the JSON response to an API call or a MySQL table schema—these are all instances of data modeling and design.

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

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

Data lineage

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

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

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

Data integration and interoperability

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

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

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

Data lifecycle management

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

First, data is increasingly stored in the cloud. This means we have pay-as-you-go storage costs instead of large up-front capital expenditures for an on-premises data lake. When every byte shows up on a monthly AWS statement, CFOs see opportunities for savings. Cloud environments make data archival a relatively straightforward process. Major cloud vendors offer archival-specific object storage classes that allow long-term data retention at an extremely low cost, assuming very infrequent access (it should be noted that data retrieval isn't so cheap, but that's for another conversation). These storage classes also support extra policy controls to prevent accidental or deliberate deletion of critical archives.

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

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

Ethics and privacy

> *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 data's ethical and privacy implications were once considered nice to have, like security, they're now central to the general data lifecycle. Data engineers need to do the right thing when no one else is

watching, because everyone will be watching someday. We hope that more organizations will encourage a culture of good data ethics and privacy.

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

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

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

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

As we discuss orchestration throughout this book, we assume that an orchestration system stays online with high availability. This allows the orchestration system to sense and monitor constantly without human intervention and run new jobs anytime they are deployed. An orchestration

system monitors jobs that it manages and kicks off new tasks as internal DAG dependencies are completed. It can also monitor external systems and tools to watch for data to arrive and criteria to be met. When certain conditions go out of bounds, the system also sets error conditions and sends alerts through email or other channels. You might set an expected completion time of 10 a.m. for overnight daily data pipelines. If jobs are not done by this time, alerts go out to data engineers and consumers.

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

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

Airflow was open source from its inception, and was widely adopted. It was written in Python, making it highly extensible to almost any use case imaginable. While many other interesting open source orchestration projects exist, such as Luigi and Conductor, Airflow is arguably the mindshare leader for the time being. Airflow arrived just as data processing was becoming more abstract and accessible, and engineers were increasingly interested in coordinating complex flows across multiple processors and storage systems, especially in cloud environments.

At this writing, several nascent open source projects aim to mimic the best elements of Airflow's core design while improving on it in key areas. Some

of the most interesting examples are Prefect and Dagster, which aim to improve the portability and testability of DAGs to allow engineers to move from local development to production more easily. Argo is an orchestration engine built around Kubernetes primitives; Metaflow is an open source project out of Netflix that aims to improve data science orchestration.

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

DataOps

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

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

Like DevOps, DataOps borrows much from lean manufacturing and supply chain management, mixing people, processes, and technology to reduce time to value. As Data Kitchen (experts in DataOps) describes it:[6]

DataOps is a collection of technical practices, workflows, cultural norms, and architectural patterns that enable:

- *Rapid innovation and experimentation delivering new insights to customers with increasing velocity*
- *Extremely high data quality and very low error rates*
- *Collaboration across complex arrays of people, technology, and environments*
- *Clear measurement, monitoring, and transparency of results*

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

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

Depending on a company's data maturity, a data engineer has some options to build DataOps into the fabric of the overall data engineering lifecycle. If the company has no preexisting data infrastructure or practices, DataOps is very much a greenfield opportunity that can be baked in from day one. With an existing project or infrastructure that lacks DataOps, a data engineer can begin adding DataOps into workflows. We suggest first starting with observability and monitoring to get a window into the performance of a system, then adding in automation and incident response. A data engineer may work alongside an existing DataOps team to improve the data engineering lifecycle in a data-mature company. In all cases, a data engineer must be aware of the philosophy and technical aspects of DataOps.

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

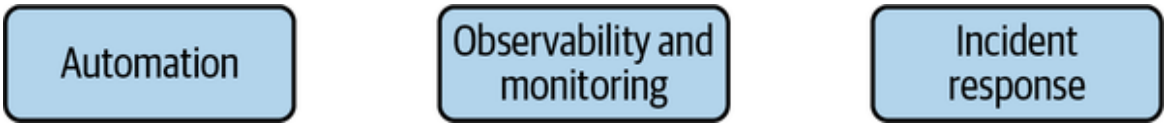

Figure 2-8. The three pillars of DataOps

Automation

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

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

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

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

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

Observability and monitoring

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

Another horror story occurs when the systems that create the data for reports randomly stop working, resulting in reports being delayed by

several days. The data team doesn't know until they're asked by stakeholders why reports are late or producing stale information. Eventually, various stakeholders lose trust in the capabilities of the core data team and start their own splinter teams. The result is many different unstable systems, inconsistent reports, and silos.

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

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

> *The purpose of DODD is to give everyone involved in the data chain visibility into the data and data applications so that everyone involved in the data value chain has the ability to identify changes to the data or data applications at every step—from ingestion to transformation to analysis—to help troubleshoot or prevent data issues. DODD focuses on making data observability a first-class consideration in the data engineering lifecycle.*[7]

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

Incident response

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

identify root causes of an incident and resolve it as reliably and quickly as possible.

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

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

DataOps summary

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

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

Data Architecture

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

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

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

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 MapReduce jobs in C, C++, and Java. At the peak of the big data era (the mid-2010s), engineers started using frameworks that abstracted away these low-level details.

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

Core data processing code

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

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

Development of open source frameworks

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

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

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

Before data engineers begin engineering new internal tools, they would do well to survey the landscape of publicly available tools. Keep an eye on the total cost of ownership (TCO) and opportunity cost associated with implementing a tool. There is a good chance that an open source project

already exists to address the problem they're looking to solve, and they would do well to collaborate rather than reinventing the wheel.

Streaming

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

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

Infrastructure as code

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

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

Pipelines as code

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

General-purpose problem solving

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

Conclusion

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

We break the data engineering lifecycle into the following stages:

- Generation

- Storage
- Ingestion
- Transformation
- Serving data

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

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

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

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

Additional Resources

Managing the data engineering lifecycle:

- "Staying Ahead of Debt" by Etai Mizrahi

Data transformation and processing:

- "Data transformation" Wikipedia page
- "Data processing" Wikipedia page
- "A Comparison of Data Processing Frameworks" by Ludovic Santos

Undercurrents:

- "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing" by Tyler Akidau et al.

DataOps:

- "Getting Started with DevOps Automation" by Jared Murrell
- "Incident Management in the Age of DevOps" Atlassian web page
- "The Seven Stages of Effective Incident Response" Atlassian web page
- "Is DevOps Related to DataOps?" by Carol Jang and Jove Kuang

Data management:

- DAMA International website

Data management and metadata:

- "What Is Metadata" by Michelle Knight

Airbnb Data Portal:

- "Democratizing Data at Airbnb" by Chris Williams et al.
- "Five Steps to Begin Collecting the Value of Your Data" Lean-Data web page

Orchestration:

- "An Introduction to Dagster: The Orchestrator for the Full Data Lifecycle" video by Nick Schrock

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

2 Eryurek, *Data Governance*, 5.

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

4 Eryurek, *Data Governance*, 113.

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

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

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

Chapter 3. Designing Good Data Architecture

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

What Is Data Architecture?

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

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

Enterprise Architecture, Defined

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

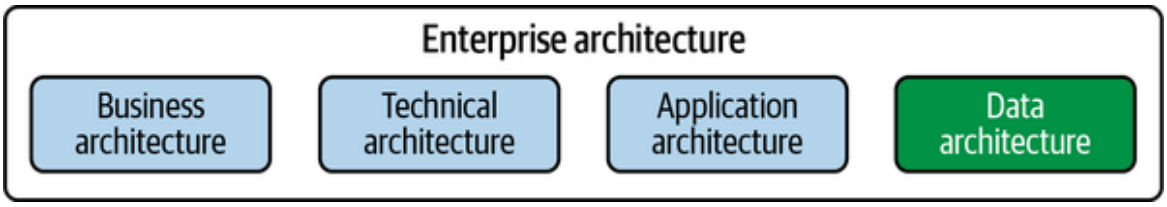

Figure 3-1. Data architecture is a subset of enterprise architecture

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

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

TOGAF's definition

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

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

Gartner's definition

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

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

EABOK's definition

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

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

Our definition

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

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

Here, we touch on some key areas we'll return to throughout the book: flexible and reversible decisions, change management, and evaluation of trade-offs. We discuss each theme at length in this section and then make the definition more concrete in the latter part of the chapter by giving various examples of data architecture.

Flexible and reversible decisions are essential for two reasons. First, the world is constantly changing, and predicting the future is impossible. Reversible decisions allow you to adjust course as the world changes and you gather new information. Second, there is a natural tendency toward enterprise ossification as organizations grow. As organizations grow, they become increasingly risk averse and cumbersome. Adopting a culture of reversible decisions helps overcome this tendency by reducing the risk attached to a decision.

Jeff Bezos is credited with the idea of one-way and two-way doors.[1] A *one-way door* is a decision that is almost impossible to reverse. For example, Amazon could have decided to sell AWS or shut it down. It would be nearly impossible for Amazon to rebuild a public cloud with the same market position after such an action.

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

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

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

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

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

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

Data Architecture Defined

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

TOGAF's definition

TOGAF defines data architecture as follows:

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

DAMA's definition

The DAMA *DMBOK* defines data architecture as follows:

> *Identifying the data needs of the enterprise (regardless of structure) and designing and maintaining the master blueprints to meet those needs. Using master blueprints to guide data integration, control data assets, and align data investments with business strategy.*

Our definition

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

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

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

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

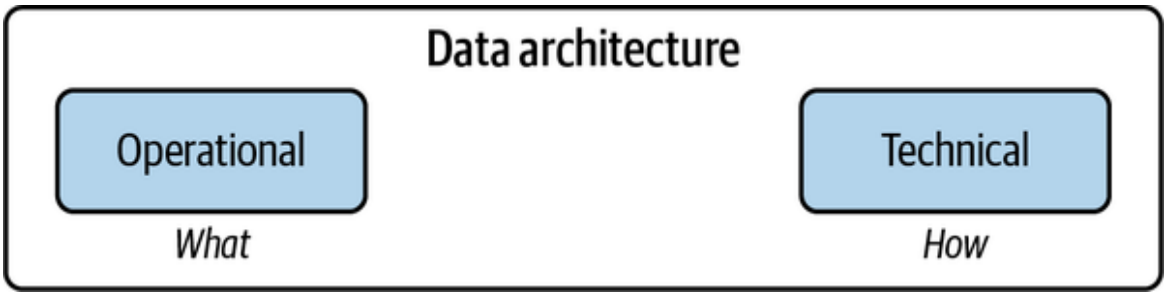

Figure 3-2. Operational and technical data architecture

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

"Good" Data Architecture

> *Never shoot for the best architecture, but rather the least worst architecture.*
>
> —Neal Ford, Mark Richards

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

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

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

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

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

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

Principles of Good Data Architecture

This section takes a 10,000-foot view of good architecture by focusing on principles—key ideas useful in evaluating major architectural decisions and practices. We borrow inspiration for our architecture principles from several

sources, especially the AWS Well-Architected Framework and Google Cloud's Five Principles for Cloud-Native Architecture.

The AWS Well-Architected Framework consists of six pillars:

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

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

- Design for automation.
- Be smart with state.
- Favor managed services.
- Practice defense in depth.
- Always be architecting.

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

1. Choose common components wisely.
2. Plan for failure.
3. Architect for scalability.
4. Architecture is leadership.
5. Always be architecting.

6. Build loosely coupled systems.
7. Make reversible decisions.
8. Prioritize security.
9. Embrace FinOps.

Principle 1: Choose Common Components Wisely

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

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

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

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

Principle 2: Plan for Failure

Everything fails, all the time.

—Werner Vogels

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

Availability

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

Reliability

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

Recovery time objective

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

Recovery point objective

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

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

Principle 3: Architect for Scalability

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

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

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

Principle 4: Architecture Is Leadership

Data architects are responsible for technology decisions and architecture descriptions and disseminating these choices through effective leadership and training. Data architects should be highly technically competent but

delegate most individual contributor work to others. Strong leadership skills combined with high technical competence are rare and extremely valuable. The best data architects take this duality seriously.

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

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

> *In many ways, the most important activity of Architectus Oryzus is to mentor the development team, to raise their level so they can take on more complex issues. Improving the development team's ability gives an architect much greater leverage than being the sole decision-maker and thus running the risk of being an architectural bottleneck.*[2]

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

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

Principle 5: Always Be Architecting

We borrow this principle directly from Google Cloud's Five Principles for Cloud-Native Architecture. Data architects don't serve in their role simply

to maintain the existing state; instead, they constantly design new and exciting things in response to changes in business and technology. Per the EABOK, an architect's job is to develop deep knowledge of the *baseline architecture* (current state), develop a *target architecture*, and map out a *sequencing plan* to determine priorities and the order of architecture changes.

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

Principle 6: Build Loosely Coupled Systems

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

In 2002, Bezos wrote an email to Amazon employees that became known as the Bezos API Mandate:[3]

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter.

5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

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

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

1. Systems are broken into many small components.
2. These systems interface with other services through abstraction layers, such as a messaging bus or an API. These abstraction layers hide and protect internal details of the service, such as a database backend or internal classes and method calls.
3. As a consequence of property 2, internal changes to a system component don't require changes in other parts. Details of code updates are hidden behind stable APIs. Each piece can evolve and improve separately.
4. As a consequence of property 3, there is no waterfall, global release cycle for the whole system. Instead, each component is updated separately as changes and improvements are made.

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

1. Many small teams engineer a large, complex system. Each team is tasked with engineering, maintaining, and improving some system components.

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

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

Principle 7: Make Reversible Decisions

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

As Fowler wrote, "One of an architect's most important tasks is to remove architecture by finding ways to eliminate irreversibility in software designs."[4] What was true when Fowler wrote this in 2003 is just as accurate today.

As we said previously, Bezos refers to reversible decisions as "two-way doors." As he says, "If you walk through and don't like what you see on the

other side, you can't get back to before. We can call these Type 1 decisions. But most decisions aren't like that—they are changeable, reversible—they're two-way doors." Aim for two-way doors whenever possible.

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

Principle 8: Prioritize Security

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

Hardened-perimeter and zero-trust security models

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

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

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

For at least a decade, alarming media reports have made us aware of the growing menace of security breaches that exploit human targets inside hardened organizational security perimeters. Even as employees work on

highly secure corporate networks, they remain connected to the outside world through email and mobile devices. External threats effectively become internal threats.

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

The shared responsibility model

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

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

AWS users are responsible for security in the cloud:

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

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

Data engineers as security engineers

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

the hard security perimeter, all data engineers should consider themselves security engineers.

Failure to assume these new implicit responsibilities can lead to dire consequences. Numerous data breaches have resulted from the simple error of configuring Amazon S3 buckets with public access.[5] Those who handle data must assume that they are ultimately responsible for securing it.

Principle 9: Embrace FinOps

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

> *FinOps is an evolving cloud financial management discipline and cultural practice that enables organizations to get maximum business value by helping engineering, finance, technology, and business teams to collaborate on data-driven spending decisions.*

In addition, J.R. Sorment and Mike Fuller provide the following definition in *Cloud FinOps* (O'Reilly):

> *The term "FinOps" typically refers to the emerging professional movement that advocates a collaborative working relationship between DevOps and Finance, resulting in an iterative, data-driven management of infrastructure spending (i.e., lowering the unit economics of cloud) while simultaneously increasing the cost efficiency and, ultimately, the profitability of the cloud environment.*

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

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

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

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

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

As of this writing, FinOps is a recently formalized practice. The FinOps Foundation was started only in 2019.[6] However, we highly recommend you start thinking about FinOps early, before you encounter high cloud bills. Start your journey with the FinOps Foundation and O'Reilly's *Cloud FinOps*. We also suggest that data engineers involve themselves in the community process of creating FinOps practices for data engineering— in such a new practice area, a good deal of territory is yet to be mapped out.

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

Major Architecture Concepts

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

Domains and Services

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

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

A domain can contain multiple services. For example, you might have a sales domain with three services: orders, invoicing, and products. Each service has particular tasks that support the sales domain. Other domains may also share services (Figure 3-3). In this case, the accounting domain is responsible for basic accounting functions: invoicing, payroll, and accounts receivable (AR). Notice the accounting domain shares the invoice service with the sales domain since a sale generates an invoice, and accounting must keep track of invoices to ensure that payment is received. Sales and accounting own their respective domains.

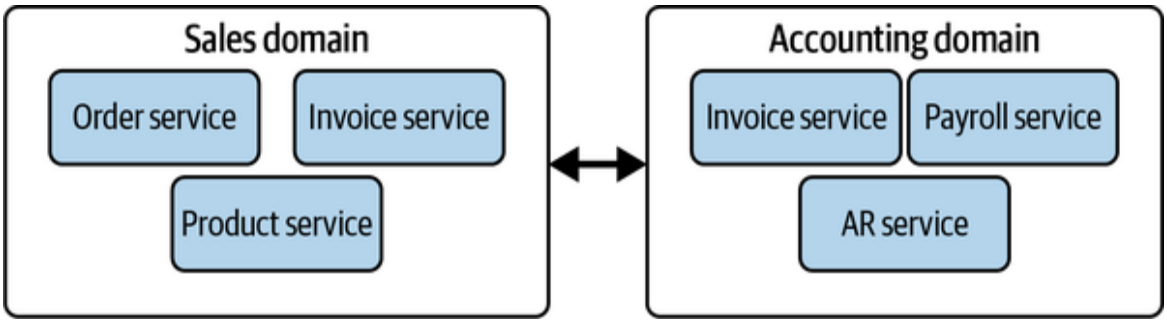

Figure 3-3. Two domains (sales and accounting) share a common service (invoices), and sales and accounting own their respective domains

When thinking about what constitutes a domain, focus on what the domain represents in the real world and work backward. In the preceding example, the sales domain should represent what happens with the sales function in your company. When architecting the sales domain, avoid cookie-cutter copying and pasting from what other companies do. Your company's sales function likely has unique aspects that require specific services to make it work the way your sales team expects.

Identify what should go in the domain. When determining what the domain should encompass and what services to include, the best advice is to simply go and talk with users and stakeholders, listen to what they're saying, and build the services that will help them do their job. Avoid the classic trap of architecting in a vacuum.

Distributed Systems, Scalability, and Designing for Failure

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

Scalability

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

Elasticity

The ability of a scalable system to scale dynamically; a highly elastic system can automatically scale up and down based on the current workload. Scaling up is critical as demand increases, while scaling down saves money in a cloud environment. Modern systems sometimes scale to zero, meaning they can automatically shut down when idle.

Availability

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

Reliability

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

TIP

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

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

Scalability can be realized in a variety of ways. For your services and domains, does a single machine handle everything? A single machine can be scaled vertically; you can increase resources (CPU, disk, memory, I/O). But there are hard limits to possible resources on a single machine. Also, what happens if this machine dies? Given enough time, some components will eventually fail. What's your plan for backup and failover? Single machines generally can't offer high availability and reliability.

We utilize a distributed system to realize higher overall scaling capacity and increased availability and reliability. *Horizontal scaling* allows you to add more machines to satisfy load and resource requirements (Figure 3-4). Common horizontally scaled systems have a leader node that acts as the main point of contact for the instantiation, progress, and completion of workloads. When a workload is started, the leader node distributes tasks to the worker nodes within its system, completing the tasks and returning the results to the leader node. Typical modern distributed architectures also build in redundancy. Data is replicated so that if a machine dies, the other machines can pick up where the missing server left off; the cluster may add more machines to restore capacity.

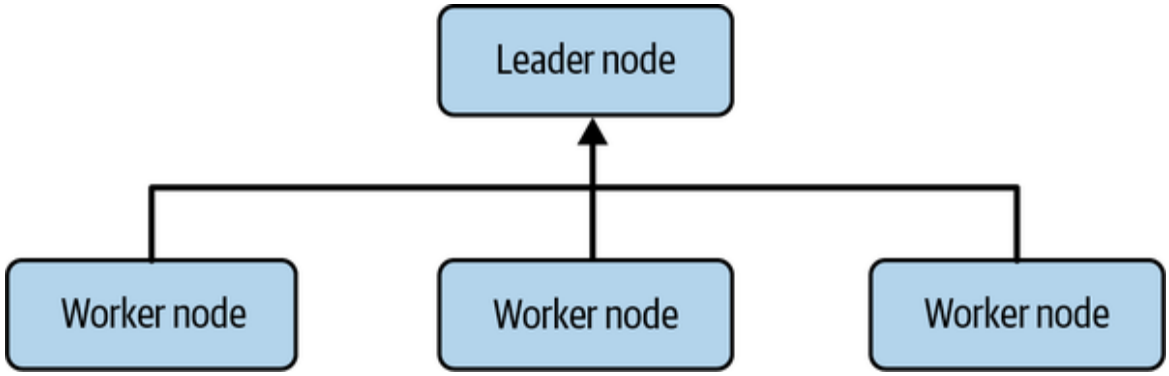

Figure 3-4. A simple horizontal distributed system utilizing a leader-follower architecture, with one leader node and three worker nodes

Distributed systems are widespread in the various data technologies you'll use across your architecture. Almost every cloud data warehouse object storage system you use has some notion of distribution under the hood. Management details of the distributed system are typically abstracted away, allowing you to focus on high-level architecture instead of low-level plumbing. However, we highly recommend that you learn more about distributed systems because these details can be extremely helpful in understanding and improving the performance of your pipelines; Martin Kleppmann's *Designing Data-Intensive Applications* (O'Reilly) is an excellent resource.

Tight Versus Loose Coupling: Tiers, Monoliths, and Microservices

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

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

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

Architecture tiers

As you develop your architecture, it helps to be aware of architecture tiers. Your architecture has layers—data, application, business logic, presentation, and so forth—and you need to know how to decouple these layers. Because tight coupling of modalities presents obvious vulnerabilities, keep in mind how you structure the layers of your architecture to achieve maximum reliability and flexibility. Let's look at single-tier and multitier architecture.

Single tier

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

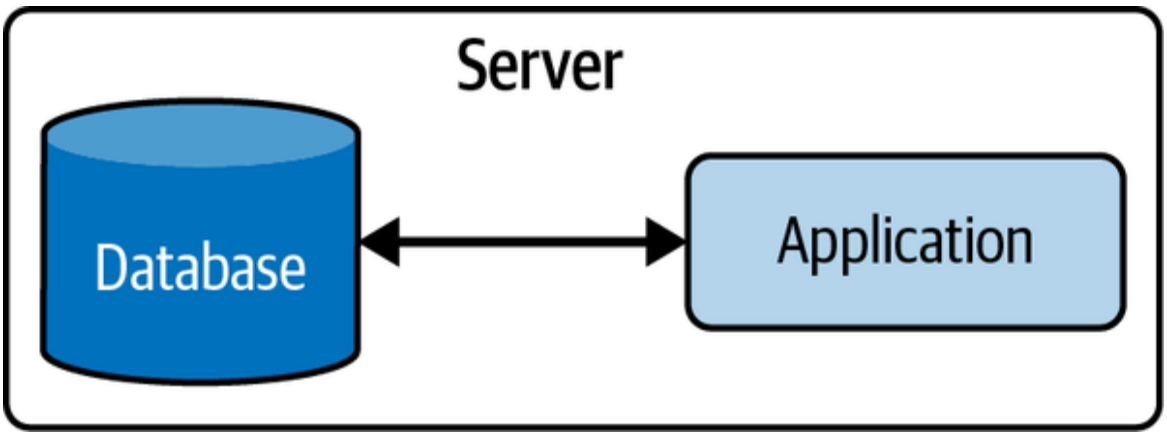

Figure 3-5. Single-tier architecture

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

Multitier

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

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

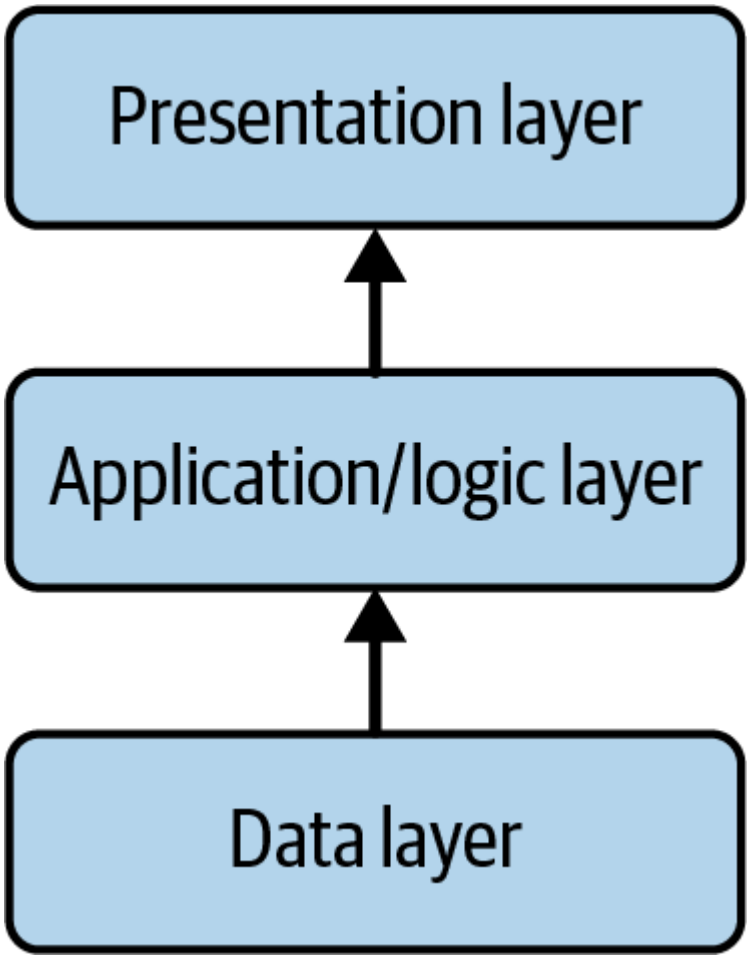

Figure 3-6. A three-tier architecture

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

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