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 five stages (Figure 2-1, top):

- Generation
- Storage
- Ingestion
- Transformation
- Serving data

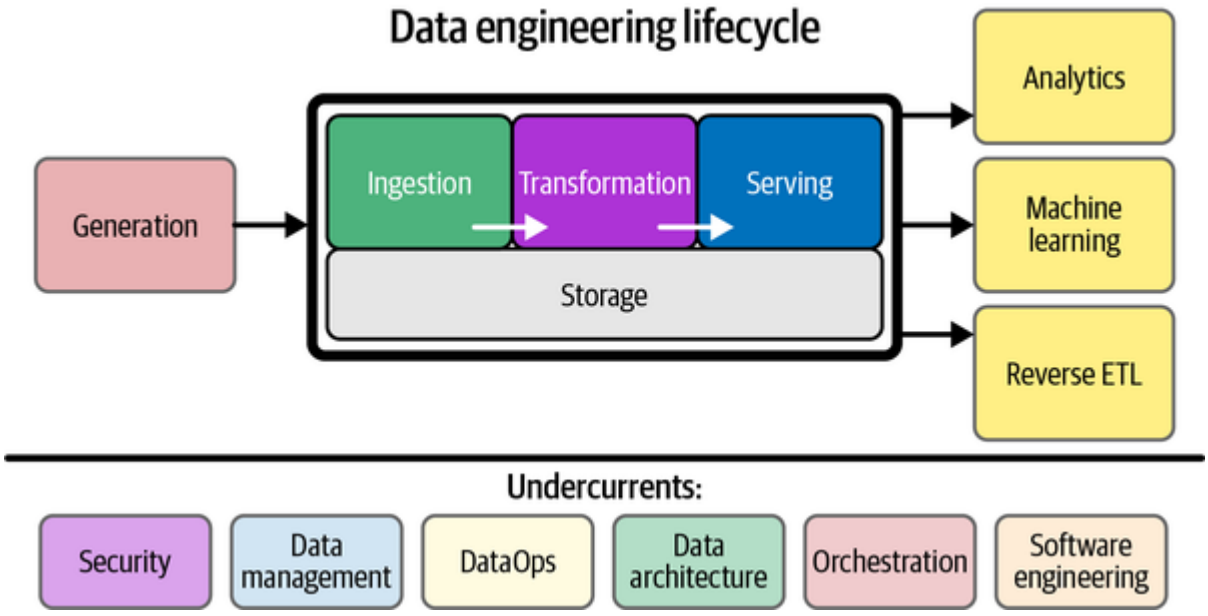

Figure 2-1. Components and undercurrents of the data engineering lifecycle

We begin the data engineering lifecycle by getting data from source systems and storing it. Next, we transform the data and then proceed to our central goal, serving data to analysts, data scientists, ML

engineers, and others. In reality, storage occurs throughout the lifecycle as data flows from beginning to end—hence, the diagram shows the storage "stage" as a foundation that underpins other stages.

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

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

The Data Lifecycle Versus the Data Engineering Lifecycle

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

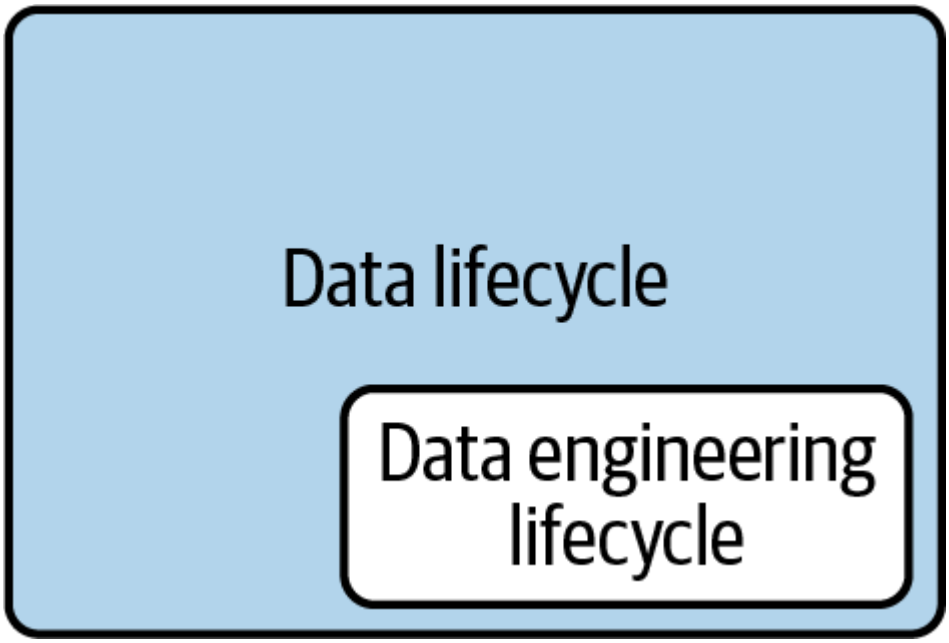

Figure 2-2. The data engineering lifecycle is a subset of the full data lifecycle

Generation: Source Systems

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

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

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

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

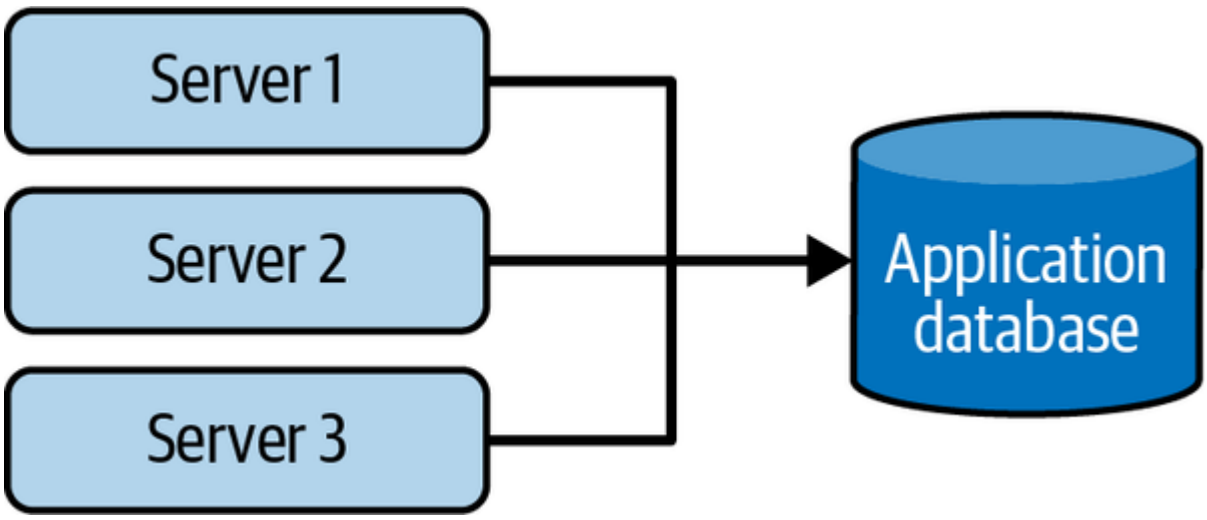

Figure 2-3. Source system example: an application database

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

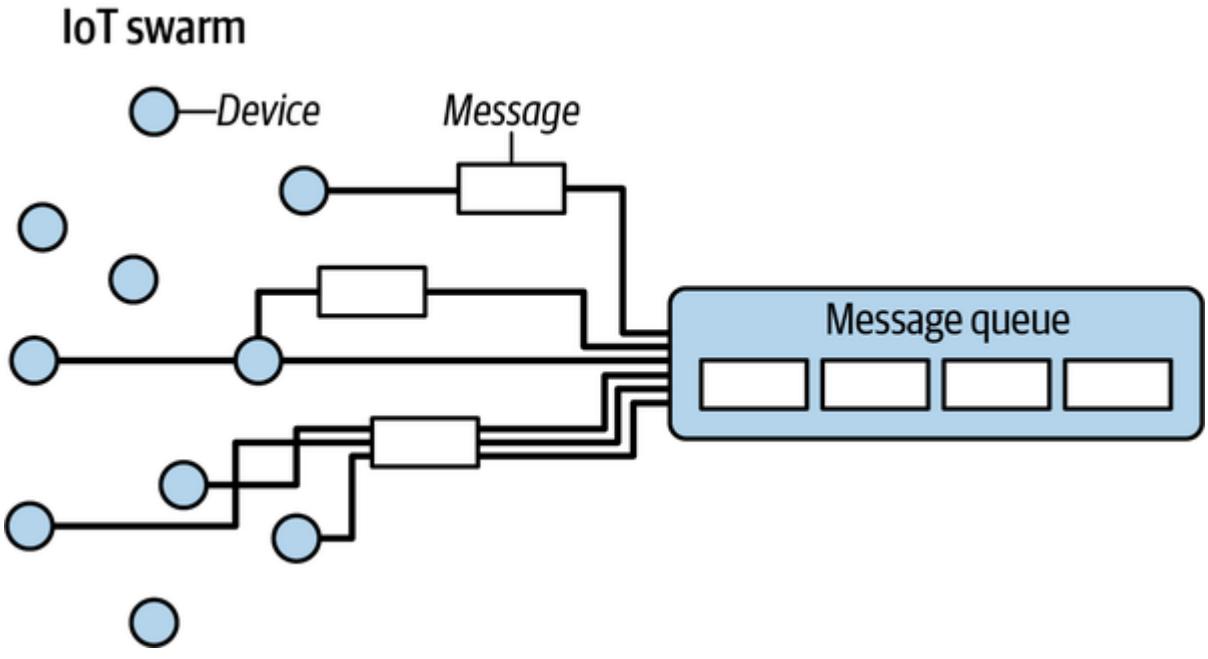

Figure 2-4. Source system example: an IoT swarm and message queue

Evaluating source systems: Key engineering considerations

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

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

Sources produce data consumed by downstream systems, including 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 message queue, a flat file, a blob, or a document database such as MongoDB. A more traditional model built on relational database storage uses a *fixed schema* enforced in the database, to which application writes must conform.

Either of these models presents challenges for data engineers. Schemas change over time; in fact, schema evolution is encouraged in the Agile approach to software development. A key part of the data

engineer's job is taking raw data input in the source system schema and transforming this into valuable output for analytics. This job becomes more challenging as the source schema evolves.

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

Storage

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

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

Evaluating storage systems: Key engineering considerations

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

- Is this storage solution compatible with the architecture's required write and read speeds?
- Will storage create a bottleneck for downstream processes?
- Do you understand how this storage technology works? Are you utilizing the storage system optimally or committing unnatural acts? For instance, are you applying a high rate of random access updates in an object storage system? (This is an antipattern with significant performance overhead.)
- Will this storage system handle anticipated future scale? You should consider all capacity limits on the storage system: total available storage, read operation rate, write volume, etc.
- Will downstream users and processes be able to retrieve data in the required service-level agreement (SLA)?
- Are you capturing metadata about schema evolution, data flows, data lineage, and so forth? Metadata has a significant impact on the utility of data. Metadata represents an investment in the future, dramatically enhancing discoverability and institutional knowledge to streamline future projects and architecture changes.
- Is this a pure storage solution (object storage), or does it support complex query patterns (i.e., a cloud data warehouse)?
- Is the storage system schema-agnostic (object storage)? Flexible schema (Cassandra)? Enforced schema (a cloud data warehouse)?
- How are you tracking master data, golden records data quality, and data lineage for data governance? (We have more to say on these in "Data Management".)

- How are you handling regulatory compliance and data sovereignty? For example, can you store your data in certain geographical locations but not others?

Understanding data access frequency

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

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

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

Selecting a storage system

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

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

Ingestion

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

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

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

Key engineering considerations for the ingestion phase

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

- What are the use cases for the data I'm ingesting? Can I reuse this data rather than create multiple versions of the same dataset?

- Are the systems generating and ingesting this data reliably, and is the data available when I need it?

- What is the data destination after ingestion?
- How frequently will I need to access the data?
- In what volume will the data typically arrive?
- What format is the data in? Can my downstream storage and transformation systems handle this format?
- Is the source data in good shape for immediate downstream use? If so, for how long, and what may cause it to be unusable?
- If the data is from a streaming source, does it need to be transformed before reaching its destination? Would an in-flight transformation be appropriate, where the data is transformed within the stream itself?

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

Batch versus streaming

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

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

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

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

Key considerations for batch versus stream ingestion

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

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

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

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

Push versus pull

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

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

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

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

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

Transformation

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

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

Key considerations for the transformation phase

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

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

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

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

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

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

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

Serving Data

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

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

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

Analytics

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

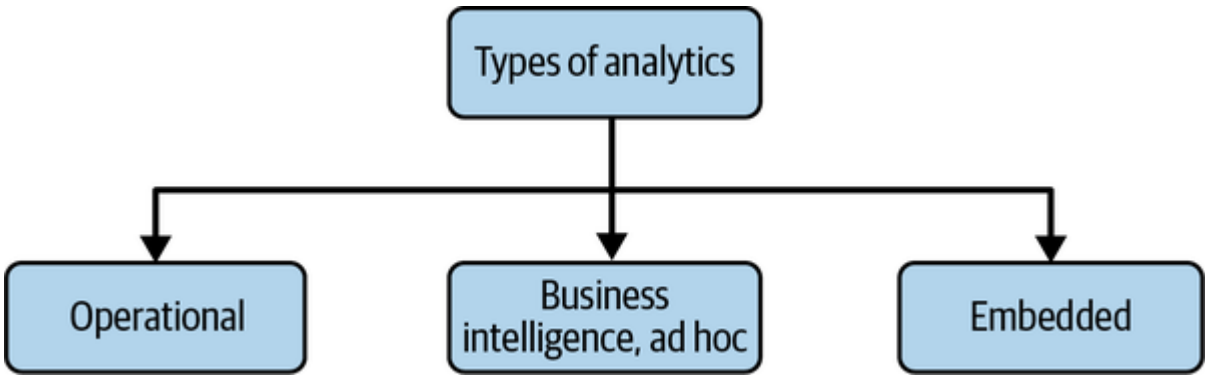

Figure 2-5. Types of analytics

Business intelligence

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

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

Operational analytics

Operational analytics focuses on the fine-grained details of operations, promoting actions that a user of the reports can act upon immediately. Operational analytics could be a live view of inventory or real-time dashboarding of website or application health. In this case, data is consumed in real time, either directly from a source system or from a streaming data pipeline. The types of insights in operational

analytics differ from traditional BI since operational analytics is focused on the present and doesn't necessarily concern historical trends.

Embedded analytics

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

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

Multitenancy

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

Machine learning

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

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

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

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

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

"Additional Resources".

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

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

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

Reverse ETL

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

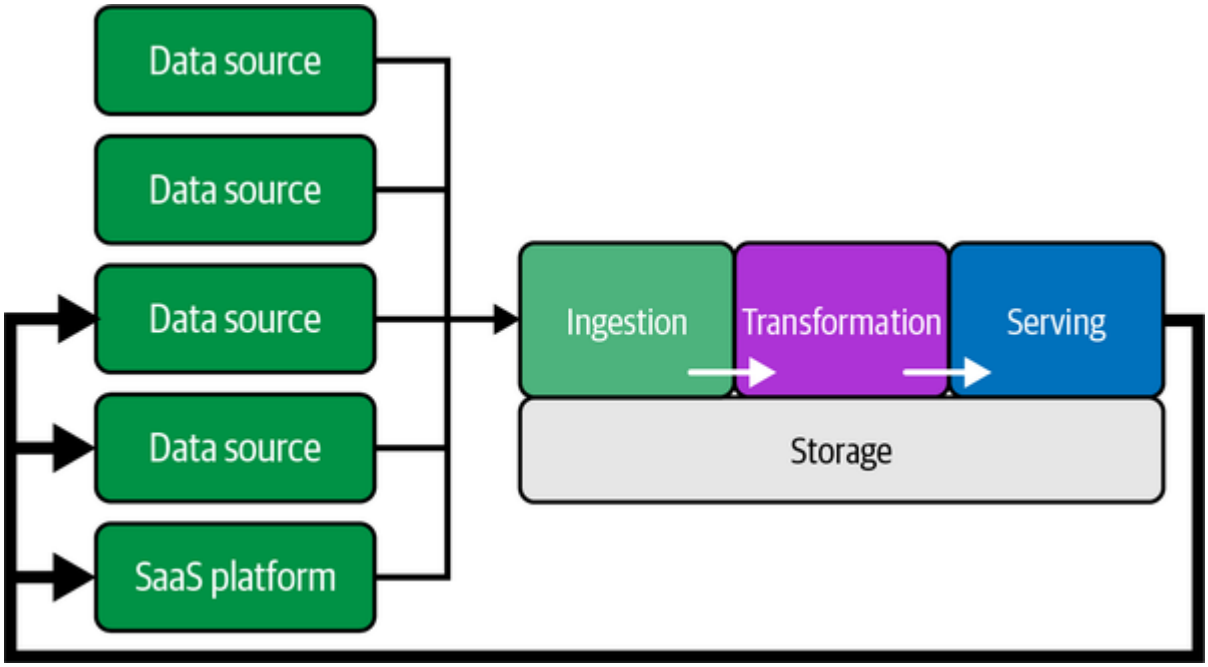

Figure 2-6. Reverse ETL

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

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

Reverse ETL has become especially important as businesses rely increasingly on SaaS and external platforms. For example, companies may want to push specific metrics from their data warehouse to a

customer data platform or CRM system. Advertising platforms are another everyday use case, as in the Google Ads example. Expect to see more activity in reverse ETL, with an overlap in both data engineering and ML engineering.

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

Major Undercurrents Across the Data Engineering Lifecycle

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

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

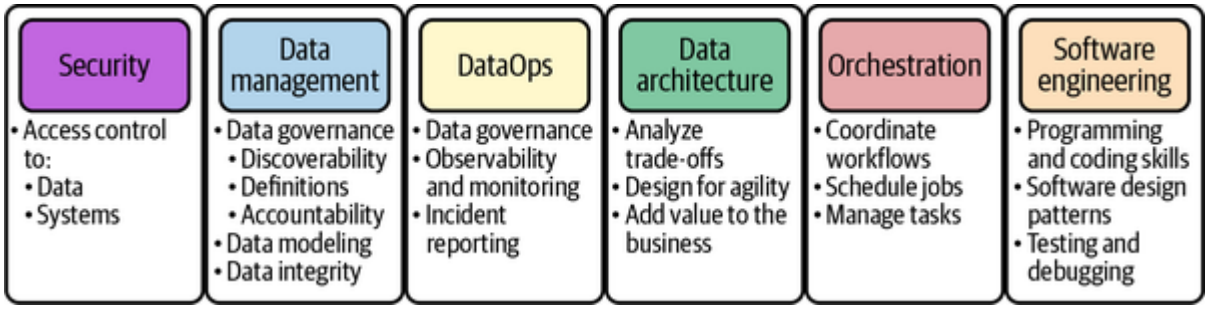

Figure 2-7. The major undercurrents of data engineering

Security

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

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

People and organizational structure are always the biggest security vulnerabilities in any company. When we hear about major security breaches in the media, it often turns out that someone in the company ignored basic precautions, fell victim to a phishing attack, or otherwise acted irresponsibly. The first line of defense for data security is to create a culture of security that permeates the

organization. All individuals who have access to data must understand their responsibility in protecting the company's sensitive data and its customers.

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

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

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

Data Management

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

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 might collaborate on MDM initiatives.

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

Data modeling and design

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

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

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

Data lineage

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

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

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

Data integration and interoperability

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

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

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

Data lifecycle management

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

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

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

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

Ethics and privacy

The last several years of data breaches, misinformation, and mishandling of data make one thing clear: data impacts people. Data used to live in the Wild West, freely collected and traded like baseball cards. Those days are long gone. Whereas data's ethical and privacy implications were once considered nice to have, like security, they're now central to the general data lifecycle. Data engineers need to do the right thing when no one else is watching, because everyone will be watching someday.[6] 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.

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:[7]

> 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 rather goal-oriented change. At each stage of our automation journey, opportunities exist for operational improvement. Even at the high level of maturity that we've described here, further room for improvement remains. Engineers might embrace a next-generation orchestration framework that builds in better metadata capabilities. Or they might try to develop a framework that builds DAGs automatically based on data-lineage specifications. The main point is that engineers constantly seek to implement improvements in automation that will reduce their workload and increase the value that they deliver to the business.

Observability and monitoring

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

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

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

Petrella's DODD method mentioned previously in this chapter provides an excellent framework for thinking about data observability. DODD is much like test-driven development (TDD) in software engineering:[8]

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

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

Incident response

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

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

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

DataOps summary

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

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

Data Architecture

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

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

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

Orchestration

> We think that orchestration matters because we view it as really the center of gravity of both the data platform as well as the data lifecycle, the software development lifecycle as it comes to data.
>
> Nick Schrock, founder of Elementl[9]

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 in Chapter 8.

Software Engineering

Software engineering has always been a central skill for data engineers. In the early days of contemporary data engineering (2000–2010), data engineers worked on 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 Elastic MapReduce (EMR)—and cloud data warehouses. When data engineers have to manage their infrastructure in a cloud environment, they increasingly do this through IaC frameworks rather than manually spinning up instances and installing software. Several general-purpose and 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 vital throughout the data engineering lifecycle, especially as we adopt DataOps practices.

Pipelines as code

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

General-purpose problem solving

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

Conclusion

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

We break the data engineering lifecycle into the following stages:

- Generation
- Storage
- Ingestion
- Transformation
- Serving data

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

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

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

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

Additional Resources

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

- "The Seven Stages of Effective Incident Response" Atlassian web page
- "Staying Ahead of Debt" by Etai Mizrahi
- "What Is Metadata" by Michelle Knight

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

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

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

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