## Patch and Update Systems

Software gets stale, and security vulnerabilities are constantly discovered. To avoid exposing a security flaw in an older version of the tools you're using, always patch and update operating systems and software as new updates become available. Thankfully, many SaaS and cloud-managed services automatically perform upgrades and other maintenance without your intervention. To update your own code and dependencies, either automate builds or set alerts on releases and vulnerabilities so you can be prompted to perform the updates manually.

## Encryption

Encryption is not a magic bullet. It will do little to protect you in the event of a *human* security breach that grants access to credentials. Encryption is a baseline requirement for any organization that respects security and privacy. It will protect you from basic attacks, such as network traffic interception.

Let's look separately at encryption at rest and in transit.

### Encryption at rest

Be sure your data is encrypted when it is at rest (on a storage device). Your company laptops should have full-disk encryption enabled to protect data if a device is stolen. Implement server-side encryption for all data stored in servers, filesystems, databases, and object storage in the cloud. All data backups for archival purposes should also be encrypted. Finally, incorporate application-level encryption where applicable.

### Encryption over the wire

Encryption over the wire is now the default for current protocols. For instance, HTTPS is generally required for modern cloud APIs. Data engineers should always be aware of how keys are handled; bad key handling is a significant source of data leaks. In addition, HTTPS does nothing to protect data if bucket permissions are left open to the public, another cause of several data scandals over the last decade.

Engineers should also be aware of the security limitations of older protocols. For example, FTP is simply not secure on a public network. While this may not appear to be a problem when data is already public, FTP is vulnerable to man-in-the-middle attacks, whereby an attacker intercepts downloaded data and changes it before it arrives at the client. It is best to simply avoid FTP.

Make sure everything is encrypted over the wire, even with legacy protocols. When in doubt, use robust technology with encryption baked in.

## Logging, Monitoring, and Alerting

Hackers and bad actors typically don't announce that they're infiltrating your systems. Most companies don't find out about security incidents until well after the fact. Part of DataOps is to observe, detect, and alert on incidents. As a data engineer, you should set up automated monitoring, logging, and alerting to be aware of peculiar events when they happen in your systems. If possible, set up automatic anomaly detection.

Here are some areas you should monitor:

*Access*

> Who's accessing what, when, and from where? What new accesses were granted? Are there strange patterns with your current users that might indicate their account is compromised, such as trying to access systems they don't usually access or shouldn't have access to? Do you see new unrecognized users accessing your system? Be sure to regularly comb through access logs, users, and their roles to ensure that everything looks OK.

*Resources*

> Monitor your disk, CPU, memory, and I/O for patterns that seem out of the ordinary. Did your resources suddenly change? If so, this might

indicate a security breach.

*Billing*

Especially with SaaS and cloud-managed services, you need to oversee costs. Set up budget alerts to make sure your spending is within expectations. If an unexpected spike occurs in your billing, this might indicate someone or something is utilizing your resources for malicious purposes.

*Excess permissions*

Increasingly, vendors are providing tools that monitor for permissions that are *not utilized* by a user or service account over some time. These tools can often be configured to automatically alert an administrator or remove permissions after a specified elapsed time.

For example, suppose that a particular analyst hasn't accessed Redshift for six months. These permissions can be removed, closing a potential security hole. If the analyst needs to access Redshift in the future, they can put in a ticket to restore permissions.

It's best to combine these areas in your monitoring to get a cross-sectional view of your resource, access, and billing profile. We suggest setting up a dashboard for everyone on the data team to view monitoring and receive alerts when something seems out of the ordinary. Couple this with an effective incident response plan to manage security breaches when they occur, and run through the plan on a regular basis so you are prepared.

## Network Access

We often see data engineers doing pretty wild things regarding network access. In several instances, we've seen publicly available Amazon S3 buckets housing lots of sensitive data. We've also witnessed Amazon EC2 instances with inbound SSH access open to the whole world for 0.0.0.0/0

(all IPs) or databases with open access to all inbound requests over the public internet. These are just a few examples of terrible network security practices.

In principle, network security should be left to security experts at your company. (In practice, you may need to assume significant responsibility for network security in a small company.) As a data engineer, you will encounter databases, object storage, and servers so often that you should at least be aware of simple measures you can take to make sure you're in line with good network access practices. Understand what IPs and ports are open, to whom, and why. Allow the incoming IP addresses of the systems and users that will access these ports and avoid broadly opening connections for any reason. When accessing the cloud or a SaaS tool, use an encrypted connection. For example, don't use an unencrypted website from a coffee shop.

Also, while this book has focused almost entirely on running workloads in the cloud, we add a brief note here about hosting on-premises servers. Recall that in Chapter 3, we discussed the difference between a hardened perimeter and zero-trust security. The cloud is generally closer to zero-trust security—every action requires authentication. We believe that the cloud is a more secure option for most organizations because it imposes zero-trust practices and allows companies to leverage the army of security engineers employed by the public clouds.

However, sometimes hardened perimeter security still makes sense; we find some solace in the knowledge that nuclear missile silos are air gapped (not connected to any networks). Air-gapped servers are the ultimate example of a hardened security perimeter. Just keep in mind that even on premises, air-gapped servers are vulnerable to human security failings.

## Security for Low-Level Data Engineering

For engineers who work in the guts of data storage and processing systems, it is critical to consider the security implications of every element. Any software library, storage system, or compute node is a potential security

vulnerability. A flaw in an obscure logging library might allow attackers to bypass access controls or encryption. Even CPU architectures and microcode represent potential vulnerabilities; sensitive data can be vulnerable when it's at rest in memory or a CPU cache. No link in the chain can be taken for granted.

Of course, this book is principally about high-level data engineering—stitching together tools to handle the entire lifecycle. Thus, we'll leave it to you to dig into the gory technical details.

### Internal security research

We discussed the idea of *active security* in "Processes". We also highly recommend adopting an *active security* approach to technology. Specifically, this means that every technology employee should think about security problems.

Why is this important? Every technology contributor develops a domain of technical expertise. Even if your company employs an army of security researchers, data engineers will become intimately familiar with specific data systems and cloud services in their purview. Experts in a particular technology are well positioned to identify security holes in this technology.

Encourage every data engineer to be actively involved in security. When they identify potential security risks in their systems, they should think through mitigations and take an active role in deploying these.

## Conclusion

Security needs to be a habit of mind and action; treat data like your wallet or smartphone. Although you won't likely be in charge of security for your company, knowing basic security practices and keeping security top of mind will help reduce the risk of data security breaches at your organization.

# Additional Resources

- Open Web Application Security Project (OWASP) publications
- *Building Secure and Reliable Systems* by Heather Adkins et al. (O'Reilly)
- *Practical Cloud Security* by Chris Dotson (O'Reilly)

# Chapter 11. The Future of Data Engineering

This book grew out of the authors' recognition that warp speed changes in the field have created a significant knowledge gap for existing data engineers, people interested in moving into a career in data engineering, technology managers, and executives who want to better understand how data engineering fits into their companies. When we started thinking about how to organize this book, we got quite a bit of pushback from friends who'd ask, "How dare you write about a field that is changing so quickly?!" In many ways, they're right. It certainly feels like the field of data engineering—and, really, all things data—is changing daily. Sifting through the noise and finding the signal of *what's unlikely to change* was among the most challenging parts of organizing and writing this book.

In this book, we focus on big ideas that we feel will be useful for the next several years—hence the continuum of the data engineering lifecycle and its undercurrents. The order of operations and names of best practices and technologies might change, but the primary stages of the lifecycle will likely remain intact for many years to come. We're keenly aware that technology continues to change at an exhausting pace; working in the technology sector in our present era can feel like a rollercoaster ride or perhaps a hall of mirrors.

Several years ago, data engineering didn't even exist as a field or job title. Now you're reading a book called *Fundamentals of Data Engineering*! You've learned all about the fundamentals of data engineering—its lifecycle, undercurrents, technologies, and best practices. You might be asking yourself, what's next in data engineering? While nobody can predict the future, we have a good perspective on the past, the present, and current trends. We've been fortunate to watch the genesis and evolution of data engineering from a front-row seat. This final chapter presents our thoughts

on the future, including observations of ongoing developments and wild future speculation.

## The Data Engineering Lifecycle Isn't Going Away

While data science has received the bulk of the attention in recent years, data engineering is rapidly maturing into a distinct and visible field. It's one of the fastest-growing careers in technology, with no signs of losing momentum. As companies realize they first need to build a data foundation before moving to "sexier" things like AI and ML, data engineering will continue growing in popularity and importance. This progress centers around the data engineering lifecycle.

Some question whether increasingly simple tools and practices will lead to the disappearance of data engineers. This thinking is shallow, lazy, and shortsighted. As organizations leverage data in new ways, new foundations, systems, and workflows will be needed to address these needs. Data engineers sit at the center of designing, architecting, building, and maintaining these systems. If tooling becomes easier to use, data engineers will move up the value chain to focus on higher-level work. The data engineering lifecycle isn't going away anytime soon.

## The Decline of Complexity and the Rise of Easy-to-Use Data Tools

Simplified, easy-to-use tools continue to lower the barrier to entry for data engineering. This is a great thing, especially given the shortage of data engineers we've discussed. The trend toward simplicity will continue. Data engineering isn't dependent on a particular technology or data size. It's also not just for large companies. In the 2000s, deploying "big data" technologies required a large team and deep pockets. The ascendance of SaaS-managed services has largely removed the complexity of

understanding the guts of various "big data" systems. Data engineering is now something that *all* companies can do.

Big data is a victim of its extraordinary success. For example, Google BigQuery, a descendant of GFS and MapReduce, can query petabytes of data. Once reserved for internal use at Google, this insanely powerful technology is now available to anybody with a GCP account. Users simply pay for the data they store and query rather than having to build a massive infrastructure stack. Snowflake, Amazon EMR, and many other hyper-scalable cloud data solutions compete in the space and offer similar capabilities.

The cloud is responsible for a significant shift in the usage of open source tools. Even in the early 2010s, using open source typically entailed downloading the code and configuring it yourself. Nowadays, many open source data tools are available as managed cloud services that compete directly with proprietary services. Linux is available preconfigured and installed on server instances on all major clouds. Serverless platforms like AWS Lambda and Google Cloud Functions allow you to deploy event-driven applications in minutes, using mainstream languages such as Python, Java, and Go running atop Linux behind the scenes. Engineers wishing to use Apache Airflow can adopt Google's Cloud Composer or AWS's managed Airflow service. Managed Kubernetes allows us to build highly scalable microservice architectures. And so on.

This fundamentally changes the conversation around open source code. In many cases, managed open source is just as easy to use as its proprietary service competitors. Companies with highly specialized needs can also deploy managed open source, then move to self-managed open source later if they need to customize the underlying code.

Another significant trend is the growth in popularity of off-the-shelf data connectors (at the time of this writing, popular ones include Fivetran and Airbyte). Data engineers have traditionally spent a lot of time and resources building and maintaining plumbing to connect to external data sources. The new generation of managed connectors is highly compelling, even for

highly technical engineers, as they begin to recognize the value of recapturing time and mental bandwidth for other projects. API connectors will be an outsourced problem so that data engineers can focus on the unique issues that drive their businesses.

The intersection of red-hot competition in the data-tooling space with a growing number of data engineers means data tools will continue decreasing in complexity while adding even more functionality and features. This simplification will only grow the practice of data engineering, as more and more companies find opportunities to discover value in data.

## The Cloud-Scale Data OS and Improved Interoperability

Let's briefly review some of the inner workings of (single-device) operating systems, then tie this back to data and the cloud. Whether you're utilizing a smartphone, a laptop, an application server, or a smart thermostat, these devices rely on an operating system to provide essential services and orchestrate tasks and processes. For example, I can see roughly 300 processes running on the MacBook Pro that I'm typing on. Among other things, I see services such as WindowServer (responsible for providing windows in a graphical interface) and CoreAudio (tasked with providing low-level audio capabilities).

When I run an application on this machine, it doesn't directly access sound and graphics hardware. Instead, it sends commands to operating system services to draw windows and play sound. These commands are issued to standard APIs; a specification tells software developers how to communicate with operating system services. The operating system *orchestrates* a boot process to provide these services, starting each service in the correct order based on dependencies among them; it also maintains services by monitoring them and restarting them in the correct order in case of a failure.

Now let's return to data in the cloud. The simplified data services that we've mentioned throughout this book (e.g., Google Cloud BigQuery, Azure Blob Storage, Snowflake, and AWS Lambda) resemble operating system services, but at a much larger scale, running across many machines rather than a single server.

Now that these simplified services are available, the next frontier of evolution for this notion of a cloud data operating system will happen at a higher level of abstraction. Benn Stancil called for the emergence of standardized data APIs for building data pipelines and data applications.[1] We predict that data engineering will gradually coalesce around a handful of data interoperability standards. Object storage in the cloud will grow in importance as a batch interface layer between various data services. New generation file formats (such as Parquet and Avro) are already taking over for the purposes of cloud data interchange, significantly improving on the dreadful interoperability of CSV, and the poor performance of raw JSON.

Another critical ingredient of a data API ecosystem is a metadata catalog that describes schemas and data hierarchies. Currently, this role is largely filled by the legacy Hive Metastore. We expect that new entrants will emerge to take its place. Metadata will play a crucial role in data interoperability, both across applications and systems, and across clouds and networks, driving automation and simplification.

We will also see significant improvements in the scaffolding that manages cloud data services. Apache Airflow has emerged as the first truly cloud-oriented data orchestration platform, but we are on the cusp of significant enhancement. Airflow will grow in capabilities, building on its massive mindshare. New entrants such as Dagster and Prefect will compete by rebuilding orchestration architecture from the ground up.

This next generation of data orchestration platforms will feature enhanced data integration and data awareness. Orchestration platforms will integrate with data cataloging and lineage, becoming significantly more data-aware in the process. In addition, orchestration platforms will build IaC capabilities (similar to Terraform) and code deployment features (like

GitHub Actions and Jenkins). This will allow engineers to code a pipeline, and then pass it to the orchestration platform to automatically build, test, deploy and monitor. Engineers will be able to write infrastructure specifications directly into their pipelines; missing infrastructure and services (e.g., Snowflake databases, Databricks clusters, and Amazon Kinesis streams) will be deployed the first time the pipeline runs.

We will also see significant enhancements in the domain of *live data*—e.g., streaming pipelines and databases capable of ingesting and querying streaming data. In the past, building a streaming DAG was an extremely complex process with a high ongoing operational burden (see Chapter 8). Tools like Apache Pulsar point the way toward a future in which streaming DAGs can be deployed with complex transformations using relatively simple code. We have already seen the emergence of managed stream processors (such as Amazon Kinesis Data Analytics and Google Cloud Dataflow), but we will see a new generation of orchestration tools for managing these services, stitching them together, and monitoring them. We discuss live data in "The Live Data Stack".

What does this enhanced abstraction mean for data engineers? As we've already argued in this chapter, the role of the data engineer won't go away, but it will evolve significantly. By comparison, more sophisticated mobile operating systems and frameworks have not eliminated mobile app developers. Instead, mobile app developers can now focus on building better-quality, more sophisticated applications. We expect similar developments for data engineering as the cloud-scale data OS paradigm increases interoperability and simplicity across various applications and systems.

## "Enterprisey" Data Engineering

The increasing simplification of data tools and the emergence and documentation of best practices means data engineering will become more "enterprisey."[2] This will make many readers violently cringe. The term *enterprise*, for some, conjures Kafkaesque nightmares of faceless

committees dressed in overly starched blue shirts and khakis, endless red tape, and waterfall-managed development projects with constantly slipping schedules and ballooning budgets. In short, some of you read "enterprise" and imagine a soulless place where innovation goes to die.

Fortunately, this is not what we're talking about; we're referring to some of the *good* things that larger companies do with data—management, operations, governance, and other "boring" stuff. We're presently living through the golden age of "enterprisey" data management tools. Technologies and practices once reserved for giant organizations are trickling downstream. The once hard parts of big data and streaming data have now largely been abstracted away, with the focus shifting to ease of use, interoperability, and other refinements.

This allows data engineers working on new tooling to find opportunities in the abstractions of data management, DataOps, and all the other undercurrents of data engineering. Data engineers will become "enterprisey." Speaking of which…

## Titles and Responsibilities Will Morph...

While the data engineering lifecycle isn't going anywhere anytime soon, the boundaries between software engineering, data engineering, data science, and ML engineering are increasingly fuzzy. In fact, like the authors, many data scientists are transformed into data engineers through an organic process; tasked with doing "data science" but lacking the tools to do their jobs, they take on the job of designing and building systems to serve the data engineering lifecycle.

As simplicity moves up the stack, data scientists will spend a smaller slice of their time gathering and munging data. But this trend will extend beyond data scientists. Simplification also means data engineers will spend less time on low-level tasks in the data engineering lifecycle (managing servers, configuration, etc.), and "enterprisey" data engineering will become more prevalent.

As data becomes more tightly embedded in every business's processes, new roles will emerge in the realm of data and algorithms. One possibility is a role that sits between ML engineering and data engineering. As ML toolsets become easier to use and managed cloud ML services grow in capabilities, ML is shifting away from ad hoc exploration and model development to become an operational discipline.

This new ML-focused engineer who straddles this divide will know algorithms, ML techniques, model optimization, model monitoring, and data monitoring. However, their primary role will be to create or utilize the systems that automatically train models, monitor performance, and operationalize the full ML process for model types that are well understood. They will also monitor data pipelines and quality, overlapping into the current realm of data engineering. ML engineers will become more specialized to work on model types that are closer to research and less well understood.

Another area in which titles may morph is at the intersection of software engineering and data engineering. Data applications, which blend traditional software applications with analytics, will drive this trend. Software engineers will need to have a much deeper understanding of data engineering. They will develop expertise in things like streaming, data pipelines, data modeling, and data quality. We will move beyond the "throw it over the wall" approach that is now pervasive. Data engineers will be integrated into application development teams, and software developers will acquire data engineering skills. The boundaries that exist between application backend systems and data engineering tools will be lowered as well, with deep integration through streaming and event-driven architectures.

## Moving Beyond the Modern Data Stack, Toward the Live Data Stack

We'll be frank: the modern data stack (MDS) isn't so modern. We applaud the MDS for bringing a great selection of powerful data tools to the masses,

lowering prices, and empowering data analysts to take control of their data stack. The rise of ELT, cloud data warehouses, and the abstraction of SaaS data pipelines certainly changed the game for many companies, opening up new powers for BI, analytics, and data science.

Having said that, the MDS is basically a repackaging of old data warehouse practices using modern cloud and SaaS technologies; because the MDS is built around the cloud data warehouse paradigm, it has some serious limitations when compared to the potential of next-generation real-time data applications. From our point of view, the world is moving beyond the use of data-warehouse-based internal-facing analytics and data science, toward powering entire businesses and applications in real time with next generation real-time databases.

What's driving this evolution? In many cases, analytics (BI and operational analytics) will be replaced by automation. Presently, most dashboards and reports answer questions concerning *what* and *when*. Ask yourself, "If I'm asking a *what* or *when* question, what action do I take next?" If the action is repetitive, it is a candidate for automation. Why look at a report to determine whether to take action when you can instead automate the action based on events as they occur?

And it goes much further than this. Why does using a product like TikTok, Uber, Google, or DoorDash feel like magic? While it seems to you like a click of a button to watch a short video, order a ride or a meal, or find a search result, a lot is happening under the hood. These products are examples of true real-time data applications, delivering the actions you need at the click of a button while performing extremely sophisticated data processing and ML behind the scenes with miniscule latency. Presently, this level of sophistication is locked away behind custom-built technologies at large technology companies, but this sophistication and power are becoming democratized, similar to the way the MDS brought cloud-scale data warehouses and pipelines to the masses. The data world will soon go "live."

## The Live Data Stack

This democratization of real-time technologies will lead us to the successor to the MDS: the *live data stack* will soon be accessible and pervasive. The live data stack, depicted in Figure 11-1, will fuse real-time analytics and ML into applications by using streaming technologies, covering the full data lifecycle from application source systems to data processing to ML, and back.

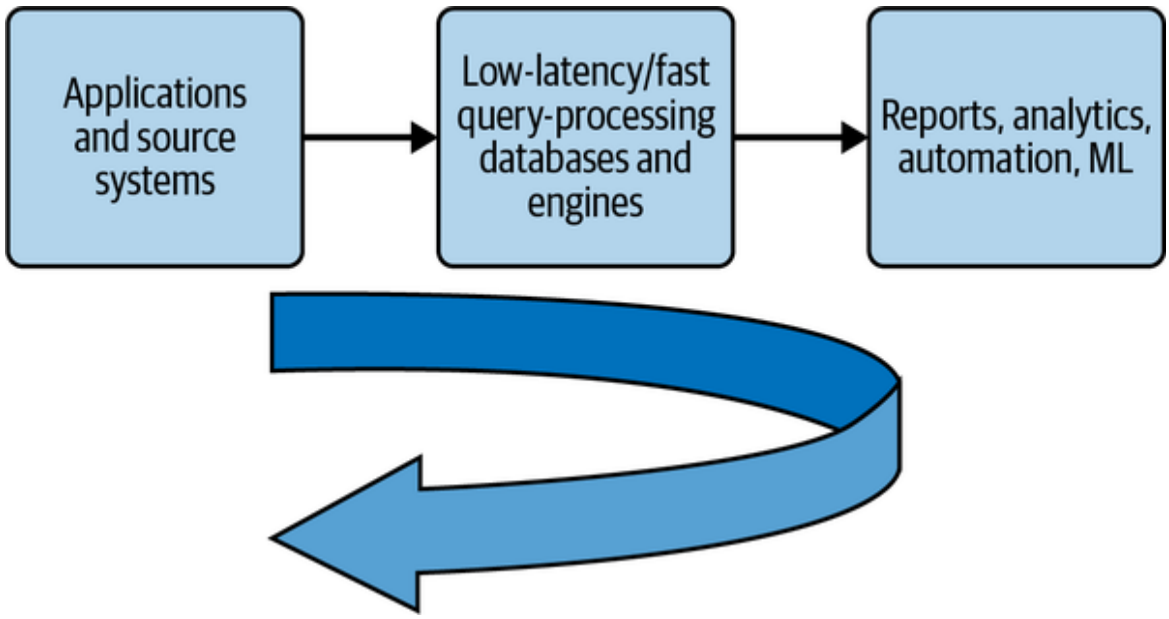


*Figure 11-1. In the live data stack, data and intelligence moves in real time between the application and supporting systems*

Just as the MDS took advantage of the cloud and brought on-premises data warehouse and pipeline technologies to the masses, the live data stack takes real-time data application technologies used at elite tech companies and makes them available to companies of all sizes as easy-to-use cloud-based offerings. This will open up a new world of possibilities for creating even better user experiences and business value.

## Streaming Pipelines and Real-Time Analytical Databases

The MDS limits itself to batch techniques that treat data as bounded. In contrast, real-time data applications treat data as an unbounded, continuous stream. Streaming pipelines and real-time analytical databases are the two core technologies that will facilitate the move from the MDS to the live data

stack. While these technologies have been around for some time, rapidly maturing managed cloud services will see them be deployed much more widely.

Streaming technologies will continue to see extreme growth for the foreseeable future. This will happen in conjunction with a clearer focus on the business utility of streaming data. Up to the present, streaming systems have frequently been treated like an expensive novelty or a dumb pipe for getting data from A to B. In the future, streaming will radically transform organizational technology and business processes; data architects and engineers will take the lead in these fundamental changes.

Real-time analytical databases enable both fast ingestion and subsecond queries on this data. This data can be enriched or combined with historical datasets. When combined with a streaming pipeline and automation, or dashboard that is capable of real-time analytics, a whole new level of possibilities opens up. No longer are you constrained by slow-running ELT processes, 15-minute updates, or other slow-moving parts. Data moves in a continuous flow. As streaming ingestion becomes more prevalent, batch ingestion will be less and less common. Why create a batch bottleneck at the head of your data pipeline? We'll eventually look at batch ingestion the same way we now look at dial-up modems.

In conjunction with the rise of streams, we expect a back-to-the-future moment for data transformations. We'll shift away from ELT—in database transformations—to something that looks more like ETL. We provisionally refer to this as *stream, transform, and load* (STL). In a streaming context, extraction is an ongoing, continuous process. Of course, batch transformations won't entirely go away. Batch will still be very useful for model training, quarterly reporting, and more. But streaming transformation will become the norm.

While the data warehouse and data lake are great for housing large amounts of data and performing ad hoc queries, they are not so well optimized for low-latency data ingestion or queries on rapidly moving data. The live data stack will be powered by OLAP databases that are purpose-built for

streaming. Today, databases like Druid, ClickHouse, Rockset, and Firebolt are leading the way in powering the backend of the next generation of data applications. We expect that streaming technologies will continue to evolve rapidly and that new technologies will proliferate.

Another area we think is ripe for disruption is data modeling, where there hasn't been serious innovation since the early 2000s. The traditional batch-oriented data modeling techniques you learned about in Chapter 8 aren't suited for streaming data. New data-modeling techniques will occur not within the data warehouse, but in the systems that generate the data. We expect data modeling will involve some notion of an upstream definitions layer—including semantics, metrics, lineage, and data definitions (see Chapter 9)—beginning where data is generated in the application. Modeling will also happen at every stage as data flows and evolves through the full lifecycle.

## The Fusion of Data with Applications

We expect the next revolution will be the fusion of the application and data layers. Right now, applications sit in one area, and the MDS sits in another. To make matters worse, data is created with no regard for how it will be used for analytics. Consequently, lots of duct tape is needed to make systems talk with one another. This patchwork, siloed setup is awkward and ungainly.

Soon, application stacks will be data stacks, and vice versa. Applications will integrate real-time automation and decision making, powered by the streaming pipelines and ML. The data engineering lifecycle won't necessarily change, but the time between stages of the lifecycle will drastically shorten. A lot of innovation will occur in new technologies and practices that will improve the experience of engineering the live data stack. Pay attention to emerging database technologies designed to address the mix of OLTP and OLAP use cases; feature stores may also play a similar role for ML use cases.

## The Tight Feedback Between Applications and ML

Another area we're excited about is the fusion of applications and ML. Today, applications and ML are disjointed systems, like applications and analytics. Software engineers do their thing over here, data scientists and ML engineers do their thing over there.

ML is well-suited for scenarios where data is generated at such a high rate and volume that humans cannot feasibly process it by hand. As data sizes and velocity grow, this applies to every scenario. High volumes of fast-moving data, coupled with sophisticated workflows and actions, are candidates for ML. As data feedback loops become shorter, we expect most applications to integrate ML. As data moves more quickly, the feedback loop between applications and ML will tighten. The applications in the live data stack are intelligent and able to adapt in real time to changes in the data. This creates a cycle of ever-smarter applications and increasing business value.

## Dark Matter Data and the Rise of...Spreadsheets?!

We've talked about fast-moving data and how feedback loops will shrink as applications, data, and ML work more closely together. This section might seem odd, but we need to address something that's widely ignored in today's data world, especially by engineers.

What's the most widely used data platform? It's the humble spreadsheet. Depending on the estimates you read, the user base of spreadsheets is between 700 million and 2 billion people. Spreadsheets are the dark matter of the data world. A good deal of data analytics runs in spreadsheets and never makes its way into the sophisticated data systems that we describe in this book. In many organizations, spreadsheets handle financial reporting, supply-chain analytics, and even CRM.

At heart, what is a spreadsheet? A *spreadsheet* is an interactive data application that supports complex analytics. Unlike purely code-based tools such as pandas (Python Data Analysis Library), spreadsheets are accessible to a whole spectrum of users, ranging from those who just know how to

open files and look at reports to power users who can script sophisticated procedural data processing. So far, BI tools have failed to bring comparable interactivity to databases. Users who interact with the UI are typically limited to slicing and dicing data within certain guardrails, not general-purpose programmable analytics.

We predict that a new class of tools will emerge that combines the interactive analytics capabilities of a spreadsheet with the backend power of cloud OLAP systems. Indeed, some candidates are already in the running. The ultimate winner in this product category may continue to use spreadsheet paradigms, or may define entirely new interface idioms for interacting with data.

## Conclusion

Thank you for joining us on this journey through data engineering! We traversed good architecture, the stages of the data engineering lifecycle, and security best practices. We've discussed strategies for choosing technologies at a time when our field continues to change at an extraordinary pace. In this chapter, we laid out our wild speculation about the near and intermediate future.

Some aspects of our prognostication sit on a relatively secure footing. The simplification of managed tooling and the rise of "enterprisey" data engineering have proceeded day by day as we've written this book. Other predictions are much more speculative in nature; we see hints of an emerging *live data stack*, but this entails a significant paradigm shift for both individual engineers and the organizations that employ them. Perhaps the trend toward real-time data will stall once again, with most companies continuing to focus on basic batch processing. Surely, other trends exist that we have completely failed to identify. The evolution of technology involves complex interactions of technology and culture. Both are unpredictable.

Data engineering is a vast topic; while we could not go into any technical depth in individual areas, we hope that we have succeeded in creating a kind of travel guide that will help current data engineers, future data

engineers, and those who work adjacent to the field to find their way in a domain that is in flux. We advise you to continue exploration on your own. As you discover interesting topics and ideas in this book, continue the conversation as part of a community. Identify domain experts who can help you to uncover the strengths and pitfalls of trendy technologies and practices. Read extensively from the latest books, blog posts, and papers. Participate in meetups and listen to talks. Ask questions and share your own expertise. Keep an eye on vendor announcements to stay abreast of the latest developments, taking all claims with a healthy grain of salt.

Through this process, you can choose technology. Next, you will need to adopt technology and develop expertise, perhaps as an individual contributor, perhaps within your team as a lead, perhaps across an entire technology organization. As you do this, don't lose sight of the larger goals of data engineering. Focus on the lifecycle, on serving your customers—internal and external—on your business, on serving and on your larger goals.

Regarding the future, many of you will play a role in determining what comes next. Technology trends are defined not only by those who create the underlying technology, but by those who adopt it and put it to good use. Successful tool *use* is as critical as tool *creation*. Find opportunities to apply real-time technology that will improve the user experience, create value, and define entirely new types of applications. It is this kind of practical application that will materialize the *live data stack* as a new industry standard; or perhaps some other new technology trend that we failed to identify will win the day.

Finally, we wish you an exciting career! We chose to work in data engineering, to consult, and to write this book not simply because it was trendy, but because it was fascinating. We hope that we've managed to convey to you a bit of the joy we've found working in this field.

---

1 Benn Stancil, "The Data OS," *benn.substack*, September 3, 2021, *https://oreil.ly/HetE9*.

2 Ben Rogojan, “Three Data Engineering Experts Share Their Thoughts on Where Data Is Headed,” *Better Programming*, May 27, 2021, *https://oreil.ly/IsY4W*.

# Appendix A. Serialization and Compression Technical Details

Data engineers working in the cloud are generally freed from the complexities of managing object storage systems. Still, they need to understand details of serialization and deserialization formats. As we mentioned in Chapter 6 about storage raw ingredients, serialization and compression algorithms go hand in hand.

## Serialization Formats

Many serialization algorithms and formats are available to data engineers. While the abundance of options is a significant source of pain in data engineering, they are also a massive opportunity for performance improvements. We've sometimes seen job performance improve by a factor of 100 simply by switching from CSV to Parquet serialization. As data moves through a pipeline, engineers will also manage reserialization—conversion from one format to another. Sometimes data engineers have no choice but to accept data in an ancient, nasty form; they must design processes to deserialize this format and handle exceptions, and then clean up and convert data for consistent, fast downstream processing and consumption.

### Row-Based Serialization

As its name suggests, *row-based serialization* organizes data by row. CSV format is an archetypal row-based format. For semistructured data (data objects that support nesting and schema variation), row-oriented serialization entails storing each object as a unit.

#### CSV: The nonstandard standard

We discussed CSV in Chapter 7. CSV is a serialization format that data engineers love to hate. The term *CSV* is essentially a catchall for delimited text, but there is flexibility in conventions of escaping, quote characters, delimiter, and more.

Data engineers should avoid using CSV files in pipelines because they are highly error-prone and deliver poor performance. Engineers are often required to use CSV format to exchange data with systems and business processes outside their control. CSV is a common format for data archival. If you use CSV for archival, include a complete technical description of the serialization configuration for your files so that future consumers can ingest the data.

### XML

Extensible Markup Language (XML) was popular when HTML and the internet were new, but is now viewed as legacy; it is generally slow to deserialize and serialize for data engineering applications. XML is another standard that data engineers are often forced to interact with as they exchange data with legacy systems and software. JSON has largely replaced XML for plain-text object serialization.

### JSON and JSONL

JavaScript Object Notation (JSON) has emerged as the new standard for data exchange over APIs, and it has also become an extremely popular format for data storage. In the context of databases, the popularity of JSON has grown apace with the rise of MongoDB and other document stores. Databases such as Snowflake, BigQuery, and SQL Server also offer extensive native support, facilitating easy data exchange between applications, APIs, and database systems.

JSON Lines (JSONL) is a specialized version of JSON for storing bulk semistructured data in files. JSONL stores a sequence of JSON objects, with objects delimited by line breaks. From our perspective, JSONL is an extremely useful format for storing data right after it is ingested from API or applications. However, many columnar formats offer significantly better

performance. Consider moving to another format for intermediate pipeline stages and serving.

### Avro

Avro is a row-oriented data format designed for RPCs and data serialization. Avro encodes data into a binary format, with schema metadata specified in JSON. Avro is popular in the Hadoop ecosystem and is also supported by various cloud data tools.

## Columnar Serialization

The serialization formats we've discussed so far are row-oriented. Data is encoded as complete relations (CSV) or documents (XML and JSON), and these are written into files sequentially.

With *columnar serialization*, data organization is essentially pivoted by storing each column into its own set of files. One obvious advantage to columnar storage is that it allows us to read data from only a subset of fields rather than having to read full rows at once. This is a common scenario in analytics applications and can dramatically reduce the amount of data that must be scanned to execute a query.

Storing data as columns also puts similar values next to each other, allowing us to encode columnar data efficiently. One common technique involves looking for repeated values and tokenizing these, a simple but highly efficient compression method for columns with large numbers of repeats.

Even when columns don't contain large numbers of repeated values, they may manifest high redundancy. Suppose that we organized customer support messages into a single column of data. We likely see the same themes and verbiage again and again across these messages, allowing data compression algorithms to realize a high ratio. For this reason, columnar storage is usually combined with compression, allowing us to maximize disk and network bandwidth resources.

Columnar storage and compression come with some disadvantages too. We cannot easily access individual data records; we must reconstruct records by reading data from several column files. Record updates are also challenging. To change one field in one record, we must decompress the column file, modify it, recompress it, and write it back to storage. To avoid rewriting full columns on each update, columns are broken into many files, typically using partitioning and clustering strategies that organize data according to query and update patterns for the table. Even so, the overhead for updating a single row is horrendous. Columnar databases are a terrible fit for transactional workloads, so transactional databases generally utilize some form of row- or record-oriented storage.

### Parquet

Parquet stores data in a columnar format and is designed to realize excellent read and write performance in a data lake environment. Parquet solves a few problems that frequently bedevil data engineers. Parquet-encoded data builds in schema information and natively supports nested data, unlike CSV. Furthermore, Parquet is portable; while databases such as BigQuery and Snowflake serialize data in proprietary columnar formats and offer excellent query performance on data stored internally, a huge performance hit occurs when interoperating with external tools. Data must be deserialized, reserialized into an exchangeable format, and exported to use data lake tools such as Spark and Presto. Parquet files in a data lake may be a superior option to proprietary cloud data warehouses in a polyglot tool environment.

Parquet format is used with various compression algorithms; speed optimized compression algorithms such as Snappy (discussed later in this appendix) are especially popular.

### ORC

Optimized Row Columnar (ORC) is a columnar storage format similar to Parquet. ORC was very popular for use with Apache Hive; while still widely used, we generally see it much less than Apache Parquet, and it

enjoys somewhat less support in modern cloud ecosystem tools. For example, Snowflake and BigQuery support Parquet file import and export; while they can read from ORC files, neither tool can export to ORC.

### Apache Arrow or in-memory serialization

When we introduced serialization as a storage raw ingredient at the beginning of this chapter, we mentioned that software could store data in complex objects scattered in memory and connected by pointers, or more orderly, densely packed structures such as Fortran and C arrays. Generally, densely packed in-memory data structures were limited to simple types (e.g., INT64) or fixed-width data structures (e.g., fixed-width strings). More complex structures (e.g., JSON documents) could not be densely stored in memory and required serialization for storage and transfer between systems.

The idea of Apache Arrow is to rethink serialization by utilizing a binary data format that is suitable for both in-memory processing and export.[1] This allows us to avoid the overhead of serialization and deserialization; we simply use the same format for in-memory processing, export over the network, and long-term storage. Arrow relies on columnar storage, where each column essentially gets its own chunks of memory. For nested data, we use a technique called *shredding*, which maps each location in the schema of JSON documents into a separate column.

This technique means that we can store a data file on disk, swap it directly into program address space by using virtual memory, and begin running a query against the data without deserialization overhead. In fact, we can swap chunks of the file into memory as we scan it, and then swap them back out to avoid running out of memory for large datasets.

One obvious headache with this approach is that different programming languages serialize data in different ways. To address this issue, the Arrow Project has created software libraries for a variety of programming languages (including C, Go, Java, JavaScript, MATLAB, Python, R, and Rust) that allow these languages to interoperate with Arrow data in memory. In some cases, these libraries use an interface between the chosen

language and low-level code in another language (e.g., C) to read and write from Arrow. This allows high interoperability between languages without extra serialization overhead. For example, a Scala program can use the Java library to write arrow data and then pass it as a message to a Python program.

Arrow is seeing rapid uptake with a variety of popular frameworks such as Apache Spark. Arrow has also spanned a new data warehouse product; Dremio is a query engine and data warehouse built around Arrow serialization to support fast queries.

## Hybrid Serialization

We use the term *hybrid serialization* to refer to technologies that combine multiple serialization techniques or integrate serialization with additional abstraction layers, such as schema management. We cite as examples Apache Hudi and Apache Iceberg.

### Hudi

*Hudi* stands for *Hadoop Update Delete Incremental*. This table management technology combines multiple serialization techniques to allow columnar database performance for analytics queries while also supporting atomic, transactional updates. A typical Hudi application is a table that is updated from a CDC stream from a transactional application database. The stream is captured into a row-oriented serialization format, while the bulk of the table is retained in a columnar format. A query runs over both columnar and row-oriented files to return results for the current state of the table. Periodically, a repacking process runs that combines the row and columnar files into updated columnar files to maximize query efficiency.

### Iceberg

Like Hudi, Iceberg is a table management technology. Iceberg can track all files that make up a table. It can also track files in each table snapshot over

time, allowing table time travel in a data lake environment. Iceberg supports schema evolution and can readily manage tables at a petabyte scale.

## Database Storage Engines

To round out the discussion of serialization, we briefly discuss database storage engines. All databases have an underlying storage engine; many don't expose their storage engines as a separate abstraction (for example, BigQuery, Snowflake). Some (notably, MySQL) support fully pluggable storage engines. Others (e.g., SQL Server) offer major storage engine configuration options (columnar versus row-based storage) that dramatically affect database behavior.

Typically, the storage engine is a separate software layer from the query engine. The storage engine manages all aspects of how data is stored on a disk, including serialization, the physical arrangement of data, and indexes.

Storage engines have seen significant innovation in the 2000s and 2010s. While storage engines in the past were optimized for direct access to spinning disks, modern storage engines are much better optimized to support the performance characteristics of SSDs. Storage engines also offer improved support for modern types and data structures, such as variable-length strings, arrays, and nested data.

Another major change in storage engines is a shift toward columnar storage for analytics and data warehouse applications. SQL Server, PostgreSQL, and MySQL offer robust columnar storage support.

## Compression: gzip, bzip2, Snappy, etc.

The math behind compression algorithms is complex, but the basic idea is easy to understand: compression algorithms look for redundancy and repetition in data, then re-encode data to reduce redundancy. When we want to read the raw data, we *decompress* it by reversing the algorithm and putting the redundancy back in.

For example, you've noticed that certain words appear repeatedly in reading this book. Running some quick analytics on the text, you could identify the words that occur most frequently and create shortened tokens for these words. To compress, you would replace common words with their tokens; to decompress, you would replace the tokens with their respective words.

Perhaps we could use this naive technique to realize a compression ratio of 2:1 or more. Compression algorithms utilize more sophisticated mathematical techniques to identify and remove redundancy; they can often realize compression ratios of 10: 1 on text data.

Note that we're talking about *lossless compression algorithms*. Decompressing data encoded with a lossless algorithm recovers a bit-for-bit exact copy of the original data. *Lossy compression algorithms* for audio, images, and video aim for sensory fidelity; decompression recovers something that sounds like or looks like the original but is not an exact copy. Data engineers might deal with lossy compression algorithms in media processing pipelines but not in serialization for analytics, where exact data fidelity is required.

Traditional compression engines such as gzip and bzip2 compress text data extremely well; they are frequently applied to JSON, JSONL, XML, CSV, and other text-based data formats. Engineers have created a new generation of compression algorithms that prioritize speed over compression ratio in recent years. Major examples are Snappy, Zstandard, LZFSE, and LZ4. These algorithms are frequently used to compress data in data lakes or columnar databases to optimize fast query performance.

---

1 Dejan Simic, "Apache Arrow: Read DataFrame with Zero Memory," Towards Data Science, June 25, 2020, *https://oreil.ly/TDAdY*.

# Appendix B. Cloud Networking

This appendix discusses some factors data engineers should consider about networking in the cloud. Data engineers frequently encounter networking in their careers, and often ignore it despite its importance.

## Cloud Network Topology

A *cloud network topology* describes how various components in the cloud are arranged and connected, such as cloud services, networks, locations (zones, regions), and more. Data engineers should always know how cloud network topology will affect connectivity across the data systems they build. Microsoft Azure, Google Cloud Platform (GCP), and Amazon Web Services (AWS) all use remarkably similar resource hierarchies of availability zones and regions. At the time of this writing, GCP has added one additional layer, discussed in "GCP-Specific Networking and Multiregional Redundancy".

### Data Egress Charges

Chapter 4 discusses cloud economics and how actual provider costs don't necessarily drive cloud pricing. Regarding networking, clouds allow inbound traffic for free but charge for outbound traffic to the internet. Outbound traffic is not inherently cheaper, but clouds use this method to create a moat around their services and increase the stickiness of stored data, a practice that has been widely criticized.[1] Note that data egress charges can also apply to data passing between availability zones and regions within a cloud.

### Availability Zones

The *availability zone* is the smallest unit of network topology that public clouds make visible to customers (Figure B-1). While a zone can potentially consist of multiple data centers, cloud customers cannot control resource placement at this level.

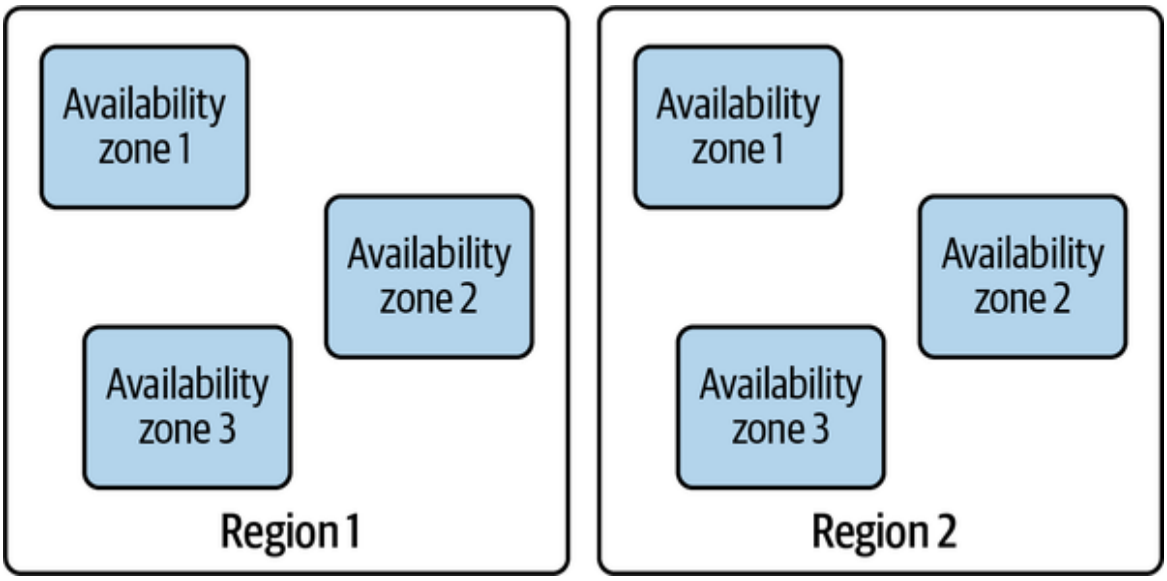


*Figure B-1. Availability zones in two separate regions*

Generally, clouds support their highest network bandwidth and lowest latency between systems and services within a zone. High throughput data workloads should run on clusters located in a single zone for performance and cost reasons. For example, an ephemeral Amazon EMR cluster should generally sit in a single availability zone.

In addition, network traffic sent to VMs within a zone is free, but with a significant caveat: traffic must be sent to private IP addresses. The major clouds utilize virtual networks known as *virtual private clouds* (VPCs). Virtual machines have private IP addresses within the VPC. They may also be assigned public IP addresses to communicate with the outside world and receive traffic from the internet, but communications using external IP addresses can incur data egress charges.

## Regions

A *region* is a collection of two or more availability zones. Data centers require many resources to run (electrical power, water, etc.). The resources of separate availability zones are independent so that a local power outage doesn't take down multiple availability zones. Engineers can build highly resilient, separate infrastructure even within a single region by running servers in multiple zones or creating automated cross-zone failover processes.

Offering multiple regions allows engineers to put resources close to any of their users. *Close* means that users can realize good network performance in connecting to services, minimizing physical distance along the network path, and a minimal number of hops through routers. Both physical distance and network hops can increase latency and decrease performance. Major cloud providers continue to add new regions.

In general, regions support fast, low-latency networking between zones; networking performance between zones will be worse than within a single zone and incur nominal data egress charges between VMs. Network data movement between regions is even slower and may incur higher egress fees.

In general, object storage is a regional resource. Some data may pass between zones to reach a virtual machine, but this is mainly invisible to cloud customers, and there are no direct networking charges for this. (Of course, customers are still responsible for object access costs.)

Despite regions' geo-redundant design, many major cloud service failures have affected entire regions, an example of *correlated failure*. Engineers often deploy code and configuration to entire regions; the regional failures we've observed have generally resulted from code or configuration problems occurring at the regional level.

## GCP-Specific Networking and Multiregional Redundancy

GCP offers a handful of unique abstractions that engineers should be aware of if they work in this cloud. The first is the *multiregion*, a layer in the resource hierarchy; a multiregion contains multiple regions. Current

multiregions are US (data centers in the United States), EU (data centers in European Union member states), and ASIA.

Several GCP resources support multiregions, including Cloud Storage and BigQuery. Data is stored in multiple zones within the multiregion in a geo-redundant manner so that it should remain available in the event of a regional failure. Multiregional storage is also designed to deliver data efficiently to users within the multiregion without setting up complex replication processes between regions. In addition, there are no data egress fees for VMs in a multiregion to access Cloud Storage data in the same multiregion.

Cloud customers can set up multiregional infrastructure on AWS or Azure. In the case of databases or object storage, this involves duplicating data between regions to increase redundancy and put data closer to users.

Google also essentially owns significantly more global scale networking resources than other cloud providers, something which it offers to its customers as *premium tier networking*. Premium tier networking allows traffic between zones and regions to pass entirely over Google-owned networks without traversing the public internet.

### Direct Network Connections to the Clouds

Each major public cloud offers enhanced connectivity options, allowing customers to integrate their networks with a cloud region or VPC directly. For example, Amazon offers AWS Direct Connect. In addition to providing higher bandwidth and lower latency, these connection options often offer dramatic discounts on data egress charges. In a typical scenario in the US, AWS egress charges drop from 9 cents per gigabyte over the public internet to 2 cents per gigabyte over direct connect.

## CDNs

*Content delivery networks* (CDNs) can offer dramatic performance enhancements and discounts for delivering data assets to the public or

customers. Cloud providers offer CDN options and many other providers, such as Cloudflare. CDNs work best when delivering the same data repeatedly, but make sure that you read the fine print. Remember that CDNs don't work everywhere, and certain countries may block internet traffic and CDN delivery.

## The Future of Data Egress Fees

Data egress fees are a significant impediment to interoperability, data sharing, and data movement to the cloud. Right now, data egress fees are a moat designed to prevent public cloud customers from leaving or deploying across multiple clouds.

But interesting signals indicate that change may be on the horizon. In particular, Zoom's announcement in 2020 near the beginning of the COVID-19 pandemic that it chose Oracle as its cloud infrastructure provider caught the attention of many cloud watchers.[2] How did Oracle win this significant cloud contract for critical remote work infrastructure against the cloud heavyweights? AWS expert Corey Quinn offers a reasonably straightforward answer.[3] By his back-of-the-envelope calculation, Zoom's AWS monthly data egress fees would run over $11 million dollars at list price; Oracle's would cost less than $2 million.

We suspect that one of GCP, AWS, or Azure will announce significant cuts in egress fees in the next few years, leading to a sea change in the cloud business model.

---

1 Matthew Prince and Nitin Rao, "AWS's Egregious Egress," *The Cloudflare Blog*, July 23, 2021, *https://oreil.ly/NZqKa*.

2 Mark Haranas and Steven Burke, "Oracle Bests Cloud Rivals to Win Blockbuster Cloud Deal," CRN, April 28, 2020, *https://oreil.ly/LkqOi*.

3 Corey Quinn, "Why Zoom Chose Oracle Cloud Over AWS and Maybe You Should Too," Last Week in AWS, April 28, 2020, *https://oreil.ly/Lx5uu*.

# Index

## About the Authors

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

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

## Colophon

The animal on the cover of *Fundamentals of Data Engineering* is the white-eared puffbird (*Nystalus chacuru*).

So named for the conspicuous patch of white at their ears, as well as for their fluffy plumage, these small, rotund birds are found across a wide swath of central South America, where they inhabit forest edges and savanna.

White-eared puffbirds are sit-and-wait hunters, perching in open spaces for long periods and feeding opportunistically on insects, lizards, and even small mammals that happen to come near. They are most often found alone or in pairs and are relatively quiet birds, vocalizing only rarely.

The International Union for Conservation of Nature has listed the white-eared puffbird as being of *least concern*, due, in part, to their extensive range and stable population. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from Shaw's *General Zoology*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.