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

Data Engineers and Other Technical Roles

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

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

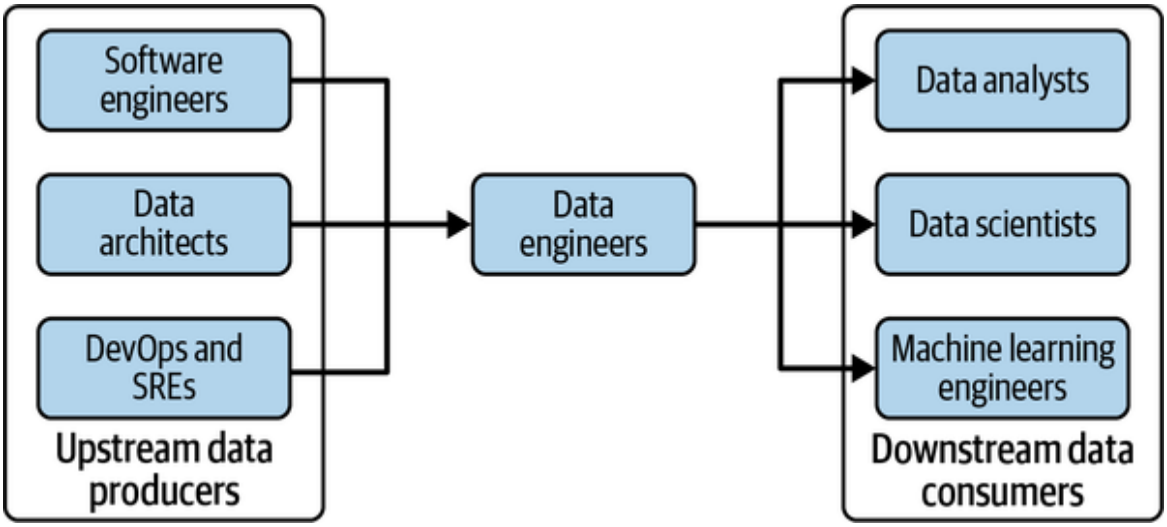

Figure 1-12. Key technical stakeholders of data engineering

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

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

Upstream stakeholders

To be successful as a data engineer, you need to understand the data architecture you're using or designing and the source systems producing the data you'll need. Next, we discuss a few familiar upstream stakeholders: data architects, software engineers, and DevOps engineers.

Data architects

Data architects function at a level of abstraction one step removed from data engineers. Data architects design the blueprint for organizational data management, mapping out processes and overall data architecture and systems.[11] They also serve as a bridge between an organization's technical and nontechnical sides. Successful data architects generally have "battle scars" from extensive engineering experience, allowing them to guide and assist engineers while successfully communicating engineering challenges to nontechnical business stakeholders.

Data architects implement policies for managing data across silos and business units, steer global strategies such as data management and data governance, and guide significant initiatives. Data architects often play a central role in cloud migrations and greenfield cloud design.

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

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

Note that we have placed data architects in the *upstream stakeholders* section. Data architects often help design application data layers that are source systems for data engineers. Architects may also interact with data

engineers at various other stages of the data engineering lifecycle. We cover "good" data architecture in Chapter 3.

Software engineers

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

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

DevOps engineers and site-reliability engineers

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

Downstream stakeholders

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

Data scientists

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

According to common industry folklore, data scientists spend 70% to 80% of their time collecting, cleaning, and preparing data.[12] In our experience, these numbers often reflect immature data science and data engineering practices. In particular, many popular data science frameworks can become bottlenecks if they are not scaled up appropriately. Data scientists who work exclusively on a single workstation force themselves to downsample data, making data preparation significantly more complicated and potentially compromising the quality of the models they produce. Furthermore, locally developed code and environments are often difficult to deploy in production, and a lack of automation significantly hampers data science workflows. If data engineers do their job and collaborate successfully, data scientists shouldn't spend their time collecting, cleaning, and preparing data after initial exploratory work. Data engineers should automate this work as much as possible.

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

Data analysts

Data analysts (or business analysts) seek to understand business performance and trends. Whereas data scientists are forward-looking, a data analyst typically focuses on the past or present. Data analysts usually run SQL queries in a data warehouse or a data lake. They may also utilize spreadsheets for computation and analysis and various BI tools such as

Microsoft Power BI, Looker, or Tableau. Data analysts are domain experts in the data they work with frequently and become intimately familiar with data definitions, characteristics, and quality problems. A data analyst's typical downstream customers are business users, management, and executives.

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

Machine learning engineers and AI researchers

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

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

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

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

AI researchers work on new, advanced ML techniques. AI researchers may work inside large technology companies, specialized intellectual property startups (OpenAI, DeepMind), or academic institutions. Some practitioners are dedicated to part-time research in conjunction with ML engineering responsibilities inside a company. Those working inside specialized ML labs are often 100% dedicated to research. Research problems may target immediate practical applications or more abstract demonstrations of AI. AlphaGo and GPT-3/GPT-4 are great examples of ML research projects. We've provided some references in "Additional Resources".

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

Data Engineers and Business Leadership

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

Data in the C-suite

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

Chief executive officer

Chief executive officers (CEOs) at nontech companies generally don't concern themselves with the nitty-gritty of data frameworks and software. Instead, they define a vision in collaboration with technical C-suite roles and company data leadership. Data engineers provide a window into what's possible with data. Data engineers and their managers maintain a map of what data is available to the organization—both internally and from third parties—in what time frame. They are also tasked to study primary data architectural changes in collaboration with other engineering roles. For example, data engineers are often heavily involved in cloud migrations, migrations to new data systems, or deployment of streaming technologies.

Chief information officer

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

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

Chief technology officer

A chief technology officer (CTO) is similar to a CIO but faces outward. A CTO owns the key technological strategy and architectures for 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 and system architecture. In some organizations without a CIO, the CTO or sometimes the chief operating officer (COO) plays the role of CIO. Data engineers often report directly or indirectly through a CTO.

Chief data officer

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

Chief analytics officer

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

Chief algorithms officer

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

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

Data engineers and project managers

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

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

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

Data engineers and product managers

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

Data engineers and other management roles

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

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

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

Conclusion

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

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

We hope that this first chapter has whetted your appetite, whether you are a software development practitioner, data scientist, ML engineer, business stakeholder, entrepreneur, or venture capitalist. Of course, a great deal still remains to elucidate in subsequent chapters. Chapter 2 covers the data engineering lifecycle, followed by architecture in Chapter 3. The following

chapters get into the nitty-gritty of technology decisions for each part of the lifecycle. The entire data field is in flux, and as much as possible, each chapter focuses on the *immutables*—perspectives that will be valid for many years amid relentless change.

Additional Resources

- "On Complexity in Big Data" by Jesse Anderson (O'Reilly)
- "Which Profession Is More Complex to Become, a Data Engineer or a Data Scientist?" thread on Quora
- "The Future of Data Engineering Is the Convergence of Disciplines" by Liam Hausmann
- The Information Management Body of Knowledge website
- "Doing Data Science at Twitter" by Robert Chang
- "A Short History of Big Data" by Mark van Rijmenam
- "Data Engineering: A Quick and Simple Definition" by James Furbush (O'Reilly)
- "Big Data Will Be Dead in Five Years" by Lewis Gavin
- "The AI Hierarchy of Needs" by Monica Rogati
- Chapter 1 of*What Is Data Engineering?* by Lewis Gavin (O'Reilly)
- "The Three Levels of Data Analysis: A Framework for Assessing Data Organization Maturity" by Emilie Schario
- "Data as a Product vs. Data as a Service" by Justin Gage
- "The Downfall of the Data Engineer" by Maxime Beauchemin
- "The Rise of the Data Engineer" by Maxime Beauchemin
- "Skills of the Data Architect" by Bob Lambert

- "What Is a Data Architect? IT's Data Framework Visionary" by Thor Olavsrud
- "OpenAI's New Language Generator GPT-3 Is Shockingly Good—and Completely Mindless" by Will Douglas Heaven
- The AlphaGo research web page
- "How CEOs Can Lead a Data-Driven Culture" by Thomas H. Davenport and Nitin Mittal
- "Why CEOs Must Lead Big Data Initiatives" by John Weathington
- "How Creating a Data-Driven Culture Can Drive Success" by Frederik Bussler
- *Building Analytics Teams* by John K. Thompson (Packt)
- *Data Teams* by Jesse Anderson (Apress)
- "Information Management Body of Knowledge" Wikipedia page
- "Information management" Wikipedia page

---

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 2. The Data Engineering Lifecycle

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

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

What Is the Data Engineering Lifecycle?

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

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

- Generation
- Storage
- Ingestion

- Transformation
- Serving data

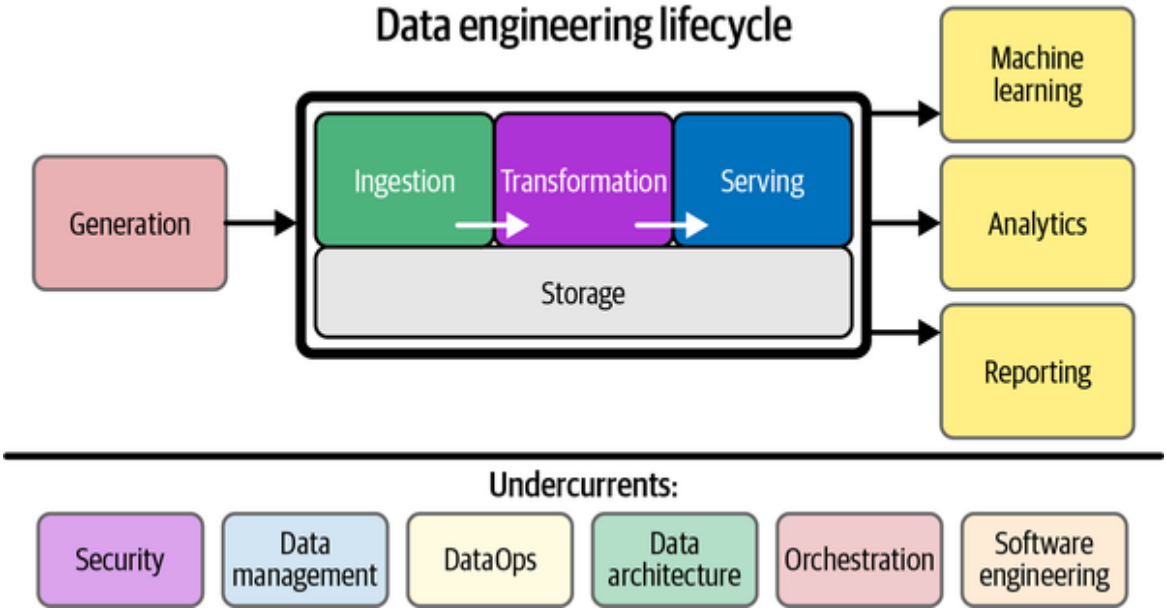

Figure 2-1. Components and undercurrents of the data engineering lifecycle

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

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

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

The Data Lifecycle Versus the Data Engineering Lifecycle

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

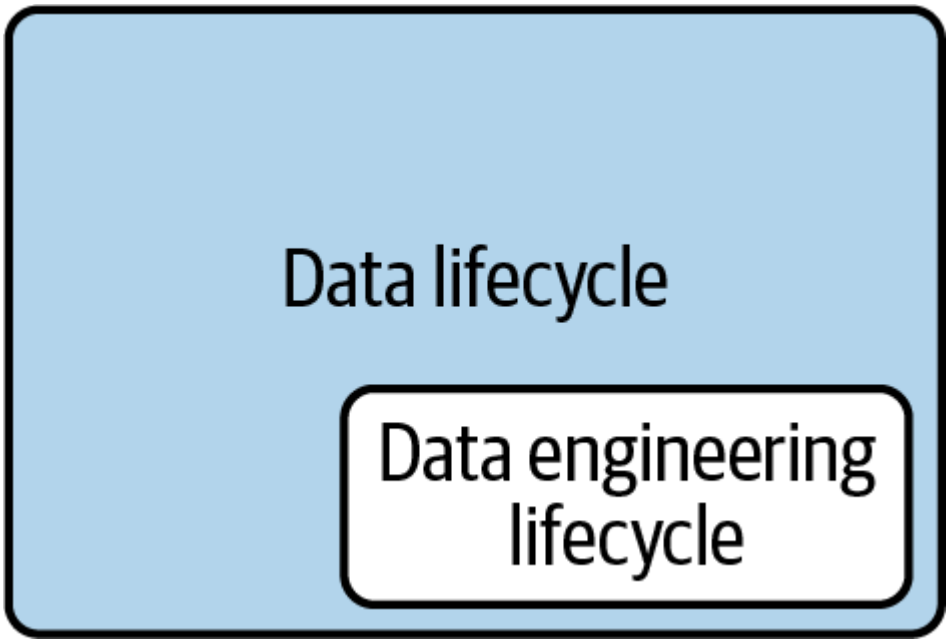

Figure 2-2. The data engineering lifecycle is a subset of the full data lifecycle

Generation: Source Systems

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

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

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

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

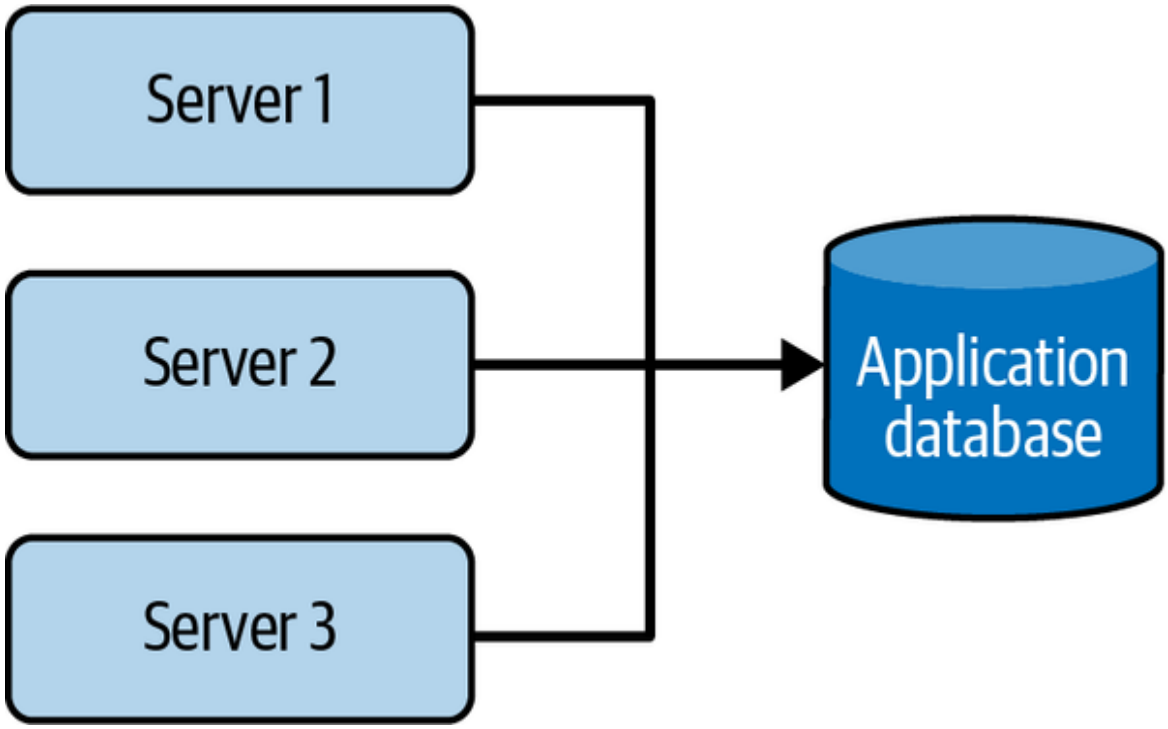

Figure 2-3. Source system example: an application database

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

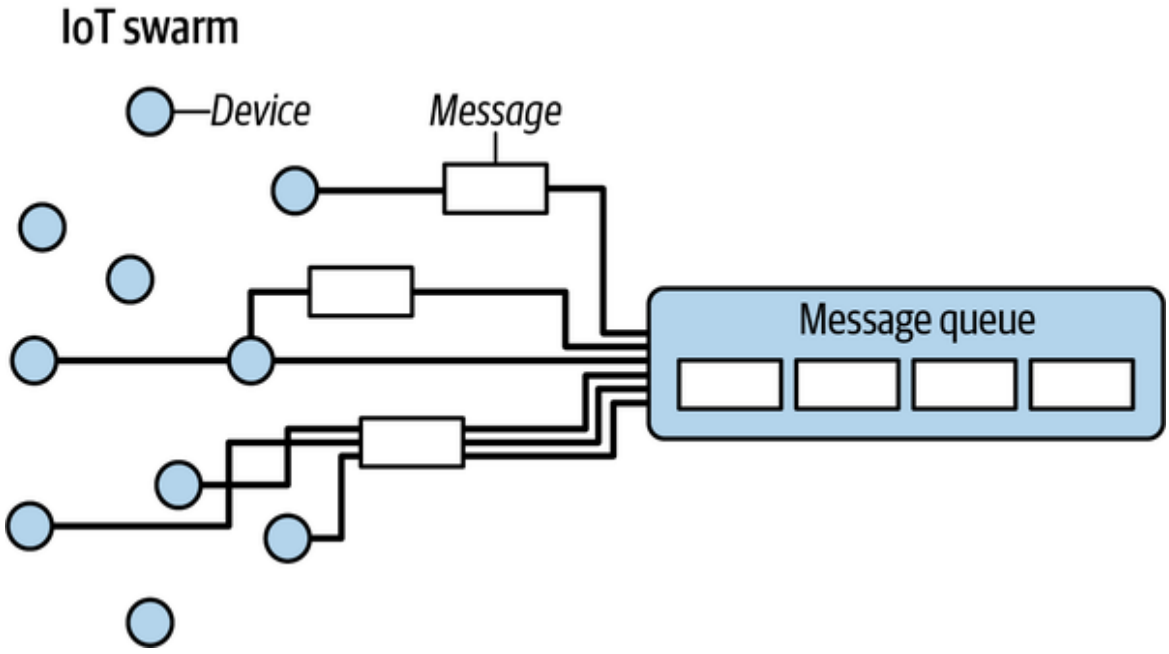

Figure 2-4. Source system example: an IoT swarm and messaging queue

Evaluating source systems: Key engineering considerations

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

- What are the essential characteristics of the data source? Is it an application? A swarm of IoT devices?
- How is data persisted in the source system? Is data persisted long term, or is it temporary and quickly deleted?
- At what rate is data generated? How many events per second? How many gigabytes per hour?

- What level of consistency can data engineers expect from the output data? If you're running data-quality checks against the output data, how often do data inconsistencies occur—nulls where they aren't expected, lousy formatting, etc.?
- How often do errors occur?
- Will the data contain duplicates?
- Will some data values arrive late, possibly much later than other messages produced simultaneously?
- What is the schema of the ingested data? Will data engineers need to join across several tables or even several systems to get a complete picture of the data?
- If schema changes (say, a new column is added), how is this dealt with and communicated to downstream stakeholders?
- How frequently should data be pulled from the source system?
- For stateful systems (e.g., a database tracking customer account information), is data provided as periodic snapshots or update events from change data capture (CDC)? What's the logic for how changes are performed, and how are these tracked in the source database?
- Who/what is the data provider that will transmit the data for downstream consumption?
- Will reading from a data source impact its performance?
- Does the source system have upstream data dependencies? What are the characteristics of these upstream systems?
- Are data-quality checks in place to check for late or missing data?

Sources produce data consumed by downstream systems, including human-generated spreadsheets, IoT sensors, and web and mobile applications. Each source has its unique volume and cadence of data generation. A data engineer should know how the source generates data, including relevant

quirks or nuances. Data engineers also need to understand the limits of the source systems they interact with. For example, will analytical queries against a source application database cause resource contention and performance issues?

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

Schemaless doesn't mean the absence of schema. Rather, it means that the application defines the schema as data is written, whether to a 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, to which application writes must conform.

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

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

Storage

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, data architectures in the cloud often leverage *several* storage solutions. Second, few data storage solutions function purely as storage, with many supporting complex transformation queries; even object storage solutions may support powerful query capabilities—e.g., Amazon S3

Select. Third, while storage is a stage of the data engineering lifecycle, it frequently touches on other stages, such as ingestion, transformation, and serving.

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

Evaluating storage systems: Key engineering considerations

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

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

- Are you capturing metadata about schema evolution, data flows, data lineage, and so forth? Metadata has a significant impact on the utility of data. Metadata represents an investment in the future, dramatically enhancing discoverability and institutional knowledge to streamline future projects and architecture changes.
- Is this a pure storage solution (object storage), or does it support complex query patterns (i.e., a cloud data warehouse)?
- Is the storage system schema-agnostic (object storage)? Flexible schema (Cassandra)? Enforced schema (a cloud data warehouse)?
- How are you tracking master data, golden records data quality, and data lineage for data governance? (We have more to say on these in "Data Management".)
- How are you handling regulatory compliance and data sovereignty? For example, can you store your data in certain geographical locations but not others?

Understanding data access frequency

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

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

Cold data is seldom queried and is appropriate for storing in an archival system. Cold data is often retained for compliance purposes or in case of a catastrophic failure in another system. In the "old days," cold data would be stored on tapes and shipped to remote archival facilities. In cloud

environments, vendors offer specialized storage tiers with very cheap monthly storage costs but high prices for data retrieval.

Selecting a storage system

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

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

Ingestion

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

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

Key engineering considerations for the ingestion phase

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

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

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

Batch versus streaming

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

Streaming ingestion allows us to provide data to downstream systems—whether other applications, databases, or analytics systems—in a continuous, real-time fashion. Here, *real-time* (or *near real-time*) means that the data is available to a downstream system a short time after it is

produced (e.g., less than one second later). The latency required to qualify as real-time varies by domain and requirements.

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

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

Key considerations for batch versus stream ingestion

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

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

batch?

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

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

Push versus pull

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

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

schedule. You'll learn more about ETL and extract, load, transform (ELT) throughout this book.

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

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

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

Transformation

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

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

Key considerations for the transformation phase

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

- What's the cost and return on investment (ROI) of the transformation? What is the associated business value?
- Is the transformation as simple and self-isolated as possible?
- What business rules do the transformations support?
- Am I minimizing data movement between the transformation and the storage system during transformation?

You can transform data in batch or while streaming in flight. As mentioned in "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 and the general increase in the amount of streaming data, we expect the popularity of streaming transformations to continue growing, perhaps entirely replacing batch processing in certain domains soon.

Logically, we treat transformation as a standalone area of the data engineering lifecycle, but the realities of the lifecycle can be much more complicated in practice. Transformation is often entangled in other phases of the lifecycle. Typically, data is transformed in source systems or in flight during ingestion. For example, a source system may add an event timestamp to a record before forwarding it to an ingestion process. Or a record within a streaming pipeline may be "enriched" with additional fields

and calculations before it's sent to a data warehouse. Transformations are ubiquitous in various parts of the lifecycle. Data preparation, data wrangling, and cleaning—these transformative tasks add value for end consumers of data.

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

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

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

Serving Data

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

Data has *value* when it's used for practical purposes. Data that is not consumed or queried is simply inert. Data vanity projects are a major risk for companies. Many companies pursued vanity projects in the big data era,

gathering massive datasets in data lakes that were never consumed in any useful way. The cloud era is triggering a new wave of vanity projects built on the latest data warehouses, object storage systems, and streaming technologies. Data projects must be intentional across the lifecycle. What is the ultimate business purpose of the data so carefully collected, cleaned, and stored?

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

Analytics

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

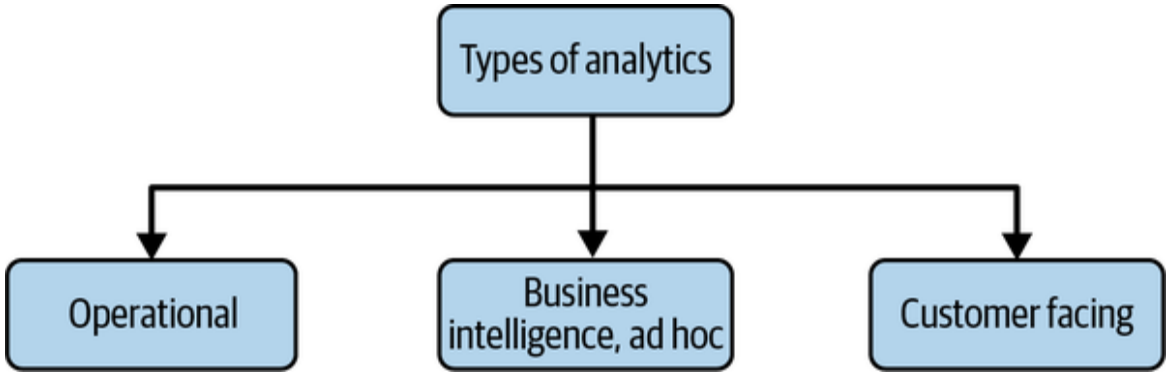

Figure 2-5. Types of analytics

Business intelligence

BI marshals collected data to describe a business's past and current state. BI requires using business logic to process raw data. Note that data serving for analytics is yet another area where the stages of the data engineering lifecycle can get tangled. As we mentioned earlier, business logic is often applied to data in the transformation stage of the data engineering lifecycle, but a logic-on-read approach has become increasingly popular. Data is

stored in a clean but fairly raw form, with minimal postprocessing business logic. A BI system maintains a repository of business logic and definitions. This business logic is used to query the data warehouse so that reports and dashboards align with business definitions.

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

Operational analytics

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

Embedded analytics

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

With 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. An internal data-access error at a company would likely lead to a procedural review. A data leak between customers would be considered a massive breach of trust, leading to media attention and a significant loss of customers. Minimize your blast radius related to data leaks and security vulnerabilities. Apply tenant- or data-level security within your storage, and anywhere there's a possibility of data leakage.

MULTITENANCY

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

Machine learning

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

The responsibilities of data engineers overlap significantly in analytics and ML, and the boundaries between data engineering, ML engineering, and analytics engineering can be fuzzy. For example, a data engineer may need to support Spark clusters that facilitate analytics pipelines and ML model training. They may also need to provide a system that orchestrates tasks

across teams and support metadata and cataloging systems that track data history and lineage. Setting these domains of responsibility and the relevant reporting structures is a critical organizational decision.

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

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

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

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

- Is the data of sufficient quality to perform reliable feature engineering? Quality requirements and assessments are developed in close collaboration with teams consuming the data.
- Is the data discoverable? Can data scientists and ML engineers easily find valuable data?

- Where are the technical and organizational boundaries between data engineering and ML engineering? This organizational question has significant architectural implications.
- Does the dataset properly represent ground truth? Is it unfairly biased?

While ML is exciting, our experience is that companies often prematurely dive into it. Before investing a ton of resources into ML, take the time to build a solid data foundation. This means setting up the best systems and architecture across the data engineering and ML lifecycle. It's generally best to develop competence in analytics before moving to ML. Many companies have dashed their ML dreams because they undertook initiatives without appropriate foundations.

Reverse ETL

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

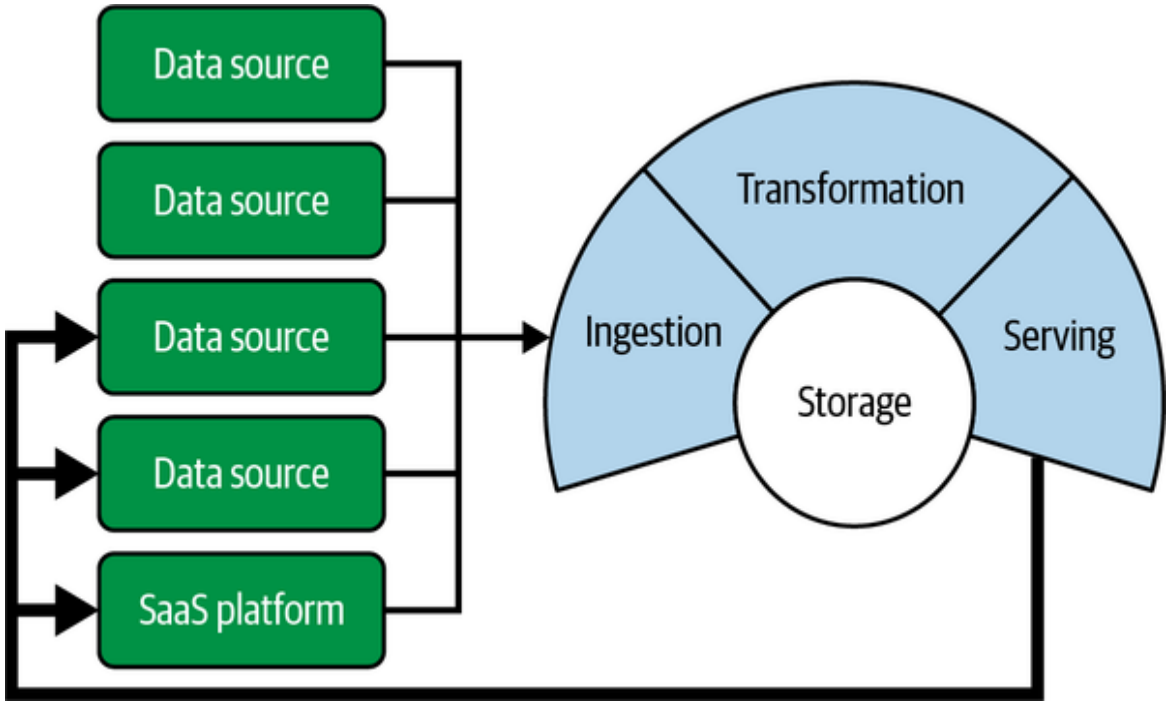

Figure 2-6. Reverse ETL

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

As we've written this book, several vendors have embraced the concept of reverse ETL and built products around it, such as Hightouch and Census. Reverse ETL remains nascent as a 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, companies may want to push specific metrics from their data warehouse to a customer data platform or CRM system. Advertising platforms are another everyday use case, as in the Google Ads example. Expect to see more activity in reverse ETL, with an overlap in both data engineering and ML engineering.

The jury is out on whether the term *reverse ETL* will stick. And the practice may evolve. Some engineers claim that we can eliminate reverse ETL by handling data transformations in an event stream and sending those events back to source systems as needed. Realizing widespread adoption of this

pattern across businesses is another matter. The gist is that transformed data will need to be returned to source systems in some manner, ideally with the correct lineage and business process associated with the source system.

Major Undercurrents Across the Data Engineering Lifecycle

Data engineering is rapidly maturing. Whereas prior cycles of data engineering simply focused on the technology layer, the continued abstraction and simplification of tools and practices have shifted this focus. Data engineering now encompasses far more than tools and technology. The field is now moving up the value chain, incorporating traditional enterprise practices such as data management and cost optimization, and newer practices like DataOps.

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

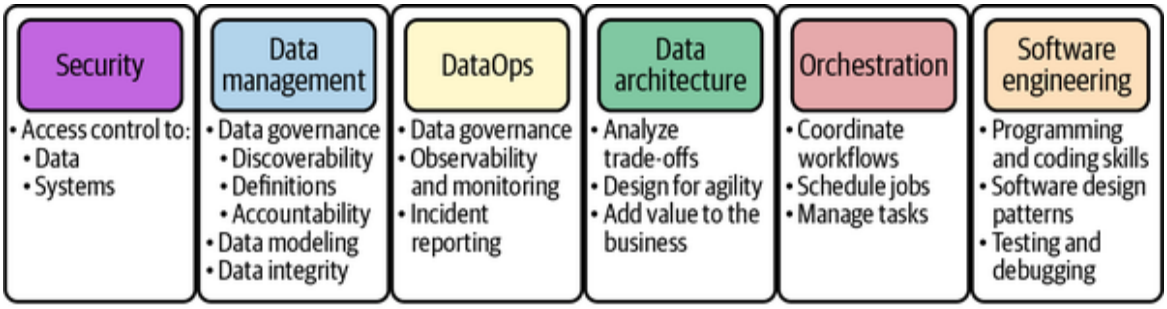

Figure 2-7. The major undercurrents of data engineering

Security

Security must be top of mind for data engineers, and those who ignore it do so at their peril. That's why security is the first undercurrent. Data engineers must understand both data and access security, exercising the principle of

least privilege. The principle of least privilege means giving a user or system access to only the essential data and resources to perform an intended function. A common antipattern we see with data engineers with little security experience is to give admin access to all users. This is a catastrophe waiting to happen!

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

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

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

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

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

Data Management

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

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

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

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

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

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

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

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

Data governance

According to *Data Governance: The Definitive Guide,* "Data governance is, first and foremost, a data management function to ensure the quality, integrity, security, and usability of the data collected by an organization."[1]

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

Think of the typical example of data governance being done poorly. A business analyst gets a request for a report but doesn't know what data to use to answer the question. They may spend hours digging through dozens of tables in a transactional database, wildly guessing at which fields might be useful. The analyst compiles a "directionally correct" report but isn't entirely sure that the report's underlying data is accurate or sound. The recipient of the report also questions the validity of the data. The integrity of the analyst—and of all data in the company's systems—is called into question. The company is confused about its performance, making business planning impossible.

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

The core categories of data governance are discoverability, security, and accountability.[2] Within these core categories are subcategories, such as data quality, metadata, and privacy. Let's look at each core category in turn.

Discoverability

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

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

Metadata

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

We divide metadata into two major categories: autogenerated and human generated. Modern data engineering revolves around automation, but

metadata collection is often manual and error prone.

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

Metadata becomes a byproduct of data and data processes. However, key challenges remain. In particular, interoperability and standards are still lacking. Metadata tools are only as good as their connectors to data systems and their ability to share metadata. In addition, automated metadata tools should not entirely take humans out of the loop.

Data has a social element; each organization accumulates social capital and knowledge around processes, datasets, and pipelines. Human-oriented metadata systems focus on the social aspect of metadata. This is something that Airbnb has emphasized in its various blog posts on data tools, particularly its original Dataportal concept.[3] Such tools should provide a place to disclose data owners, data consumers, and domain experts. Documentation and internal wiki tools provide a key foundation for metadata management, but these tools should also integrate with automated data cataloging. For example, data-scanning tools can generate wiki pages with links to relevant data objects.

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

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

- Business metadata

- Technical metadata
- Operational metadata
- Reference metadata

Let's briefly describe each category of metadata.

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

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

Technical metadata describes the data created and used by systems across the data engineering lifecycle. It includes the data model and schema, data lineage, field mappings, and pipeline workflows. A data engineer uses technical metadata to create, connect, and monitor various systems across the data engineering lifecycle.

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

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

Orchestration is a central hub that coordinates workflow across various systems. *Pipeline metadata* captured in orchestration systems provides

details of the workflow schedule, system and data dependencies, configurations, connection details, and much more.

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

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

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

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

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

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

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

Data accountability

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

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

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

Data quality

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

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

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

According to *Data Governance: The Definitive Guide*, data quality is defined by three main characteristics:[4]

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

A variety of interesting problems arise concerning completeness and timeliness. In the Google paper introducing the Dataflow model, the authors give the example of an offline video platform that displays ads.[5] The platform downloads video and ads while a connection is present, allows the user to watch these while offline, and then uploads ad view data once a connection is present again. This data may arrive late, well after the ads are watched. How does the platform handle billing for the ads?

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

Data quality sits across the boundary of human and technology problems. Data engineers need robust processes to collect actionable human feedback on data quality and use technology tools to detect quality issues

preemptively before downstream users ever see them. We cover these collection processes in the appropriate chapters throughout this book.

MASTER DATA MANAGEMENT

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

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

MDM reaches across the full data cycle into operational databases. It may fall directly under the purview of data engineering, but is often the assigned responsibility of a dedicated team that works across the organization. Even if they don't own MDM, data engineers must always be aware of it, as they will collaborate on MDM initiatives.

Data modeling and design

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