you've agreed to. For example, given the preceding example SLA, an SLO might be, "Our data pipelines to your dashboard or ML workflow will have 99% uptime, with 95% of data free of defects." Be sure expectations are clear and you have the ability to verify you're operating within your agreed SLA and SLO parameters.

It's not enough to simply agree on an SLA. Ongoing communication is a central feature of a good SLA. Have you communicated possible issues that might affect your SLA or SLO expectations? What's your process for remediation and improvement?

Trust is everything. It takes a long time to earn, and it's easy to lose.

What's the Use Case, and Who's the User?

The serving stage is about data in action. But what is a *productive* use of data? You need to consider two things to answer this question: what's the use case, and who's the user?

The use case for data goes well beyond viewing reports and dashboards. Data is at its best when it leads to *action*. Will an executive make a strategic decision from a report? Will a user of a mobile food delivery app receive a coupon that entices them to purchase in the next two minutes? The data is often used in more than one use case—e.g., to train an ML model that does lead scoring and populates a CRM (reverse ETL). High-quality, high-impact data will inherently attract many interesting use cases. But in seeking use cases, always ask, "What *action* will this data trigger, and *who* will be performing this action?," with the appropriate follow-up question, "Can this action be automated?"

Whenever possible, prioritize use cases with the highest possible ROI. Data engineers love to obsess over the technical implementation details of the systems they build while ignoring the basic question of purpose. Engineers want to do what they do best: engineer things. When engineers recognize the need to focus on value and use cases, they become much more valuable and effective in their roles.

When starting a new data project, working backward is helpful. While it's tempting to focus on tools, we encourage you to start with the use case and the users. Here are some questions to ask yourself as you get started:

- Who will use the data, and how will they use it?
- What do stakeholders expect?
- How can I collaborate with data stakeholders (e.g., data scientists, analysts, business users) to understand how the data I'm working with will be used?

Again, always approach data engineering from the perspective of the user and their use case. By understanding their expectations and goals, you can work backward to create amazing data products more easily. Let's take a moment to expand on our discussion of a data product.

Data Products

> *A good definition of a data product is a product that facilitates an end goal through the use of data.*
>
> —DJ Patil

Data products aren't created in a vacuum. Like so many other organizational processes that we've discussed, making data products is a full-contact sport, involving a mix of product and business alongside technology. It's important to involve key stakeholders in developing a data product. In most companies, a data engineer is a couple of steps removed from the end users of a data product; a good data engineer will seek to fully understand outcomes for direct users such as data analysts and data scientists or customers external to the company.

When creating a data product, it's useful to think of the "jobs to be done."[1] A user "hires" a product for a "job to be done." This means you need to know what the user wants—i.e., their motivation for "hiring" your product. A classic engineering mistake is simply building without understanding the

requirements, needs of the end user, or product/market fit. This disaster happens when you build data products nobody wants to use.

A good data product has positive feedback loops. More usage of a data product generates more useful data, which is used to improve the data product. Rinse and repeat.

When building a data product, keep these considerations in mind:

- When someone uses the data product, what do they hope to accomplish? All too often, data products are made without a clear understanding of the outcome expected by the user.
- Will the data product serve internal or external users? In Chapter 2, we discussed internal- and external-facing data engineering. When creating a data product, knowing whether your customer is internal or external facing will impact the way data is served.
- What are the outcomes and ROI of the data product you're building?

Building data products that people will use and love is critical. Nothing will ruin the adoption of a data product more than unwanted utility and loss of trust in the data outputs. Pay attention to the adoption and usage of data products, and be willing to adjust to make users happy.

Self-Service or Not?

How will users interface with your data product? Will a business director request a report from the data team, or can this director simply build the report? Self-service data products—giving the user the ability to build data products on their own—have been a common aspiration of data users for many years. What's better than just giving the end user the ability to directly create reports, analyses, and ML models?

Today, self-service BI and data science is still mostly aspirational. While we occasionally see companies successfully doing self-service with data, this is rare. Most of the time, attempts at self-service data begin with great intentions but ultimately fail; self-service data is tough to implement in

practice. Thus, the analyst or data scientist is left to perform the heavy lifting of providing ad hoc reports and maintaining dashboards.

Why is self-service data so hard? The answer is nuanced, but it generally involves understanding the end user. If the user is an executive who needs to understand how the business is doing, that person probably just wants a predefined dashboard of clear and actionable metrics. The executive will likely ignore any self-serve tools for creating custom data views. If reports provoke further questions, they might have analysts at their disposal to pursue a deeper investigation. On the other hand, a user who is an analyst might already be pursuing self-service analytics via more powerful tools such as SQL. Self-service analytics through a BI layer is not useful. The same considerations apply to data science. Although granting self-service ML to "citizen data scientists" has been a goal of many automated ML vendors, adoption is still nascent for the same reasons as self-service analytics. In these two extreme cases, a self-service data product is a wrong tool for the job.

Successful self-service data projects boil down to having the right audience. Identify the self-service users and the "job" they want to do. What are they trying to accomplish by using a self-service data product versus partnering with a data analyst to get the job done? A group of executives with a background in data forms an ideal audience for self-service; they likely want to slice and dice data themselves without needing to dust off their languishing SQL skills. Business leaders willing to invest the time to learn data skills through a company initiative and training program could also realize significant value from self-service.

Determine how you will provide data to this group. What are their time requirements for new data? What happens if they inevitably want more data or change the scope of what's required from self-service? More data often means more questions, which requires more data. You'll need to anticipate the growing needs of your self-service users. You also need to understand the fine balance between flexibility and guardrails that will help your audience find value and insights without incorrect results and confusion.

Data Definitions and Logic

As we've emphatically discussed, the utility of data in an organization is ultimately derived from its correctness and trustworthiness. Critically, the correctness of data goes beyond faithful reproduction of event values from source systems. Data correctness also encompasses proper data definitions and logic; these must be baked into data through all lifecycle stages, from source systems to data pipelines to BI tools and much more.

Data definition refers to the meaning of data as it is understood throughout the organization. For example, *customer* has a precise meaning within a company and across departments. When the definition of a customer varies, these must be documented and made available to everyone who uses the data.

Data logic stipulates formulas for deriving metrics from data—say, gross sales or customer lifetime value. Proper data logic must encode data definitions and details of statistical calculations. To compute customer churn metrics, we would need a definition: who is a customer? To calculate net profits, we would need a set of logical rules to determine which expenses to deduct from gross revenue.

Frequently, we see data definitions and logic taken for granted, often passed around the organization in the form of tribal knowledge. *Tribal knowledge* takes on a life of its own, often at the expense of anecdotes replacing data-driven insights, decisions, and actions. Instead, formally declaring data definitions and logic both in a data catalog and within the systems of the data engineering lifecycle goes a long way to ensuring data correctness, consistency, and trustworthiness.

Data definitions can be served in many ways, sometimes explicitly, but mostly implicitly. By *implicit*, we mean that anytime you serve data for a query, a dashboard, or an ML model, the data and derived metrics are presented consistently and correctly. When you write a SQL query, you're implicitly assuming that the inputs to this query are correct, including upstream pipeline logic and definitions. This is where data modeling

(described in Chapter 8) is incredibly useful to capture data definitions and logic in a way that's understandable and usable by multiple end users.

Using a semantic layer, you consolidate business definitions and logic in a reusable fashion. Write once, use anywhere. This paradigm is an object-oriented approach to metrics, calculations, and logic. We'll have more to say in "Semantic and Metrics Layers".

Data Mesh

Data mesh will increasingly be a consideration when serving data. Data mesh fundamentally changes the way data is served within an organization. Instead of siloed data teams serving their internal constituents, every domain team takes on two aspects of decentralized, peer-to-peer data serving.

First, teams are responsible for serving data *to other teams* by preparing it for consumption. Data must be good for use in data apps, dashboards, analytics, and BI tools across the organization. Second, each team potentially runs its dashboards and analytics for *self-service*. Teams consume data from across the organization based on the particular needs in their domain. Data consumed from other teams may also make its way into the software designed by a domain team through user-facing analytics or an ML feature.

This dramatically changes the details and structure of serving. We introduced the concept of a data mesh in Chapter 3. Now that we've covered some general considerations for serving data, let's look at the first major area: analytics.

Analytics

The first data serving use case you'll likely encounter is *analytics*, which is discovering, exploring, identifying, and making visible key insights and patterns within data. Analytics has many aspects. As a practice, analytics is carried out using statistical methods, reporting, BI tools, and more. As a

data engineer, knowing the various types and techniques of analytics is key to accomplishing your work. This section aims to show how you'll serve data for analytics and presents some points to think about to help your analysts succeed.

Before you even serve data for analytics, the first thing you need to do (which should sound familiar after reading the preceding section) is identify the end use case. Is the user looking at historical trends? Should users be immediately and automatically notified of an anomaly, such as a fraud alert? Is someone consuming a real-time dashboard on a mobile application? These examples highlight the differences between business analytics (usually BI), operational analytics, and user-facing analytics. Each of these analytics categories has different goals and unique serving requirements. Let's look at how you'll serve data for these types of analytics.

Business Analytics

Business analytics uses historical and current data to make strategic and actionable decisions. The types of decisions tend to factor in longer-term trends, and often involve a mix of statistical and trend analysis, alongside domain expertise and human judgment. Business analysis is as much an art as it is a science.

Business analytics typically falls into a few big areas—dashboards, reports, and ad hoc analysis. A business analyst might focus on one or all of these categories. Let's quickly look at the differences between these practices and related tools. Understanding an analyst's workflow will help you, the data engineer, understand how to serve data.

A *dashboard* concisely shows decision makers how an organization is performing against a handful of core metrics, such as sales and customer retention. These core metrics are presented as visualizations (e.g., charts or heatmaps), summary statistics, or even a single number. This is similar to a car dashboard, which gives you a single readout of the critical things you need to know while driving a vehicle. An organization may have more than

one dashboard, with C-level executives using an overarching dashboard, and their direct reports using dashboards with their particular metrics, KPIs, or objectives and key results (OKRs). Analysts help create and maintain these dashboards. Once business stakeholders embrace and rely on a dashboard, the analyst usually responds to requests to look into a potential issue with a metric or add a new metric to the dashboard. Currently, you might use BI platforms to create dashboards, such as Tableau, Looker, Sisense, Power BI, or Apache Superset/Preset.

Analysts are often tasked by business stakeholders with creating a *report*. The goal of a report is to use data to drive insights and action. An analyst working at an online retail company is asked to investigate which factors are driving a higher-than-expected rate of returns for women's running shorts. The analyst runs some SQL queries in the data warehouse, aggregates the return codes that customers provide as the reason for their return, and discovers that the fabric in the running shorts is of inferior quality, often wearing out within a few uses. Stakeholders such as manufacturing and quality control are notified of these findings. Furthermore, the findings are summarized in a report and distributed in the same BI tool where the dashboard resides.

The analyst was asked to dig into a potential issue and come back with insights. This represents an example of *ad hoc analysis*. Reports typically start as ad hoc requests. If the results of the ad hoc analysis are impactful, they often end up in a report or dashboard. The technologies used for reports and ad hoc analysis are similar to dashboards but may include Excel, Python, R-based notebooks, SQL queries, and much more.

Good analysts constantly engage with the business and dive into the data to answer questions and uncover hidden and counterintuitive trends and insights. They also work with data engineers to provide feedback on data quality, reliability issues, and requests for new datasets. The data engineer is responsible for addressing this feedback and providing new datasets for the analyst to use.

Returning to the running shorts example, suppose that after communicating their findings, analysts learn that manufacturing can provide them with various supply-chain details regarding the materials used in the running shorts. Data engineers undertake a project to ingest this data into the data warehouse. Once the supply-chain data is present, analysts can correlate specific garment serial numbers with the supplier of the fabric used in the item. They discover that most failures are tied to one of their three suppliers, and the factory stops using fabric from this supplier.

The data for business analytics is frequently served in batch mode from a data warehouse or a data lake. This varies wildly across companies, departments, and even data teams within companies. New data might be available every second, minute, every 30 minutes, every day, or once a week. The frequency of the batches can vary for several reasons. One key thing to note is that engineers working on analytics problems should consider various potential applications of data, current, and future. It is common to have mixed data update frequencies to serve use cases appropriately but remember that the frequency of ingestion sets a ceiling on downstream frequency. If streaming applications exist for the data, it should be ingested as a stream even if some downstream processing and serving steps are handled in batches.

Of course, data engineers must address various backend technical considerations in serving business analytics. Some BI tools store data in an internal storage layer. Other tools run queries on your data lake or data warehouse. This is advantageous because you can take full advantage of your OLAP database's power. As we've discussed in earlier chapters, the downside is cost, access control, and latency.

Operational Analytics

If business analytics is about using data to discover actionable insights, then operational analytics uses data to take *immediate action*:

- Operational analytics versus business analytics = immediate action versus actionable insights

The big difference between operational and business analytics is *time*. Data used in business analytics takes a longer view of the question under consideration. Up-to-the-second updates are nice to know but won't materially impact the quality or outcome. Operational analytics is quite the opposite, as real-time updates can be impactful in addressing a problem when it occurs.

An example of operational analytics is real-time application monitoring. Many software engineering teams want to know how their application is performing; if issues arise, they want to be notified immediately. The engineering team might have a dashboard (see, e.g., Figure 9-2) that shows the key metrics such as requests per second, database I/O, or whatever metrics are important. Certain conditions can trigger scaling events, adding more capacity if servers are overloaded. If certain thresholds are breached, the monitoring system might also send alerts via text message, group chat, and email.

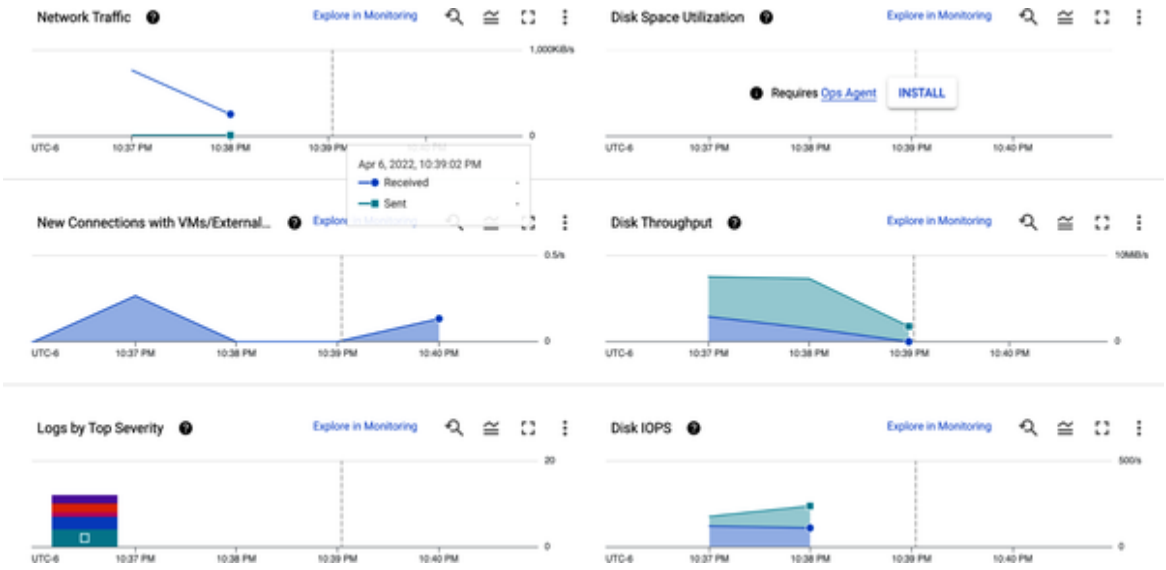

Figure 9-2. An operational analytics dashboard showing some key metrics from Google Compute Engine

BUSINESS AND OPERATIONAL ANALYTICS

The line between business and operational analytics has begun to blur. As streaming and low-latency data become more pervasive, it is only natural to apply operational approaches to business analytics problems; in addition to monitoring website performance on Black Friday, an online retailer could also analyze and present sales, revenue and the impact of advertising campaigns in real time.

The data architectures will change to fit into a world where you can have both your red hot and warm data in one place. The central question you should always ask yourself, and your stakeholders, is this: if you have streaming data, what are you going to do with it? What action should you take? Correct action creates impact and value. Real-time data without action is an unrelenting distraction.

In the long term, we predict that streaming will supplant batch. Data products over the next 10 years will likely be streaming-first, with the ability to seamlessly blend historical data. After real-time collection, data can still be consumed and processed in batches as required.

Let's return once again to our running shorts example. Using analytics to discover bad fabric in the supply chain was a huge success; business leaders and data engineers want to find more opportunities to utilize data to improve product quality. The data engineers suggest deploying real-time analytics at the factory. The plant already uses a variety of machines capable of streaming real-time data. In addition, the plant has cameras recording video on the manufacturing line. Right now, technicians watch the footage in real time, look for defective items, and alert those running the line when they see a high rate of snags appearing in items.

Data engineers realize that they can use an off-the-shelf cloud machine vision tool to identify defects in real time automatically. Defect data is tied to specific item serial numbers and streamed. From here, a real-time analytics process can tie defective items to streaming events from machines further up the assembly line.

Using this approach, factory floor analysts discover that the quality of raw fabric stock varies significantly from box to box. When the monitoring system shows a high rate of snag defects, line workers can remove the defective box and charge it back to the supplier.

Seeing the success of this quality improvement project, the supplier decides to adopt similar quality-control processes. Data engineers from the retailer work with the supplier to deploy their real-time data analytics, dramatically improving the quality of their fabric stock.

Embedded Analytics

Whereas business and operational analytics are internally focused, a recent trend is external- or user-facing analytics. With so much data powering applications, companies increasingly provide analytics to end users. These are typically referred to as *data applications*, often with analytics dashboards embedded within the application itself. Also known as *embedded analytics*, these end-user-facing dashboards give users key metrics about their relationship with the application.

A smart thermostat has a mobile application that shows the temperature in real time and up-to-date power consumption metrics, allowing the user to create a better energy-efficient heating or cooling schedule. In another example, a third-party ecommerce platform provides its sellers a real-time dashboard on sales, inventory, and returns. The seller has the option to use this information to offer deals to customers in near real time. In both cases, an application allows users to make real-time decisions (manually or automatically) based on data.

The landscape of embedded analytics is snowballing, and we expect that such data applications will become increasingly pervasive within the next few years. As a data engineer, you're probably not creating the embedded analytics frontend, as the application developers handle that. Since you're responsible for the databases serving the embedded analytics, you'll need to understand the speed and latency requirements for user-facing analytics.

Performance for embedded analytics encompasses three problems. First, app users are not as tolerant of infrequent batch processing as internal company analysts; users of a recruiting SaaS platform may expect to see a change in their statistics as soon as they upload a new resume. Users want low *data latency*. Second, users of data apps expect fast *query performance*. When they adjust parameters in an analytics dashboard, they want to see refreshed results appear in seconds. Third, data apps must often support extremely high query rates across many dashboards and numerous customers. High *concurrency* is critical.

Google and other early major players in the data apps space developed exotic technologies to cope with these challenges. For new startups, the default is to use conventional transactional databases for data applications. As their customer bases expand, they outgrow their initial architecture. They have access to a new generation of databases that combine high performance—fast queries, high concurrency, and near real-time updates—with relative ease of use (e.g., SQL-based analytics).

Machine Learning

The second major area for serving data is machine learning. ML is increasingly common, so we'll assume you're at least familiar with the concept. With the rise of ML engineering (itself almost a parallel universe to data engineering), you might ask yourself where a data engineer fits into the picture.

Admittedly, the boundary between ML, data science, data engineering, and ML engineering is increasingly fuzzy, and this boundary varies dramatically between organizations. In some organizations, ML engineers take over data processing for ML applications right after data collection or may even form an entirely separate and parallel data organization that handles the entire lifecycle for all ML applications. Data engineers handle all data processing in other settings and then hand off data to ML engineers for model training. Data engineers may even handle some extremely ML-specific tasks, such featurization of data.

Let's return to our example of quality for control of running shorts produced by an online retailer. Suppose that streaming data has been implemented in the factory that makes the raw fabric stock for the shorts. Data scientists discovered that the quality of the manufactured fabric is susceptible to characteristics of the input raw polyester, temperature, humidity, and various tunable parameters of the loom that weaves the fabric. Data scientists develop a basic model to optimize loom parameters. ML engineers automate model training and set up a process to automatically tune the loom based on input parameters. Data and ML engineers work together to design a featurization pipeline, and data engineers implement and maintain the pipeline.

What a Data Engineer Should Know About ML

Before we discuss serving data for ML, you may ask yourself how much ML you need to know as a data engineer. ML is an incredibly vast topic, and we won't attempt to teach you the field; countless books and courses are available to learn ML.

While a data engineer doesn't need to have a deep understanding of ML, it helps tremendously to know the basics of how classical ML works and the fundamentals of deep learning. Knowing the basics of ML will go a long way in helping you work alongside data scientists in building data products.

Here are some areas of ML that we think a data engineer should be familiar with:

- The difference between supervised, unsupervised, and semisupervised learning.
- The difference between classification and regression techniques.
- The various techniques for handling time-series data. This includes time-series analysis, as well as time-series forecasting.
- When to use the "classical" techniques (logistic regression, tree-based learning, support vector machines) versus deep learning. We constantly

see data scientists immediately jump to deep learning when it's overkill. As a data engineer, your basic knowledge of ML can help you spot whether an ML technique is appropriate and scales the data you'll need to provide.

- When would you use automated machine learning (AutoML) versus handcrafting an ML model? What are the trade-offs with each approach regarding the data being used?
- What are data-wrangling techniques used for structured and unstructured data?
- All data that is used for ML is converted to numbers. If you're serving structured or semistructured data, ensure that the data can be properly converted during the feature-engineering process.
- How to encode categorical data and the embeddings for various types of data.
- The difference between batch and online learning. Which approach is appropriate for your use case?
- How does the data engineering lifecycle intersect with the ML lifecycle at your company? Will you be responsible for interfacing with or supporting ML technologies such as feature stores or ML observability?
- Know when it's appropriate to train locally, on a cluster, or at the edge. When would you use a GPU over a CPU? The type of hardware you use largely depends on the type of ML problem you're solving, the technique you're using, and the size of your dataset.
- Know the difference between the applications of batch and streaming data in training ML models. For example, batch data often fits well with offline model training, while streaming data works with online training.
- What are data cascades, and how might they impact ML models?

- Are results returned in real time or in batch? For example, a batch speech transcription model might process speech samples and return text in batch after an API call. A product recommendation model might need to operate in real time as the customer interacts with an online retail site.
- The use of structured versus unstructured data. We might cluster tabular (structured) customer data or recognize images (unstructured) by using a neural net.

ML is a *vast* subject area, and this book won't teach you these topics, or even ML generalities. If you'd like to learn more about ML, we suggest reading *Hands on Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron (O'Reilly); countless other ML courses and books are available online. Because the books and online courses evolve so rapidly, do your research on what seems like a good fit for you.

Ways to Serve Data for Analytics and ML

As with analytics, data engineers provide data scientists and ML engineers with the data they need to do their jobs. We have placed serving for ML alongside analytics because the pipelines and processes are extremely similar. There are many ways to serve data for analytics and ML. Some common ways to serve this data include files, databases, query engines, and data sharing. Let's briefly look at each.

File Exchange

File exchange is ubiquitous in data serving. We process data and generate files to pass to data consumers.

Keep in mind that a file might be used for many purposes. A data scientist might load a text file (unstructured data) of customer messages to analyze the sentiments of customer complaints. A business unit might receive invoice data from a partner company as a collection of CSVs (structured data), and an analyst must perform some statistical analysis on these files.

Or, a data vendor might provide an online retailer with images of products on a competitor's website (unstructured data) for automated classification using computer vision.

The way you serve files depends on several factors, such as these:

- Use case—business analytics, operational analytics, user-facing analytics
- The data consumer's data-handling processes
- The size and number of individual files in storage
- Who is accessing this file
- Data type—structured, semistructured, or unstructured

The second bullet point is one of the main considerations. It is often necessary to serve data through files rather than data sharing because the data consumer cannot use a sharing platform.

The simplest file to serve is something along the lines of emailing a single Excel file. This is still a common workflow even in an era when files can be collaboratively shared. The problem with emailing files is each recipient gets their version of the file. If a recipient edits the file, these edits are specific to that user's file. Deviations among files inevitably result. If you need a coherent, consistent version of a file, we suggest using a collaboration platform such as Microsoft 365 or Google Docs.

Of course, serving single files is hard to scale, and your needs will eventually outgrow simple cloud file storage. You'll likely grow into an object storage bucket if you have a handful of large files, or a data lake if you have a steady supply of files. Object storage can store any type of blob file, and is especially useful for semistructured or unstructured files.

We'll note that we generally consider file exchange through object storage (data lake) to land under "data sharing" rather than file exchange since the process can be significantly more scalable and streamlined than ad hoc file exchange.

Databases

Databases are a critical layer in serving data for analytics and ML. For this discussion, we'll implicitly keep our focus on serving data from OLAP databases (e.g., data warehouses and data lakes). In the previous chapter, you learned about querying databases. Serving data involves querying a database, and then consuming those results for a use case. An analyst or data scientist might query a database by using a SQL editor and export those results to a CSV file for consumption by a downstream application, or analyze the results in a notebook (described in "Serving Data in Notebooks").

Serving data from a database carries a variety of benefits. A database imposes order and structure on the data through schema; databases can offer fine-grained permission controls at the table, column, and row level, allowing database administrators to craft complex access policies for various roles; and, databases can offer high serving performance for large, computationally intensive queries, high query concurrency, or both.

BI systems usually share the data processing workload with a source database, but the boundary between processing in the two systems varies. For example, a Tableau server runs an initial query to pull data from a database and stores it locally. Basic OLAP/BI slicing and dicing (interactive filtering and aggregation) runs directly on the server from the local data copy. On the other hand, Looker relies on a computational model called *query pushdown*; Looker encodes data processing logic in a specialized language (LookML), combines this with dynamic user input to generate SQL queries, runs these against the source database, and presents the output. (See "Semantic and Metrics Layers".) Both Tableau and Looker have various configuration options for caching results to reduce the processing burden for frequently run queries.

A data scientist might connect to a database, extract data, and perform feature engineering and selection. This converted dataset is then fed into an ML model; the offline model is trained and produces predictive results.

Data engineers are quite often tasked with managing the database serving layer. This includes management of performance and costs. In databases that separate compute and storage, this is a somewhat more subtle optimization problem than in the days of fixed on-premises infrastructure. For example, it is now possible to spin up a new Spark cluster or Snowflake warehouse for each analytical or ML workload. It is generally recommended to at least split out clusters by major use cases, such as ETL and serving for analytics and data science. Often data teams choose to slice more finely, assigning one warehouse per major area. This makes it possible for different teams to budget for their query costs under the supervision of a data engineering team.

Also, recall the three performance considerations that we discussed in "Embedded Analytics". These are data latency, query performance, and concurrency. A system that can ingest directly from a stream can lower data latency. And many database architectures rely on SSD or memory caching to enhance query performance and concurrency to serve the challenging use cases inherent in user-facing analytics.

Increasingly, data platforms like Snowflake and Databricks allow analysts and data scientists to operate under a single environment, providing SQL editors and data science notebooks under one roof. Because compute and storage are separated, the analysts and data scientists can consume the underlying data in various ways without interfering with each other. This will allow high throughput and faster delivery of data products to stakeholders.

Streaming Systems

Streaming analytics are increasingly important in the realm of serving. At a high level, understand that this type of serving may involve *emitted metrics*, which are different from traditional queries.

Also, we see operational analytics databases playing a growing role in this area (see "Operational Analytics"). These databases allow queries to run across a large range of historical data, encompassing up-to-the-second

current data. Essentially, they combine aspects of OLAP databases with stream-processing systems. Increasingly, you'll work with streaming systems to serve data for analytics and ML, so get familiar with this paradigm.

You've learned about streaming systems throughout the book. For an idea of where it's going, read about the live data stack in Chapter 11.

Query Federation

As you learned in Chapter 8, query federation pulls data from multiple sources, such as data lakes, RDBMSs, and data warehouses. Federation is becoming more popular as distributed query virtualization engines gain recognition as ways to serve queries without going through the trouble of centralizing data in an OLAP system. Today, you can find OSS options like Trino and Presto and managed services such as Starburst. Some of these offerings describe themselves as ways to enable the data mesh; time will tell how that unfolds.

When serving data for federated queries, you should be aware that the end user might be querying several systems—OLTP, OLAP, APIs, filesystems, etc. (Figure 9-3). Instead of serving data from a single system, you're now serving data from multiple systems, each with its usage patterns, quirks, and nuances. This poses challenges for serving data. If federated queries touch live production source systems, you must ensure that the federated query won't consume excessive resources in the source.

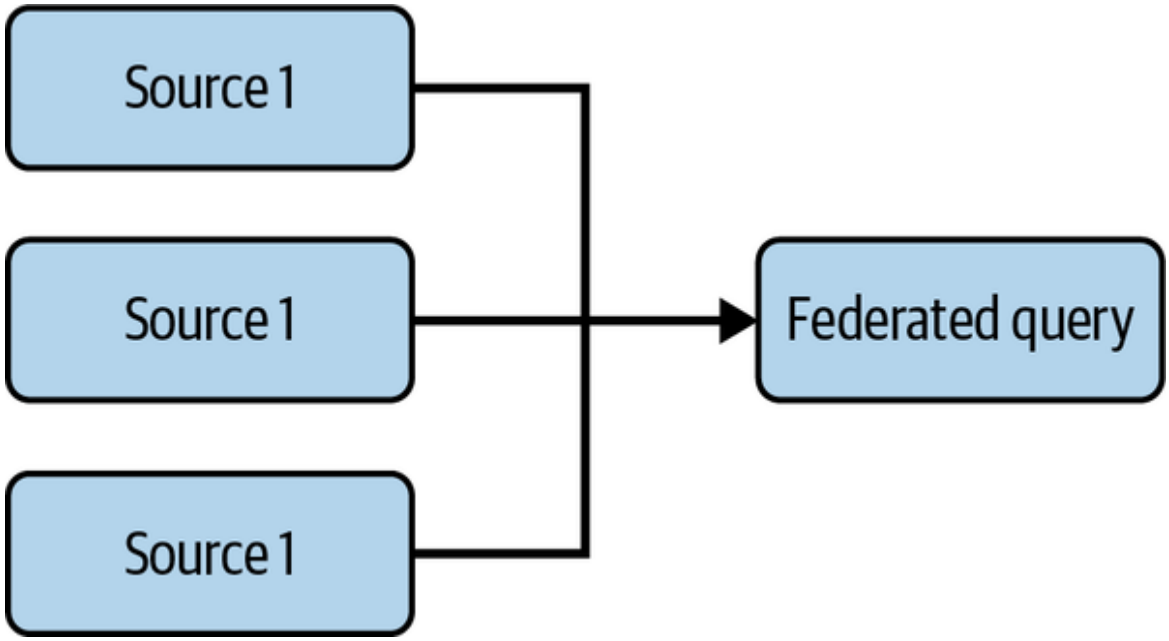

Figure 9-3. A federated query with three data sources

In our experience, federated queries are ideally suited when you want flexibility in analyzing data or the source data needs to be tightly controlled. Federation allows ad hoc queries for performing exploratory analysis, blending data from various systems without the complexity of setting up data pipelines or ETL. This will allow you to determine whether the performance of a federated query is sufficient for ongoing purposes or you need to set up ingestion on some or all data sources and centralize the data in an OLAP database or data lake.

Federated queries also provide read-only access to source systems, which is great when you don't want to serve files, database access, or data dumps. The end user reads only the version of the data they're supposed to access and nothing more. Query federation is a great option to explore for situations where access and compliance are critical.

Data Sharing

Chapter 5 includes an extensive discussion of data sharing. Any data exchange between organizations or units within a larger organization can be viewed as data sharing. Still, we mean specifically sharing through

massively multitenant storage systems in a cloud environment. Data sharing generally turns data serving into a security and access control problem.

The actual queries are now handled by the data consumers (analysts and data scientists) rather than the engineers sourcing the data. Whether serving data in a data mesh within an organization, providing data to the public, or serving to partner businesses, data sharing is a compelling serving model. Data sharing is increasingly a core feature of major data platforms like Snowflake, Redshift, and BigQuery allowing companies to share data safely and securely with each other.

Semantic and Metrics Layers

When data engineers think about serving, they naturally tend to gravitate toward the data processing and storage technologies—i.e., will you use Spark or a cloud data warehouse? Is your data stored in object storage or cached in a fleet of SSDs? But powerful processing engines that deliver quick query results across vast datasets don't inherently make for quality business analytics. When fed poor-quality data or poor-quality queries, powerful query engines quickly return bad results.

Where data quality focuses on characteristics of the data itself and various techniques to filter or improve bad data, query quality is a question of building a query with appropriate logic that returns accurate answers to business questions. Writing high-quality ETL queries and reporting is time-intensive, detailed work. Various tools can help automate this process while facilitating consistency, maintenance, and continuous improvement.

Fundamentally, a *metrics layer* is a tool for maintaining and computing business logic.[2] (A *semantic layer* is extremely similar conceptually,[3] and *headless BI* is another closely related term.) This layer can live in a BI tool or in software that builds transformation queries. Two concrete examples are Looker and Data Build Tool (dbt).

For instance, Looker's LookML allows users to define virtual, complex business logic. Reports and dashboards point to specific LookML for computing metrics. Looker allows users to define standard metrics and

reference them in many downstream queries; this is meant to solve the traditional problem of repetition and inconsistency in traditional ETL scripts. Looker uses LookML to generate SQL queries, which are pushed down to the database. Results can be persisted in the Looker server or in the database itself for large result sets.

dbt allows users to define complex SQL data flows encompassing many queries and standard definitions of business metrics, much like Looker. Unlike Looker, dbt runs exclusively in the transform layer, although this can include pushing queries into views that are computed at query time. Whereas Looker focuses on serving queries and reporting, dbt can serve as a robust data pipeline orchestration tool for analytics engineers.

We believe that metrics layer tools will grow more popular with wider adoption and more entrants, as well as move upstream toward the application. Metrics layer tools help solve a central question in analytics that has plagued organizations since people have analyzed data: "Are these numbers correct?" Many new entrants are in the space beside the ones we've mentioned.

Serving Data in Notebooks

Data scientists often use notebooks in their day-to-day work. Whether it's exploring data, engineering features, or training a model, the data scientist will likely use a notebook. At this writing, the most popular notebook platform is Jupyter Notebook, along with its next-generation iteration, JupyterLab. Jupyter is open source and can be hosted locally on a laptop, on a server, or through various cloud-managed services. *Jupyter* stands for *Julia, Python, and R* —the latter two are popular for data science applications, especially notebooks. Regardless of the language used, the first thing you'll need to consider is how data can be accessed from a notebook.

Data scientists will programmatically connect to a data source, such as an API, a database, a data warehouse, or a data lake (Figure 9-4). In a notebook, all connections are created using the appropriate built-in or

imported libraries to load a file from a filepath, connect to an API endpoint, or make an ODBC connection to a database. A remote connection may require the correct credentials and privileges to establish a connection. Once connected, a user may need the correct access to tables (and rows/columns) or files stored in object storage. The data engineer will often assist the data scientist in finding the right data, and then ensure that they have the right permissions to access the rows and columns required.

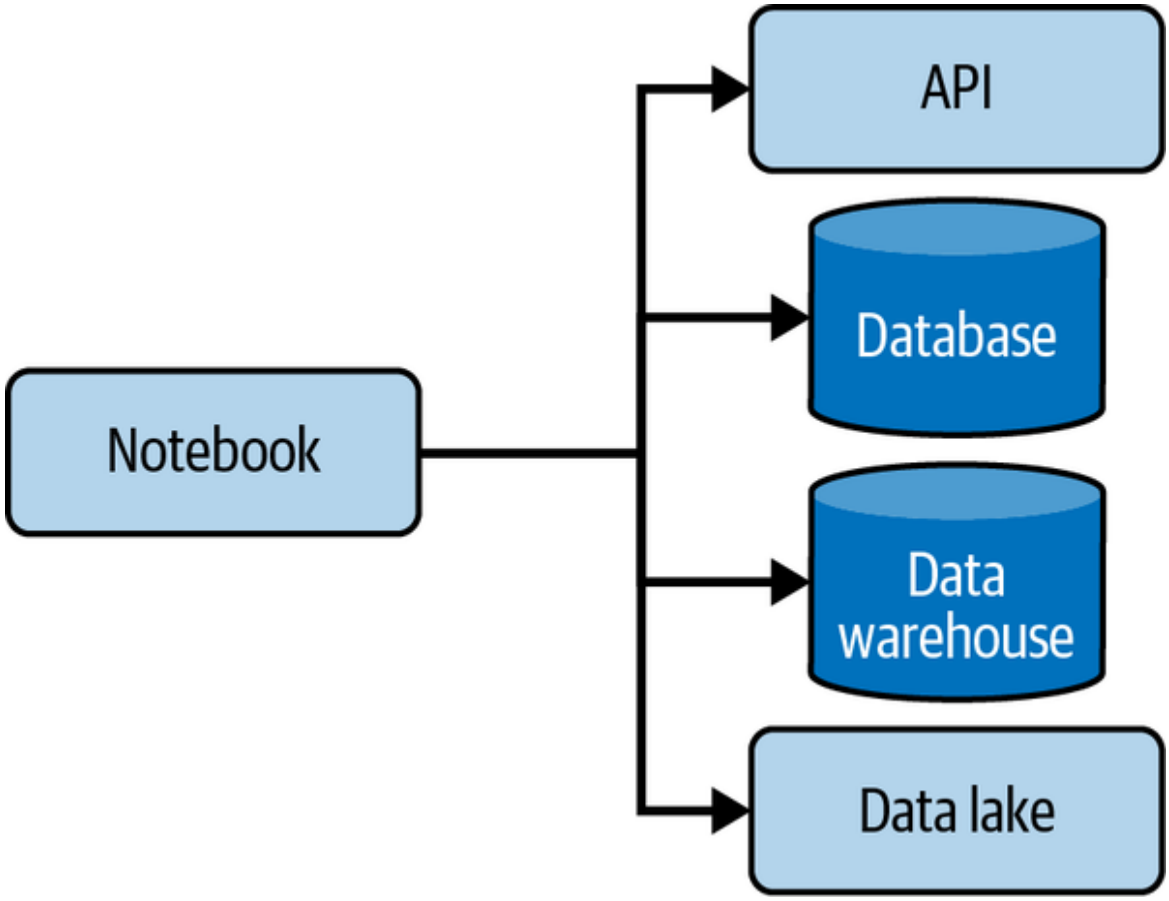

Figure 9-4. A notebook can be served data from many sources, such as object storage, a database, data warehouse, or data lake

CREDENTIAL HANDLING

Incorrectly handled credentials in notebooks and data science code are a major security risk; we constantly see credentials mishandled in this domain. It is common to embed credentials directly in code, where they often leak into version control repos. Credentials are also frequently passed around through messages and email.

We encourage data engineers to audit data science security practices and work collaboratively on improvements. Data scientists are highly receptive to these conversations if they are given alternatives. Data engineers should set standards for handling credentials. Credentials should never be embedded in code; ideally, data scientists use credential managers or CLI tools to manage access.

Let's look at an incredibly common workflow for data scientists: running a local notebook and loading data into a pandas dataframe. *Pandas* is a prevalent Python library used for data manipulation and analysis and is commonly used to load data (say, a CSV file) into a Jupyter notebook. When pandas loads a dataset, it stores this dataset in memory.

What happens when the dataset size exceeds the local machine's available memory? This inevitably happens given the limited memory of laptops and workstations: it stops a data science project dead in its tracks. It's time to consider more scalable options. First, move to a cloud-based notebook where the underlying storage and memory for the notebook can be flexibly scaled. Upon outgrowing this option, look at distributed execution systems; popular Python-based options include Dask, Ray, and Spark. If a full-fledged cloud-managed offering seems appealing, consider setting up a data science workflow using Amazon SageMaker, Google Cloud Vertex AI, or Microsoft Azure Machine Learning. Finally, open source end-to-end ML workflow options such as Kubeflow and MLflow make it easy to scale ML workloads in Kubernetes and Spark, respectively. The point is to get data scientists off their laptops and take advantage of the cloud's power and scalability.

Data engineers and ML engineers play a key role in facilitating the move to scalable cloud infrastructure. The exact division of labor depends a great deal on the details of your organization. They should take the lead in setting up cloud infrastructure, overseeing the management of environments, and training data scientists on cloud-based tools.

Cloud environments require significant operational work, such as managing versions and updates, controlling access, and maintaining SLAs. As with other operational work, a significant payoff can result when "data science ops" are done well.

Notebooks may even become a part of production data science; notebooks are widely deployed at Netflix. This is an interesting approach with advantages and trade-offs. Productionized notebooks allow data scientists to get their work into production much faster, but they are also inherently a substandard form of production. The alternative is to have ML and data engineers convert notebooks for production use, placing a significant burden on these teams. A hybrid of these approaches may be ideal, with notebooks used for "light" production and a full productionization process for high-value projects.

Reverse ETL

Today, *reverse ETL* is a buzzword that describes serving data by loading it from an OLAP database back into a source system. That said, any data engineer who's worked in the field for more than a few years has probably done some variation of reverse ETL. Reverse ETL grew in popularity in the late 2010s/early 2020s and is increasingly recognized as a formal data engineering responsibility.

A data engineer might pull customers and order data from a CRM and store it in a data warehouse. This data is used to train a lead scoring model, whose results are returned to the data warehouse. Your company's sales team wants access to these scored leads to try to generate more sales. You have a few options to get the results of this lead scoring model into the

hands of the sales team. You can put the results in a dashboard for them to view. Or you might email the results to them as an Excel file.

The challenge with these approaches is that they are not connected to the CRM, where a salesperson does their work. Why not just put the scored leads back into the CRM? As we mentioned, successful data products reduce friction with the end user. In this case, the end user is the sales team.

Using reverse ETL and loading the scored leads back into the CRM is the easiest and best approach for this data product. Reverse ETL takes processed data from the output side of the data engineering lifecycle and feeds it back into source systems (Figure 9-5).

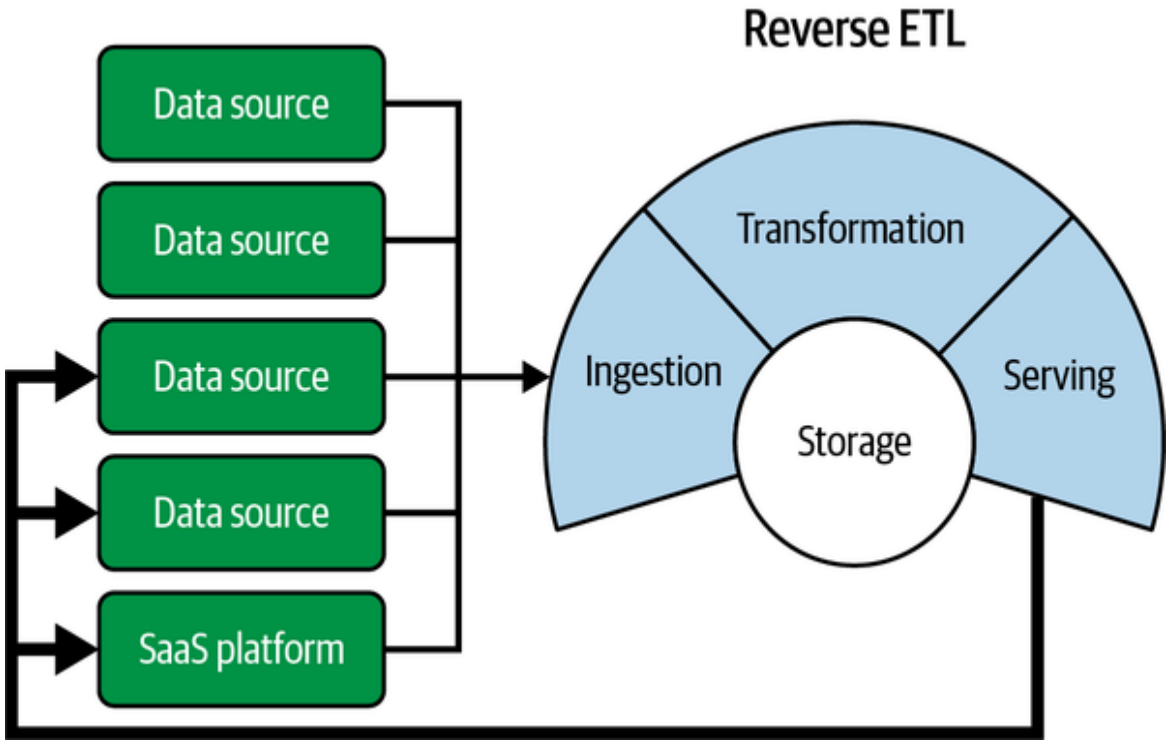

Figure 9-5. Reverse ETL

> **NOTE**
>
> Instead of reverse ETL, we, the authors, half-jokingly call it bidirectional load and transform (BLT). The term *reverse ETL* doesn't quite accurately describe what's happening in this process. Regardless, the term has stuck in the popular imagination and press, so we'll use it throughout the book. More broadly, whether the term *reverse ETL* sticks around is anyone's guess, but the practice of loading data from OLAP systems back into source systems will remain important.

Ways to Serve Data with Reverse ETL

How do you begin serving data with reverse ETL? While you can roll your reverse ETL solution, many off-the-shelf reverse ETL options are available. We suggest using open source, or a commercial managed service. That said, the reverse ETL space is changing extremely quickly. No clear winners have emerged, and many reverse ETL products will be absorbed by major clouds or other data product vendors. Choose carefully.

We do have a few words of warning regarding reverse ETL. Reverse ETL inherently creates feedback loops. For example, imagine that we download Google Ads data, use a model to compute new bids, load the bids back into Google Ads, and start the process again. Suppose that because of an error in your bid model, the bids trend ever higher, and your ads get more and more clicks. You can quickly waste massive amounts of money! Be careful, and build in monitoring and guardrails.

Whom You'll Work With

As we've discussed, in the serving stage, a data engineer will interface with a lot of stakeholders. These include (but aren't limited to) the following:

- Data analysts
- Data scientists
- MLOps/ML engineers

- The business—nondata or nontechnical stakeholders, managers, and executives

As a reminder, the data engineer operates in a *support* role for these stakeholders and is not necessarily responsible for the end uses of data. For example, a data engineer supplies the data for a report that analysts interpret, but the data engineer isn't responsible for these interpretations. Instead, the data engineer is responsible for producing the highest-quality data products possible.

A data engineer should be aware of feedback loops between the data engineering lifecycle and the broader use of data once it's in the hands of stakeholders. Data is rarely static, and the outside world will influence the data that is ingested and served and reingested and re-served.

A big consideration for data engineers in the serving stage of the lifecycle is the separation of duties and concerns. If you're at an early-stage company, the data engineer may also be an ML engineer or data scientist; this is not sustainable. As the company grows, you need to establish a clear division of duties with other data team members.

Adopting a data mesh dramatically reorganizes team responsibilities, and every domain team takes on aspects of serving. For a data mesh to be successful, each team must work effectively on its data serving responsibilities, and teams must also effectively collaborate to ensure organizational success.

Undercurrents

The undercurrents come to finality with serving. Remember that the data engineering lifecycle is just that—a lifecycle. What goes around comes around. We see many instances where serving data highlights something missed earlier in the lifecycle. Always be on the lookout for how the undercurrents can help you spot ways to improve data products.

We're fond of saying, "Data is a silent killer," and the undercurrents come to a head in the serving stage. Serving is your final chance to make sure

your data is in great shape before it gets into the hands of end users.

Security

The same security principles apply whether sharing data with people or systems. We often see data shared indiscriminately, with little to no access controls or thought as to what the data will be used for. This is a huge mistake that can have catastrophic results, such as a data breach and the resulting fines, bad press, and lost jobs. Take security seriously, especially in this stage of the lifecycle. Of all the lifecycle stages, serving presents the largest security surface.

As always, exercise the principle of least privilege both for people and systems, and provide only the access required for the purpose at hand, and the job to be done. What data does an executive need versus an analyst or data scientist? What about an ML pipeline or reverse ETL process? These users and destinations all have different data needs, and access should be provided accordingly. Avoid giving carte blanche permissions to everyone and everything.

Serving data is often read-only unless a person or process needs to update data in the system from which it is queried. People should be given read-only access to specific databases and datasets unless their role requires something more advanced like write or update access. This can be accomplished by combining groups of users with certain IAM roles (i.e., analysts group, data scientist group) or custom IAM roles if this makes sense. For systems, provide service accounts and roles in a similar fashion. For both users and systems, narrow access to a dataset's fields, rows, columns, and cells if this is warranted. Access controls should be as fine-grained as possible and revoked when access is no longer required.

Access controls are critical when serving data in a multitenant environment. Make sure users can access only *their* data and nothing more. A good approach is to mediate access through filtered views, thus alleviating the security risks inherent in sharing access to a common table. Another

suggestion is to use data sharing in your workflows, which allows for read-only granular controls between you and people consuming your data.

Check how often data products are used, and whether it makes sense to stop sharing certain data products. It's extremely common for an executive to urgently request an analyst to create a report, only to have this report very quickly go unused. If data products aren't used, ask the users if they're still needed. If not, kill off the data product. This means one less security vulnerability floating around.

Finally, you should view access control and security not as impediments to serving but as key enablers. We're aware of many instances where complex, advanced data systems were built, potentially having a significant impact on a company. Because security was not implemented correctly, few people were allowed to access the data, so it languished. Fine-grained, robust access control means that more interesting data analytics and ML can be done while still protecting the business and its customers.

Data Management

You've been incorporating data management along the data engineering lifecycle, and the impact of your efforts will soon become apparent as people use your data products. At the serving stage, you're mainly concerned with ensuring that people can access high-quality and trustworthy data.

As we mentioned at the beginning of this chapter, trust is perhaps the most critical variable in data serving. If people trust their data, they will use it; untrusted data will go unused. Be sure to make data trust and data improvement an active process by providing feedback loops. As users interact with data, they can report problems and request improvements. Actively communicate back to your users as changes are made.

What data do people need to do their jobs? Especially with regulatory and compliance concerns weighing on data teams, giving people access to the raw data—even with limited fields and rows—poses a problem of tracing data back to an entity, such as a person or a group of people. Thankfully,

advancements in data obfuscation allow you to serve synthetic, scrambled, or anonymized data to end users. These "fake" datasets should sufficiently allow an analyst or data scientist to get the necessary signal from the data, but in a way that makes identifying protected information difficult. Though this isn't a perfect process—with enough effort, many datasets can be de-anonymized or reverse-engineered—it at least reduces the risk of data leakage.

Also, incorporate semantic and metrics layers into your serving layer, alongside rigorous data modeling that properly expresses business logic and definitions. This provides a single source of truth, whether for analytics, ML, reverse ETL, or other serving uses.

DataOps

The steps you take in data management—data quality, governance, and security—are monitored in DataOps. Essentially, DataOps operationalizes data management. The following are some things to monitor:

- Data health and data downtime
- Latency of systems serving data—dashboards, databases, etc.
- Data quality
- Data and system security and access
- Data and model versions being served
- Uptime to achieve an SLO

A variety of new tools have sprung up to address various monitoring aspects. For example, many popular data observability tools aim to minimize *data downtime* and maximize data quality. Observability tools may cross over from data to ML, supporting monitoring of models and model performance. More conventional DevOps monitoring is also critical to DataOps—e.g., you need to monitor whether connections are stable among storage, transformation, and serving.

As in every stage of the data engineering lifecycle, version-control code and operationalize deployment. This applies to analytical code, data logic code, ML scripts, and orchestration jobs. Use multiple stages of deployment (dev, test, prod) for reports and models.

Data Architecture

Serving data should have the same architectural considerations as other data engineering lifecycle stages. At the serving stage, feedback loops must be fast and tight. Users should be able to access the data they need as quickly as possible when they need it.

Data scientists are notorious for doing most development on their local machines. As discussed earlier, encourage them to migrate these workflows to common systems in a cloud environment, where data teams can collaborate in dev, test, and production environments and create proper production architectures. Facilitate your analysts and data scientists by supporting tools for publishing data insights with little encumbrance.

Orchestration

Data serving is the last stage of the data engineering lifecycle. Because serving is downstream of so many processes, it's an area of extremely complex overlap. Orchestration is not simply a way of organizing and automating complex work but a means of coordinating data flow across teams so that data is made available to consumers at the promised time.

Ownership of orchestration is a key organizational decision. Will orchestration be centralized or decentralized? A decentralized approach allows small teams to manage their data flows, but it can increase the burden of cross-team coordination. Instead of simply managing flows within a single system, directly triggering the completion of DAGs or tasks belonging to other teams, teams must pass messages or queries between systems.

A centralized approach means that work is easier to coordinate, but significant gatekeeping must also exist to protect a single production asset.

For example, a poorly written DAG can bring Airflow to a halt. The centralized approach would mean bringing down data processes and serving across the whole organization. Centralized orchestration management requires high standards, automated testing of DAGs, and gatekeeping.

If orchestration is centralized, who will own it? When a company has a DataOps team, orchestration usually lands here. Often, a team involved in serving is a natural fit because it has a fairly holistic view of all data engineering lifecycle stages. This could be the DBAs, analytics engineers, data engineers, or ML engineers. ML engineers coordinate complex model-training processes but may or may not want to add the operational complexity of managing orchestration to an already crowded docket of responsibilities.

Software Engineering

Compared to a few years ago, serving data has become simpler. The need to write code has been drastically simplified. Data has also become more code-first, with the proliferation of open source frameworks focused on simplifying the serving of data. Many ways exist to serve data to end users, and a data engineer's focus should be on knowing how these systems work and how data is delivered.

Despite the simplicity of serving data, if code is involved, a data engineer should still understand how the main serving interfaces work. For example, a data engineer may need to translate the code a data scientist is running locally on a notebook and convert it into a report or a basic ML model to operate.

Another area where data engineers will be useful is understanding the impact of how code and queries will perform against the storage systems. Analysts can generate SQL in various programmatic ways, including LookML, Jinja via dbt, various object-relational mapping (ORM) tools, and metrics layers. When these programmatic layers compile to SQL, how will this SQL perform? A data engineer can suggest optimizations where the SQL code might not perform as well as handwritten SQL.

The rise of analytics and ML IaC means the role of writing code is moving toward building the systems that support data scientists and analysts. Data engineers might be responsible for setting up the CI/CD pipelines and building processes for their data team. They would also do well to train and support their data team in using the Data/MLOps infrastructure they've built so that these data teams can be as self-sufficient as possible.

For embedded analytics, data engineers may need to work with application developers to ensure that queries are returned quickly and cost-effectively. The application developer will control the frontend code that users deal with. The data engineer is there to ensure that developers receive the correct payloads as they're requested.

Conclusion

The data engineering lifecycle has a logical ending at the serving stage. As with all lifecycles, a feedback loop occurs (Figure 9-6). You should view the serving stage as a chance to learn what's working and what can be improved. Listen to your stakeholders. If they bring up issues—and they inevitably will—try not to take offense. Instead, use this as an opportunity to improve what you've built.

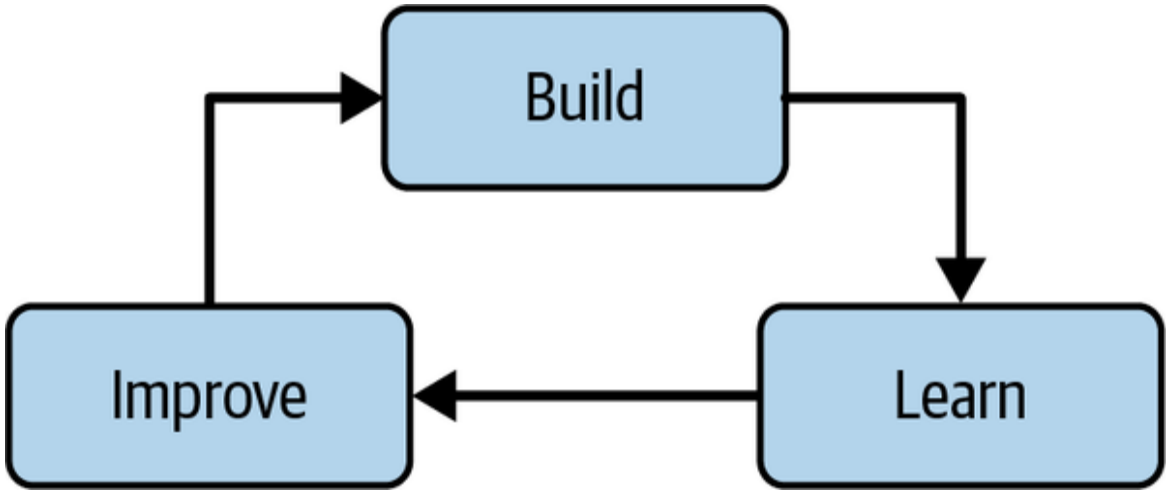

Figure 9-6. Build, learn, improve

A good data engineer is always open to new feedback and constantly finds ways to improve their craft. Now that we've taken a journey through the data engineering lifecycle, you know how to design, architect, build, maintain, and improve your data engineering systems and products. Let's turn our attention to Part III of the book, where we'll cover some aspects of data engineering we're constantly asked about and, frankly, deserve more attention.

Additional Resources

- "Designing Data Products" by Seth O'Regan
- "Data Jujitsu: The Art of Turning Data into Product" by DJ Patil
- "What Is User-Facing Analytics?" by Chinmon Soman
- "Data as a Product vs. Data Products: What Are the Differences?" by Xavier Gumara Rigol
- "How to Build Great Data Products" by Emily Glassberg Sands
- "The Evolution of Data Products" and "What Is Data Science" by Mike Loukides
- "Know Your Customers' 'Jobs to Be Done'" by Clayton M. Christensen et al.
- "Data Mesh Principles and Logical Architecture" by Martin Fowler
- "Understanding the Superset Semantic Layer" by Srini Kadamati
- "The Future of BI Is Headless" by ZD
- "How to Structure a Data Analytics Team" by Niall Napier
- "What Is Operational Analytics (and How Is It Changing How We Work with Data)?" by Sylvain Giuliani

- "Fundamentals of Self-Service Machine Learning" by Paramita (Guha) Ghosh
- "What Do Modern Self-Service BI and Data Analytics Really Mean?" by Harry Dix
- "Self-Service Analytics" in the Gartner Glossary
- "The Missing Piece of the Modern Data Stack" and "Why Is Self-Serve Still a Problem?" by Benn Stancil
- Forrester's "Self-Service Business Intelligence: Dissolving the Barriers to Creative Decision-Support Solutions" blog article
- *Data Mesh* by Zhamak Dehghani (O'Reilly)

---

1 "Know Your Customers' 'Jobs to Be Done'" by Clayton M. Christensen et al., *Harvard Business Review*, September 2016, *https://oreil.ly/3uU4j*.

2 Benn Stancil, "The Missing Piece of the Modern Data Stack," *benn.substack*, April 22, 2021, *https://oreil.ly/wQyPb*.

3 Srini Kadamati, "Understanding the Superset Semantic Layer," Preset blog, December 21, 2021, *https://oreil.ly/6smWC*.

Part III. Security, Privacy, and the Future of Data Engineering

Chapter 10. Security and Privacy

Security is vital to the practice of data engineering. This should be blindingly obvious, but we're constantly amazed at how often data engineers view security as an afterthought. We believe that security is the first thing a data engineer needs to think about in every aspect of their job and every stage of the data engineering lifecycle. You deal with sensitive data, information, and access daily. Your organization, customers, and business partners expect these valuable assets to be handled with the utmost care and concern. One security breach or a data leak can leave your business dead in the water; your career and reputation are ruined if it's your fault.

Security is a key ingredient for privacy. Privacy has long been critical to trust in the corporate information technology space; engineers directly or indirectly handle data related to people's private lives. This includes financial information, data on private communications (emails, texts, phone calls), medical history, educational records, and job history. A company that leaked this information or misused it could find itself a pariah when the breach came to light.

Increasingly, privacy is a matter of significant legal importance. For example, the Family Educational Rights and Privacy Act (FERPA) went into effect in the US in the 1970s; the Health Insurance Portability and Accountability Act (HIPAA) followed in the 1990s; GDPR was passed in Europe in the mid-2010s. Several US-based privacy bills have passed or will soon. This is just a tiny sampling of privacy-related statutes (and we believe just the beginning). Still, the penalties for violation of any of these laws can be significant, even devastating, to a business. And because data systems are woven into the fabric of education, health care, and business, data engineers handle sensitive data related to each of these laws.

A data engineer's exact security and privacy responsibilities will vary significantly between organizations. At a small startup, a data engineer may do double duty as a data security engineer. A large tech company will have armies of security engineers and security researchers. Even in this situation, data engineers will often be able to identify security practices and technology vulnerabilities within their own teams and systems that they can report and mitigate in collaboration with dedicated security personnel.

Because security and privacy are critical to data engineering (security being an undercurrent), we want to spend some more time covering security and privacy. In this chapter, we lay out some things data engineers should consider around security, particularly in people, processes, and technology (in that order). This isn't a complete list, but lays out the major things we'd wish would improve based on our experience.

People

The weakest link in security and privacy is *you*. Security is often compromised at the human level, so conduct yourself as if you're always a target. A bot or human actor is trying to infiltrate your sensitive credentials and information at any given time. This is our reality, and it's not going away. Take a defensive posture with everything you do online and offline. Exercise the power of negative thinking and always be paranoid.

The Power of Negative Thinking

In a world obsessed with positive thinking, negative thinking is distasteful. However, American surgeon Atul Gawande wrote a 2007 op-ed in the *New York Times* on precisely this subject. His central thesis is that positive thinking can blind us to the possibility of terrorist attacks or medical emergencies and deter preparation. Negative thinking allows us to consider disastrous scenarios and act to prevent them.

Data engineers should actively think through the scenarios for data utilization and collect sensitive data only if there is an actual need

downstream. The best way to protect private and sensitive data is to avoid ingesting this data in the first place.

Data engineers should think about the attack and leak scenarios with any data pipeline or storage system they utilize. When deciding on security strategies, ensure that your approach delivers proper security and not just the illusion of safety.

Always Be Paranoid

Always exercise caution when someone asks you for your credentials. When in doubt—and you should always be in extreme doubt when asked for credentials—hold off and get second opinions from your coworkers and friends. Confirm with other people that the request is indeed legitimate. A quick chat or phone call is cheaper than a ransomware attack triggered through an email click. Trust nobody at face value when asked for credentials, sensitive data, or confidential information, including from your coworkers.

You are also the first line of defense in respecting privacy and ethics. Are you uncomfortable with sensitive data you've been tasked to collect? Do you have ethical questions about the way data is being handled in a project? Raise your concerns with colleagues and leadership. Ensure that your work is both legally compliant and ethical.

Processes

When people follow regular security processes, security becomes part of the job. Make security a habit, regularly practice real security, exercise the principle of least privilege, and understand the shared responsibility model in the cloud.

Security Theater Versus Security Habit

With our corporate clients, we see a pervasive focus on compliance (with internal rules, laws, recommendations from standards bodies), but not

enough attention to potentially bad scenarios. Unfortunately, this creates an illusion of security but often leaves gaping holes that would be evident with a few minutes of reflection.

Security needs to be simple and effective enough to become habitual throughout an organization. We're amazed at the number of companies with security policies in the hundreds of pages that nobody reads, the annual security policy review that people immediately forget, all in checking a box for a security audit. This is security theater, where security is done in the letter of compliance (SOC-2, ISO 27001, and related) without real *commitment*.

Instead, pursue the spirit of genuine and habitual security; bake a security mindset into your culture. Security doesn't need to be complicated. For example, at our company, we run security training and policy review at least once a month to ingrain this into our team's DNA and update each other on security practices we can improve. Security must not be an afterthought for your data team. Everyone is responsible and has a role to play. It must be the priority for you and everyone else you work with.

Active Security

Returning to the idea of negative thinking, *active security* entails thinking about and researching security threats in a dynamic and changing world. Rather than simply deploying scheduled simulated phishing attacks, you can take an active security posture by researching successful phishing attacks and thinking through your organizational security vulnerabilities. Rather than simply adopting a standard compliance checklist, you can think about internal vulnerabilities specific to your organization and incentives employees might have to leak or misuse private information.

We have more to say about active security in "Technology".

The Principle of Least Privilege

The *principle of least privilege* means that a person or system should be given only the privileges and data they need to complete the task at hand

and nothing more. Often, we see an antipattern in the cloud: a regular user is given administrative access to everything, when that person may need just a handful of IAM roles to do their work. Giving someone carte blanche administrative access is a huge mistake and should never happen under the principle of least privilege.

Instead, provide the user (or group they belong to) the IAM roles they need when they need them. When these roles are no longer needed, take them away. The same rule applies to service accounts. Treat humans and machines the same way: give them only the privileges and data they need to do their jobs, and only for the timespan when needed.

Of course, the principle of least privilege is also critical to privacy. Your users and customers expect that people will look at their sensitive data only when necessary. Make sure that this is the case. Implement column, row, and cell-level access controls around sensitive data; consider masking PII and other sensitive data and create views that contain only the information the viewer needs to access. Some data must be retained, but should be accessed only in an emergency. Put this data behind a *broken glass process*: users can access it only after going through an emergency approval process to fix a problem, query critical historical information, etc. Access is revoked immediately once the work is done.

Shared Responsibility in the Cloud

Security is a shared responsibility in the cloud. The cloud vendor is responsible for ensuring the physical security of its data center and hardware. At the same time, you are responsible for the security of the applications and systems you build and maintain in the cloud. Most cloud security breaches continue to be caused by end users, not the cloud. Breaches occur because of unintended misconfigurations, mistakes, oversights, and sloppiness.

Always Back Up Your Data

Data disappears. Sometimes it's a dead hard drive or server; in other cases, someone might accidentally delete a database or an object storage bucket. A bad actor can also lock away data. Ransomware attacks are widespread these days. Some insurance companies are reducing payouts in the event of an attack, leaving you on the hook both to recover your data and pay the bad actor who's holding it hostage. You need to back up your data regularly, both for disaster recovery and continuity of business operations, if a version of your data is compromised in a ransomware attack. Additionally, test the restoration of your data backups on a regular basis.

Data backup doesn't strictly fit under security and privacy practices; it goes under the larger heading of *disaster prevention*, but it's adjacent to security, especially in the era of ransomware attacks.

An Example Security Policy

This section presents a sample security policy regarding credentials, devices, and sensitive information. Notice that we don't overcomplicate things; instead, we give people a short list of practical actions they can take immediately.

EXAMPLE SECURITY POLICY

Protect Your Credentials

Protect your credentials at all costs. Here are some ground rules for credentials:

- Use a single-sign-on (SSO) for everything. Avoid passwords whenever possible, and use SSO as the default.
- Use multifactor authentication with SSO.
- Don’t share passwords or credentials. This includes client passwords and credentials. If in doubt, see the person you report to. If that person is in doubt, keep digging until you find an answer.
- Beware of phishing and scam calls. Don’t ever give your passwords out. (Again, prioritize SSO.)
- Disable or delete old credentials. Preferably the latter.
- Don’t put your credentials in code. Handle secrets as configuration and never commit them to version control. Use a secrets manager where possible.
- Always exercise the principle of least privilege. Never give more access than is required to do the job. This applies to all credentials and privileges in the cloud and on premises.

Protect Your Devices

- Use device management for all devices used by employees. If an employee leaves the company or your device gets lost, the device can be remotely wiped.
- Use multifactor authentication for all devices.
- Sign in to your device using your company email credentials.

- All policies covering credentials and behavior apply to your device(s).
- Treat your device as an extension of yourself. Don't let your assigned device(s) out of your sight.
- When screen sharing, be aware of exactly what you're sharing to protect sensitive information and communications. Share only single documents, browser tabs, or windows, and avoid sharing your full desktop. Share only what's required to convey your point.
- Use "do not disturb" mode when on video calls; this prevents messages from appearing during calls or recordings.

Software Update Policy

- Restart your web browser when you see an update alert.
- Run minor OS updates on company and personal devices.
- The company will identify critical major OS updates and provide guidance.
- Don't use the beta version of an OS.
- Wait a week or two for new major OS version releases.

These are some basic examples of how security can be simple and effective. Based on your company's security profile, you may need to add more requirements for people to follow. And again, always remember that people are your weakest link in security.

Technology

After you've addressed security with people and processes, it's time to look at how you leverage technology to secure your systems and data assets. The following are some significant areas you should prioritize.