Chapter 4. Choosing Technologies Across the Data Engineering Lifecycle

Data engineering nowadays suffers from an embarrassment of riches. We have no shortage of technologies to solve various types of data problems. Data technologies are available as turnkey offerings consumable in almost every way—open source, managed open source, proprietary software, proprietary service, and more. However, it's easy to get caught up in chasing bleeding-edge technology while losing sight of the core purpose of data engineering: designing robust and reliable systems to carry data through the full lifecycle and serve it according to the needs of end users. Just as structural engineers carefully choose technologies and materials to realize an architect's vision for a building, data engineers are tasked with making appropriate technology choices to shepherd data through the lifecycle to serve data applications and users.

Chapter 3 discussed "good" data architecture and why it matters. We now explain how to choose the right technologies to serve this architecture. Data engineers must choose good technologies to make the best possible data product. We feel the criteria to choose a good data technology is simple: does it add value to a data product and the broader business?

A lot of people confuse architecture and tools. Architecture is *strategic*; tools are *tactical*. We sometimes hear, "Our data architecture are tools *X*, *Y*, and *Z*." This is the wrong way to think about architecture. Architecture is the high-level design, roadmap, and blueprint of data systems that satisfy the strategic aims for the business. Architecture is the *what*, *why*, and *when*. Tools are used to make the architecture a reality; tools are the *how*.

We often see teams going "off the rails" and choosing technologies before mapping out an architecture. The reasons vary: shiny object syndrome, resume-driven development, and a lack of expertise in architecture. In practice, this prioritization of technology often means they cobble together a kind of Dr. Seuss fantasy machine rather than a true data architecture. We strongly advise against choosing technology before getting your architecture right. Architecture first, technology second.

This chapter discusses our tactical plan for making technology choices once we have a strategic architecture blueprint. The following are some considerations for choosing data technologies across the data engineering lifecycle:

- Team size and capabilities
- Speed to market
- Interoperability
- Cost optimization and business value
- Today versus the future: immutable versus transitory technologies
- Location (cloud, on prem, hybrid cloud, multicloud)
- Build versus buy
- Monolith versus modular
- Serverless versus servers
- Optimization, performance, and the benchmark wars
- The undercurrents of the data engineering lifecycle

Team Size and Capabilities

The first thing you need to assess is your team's size and its capabilities with technology. Are you on a small team (perhaps a team of one) of people who are expected to wear many hats, or is the team large enough that people work in specialized roles? Will a handful of people be responsible for multiple stages of the data engineering lifecycle, or do people cover particular niches? Your team's size will influence the types of technologies you adopt.

There is a continuum of simple to complex technologies, and a team's size roughly determines the amount of bandwidth your team can dedicate to complex solutions. We sometimes see small data teams read blog posts about a new cutting-edge technology at a giant tech company and then try to emulate these same extremely complex technologies and practices. We call this *cargo-cult engineering*, and it's generally a big mistake that consumes a lot of valuable time and money, often with little to nothing to show in return. Especially for small teams or teams with weaker technical chops, use as many managed and SaaS tools as possible, and dedicate your limited bandwidth to solving the complex problems that directly add value to the business.

Take an inventory of your team's skills. Do people lean toward low-code tools, or do they favor code-first approaches? Are people strong in certain languages like Java, Python, or Go? Technologies are available to cater to every preference on the low-code to code-heavy spectrum. Again, we suggest sticking with technologies and workflows with which the team is familiar. We've seen data teams invest a lot of time in learning the shiny new data technology, language, or tool, only to never use it in production. Learning new technologies, languages, and tools is a considerable time investment, so make these investments wisely.

Speed to Market

In technology, speed to market wins. This means choosing the right technologies that help you deliver features and data faster while maintaining high-quality standards and security. It also means working in a tight feedback loop of launching, learning, iterating, and making improvements.

Perfect is the enemy of good. Some data teams will deliberate on technology choices for months or years without reaching any decisions. Slow decisions and output are the kiss of death to data teams. We've seen more than a few data teams dissolve for moving too slow and failing to deliver the value they were hired to produce.

Deliver value early and often. As we've mentioned, use what works. Your team members will likely get better leverage with tools they already know. Avoid undifferentiated heavy lifting that engages your team in unnecessarily complex work that adds little to no value. Choose tools that help you move quickly, reliably, safely, and securely.

Interoperability

Rarely will you use only one technology or system. When choosing a technology or system, you'll need to ensure that it interacts and operates with other technologies. *Interoperability* describes how various technologies or systems connect, exchange information, and interact.

Let's say you're evaluating two technologies, A and B. How easily does technology A integrate with technology B when thinking about interoperability? This is often a spectrum of difficulty, ranging from seamless to time-intensive. Is seamless integration already baked into each product, making setup a breeze? Or do you need to do a lot of manual configuration to integrate these technologies?

Often, vendors and open source projects will target specific platforms and systems to interoperate. Most data ingestion and visualization tools have built-in integrations with popular data warehouses

and data lakes. Furthermore, popular data-ingestion tools will integrate with common APIs and services, such as CRMs, accounting software, and more.

Sometimes standards are in place for interoperability. Almost all databases allow connections via Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC), meaning that you can easily connect to a database by using these standards. In other cases, interoperability occurs in the absence of standards. Representational state transfer (REST) is not truly a standard for APIs; every REST API has its quirks. In these cases, it's up to the vendor or open source software (OSS) project to ensure smooth integration with other technologies and systems.

Always be aware of how simple it will be to connect your various technologies across the data engineering lifecycle. As mentioned in other chapters, we suggest designing for modularity and giving yourself the ability to easily swap out technologies as new practices and alternatives become available.

Cost Optimization and Business Value

In a perfect world, you'd get to experiment with all the latest, coolest technologies without considering cost, time investment, or value added to the business. In reality, budgets and time are finite, and the cost is a major constraint for choosing the right data architectures and technologies. Your organization expects a positive ROI from your data projects, so you must understand the basic costs you can control. Technology is a major cost driver, so your technology choices and management strategies will significantly impact your budget. We look at costs through three main lenses: total cost of ownership, opportunity cost, and FinOps.

Total Cost of Ownership

Total cost of ownership (TCO) is the total estimated cost of an initiative, including the direct and indirect costs of products and services utilized. *Direct costs* can be directly attributed to an initiative. Examples are the salaries of a team working on the initiative or the AWS bill for all services consumed. *Indirect costs*, also known as *overhead*, are independent of the initiative and must be paid regardless of where they're attributed.

Apart from direct and indirect costs, *how something is purchased* impacts the way costs are accounted for. Expenses fall into two big groups: capital expenses and operational expenses.

Capital expenses, also known as *capex*, require an up-front investment. Payment is required *today*. Before the cloud existed, companies would typically purchase hardware and software up front through large acquisition contracts. In addition, significant investments were required to host hardware in server rooms, data centers, and colocation facilities. These up-front investments—commonly hundreds of thousands to millions of dollars or more—would be treated as assets and slowly depreciate over time. From a budget perspective, capital was required to fund the entire purchase. This is capex, a significant capital outlay with a long-term plan to achieve a positive ROI on the effort and expense put forth.

Operational expenses, also known as *opex*, are the opposite of capex in certain respects. Opex is gradual and spread out over time. Whereas capex is long-term focused, opex is short-term. Opex can be pay-as-you-go or similar and allows a lot of flexibility. Opex is closer to a direct cost, making it easier to attribute to a data project.

Until recently, opex wasn't an option for large data projects. Data systems often required multimillion-dollar contracts. This has changed with the advent of the cloud, as data platform services allow engineers to pay on a consumption-based model. In general, opex allows for a far greater ability for engineering teams to choose their software and hardware. Cloud-based services let data engineers iterate quickly with various software and technology configurations, often inexpensively.

Data engineers need to be pragmatic about flexibility. The data landscape is changing too quickly to invest in long-term hardware that inevitably goes stale, can't easily scale, and potentially hampers a data engineer's flexibility to try new things. Given the upside for flexibility and low initial costs, we urge data engineers to take an opex-first approach centered on the cloud and flexible, pay-as-you-go technologies.

Total Opportunity Cost of Ownership

Any choice inherently excludes other possibilities. *Total opportunity cost of ownership* (TOCO) is the cost of lost opportunities that we incur in choosing a technology, an architecture, or a process.[1] Note that ownership in this setting doesn't require long-term purchases of hardware or licenses. Even in a cloud environment, we effectively own a technology, a stack, or a pipeline once it becomes a core part of our production data processes and is difficult to move away from. Data engineers often fail to evaluate TOCO when undertaking a new project; in our opinion, this is a massive blind spot.

If you choose data stack A, you've chosen the benefits of data stack A over all other options, effectively excluding data stacks B, C, and D. You're committed to data stack A and everything it entails—the team to support it, training, setup, and maintenance. What happens if data stack A was a poor choice? What happens when data stack A becomes obsolete? Can you still move to other data stacks?

How quickly and cheaply can you move to something newer and better? This is a critical question in the data space, where new technologies and products seem to appear at an ever-faster rate. Does the expertise you've built up on data stack A translate to the next wave? Or are you able to swap out components of data stack A and buy yourself some time and options?

The first step to minimizing opportunity cost is evaluating it with eyes wide open. We've seen countless data teams get stuck with technologies that seemed good at the time and are either not flexible for future growth or simply obsolete. Inflexible data technologies are a lot like bear traps. They're easy to get into and extremely painful to escape.

FinOps

We already touched on FinOps in "Principle 9: Embrace FinOps". As we've discussed, typical cloud spending is inherently opex: companies pay for services to run critical data processes rather than making up-front purchases and clawing back value over time. The goal of FinOps is to fully operationalize financial accountability and business value by applying the DevOps-like practices of monitoring and dynamically adjusting systems.

In this chapter, we want to emphasize one thing about FinOps that is well embodied in this quote:[2]

> If it seems that FinOps is about saving money, then think again. FinOps is about making money. Cloud spend can drive more revenue, signal customer base growth, enable more product and feature release velocity, or even help shut down a data center.

In our setting of data engineering, the ability to iterate quickly and scale dynamically is invaluable for creating business value. This is one of the major motivations for shifting data workloads to the cloud.

Today Versus the Future: Immutable Versus Transitory Technologies

In an exciting domain like data engineering, it's all too easy to focus on a rapidly evolving future while ignoring the concrete needs of the present. The intention to build a better future is noble but often leads to overarchitecting and overengineering. Tooling chosen for the future may be stale and

out-of-date when this future arrives; the future frequently looks little like what we envisioned years before.

As many life coaches would tell you, focus on the present. You should choose the best technology for the moment and near future, but in a way that supports future unknowns and evolution. Ask yourself: where are you today, and what are your goals for the future? Your answers to these questions should inform your decisions about your architecture and thus the technologies used within that architecture. This is done by understanding what is likely to change and what tends to stay the same.

We have two classes of tools to consider: immutable and transitory. *Immutable technologies* might be components that underpin the cloud or languages and paradigms that have stood the test of time. In the cloud, examples of immutable technologies are object storage, networking, servers, and security. Object storage such as Amazon S3 and Azure Blob Storage will be around from today until the end of the decade, and probably much longer. Storing your data in object storage is a wise choice. Object storage continues to improve in various ways and constantly offers new options, but your data will be safe and usable in object storage regardless of the rapid evolution of technology as a whole.

For languages, SQL and bash have been around for many decades, and we don't see them disappearing anytime soon. Immutable technologies benefit from the Lindy effect: the longer a technology has been established, the longer it will be used. Think of the power grid, relational databases, the C programming language, or the x86 processor architecture. We suggest applying the Lindy effect as a litmus test to determine whether a technology is potentially immutable.

Transitory technologies are those that come and go. The typical trajectory begins with a lot of hype, followed by meteoric growth in popularity, then a slow descent into obscurity. The JavaScript frontend landscape is a classic example. How many JavaScript frontend frameworks have come and gone between 2010 and 2020? Backbone.js, Ember.js, and Knockout were popular in the early 2010s, and React and Vue.js have massive mindshare today. What's the popular frontend JavaScript framework three years from now? Who knows.

New well-funded entrants and open source projects arrive on the data front every day. Every vendor will say their product will change the industry and "make the world a better place". Most of these companies and projects don't get long-term traction and fade slowly into obscurity. Top VCs are making big-money bets, knowing that most of their data-tooling investments will fail. If VCs pouring millions (or billions) into data-tooling investments can't get it right, how can you possibly know which technologies to invest in for your data architecture? It's hard. Just consider the number of technologies in Matt Turck's (in)famous depictions of the ML, AI, and data (MAD) landscape that we introduced in Chapter 1 (Figure 4-1).

Even relatively successful technologies often fade into obscurity quickly, after a few years of rapid adoption, a victim of their success. For instance, in the early 2010s, Hive was met with rapid uptake because it allowed both analysts and engineers to query massive datasets without coding complex MapReduce jobs by hand. Inspired by the success of Hive but wishing to improve on its shortcomings, engineers developed Presto and other technologies. Hive now appears primarily in legacy deployments. Almost every technology follows this inevitable path of decline.

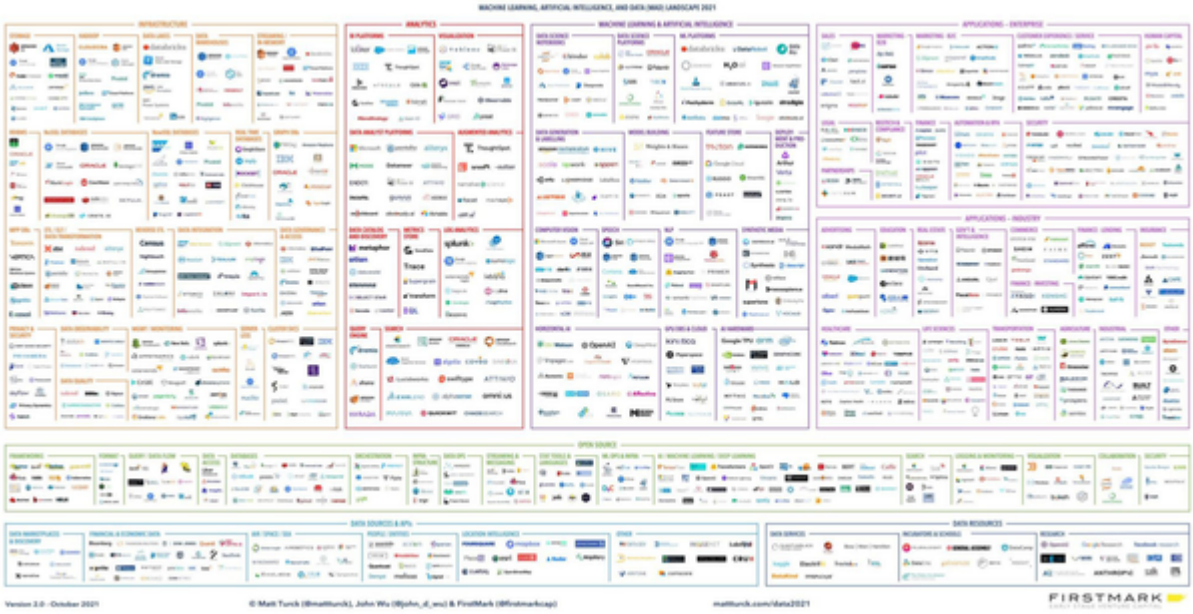

Figure 4-1. Matt Turck's 2021 MAD data landscape

Our Advice

Given the rapid pace of tooling and best-practice changes, we suggest evaluating tools every two years (Figure 4-2). Whenever possible, find the immutable technologies along the data engineering lifecycle, and use those as your base. Build transitory tools around the immutables.

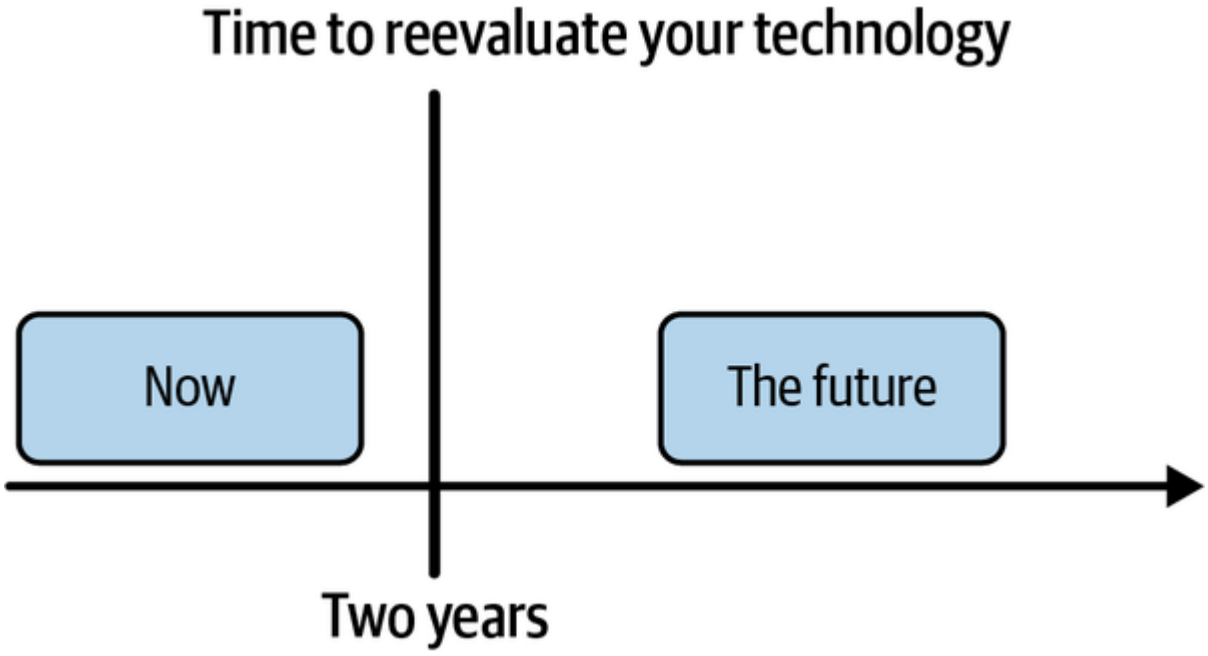

Figure 4-2. Use a two-year time horizon to reevaluate your technology choices

Given the reasonable probability of failure for many data technologies, you need to consider how easy it is to transition from a chosen technology. What are the barriers to leaving? As mentioned previously in our discussion about opportunity cost, avoid "bear traps." Go into a new technology with eyes wide open, knowing the project might get abandoned, the company may not be viable, or the technology simply isn't a good fit any longer.

Location

Companies now have numerous options when deciding where to run their technology stacks. A slow shift toward the cloud culminates in a veritable stampede of companies spinning up workloads on AWS, Azure, and Google Cloud Platform (GCP). In the last decade, many CTOs have come to view their decisions around technology hosting as having existential significance for their organizations. If they move too slowly, they risk being left behind by their more agile competition; on the other hand, a poorly planned cloud migration could lead to technological failure and catastrophic costs.

Let's look at the principal places to run your technology stack: on premises, the cloud, hybrid cloud, and multicloud.

On Premises

While new startups are increasingly born in the cloud, on-premises systems are still the default for established companies. Essentially, these companies own their hardware, which may live in data centers they own or in leased colocation space. In either case, companies are operationally responsible for their hardware and the software that runs on it. If hardware fails, they have to repair or replace it. They also have to manage upgrade cycles every few years as new, updated hardware is released and older hardware ages and becomes less reliable. They must ensure that they have enough hardware to handle peaks; for an online retailer, this means hosting enough capacity to handle the load spikes of Black Friday. For data engineers in charge of on-premises systems, this means buying large-enough systems to allow good performance for peak load and large jobs without overbuying and overspending.

On the one hand, established companies have established operational practices that have served them well. Suppose a company that relies on information technology has been in business for some time. This means it has managed to juggle the cost and personnel requirements of running its hardware, managing software environments, deploying code from dev teams, and running databases and big data systems.

On the other hand, established companies see their younger, more agile competition scaling rapidly and taking advantage of cloud-managed services. They also see established competitors making forays into the cloud, allowing them to temporarily scale up enormous computing power for massive data jobs or the Black Friday shopping spike.

Companies in competitive sectors generally don't have the option to stand still. Competition is fierce, and there's always the threat of being "disrupted" by more agile competition, often backed by a large pile of venture capital dollars. Every company must keep its existing systems running efficiently while deciding what moves to make next. This could involve adopting newer DevOps practices, such as containers, Kubernetes, microservices, and continuous deployment while keeping their hardware running on premises. It could involve a complete migration to the cloud, as discussed next.

Cloud

The cloud flips the on-premises model on its head. Instead of purchasing hardware, you simply rent hardware and managed services from a cloud provider (such as AWS, Azure, or Google Cloud). These resources can often be reserved on an extremely short-term basis; VMs spin up in less than a minute, and subsequent usage is billed in per-second increments. This allows cloud users to dynamically scale resources that were inconceivable with on-premises servers.

In a cloud environment, engineers can quickly launch projects and experiment without worrying about long lead time hardware planning. They can begin running servers as soon as their code is ready to deploy. This makes the cloud model extremely appealing to startups that are tight on budget and time.

The early cloud era was dominated by infrastructure as a service (IaaS) offerings—products such as VMs and virtual disks that are essentially rented slices of hardware. Slowly, we've seen a shift toward platform as a service (PaaS), while SaaS products continue to grow at a rapid clip.

PaaS includes IaaS products but adds more sophisticated managed services to support applications. Examples are managed databases such as Amazon Relational Database Service (RDS) and Google Cloud SQL, managed streaming platforms such as Amazon Kinesis and Simple Queue Service (SQS), and managed Kubernetes such as Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS). PaaS services allow engineers to ignore the operational details of managing individual machines and deploying frameworks across distributed systems. They provide turnkey access to complex, autoscaling systems with minimal operational overhead.

SaaS offerings move one additional step up the ladder of abstraction. SaaS typically provides a fully functioning enterprise software platform with little operational management. Examples of SaaS include Salesforce, Google Workspace, Microsoft 365, Zoom, and Fivetran. Both the major public clouds and third parties offer SaaS platforms. SaaS covers a whole spectrum of enterprise domains, including video conferencing, data management, ad tech, office applications, and CRM systems.

This chapter also discusses serverless, increasingly important in PaaS and SaaS offerings. Serverless products generally offer automated scaling from zero to extremely high usage rates. They are billed on a pay-as-you-go basis and allow engineers to operate without operational awareness of underlying servers. Many people quibble with the term *serverless*; after all, the code must run somewhere. In practice, serverless usually means *many invisible servers*.

Cloud services have become increasingly appealing to established businesses with existing data centers and IT infrastructure. Dynamic, seamless scaling is extremely valuable to businesses that deal with seasonality (e.g., retail businesses coping with Black Friday load) and web traffic load spikes. The advent of COVID-19 in 2020 was a major driver of cloud adoption, as companies recognized the value of rapidly scaling up data processes to gain insights in a highly uncertain business climate; businesses also had to cope with substantially increased load due to a spike in online shopping, web app usage, and remote work.

Before we discuss the nuances of choosing technologies in the cloud, let's first discuss why migration to the cloud requires a dramatic shift in thinking, specifically on the pricing front; this is closely related to FinOps, introduced in "FinOps". Enterprises that migrate to the cloud often make major deployment errors by not appropriately adapting their practices to the cloud pricing model.

A Brief Detour on Cloud Economics

To understand how to use cloud services efficiently through cloud native architecture, you need to know how clouds make money. This is an extremely complex concept and one on which cloud providers offer little transparency. Consider this sidebar a starting point for your research, discovery, and process development.

Cloud Services and Credit Default Swaps

Let's go on a little tangent about credit default swaps. Don't worry, this will make sense in a bit. Recall that credit default swaps rose to infamy after the 2007 global financial crisis. A credit default swap was a mechanism for selling different tiers of risk attached to an asset (e.g., a mortgage). It is not our intention to present this idea in any detail, but rather to offer an analogy wherein many cloud services are similar to financial derivatives; cloud providers not only slice hardware assets into small pieces through virtualization, but also sell these pieces with varying technical characteristics and risks attached. While providers are extremely tight-lipped about details of their internal systems, there are massive opportunities for optimization and scaling by understanding cloud pricing and exchanging notes with other users.

Look at the example of archival cloud storage. At the time of this writing, GCP openly admits that its archival class storage runs on the same clusters as standard cloud storage, yet the price per gigabyte

per month of archival storage is roughly 1/17 that of standard storage. How is this possible?

Here's our educated guess. When purchasing cloud storage, each disk in a storage cluster has three assets that cloud providers and consumers use. First, it has a certain storage capacity—say, 10 TB. Second, it supports a certain number of input/output operations (IOPs) per second—say, 100. Third, disks support a certain maximum bandwidth, the maximum read speed for optimally organized files. A magnetic drive might be capable of reading at 200 MB/s.

Any of these limits (IOPs, storage capacity, bandwidth) is a potential bottleneck for a cloud provider. For instance, the cloud provider might have a disk storing 3 TB of data but hitting maximum IOPs. An alternative to leaving the remaining 7 TB empty is to sell the empty space without selling IOPs. Or, more specifically, sell cheap storage space and expensive IOPs to discourage reads.

Much like traders of financial derivatives, cloud vendors also deal in risk. In the case of archival storage, vendors are selling a type of insurance, but one that pays out for the insurer rather than the policy buyer in the event of a catastrophe. While data storage costs per month are extremely cheap, I risk paying a high price if I ever need to retrieve data. But this is a price that I will happily pay in a true emergency.

Similar considerations apply to nearly any cloud service. While on-premises servers are essentially sold as commodity hardware, the cost model in the cloud is more subtle. Rather than just charging for CPU cores, memory, and features, cloud vendors monetize characteristics such as durability, reliability, longevity, and predictability; a variety of compute platforms discount their offerings for workloads that are ephemeral or can be arbitrarily interrupted when capacity is needed elsewhere.

Cloud ≠ On Premises

This heading may seem like a silly tautology, but the belief that cloud services are just like familiar on-premises servers is a widespread cognitive error that plagues cloud migrations and leads to horrifying bills. This demonstrates a broader issue in tech that we refer to as *the curse of familiarity*. Many new technology products are intentionally designed to look like something familiar to facilitate ease of use and accelerate adoption. But, any new technology product has subtleties and wrinkles that users must learn to identify, accommodate, and optimize.

Moving on-premises servers one by one to VMs in the cloud—known as simple *lift and shift*—is a perfectly reasonable strategy for the initial phase of cloud migration, especially when a company is facing some kind of financial cliff, such as the need to sign a significant new lease or hardware contract if existing hardware is not shut down. However, companies that leave their cloud assets in this initial state are in for a rude shock. On a direct comparison basis, long-running servers in the cloud are significantly more expensive than their on-premises counterparts.

The key to finding value in the cloud is understanding and optimizing the cloud pricing model. Rather than deploying a set of long-running servers capable of handling full peak load, use autoscaling to allow workloads to scale down to minimal infrastructure when loads are light and up to massive clusters during peak times. To realize discounts through more ephemeral, less durable workloads, use reserved or spot instances, or use serverless functions in place of servers.

We often think of this optimization as leading to lower costs, but we should also strive to *increase business value* by exploiting the dynamic nature of the cloud.[3] Data engineers can create new value in the cloud by accomplishing things that were impossible in their on-premises environment. For example, it is possible to quickly spin up massive compute clusters to run complex transformations at scales that were unaffordable for on-premises hardware.

Data Gravity

In addition to basic errors such as following on-premises operational practices in the cloud, data engineers need to watch out for other aspects of cloud pricing and incentives that frequently catch users unawares.

Vendors want to lock you into their offerings. Getting data onto the platform is cheap or free on most cloud platforms, but getting data out can be extremely expensive. Be aware of data egress fees and their long-term impacts on your business before getting blindsided by a large bill. *Data gravity* is real: once data lands in a cloud, the cost to extract it and migrate processes can be very high.

Hybrid Cloud

As more established businesses migrate into the cloud, the hybrid cloud model is growing in importance. Virtually no business can migrate all of its workloads overnight. The hybrid cloud model assumes that an organization will indefinitely maintain some workloads outside the cloud.

There are many reasons to consider a hybrid cloud model. Organizations may believe that they have achieved operational excellence in certain areas, such as their application stack and associated hardware. Thus, they may migrate only to specific workloads where they see immediate benefits in the cloud environment. For example, an on-premises Spark stack is migrated to ephemeral cloud clusters, reducing the operational burden of managing software and hardware for the data engineering team and allowing rapid scaling for large data jobs.

This pattern of putting analytics in the cloud is beautiful because data flows primarily in one direction, minimizing data egress costs (Figure 4-3). That is, on-premises applications generate event data that can be pushed to the cloud essentially for free. The bulk of data remains in the cloud where it is analyzed, while smaller amounts of data are pushed back to on premises for deploying models to applications, reverse ETL, etc.

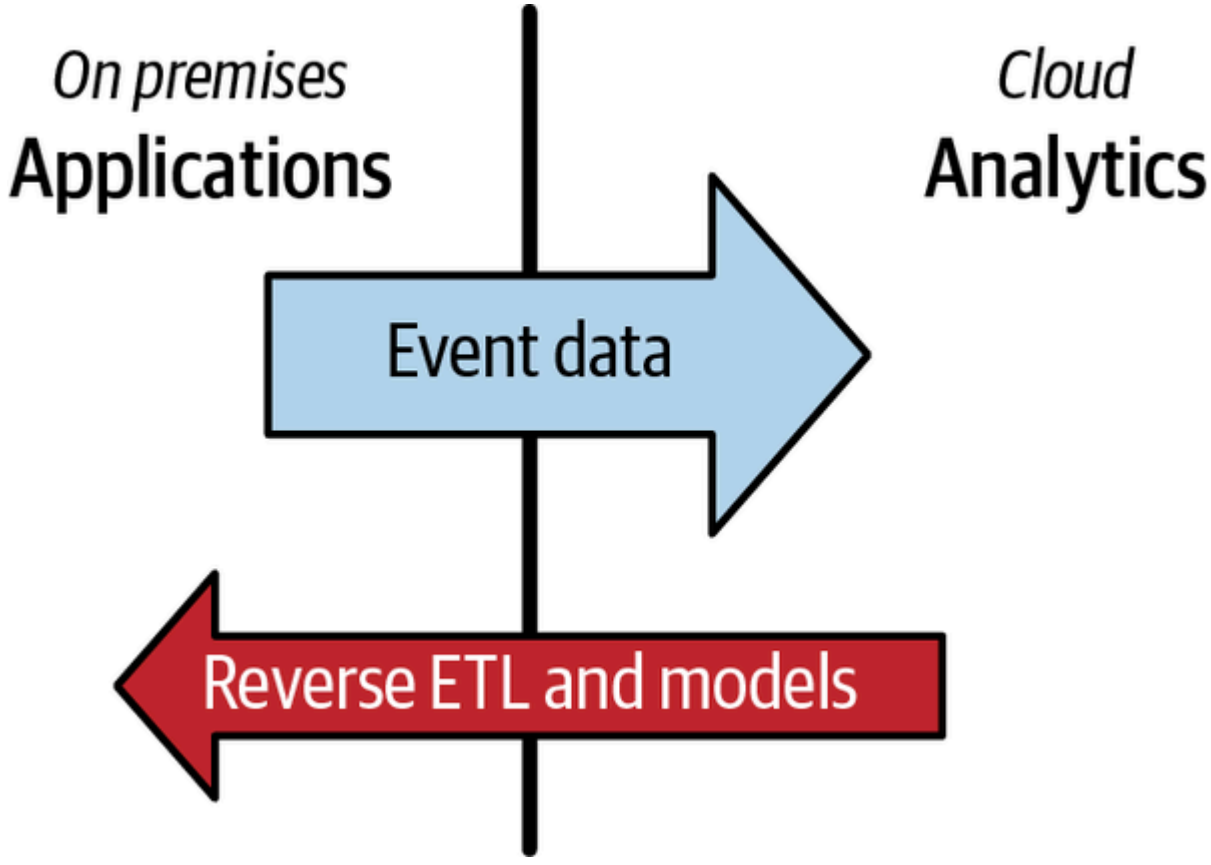

Figure 4-3. A hybrid cloud data flow model minimizing egress costs

A new generation of managed hybrid cloud service offerings also allows customers to locate cloud-managed servers in their data centers.[4] This gives users the ability to incorporate the best features in each cloud alongside on-premises infrastructure.

Multicloud

Multicloud simply refers to deploying workloads to multiple public clouds. Companies may have several motivations for multicloud deployments. SaaS platforms often wish to offer services close to existing customer cloud workloads. Snowflake and Databricks provide their SaaS offerings across multiple clouds for this reason. This is especially critical for data-intensive applications, where network latency and bandwidth limitations hamper performance, and data egress costs can be prohibitive.

Another common motivation for employing a multicloud approach is to take advantage of the best services across several clouds. For example, a company might want to handle its Google Ads and Analytics data on Google Cloud and deploy Kubernetes through GKE. And the company might also adopt Azure specifically for Microsoft workloads. Also, the company may like AWS because it has several best-in-class services (e.g., AWS Lambda) and enjoys huge mindshare, making it relatively easy to hire AWS-proficient engineers. Any mix of various cloud provider services is possible. Given the intense competition among the major cloud providers, expect them to offer more best-of-breed services, making multicloud more compelling.

A multicloud methodology has several disadvantages. As we just mentioned, data egress costs and networking bottlenecks are critical. Going multicloud can introduce significant complexity. Companies must now manage a dizzying array of services across several clouds; cross-cloud integration and security present a considerable challenge; multicloud networking can be diabolically complicated.

A new generation of “cloud of clouds” services aims to facilitate multicloud with reduced complexity by offering services across clouds and seamlessly replicating data between clouds or managing workloads on several clouds through a single pane of glass. To cite one example, a Snowflake account runs in a single cloud region, but customers can readily spin up other accounts in GCP, AWS, or Azure. Snowflake provides simple scheduled data replication between these various cloud accounts. The Snowflake interface is essentially the same in all of these accounts, removing the training burden of switching between cloud-native data services.

The “cloud of clouds” space is evolving quickly; within a few years of this book’s publication, many more of these services will be available. Data engineers and architects would do well to maintain awareness of this quickly changing cloud landscape.

Decentralized: Blockchain and the Edge

Though not widely used now, it’s worth briefly mentioning a new trend that might become popular over the next decade: decentralized computing. Whereas today’s applications mainly run on premises and in the cloud, the rise of blockchain, Web 3.0, and edge computing may invert this paradigm. For the moment, decentralized platforms have proven extremely popular but have not had a significant impact in the data space; even so, keeping an eye on these platforms is worthwhile as you assess technology decisions.

Our Advice

From our perspective, we are still at the beginning of the transition to the cloud. Thus the evidence and arguments around workload placement and migration are in flux. The cloud itself is changing, with a shift from the IaaS model built around Amazon EC2 that drove the early growth of AWS and more generally toward more managed service offerings such as AWS Glue, Google BigQuery, and Snowflake.

We’ve also seen the emergence of new workload placement abstractions. On-premises services are becoming more cloud-like and abstracted. Hybrid cloud services allow customers to run fully managed services within their walls while facilitating tight integration between local and remote

environments. Further, the "cloud of clouds" is beginning to take shape, fueled by third-party services and public cloud vendors.

Choose technologies for the present, but look toward the future

As we mentioned in "Today Versus the Future: Immutable Versus Transitory Technologies", you need to keep one eye on the present while planning for unknowns. Right now is a tough time to plan workload placements and migrations. Because of the fast pace of competition and change in the cloud industry, the decision space will look very different in five to ten years. It is tempting to take into account every possible future architecture permutation.

We believe that it is critical to avoid this endless trap of analysis. Instead, plan for the present. Choose the best technologies for your current needs and concrete plans for the near future. Choose your deployment platform based on real business needs while focusing on simplicity and flexibility.

In particular, don't choose a complex multicloud or hybrid-cloud strategy unless there's a compelling reason. Do you need to serve data near customers on multiple clouds? Do industry regulations require you to house certain data in your data centers? Do you have a compelling technology need for specific services on two different clouds? Choose a single-cloud deployment strategy if these scenarios don't apply to you.

On the other hand, have an escape plan. As we've emphasized before, every technology—even open source software—comes with some degree of lock-in. A single-cloud strategy has significant advantages of simplicity and integration but comes with significant lock-in attached. In this instance, we're talking about mental flexibility, the flexibility to evaluate the current state of the world and imagine alternatives. Ideally, your escape plan will remain locked behind glass, but preparing this plan will help you to make better decisions in the present and give you a way out if things go wrong in the future.

Cloud Repatriation Arguments

As we wrote this book, Sarah Wang and Martin Casado published "The Cost of Cloud, A Trillion Dollar Paradox", an article that generated significant sound and fury in the tech space. Readers widely interpreted the article as a call for the repatriation of cloud workloads to on-premises servers. They make a somewhat more subtle argument that companies should expend significant resources to control cloud spending and should consider repatriation as a possible option.

We want to take a moment to dissect one part of their discussion. Wang and Casado cite Dropbox's repatriation of significant workloads from AWS to Dropbox-owned servers as a case study for companies considering similar repatriation moves.

You are not Dropbox, nor are you Cloudflare

We believe that this case study is frequently used without appropriate context and is a compelling example of the *false equivalence* logical fallacy. Dropbox provides particular services where ownership of hardware and data centers can offer a competitive advantage. Companies should not rely excessively on Dropbox's example when assessing cloud and on-premises deployment options.

First, it's important to understand that Dropbox stores enormous amounts of data. The company is tight-lipped about exactly how much data it hosts but says it is many exabytes and continues to grow.

Second, Dropbox handles a vast amount of network traffic. We know that its bandwidth consumption in 2017 was significant enough for the company to add "hundreds of gigabits of internet connectivity with transit providers (regional and global ISPs), and hundreds of new peering partners (where we exchange traffic directly rather than through an ISP)."[5] The data egress costs would be extremely high in a public cloud environment.

Third, Dropbox is essentially a cloud storage vendor, but one with a highly specialized storage product that combines object and block storage characteristics. Dropbox's core competence is a differential file-update system that can efficiently synchronize actively edited files among users while minimizing network and CPU usage. The product is not a good fit for object storage, block storage, or other standard cloud offerings. Dropbox has instead benefited from building a custom, highly integrated software and hardware stack.[6]

Fourth, while Dropbox moved its core product to its hardware, it continued building out other AWS workloads. This allows Dropbox to focus on building one highly tuned cloud service at an extraordinary scale rather than trying to replace multiple services. Dropbox can focus on its core competence in cloud storage and data synchronization while offloading software and hardware management in other areas, such as data analytics.[7]

Other frequently cited success stories that companies have built outside the cloud include Backblaze and Cloudflare, but these offer similar lessons. Backblaze began life as a personal cloud data backup product but has since begun to offer B2, an object storage service similar to Amazon S3. Backblaze currently stores over an exabyte of data. Cloudflare claims to provide services for over 25 million internet properties, with points of presence in over 200 cities and 51 terabits per second (Tbps) of total network capacity.

Netflix offers yet another useful example. The company is famous for running its tech stack on AWS, but this is only partially true. Netflix does run video transcoding on AWS, accounting for roughly 70% of its compute needs in 2017.[8] Netflix also runs its application backend and data analytics on AWS. However, rather than using the AWS content distribution network, Netflix has built a custom CDN in collaboration with internet service providers, utilizing a highly specialized combination of software and hardware. For a company that consumes a substantial slice of all internet traffic,[9] building out this critical infrastructure allowed it to deliver high-quality video to a huge customer base cost-effectively.

These case studies suggest that it makes sense for companies to manage their own hardware and network connections in particular circumstances. The biggest modern success stories of companies building and maintaining hardware involve extraordinary scale (exabytes of data, terabits per second of bandwidth, etc.) and limited use cases where companies can realize a competitive advantage by engineering highly integrated hardware and software stacks. In addition, all of these companies consume massive network bandwidth, suggesting that data egress charges would be a major cost if they chose to operate fully from a public cloud.

Consider continuing to run workloads on premises or repatriating cloud workloads if you run a truly cloud-scale service. What is cloud scale? You might be at cloud scale if you are storing an exabyte of data or handling terabits per second of traffic *to and from the internet*. (Achieving a terabit per second of *internal* network traffic is fairly easy.) In addition, consider owning your servers if data egress costs are a major factor for your business. To give a concrete example of cloud scale workloads that could benefit from repatriation, Apple might gain a significant financial and performance advantage by migrating iCloud storage to its own servers.[10]

Build Versus Buy

Build versus buy is an age-old debate in technology. The argument for building is that you have end-to-end control over the solution and are not at the mercy of a vendor or open source community. The argument supporting buying comes down to resource constraints and expertise; do you have the expertise to build a better solution than something already available? Either decision comes down to TCO, TOCO, and whether the solution provides a competitive advantage to your organization.

If you've caught on to a theme in the book so far, it's that we suggest investing in building and customizing *when doing so will provide a competitive advantage* for your business. Otherwise, stand on the shoulders of giants and *use what's already available* in the market. Given the number of open

source and paid services—both of which may have communities of volunteers or highly paid teams of amazing engineers—you're foolish to build everything yourself.

As we often ask, "When you need new tires for your car, do you get the raw materials, create the tires from scratch, and install them yourself?" Like most people, you're probably buying tires and having someone install them. The same argument applies to build versus buy. We've seen teams that have built their databases from scratch. A simple open source RDBMS would have served their needs much better upon closer inspection. Imagine the amount of time and money invested in this homegrown database. Talk about low ROI for TCO and opportunity cost.

This is where the distinction between the type A and type B data engineer comes in handy. As we pointed out earlier, type A and type B roles are often embodied in the same engineer, especially in a small organization. Whenever possible, lean toward type A behavior; avoid undifferentiated heavy lifting and embrace abstraction. Use open source frameworks, or if this is too much trouble, look at buying a suitable managed or proprietary solution. Plenty of great modular services are available to choose from in either case.

The shifting reality of how companies adopt software is worth mentioning. Whereas in the past, IT used to make most of the software purchase and adoption decisions in a top-down manner, these days, the trend is for bottom-up software adoption in a company, driven by developers, data engineers, data scientists, and other technical roles. Technology adoption within companies is becoming an organic, continuous process.

Let's look at some options for open source and proprietary solutions.

Open Source Software

Open source software (OSS) is a software distribution model in which software, and the underlying codebase, is made available for general use, typically under specific licensing terms. Often OSS is created and maintained by a distributed team of collaborators. OSS is free to use, change, and distribute most of the time, but with specific caveats. For example, many licenses require that the source code of open source–derived software be included when the software is distributed.

The motivations for creating and maintaining OSS vary. Sometimes OSS is organic, springing from the mind of an individual or a small team that creates a novel solution and chooses to release it into the wild for public use. Other times, a company may make a specific tool or technology available to the public under an OSS license.

OSS has two main flavors: community managed and commercial OSS.

Community-managed OSS

OSS projects succeed with a strong community and vibrant user base. *Community-managed OSS* is a prevalent path for OSS projects. The community opens up high rates of innovations and contributions from developers worldwide with popular OSS projects.

The following are factors to consider with a community-managed OSS project:

Mindshare

> Avoid adopting OSS projects that don't have traction and popularity. Look at the number of GitHub stars, forks, and commit volume and recency. Another thing to pay attention to is community activity on related chat groups and forums. Does the project have a strong sense of community? A strong community creates a virtuous cycle of strong adoption. It also means that you'll have an easier time getting technical assistance and finding talent qualified to work with the framework.

Maturity

How long has the project been around, how active is it today, and how usable are people finding it in production? A project's maturity indicates that people find it useful and are willing to incorporate it into their production workflows.

Troubleshooting

How will you have to handle problems if they arise? Are you on your own to troubleshoot issues, or can the community help you solve your problem?

Project management

Look at Git issues and the way they're addressed. Are they addressed quickly? If so, what's the process to submit an issue and get it resolved?

Team

Is a company sponsoring the OSS project? Who are the core contributors?

Developer relations and community management

What is the project doing to encourage uptake and adoption? Is there a vibrant chat community (e.g., in Slack) that provides encouragement and support?

Contributing

Does the project encourage and accept pull requests? What are the process and timelines for pull requests to be accepted and included in main codebase?

Roadmap

Is there a project roadmap? If so, is it clear and transparent?

Self-hosting and maintenance

Do you have the resources to host and maintain the OSS solution? If so, what's the TCO and TOCO versus buying a managed service from the OSS vendor?

Giving back to the community

If you like the project and are actively using it, consider investing in it. You can contribute to the codebase, help fix issues, and give advice in the community forums and chats. If the project allows donations, consider making one. Many OSS projects are essentially community-service projects, and the maintainers often have full-time jobs in addition to helping with the OSS project. Sadly, it's often a labor of love that doesn't afford the maintainer a living wage. If you can afford to donate, please do so.

Commercial OSS

Sometimes OSS has some drawbacks. Namely, you have to host and maintain the solution in your environment. This may be trivial or extremely complicated and cumbersome, depending on the OSS application. Commercial vendors try to solve this management headache by hosting and managing the OSS solution for you, typically as a cloud SaaS offering. Examples of such vendors include Databricks (Spark), Confluent (Kafka), DBT Labs (dbt), and there are many, many others.

This model is called *commercial OSS* (COSS). Typically, a vendor will offer the "core" of the OSS for free while charging for enhancements, curated code distributions, or fully managed services.

A vendor is often affiliated with the community OSS project. As an OSS project becomes more popular, the maintainers may create a separate business for a managed version of the OSS. This

typically becomes a cloud SaaS platform built around a managed version of the open source code. This is a widespread trend: an OSS project becomes popular, an affiliated company raises truckloads of venture capital (VC) money to commercialize the OSS project, and the company scales as a fast-moving rocket ship.

At this point, the data engineer has two options. You can continue using the community-managed OSS version, which you need to continue maintaining on your own (updates, server/container maintenance, pull requests for bug fixes, etc.). Or, you can pay the vendor and let it take care of the administrative management of the COSS product.

The following are factors to consider with a commercial OSS project:

Value

> Is the vendor offering a better value than if you managed the OSS technology yourself? Some vendors will add many bells and whistles to their managed offerings that aren't available in the community OSS version. Are these additions compelling to you?

Delivery model

> How do you access the service? Is the product available via download, API, or web/mobile UI? Be sure you can easily access the initial version and subsequent releases.

Support

> Support cannot be understated, and it's often opaque to the buyer. What is the support model for the product, and is there an extra cost for support? Frequently, vendors will sell support for an additional fee. Be sure you clearly understand the costs of obtaining support. Also, understand what is covered in support, and what is not covered. Anything that's not covered by support will be your responsibility to own and manage.

Releases and bug fixes

> Is the vendor transparent about the release schedule, improvements, and bug fixes? Are these updates easily available to you?

Sales cycle and pricing

> Often a vendor will offer on-demand pricing, especially for a SaaS product, and offer you a discount if you commit to an extended agreement. Be sure to understand the trade-offs of paying as you go versus paying up front. Is it worth paying a lump sum, or is your money better spent elsewhere?

Company finances

> Is the company viable? If the company has raised VC funds, you can check their funding on sites like Crunchbase. How much runway does the company have, and will it still be in business in a couple of years?

Logos versus revenue

> Is the company focused on growing the number of customers (logos), or is it trying to grow revenue? You may be surprised by the number of companies primarily concerned with growing their customer count, GitHub stars, or Slack channel membership without the revenue to establish sound finances.

Community support

> Is the company truly supporting the community version of the OSS project? How much is the company contributing to the community OSS codebase? Controversies have arisen with certain

vendors co-opting OSS projects and subsequently providing little value back to the community. How likely will the product remain viable as a community-supported open source if the company shuts down?

Note also that clouds offer their own managed open source products. If a cloud vendor sees traction with a particular product or project, expect that vendor to offer its version. This can range from simple examples (open source Linux offered on VMs) to extremely complex managed services (fully managed Kafka). The motivation for these offerings is simple: clouds make their money through consumption. More offerings in a cloud ecosystem mean a greater chance of "stickiness" and increased customer spending.

Proprietary Walled Gardens

While OSS is ubiquitous, a big market also exists for non-OSS technologies. Some of the biggest companies in the data industry sell closed source products. Let's look at two major types of *proprietary walled gardens*, independent companies and cloud-platform offerings.

Independent offerings

The data-tool landscape has seen exponential growth over the last several years. Every day, new independent offerings arise for data tools. With the ability to raise funds from VCs flush with capital, these data companies can scale and hire great engineering, sales, and marketing teams. This presents a situation where users have some great product choices in the marketplace while having to wade through endless sales and marketing clutter. At the time of this writing, the good times of freely available capital for data companies are coming to an end, but that's another long story whose consequences are still unfolding.

Often a company selling a data tool will not release it as OSS, instead offering a proprietary solution. Although you won't have the transparency of a pure OSS solution, a proprietary independent solution can work quite well, especially as a fully managed service in the cloud.

The following are things to consider with an independent offering:

Interoperability

Make sure that the tool interoperates with other tools you've chosen (OSS, other independents, cloud offerings, etc.). Interoperability is key, so make sure you can try it before you buy.

Mindshare and market share

Is the solution popular? Does it command a presence in the marketplace? Does it enjoy positive customer reviews?

Documentation and support

Problems and questions will inevitably arise. Is it clear how to solve your problem, either through documentation or support?

Pricing

Is the pricing understandable? Map out low-, medium-, and high-probability usage scenarios, with respective costs. Are you able to negotiate a contract, along with a discount? Is it worth it? How much flexibility do you lose if you sign a contract, both in negotiation and the ability to try new options? Are you able to obtain contractual commitments on future pricing?

Longevity

Will the company survive long enough for you to get value from its product? If the company has raised money, search around for its funding situation. Look at user reviews. Ask friends and post questions on social networks about users' experiences with the product. Make sure you know what you're getting into.

Cloud platform proprietary service offerings

Cloud vendors develop and sell their proprietary services for storage, databases, and more. Many of these solutions are internal tools used by respective sibling companies. For example, Amazon created the database DynamoDB to overcome the limitations of traditional relational databases and handle the large amounts of user and order data as Amazon.com grew into a behemoth. Amazon later offered the DynamoDB service solely on AWS; it's now a top-rated product used by companies of all sizes and maturity levels. Cloud vendors will often bundle their products to work well together. Each cloud can create stickiness with its user base by creating a strong integrated ecosystem.

The following are factors to consider with a proprietary cloud offering:

Performance versus price comparisons

Is the cloud offering substantially better than an independent or OSS version? What's the TCO of choosing a cloud's offering?

Purchase considerations

On-demand pricing can be expensive. Can you lower your cost by purchasing reserved capacity or entering into a long-term commitment agreement?

Our Advice

Build versus buy comes back to knowing your competitive advantage and where it makes sense to invest resources toward customization. In general, we favor OSS and COSS by default, which frees you to focus on improving those areas where these options are insufficient. Focus on a few areas where building something will add significant value or reduce friction substantially.

Don't treat internal operational overhead as a sunk cost. There's excellent value in upskilling your existing data team to build sophisticated systems on managed platforms rather than babysitting on-premises servers. In addition, think about how a company makes money, especially its sales and customer experience teams, which will generally indicate how you're treated during the sales cycle and when you're a paying customer.

Finally, who is responsible for the budget at your company? How does this person decide the projects and technologies that get funded? Before making the business case for COSS or managed services, does it make sense to try to use OSS first? The last thing you want is for your technology choice to be stuck in limbo while waiting for budget approval. As the old saying goes, *time kills deals*. In your case, more time spent in limbo means a higher likelihood your budget approval will die. Know beforehand who controls the budget and what will successfully get approved.

Monolith Versus Modular

Monoliths versus modular systems is another longtime debate in the software architecture space. Monolithic systems are self-contained, often performing multiple functions under a single system. The monolith camp favors the simplicity of having everything in one place. It's easier to reason about a single entity, and you can move faster because there are fewer moving parts. The *modular* camp leans toward decoupled, best-of-breed technologies performing tasks at which they are uniquely great. Especially given the rate of change in products in the data world, the argument is you should aim for interoperability among an ever-changing array of solutions.

What approach should you take in your data engineering stack? Let's explore the trade-offs.

Monolith

The *monolith* (Figure 4-4) has been a technology mainstay for decades. The old days of waterfall meant that software releases were huge, tightly coupled, and moved at a slow cadence. Large teams worked together to deliver a single working codebase. Monolithic data systems continue to this day, with older software vendors such as Informatica and open source frameworks such as Spark.

The pros of the monolith are it's easy to reason about, and it requires a lower cognitive burden and context switching since everything is self-contained. Instead of dealing with dozens of technologies, you deal with "one" technology and typically one principal programming language. Monoliths are an excellent option if you want simplicity in reasoning about your architecture and processes.

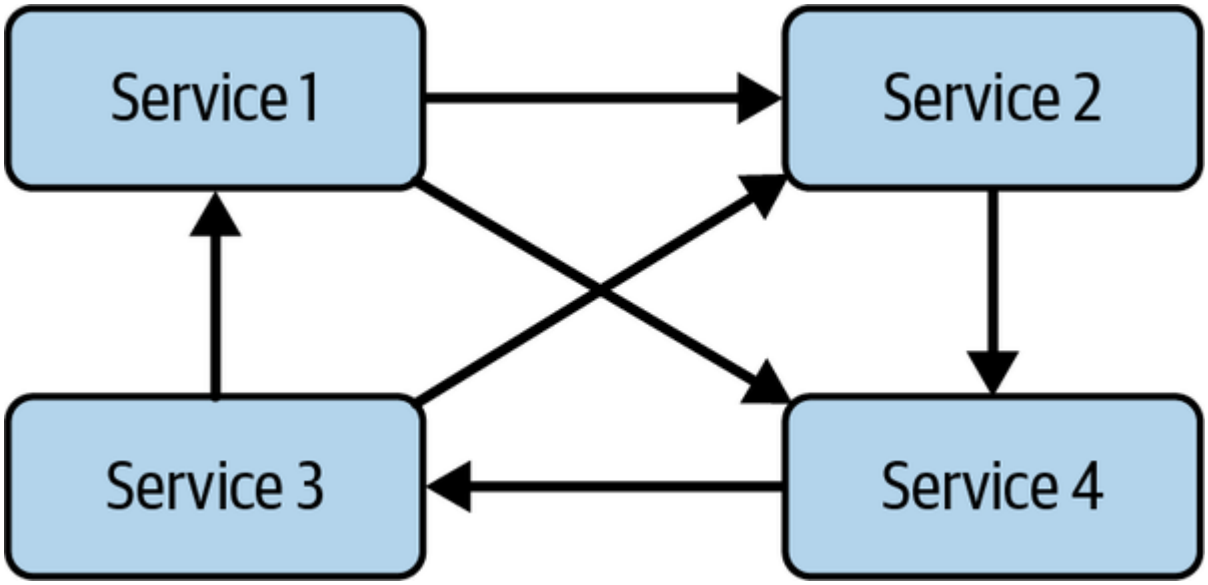

Figure 4-4. The monolith tightly couples its services

Of course, the monolith has cons. For one thing, monoliths are brittle. Because of the vast number of moving parts, updates and releases take longer and tend to bake in "the kitchen sink." If the system has a bug—hopefully, the software's been thoroughly tested before release!—it can harm the entire system.

User-induced problems also happen with monoliths. For example, we saw a monolithic ETL pipeline that took 48 hours to run. If anything broke anywhere in the pipeline, the entire process had to restart. Meanwhile, anxious business users were waiting for their reports, which were already two days late by default and usually arrived much later. Breakages were common enough that the monolithic system was eventually thrown out.

Multitenancy in a monolithic system can also be a significant problem. It can be challenging to isolate the workloads of multiple users. In an on-prem data warehouse, one user-defined function might consume enough CPU to slow the system for other users. Conflicts between dependencies and resource contention are frequent sources of headaches.

Another con of monoliths is that switching to a new system will be painful if the vendor or open source project dies. Because all of your processes are contained in the monolith, extracting yourself out of that system, and onto a new platform, will be costly in both time and money.

Modularity

Modularity (Figure 4-5) is an old concept in software engineering, but modular distributed systems truly came into vogue with the rise of microservices. Instead of relying on a massive monolith to handle your needs, why not break apart systems and processes into their self-contained areas of concern? Microservices can communicate via APIs, allowing developers to focus on their domains while making their applications accessible to other microservices. This is the trend in software engineering and is increasingly seen in modern data systems.

Figure 4-5. With modularity, each service is decoupled from another

Major tech companies have been key drivers in the microservices movement. The famous Bezos API mandate decreases coupling between applications, allowing refactoring and decomposition. Bezos also imposed the two-pizza rule (no team should be so large that two pizzas can't feed the whole group). Effectively, this means that a team will have at most five members. This cap also limits the complexity of a team's domain of responsibility—in particular, the codebase that it can manage. Whereas an extensive monolithic application might entail a group of one hundred people, dividing developers into small groups of five requires that this application be broken into small, manageable, loosely coupled pieces.

In a modular microservice environment, components are swappable, and it's possible to create a *polyglot* (multiprogramming language) application; a Java service can replace a service written in Python. Service customers need worry only about the technical specifications of the service API, not behind-the-scenes details of implementation.

Data-processing technologies have shifted toward a modular model by providing strong support for interoperability. Data is stored in object storage in a standard format such as Parquet in data lakes and lakehouses. Any processing tool that supports the format can read the data and write processed results back into the lake for processing by another tool. Cloud data warehouses support interoperation with object storage through import/export using standard formats and external tables—i.e., queries run directly on data in a data lake.

New technologies arrive on the scene at a dizzying rate in today's data ecosystem, and most get stale and outmoded quickly. Rinse and repeat. The ability to swap out tools as technology changes is invaluable. We view data modularity as a more powerful paradigm than monolithic data engineering. Modularity allows engineers to choose the best technology for each job or step along the pipeline.

The cons of modularity are that there's more to reason about. Instead of handling a single system of concern, now you potentially have countless systems to understand and operate. Interoperability is a potential headache; hopefully, these systems all play nicely together.

This very problem led us to break out orchestration as a separate undercurrent instead of placing it under data management. Orchestration is also important for monolithic data architectures; witness the success of tools like BMC Software's Control-M in the traditional data warehousing space. But orchestrating five or ten tools is dramatically more complex than orchestrating one. Orchestration becomes the glue that binds data stack modules together.

The Distributed Monolith Pattern

The *distributed monolith pattern* is a distributed architecture that still suffers from many of the limitations of monolithic architecture. The basic idea is that one runs a distributed system with

different services to perform different tasks. Still, services and nodes share a common set of dependencies or a common codebase.

One standard example is a traditional Hadoop cluster. A Hadoop cluster can simultaneously host several frameworks, such as Hive, Pig, or Spark. The cluster also has many internal dependencies. In addition, the cluster runs core Hadoop components: Hadoop common libraries, HDFS, YARN, and Java. In practice, a cluster often has one version of each component installed.

A standard on-prem Hadoop system entails managing a common environment that works for all users and all jobs. Managing upgrades and installations is a significant challenge. Forcing jobs to upgrade dependencies risks breaking them; maintaining two versions of a framework entails extra complexity.

Some modern Python-based orchestration technologies—e.g., Apache Airflow—also suffer from this problem. While they utilize a highly decoupled and asynchronous architecture, every service runs the same codebase with the same dependencies. Any executor can execute any task, so a client library for a single task run in one DAG must be installed on the whole cluster. Orchestrating many tools entails installing client libraries for a host of APIs. Dependency conflicts are a constant problem.

One solution to the problems of the distributed monolith is ephemeral infrastructure in a cloud setting. Each job gets its own temporary server or cluster installed with dependencies. Each cluster remains highly monolithic, but separating jobs dramatically reduces conflicts. For example, this pattern is now quite common for Spark with services like Amazon EMR and Google Cloud Dataproc.

A second solution is to properly decompose the distributed monolith into multiple software environments using containers. We have more to say on containers in “Serverless Versus Servers”.

Our Advice

While monoliths are attractive because of ease of understanding and reduced complexity, this comes at a high cost. The cost is the potential loss of flexibility, opportunity cost, and high-friction development cycles.

Here are some things to consider when evaluating monoliths versus modular options:

Interoperability

: Architect for sharing and interoperability.

Avoiding the “bear trap”

: Something that is easy to get into might be painful or impossible to escape.

Flexibility

: Things are moving so fast in the data space right now. Committing to a monolith reduces flexibility and reversible decisions.

Serverless Versus Servers

A big trend for cloud providers is *serverless*, allowing developers and data engineers to run applications without managing servers behind the scenes. Serverless provides a quick time to value for the right use cases. For other cases, it might not be a good fit. Let’s look at how to evaluate whether serverless is right for you.

Serverless

Though serverless has been around for quite some time, the serverless trend kicked off in full force with AWS Lambda in 2014. With the promise of executing small chunks of code on an as-needed basis without having to manage a server, serverless exploded in popularity. The main reasons for its popularity are cost and convenience. Instead of paying the cost of a server, why not just pay when your code is evoked?

Serverless has many flavors. Though function as a service (FaaS) is wildly popular, serverless systems predate the advent of AWS Lambda. For example, Google Cloud's BigQuery is serverless in that data engineers don't need to manage backend infrastructure, and the system scales to zero and scales up automatically to handle large queries. Just load data into the system and start querying. You pay for the amount of data your query consumes and a small cost to store your data. This payment model—paying for consumption and storage—is becoming more prevalent.

When does serverless make sense? As with many other cloud services, it depends; and data engineers would do well to understand the details of cloud pricing to predict when serverless deployments will become expensive. Looking specifically at the case of AWS Lambda, various engineers have found hacks to run batch workloads at meager costs.[11] On the other hand, serverless functions suffer from an inherent overhead inefficiency. Handling one event per function call at a high event rate can be catastrophically expensive, especially when simpler approaches like multithreading or multiprocessing are great alternatives.

As with other areas of ops, it's critical to monitor and model. *Monitor* to determine cost per event in a real-world environment and maximum length of serverless execution, and *model* using this cost per event to determine overall costs as event rates grow. Modeling should also include worst-case scenarios—what happens if my site gets hit by a bot swarm or DDoS attack?

Containers

In conjunction with serverless and microservices, *containers* are one of the most powerful trending operational technologies as of this writing. Containers play a role in both serverless and microservices.

Containers are often referred to as *lightweight virtual machines*. Whereas a traditional VM wraps up an entire operating system, a container packages an isolated user space (such as a filesystem and a few processes); many such containers can coexist on a single host operating system. This provides some of the principal benefits of virtualization (i.e., dependency and code isolation) without the overhead of carrying around an entire operating system kernel.

A single hardware node can host numerous containers with fine-grained resource allocations. At the time of this writing, containers continue to grow in popularity, along with Kubernetes, a container management system. Serverless environments typically run on containers behind the scenes. Indeed, Kubernetes is a kind of serverless environment because it allows developers and ops teams to deploy microservices without worrying about the details of the machines where they are deployed.

Containers provide a partial solution to problems of the distributed monolith mentioned earlier in this chapter. For example, Hadoop now supports containers, allowing each job to have its own isolated dependencies.

Warning

Container clusters do not provide the same security and isolation that full VMs offer. *Container escape*—broadly, a class of exploits whereby code in a container gains privileges outside the container at the OS level—is common enough to be considered a risk for multitenancy. While Amazon EC2 is a truly multitenant environment with VMs from many customers hosted on the same hardware, a Kubernetes cluster should host code only within an environment of mutual trust (e.g., inside the walls of a single

company). In addition, code review processes and vulnerability scanning are critical to ensure that a developer doesn't introduce a security hole.

Various flavors of container platforms add additional serverless features. Containerized function platforms run containers as ephemeral units triggered by events rather than persistent services.[12] This gives users the simplicity of AWS Lambda with the full flexibility of a container environment instead of the highly restrictive Lambda runtime. And services such as AWS Fargate and Google App Engine run containers without managing a compute cluster required for Kubernetes. These services also fully isolate containers, preventing the security issues associated with multitenancy.

Abstraction will continue working its way across the data stack. Consider the impact of Kubernetes on cluster management. While you can manage your Kubernetes cluster—and many engineering teams do so—even Kubernetes is widely available as a managed service. What comes after Kubernetes? We're as excited as you to find out.

How to Evaluate Server Versus Serverless

Why would you want to run your own servers instead of using serverless? There are a few reasons. Cost is a big factor. Serverless makes less sense when the usage and cost exceed the ongoing cost of running and maintaining a server (Figure 4-6). However, at a certain scale, the economic benefits of serverless may diminish, and running servers becomes more attractive.

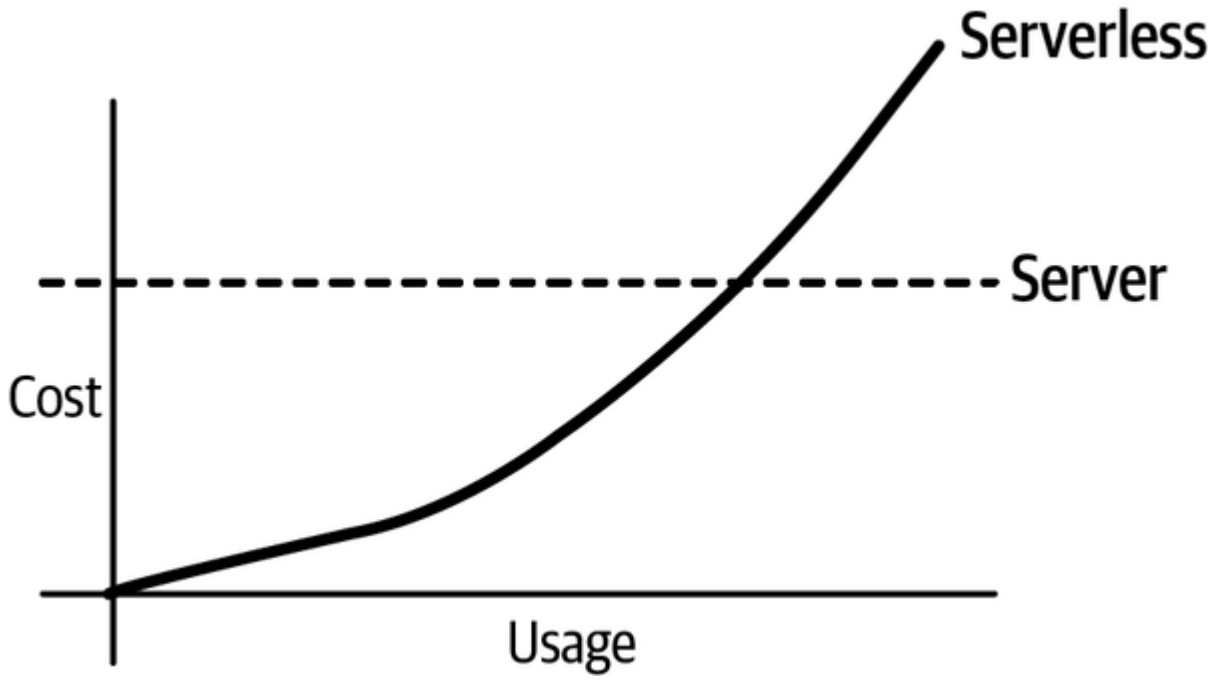

Figure 4-6. Cost of serverless versus utilizing a server

Customization, power, and control are other major reasons to favor servers over serverless. Some serverless frameworks can be underpowered or limited for certain use cases. Here are some things to consider when using servers, particularly in the cloud, where server resources are ephemeral:

Expect servers to fail.

> Server failure will happen. Avoid using a "special snowflake" server that is overly customized and brittle, as this introduces a glaring vulnerability in your architecture. Instead, treat servers as ephemeral resources that you can create as needed and then delete. If your application requires specific code to be installed on the server, use a boot script or build an image. Deploy code to the server through a CI/CD pipeline.

Use clusters and autoscaling.

Take advantage of the cloud's ability to grow and shrink compute resources on demand. As your application increases its usage, cluster your application servers, and use autoscaling capabilities to automatically horizontally scale your application as demand grows.

Treat your infrastructure as code.

Automation doesn't apply to just servers and should extend to your infrastructure whenever possible. Deploy your infrastructure (servers or otherwise) using deployment managers such as Terraform, AWS CloudFormation, and Google Cloud Deployment Manager.

Use containers.

For more sophisticated or heavy-duty workloads with complex installed dependencies, consider using containers on either a single server or Kubernetes.

Our Advice

Here are some key considerations to help you determine whether serverless is right for you:

Workload size and complexity

Serverless works best for simple, discrete tasks and workloads. It's not as suitable if you have many moving parts or require a lot of compute or memory horsepower. In that case, consider using containers and a container workflow orchestration framework like Kubernetes.

Execution frequency and duration

How many requests per second will your serverless application process? How long will each request take to process? Cloud serverless platforms have limits on execution frequency, concurrency, and duration. If your application can't function neatly within these limits, it is time to consider a container-oriented approach.

Requests and networking

Serverless platforms often utilize some form of simplified networking and don't support all cloud virtual networking features, such as VPCs and firewalls.

Language

What language do you typically use? If it's not one of the officially supported languages supported by the serverless platform, you should consider containers instead.

Runtime limitations

Serverless platforms don't give you complete operating system abstractions. Instead, you're limited to a specific runtime image.

Cost

Serverless functions are incredibly convenient but potentially expensive. When your serverless function processes only a few events, your costs are low; costs rise rapidly as the event count increases. This scenario is a frequent source of surprise cloud bills.

In the end, abstraction tends to win. We suggest looking at using serverless first and then servers—with containers and orchestration if possible—once you've outgrown serverless options.

Optimization, Performance, and the Benchmark Wars

Imagine that you are a billionaire shopping for new transportation. You've narrowed your choice to two options:

- 787 Business Jet
  - Range: 9,945 nautical miles (with 25 passengers)
  - Maximum speed: 0.90 Mach
  - Cruise speed: 0.85 Mach
  - Fuel capacity: 101,323 kilograms
  - Maximum takeoff weight: 227,930 kilograms
  - Maximum thrust: 128,000 pounds
- Tesla Model S Plaid
  - Range: 560 kilometers
  - Maximum speed: 322 kilometers/hour
  - 0–100 kilometers/hour: 2.1 seconds
  - Battery capacity: 100 kilowatt hours
  - Nurburgring lap time: 7 minutes, 30.9 seconds
  - Horsepower: 1020
  - Torque: 1050 lb-ft

Which of these options offers better performance? You don't have to know much about cars or aircraft to recognize that this is an idiotic comparison. One option is a wide-body private jet designed for intercontinental operation, while the other is an electric supercar.

We see such apples-to-oranges comparisons made all the time in the database space. Benchmarks either compare databases that are optimized for completely different use cases, or use test scenarios that bear no resemblance to real-world needs.

Recently, we saw a new round of benchmark wars flare up among major vendors in the data space. We applaud benchmarks and are glad to see many database vendors finally dropping DeWitt clauses from their customer contracts.[13] Even so, let the buyer beware: the data space is full of nonsensical benchmarks.[14] Here are a few common tricks used to place a thumb on the benchmark scale.

Big Data...for the 1990s

Products that claim to support "big data" at petabyte scale will often use benchmark datasets small enough to easily fit in the storage on your smartphone. For systems that rely on caching layers to deliver performance, test datasets fully reside in solid-state drive (SSD) or memory, and benchmarks can show ultra-high performance by repeatedly querying the same data. A small test dataset also minimizes RAM and SSD costs when comparing pricing.

To benchmark for real-world use cases, you must simulate anticipated real-world data and query size. Evaluate query performance and resource costs based on a detailed evaluation of your needs.

Nonsensical Cost Comparisons

Nonsensical cost comparisons are a standard trick when analyzing a price/performance or TCO. For instance, many MPP systems can't be readily created and deleted even when they reside in a cloud environment; these systems run for years on end once they've been configured. Other databases support a dynamic compute model and charge either per query or per second of use. Comparing ephemeral and non-ephemeral systems on a cost-per-second basis is nonsensical, but we see this all the time in benchmarks.

Asymmetric Optimization

The deceit of asymmetric optimization appears in many guises, but here's one example. Often a vendor will compare a row-based MPP system against a columnar database by using a benchmark that runs complex join queries on highly normalized data. The normalized data model is optimal for the row-based system, but the columnar system would realize its full potential only with some schema changes. To make matters worse, vendors juice their systems with an extra shot of join optimization (e.g., preindexing joins) without applying comparable tuning in the competing database (e.g., putting joins in a materialized view).

Caveat Emptor

As with all things in data technology, let the buyer beware. Do your homework before blindly relying on vendor benchmarks to evaluate and choose technology.

Undercurrents and Their Impacts on Choosing Technologies

As seen in this chapter, a data engineer has a lot to consider when evaluating technologies. Whatever technology you choose, be sure to understand how it supports the undercurrents of the data engineering lifecycle. Let's briefly review them again.

Data Management

Data management is a broad area, and concerning technologies, it isn't always apparent whether a technology adopts data management as a principal concern. For example, behind the scenes, a third-party vendor may use data management best practices—regulatory compliance, security, privacy, data quality, and governance—but hide these details behind a limited UI layer. In this case, while evaluating the product, it helps to ask the company about its data management practices. Here are some sample questions you should ask:

- How are you protecting data against breaches, both from the outside and within?
- What is your product's compliance with GDPR, CCPA, and other data privacy regulations?
- Do you allow me to host my data to comply with these regulations?
- How do you ensure data quality and that I'm viewing the correct data in your solution?

There are many other questions to ask, and these are just a few of the ways to think about data management as it relates to choosing the right technologies. These same questions should also apply to

the OSS solutions you're considering.

DataOps

Problems will happen. They just will. A server or database may die, a cloud's region may have an outage, you might deploy buggy code, bad data might be introduced into your data warehouse, and other unforeseen problems may occur.

When evaluating a new technology, how much control do you have over deploying new code, how will you be alerted if there's a problem, and how will you respond when there's a problem? The answer largely depends on the type of technology you're considering. If the technology is OSS, you're likely responsible for setting up monitoring, hosting, and code deployment. How will you handle issues? What's your incident response?

Much of the operations are outside your control if you're using a managed offering. Consider the vendor's SLA, the way they alert you to issues, and whether they're transparent about how they're addressing the case, including providing an ETA to a fix.

Data Architecture

As discussed in Chapter 3, good data architecture means assessing trade-offs and choosing the best tools for the job while keeping your decisions reversible. With the data landscape morphing at warp speed, the *best tool* for the job is a moving target. The main goals are to avoid unnecessary lock-in, ensure interoperability across the data stack, and produce high ROI. Choose your technologies accordingly.

Orchestration Example: Airflow

Throughout most of this chapter, we have actively avoided discussing any particular technology too extensively. We make an exception for orchestration because the space is currently dominated by one open source technology, Apache Airflow.

Maxime Beauchemin kicked off the Airflow project at Airbnb in 2014. Airflow was developed from the beginning as a noncommercial open source project. The framework quickly grew significant mindshare outside Airbnb, becoming an Apache Incubator project in 2016 and a full Apache-sponsored project in 2019.

Airflow enjoys many advantages, largely because of its dominant position in the open source marketplace. First, the Airflow open source project is extremely active, with a high rate of commits and a quick response time for bugs and security issues, and the project recently released Airflow 2, a major refactor of the codebase. Second, Airflow enjoys massive mindshare. Airflow has a vibrant, active community on many communications platforms, including Slack, Stack Overflow, and GitHub. Users can easily find answers to questions and problems. Third, Airflow is available commercially as a managed service or software distribution through many vendors, including GCP, AWS, and Astronomer.io.

Airflow also has some downsides. Airflow relies on a few core nonscalable components (the scheduler and backend database) that can become bottlenecks for performance, scale, and reliability; the scalable parts of Airflow still follow a distributed monolith pattern. (See "Monolith Versus Modular".) Finally, Airflow lacks support for many data-native constructs, such as schema management, lineage, and cataloging; and it is challenging to develop and test Airflow workflows.

We do not attempt an exhaustive discussion of Airflow alternatives here but just mention a couple of the key orchestration contenders at the time of writing. Prefect and Dagster aim to solve some of the problems discussed previously by rethinking components of the Airflow architecture. Will there be other orchestration frameworks and technologies not discussed here? Plan on it.

We highly recommend that anyone choosing an orchestration technology study the options discussed here. They should also acquaint themselves with activity in the space, as new developments will certainly occur by the time you read this.

Software Engineering

As a data engineer, you should strive for simplification and abstraction across the data stack. Buy or use prebuilt open source solutions whenever possible. Eliminating undifferentiated heavy lifting should be your big goal. Focus your resources—custom coding and tooling—on areas that give you a solid competitive advantage. For example, is hand-coding a database connection between your production database and your cloud data warehouse a competitive advantage for you? Probably not. This is very much a solved problem. Pick an off-the-shelf solution (open source or managed SaaS) instead. The world doesn't need the millionth +1 database-to-cloud data warehouse connector.

On the other hand, why do customers buy from you? Your business likely has something special about the way it does things. Maybe it's a particular algorithm that powers your fintech platform. By abstracting away a lot of the redundant workflows and processes, you can continue chipping away, refining, and customizing the things that move the needle for the business.

Conclusion

Choosing the right technologies is no easy task, especially when new technologies and patterns emerge daily. Today is possibly the most confusing time in history for evaluating and selecting technologies. Choosing technologies is a balance of use case, cost, build versus buy, and modularization. Always approach technology the same way as architecture: assess trade-offs and aim for reversible decisions.

Additional Resources

- *Cloud FinOps* by J. R. Storment and Mike Fuller (O'Reilly)
- "Cloud Infrastructure: The Definitive Guide for Beginners" by Matthew Smith
- "The Cost of Cloud, a Trillion Dollar Paradox" by Sarah Wang and Martin Casado
- FinOps Foundation's "What Is FinOps" web page
- "Red Hot: The 2021 Machine Learning, AI and Data (MAD) Landscape" by Matt Turck
- Ternary Data's "What's Next for Analytical Databases? w/ Jordan Tigani (MotherDuck)" video
- "The Unfulfilled Promise of Serverless" by Corey Quinn
- "What Is the Modern Data Stack?" by Charles Wang

[1] For more details, see "Total Opportunity Cost of Ownership" by Joseph Reis in *97 Things Every Data Engineer Should Know* (O'Reilly).

[2] J. R. Storment and Mike Fuller, *Cloud FinOps* (Sebastopol, CA: O'Reilly, 2019), 6, *https://oreil.ly/RvRvX*.

[3] This is a major point of emphasis in Storment and Fuller, *Cloud FinOps*.

[4] Examples include Google Cloud Anthos and AWS Outposts.

[5] Raghav Bhargava, “Evolution of Dropbox’s Edge Network,” Dropbox.Tech, June 19, 2017, *https://oreil.ly/RAwPf*.

[6] Akhil Gupta, “Scaling to Exabytes and Beyond,” Dropbox.Tech, March 14, 2016, *https://oreil.ly/5XPKv*.

[7] “Dropbox Migrates 34 PB of Data to an Amazon S3 Data Lake for Analytics,” AWS website, 2020, *https://oreil.ly/wpVoM*.

[8] Todd Hoff, “The Eternal Cost Savings of Netflix’s Internal Spot Market,” High Scalability, December 4, 2017, *https://oreil.ly/LLoFt*.

[9] Todd Spangler, “Netflix Bandwidth Consumption Eclipsed by Web Media Streaming Applications,” *Variety*, September 10, 2019, *https://oreil.ly/tTm3k*.

[10] Amir Efrati and Kevin McLaughlin, “Apple’s Spending on Google Cloud Storage on Track to Soar 50% This Year,” *The Information*, June 29, 2021, *https://oreil.ly/OlFyR*.

[11] Evan Sangaline, “Running FFmpeg on AWS Lambda for 1.9% the Cost of AWS Elastic Transcoder,” Intoli blog, May 2, 2018, *https://oreil.ly/myzOv*.

[12] Examples include OpenFaaS, Knative, and Google Cloud Run.

[13] Justin Olsson and Reynold Xin, “Eliminating the Anti-competitive DeWitt Clause for Database Benchmarking,” Databricks, November 8, 2021, *https://oreil.ly/3iFOE*.

[14] For a classic of the genre, see William McKnight and Jake Dolezal, “Data Warehouse in the Cloud Benchmark,” GigaOm, February 7, 2019, *https://oreil.ly/QjCmA*.