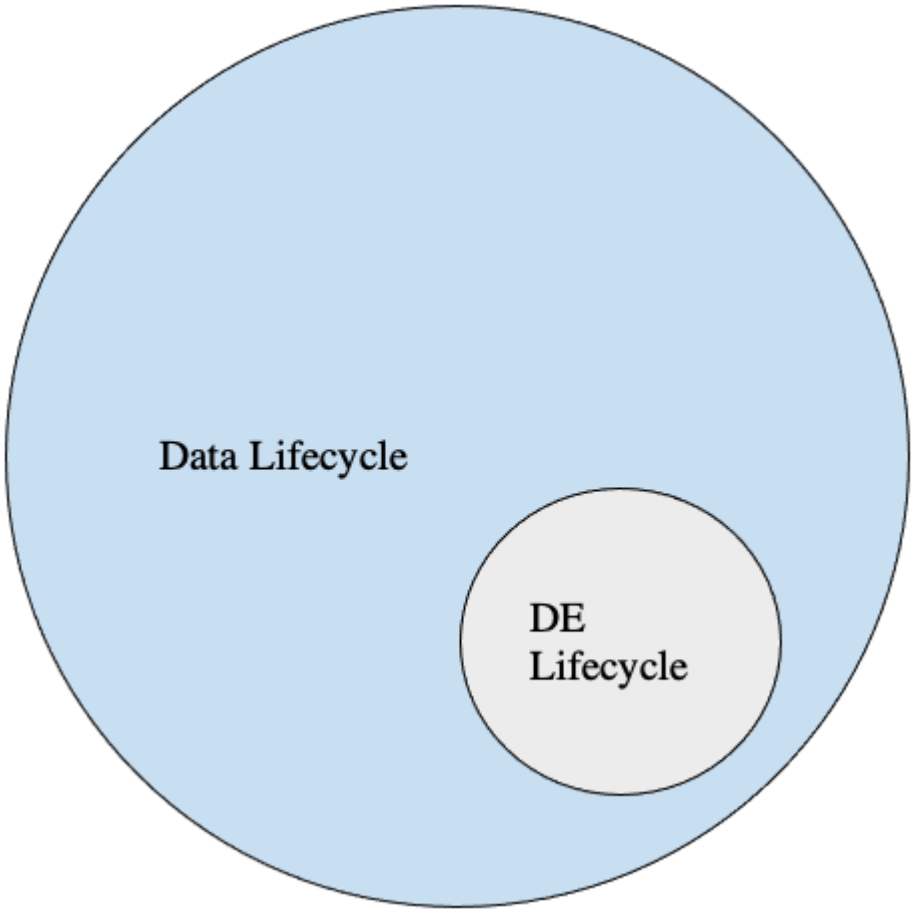


Figure 2-2. The data engineering lifecycle is a subset of the full data lifecycle

Generation: Source Systems

The data engineer needs to understand how source systems work, how they generate data, the frequency and velocity of the data, and the variety of data they generate. A major challenge in modern data engineering is that

engineers must work with and understand a dizzying array of data source systems. As an illustration, let's look at two common source systems, one very traditional and the other a modern example.

Figure 2-3 illustrates a traditional source system, with applications backed by a database. This source system pattern became popular in the 1980s with the explosive success of relational databases. The application + database pattern remains popular today with various modern evolutions of software development practices. For example, with microservices, applications often consist of many small service/database pairs rather than a single monolith. NoSQL databases like MongoDB, CosmosDB, Spanner, and DynamoDB are compelling alternatives to traditional RDBMS systems.

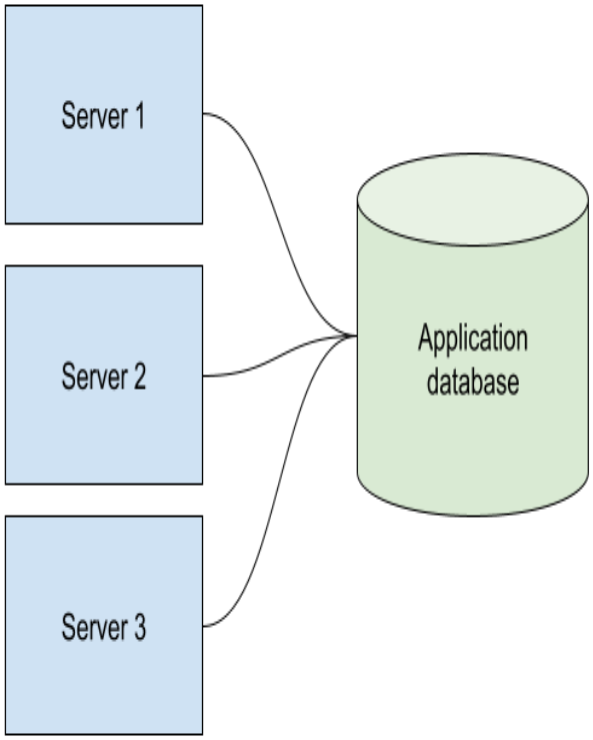


Figure 2-3. Source system example: an application database.

Let's look at another example of a source system. Figure 2-4 illustrates an IoT swarm, where a fleet of devices (circles) sends data messages (rectangles) to a central collection system. This type of system is

increasingly common as IoT devices—sensors, smart devices, and much more—proliferate in the wild.

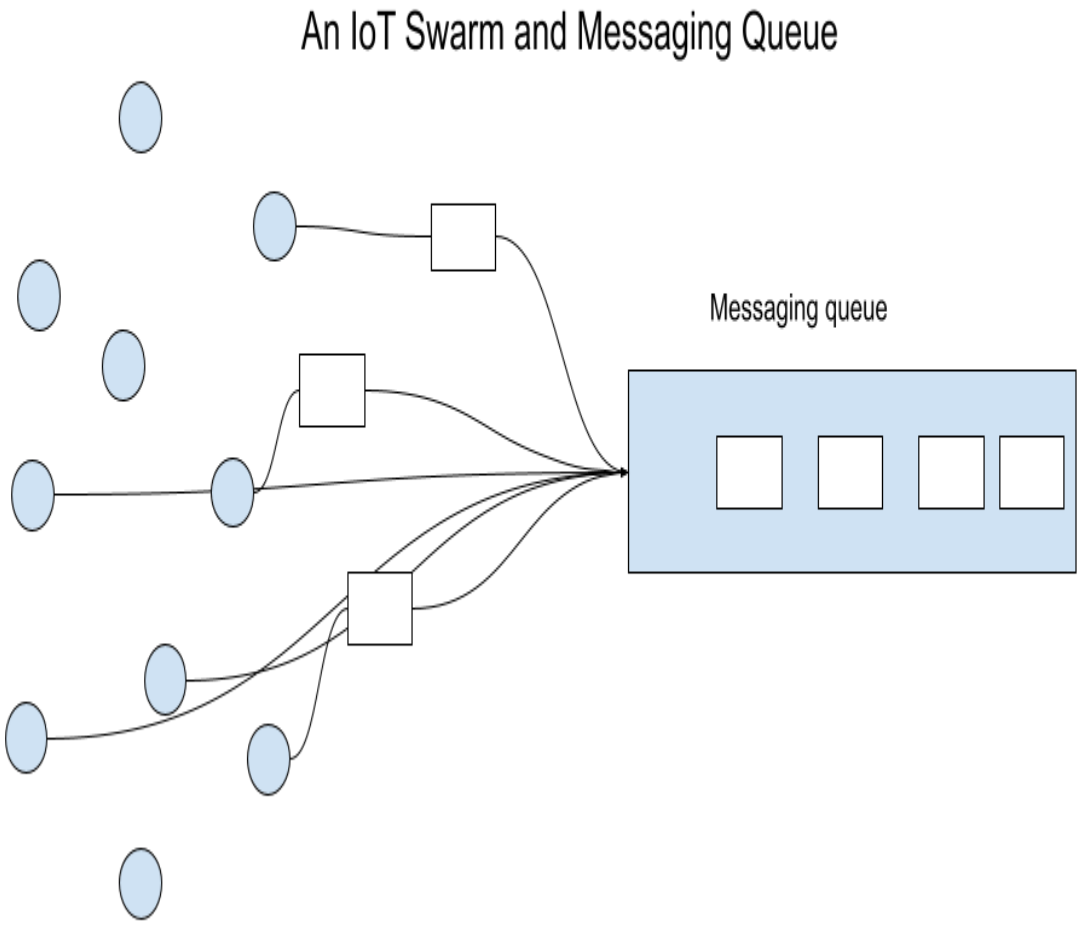


Figure 2-4. Source system example: an IoT Swarm and Messaging Queue

Evaluating source systems: key engineering considerations

Below are some important characteristics of source systems that data engineers must think about. This is by no means an exhaustive list, but rather a starting set of evaluation questions:

- What are the basic characteristics of the data source? Is it an application? A swarm of IoT devices?
- How does the source handle state?
- At what rate is data generated? How many events per second? How many GB 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, bad formatting, etc—deviate from the norm?
- How often do errors occur?
- Will the data contain duplicates?
- Will some data values arrive late, possibly much later than other messages produced at the same time?
- What is the schema of the ingested data? Will data engineers need to join across several tables or even several systems to get a full 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 as 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 that is consumed by downstream systems. This includes human-generated spreadsheets, IOT sensors, web and mobile applications, and everything in between. Each source has its unique volume and cadence of data generation. A data engineer should understand 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 queries to feed analytics cause performance issues with an application?

One of the most challenging nuances of source data is a *schema*. The schema defines the hierarchical organization of data. Logically, we can think of data at the level of a full source system, drilling down into individual tables, all the way to the structure of individual fields. The schema of data shipped from source systems is handled in a variety of ways. One popular model is *schemaless*. Schemaless doesn't mean the absence of schema—rather, it means that the schema is defined by the application as data is written, whether to a messaging 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, which application writes must conform to.

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. Taking raw data input in the source system schema and transforming this into output useful for analytics is a key component of the data engineer's job. This job becomes more challenging as the source schema evolves.

There are numerous ways to transmit data from a source, including:

- Programmatically
- Message brokers
- APIs
- RPC
- Streams
- Output files

We will dive into source systems in greater detail in Chapter 5.

Ingestion

After you understand the data source and the characteristics of the source system you're using, you need to gather the data. The second stage of the data engineering lifecycle is data ingestion from source systems. In our experience, source systems and ingestion represent the biggest bottlenecks of the data engineering lifecycle. The source systems are normally outside of your direct control, and might randomly become unresponsive or provide data of poor quality. Or, your data ingestion service might mysteriously stop working for any number of reasons. As a result, data flow stops or delivers bad data for storage, processing, and serving. Unreliable source and ingestion systems have a ripple effect across the data engineering lifecycle.

Assuming you've answered the big questions listed above about source systems, and you're in good shape, you're now ready to ingest data. Let's cover some key things to think about.

Key engineering considerations for the ingestion phase

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

- What's the use case for the data I'm ingesting?

- Can I re-use this data, versus having to create multiple versions of the same dataset?
- Where is the data going? What's the destination?
- 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 the data need to be transformed before it reaches its destination? If so, would an in-flight transformation, where the data is transformed within the stream itself, be appropriate?

These are just a sample of the things you'll need to think about with ingestion, and we'll cover those questions and more in Chapter 6. Before we leave, let's briefly turn our attention to two major data ingestion paradigms—batch versus streaming, and push versus pull.

Batch versus streaming

Virtually all data we deal with is inherently streaming. That is, 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, for example, 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. Due to factors that limited the ways that data could be processed, batch was—and still is—a very popular way to ingest data for downstream consumption, particularly in the areas of analytics and machine learning. However, the separation of storage and compute in many systems, as well as the ubiquity of event streaming and processing platforms, make 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 tradeoffs to understand and think about. Below are some questions to ask yourself when determining if 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 true, to the millisecond real-time data ingestion? Or would a micro-batch approach work where I accumulate and ingest 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?
- Is my streaming pipeline and system reliable and redundant in the event of infrastructure failure?

- What tools are most appropriate for the use case? Should I use a managed service (Kinesis, Pubsub, 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 tradeoffs?
- If I'm deploying a machine learning 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; there are inherently extra costs and complexities. Many great ingestion frameworks do handle both batch and micro-batch ingestion styles. That said, we think batch is a perfectly fine approach for many data ingestion very common use-cases such as model training and weekly reporting, which are inherently batch-oriented. Adopt true real-time streaming-only after identifying a business use case that justifies the extra complexity.

Push versus pull

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

Consider, for example, the ETL (extract, transform, load) process, commonly used in batch-oriented ingestion workflows. The *extract* part of ETL makes it clear that we're dealing with a pull ingestion model. In traditional ETL, the ingestion system queries a current table snapshot on a fixed schedule.

In another example, consider continuous change data capture (CDC), which is achieved in a few different 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 it is picked up by the ingestion system. 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 some kind of data buffering such as a queue. 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 the needs of downstream analytics, and greatly simplifies the lives of data engineers.

We'll discuss ingestion best practices and techniques in depth in Chapter 6.

Storage

After ingesting data, you need a place to store it. 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, modern data architectures in the cloud often leverage *several* storage solutions. Second, few data storage solutions function purely as storage, with many also supporting complex transformation queries; even object storage solutions may support powerful query capabilities (e.g AWS 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 frequently occurs in multiple places in a data pipeline, with storage systems crossing over with processing, serving, and ingestion stages. For

example, cloud data warehouses can store data, process data in pipelines, and serve it to analysts; streaming frameworks such as Kafka and Pulsar can function simultaneously as ingestion, storage, and query systems for messages; and object storage is a standard layer for the transmission of data.

In general with storage, data moves and goes somewhere, gets stored temporarily or permanently, then goes somewhere else, gets stored again, and so on.

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, object storage, etc.:

- Is this storage solution compatible with the required write speeds for the architecture?
- 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, an antipattern with significant performance overhead?
- Will this storage system handle anticipated future scale? You should take into account 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 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)
- For data governance, how are you tracking master data, golden records data quality, and data lineage? (We'll have more to say on these in the *data management* subsection of this chapter.)
- How are you handling compliance and data sovereignty? For example, can you store your data in certain geographical locations, but not others?

Data access frequency

Not all data is accessed in the same way. Retrieval patterns will greatly vary, based upon the type of data being stored and queried. This brings up the notion of the "temperatures" of data. Data access frequency will determine what "temperature" your data is. 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 in systems that serve user requests. This is data that should be stored for fast retrieval, where "fast" is relative to the use case. Lukewarm data is data that 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 data that is 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 extremely low 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 above. There is

not a one-size-fits-all universal storage recommendation. Every storage technology has its tradeoffs. Let's quickly list some of the common storage systems that a data engineer will encounter. There are countless varieties of storage technologies, and it's easy to be overwhelmed when deciding the best option for your data architecture. Here are several popular storage options:

- Relational database management systems (RDBMS)
- Data lake
- Data warehouse
- Data lakehouse
- Streaming systems with data retention capabilities.
- Graph database
- In-memory (Redis, Memcached, etc.)
- High performance NoSQL databases: RocksDB, etc.
- Feature stores
- Data catalog
- Metadata stores
- Spreadsheets

Chapter 7 will cover storage best practices and approaches in greater detail, as well as the crossover between storage and other lifecycle stages.

Transformation

The next stage of the data engineering lifecycle is *transformation*. As we mentioned, data can be ingested in its raw form, with no transformations performed. In most cases, however, data needs to be changed from its original form into something useful for downstream use cases. Without

proper transformations, data will not be in an appropriate form for reports, analysis, or machine learning. Typically, the transformation stage is where data begins to create value for downstream user consumption.

There are a whole host of data transformation types, which we will cover extensively in chapter 8. Immediately after ingestion, basic transformations map data into correct types (changing ingested string data into numeric and date types for example), put records into standard formats and remove bad records. 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 machine learning processes.

Key considerations for the transformation phase

When thinking about data transformations within the context of the data engineering lifecycle, it helps to consider the following. We'll cover transformations in-depth in Chapter 8.

- What's the cost and ROI of the transformation? All transformations should have an associated business value attached.
- Is the transformation expensive from a time and resource perspective?
- What value will the transformation bring downstream? In other words, what is the ROI of a transformation?
- Is the transformation as simple and self-isolated as possible?
- What business rules do the transformations support?
- During transformation, am I minimizing data movement between the transformation and the storage system?

Data can be transformed in batch, or while streaming in-flight. As mentioned in the section on ingestion, 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 such as Flink, Spark,

Beam, etc., as well as 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 data engineering lifecycle. Typically, data is transformed in source systems or 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—all of these transformative tasks add value to end-consumers of data.

Business logic is a major driver of data transformation. Business logic is critical for obtaining a clear and current picture of business processes. A simple view of raw retail transactions might not be useful in itself without adding the logic of accounting rules so that the CFO has a clear picture of financial health. In general, ensure a standard approach for implementing business logic across your transformations.

Data featurization for machine learning is another data transformation process. Featurization intends to extract and enhance features of data that will be useful for training machine learning models. Featurization is something of 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 to take away 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 very deep subject, and we cannot do it justice in this very brief introduction. Chapter 8 will delve into the various practices and nuances of data transformations.

Serving Data for Analytics, Machine Learning, and Reverse ETL

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 machine learning engineers can apply the most advanced modern techniques.

For now, let's take a look at some of the popular uses of data—analytics, machine learning, 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, dashboards, and do ad hoc analysis on the data. Whereas the bulk of analytics used to encompass business intelligence (BI), it now includes other facets such as operational analytics and customer-facing analytics (Figure 2-5). Let's briefly touch on these variations of analytics:

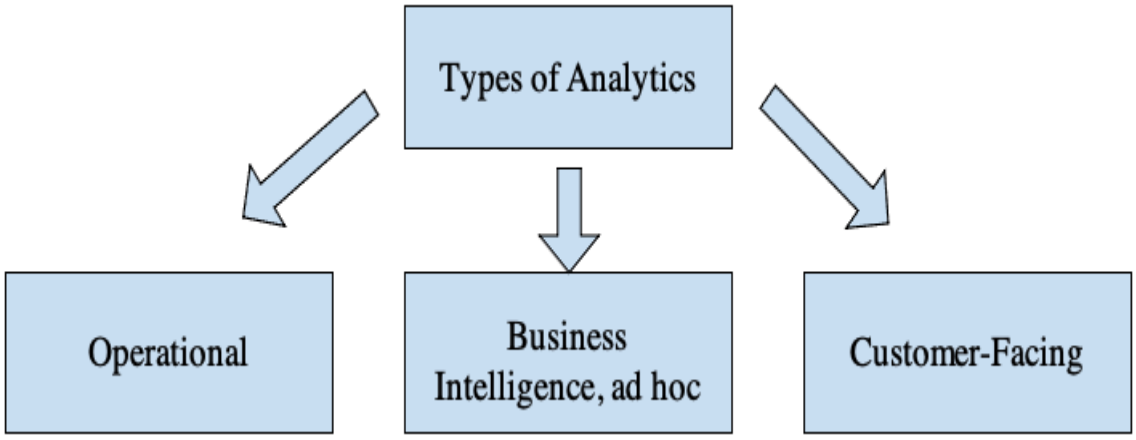


Figure 2-5. Types of Analytics

Business intelligence

Business intelligence (BI) marshalls collected data to describe the past and current state of a business. BI requires the processing of raw data using business logic. 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. That is, data is stored in a cleansed but fairly raw form, with minimal business logic post-processing. 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 accord with business definitions.55

As a company grows its data maturity, it will move from ad hoc data analysis to self-service analytics, which allows democratized data access to business users, without the need for IT to intervene. The capability to do self-service analytics assumes that data is in a good enough place that people across the organization can simply access the data themselves, slice and dice it however they choose and get immediate insights from it. We find that though self-service analytics is simple in theory, it's incredibly difficult to pull off in practice. The main reason is that poor data quality and organizational silos get in the way of

allowing widespread use of analytics, free of gatekeepers such as IT or reporting departments.

Operational analytics

Operational analytics focuses on the fine-grained details of operations, promoting actions that a user of the reports can act upon immediately. For example, operational analytics could be a live view of inventory, or real-time dashboarding of website health. In this case, data is consumed in a real-time manner, either directly from a source system, from a streaming data pipeline such as AWS Kinesis or Kafka, or aggregated in a real-time OLAP solution like Druid. 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.

Customer-facing analytics

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

With customer-facing analytics, the request rate for reports, and the corresponding burden on analytics systems, go 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. Where 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, likely 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 else there's a possibility of data leakage.

MULTITENANCY

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

Machine learning

Machine learning is one of the most exciting technology revolutions of our time. Once organizations reach a high level of data maturity, they can begin to identify problems that are amenable to machine learning and start organizing a machine learning practice.

The responsibilities of data engineers overlap significantly in analytics and machine learning, and the boundaries between data engineering, machine learning engineering, and analytics engineering can be fuzzy. For example, a data engineer may need to support Apache Spark clusters that facilitate both analytics pipelines and machine learning model training. They may also need to provide a Prefect or Dagster 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.

One recently developed tool that combines data engineering and ML engineering is the *feature store* (e.g. FEAST and Tecton, among others). Feature stores are designed to reduce the operational burden for machine learning engineers, by maintaining feature history and versions, supporting

feature sharing between 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 machine learning? It certainly helps. Regardless of the operational boundary between data engineering, ML engineering, and business analytics, etc., data engineers should maintain operational knowledge about the teams they work with. A good data engineer is conversant in the fundamental machine learning techniques and related data processing requirements (deep learning, featurization, etc.), in the use cases for models within their company, and the responsibilities of the organization's various analytics teams. This helps to maintain efficient communication and facilitate collaboration. Ideally, data engineers will build tools in collaboration with other teams that neither team is capable of building on its own.

This book cannot possibly cover machine learning in-depth. There's a growing ecosystem of books, videos, articles, and communities if you're interested in learning more; we include a few additional references in the *further reading* section at the end of this chapter.

Below are some considerations for the Serving Data phase, specific to machine learning:

- 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 machine learning engineers easily find useful data?
- Where are the technical and organizational boundaries between data engineering and machine learning engineering? This organizational question has major architectural implications.
- The dataset properly represents ground truth and isn't unfairly biased.

While machine learning is exciting, our experience is that companies often prematurely dive into it. Before investing a ton of resources into machine learning, take the time to build a solid data foundation. This means setting up the best systems and architecture across the data engineering lifecycle, as well as the machine learning lifecycle. More often than not, get good with analytics before moving to machine learning. Many companies have seen their machine learning dreams dashed because they undertook initiatives without appropriate foundations in place. On the other hand, data engineers should keep the serving stage of the data engineering lifecycle in mind at all times as they're building out the other stages, both as motivation and to steer and shape their architectures and initiatives.

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* is the process of taking processed data from the output side of the data engineering lifecycle and feeding it back into source systems as shown in Figure 2-6. In reality, this flow is extremely useful, and often necessary; reverse ETL allows us to take analytics, scored models, etc., and feed these back into production systems or SAAS platforms. Marketing analysts might calculate bids in Excel using the data in their data warehouse, then upload these bids to Google Ads. This process was often quite manual and primitive.

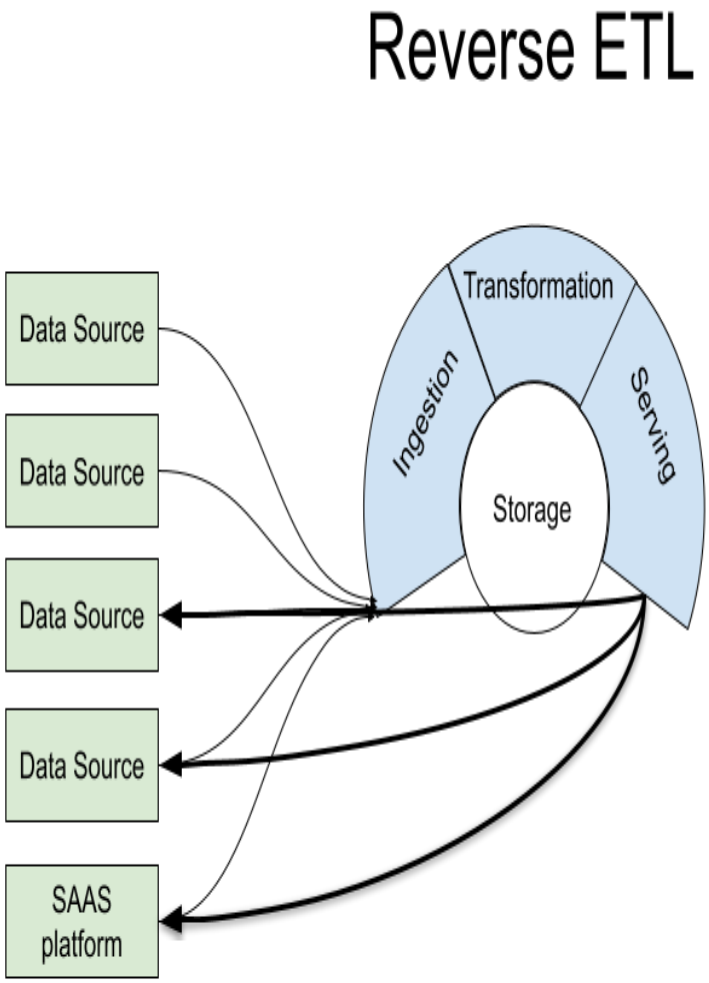


Figure 2-6. Reverse ETL

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 field, 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, businesses may want to push certain metrics from their data warehouse to a customer data platform or customer relationship management (CRM) system. Advertising platforms are another common use case, as in the Google Ads example. Expect to see more activity in the Reverse ETL, with an overlap in both data engineering and ML engineering.

The jury is out 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.

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

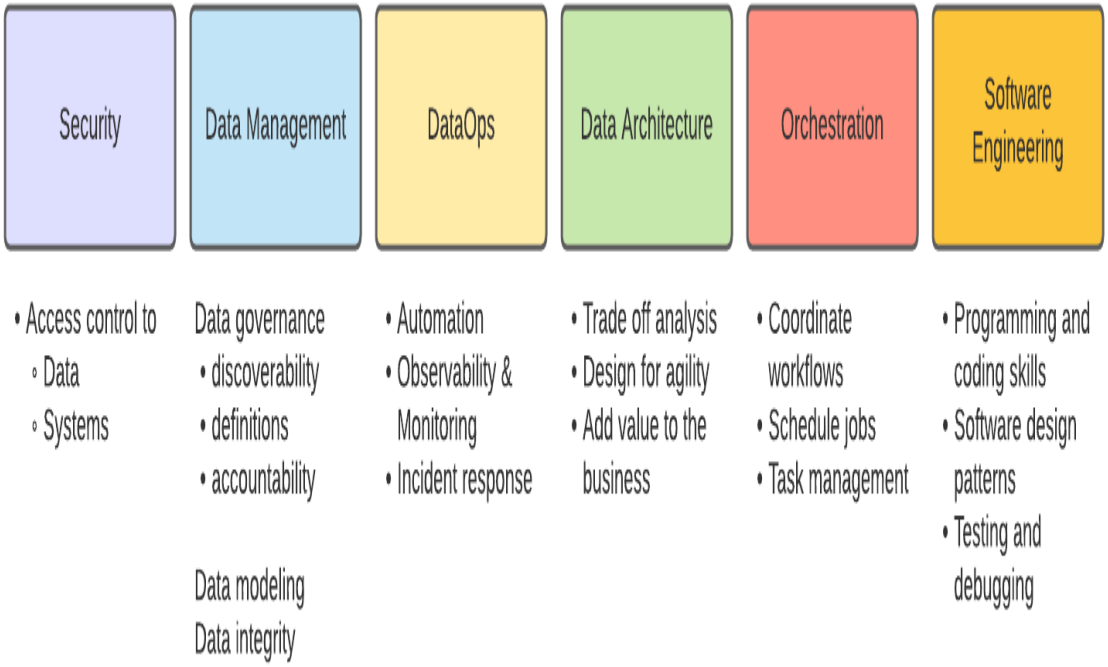


Figure 2-7. The major undercurrents of data engineering

Data Management

Data management? You're probably thinking that sounds very… corporate. "Old school" data management practices are making their way into data and machine learning 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. We sense the motivation for adopting data management is because data tools are becoming simpler to use, and a data engineer needs to manage less complexity. As a result, the data engineer is moving up the value chain toward the next rung of best practices. Data best practices once reserved for very large companies—data governance, master data management, data quality management, metadata management, etc—are now filtering down into companies of all sizes and maturity levels. As we like to say, data engineering is becoming "enterprisey". This is a great thing!

According to the DAMA-DMBOK, which we consider to be the definitive book for enterprise data management, "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 verbose, so let's look at how it ties to data engineering, specifically. 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 the 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 a vital asset 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 make sure that the organization is getting value from data and handling it appropriately.

There are quite a few facets to data management, including the following:

- Data governance, including discoverability, security, and accountability[1]
- 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."[2]

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 value generated from data. It will also (hopefully) keep a company out of headlines for questionable or downright reckless data practices.

Think of the typical example where data governance is done poorly. A business analyst gets a request for a report, but doesn't know what data to use to answer the question. He or she may spend hours digging through dozens of tables in a transactional database, taking wild guesses at what fields might be useful. The analyst compiles a report that is "directionally correct," 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. In the end, the integrity of the analyst—and all data in the company's systems—is called into question. The company is confused about its performance, making business planning next to impossible.

Data governance is a foundation for data-driven business practices and a mission-critical part of the data engineering lifecycle. When data governance is practiced well, people, processes, and technologies are all aligned to treat data as a key driver of the business. And, if data issues occur, they are promptly handled and things carry on.

The core categories of data governance are discoverability, security, and accountability.[3] Within these core categories are subcategories, such as data quality, metadata, privacy, and much more. 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 ("Golden source"), how it relates to other data, and what the data means.

There are some key components to data discoverability, including metadata management and master data management. Let's briefly describe these components.

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: auto-generated and human generated. Modern data engineering revolves around automation, but too often, metadata collection is still a manual, error prone process.

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 is to use an internally-led effort where metadata collection is crowdsourced by various stakeholders within the organization. These data management tools will be 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 only as good as their connectors to data systems, and their ability to share metadata with each other. 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 their various blog posts on data tools, particularly their original Data Portal concept.[4] Such tools should provide a place to disclose data owners, data consumers, domain experts, etc. Documentation and internal wiki tools provide a key foundation for metadata management, but these tools should also integrate with automated data cataloging as mentioned above. 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 all kinds of useful ways. Metadata becomes a foundation for designing pipelines and managing data throughout the lifecycle.

DMBOK identifies several 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

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

A data engineer uses business metadata to answer non-technical 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? To use the correct data, a data engineer would 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

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 mappings, pipeline workflows, and much more. 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 the 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 that is stored in a system such as a database, a data warehouse, a data lake, or a file system; it is one of the key differentiators across different types of storage systems. Object

stores, for example, don't manage schema metadata—instead, this must be managed in a system like the Hive Metastore. On the other hand, cloud data warehouses manage schema metadata for engineers and users.

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

Operational metadata

Operational metadata describes the operational results of various systems and includes statistics about processes, job ids, application runtime logs, data used in a process, error logs, etc. 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

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, internal calendar standards. Note that much of reference data is fully managed internally, but items such as geographic codes might come from external standard references. Reference data is essentially a standard for interpreting other data, so if it changes at all, this change happens slowly over time.

Security

People and organizational structure are always the biggest security vulnerabilities in any company. When we hear about major security breaches in the media, it quite often turns out that someone in the company ignored basic precautions, fell victim to a phishing attack, or otherwise

acted in an irresponsible manner. As such, the first line of defense for data security is to create a culture of security that pervades the organization. All individuals who have access to data must understand their responsibility in protecting the sensitive data of the company and its customers.

Security must be top of mind for data engineers, and those who ignore it do so at their peril. Data engineers must understand both data and access security, at all times exercising the principle of least privilege. The principle of least privilege[5] means giving a user or system access only to the data and resources that are essential to perform an intended function. A common anti-pattern we see with data engineers with little security experience is to give admin access to all users. Please avoid this! 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 files that are visible with standard user access. In a database, don't use the superuser role when you're just querying tables visible with a lesser role. Imposing the principle of least privilege on ourselves can prevent a lot of accidental damage, and also keeps you into a security-first mindset.

Data security is also about timing—providing data access to only the people and systems that need to access it, and *only for the duration necessary to perform their work*. Data should be protected at all times from unwanted visibility—both in flight and at rest—using techniques such as encryption, tokenization, data masking, obfuscation or 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, both for 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'll highlight areas where security should be top of mind in the data engineering lifecycle.

Data accountability

Data accountability means assigning an individual to be responsible for governing some portion of data. The responsible person then coordinates the governance activities of other stakeholders. Fundamentally, it is extremely difficult to manage data quality if no one is accountable for the data in question.

Note that people accountable for data need not be data engineers. In fact, the accountable person might be a software engineer, 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 a variety of 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 different systems. For the purposes of 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 fastidious, but in fact, can have significant implications for 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 is responsible for ensuring data quality across the entire data engineering lifecycle. This involves performing such tasks as data quality tests, ensuring data conformance to schema expectations, data completeness and precision, and much more.

According to *Data Governance: The Definitive Guide,* data quality is defined by three main characteristics—accuracy, completeness, timeliness:

Accuracy

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

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, when dealing with web event data, how do we think about bots and web scrapers? If we intend to analyze the customer journey, then we must have a process that lets us separate human 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 (see further reading), the authors give the example of an offline video platform that displays ads. The platform downloads video and ads while a connection is present, allows the user to watch these while offline, 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 is a problem that 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.

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

MASTER DATA MANAGEMENT (MDM)

Master data is data about business entities such as employees, customers, products, locations, etc. As organizations grow larger and more complex through organic growth and acquisitions, and as they collaborate with other businesses, maintaining a consistent picture of entities and identities becomes more and more challenging. MDM isn't always strictly under the purview of data engineers, but they must always be aware of it; if they don't own MDM, they will still collaborate on MDM initiatives.

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. Master data management is a business operations process that is facilitated by building and deploying technology tools. For example, a master data management team might determine a standard format for addresses, 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 divisions of the company.

Master data management reaches across the full data cycle into operational databases. As such, it may fall directly under the purview of data engineering, but is often the assigned responsibility of a dedicated team that works across the organization.

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. Where we traditionally think of data modeling as a problem for 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 due to 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 ranging from flat structured relational records to raw unstructured text. We'll discuss these transformation patterns in greater detail in Chapter 8.

With the vast 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 WORN (write once, read never) access pattern or refer to a 'data swamp'. Data engineers need to understand modeling best practices, and also 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, as well as what data it depends upon.

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

Though data lineage has been around for a long time in larger companies with strict compliance standards for tracking its data, it's now being more widely adopted in smaller companies as data management becomes mainstream. A data engineer should be aware of data lineage, as it's becoming a key piece both for debugging data workflows, as well as tracing data that may need to be removed wherever it resides.

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 towards a heterogeneous cloud environment where a variety of 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. A 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, then export the results to S3 again where it can be consumed by Spark.

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 increased dramatically. 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 below.

Data lifecycle management

The advent of data lakes encouraged organizations to ignore this data archival and destruction. Why discard data when you can simply add more storage and archive it eternally? 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 pay-as-you-go storage costs, instead of large block capex 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 such as Amazon Glacier and Google Cloud Coldline Storage 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 GDPR and CCPA now require data engineers to actively manage data destruction, with consideration to the "right to be forgotten". Data engineers must be aware of what consumer data they retain and must have procedures in place 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 now 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

> *Ethical behavior is doing the right thing when no one else is watching.*
>
> —Aldo Leopold

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 the ethical and privacy implications of data were once considered a 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 someday, everyone will be watching. 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 scrutiny—and penalties for failing to comply—is only growing for data. Make sure your data assets are compliant with a growing number of data regulations, such as GDPR and CCPA. At the time of this writing, the EU is working on a GDPR for AI. We don't expect any slowdown in the popularity of ethics, privacy, or resultant regulation, so please take this seriously.

Because ethics and privacy are top of mind, we will have some tips to ensure you're baking ethics and privacy into the data engineering lifecycle.

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

While Nick's definition of the data lifecycle is somewhat different from ours, we fundamentally agree with his idea. Not only is orchestration a central DataOps process, but orchestration systems are a critical part of the engineering and deployment flow for data jobs, just as tools like Jenkins are critical to the continuous delivery and deployment of software.

So, what is orchestration? Orchestration is the process of coordinating many jobs to run as quickly and efficiently as possible on a scheduled cadence. People often mistakenly refer to orchestration tools, like Apache Airflow, as *schedulers*. This isn't quite accurate, and there are key differences. A pure scheduler, such as cron, is aware only of time. An orchestration engine, on the other hand, builds in metadata on job dependencies, generally in the form of a DAG (directed acyclic graph). The DAG can be run once, or be

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 to run new jobs any time 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. Commonly, you might set an expected completion time for overnight daily data pipelines, perhaps 10 am. If jobs are not done by this time, alerts go out to data engineers and data consumers.

Orchestration systems typically also build in 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 a variety of 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 Data Swarm for internal use in the late 2000s; this inspired popular tools such as Airflow, which Airbnb open-sourced in 2014.

Airflow was open source from its inception, a key strength to its strong adoption. The fact that it was written in Python made it highly extensible to almost any use case imaginable. While there are many other interesting

open-source orchestration projects 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 the time of 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 at the moment are Prefect and Dagster, which aim to improve the portability and testability of DAGs to allow engineers to more easily move from local development to production. There's also Argo, which is built around Kubernetes primitives, and Metaflow, an open-source project out of Netflix that aims to improve data science orchestration. Which framework ultimately wins the mindshare for orchestration is still to be determined.

We also want to mention one interesting non-open source orchestration platform, Datacoral. Datacoral has introduced the idea of auto-orchestration through data definitions. That is, one can define a table in Datacoral by using a SQL statement; the orchestrator analyzes the source tables in the query to determine upstream dependencies, then triggers a table refresh as these dependencies are met. We believe that auto-orchestration will become a standard feature of orchestration engines in the future.

We also 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 will talk more about these developments in the chapter on transformation.

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 how 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 to provide good user experiences to downstream users, as well as the business logic, quality, and metrics that will create excellent data products not just for immediate downstream users, but for users across the business.

Like DevOps, DataOps borrows much from lean manufacturing and supply chain, mixing people, processes, and technology to reduce time to value.

As Data Kitchen (the experts in DataOps) describes it, "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"[6]

Lean practices (such as lead time reduction, 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 are a cultural habit, and 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. The reverse is rarely true.

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

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

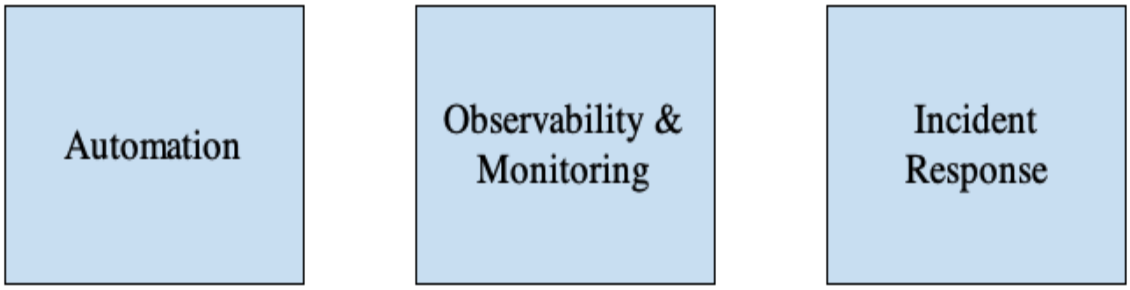


Figure 2-8. The three pillars of DataOps

Automation

Automation enables reliability and consistency in the DataOps process and allows data engineers to quickly deploy new product features and functionality, as well as changes to existing parts of the workflow. 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 ("X 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 much 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 data 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 out-of-date 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 itself presents an operational burden, but this eventually outweighs the operational difficulties of cron jobs. Engineers will gradually migrate their cron jobs to Airflow jobs. Now, dependencies are checked before jobs run. More 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. For instance, eventually, a data scientist deploys a broken DAG, bringing down the Airflow webserver and leaving the data team operationally blind. After enough such headaches, the data engineering team realizes 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 goal-oriented change. At each stage of our automation journey, there are opportunities for operational improvement. Even at the high level of maturity that we’ve described here, there is further room for improvement. Engineers might choose to 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, only to discover a substantial time later that the data was wrong. 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, companies may be led to financial or technical disaster by bad data.

Another horror story is 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, different stakeholders lose trust in the capabilities of the data system and start their quasi-data team. The result is a ton of different unstable systems, 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, tracing are all critical to getting ahead of any problems that will occur along the data engineering lifecycle. We recommend you incorporate statistical process control (SPC) to understand whether events being monitored are out of line, and which incidents are worth responding to.

We'll 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 quickly ship new data products. But mistakes will inevitably happen. A system may have downtime, a new data model may break downstream reports, a machine learning model may become stale and provide bad predictions, and countless other things may happen that interrupt the data engineering lifecycle. Incident response is about using the automation and observability capabilities mentioned above to rapidly identify root causes of an incident, and resolve the incident as reliably and quickly as possible.

Incident response isn't just about technology and tools, though they are immensely useful. It's also about open and blameless communication, both on the data engineering team, and across the organization. As Werner Vogels is famous for saying, "Everything breaks all the time." Data engineers must be prepared for disaster, and ready to respond as swiftly and efficiently as possible. Even better, data engineers should proactively find issues before the business reports them. Trust takes a long time to build and can be lost in minutes. In the end, incident response is as much about retroactively responding to incidents as proactively addressing them before they happen.

DataOps summary

At this point, DataOps is still a work in progress. Even so, practitioners have done a good job of adapting the principles of DevOps to the data domain and mapping out an initial vision through the DataOps Manifesto and other resources. Data engineers would do well to implement DataOps practices a high priority in all of their work. The upfront 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. That said, there's a recent movement 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. Because the landscape is changing so fast, we'll take a wait-and-see approach before we suggest specific 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 the long-term data needs and strategy of an organization. Because the data requirements of an organization will likely change very rapidly, and new tools and practices seem to arrive on a near-daily basis, data engineers must understand good data architecture. We will cover data architecture in-depth in Chapter 3, but we want to highlight how 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 tradeoffs with design patterns, technologies, and tools in the areas of 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, as well as provide architectural feedback. Without a dedicated data architect, however, a data engineer will realistically be entrusted to be the default data architect. We will deep dive into "good" data architecture in Chapter 3.

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 low-level frameworks and wrote map-reduce 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. Data warehouses support powerful transformations using SQL semantics. Tools like Spark have become more user-friendly over time, transitioning away from RDD-based coding towards easy-to-use dataframes. Despite this abstraction, software engineering is still a central theme of modern 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, smoke testing, and others. Especially with the growing number of tools and modularization, and the automation involved with DataOps, data engineers need to ensure their systems work end-to-end.

Development of open source frameworks

Many data engineers are heavily involved in the development of open-source frameworks. They adopt these frameworks to solve specific problems in the data engineering lifecycle, then continue to develop the framework code to improve the tools for their use cases and to make a contribution back to the community.

In the big data era, we saw a Cambrian explosion of data processing frameworks inside of the Hadoop ecosystem. These tools were primarily focused on the transformation and serving parts of the data engineering lifecycle. Data engineering tool speciation has not ceased or even 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, Apache Airflow dominated the orchestration space from 2015 on. Now, a new batch of open source competitors (Prefect, Dagster, Metaflow, etc) has sprung up to fix perceived limitations of Airflow, providing better metadata handling, portability, dependency management, etc.

Before data engineers begin engineering new internal tools, they would do well to survey the landscape of projects. Keep an eye on the total cost of ownership (TCO) and opportunity cost associated with implementing a tool. There is a good chance that some open source project already exists to address the problem that they're looking to solve, and they would do well to collaborate on an existing project 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 a host of 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 a real-time environment, and more complex software engineering is required. Engineers must also write code to apply a variety of *windowing* methods. Windowing allows real-time systems to calculate useful metrics such as trailing statistics; there are a variety of techniques for breaking data into windows, all with subtly different nuances. 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 (Apache Spark, Beam, or Pulsar) for analyzing streams for 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 precipitously as companies have migrated to managed big data systems (Databricks, 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 (Terraform, Ansible...) and cloud platform-specific (AWS Cloudformation, Google Cloud Deployment Manager...) frameworks allow automated deployment of infrastructure based on a set of specifications. Many of these frameworks can manage services as well as infrastructure. For example, we can use CloudFormation to spin up Amazon Redshift, or Deployment Manager to start Google Cloud Composer (managed Airflow). There is also a notion of infrastructure as code with containers and Kubernetes, using tools like Helm.

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

Pipelines as code

Pipelines as code are the core concept of modern orchestration systems, which touch every stage of the data engineering lifecycle. Data engineers use code (typically Python) to declare data tasks and dependencies between 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 requires them to solve problems outside the boundaries of their chosen tools, and to write custom code. Even when using frameworks like Fivetran, Airbyte, or Singer, data engineers will encounter data sources without existing connectors and need to write something custom. They should be proficient enough in software engineering to understand APIs, pull and transform data, handle exceptions, etc.

Conclusion

Most discussions we've seen in the past about data engineering involve technologies but miss the bigger picture of the 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 modern data engineering.

We break the data engineering lifecycle into the following stages:

- Generation: source systems
- Ingestion
- Storage
- Transformation
- Serving

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:

- Data management
- DataOps

- Data architecture
- Software engineering
- Orchestration

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. In the next chapter, we'll discuss how these elements impact the design of good architecture. Subsequently, we've devoted a chapter to each of the stages of the data engineering lifecycle.

---

1 Evren Eryurek, Uri Gilad, Valliappa Lakshmanan, Anita Kibunguchy-Grant, and Jessi Ashdown, *Data Governance: The Definitive Guide* (O'Reilly), https://learning.oreilly.com/library/view/data-governance-the/9781492063483/.

2 Eryurek et al.

3 Eryurek et al.

4 https://medium.com/airbnb-engineering/democratizing-data-at-airbnb-852d76c51770

5 https://en.wikipedia.org/wiki/Principle_of_least_privilege

6 https://datakitchen.io/what-is-dataops/

Chapter 3. Choosing Technologies Across the Data Engineering Lifecycle

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the fourth chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at book_feedback@ternarydata.com.

Modern data engineering suffers from an embarrassment of riches. Data technologies to solve nearly any problem are available as turnkey offerings consumable in almost every way—open-source, managed open source, proprietary software, proprietary service, etc. 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 life cycle 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 best serve data applications and users.

In the last chapter, we discussed “good” data architecture, and why it matters. We now explain how to choose the right technologies to serve this architecture. Data engineers must choose “good” technologies to make the best possible data product. We feel the criteria to choose a “good” data technology is simple: *does it add value to a data product and the broader business?*

A lot of people confuse architecture and tools. *Architecture is strategic; tools are tactical.* We sometimes hear, “our data architecture are tools X, Y, and Z”. This is the wrong way to think about architecture. Architecture is the top-level design, roadmap, and blueprint of data systems that satisfy the strategic aims for the business. Architecture is the “What”, “Why” and “When”. Tools are used to make the architecture a reality; tools are the “How”.

We often see teams going “off the rails” and choosing technologies before mapping out an architecture. The reasons for this are varied—shiny object syndrome, resume-driven development, a lack of expertise in architecture—but in practice, this prioritization of technology often means that they don’t choose architecture, but instead cobble together a kind of Rube Goldberg machine of technologies. This is exactly backward. 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 an architecture blueprint in hand. Below are some considerations for choosing data technologies across the data engineering lifecycle:

- Cost optimization and adding value to the business
- Friction to deliver
- Team ability and maturity
- Speed to market
- Today versus the future: immutable versus transitory technologies
- Location: cloud, on-premises, hybrid-cloud, multi-cloud

- Build versus buy
- Monolith versus modular
- Serverless versus servers
- The undercurrents of the data engineering lifecycle, and how they impact choosing technologies

Cost Optimization: Total Cost of Ownership and Opportunity Cost

In a perfect world, we'd get to experiment with all of the latest, coolest technologies with no consideration of cost, time investment, or value added for the business. In reality, budgets and time are finite, and cost is a major constraint for choosing the right data architectures and technologies. Your organization expects a positive return on investment (ROI) from your data projects, so it's critical that you understand the basic costs you can control. Technology is a major cost driver, so your technology choices and management strategies will have a significant impact on your budget. We look at costs through two main lenses—total cost of ownership and opportunity cost.

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 are the costs you can directly attribute to an initiative, such as 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* has a big impact on a budget's bottom line. These expenses fall into two big groups—capital expense and operational expense.

Capital expenses, also known as capex, require an upfront investment. Payment is required *today*. Before the cloud, companies would typically purchase hardware and software upfront through large acquisition contracts, and house this infrastructure in huge server rooms, or pay to house their servers in a colocation facility. These upfront investments in hardware and software—commonly in the hundreds of thousands to millions of dollars, or higher—would be treated as assets, and slowly depreciate over time. From a budget perspective, capital would need to be available to pay for the entire purchase upfront or funding secured through a loan vehicle. In addition, there's the extra cost of maintaining the hardware and software over the lifetime of those assets. Capex is generally reserved for long-term investments, where there's a well-thought-out plan for achieving a positive ROI on the effort and expense put forth.

Operational expenses, known as opex, are almost the opposite of capex. 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 has the added benefit of potentially being closer to a direct cost, making it easier to attribute to a data project.

Until recently, opex simply wasn't an option for large data projects—data warehouse systems required multimillion dollar contracts. This has all changed with the advent of the cloud, where data platform services allow engineers to pay on a consumption based model. In general, we find that opex allows for a far greater ability for engineering teams to choose both their software and hardware. Cloud based services allow data engineers to iterate quickly with different software and technology configurations, often very 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 technologies.*

Opportunity Cost

Data engineers often fail to evaluate opportunity cost when they undertake a new project. In our opinion, this is a massive blind spot. *Opportunity cost* is the cost of choosing one thing at the expense of something else. As it relates to data engineering, if you choose Data Stack A, that means you've chosen the benefits of Data Stack A over all other options, effectively excluding Data Stack B through Z. You're now committed to Data Stack A, and everything it entails—the team to support it, training, setup, maintenance, and so forth. But in an incredibly fast-moving field like data, what happens when Data Stack A becomes obsolete? Can you still move to Data Stack B?

A big question in a world where newer and better technologies arrive on the scene at an ever-faster rate—how quickly and cheaply can you move to something newer and better? 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, while they seemed good at the time, are either not flexible for future growth, or are simply obsolete. *Inflexible data technologies are a lot like bear traps. They're easy to get into and extremely painful to escape.* Don't let this happen to you.

Today Versus the Future: Immutable Versus Transitory Technologies

Where are you today? What are your goals for the future? Your answers to these questions should inform the decisions you make about your architecture, and thus the technologies used within that architecture. Too often, we see data engineers and architects scoping and building for "the future" whose date and specific needs are not well defined. These intentions are noble but often lead to over-architecting and over-engineering. It often happens that tooling chosen *for the future* is 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; choose the best technology for the moment and near future, but in a way that supports future unknowns and evolution. This is done by understanding what is likely to change, and what tends to stay the same.

There are 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 AWS 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 for the foreseeable future.

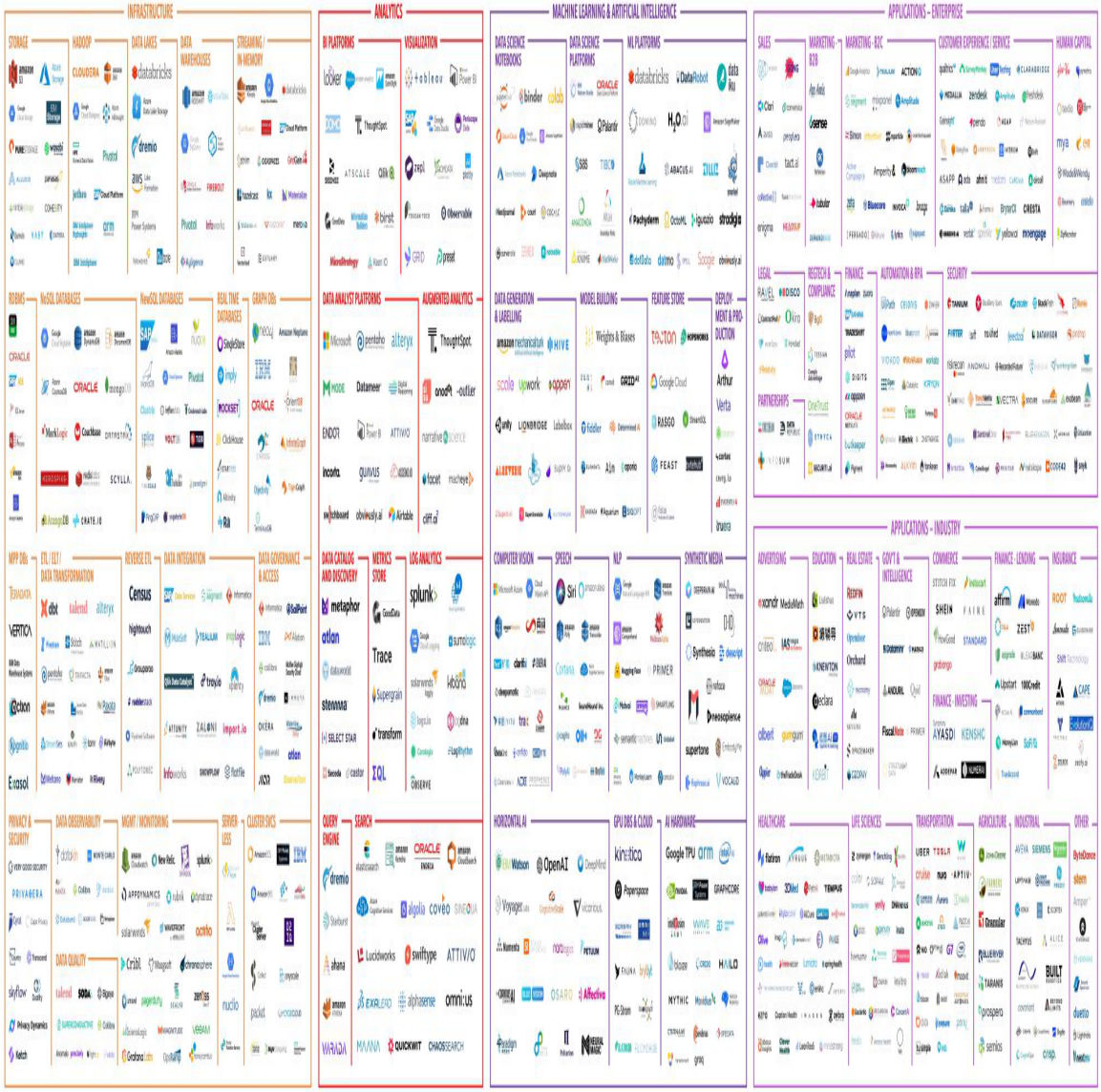
For languages, SQL and Bash have been around for many decades, and we don't see them disappearing anytime soon. Immutable technologies benefit from the Lindy Effect, which says the longer a technology has been established, the longer it will continue to be used. Think, for example, of relational databases, C[1], or the x86 processor architecture. We suggest applying the Lindy Effect as a litmus test to determine if a technology has immutable potential.

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 of this. How many Javascript frontend frameworks have come and gone between 2010 and 2020? Backbone, Ember and Knockout were popular in the early 2010's, AngularJS in the mid 2010's, and React and Vue have massive mindshare today. What's the popular frontend framework 3 years from now? Who knows.

On the data front, new well-funded entrants and open source projects arrive on the scene every day. Every vendor will say their product is going to

change the industry and “make the world a better place.”[2] Sadly, the vast majority of these companies and projects won’t ultimately get traction, and will fade into obscurity. Top VC’s are making big-money bets, knowing that most of their data tooling investments will fail. How can you possibly know what 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 machine learning, AI, and data (MAD) landscape[3] (Figure 3-1).

MACHINE LEARNING, ARTIFICIAL INTELLIGENCE, AND DATA (MAD) LANDSCAPE 2021



© Matt Turck (@mattturck), John Wu (@john_d_wu) & FirstMark (@firstmarkcap)

mattturck.com/data2021

FIRSTMARK
EARLY STAGE VENTURE CAPITAL

Figure 3-1. Matt Turck's MAD data landscape

Even relatively successful technologies often fade into obscurity quickly after a few years of rapid adoption, a victim of their success. For instance, Hive was met with rapid uptake because it allowed both analysts and engineers to query massive data sets without coding complex MapReduce jobs by hand. Inspired by the success of Hive, but wishing to improve on its shortcomings, engineers developed SparkSQL, Presto, and other technologies. Hive now appears primarily in legacy deployments. This is the general cycle of data technologies—incumbents reign for a time, then get disrupted and replaced by new technologies that also reign for a time, and so on.

Our Advice

Given the rapid pace of tooling and best practice changes, we suggest evaluating tools on a two-year basis (Figure 3-2). Whenever possible, find the immutable technologies along the data engineering lifecycle, and use those as your base. Build transitory tools around the immutables.

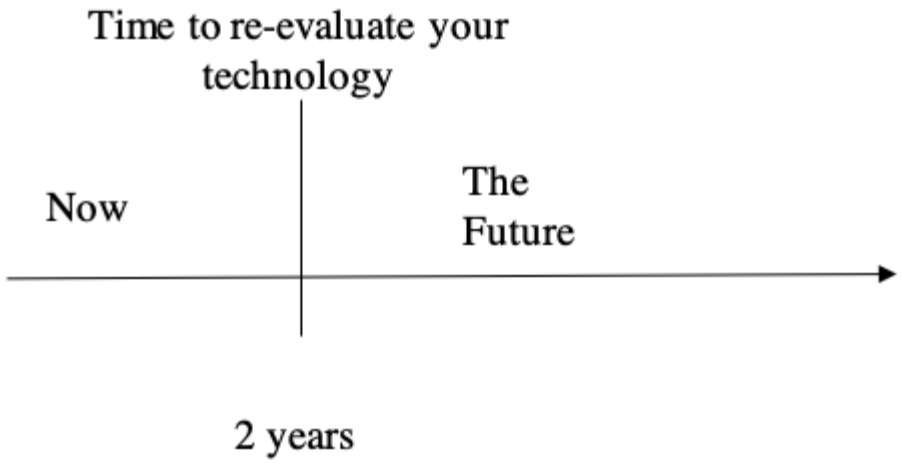


Figure 3-2. Use a two-year time horizon to reevaluate your technology choices

Location: On-Premises, Cloud, Hybrid, Multi-Cloud, and More

Companies now have numerous options when they decide where to run their technology stacks. A slow shift toward the cloud in the last decade now culminates in a veritable stampede of companies spinning up workloads on AWS, Azure, and GCP. Many CTOs now 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 nimble competition; on the other hand, a poorly planned cloud migration could lead to poorly functioning technology and catastrophic costs. Let's look at the main places to run your technology stack—on-premises, the cloud, hybrid-cloud, and multi-cloud.

On-premises

While new startups are increasingly born in the cloud, on-premises systems are still very much the default for established companies. Essentially, this means that they own their own hardware, which may live in data centers that they own, or in leased colocation space. In either case, companies are operationally responsible for their hardware and the software that runs on it. As 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 also 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 reasonable performance for peak load and large jobs without overbuying and spending excessively.

On the one hand, established companies have established operational practices that have served them well for some time. If a company that relies on information technology has been in business for some time, this means that they have managed to juggle the cost and personnel requirements of

running their own hardware, managing software environments, deploying code from dev teams, running databases, and big data systems, etc.

On the other hand, established companies see their younger, more nimble 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 to huge clusters for massive data jobs, or turn on temporary server capacity for Black Friday.

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 significant venture capital dollars. Plus, many large incumbent companies are also modernizing and moving their workloads to the cloud. 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 own hardware running on-premises, or it could involve a full migration to the cloud, as discussed below.

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 (AWS, Azure, Google Cloud, etc). These resources can often be reserved on an extremely short term basis—virtual machines typically spin up in less than a minute, and subsequent usage is billed in per second increments. This allows cloud users to dynamically scale resources in a way that is 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 who are tight on budget and time.

The early era of cloud was dominated by Infrastructure as a Service (IaaS) offerings, i.e., products such as virtual machines, virtual disks, etc. that are essentially rented slices of hardware. Slowly, we've seen a shift toward Platform as a Service (PaaS), while Software as a Service (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 (Amazon RDS, Google Cloud SQL), managed streaming platforms (Amazon Kinesis and SQS), managed Kubernetes (Google Kubernetes Engine, Azure AKS), etc. PaaS services can allow engineers to ignore the operational details of managing individual machines, and deploying and configuring frameworks across distributed systems. They often provide turnkey access to complex systems with simple scaling and minimal operational overhead.

SaaS offerings move one additional step up the ladder of abstraction. Typically, Software as a Service provides a fully functioning enterprise software platform with very little operational management. Examples of SaaS include Salesforce, Google Workspace, Office 365, Zoom, Fivetran, etc. 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, CRM systems, and so on. It can be difficult to define the boundary between PaaS and SaaS offerings; a CRM platform may be an important component of backend application systems, and Fivetran functions as a critical piece of data infrastructure.

We also mention serverless in this section, which is increasingly important in PaaS and SaaS offerings. Serverless products generally offer fully automated scaling from 0 up to extremely high usage rates. They are billed on a pay-as-you-go basis and allow engineers to operate with no operational awareness of underlying servers. Many people quibble with the term serverless—after all, the code must run somewhere. In practice, serverless platforms usually run on highly distributed multi-tenant infrastructure over many servers. AWS Lambda and Google BigQuery are best in class

serverless offerings in the cloud functions and cloud data warehouse domains respectively.

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 (i.e. retail business coping with Black Friday load) and web traffic load spikes. The COVID 19 Global Pandemic has also been a major driver of cloud adoption, as numerous businesses have had to cope with substantially increased web traffic, and internal IT infrastructure has groaned under the load of mass 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. Enterprises that migrate to the cloud often make major deployment errors by not appropriately adapting their practices to the cloud pricing model. Let's take a bit of a detour into cloud economics.

A brief detour on cloud economics

To understand how to use cloud services efficiently, you need to know how clouds make money. This is actually an extremely complex concept and one on which cloud providers offer little transparency. Our guidance in this section is largely anecdotal, and won't cover the full scale of what you'll likely encounter in your own experience.

Cloud services and credit default swaps

Recall that credit default swaps rose to infamy after the 2007 global financial crisis. Roughly speaking, a credit default swap was a mechanism for selling different tiers of risk attached to an asset (i.e. a mortgage.) It is not our intention to understand this idea in any detail, but rather to present an analogy wherein many cloud services are similar to financial derivatives; cloud providers not only slice hardware assets into small pieces through virtualization, but they also sell these pieces with varying technical characteristics and risks attached. While providers are extremely tight-

lipped about many details of their internal systems, there are massive opportunities for optimization and scaling by understanding cloud pricing and exchanging notes with other users.

Let's look at the example of cloud archival storage. Google Cloud Platform openly admits that their archival class storage runs on the same clusters as standard cloud storage, yet the price per GB per month of archival storage is roughly 1/17th that of standard storage. How is this possible?

Here, we are giving our best-educated guess. When we purchase cloud storage, we think in terms of cost per GB, but each disk in a storage cluster has three different assets that cloud providers and consumers make use of. First, it has a certain storage capacity, say 10 TB. Second, it supports a certain number of IOPs (input/output operations per second), say 100. Third, disks support a certain maximum bandwidth, which is the maximum read speed for optimally organized files. A magnetic drive might be capable of reading at 200 MB/s.

For a cloud provider, any of these limits (IOPs, storage capacity, bandwidth) is a potential bottleneck. For instance, the cloud provider might have a disk storing 3 TB of data, but hitting maximum IOPS. There's an alternative to leaving the remaining 7 TB empty: 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.

Similar considerations apply to nearly any cloud service. To provide another example, compute is no longer a simple commodity in the cloud as it is with on-premises hardware. Rather than simply 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 (AWS

Lambda) or can be arbitrarily interrupted when capacity is needed elsewhere (Google Compute Engine Interruptible Instances).

Cloud ≠ on-premises

This heading may seem like a silly tautology; in reality, the belief that cloud services are just like familiar on-premises servers is an extremely common cognitive error that plagues cloud migrations and leads to horrifying bills.

This cognitive error demonstrates a broader issue in tech that we refer to as *the curse of familiarity*. Generally, any technology product is designed to look like something familiar to facilitate ease of use and adoption; but, any technology product has subtleties and wrinkles that users must learn to identify and work with/around after the initial adoption phase.

Moving on-premises servers one by one to virtual machines in the cloud—known as simple lift and shift—is a perfectly reasonable strategy for the initial phase of cloud migration, especially where 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 to understand and optimize for 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. Or take advantage of AWS spot instances or AWS Lambda to realize discounts through more ephemeral, less durable workloads.

Data engineers can also realize new value in the cloud not by simply saving money, but by doing things that were simply not possible in their on-premises environment. The cloud gives data engineers the opportunity to spin up massive compute clusters quickly in order to run complex transformations, at scales that are 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. For cloud platforms, this means that getting data onto the platform is cheap or free, but getting data out can be extremely expensive. Be aware of data egress fees and their long-term impacts on your business before you get 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 and 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 maintain some workloads outside the cloud indefinitely.

There are a variety of reasons to consider a hybrid cloud model. Organizations may believe that they have achieved operational excellence in certain areas, for instance with their application stack and associated hardware. Thus, they may choose to migrate only specific workloads where they see immediate benefits in the cloud environment. They might choose to migrate their Hadoop stack to ephemeral cloud clusters, reducing the operational burden of managing software for the data engineering team and allowing rapid scaling for large data jobs.

This pattern of putting analytics in the cloud is particularly attractive because data flows primarily in one direction, minimizing data egress costs (Figure 3-3). 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 might be pushed back for deploying models to applications, reverse ETL, etc.

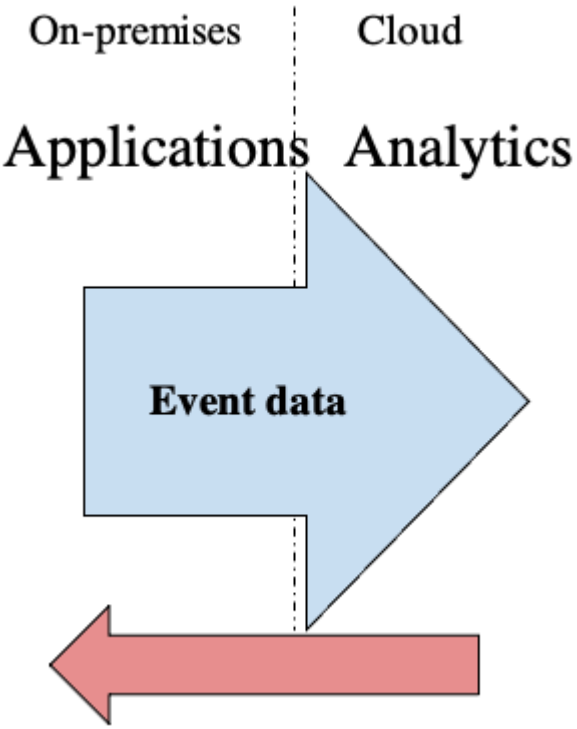


Figure 3-3. Hybrid cloud data flow model that minimizes egress costs.

There's also a new generation of managed hybrid cloud service offerings, including Google Anthos and AWS Outposts. These services allow customers to locate cloud-managed servers in their data centers. Thus, they can take advantage of auto-scaling with the EC2 Management Console, or spin up Google Cloud Composer, while still realizing low latency and network access control of having servers within their own walls. These services integrate seamlessly with the cloud offerings of AWS and GCP respectively, allowing customers to burst data and application workloads to the cloud as the need arises.

Multi-cloud

Multi-cloud simply refers to the practice of deploying workloads to multiple clouds, as opposed to deliberately choosing and using only a single cloud for the sake of simplicity. Companies may have a variety of motivations for multi-cloud deployments. SAAS platforms often wish to offer their services close to existing customer cloud workloads. Snowflake

and Databricks are two organizations that offer multi-cloud services 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 multi-cloud approach is to take advantage of the best services across several clouds. Customers might want to handle their Google Ads and Analytics data on GCP and deploy Kubernetes through Google GKE. They might adopt Azure specifically for Microsoft workloads. And AWS has several best-in-class services (i.e. AWS Lambda) and enjoys huge mindshare, making it relatively easy to hire AWS proficient engineers.

There are several disadvantages to a multi-cloud methodology. As we just mentioned, data egress costs and networking bottlenecks are critical. And going multi-cloud can introduce significant complexity. Companies must now manage a dizzying array of services across several clouds, and cross-cloud integration and security present a huge challenge. Among other things, it may become necessary to integrate virtual networks across several clouds.

A new generation of "cloud of clouds" services aims to facilitate multi-cloud with reduced complexity by not only offering services across clouds, but seamlessly replicating data between clouds, or managing workloads on several clouds through a single pane of glass. A Snowflake account runs in a single cloud region, but customers can readily spin up accounts in GCP, AWS, or Azure; Snowflake provides simple scheduled data replication between these accounts. In addition, the Snowflake interface is essentially the same in all of these accounts, removing the training burden associated with switching between cloud-native services such as Amazon Redshift and Azure Synapse.

Google BigQuery Omni will deliver similar functionality once it goes into general availability, extending the GCP native experience to multiple clouds. In fact, we're seeing a growing trend of "cloud of clouds" services offered directly by the public clouds. Google Anthos allows users to run

workloads on AWS/Azure, and Amazon EKS Anywhere offers similar functionality for Azure/GCP.

The "cloud of clouds" space is evolving rapidly; within a few years of this book's publication, there will be far more such services available, and data engineers and architects would do well to maintain awareness of this quickly changing 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 possibly invert this paradigm. While there are not a lot of popular examples to point at right now, it's worth keeping platform decentralization in the back of your mind as you consider technologies.

Be Cautious with Repatriation Arguments

As we were writing this book, Sarah Wang and Martin Casado published an article for Andreesen Horowitz[4] that generated significant sound and fury in the tech space. This article was widely interpreted as a call for repatriation of cloud workloads to on-premises servers. In fact, they make a somewhat more subtle argument that companies should expend significant resources to control cloud spend, 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 their servers as a case study for companies considering the same move or assessing retention of existing data centers.

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 very specific 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 a vast quantity of data. The company is tight-lipped about exactly how much data they host, but they do say that it is many exabytes and that it continues to grow.

Second, Dropbox handles a vast amount of network traffic. We know that their 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)."[5] Their 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 characteristics of object and block storage. Dropbox's core competence is a differential file update system that can efficiently synchronize actively edited files between users while minimizing network and CPU usage. As such, their 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.[6]

Fourth, while Dropbox moved their core product to their own hardware, they continue to build out other workloads on AWS. In effect, this allows them to focus on building one highly tuned cloud service at extraordinary scale rather than trying to replace numerous different services; they can focus on their core competence in cloud storage and data synchronization, while offloading management of software and hardware in other areas, such as data analytics.[7]

Other frequently cited success stories where companies have built outside the cloud include Backblaze and Cloudflare, and these offer similar lessons. Backblaze began life as a personal cloud data backup product, but has since begun to offer its own object storage similar to Amazon S3. They currently

store over an exabyte of data.[8] Cloudflare claims to provide services for over 25 million internet properties, with points of presence in over 200 cities and 51 Tbps (terabits per second) of total network capacity.[9]

Netflix offers yet another useful example. Netflix 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.[10] They also run their application backend and data analytics on AWS. However, rather than using the AWS content distribution network, they have built out 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,[11] building out this critical infrastructure allowed them to cost effectively deliver high quality video to a huge customer base.

These case studies suggest that it makes sense for companies to manage their own hardware in very specific circumstances. The biggest modern success stories of companies building and maintaining hardware involve extraordinary scale (exabytes of data, Tbps of bandwidth, etc.), and narrow use cases where companies can realize a competitive advantage by engineering highly integrated hardware and software stacks. Thus, Apple might gain a competitive edge by building its own storage system for iCloud,[12] but there is less evidence to suggest that general purpose repatriation efforts will be beneficial.

Our Advice

From our perspective, we are still at the beginning of the transition to cloud, and thus the evidence and arguments around workload placement and migration will continue to evolve rapidly. Cloud itself is changing, with a shift from the IAAS (infrastructure as a service) model built around Amazon EC2 that drove the early growth of AWS, toward more managed service offerings such as AWS Glue, Google BigQuery, Snowflake, etc. We've also seen the emergence of new workload placement abstractions. On-premises services are becoming more cloud-like (Think Kubernetes, which abstracts away hardware details); And hybrid cloud services such as

Google Anthos and AWS Outposts allow customers to run fully managed services within their own walls, while also facilitating tight integration between local and remote environments. Further, the "cloud of clouds" is just beginning to take shape, fueled by third party services, and increasingly the public cloud vendors themselves.

Choose technologies for the present, but look toward the future

In sum, it is a very difficult time to plan workload placements and migrations. 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 multi-cloud or hybrid-cloud strategy unless there's a compelling reason to do so. Do you need to serve data near customers on multiple clouds? Do industry regulations require you to house certain data in your own data centers? Do you have a compelling technology need for specific services on two different clouds? If these scenarios don't apply to you, then choose a simple single cloud deployment strategy.

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 also significant lock-in. In this instance, we're really 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 do go wrong in the future.

Build Versus Buy

Build versus buy is a very old debate in technology. The argument for "build" 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 "buy" comes down to resource constraints and expertise—do you have the expertise to build a solution that's better than something already available? Either decision comes down to TCO, opportunity cost, 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. 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 try to build everything yourself.

As we often ask, "When you need new tires for your car, do you get the raw materials, build the tires from scratch, and install them yourself? Or do you go to the tire shop, buy the tires, and let a team of experts install them for you?" If you're like most people, you're probably buying tires and having someone install them. The same argument applies to build vs buy. We've seen teams who've built their own database from scratch. Upon closer inspection, a simple open-source RDBMS would have served their needs much better. 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.* In either case, there are plenty of great, modular services to choose from.

It’s worth mentioning the shifting reality of how software is adopted within companies. Whereas in the past, IT used to make most of the software purchase and adoption decisions in a very 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 (OSS)

Open source software (OSS) is a software distribution model where software—and the underlying codebase—is made available for general use, typically under certain licensing terms. Oftentimes, OSS is created and maintained by a distributed team of collaborators. Most of the time, OSS is free to use, change, and distribute, 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 who create a novel solution and choose 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.

There are two main flavors of OSS—community managed, and commercial OSS.

Community-managed OSS

OSS projects succeed when there’s a strong community and vibrant user base. Community-managed OSS is a very common path for OSS projects. With popular OSS projects, the community really opens up high rates of innovations and contributions from developers all over the world.

Some things 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.

Project management

How is the project managed? Look at Git issues and if/how they're addressed.

Team

Is there 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 Slack community that provides encouragement and support?

Contributing

Does the project encourage and accept pull requests?

Roadmap

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

Hosting and maintenance

Do you have the resources to host and maintain the OSS solution? If so, what's the TCO and opportunity cost 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 the project. You can contribute to the codebase, help fix issues, 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. It's a labor of love that sadly 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. Depending on the OSS application you're using, this may either be trivial or extremely complicated and cumbersome. Commercial vendors try to solve this management headache by hosting and managing the OSS solution for you, typically as a cloud SAAS offering. Some examples of such vendors include Databricks (Spark), Confluent (Kafka), DBT Labs (dbt), and 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 very common trend—an OSS project becomes very popular, an affiliated company raises truckloads of VC money to commercialize the OSS project, and the company scales as a fast-moving rocketship.

At this point, there are two options for a data engineer. 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 them take care of the administrative management of the COSS product.

Some things 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 lots of 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're easily able to access the initial version and subsequent releases.

Support

> Support cannot be understated, and it's sadly often opaque to the buyer. What is the support model for the product, and is there an extra cost for support? Oftentimes, vendors will sell support for an additional fee. Be sure you clearly understand the costs of obtaining support.

Releases and bug fixes

> Is the vendor transparent about their 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 tradeoffs of paying as you go versus paying upfront. Is it worth it to pay 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 they 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 that are primarily concerned with growing their customer count without the revenue to establish sound finances.

Community support

Is the company truly supporting the community version of the OSS project? How much are they contributing to the community OSS codebase? There's been controversy with certain vendors co-opting OSS projects, and subsequently providing very little value back to the community.

Proprietary Walled-Gardens

While OSS is extremely common, there is also a giant market 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 cloud-platform offerings.

Independent offerings

The data tool landscape has seen exponential growth over the last several years. Every day, it seems like there are new independent offerings for data tools, and there's no indication of a slowdown anytime soon. With the ability to raise funds from VCs who are flush with capital, these data companies can scale, and hire great teams in engineering, sales and marketing. This presents a situation where users have some great product choices in the marketplace while having to wade through endless sales and marketing clutter.

Quite often, a company selling a data tool will not release it as OSS, instead offering a black box 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. For comparison, Databricks uses the Apache Spark codebase with a proprietary management layer; Google BigQuery is a proprietary solution, not offering access to the underlying codebase.

Some things to consider with an independent offering:

Interoperability

> Make sure it inter-operates with other tools you've chosen—OSS, other independents, cloud offerings, etc. Interoperability is key, so make sure you are able to try 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 clearly 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? If you sign a contract, how much flexibility do you lose, both in terms of 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, check sites like Crunchbase

and see what their funding situation is like. Look at user reviews. Ask friends and post questions on social networks about other users' experiences with the product. Make sure you know what you're getting into.

Cloud platform service offerings

Cloud vendors develop and sell their own proprietary services for storage, databases, etc. 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. They later offered the DynamoDB service solely on AWS; it's now an extremely popular product used by companies of all sizes and maturity levels. Clouds will often bundle their products to work well together. By creating a strong integrated ecosystem, each cloud is able to create "stickiness" with its user base.

In addition to internal innovations that make their way to end-users, as we discussed earlier, clouds are also keen followers of strong OSS and independent offerings. If a cloud vendor sees traction with a particular product or project, expect that they will offer their own version. The reason is simple. Clouds make their money through consumption. More offerings in a cloud ecosystem mean a greater chance of stickiness and increased spending by customers.

Some things 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. Are you able to 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 up to focus on improving those areas where these options are insufficient. Focus on a few areas where building will add significant value or reduce friction substantially.

It's worth mentioning—don't treat internal operational overhead as a sunk cost. There's great value in upskilling your existing data team to build sophisticated systems on managed platforms rather than babysitting on-premises servers.

Always think about how a company makes money, especially the sales and customer experience people. This will generally indicate how you're treated during the sales cycle, as well as when you're a paying customer.

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 tradeoffs of a monolithic vs a modular approach.

Monolith

The monolith (Figure 3-4) 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.

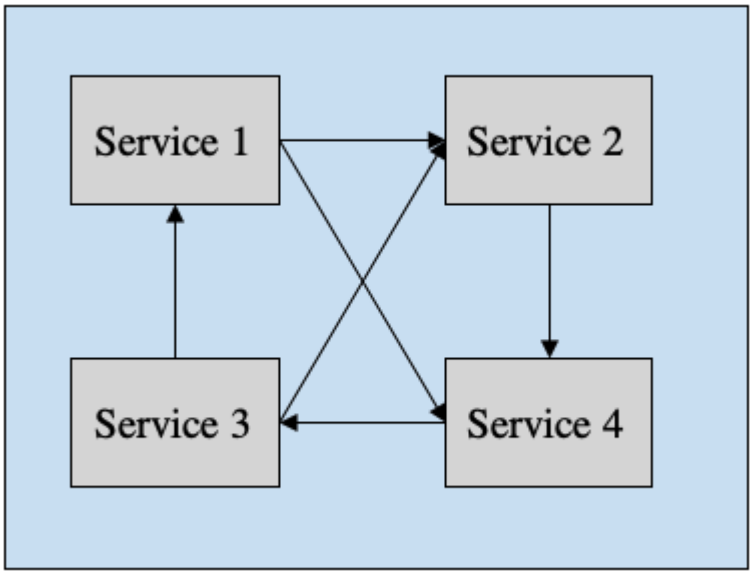


Figure 3-4. The monolith tightly couples its services

The pros of the monolith are it's easy to reason about, and there's 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 principle programming language. Monoliths are a good option if you want simplicity in reasoning about your architecture and processes.

Of course, there are cons with the monolith. For one, it's brittle. Due to the vast number of moving parts, updates and releases take longer, and tend to bake in "the kitchen sink". If there's a bug in the system (and hopefully the software's been thoroughly tested before release!), it can have a deleterious effect on 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 2 days late by default. 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 difficult to isolate the workloads of multiple users in a monolithic system. In an on-prem data warehouse, one user-defined function might consume enough CPU to slow down the system for other users. In Hadoop, Spark, and Airflow environments, conflicts between dependencies required by different users are a frequent source of headaches. (See the *distributed monolith* discussion that follows.)

Another con about monoliths—if the vendor or open source project dies, switching to a new system will be very painful. 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 3-5) is an old concept in software engineering, but modular distributed systems truly came into vogue with the rise of microservices in the 2010s. Instead of relying on a massive monolith to handle your needs, why not break apart systems and processes into their own 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 as well.

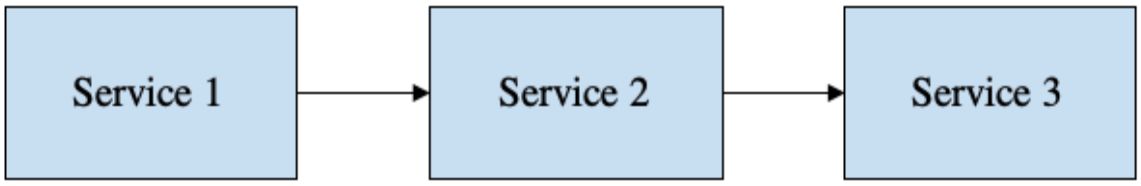


Figure 3-5. Modularity: each service is decoupled

Major tech companies have been key drivers in the microservices movement. Google engineers invented Linux containers, inspired by other operating systems such as Solaris, to allow the decomposition of applications into smaller pieces. 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 5 members. This cap also limits the complexity of a team's domain of responsibility, in particular the codebase that it can manage. Where a large, monolithic application might entail a team of 100 people, dividing developers into small teams of 5 requires that this large application be broken into small, manageable, loosely coupled pieces.

In a modular microservice environment, components are swappable; a service written in Python can be replaced with a Java service. Service customers need only worry 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. In the world of data lakes and lakehouses, data is stored in object storage in a standard format such as Parquet. Any processing tool that supports the format can read the data and write processed results back into the lake for processing by another tool. Tools such as BigQuery, Snowflake, and Redshift don't use a data lake architecture and don't make data directly accessible to external tools. However, all of these 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.

In today's data ecosystem, new technologies arrive on the scene at a dizzying rate, and most get stale and outmoded quickly. Rinse and repeat. The ability to swap out tools as technology changes are invaluable. In addition, we view data modularity as a more powerful paradigm than monolithic data engineering. It allows engineers to choose the best technology for each job, or even for each step of a 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.

In fact, it is this very problem that led us to break orchestration out as a separate undercurrent, instead of placing it under "data management." Orchestration is also important for monolithic data architectures—witness the success of 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 these tools 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 separate services to perform different tasks, but services and nodes share a common set of dependencies and/or a common codebase.

One standard example is a traditional Hadoop cluster. A standard Hadoop cluster can host a number of different frameworks simultaneously, such as Hive, Pig, Spark, etc. In addition, the cluster runs core Hadoop components: Hadoop common libraries, HDFS, Yarn, Java. In practice, a cluster generally has one version of each component installed.

Essentially, 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.

Our second example of a distributed monolith pattern is Apache Airflow. The Airflow architecture is highly decoupled and asynchronous. Interprocess communications are passed through a backend database, with a

few different services (web server, scheduler, executor) interacting. The web server and executor can be scaled horizontally to multiple copies.

The problem with this architecture is that every service needs to run the same Airflow 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. As a consequence, Google Cloud Composer (managed Airflow on Google Cloud) has a huge number of installed dependencies to handle numerous different databases, frameworks, and APIs out of the box. The pip package manager shows many dependency conflicts even in the base environment. The installed packages are essentially a house of cards. Installing additional libraries to support specific tasks sometimes works, and sometimes breaks the whole installation.

One solution to the problems of the distributed monolith is ephemeral infrastructure in a cloud setting. That is, each job gets its own temporary server or cluster, with its own dependencies installed. Each cluster remains highly monolithic, but conflicts are dramatically reduced by separating jobs. 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 separate software environments using containers. We have more to say on containers below in our section on Serverless vs Servers.

A third solution is to retain the distributed monolith, but pull functionality out into microservices. This approach has become common with Airflow; code with complex dependency requirements is moved into external containers or cloud functions to keep the core environment simple.

Our Advice

While monoliths are certainly attractive due to ease of understanding and reduced complexity, this comes at a big 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:

Interoperability.

Architect for sharing and interoperability.

Avoid the "bear trap."

Something that is very easy to get into might be very painful or impossible to extricate yourself from.

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, which allows developers and data engineers to run applications without managing servers behind the scenes. For the right use cases, serverless provides a very quick time to value. In 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 run a server, serverless exploded in popularity. The main reasons for the popularity of serverless are cost and convenience. Instead of paying the cost of a server, why not just pay when your code is evoked?

There are many flavors of serverless. Though function as a service (FAAS) is wildly popular, serverless systems actually predate the advent of AWS Lambda. As an example, Google Cloud's BigQuery is a serverless data

warehouse, in that there's no backend infrastructure for a data engineer to manage. Just load data into the system and start querying. You pay for the amount of data your query consumes, as well as a small cost to store your data. This payment model—paying for consumption and storage—is becoming more and more prevalent, particularly with serverless workloads. Another example of serverless is Google App Engine, originally released in 2008.

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 are going to become expensive. Looking specifically at the case of AWS Lambda, various engineers have found hacks to run batch workloads at incredibly low costs.[13] On the other hand, serverless functions suffer from an inherent overhead inefficiency. Every invocation involves spinning up some new resources and running some initialization code, overhead that does not apply to a long-running service. Handling one event per function call at a high event rate can be catastrophically expensive. As with other areas of ops, it's critical to *monitor* and *model.* That is, *monitor* to determine cost per event in a real world environment, and *model* by using this cost per event to determine overall costs as event rates grow in the future. Similar considerations apply to understanding costs for other serverless models.

Containers

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

Containers are often referred to as lightweight virtual machines. Where a traditional virtual machine wraps up an entire operating system, a container packages an isolated user space, i.e. a filesystem and one or 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. Hadoop now supports containers, allowing each job to have its own isolated dependencies. And as we mentioned, many Airflow users extract jobs into containers.

> **WARNING**
>
> Container clusters do not provide the same level of security and isolation offered by full virtual machines. Container escape—broadly, a class of exploits where code in a container gains privileges outside the container at the OS level—remains an unsolved problem. While Amazon EC2 is a truly multi-tenant environment with VMs from many customers hosted on the same hardware, a Kubernetes cluster should only host code within an environment of mutual trust, e.g. inside the walls of a single company.

Various flavors of container platforms add additional serverless features. Containerized function platforms (OpenFaas, Google Cloud Run) run containers as ephemeral units triggered by events, rather than persistent services. This gives users the simplicity of AWS Lambda with the full flexibility of a container environment in lieu of the highly restrictive Lambda runtime. And services such as AWS Fargate and Google App Engine run containers without the need to manage a compute cluster as is required for Kubernetes. These services also fully isolate containers, preventing the security issues associated with multi-tenancy.

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 a managed service. Quite a few commercial SAAS products

use Kubernetes behind the scenes. What you end up with is essentially a serverless service.

When Servers Make Sense

Why would you want to run a server 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 constant cost of running and maintaining a server (Figure 3-6). However, at a certain scale, the economic benefits of serverless diminish, and running servers becomes more attractive.

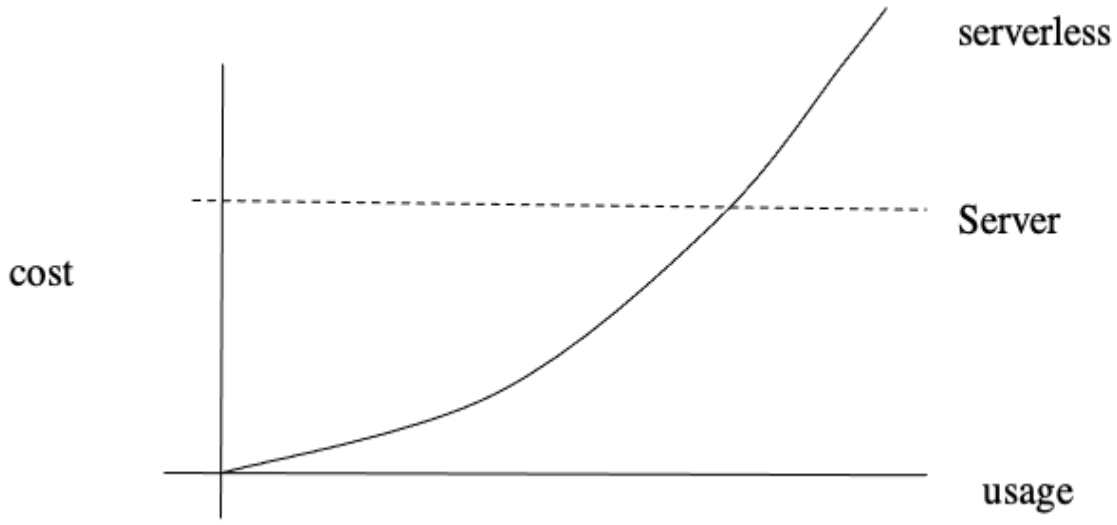


Figure 3-6. Cost of serverless vs using a server

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 when you're 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 transient resources that you can create as needed and then delete. If your

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

Use clusters and auto-scaling.

Take advantage of the cloud's ability to grow and shrink compute resources on demand. As your application grows its usage, cluster your application servers, and use auto-scaling capabilities to automatically expand your application as demand grows.

Treat your infrastructure as code.

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

Consider using containers.

For more sophisticated or heavy-duty workloads where simple startup scripts won't cut it, consider using containers and something like 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 a lot of moving parts, or if you 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, then you should consider containers instead.

Runtime limitations

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

Cost

Serverless functions are extremely convenient, but also inefficient when handling one event per call. This works fine for low event rates, but 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 servers—with containers and orchestration if possible—only after serverless makes no sense for your use case.

Undercurrents and How They Impact Choosing Technologies

As we've seen in this chapter, there's a lot for a data engineer 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 obvious whether a technology adopts data management as a principal concern. For example, a third-party vendor may use data management best practices—such as regulatory compliance, security, privacy, data quality, and governance—behind the scenes, but expose only a limited UI layer to the customer. In this case, while evaluating the product, it helps to ask the company about their data management practices. Here are some sample questions you should ask:

- How are you protecting data against breaches, both from the outside and from within?
- What is your product's compliance with GDPR, CCPA, and other data privacy regulations?
- Do you allow me to host my data to be compliant with these regulations?
- How do you ensure data quality, and that I'm viewing the right 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 any number of unforeseen problems.

When evaluating a new technology, how much control do you have over the deployment of new code, how will you be alerted if there's a problem, and how are you going to respond when there's a problem?

The answer to this largely depends on the type of technology you're considering. If the technology is OSS, then you're likely responsible for setting up monitoring, hosting, and code deployment. How will you handle issues? What's your incident response?

If you're using a managed offering, much of the operations are out of your control. Consider the vendor's SLA, how they alert you to issues, and whether they're transparent about how they're addressing the issue, an ETA to a fix, etc.

Data Architecture

As we discussed in Chapter 3, good data architecture means assessing tradeoffs 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

Through most of this chapter, we have actively avoided discussing any particular technology too extensively. We will 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 non-commercial 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.

At present, Airflow enjoys many advantages, largely due to 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 non-scalable 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 vs. Modular* above.) 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 will 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 each aim to solve some of the problems discussed above by rethinking components of the Airflow architecture. We also mention Datacoral, which intrigues us with its concept of metadata first architecture; Datacoral allows for automated workflow construction through analysis of data flows between queries.

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 there will almost certainly be new developments 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 pre-built open source whenever possible. Eliminating undifferentiated heavy lifting should be your big goal. Focus your resources—custom coding and tooling—on areas that give you a strong competitive advantage. For example, is hand-coding a database connection between MySQL and your cloud data warehouse a competitive advantage to you? Probably not. This is very much a solved problem. Pick an off-the-shelf solution (open source or managed SAAS), versus writing your own database connector. The world doesn't need the millionth +1 MySQL to cloud data warehouse connector.

On the other hand, why do customers buy from you? Your business very likely has something special about the way it does things. Maybe it's a particular algorithm that powers your fintech platform, or similar. By abstracting away a lot of the redundant workflows and processes, you can continue chipping away, refining, and customizing the things that really move the needle for the business.

Conclusion

Choosing the right technologies is no easy task, and especially at a time when new technologies and patterns emerge on a seemingly daily basis, 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.

Now that we're armed with knowledge, let's dive into the first layer of the data engineering lifecycle—source systems, and the data they generate.

---

1 As the authors were working on this chapter in September 2021, C was in position 1 on the TIOBE index. https://www.tiobe.com/tiobe-index/

2 Silicon Valley - Making the World a Better Place

3 https://mattturck.com/data2020/

4 https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/

5 https://dropbox.tech/infrastructure/evolution-of-dropboxs-edge-network

6 https://dropbox.tech/infrastructure/magic-pocket-infrastructure

7 https://aws.amazon.com/solutions/case-studies/dropbox-s3/

8 https://www.backblaze.com/company/about.html

9 https://www.cloudflare.com/what-is-cloudflare/

10 http://highscalability.com/blog/2017/12/4/the-eternal-cost-savings-of-netflixs-internal-spot-market.html

11 https://variety.com/2019/digital/news/netflix-loses-title-top-downstream-bandwidth-application-1203330313/

12 https://www.theinformation.com/articles/apples-spending-on-google-cloud-storage-on-track-to-soar-50-this-year

13 See https://intoli.com/blog/transcoding-on-aws-lambda/

Chapter 4. Ingestion

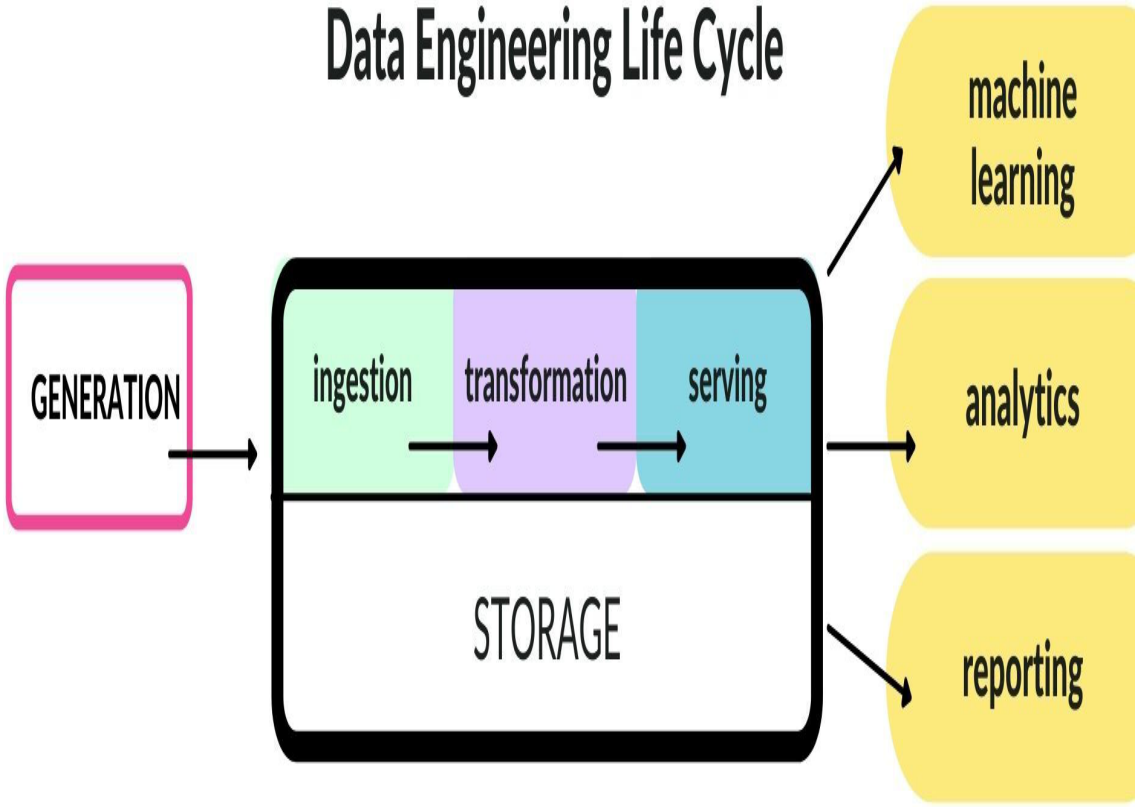
A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the seventh chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at book_feedback@ternarydata.com.

You just learned the various source systems you'll likely encounter as a data engineer, as well as ways to store data. Let's now turn our attention to the patterns and choices that apply to ingesting data from a variety of source systems. We will discuss what data ingestion is, the key engineering considerations for the ingestion phase, the major patterns for both batch and streaming ingestion, technologies you'll encounter, who you'll work with as you develop your data ingestion pipeline, and how the undercurrents feature in the ingestion phase (see Figure 4-1).



UNDERCURRENTS:

Security | Data Management | DataOps | Data Architecture | Orchestration | Software Engineering

Figure 4-1. Data engineering life cycle

What Is Data Ingestion?

Data moves from source systems into storage, with ingestion as an intermediate step (Figure 4-2). Data ingestion implies data movement and is a key component of the data engineering lifecycle.

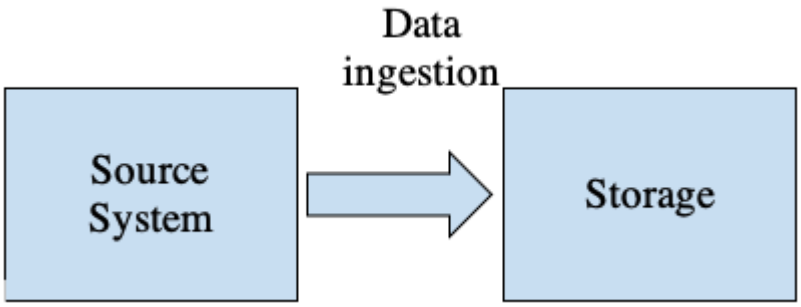


Figure 4-2. Data from System 1 is ingested into System 2

It's worth quickly contrasting data ingestion with *data integration*. Whereas data ingestion is the movement of data 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. Data integration will be described 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, where data stored in a database is copied from one table to another; we consider this to be another part of the general process of data transformation 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, there is a good deal of ceremony around different data movement and processing patterns, with older patterns such as ETL (extract, transform, load), newer patterns such ELT (extract, load, transform), and new names for long-established practices (reverse ETL).

All of these concepts are encompassed in the idea of a *data pipeline.* It is important to understand the details of these various patterns, but also know that a modern data pipeline could encompass all of these. As the world moves away from a traditional monolithic approach with very rigid constraints on data movement, towards an ecosystem of cloud services that are assembled like lego bricks to realize that products, data engineers prioritize using the right tools to accomplish the desired outcome rather than 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 different sources, combines data into 20 wide tables, trains five different machine learning 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 the ingestion 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?
- Frequency: 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?)
- If the data is from a streaming source, does it require in-flight processing for downstream ingestion?

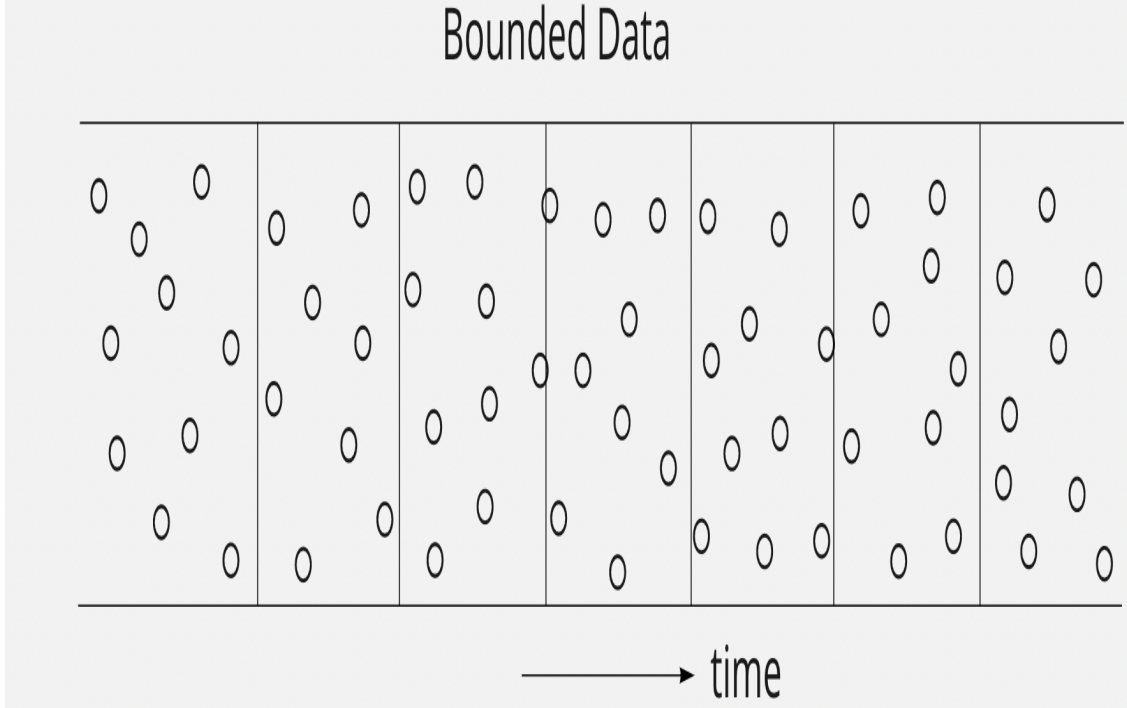
These questions undercut both batch and streaming ingestion and apply to the underlying architecture you’ll create, build, and maintain. Regardless of how often the data is ingested, you’ll want to consider these factors when designing your ingestion architecture:

- Bounded versus unbounded
- Frequency
- Synchronous versus asynchronous
- Serialization and deserialization
- Throughput and elastic scalability
- Reliability and durability
- Payload
- Push versus pull patterns

Let's look at each of these.

Bounded Versus Unbounded

As you might recall from Chapter 3, bounded and unbounded data (Figure 4-3), data comes in two forms—bounded and unbounded. Unbounded data is data as it exists in reality, where events happen when they happen, either sporadically or continuous, ongoing and flowing. Bounded data is a convenient way of bucketing data across some sort of boundary, such as time.


Bounded Data
time

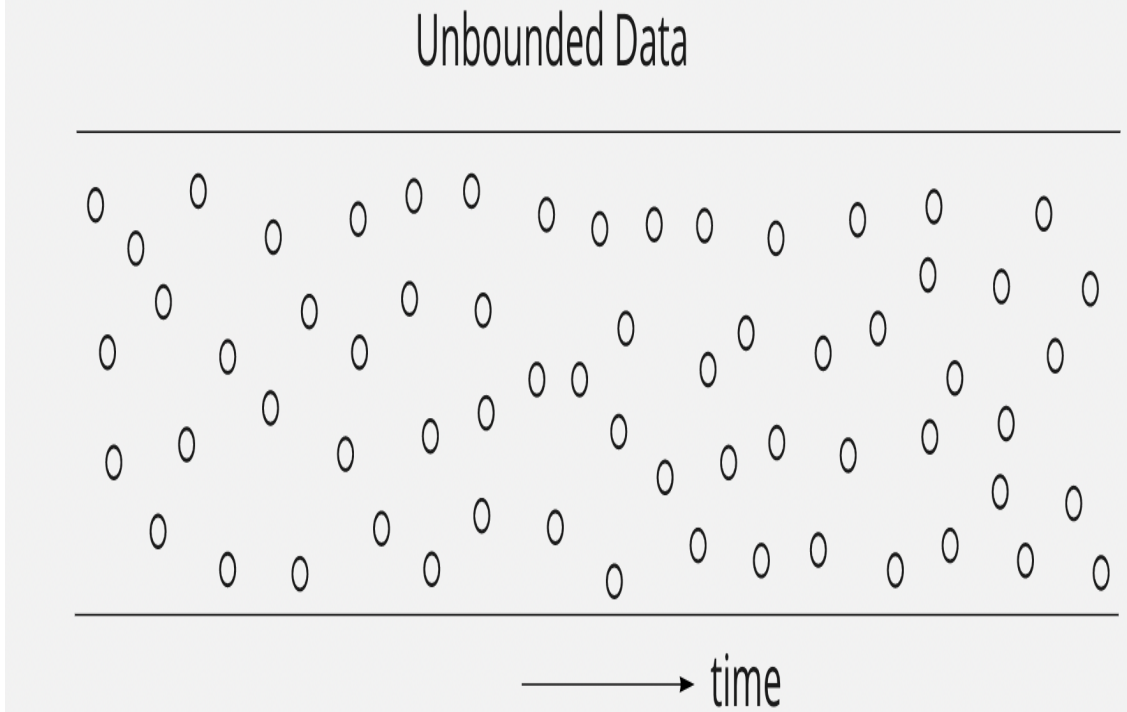

Unbounded Data
time

Figure 4-3. Bounded vs unbounded data

Let us adopt this mantra:

All data is unbounded until we bound it.

Like many mantras, this one is not precisely true 100% of the time; the grocery list that I scribbled this afternoon is truly bounded data. However, the idea is correct for practical purposes for the vast majority of data that we handle in a business context. That is, 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, but always 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 key decisions the data engineers must make in designing data ingestion processes is the data ingestion frequency. Ingestion processes can be a batch, micro-batch, or real-time.

Ingestion frequencies vary dramatically, on the spectrum from slow to fast (Figure 4-4). On the slow end, a business might ship its tax data to an accounting firm once a year. On the faster side, a change data capture system could retrieve new log updates from a source database once a minute. Even faster, a system might ingest events from IoT sensors continuously, and process these within seconds. In a company, data ingestion frequencies are often mixed, depending on the use case and technologies involved.

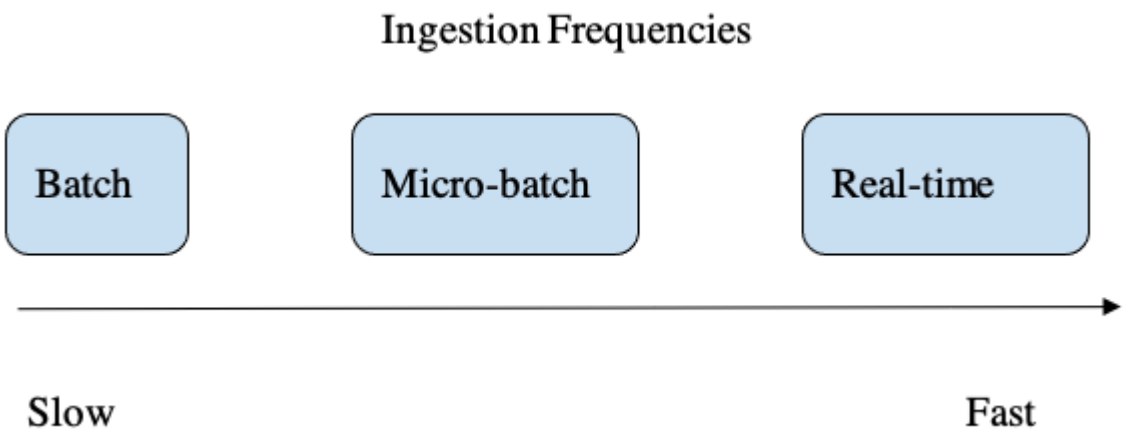


Figure 4-4. The spectrum of slow to fast ingestion, from batch to real-time

We note that "real-time" ingestion patterns are becoming increasingly common. We initially put "real-time" in quotes because no ingestion system is truly real-time. Any database, queue, pipeline, etc. has some inherent latency in delivering data to a target system. It is more accurate to speak of *near real-time*, but we often use the term real-time for the sake of 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 very short time intervals). For this book, we will use real-time and streaming interchangeably.

Even with a streaming data ingestion process in place, batch processing downstream is quite common. At the time of this writing, machine learning 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., wherein 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; you move at the cadence of how the batch moves.

We also note that streaming systems are an organic best fit for many modern data source types. In IoT applications, the common pattern is for each sensor to write events or measurements as they happen. While this data can be written 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 extremely well for event-driven architectures that are already exchanging messages through queues. And again, streaming architectures generally coexist with batch processing.

Synchronous Versus Asynchronous Ingestion

Ingestion systems can have a series of dependent steps (synchronous systems), or operate without any dependencies (asynchronous systems). When we discuss dependencies in this context, we're describing whether the completion of one step prevents downstream steps from starting.

With synchronous ingestion, the source, ingestion, and destination have hard dependencies and are tightly coupled. As you can see in Figure 4-5, each stage of the data engineering lifecycle has processes A, B, and C that are directly dependent upon each other. If Process A fails, Processes B and C cannot 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. If the ingestion or transformation process fails for any reason, the entire process must be replayed until it's successful. We've seen instances where the transformation process itself is a series of dozens of synchronous workflows, sometimes taking over 24 hours to finish. If any step of that transformation workflow failed (and it occasionally would fail), the entire transformation process needed to be restarted from the beginning!

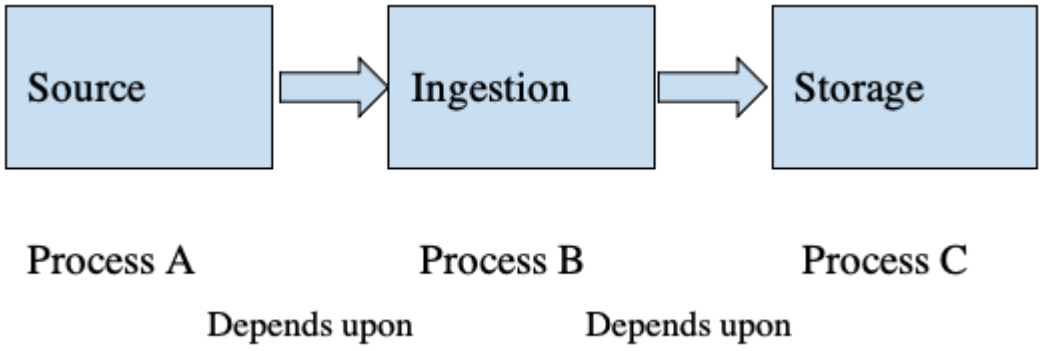


Figure 4-5. Synchronous data ingestion

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 4-6). For example, take the example of a web application that emits events into an Amazon Kinesis Data Stream (here acting as a buffer); the stream is read by Apache Beam, which parses and enriches events, then forwards them to a second Kinesis Stream; Kinesis Firehose rolls up events and writes objects to Amazon S3.

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. When the event rate is low and any backlog has cleared, events will move through the pipeline very quickly. 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.

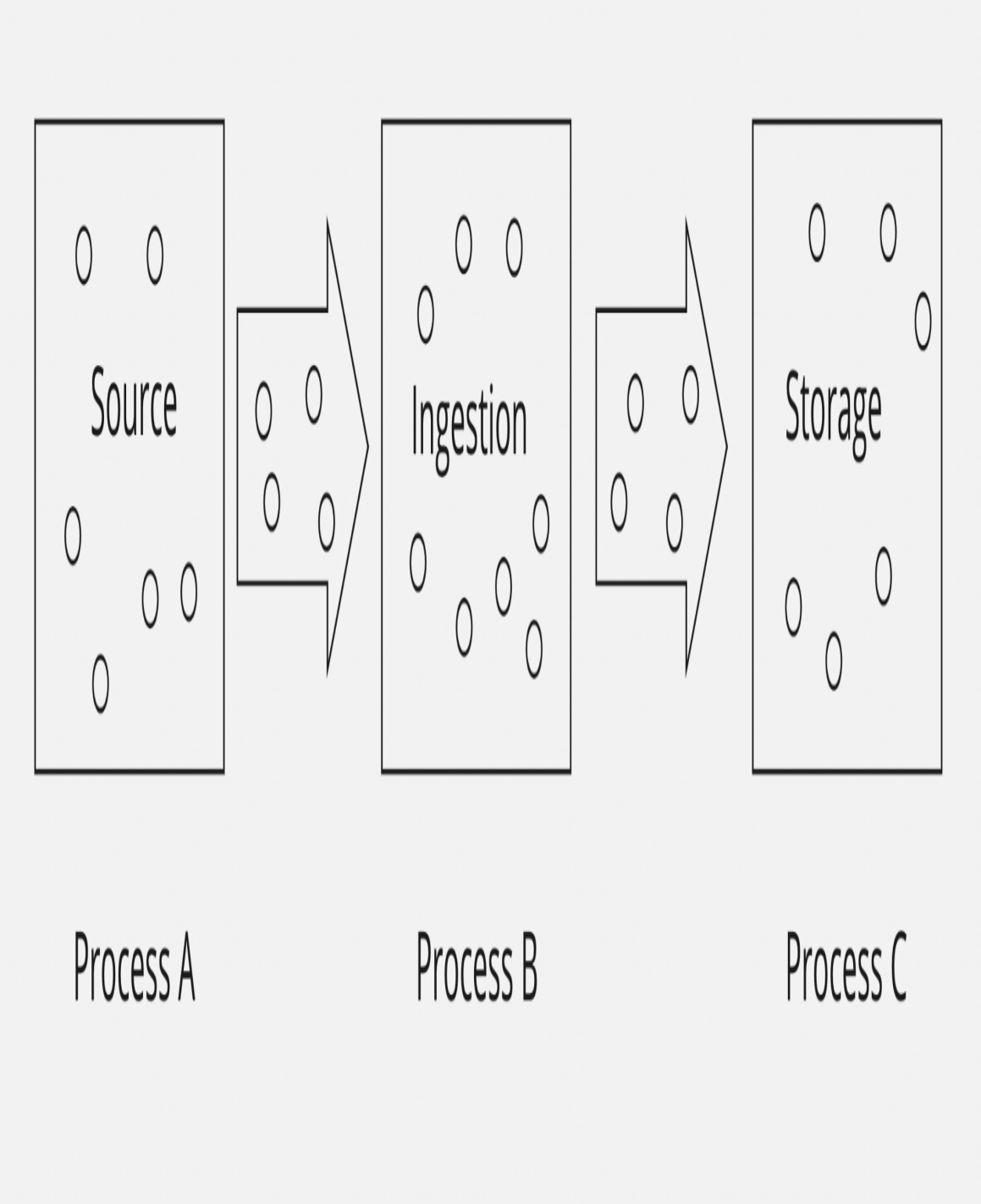


Figure 4-6. Asynchronous data ingestion

Serialization and Deserialization

Moving data from source to sink involves serialization and deserialization. Serialization means encoding the data from a source and preparing data

structures for transmission and intermediate storage stages. (See the more extensive discussion of serialization in the appendix on *Serialization and Compression.*)

Throughput and Scalability

In theory, your ingestion should never be a bottleneck. In practice, this is easier said than done. As your data volumes grow and requirements change, data throughput and system scalability become extremely critical. Design your systems to flexibly scale and shrink to match the desired data throughput.

Monitoring is key, as well as knowledge of the behavior of the upstream systems you depend upon and how they generate data. You should be aware of the number of events generated per time interval you're concerned with (events/minute, events/second, and so on), as well as the average size of each event. Your data pipeline should be able to handle both the frequency and size of the events you're ingesting.

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 is very rarely done in a constant fashion, and often ebbs and flows. Built-in buffering is required to collect events during rate spikes to prevent data from getting lost. Even in a dynamically scalable system, buffering bridges the time while the system scales. Buffering also allows storage systems to accommodate bursts.

These days, it's worth using managed services that handle the throughput scaling for you. While you can manually accomplish these tasks by adding more servers, shards, or workers, this isn't necessarily value add work.

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 especially important in the ingestion stages of data pipelines. *Reliability* entails high uptime and appropriate failover for ingestion systems. Durability entails making sure that data isn't lost or corrupted.

Some data sources (e.g., IoT devices) 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. Will your ingestion process continue if an AWS zone goes down? How about a whole region? How about the power grid or the internet? Reliability and durability have both direct and indirect costs: building a highly redundant system can entail big cloud bills while keeping a team on call 24 hours a day takes a toll on your team and resources.

Don't assume that you can build a system that will reliably and durably ingest data in every possible scenario. Even the massive budget of the US federal government can't guarantee this. In many extreme scenarios, ingesting data actually won't matter. For example, if the internet goes down, there will be little to ingest even if you build multiple data centers in underground bunkers with independent power. Always evaluate the tradeoffs and costs of reliability and durability.

Payload

The *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 get an idea of why this matters.

Kind

The *kind* of data you handle directly impacts how it's handled 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 format of the data, or how it is expressed in bytes, name, and file extension. 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 binary and 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 RGB dimensions are necessary for deep learning applications. 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 different kinds of data:

Tabular

> The number of rows and columns in the dataset, commonly expressed as M rows and N columns

Semi-structured JSON

> The key-value pairs, and depth of nesting that occurs with sub-elements

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., 10003 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 a variety of formats such as ZIP, TAR, and so on (See the discussion of compression in the appendix on *Serialization and Compression)*.

Schema and data types

Many data payloads have a schema, such as tabular and semi-structured data. As we've mentioned earlier in this book, a schema describes the fields and types of data that reside within those fields. Other types of data such as unstructured text, images, and audio will not have an explicit schema or data types, though they might come with technical file descriptions on shape, data and file format, encoding, size, etc.

You can connect to databases in a variety of ways, i.e. file export, change data capture, JDBC/ODBC, etc. The connection is the easy part of the process. The great engineering challenge is understanding the underlying schema. Applications organize data in a variety of 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 ORM (object-relational mapping), which automatically generates schemas based on object structure in languages such as Java or Python. Structures that are natural in an object-oriented language often map to something messy in an operational database. Data engineers may also need to familiarize themselves with the class structure of application code for this reason.

Schema is not only for databases. As we've discussed, APIs present their schema complications. Many vendor APIs have nice reporting methods that prepare data for analytics. In other cases, engineers are not so lucky and the API is a thin wrapper around underlying systems, requiring engineers to

gain a deep understanding of application internals to use the data. This situation is more challenging than dealing with complex data internal to an organization because communication is more difficult.

Much of the work associated with ingesting from source schemas happens in the transformation stage of the data engineering lifecycle, 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. We'll return to this topic later in this chapter, in the section on *Who you'll work with*.

Detecting and handling schema changes in upstream and downstream systems

Schema changes occur frequently in source systems and often are well out of the control of data engineers.

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 automatically respond to changes, and alert on changes that cannot be accommodated automatically. Automation is great, 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 evolving schemas. It essentially describes the data model for messages, allowing consistent serialization and deserialization between producers and consumers. Schema registries are used in Kafka, AWS Glue, and others.

Metadata

In addition to the obvious 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 major limitations of the early approach to the data lake—or data swamp, which could turn it into a data superfund site—was a complete lack of attention to metadata. Without a detailed description of the data, the data itself 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 Patterns

We introduced the concept of push vs pull when we introduced the data engineering lifecycle in Chapter 2. Roughly speaking, a *push* strategy (Figure 4-7) involves a source system sending data to a target, while a *pull* strategy (Figure 4-8) entails a target reading data directly from a source. As we mentioned in that discussion, the lines between these strategies are blurry.

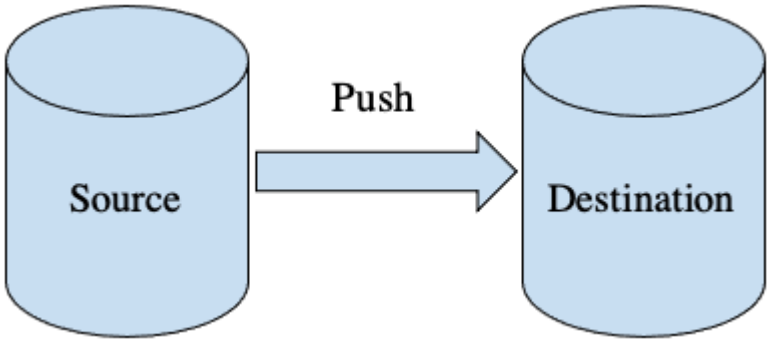


Figure 4-7. Pushing data from source to destination

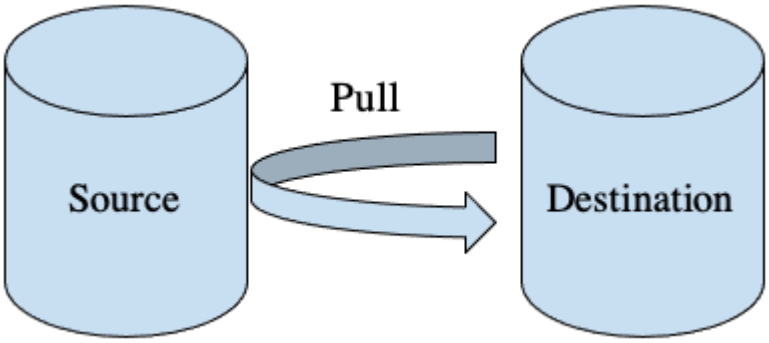


Figure 4-8. Pushing data from source to destination

We will discuss *push* versus *pull* in each subsection on ingestion patterns, and give our opinionated reasoning on when a pattern is *push* or *pull.* Let's dive in!

Batch Ingestion Patterns

It is often convenient to ingest data in batches. This means that data is ingested by either taking a subset of data from a source system, based either on a time interval or size of accumulated data (Figure 4-9).

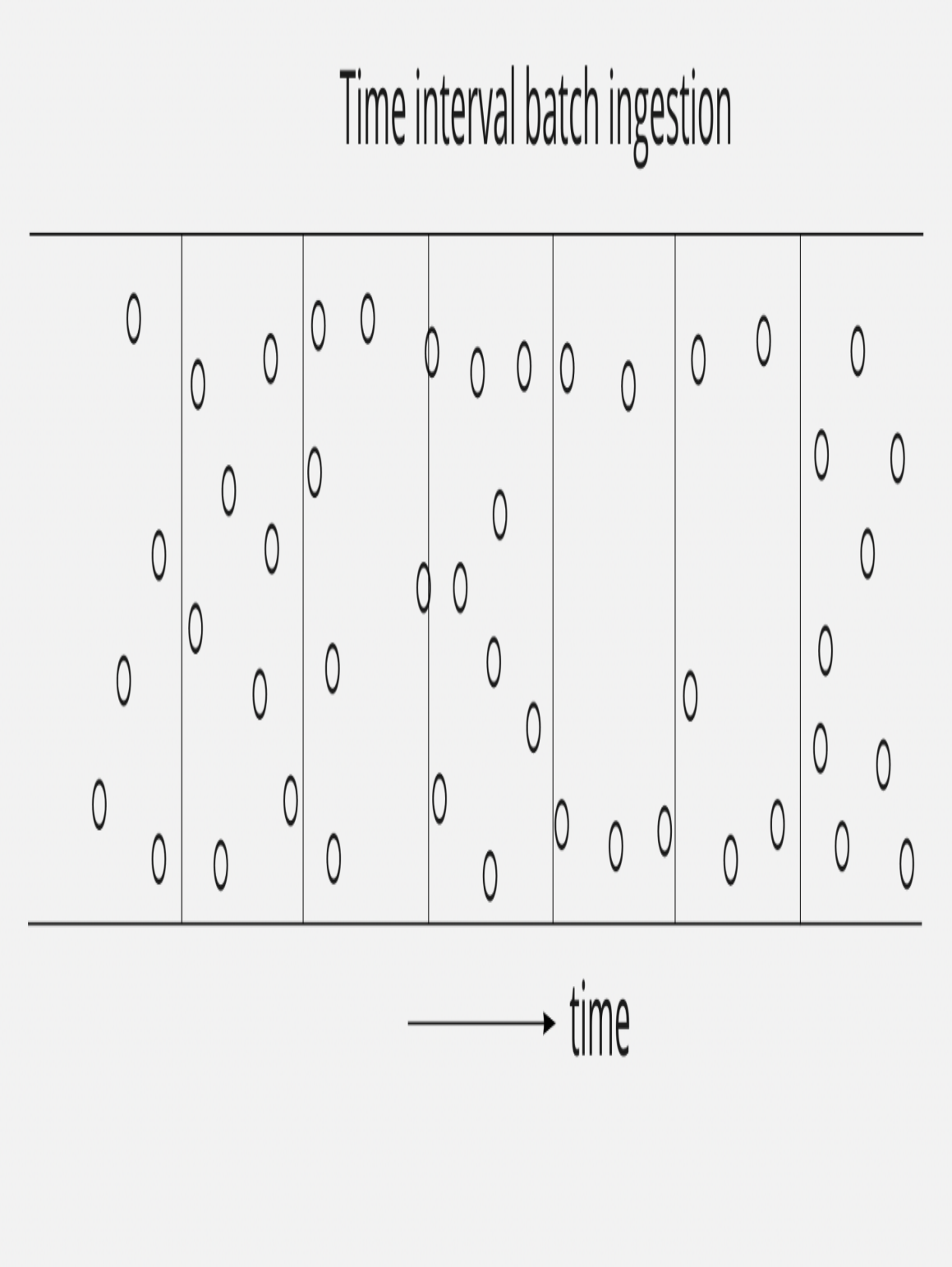


Figure 4-9. Time interval batch ingestion

Time interval batch ingestion is extremely common 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 4-10) is quite common when data is moved from a streaming-based system into object storage—ultimately, the data must be cut into discrete blocks for future processing in a data lake. Size-based ingestion systems such as Kinesis Firehose can break data into objects based on a variety of criteria, such as size bytes of the total number of events.

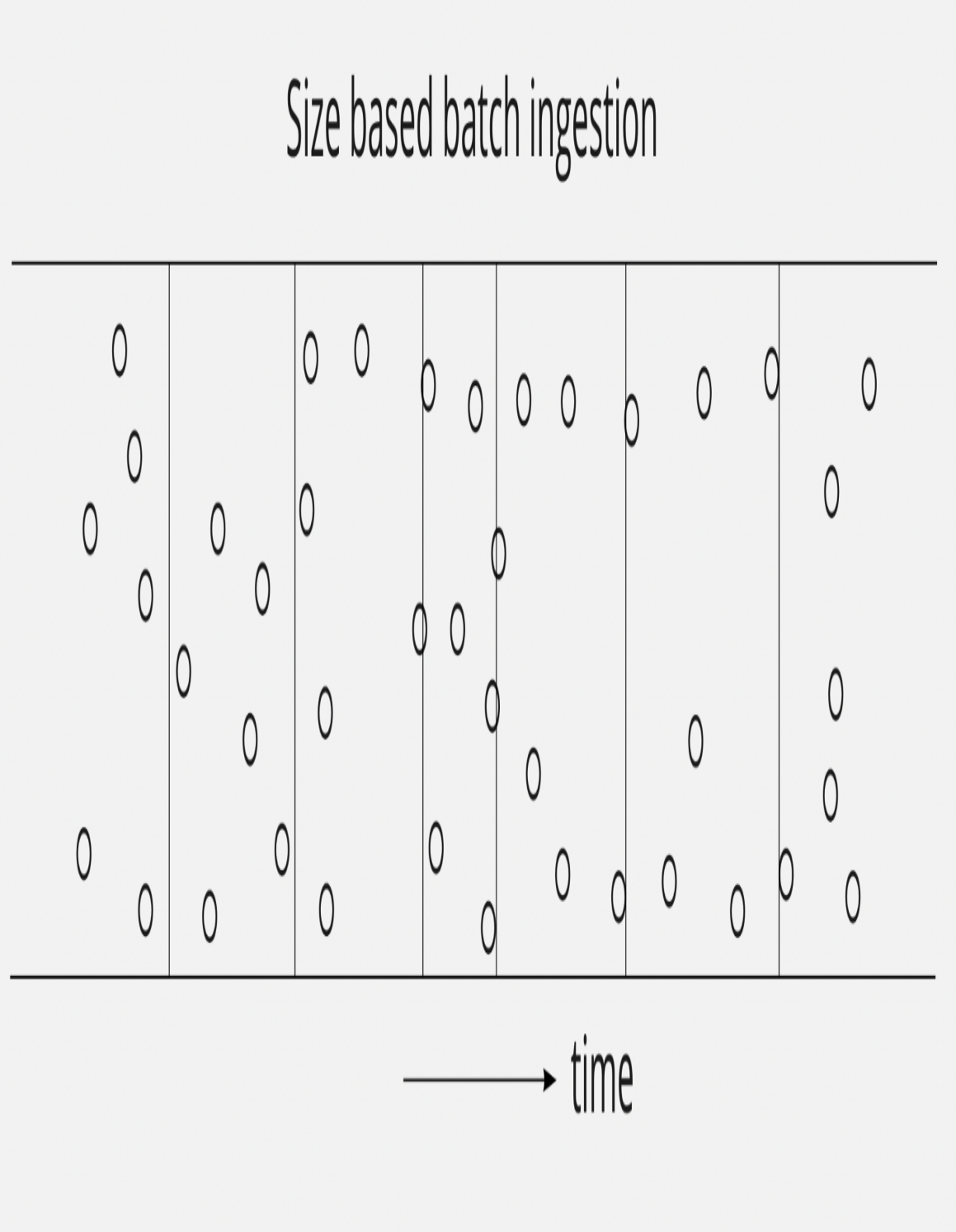


Figure 4-10. Size-based batch ingestion

Some commonly used batch ingestion patterns, which we'll discuss in this section, include:

- Snapshot or differential extraction
- File-based export and ingestion
- ETL versus ELT
- Data migration

Snapshot or Differential Extraction

Data engineers must choose whether to capture full snapshots of a source system or differential updates. With full snapshots, engineers grab the full 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, target storage usage, etc., full snapshot reads remain extremely common due to their simplicity.

File-Based Export and Ingestion

Data is quite often moved between databases and systems using files. That is, 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* ingest pattern. This is because the work of data export and preparation is done on the source system side.

File-based ingestion has several potential advantages over a direct database connection approach. For security reasons, it is often undesirable to allow any direct access to backend systems. With file-based ingestion, export processes are run on the data source side. This gives source system engineers full control over what data gets exported and how the data is pre-processed. Once files are complete, they can be provided to the target system in a variety of ways. Common file exchange methods are object storage (i.e. Amazon S3, Azure Blob Storage), SFTP, EDI, or SCP.

ETL Versus ELT

In Chapter 3, we introduced ETL and ELT, which are both extremely common ingest, storage, and transformation patterns you'll encounter in batch workloads. For this chapter, we'll cover the Extract (E) and the Load (LO) parts of ETL and ELT; transformations will be covered in Chapter 8:

Extract

Extract means getting data from a source system. While *extract* seems to imply *pulling* data, it can also be push-based. Extraction may also entail 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 into which you're loading, the schema of the data, and the performance impact of loading.

Transform

We'll cover transformations in much more detail in Chapter 8. Know that data can be transformed after it's been ingested, but before it's loaded into storage. This is common in classic ETL systems that do in-memory transformations as part of a workflow. It's also common with event streaming frameworks such as Kafka, which uses the Kafka Stream API to transform and join data before persisting the output to storage.

Inserts, Updates, and Batch Size

Batch-oriented systems often perform poorly when users attempt to perform a large number of small-batch operations rather than a smaller number of large operations. For example, while it is a common pattern 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 a large number of small in-place update operations is an even bigger problem because it forces the database to scan each existing column file to run the update.

Understanding the appropriate update patterns for the database you're working with. Also, understand that certain technologies are purpose-built for high insert rates. Systems such as Apache Druid and 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 standard 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 a new environment is not usually trivial, and data needs to be moved in bulk. Sometimes this means moving data sizes that are 100s of TBs or much larger, often involving not just the migration of specific tables, but moving entire databases and systems.

Data migrations probably aren't a regular occurrence in your role as a data engineer, but it's something you should be familiar with. As is so often the case for data ingestion, schema management is a key consideration. If you're migrating data from one database system to another, say Teradata to BigQuery, no matter how closely the two databases resemble each other, there are nearly always subtle differences in how they handle schema. Fortunately, it is generally easy to test ingestion of a sample of data and find schema issues before undertaking a full table migration.

Most modern data systems perform best when data is moved in bulk rather than as individual rows or events. File or object storage is often a good intermediate stage for moving 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.

Streaming Ingestion Patterns

Another way to ingest data is from a stream. This means that data is ingested continuously. Let's look at some patterns for ingesting streaming data.

Types of Time

While time is an important 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 the event is generated, and when it's ingested and processed (Figure 4-11):

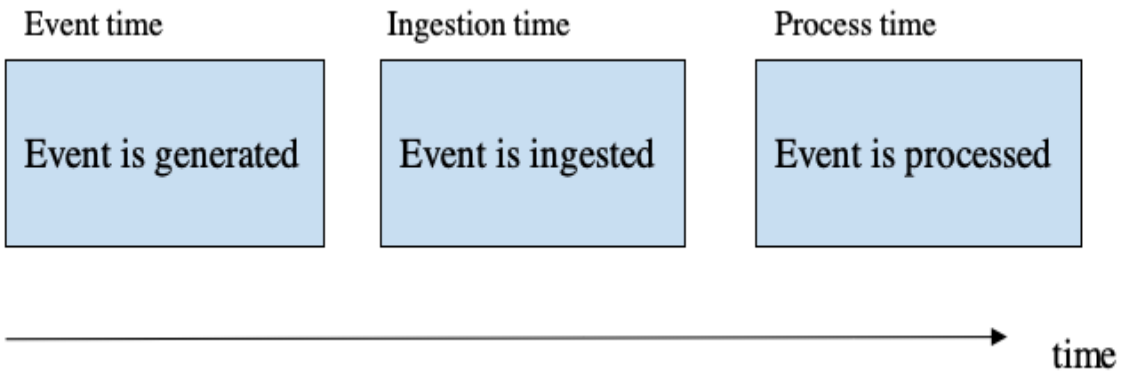


Figure 4-11. Event, ingestion, and process time

Event time

> Event time is the time at which an event is generated in a source system, including the timestamp of the original event itself. Upon event creation, there will be an undetermined time lag 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.

Ingestion time

After data is created, it is ingested somewhere. Ingestion time is the time at which 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, within minutes, hours, or days, or simply persist in storage indefinitely. We will cover the details of storage in Chapter 8.

Processing time

Processing time is the time at which data is processed, which is typically some sort of transformation. You'll learn more about various kinds of processing and transformations in Chapter 8.

Late-arriving data

A group of events might occur around the same time frame, but because of various circumstances, might be late in arriving for ingestion. This is called late-arriving data and is common when ingesting data. You should be aware of late-arriving data, and the impact on downstream systems and uses. For example, if you assume that ingestion or processing time is the same as the event time, you may get some very strange results if your reports or analysis depend upon an accurate portrayal of when events occur.

Key Ideas

We spend a good deal of time on stream processing in Chapter 8, where we discuss data transformation. We'll quickly introduce key streaming ingestion ideas here and present a more extensive discussion there:

Streaming ingestion and storage systems

Streaming storage collects messages, log entries, or events and makes them available for downstream processing. Typical examples are Apache Kafka, Amazon Kinesis Data Streams, or Google Cloud

Pub/Sub. Modern streaming storage systems support basic producer/consumer patterns (publisher/subscriber), but also support *replay*, i.e., the ability to playback a time range of historic data.

Producers and consumers

Producers write data into streaming storage, while consumers read from the stream. In practice, a streaming pipeline may have many consumers and producers at various stages. The initial producer is usually the data source itself, whether a web application that writes events into the stream or a swarm of IoT devices.

Clusters and partition

In general, modern steaming systems distribute data across clusters. In many cases, data engineers have some control of how the data is partitioned across the cluster through the use of a *partition key.*

Choice of ingestion partition key can be critical in preparing data for downstream consumption, whether for preprocessing that is a key part of the ingestion stage or more complex downstream processing. Often, we set upstream processing nodes to align with partitions in the stream messaging system.

Topics

A topic is simply a data stream; a streaming storage system can support a large number of separate topics (streams) simultaneously, just as a relational database supports many tables.

Streaming Change Data Capture

While there are batch versions of change data capture (CDC), we are primarily interested in streaming change data capture. The primary approaches are CDC by logging, wherein each writes to the database is recorded in logs and read for data extraction, and the database trigger pattern, where the database sends some kind of signal to another system

each time it makes a change to a table. For streaming ingestion, we assume that the CDC process writes into some kind of streaming storage system for downstream processing.

Real-time and Micro-batch: Considerations for Downstream Destinations

While streaming ingestion generally happens in a stream storage system, the *preprocessing* part of ingestion (e.g. data parsing) can happen in either a true stream processor or a micro-batch system. Micro-batch processing is essentially high-speed batch processing, where batches might happen every few seconds.

Apache Spark often processes data in micro-batches, though there are newer more continuous modes as well. The micro-batch approach is perfectly suitable for many applications. On the other hand, if you want to achieve operational monitoring that is much closer to real-time, a continuous approach might be more appropriate.

Ingestion Technologies

Now that we've described some of the major patterns that underlie ingestion in general, we turn our attention to the technologies you'll use for data ingestion. We'll give you a sample of the types of ingestion technologies you'll encounter as a data engineer. Keep in mind the universe of data ingestion technologies is vast and growing daily. Although we will cite common and popular examples, it is not our intention to provide an exhaustive list of technologies and vendors, especially given how fast the discipline is changing.

Batch Ingestion Technologies

To choose appropriate technologies, you must understand your data sources (see Chapter 5). In addition, you need to choose between full replication and change tracking. With full replication, we simply drop the old data in

the target and fully reload from the source. Change tracking takes many forms, but often it entails pulling rows based on update timestamp and merging these changes into the target.

In addition, preprocessing should be considered a part of ingestion. Preprocessing is data processing that happens before the data is truly considered to be ingested. In streaming pipelines, raw events often arrive in a rather rough form directly from a source application and must be parsed and enriched before they can be considered fully ingested. In batch processing scenarios, similar considerations apply, with data often arriving as raw string data, then getting parsed into correct types.

Also, think about FinOps and cost management early in the design of your ingestion architecture. FinOps entails designing systems and human processes to make costs manageable. Think about the implications of scaling up your solution as data scales, and design for cost efficiency. For example, instead of spinning up full-priced on-demand EC2 instances in AWS, instead, build in support for EC2 spot instances where this applies. Ensure that costs are visible and monitored.

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 1.0 was released in 1992. ODBC (Open Database Connectivity) 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 they are received by the driver and translated back into a standard form, and read by the client.

For purposes of 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. The ingestion tool might pull data once a day or once every five minutes.

JDBC (Java Database Connectivity) was released as a standard by Sun Microsystems in 1997. JDBC is conceptually extremely 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 a bit strange to have a database API dedicated to a single programming language, but there are strong motivations for this. The JVM (Java Virtual Machine) is standard, portable across hardware architectures and operating systems, and provides the performance of compiled code through a JIT (just in time compiler). The JVM is far and away from the most popular compiling virtual machine 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 (Java, Scala, Clojure, Kotlin, etc.) 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.

Returning to the general concept of direct database connections, both JDBC and ODBC are used extensively for data ingestion from relational databases. 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. However, 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 has to be re-encoded as string data to be sent over the

wire, and columns from columnar databases must be re-serialized as rows. Many alternatives have emerged for lower friction data export.

As discussed in the section on file-based export, many databases now support native file export that bypasses JDBC/ODBC and exports data directly in modern formats. Alternatively, many databases—Snowflake, BigQuery, ArangoDB, etc.—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, then orchestrate ingestion into a downstream system (see Figure 4-12). The reader process can run in a fully ephemeral cloud instance or directly in an orchestration system.

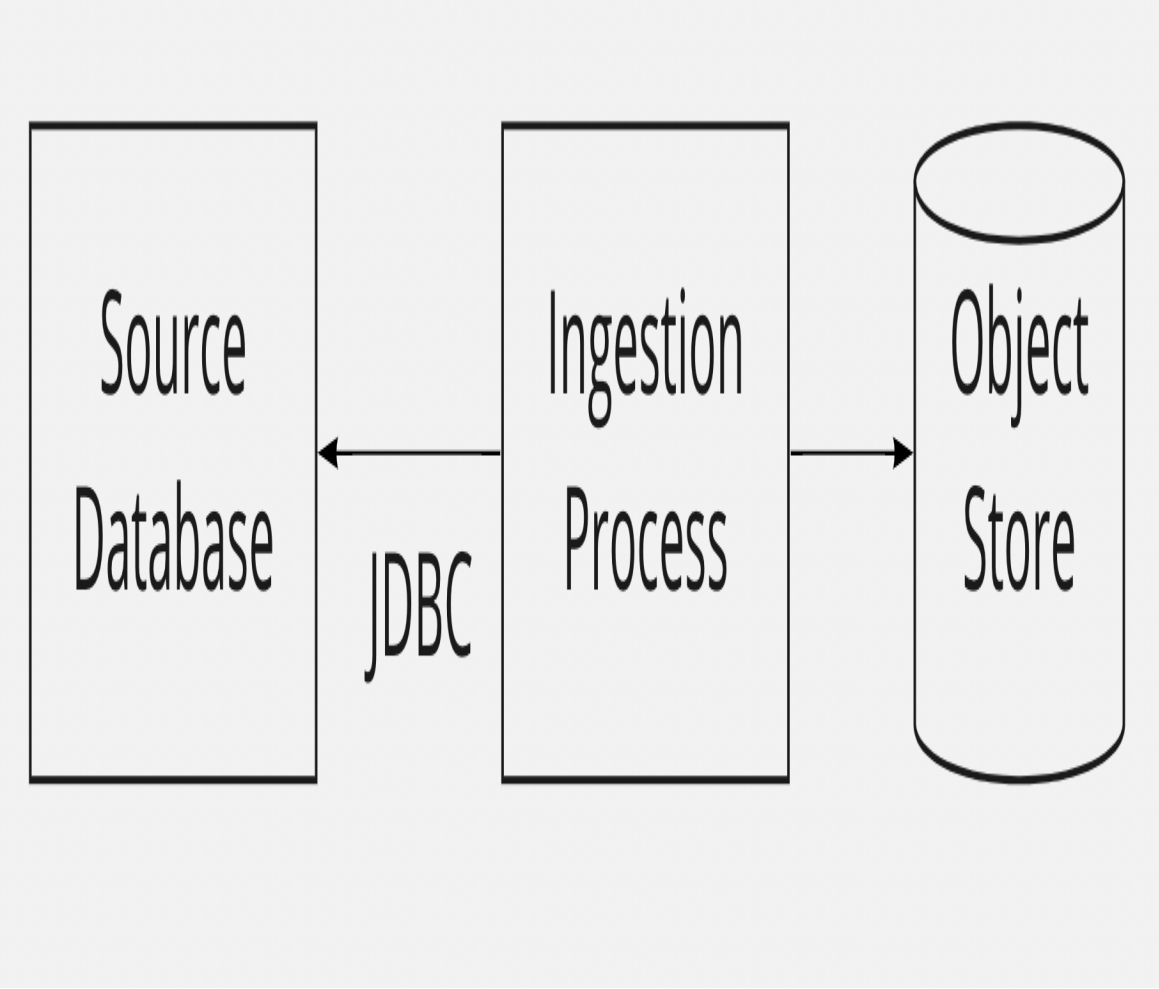


Figure 4-12. An ingestion process reads from a source database using JDBC, then writes objects into object storage. A target database (not shown) can be triggered to ingest the data with an API call from an orchestration system.

SFTP

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. That is, they work with partner businesses that either consume or provide data using SFTP, and are not willing to rely on other standards. To avoid data leaks, security analysis is critical in these situations. If SFTP is required, it can be combined with extra network security, such as only allowing authorized IP addresses to access the network, or even passing all traffic over a VPN.

Object storage

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 an extremely robust track record, and provides high-performance movement of data.

We'll discuss object storage much more extensively in Chapter 6. At a basic level, object storage is a multi-tenant 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, moving data between teams, and transferring data between organizations. You can even provide short-term access to an object with a signed URL, giving a user short-term permission.

SCP

SCP (secure copy) 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.

EDI

Another practical reality for data engineers is EDI (Electronic Data Interchange). The term is vague enough that it could refer to any method of data movement. In practice, it is used in modern parlance to refer to rather archaic means of file exchange, such as by email, or on a flash drive. Data engineers will find that some of their data sources do not support more modern means of data transport, often due to archaic IT systems, or human process limitations. They 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, something that we still frequently see.

Databases and file export

Engineers should be aware of how the source database systems handle file export. For many transactional systems, export involves large data scans that put a significant load on the database. 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 down 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 exports coincide with high source system load.

Some modern cloud databases are highly optimized for file export. Snowflake allows engineers to set up a warehouse (compute cluster) just for export, with no impact on other warehouses running analytics. BigQuery handles file exports using a backend service, completely independent of its query engine. AWS Redshift gives you the ability to issue a SQL UNLOAD command to dump tables into S3 object storage. In all cases, file export is the "best practice" approach to data movement, recommended by the vendor over JDBC/ODBC connections for cost and performance reasons.

Practical issues with common file formats

Engineers should also be aware of the file formats that they're using to export. At the time of this writing, CSV is nearly universal, but also extremely error-prone. Namely, CSV's default delimiter is also one of the most common characters in the English language—the comma! But it gets worse. CSV is by no means a uniform format. Engineers must stipulate delimiter, quote characters, escaping, etc. to appropriately handle the export of string data. CSV also doesn't natively encode schema information, nor does it directly support modern nested structures. CSV file encoding and schema information must be configured in the target system to ensure appropriate ingestion. Auto-detection is a convenience feature provided in many cloud environments but is not appropriate for production ingestion. As a best practice, engineers should record CSV encoding and schema details in file metadata.

More modern export formats include Parquet, Avro, Arrow, and ORC or JSON. These formats natively encode schema information and handle arbitrary string data with no special 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, a much lighter operation than pivoting data into rows (CSV, Avro). Modern 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 the appendix on *serialization and compression* for a more extensive discussion of file formats.

SSH

Strictly speaking, SSH is not an ingestion strategy, but a protocol that is used in conjunction with other ingestion strategies. We use SSH in a few different 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 extremely limited 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, then connects from the host machine to the database.

Shell

The shell is the interface by which you may execute commands to ingest data. In practice, the shell can be used to script workflows for virtually any software tool, and shell scripting is still used extensively in ingestion processes. For example, a shell script might read data from a database, reserialize the data 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 service level agreement grows more stringent, engineers should consider moving to a true orchestration system instead.

APIs

> *The bulk of software engineering is just plumbing.*
>
> —Karl Hughes

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—SAAS platforms, partner companies, etc. The hard reality is that there is no true standard for data exchange over APIs. Data engineers can expect to 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. Google was arguably a leader in this space, with AdWords client libraries available in various flavors. Many other vendors have since followed suit.

Second, there are numerous data connector platforms available now as proprietary software, open-source, or managed open source. These platforms provide turnkey data connectivity to many data sources; for

unsupported data sources, they offer frameworks for writing custom connectors. See the section below on managed data connectors.

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 Google BigQuery, Snowflake, Redshift, or Amazon 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 significant and rapid impact in the data engineering space. For example; Google now supports direct sharing of data from a variety of its advertising and analytics products (GoogleAds, Google Analytics, etc.) to BigQuery. Any business that advertises online can now reallocate software development resources away from Google product data ingestion and focus instead on everything downstream.

When data sharing is not an option and direct API access is necessary, don't reinvent the wheel. 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 take the form of providing API technical specifications in a standard format, or of 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. There are two main aspects of handling custom API connections: software development and ops. Follow software development best practices: you should use version control, continuous delivery, automated testing, etc. In addition to following DevOps best practices, consider an orchestration framework, which can dramatically streamline the operational burden of data ingestion.

Webhooks

Webhooks, as we discussed in Ch. 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 ingest code. The code makes requests and receives data in responses.

With a webhook (Figure 4-13), the data provider documents an API request specification, but this specification is implemented by the data consumer. The consumer sets up an endpoint, and the data source calls the endpoint, delivering data in requests. Typically, webhooks deliver one event at a time; the consumer is responsible for ingesting each request and handling data aggregation, storage, processing, etc.

Webhook-based data ingestion architectures can be brittle, difficult to maintain, and inefficient. Data engineers can build more robust webhook architectures—with lower maintenance and infrastructure costs—by using appropriate off-the-shelf tools. A common pattern uses a serverless function framework (i.e. AWS Lambda) to receive incoming events, a streaming message bus to store and buffer messages (i.e. AWS Kinesis), a stream processing framework to handle real-time analytics (i.e. Apache Flink), and an object store for long term storage (i.e. Amazon S3).

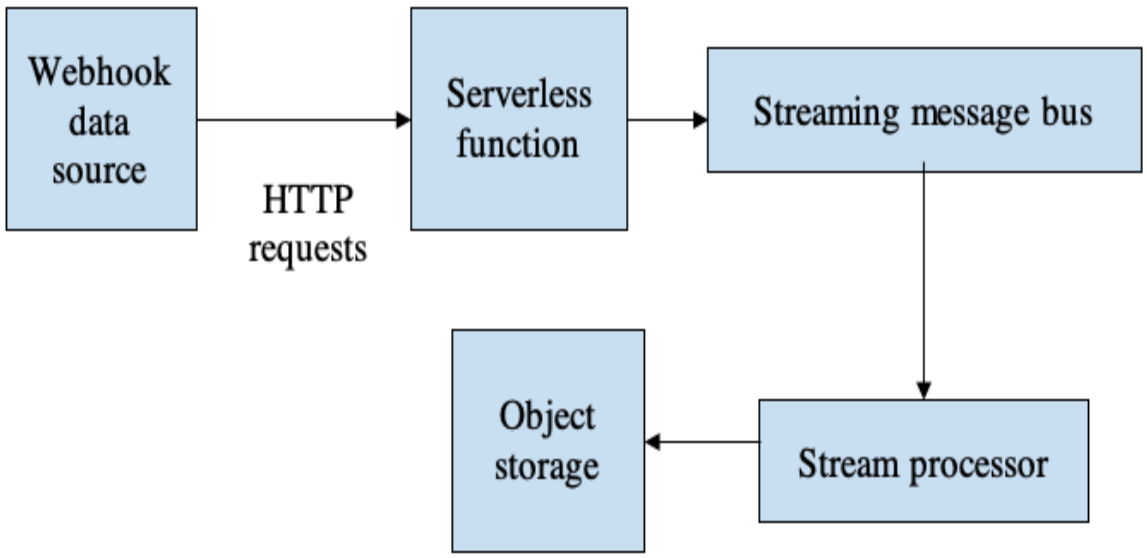


Figure 4-13. A basic webhook ingestion architecture built from cloud services. Using "serverless" services reduces operational overhead.

You'll notice that this architecture does much more than simply ingesting the data. This underscores the fact that ingestion is highly entangled with the other stages of the data engineering lifecycle—it is often impossible to define your ingestion architecture without also making decisions about storage and processing.

We note that this architecture can be simplified if one does not need real-time analytics. However, trying to simplify too much creates new problems. For instance, writing directly from a Lambda serverless function to S3 might produce a massive number of objects, with an extremely high request rate to S3, and potentially high costs. This is where data engineers need to understand the critical role played by each component in the pipeline. To simplify the architecture in a more sane way, we could remove the streaming analytics system, accumulate events into the messaging bus, and periodically query a large chunk of data over a time range to write it into S3. Better yet, we could use a serverless tool like Kinesis Firehose, which automates the process of event rollup to create S3 objects of a reasonable size.

Legacy data flow management tools

We would be remiss not to mention data flow management tools such as Informatica and Talend, to name a few. This category of tool is designed to serve several stages of the data engineering lifecycle, including ingestion, and processing.

In practice, these systems also integrate some orchestration capabilities. (See the undercurrents section in this chapter, and corresponding discussions throughout the book.) Such systems include connectors to a variety of sources, and allow users to define data pipelines, often using a visual interface that represents processing stages with icons. Processing can be set on a schedule. Each processing stage runs once its upstream dependencies are met. Of course, there are generally ingestion steps at the beginning of each pipeline.

These systems typically employ two main processing models. We have already discussed both ETL and ELT. Systems based on the ETL model

ingest data internally, transform it and write it back to external storage, such as a cloud data warehouse. Tools that employ the ELT model handle most processing in an external system such as a cloud data warehouse.

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 (software as a service) 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.

Potentially, some automation can be applied by using web interface simulation frameworks such as simulation, but web interfaces remain an area of high friction for data ingestion. For example, you might be able to automate the clicking of a web interface with Selenium. But whenever possible, find another approach.

Managed Data Connectors

In the section on ingesting data from APIs, we mention the emergence of managed data connector platforms and frameworks. The goal of these tools is to provide a standard set of connectors that are available out of the box to spare data engineers much detailed plumbing to connect to a particular source.

For now, Fivetran has emerged as a category leader in this space, but there are many up-and-coming competitors, both proprietary and open source. Generally, proprietary options in the space allow users to set a target and source, set permissions and credentials, configure an update frequency, and begin syncing data. Data syncs are fully managed and monitored—if data synchronization fails, users will receive an alert.

The open-source options in the space are usually part of a product, and available in supported and unsupported flavors. Stitch started this trend by introducing the Singer Python framework. Singer provided a standard set of abstractions for creating data connectors. Engineers could then use Singer

to manage and run their data synchronization or send their connectors to Stitch to run within their fully managed platform. Various competitors to Stitch have emerged. Some utilize the Singer framework; others, such as Airbyte and Meltano, have introduced new open-source data connector frameworks.

We note also that even proprietary data connector engines allow the creation of custom connections with some coding effort. Fivetran allows this through various serverless function frameworks, including AWS Lambda and its similar competitors. Engineers write function code that receives a request, pulls data, and returns it to Fivetran. Fivetran takes care of orchestration and synchronization to the target database.

These are just a few of *many* options for managed connectors, and we expect that this space of SAAS managed services and OSS that support the development of custom connectors will continue to grow.

Web scraping

Web scraping is another common data source for engineers. Any search engine must scrape the web to analyze links, build an index, etc. However, many other businesses also rely on web scraping to survey web content that may be relevant to their business activities.

A full discussion of web scraping is well beyond the scope of this book. There are many books specific to the subject. We recommend that readers look at web scraping books in O'Reilly's catalog and beyond. There are also numerous online resources, in video tutorials, blog posts, etc.

Here is some top-level advice to be aware of before undertaking any web scraping project. First, learn to be a good citizen. Don't inadvertently create a denial of service attack, and don't get your IP address blocked. Understand how much traffic you're generating 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; in fact, excessive web scraping could lead to the disabling of your AWS account.

Second, be aware of the legal implications of your activities. Again, generating denial of service attacks has legal implications. Activities that violate terms of service may cause legal headaches for your employer.

Web scraping has interesting implications for the processing stage of the data engineering lifecycle; there are various things that engineers should think about at the beginning of a web scraping project. What do you intend to do with the data? Are you just pulling key fields from the scraped HTML using Python code, then writing these values to a database? Do you intend to maintain the full 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 truly big data (100 TB or more), transferring data directly over the internet may be a slow and extremely expensive 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, then send it back to the cloud vendor who will upload your data into their cloud. 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 in a semi-trailer. Snowmobile is intended to lift and shift an entire data center, where data sizes are in the petabytes or greater.

Transfer appliances are particularly useful for creating hybrid-cloud or multi-cloud setups. For example, Amazon's data transfer appliance (AWS Snowball) supports both import and export. To migrate into a second cloud, users can export their data into a Snowball device, then import it into a second transfer appliance to move data into GCP or Azure. This might sound like an awkward process, but even when it's feasible to push data over the internet between clouds, data egress fees make this an extremely expensive proposition, and physical transfer appliances are a much cheaper alternative when the data volumes are significant.

Keep in mind that transfer appliances and data migration services are one-time data ingestion events, and not suggested for ongoing workloads. If you've got workloads that require constant movement of data in either a hybrid or multi-cloud scenario, your data sizes are presumably batching or streaming much smaller data sizes, on an ongoing basis.

Streaming Ingestion Technologies

There are a few main technologies for ingestion and temporary storage of streaming data. As we discuss streaming technologies, it is more useful to look at the class of all frameworks that collect and buffer messages and consider them in terms of various features they offer. We'll describe major characteristics that data engineers should consider for these systems, and cite a few examples. This discussion is by no means exhaustive—we discuss only a few examples and a handful of standard technologies. But these characteristics are a good framework for research as you choose a streaming ingestion technology. Decide what characteristics you need for your data applications and evaluate technologies according to your needs.

We'd like to note that the terminology in the streaming ingestion space can be confusing. Apache Kafka has variously described itself as a distributed event streaming platform or a distributed commit log. Per Amazon, Kinesis Data Streams "is a serverless streaming data service." RabbitMQ calls itself a message broker. Apache Kafka may or may not be a message queue depending on which author you refer to. And so on. Focus on the bigger picture and context of where various technologies fit in terms of functionality and utility and take vendor descriptions with a grain of salt.

Horizontal scaling

For our purposes, we're mostly interested in streaming ingest frameworks that support horizontal scaling, which grows and shrinks the number of nodes based on the demand placed upon your system. Horizontal scaling enhances data scale, reliability, and durability. In some cases, a single node solution may work just fine for small streams but consider the reliability and durability implications.

Stream partitions

Kafka and Kinesis partition (shard) streams to support horizontal scaling, and they allow users to specify an explicit partition key that uniquely determines which shard a message belongs to. Each consuming server can consume from a specific shard. Google Cloud Pub/Sub hides all details of partitioning; subscribers simply read messages at the level of a topic.

Explicit partitioning is a blessing and a curse, entailing engineering advantages and challenges—see the discussion of *stream partitions* in Chapter 8.

Subscriber pull and push

Kafka and Kinesis only support pull subscriptions. That is, 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.

Operational overhead

As with any data engineering technology, operational overhead is a key consideration. Would you rather manage your streaming data pipelines, or outsource this work to a team of dedicated engineers? On one end of the spectrum, Google Cloud Pub/Sub is a fully managed service with no knobs to turn; just create a topic and a consumer, give a payload, and you're good to go. On the other end, Apache Kafka is packaged in a variety of ways, all the way from raw, hot off the presses open source, to a fully managed serverless offering from Confluent.

Autoscaling

Autoscaling features vary from platform to platform. Confluent Cloud will autoscale up to 100MB/s, after which some intervention is required to set the scale. More effort is required if you run your clusters. Amazon Kinesis Data Streams recently added a feature that automatically scales the number of shards. Pub/Sub is fully autoscaling.

Message size

This is an easily overlooked issue: one must ensure that the streaming framework in question can handle the maximum expected message size. For example, 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.)

Replay

Replay is a key capability in many streaming ingest platforms. Replay allows readers to request a range of messages from the history. A system must support replay for us to truly consider it a streaming *storage* system. RabbitMQ deletes messages once they are consumed by all subscribers—storage capabilities are for temporary buffering only; for long-term storage, a separate tool is required to consume messages and durably write them. Replay also essentially lets us hybridize batch and stream processing in one system.

A key parameter for engineering decisions is maximum message retention time. Google Cloud Pub/Sub supports retention periods of up to 7 days, Amazon Kinesis Data Streams retention can be turned up to 1 year, and 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.)

Fanout

Fanout entails having more than one consumer per data stream. This is very useful in the context of complex streaming applications, where we might

want to feed the same data to multiple targets. Note that this is different from having multiple shards and consuming these on different downstream servers. Fanout entails multiple consumers *per shard*, where each consumer gets its bookmark determining the current position in the stream.

Each streaming ingestion platform has its limits for fanout. For example, a Kinesis Data Stream supports up to 20 consumers.

Exactly-once delivery

In some cases, streaming ingestion systems such as Pub/Sub may send events to consumers more than once. This is known as *at least once delivery* and is a consequence of consistency challenges in a distributed system.

Kafka[1] recently added support for exactly-once delivery. However, before relying on this feature, make sure that you read the documentation carefully to understand the exact configuration requirements and performance implications. Also, it's important to realize that there are various ways for records to be processed multiple times in a streaming system even if your streaming storage system support exactly-once delivery. For example, if a publisher crashes after writing a record into the stream before receiving confirmation that the record was consumed, it may write a second copy of the record when it comes back online. And if a subscriber crashes after processing a record but before confirming to the storage system that it has completed processing, the record may get read a second time.

In general, think through the implications of duplicate records in your stream processing applications. Designing for idempotency[2] will allow your system to properly deduplicate records. On the other hand, an occasional duplicate record may not be an issue in your system.

Record delivery order

The notion of record delivery order is challenging in a distributed system. Amazon Kinesis orders records in single shards, but this does not provide any guarantees for behavior across a whole topic, especially as the number of shards increases. Pub/Sub provides no ordering guarantees; instead,

engineers are advised to use a tool like Google Cloud Dataflow (Apache Beam) if they want to order records.

We give essentially the same advice that we gave in the context of at least once delivery: Understand the ordering characteristics of your streaming ingest system and think through the implications of data arriving out of order.

Stream processing

Some streaming storage systems support direct processing without using an external processing tool. (See Chapter 8.) For example, Kafka supports a variety of operations with the KSQL query language. Google Cloud Pub/Sub and Kinesis Data Streams rely on external tools for processing.

Streaming DAGs

Apache Pulsar has introduced an enhanced notion of stream processing that can be extremely useful for data engineers. With Kafka, engineers can potentially stitch topics together into a DAG (directed acyclic graph) to realize complex data processing, but this would also require custom services and code.

Pulsar builds in streaming DAGs as a core abstraction, supporting complex processing and transformations without ever exiting the Pulsar system.

Multisite and multiregional

It is often desirable to integrate streaming across several locations for enhanced redundancy and to consume data close to where it is generated. For example, it might be desirable to have streaming storage running across several regions to improve latency and throughput for messages sent by an IoT swarm with millions of devices.

An Amazon Kinesis Data Stream runs in a single region. Google Cloud Pub/Sub Global Endpoints allow engineers to create a single endpoint, with Google automatically storing data in the nearest region. While Kafka supports a notion of site-to-site replication through Mirror Maker, the core

architecture of Pulsar is designed to specifically support multi-cluster use cases.

Who You'll Work With

Data ingestion sits at several organizational boundaries. In the development and management of data ingestion pipelines, data engineers will work with both data producers and data consumers.

Upstream Data Producers

In practice, there is often a significant disconnect 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 at all, they usually 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, but also a significant opportunity. Data engineers can improve the quality of the data that they ingest by inviting software engineers to be stakeholders in data engineering outcomes. The vast majority of software engineers are well aware of the value of analytics and data science but don't necessarily have aligned incentives to directly contribute to data engineering efforts.

However, simply improving communication is a great first step. Often, software engineers have already identified potentially valuable data for downstream consumption. Opening a channel of communication encourages software engineers to get data into shape for consumers, and to communicate about data changes to prevent pipeline regressions.

Beyond communication, data engineers can highlight the contributions of software engineers to team members, executives, and especially product managers. Involving product managers in the outcome and treating downstream data processed as part of a product encourages them to allocate

scarce software development to collaboration with data engineers. Ideally, software engineers can work partially as extensions of the data engineering team; this allows them to collaborate on a variety of projects, such as creating an event-driven architecture to enable real-time analytics.

Downstream Data Consumers

Who is the ultimate customer for data ingestion? Data engineers tend to focus on data practitioners and technology leaders such as data scientists, analysts, and chief technical officers. They would do well to also remember their broader circle of business stakeholders such as marketing directors, vice presidents over the supply chain, chief executive officers, etc.

Too often, we see data engineers pursuing sophisticated projects (real-time streaming buses, big data systems) while digital marketing managers next door are left downloading GoogleAds reports manually. View data engineering as a business, and recognize who your customers are. Often, there is significant value in basic automation of ingestion processes, especially for organizations like marketing that control massive budgets and sit at the heart of revenue for the business. Basic ingestion work may seem boring, but delivering value to these core parts of the business will open up more budget and more exciting opportunities for data engineering in the long term.

Data engineers can also invite more executive participation in this collaborative process. For good reason, the notion of data-driven culture is quite fashionable in business leadership circles, but it is up to data engineers and other data practitioners to provide executives with guidance on the best structure for a data-driven business. This means communicating the value of lowering barriers between data producers and data engineers while supporting executives in breaking down silos and setting up incentives that will lead to a more unified data-driven culture.

Once again, *communication* is the watchword. Honest communication early and often with stakeholders will go a long way to making sure your data ingestion adds value.

Undercurrents

Virtually all the undercurrents touch the ingestion phase, but we'll emphasize the most salient ones here.

Security

Moving data introduces security vulnerabilities because you have to move data between locations. The last thing you want is for the data to be captured or compromised while it is being moved.

Consider where the data lives and where it is going. Data that needs to move within your VPC should use secure endpoints, and never leave the confines of the VPC. If you need to send data between the cloud and an on-premises network, use a VPN or a dedicated private connection. This might cost money, but the security is a good investment. If your data traverses the public internet, make sure the transmission is encrypted. Don't ever send data unencrypted over the public internet.

Data Management

Naturally, data management begins at data ingestion. This is the starting point for lineage and data cataloging; from this point on, data engineers need to think about master data management, ethics, privacy, compliance, etc.

Schema changes

Schema changes remain, from our perspective, an unsettled issue in data management. The traditional approach is a careful command and control review process. Working with clients at large enterprises, we have been quoted lead times of six months for the addition of a single field. This is an unacceptable impediment to agility.

On the opposite end of the spectrum is Fivetran, where schema changes are completely automatic. Any schema change in the source triggers target

tables to be recreated with the new schema. This solves schema problems at the ingestion stage, but can still break downstream pipelines.

One possible solution, which the authors have ruminated on for a while, is an approach pioneered by Git version control. When Linus Torvalds was developing Git, many of his choices were inspired by the limitations of CVS (Concurrent Versions System). CVS is completely centralized—it supports only one current official version of the code, stored on a central project server. To make Git a truly distributed system, Torvalds used the notion of a tree, where each developer could maintain their processed branch of the code and then merge to or from other branches.

A few years ago, such an approach to data was unthinkable. Data warehouse systems are typically operated at close to maximum storage capacity. However, in big data and cloud data warehouse environments, storage is cheap. One may quite easily maintain multiple versions of a table with different schemas and even different upstream transformations. Teams can maintain multiple "development" versions of a table using orchestration tools such as Airflow; schema changes, upstream transformation and code changes, etc. can appear in development tables before official changes to the *main* table.

Data ethics, privacy, and compliance

Clients often ask for our advice on encrypting sensitive data in databases. This generally leads us to ask a very basic question: do you need the sensitive data that you're trying to encrypt? As it turns out, in the process of creating requirements and solving problems, this question often gets overlooked.

Data engineers should train themselves to always ask this question when setting up ingestion pipelines. They will inevitably encounter sensitive data; the natural tendency is to ingest it and forward it to the next step in the pipeline. But if this data is not needed, why collect it at all? Why not simply drop sensitive fields before data is stored? Data cannot leak if it is never collected.

Where it is truly necessary to keep track of sensitive identities, it is common practice to apply tokenization to anonymize identities in model training and analytics. But engineers should look at where this tokenization is applied. If possible, hash data at ingestion time.

In some cases, data engineers cannot avoid working with highly sensitive data. Some analytics systems must present identifiable sensitive information. Whenever they handle sensitive data, engineers must act under the highest ethical standards. In addition, they can put in place a variety of practices to reduce the direct handling of sensitive data. Aim as much as possible for *touchless production* where sensitive data is involved. This means that engineers develop and test code on simulated or cleansed data in development and staging environments, but code deployments to production are automated.

Touchless production is an ideal that engineers should strive for, but situations inevitably arise that cannot be fully solved in development and staging environments. Some bugs may not be reproducible without looking at the live data that is triggering a regression. For these cases, put a broken glass process in place, i.e., require at least two people to approve access to sensitive data in the production environment. This access should be tightly scoped to a particular issue, and come with an expiration date.

Our last bit of advice on sensitive data: be wary of naive technological solutions to human problems. Both encryption and tokenization are often treated like privacy magic bullets. Most modern storage systems and databases encrypt data at rest and in motion by default. Generally, we don’t see encryption problems, but data access problems. Is the solution to apply an extra layer of encryption to a single field, or to control access to that field? After all, one must still tightly manage access to the encryption key. There are legitimate use cases for single field encryption, but watch out for ritualistic applications of encryption.

On the tokenization front, use common sense and assess data access scenarios. If someone had the email of one of your customers, could they easily hash the email and find the customer in your data? Thoughtlessly

hashing data without salting and other strategies may not protect privacy as well as you think.

DataOps

Reliable data pipelines are the cornerstone of the data engineering lifecycle. When they fail, all downstream dependencies come to a screeching halt. Data warehouses and data lakes aren't replenished. Data scientists and analysts can't effectively do their jobs; the business is forced to fly blind.

Ensuring your data pipelines are properly monitored is a key step toward reliability and effective incident response. If there's one stage in the data engineering lifecycle where monitoring is critical, it's in the ingestion stage. Weak or nonexistent monitoring means the pipelines may or may not be working. In our work, we've seen countless examples of reports and machine learning models being generated from stale data. In one extreme case, an ingestion pipeline failure wasn't detected for six months. (One might question the concrete utility of the data in this instance.) This was very much avoidable through proper monitoring.

What should you monitor? Uptime, latency, data volumes processed are a good place to start. If an ingestion job fails, how will you respond? This also applies to third-party services. In the case of these services, what you've gained in terms of lean operational efficiencies (reduced headcount) is replaced by systems you depend upon being outside of your control. If you're using a third-party service (cloud, data integration service, etc), how will you be alerted if there's an outage? What's your response plan in the event a service you depend upon suddenly goes offline?

Sadly, there's not a universal response plan for third-party failures. If you can failover to other servers (discussed in the data architecture section as well), preferably in another zone or region, definitely set this up.

If your data ingestion processes are built internally, do you have the correct testing and deployment automation to ensure the code will function in production? And if the code is buggy or fails, can you roll back to a working version?

Data quality tests

We often refer to data as a silent killer. If quality, valid data is the foundation of success in modern business, using bad data to make decisions is much worse than having no data at all; bad data has caused untold damage to businesses.

Data is entropic; it often changes in unexpected ways without warning. One of the inherent differences between DevOps and DataOps is that we only expect software regressions when we deploy changes, while data often presents regressions on its own.

DevOps engineers are typically able to detect problems by using binary conditions. Has the request failure rate breached a certain threshold? How about response latency? In the data space, regressions often manifest as subtle statistical distortions. Is a change in search term statistics a result of customer behavior? Of a spike in bot traffic that has escaped the net? Of a site test tool deployed in some other part of the company?

Like systems failures in the world of DevOps, some data regressions are immediately visible. For example, in the early 2000s, Google provided search terms to websites when users arrived from search. In 2011, “not provided” started appearing as a search term in Google Analytics[3]. Analysts quickly saw “not provided” bubbling to the tops of their reports.

The truly dangerous data regressions are silent and can come from inside or outside a business. Application developers may change the meaning of database fields without adequately communicating with data teams. Changes to data from third-party sources may go unnoticed. In the best-case scenario, reports break in obvious ways. Often, business metrics are distorted unbeknownst to decision-makers.

Traditional data testing tools are generally built on simple binary logic. Are nulls appearing in a non-nullable field? Are new, unexpected items showing up in a categorical column? Statistical data testing is a new realm, but one is likely to grow dramatically in the next five years.

Orchestration

Ingestion generally sits at the beginning of a large and complex data graph; given that ingestion is the first stage of the data engineering lifecycle, ingested data will flow into many more data processing steps, and data from many sources will flow and mingle in complex ways. As we've emphasized throughout this book, orchestration is a key process for coordinating these steps.

Organizations in an early stage of data maturity may choose to deploy ingestion processes as simple scheduled cron jobs. However, it is important to recognize that this approach is brittle and that it can slow the velocity of data engineering deployment and development.

As data pipeline complexity grows, true orchestration is necessary. By true orchestration, we mean a system capable of scheduling full task graphs rather than individual tasks. An orchestration can start each ingestion task at the appropriate scheduled time. Downstream processing and transform steps begin as ingestion tasks are completed. Further downstream, processing steps lead to further processing steps.

Software Engineering

The ingestion stage of the data engineering lifecycle is engineering intensive. This stage sits at the edge of the data engineering domain and often interfaces with external systems, where software and data engineers have to build a variety of custom plumbing.

Behind the scenes, ingestion is incredibly complicated, often with teams operating open-source frameworks like Kafka or Pulsar, or in the case of some of the biggest tech companies, running their own forked or homegrown ingestion solutions. As we've discussed in this chapter, various developments have simplified the ingestion process, such as managed data ingest platforms like Fivetran and Airbyte, and managed cloud services like AWS Lambda and Kinesis. Data engineers should take advantage of the best available tools—especially managed ones that do a lot of the heavy lifting for you—but also develop high competency in software

development. Even for simple serverless functions, it pays to use proper version control and code review processes and to implement appropriate tests.

When writing software, your code needs to be decoupled. Avoid writing monolithic systems that have tight dependencies on the source or destination systems. For example, avoid writing GET queries to an RDBMS with an ORM. This pattern tightly couples your ingestion pull to the ORM, which is tightly coupled to the RDBMS.

Conclusion

Okay, we've made it to the end of ingestion. In your work as a data engineer, ingestion will likely consume a significant part of your energy and effort. At heart, ingestion is plumbing, connecting pipes to other pipes, ensuring that data flows consistently and securely to its destination. At times, the minutiae of ingestion may feel tedious, but without ingestion, the interesting applications of data (analytics, machine learning, etc.) cannot happen.

As we've emphasized, we're also in the midst of a sea change toward streaming data pipelines. This is an opportunity for data engineers to discover interesting applications for streaming data, communicate these to the business and deploy exciting new technologies.

---

1 https://www.confluent.io/blog/simplified-robust-exactly-one-semantics-in-kafka-2-5/

2 https://discourse.getdbt.com/t/understanding-idempotent-data-transformations/518

3 https://martech.org/dark-google-search-terms-not-provided-one-year-later/

About the Authors

Joe Reis is a business-minded data nerd who's worked in the data industry for 20 years, with responsibilities ranging from statistical modeling, forecasting, machine learning, data engineering, data architecture, and almost everything else in between. Joe is the CEO and Co-Founder of Ternary Data, a data engineering and architecture consulting firm based in Salt Lake City, Utah. In addition, he volunteers with several technology groups and teaches at the University of Utah. In his spare time, Joe likes to rock climb, produce electronic music, and take his kids on crazy adventures.

Matt Housley is a data engineering consultant and cloud specialist. After some early programming experience with Logo, Basic and 6502 assembly, he completed a PhD in mathematics at the University of Utah. Matt then began working in data science, eventually specializing in cloud based data engineering. He co-founded Ternary Data with Joe Reis, where he leverages his teaching experience to train future data engineers and advise teams on robust data architecture. Matt and Joe also pontificate on all things data on The Monday Morning Data Chat.