# Fundamentals of Data Engineering - Reis & Housley

## Part I. Foundation and Building Blocks

### Chapter 4 – Choosing Technologies Across the Data Engineering Lifecycle

* DE nowadays suffers from an embarrassment of riches 🡪 no shortage of tech 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, + more
* **However, it’s easy to get caught up in chasing bleeding-edge tech while losing sight of the core purpose of DE: designing robust + reliable systems to carry data through the full lifecycle + serve it according to the needs of end users**
* Just as structural engineers carefully choose technologies + materials to realize an architect’s vision for a building, **DE’s are tasked w/ making appropriate tech choices to shepherd data through the lifecycle to serve data applications + users**
* Chapter 3 = “good” data architecture + why it matters, Chapter 4 explains **how to choose the right technologies to serve this architecture**
* **DE’s *must* choose good technologies to make the best possible data product**
* **The criteria to choose a good data technology is simple: does it add value to a data product + 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”, which is the wrong way to think about architecture
* **Architecture is the high-level design, roadmap, + blueprint of data systems that satisfy the strategic aims for the business 🡪 the what, why, + when**
* **Tools are used to make the architecture a reality 🡪 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** (<http://radar.oreilly.com/2014/10/resume-driven-development.html>), + a lack of expertise in architecture
* In practice, this **prioritization of tech often means they cobble together a kind of Dr. Seuss fantasy machine rather than a true data architecture**
* **It’s strongly advised to be against choosing tech before getting architecture right: Architecture first, technology second**
* The following are some **considerations for choosing data technologies** **(*once we have a strategic architecture blueprint*)** across the DE 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, multi-cloud)
* 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

* **1st thing you need to assess = your team’s size and its capabilities with tech**
* 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 DE 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, + **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 tech at a giant tech company + then try to emulate these same extremely complex technologies + practices = **cargo-cult engineering, generally a big mistake that consumes a lot of valuable time + money, often w/ little to nothing to show in return**
* **Especially for small teams or teams w/ weaker technical chops, use as many managed + SaaS tools as possible, + 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, stick w/ technologies + workflows with which the team is familiar**
* Sometimes 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 tech, speed to market wins** 🡪 this means **choosing the right technologies that help you deliver features + data faster while maintaining high-quality standards + security**
* Also means **working in a tight feedback loop of launching, learning, iterating, + making improvements**
* **Perfect is the enemy of good**
* Some data teams will deliberate on tech choices for months/years w/out reaching any decisions
* **Slow decisions + output = the kiss of death to data teams**
* More than a few data teams have dissolved for moving too slow + failing to deliver the value they were hired to produce
* **Deliver value early and often**
* **As mentioned, use what works**
* 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, + 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 + operates w/ other technologies**
* **Interoperability**describes **how various technologies or systems connect, exchange information, + interact**
* Say you’re valuating 2 technologies, A and B
* How easily does technology A integrate w/ 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 + open-source projects will target specific platforms + systems to interoperate**
* Most data ingestion and visualization tools have built-in integrations w/ popular DW’s + data lakes
* Furthermore, **popular data-ingestion tools will integrate w/ common APIs and services, such as CRMs, accounting software, + more**
* **Sometimes standards are in place for interoperability**
* Almost all databases allow connections via **Java Database Connectivity (JDBC)** or **Open Database Connectivity (ODBC)**, meaning 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, as **every REST API has its quirks**
* **In these cases, it’s up to the vendor or OSS project to ensure smooth integration w/ other technologies and systems**
* Always be aware of how simple it will be to connect your various technologies across the DE lifecycle
* As mentioned, **design for modularity and give yourself the ability to easily swap out technologies as new practices + alternatives become available**

#### Cost Optimization and Business Value

* In a perfect world, you’d get to experiment w/ all the latest, coolest tech w/out considering cost, time investment, or value added to the business
* In reality, budgets + time are finite, + the cost is a major constraint for choosing the right data architectures + technologies
* Your organization expects a positive ROI from your data projects, so you must understand the basic costs you can control
* Tech = major cost driver, so tech choices + management strategies will significantly impact a budget
* We **look at costs through 3 main lenses: total cost of ownership**, **opportunity cost**, and **FinOps**

##### i) Total Cost of Ownership

* **Total cost of ownership (TCO)** **= total estimated cost of an initiative, including the direct + indirect costs of products + services utilized**
* **Direct costs**can be directly attributed to an initiative
* Examples = salaries of a team working on the initiative or the AWS bill for all services consumed
* **Indirect costs**, AKA **overhead** = **independent of the initiative + must be paid regardless of where they’re attributed**
* Apart from direct + indirect costs, ***HOW* something is purchasedimpacts the way costs are accounted for**
* **Expenses fall into 2 big groups**: **capital expenses** and **operational expenses**.
* **Capital expenses**, AKA **capex**, **require an up-front investment**
* **Payment is required *today***
* **Before the cloud existed, companies would typically purchase hardware + software up front through large acquisition contracts**
* In addition, significant investments were required to host hardware in server rooms, data centers, + 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 w/ a long-term plan to achieve a positive ROI on the effort + expense put forth**
* **Operational expenses**, AKA **opex** = the opposite of capex in certain respects
* Opex is **gradual + 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**, as data systems often required multimillion-dollar contracts
* T**his 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 + hardware**
* Cloud-based services let DE’s iterate quickly w/ various software + tech configurations, often inexpensively
* **DE’s 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, + potentially hampers a DE’s flexibility to try new things**
* **Given the upside for flexibility + low initial costs, urge DE’s to take an opex-first approach centered on the cloud + flexible, pay-as-you-go technologies**

##### ii) Total Opportunity Cost of Ownership

* **Any choice inherently excludes *other* possibilities**
* **Total opportunity cost of ownership (TOCO)** = the **cost of lost opportunities that we incur in choosing a technology, an architecture, or a process**
* See more: “97 Things Every Data Engineer Should Know”, <https://www.oreilly.com/library/view/97-things-every/9781492062400/>
* 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 + is difficult to move away from.
* DE’s often fail to evaluate TOCO when undertaking a new project, which 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 + everything it entails: the team to support it, training, setup, + 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 + 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 + options?
* **The first step to minimizing opportunity cost is evaluating it with eyes wide open**
* **Countless data teams get stuck with tech that seemed good at the time + are either not flexible for future growth or simply obsolete**
* Inflexible data technologies are a lot like bear traps 🡪 easy to get into + extremely painful to escape

##### iii) FinOps

* We already touched on FinOps
* As discussed, **typical cloud spending is inherently opex: companies pay for services to run critical data processes rather than making up-front purchases + clawingback value over time**
* The **goal of FinOps is to fully operationalize financial accountability + business value by applying the DevOps-like practices of monitoring + dynamically adjusting systems**
* Emphasize one thing about FinOps that is well embodied in this quote: “*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 + feature release velocity, or even help shut down a data center”*
* **In our setting of DE, the ability to iterate quickly + scale dynamically is invaluable for creating business value**
* This is **one of the major motivations for shifting data workloads to the cloud**

#### Today Vs. The Future: Immutable Vs. Transient Technologies

* **In an exciting domain like DE, 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 over-architecting + overengineering
* **Tooling chosen for the future may be stale and out-of-date when this future arrives, as the future frequently looks little like what we envisioned years before**.
* As many life coaches would tell you, **focus on the present** 🡪 **choose the best tech for the moment + *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 + 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 **2 classes of tools to consider: immutable and transitory**
* **Immutable technologies** might be components that underpin the cloud or languages + paradigms that have stood the test of time
* Examples of **immutable cloud technologies = object storage, networking, servers, + security**
* Object storage such as Amazon S3 + Azure Blob Storage will be around from today until the end of the decade, + probably much longer
* Storing your data in object storage is a wise choice
* Object storage continues to improve in various ways + constantly offers new options, but your data will be safe + usable in object storage regardless of the rapid evolution of tech as a whole.
* For languages, **SQL + bash** have been around for many decades, + 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, RDBs, the C programming language, or the x86 processor architecture
* **Apply 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 + gone between 2010-2020?
* Backbone. js, Ember.js, + Knockout were popular in the early 2010s, and React + Vue.js have massive mindshare today
* What’s the popular frontend JavaScript framework three years from now? Who knows.
* New well-funded entrants + OSS projects arrive on the data front every day
* Every vendor will say their product will change the industry + “make the world a better place”
* **Most of these companies + projects don’t get long-term traction + 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 tech 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 introduced in Chapter 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 w/ rapid uptake because it allowed both analysts + engineers to query massive datasets w/out 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**

##### Advice

* **Given the rapid pace of tooling and best-practice changes, evaluate tools every 2 years**
* **Whenever possible, find the immutable technologies along the DE lifecycle, + use those as your base**
* **Build transitory tools *around* the immutables**
* 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, avoid “bear traps”
* **Go into a new technology w/ 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 tech stacks
* A slow shift toward the cloud culminates in a veritable stampede of companies spinning up workloads on AWS, Azure, + Google Cloud Platform (GCP)
* In the last decade, many CTOs have come to view their decisions around tech 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 + catastrophic costs.
* Let’s look at the principal places to run your tech stack: **on-prem, cloud, hybrid cloud, + multi-cloud**

##### On-Premises

* While new startups are increasingly born in the cloud, **on-prem 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 + 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 as older hardware ages + 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 DE’s in charge of on-prem systems, this means buying large-enough systems to allow good performance for peak load + large jobs *w/out overbuying and overspending*
* On the one hand, established companies have established operational practices that have served them well
* Suppose a company that relies on IT has been in business for some time
* This means it has managed to juggle the cost + personnel requirements of running its hardware, managing software environments, deploying code from dev teams, + running databases + big data systems
* On the other hand, established companies see their younger, more agile competition scaling rapidly + 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 like the Black Friday shopping spike
* **Companies in competitive sectors generally don’t have the option to stand still.**
* Competition is fierce, + there’s always the threat of being “disrupted” by more agile competition, often backed by a large pile of VC 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 (K8’s), microservices, + continuous deployment (CD) while keeping their hardware running on prem
* Or it could involve a *complete* migration to the cloud

##### Cloud

* The cloud flips the on-prem model on its head 🡪 instead of purchasing hardware, you simply **rent hardware + 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, + subsequent usage is billed in per-second increments 🡪 **allows cloud users to *dynamically* scale resources that were inconceivable w/ on-prem servers**)
* In a cloud environment, engineers **can quickly launch projects + experiment w/out 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 + time**
* The **early cloud era was dominated by infrastructure as a service (IaaS) offerings** (products such as VMs + 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 = **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), + **managed K8’s** such as Google Kubernetes Engine (GKE) and Azure Kubernetes Service (AKS)
* **PaaS services allow engineers to ignore the operational details of managing individual machines + deploying frameworks across distributed systems**
* **Provides turnkey access to complex, autoscaling systems w/ minimal operational overhead**
* **SaaS offerings move one additional step up the ladder of abstraction**
* **SaaS typically provides a fully functioning enterprise software platform w/ little operational management**
* SaaS examples: Salesforce, Google Workspace, Microsoft 365, Zoom, + Fivetran
* Both the major public clouds + 3rd parties offer SaaS platforms
* SaaS covers a whole spectrum of enterprise domains, including video conferencing, data management, ad tech, office applications, + CRM systems.
* We also discuss **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 w/out operational awareness of underlying servers**
* Many 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 w/ existing data centers + IT infrastructure
* **Dynamic, seamless scaling is extremely valuable to businesses that deal with seasonality** (like retail businesses coping w/ Black Friday load) **+ web traffic load spikes**
* COVID-19 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 w/ substantially increased load due to a spike in online shopping, web app usage, + remote work.

###### A Brief Detour on Cloud Economics

* 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** (closely related to FinOps)
* **Enterprises that migrate to the cloud often make major deployment errors by not appropriately adapting their practices to the cloud pricing model**
* 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 + 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**
* 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)**
* We 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 w/ varying technical characteristics + risks attached**
* While providers are extremely tight-lipped about details of their internal systems, **there are massive opportunities for optimization + scaling by understanding cloud pricing + exchanging notes w/ other users.**
* Look at the example of archival cloud storage
* In 2022-2023, GCP openly admits that its archival class storage runs on the same clusters as standard cloud storage, yet the price per GB per month of archival storage is roughly 1/17 that of standard storage. *How is this possible?*
* Educated guess = **When purchasing cloud storage, each disk in a storage cluster has 3 assets that cloud providers + consumers use**
* 1) It has a **certain storage capacity** (say, 10 TB)
* 2) It **supports a certain number of input/output operations (IOPs) per second** (say, 100)
* 3) **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**
* Ex: 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 w/out 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, we risk paying a high price if we ever need to retrieve data**, a price most will happily pay in a *true* emergency
* Similar considerations apply to nearly *any* cloud service
* While on-prem servers are essentially sold as commodity hardware, the cost model in the cloud is more subtle
* **Rather than just charging for CPU cores, memory, + features, cloud vendors monetize *characteristics* such as durability, reliability, longevity, + 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

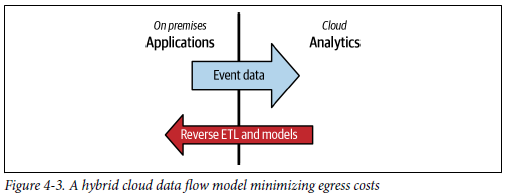
* May seem like a silly tautology, but the belief that cloud services are just like familiar on-prem servers is a widespread cognitive error that plagues cloud migrations + leads to horrifying bills
* This demonstrates a broader issue in tech that referred to as the **curse of familiarity**
* **Many new tech products are *intentionally* designed to look like something familiar to facilitate ease of use and accelerate adoption**
* **But any new tech product has subtleties + wrinkles that users must learn to identify, accommodate, and optimize**
* Moving on-prem 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-prem counterparts**
* The **key to finding value in the cloud is understanding + 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 + 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**
* DE’s can create new value in the cloud **by accomplishing things that were impossible in their on-prem environment**
* Ex: It is possible to quickly spin up massive compute clusters to run complex transformations at scales that were unaffordable for on-prem hardware

Data Gravity

* In addition to basic errors such as following on-prem operational practices in the cloud, DE’s need to watch out for other aspects of cloud pricing + 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 + 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 + migrate processes can be very high**

##### Hybrid Cloud

* As more established businesses migrate into the cloud, **the hybrid cloud model is growing in importance** since 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 + associated hardware
* Thus, they **may migrate only to specific workloads where they see immediate benefits in the cloud environment**
* Ex: An on-prem Spark stack is migrated to ephemeral cloud clusters, reducing the operational burden of managing software + hardware for the DE team + 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 (see below)**



* That is, **on-prem applications generate event data 🡪 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-prem for deploying models to applications, reverse ETL, etc.**
* A new generation of managed hybrid cloud service offerings **also allows customers to locate cloud-managed servers in their data centers**
* Gives users the **ability to incorporate the best features in each cloud alongside on-prem infrastructure**

##### Multi-cloud

* This simply refers to **deploying workloads to multiple public clouds**
* Companies may have several motivations for multi-cloud deployments
* **1) SaaS platforms often wish to offer services close to existing customer cloud workloads**
* Snowflake + Databricks provide their SaaS offerings across multiple clouds for this reason
* This is **especially critical for data-intensive applications, where network latency + bandwidth limitations hamper performance, + data egress costs can be prohibitive**
* 2) Another common motivation for a multi-cloud approach is to **take advantage of the best services across several clouds**
* Ex: A company might want to handle its Google Ads and Analytics data on Google Cloud and deploy K8s through GKE
* The company might also adopt Azure specifically for Microsoft workloads
* Also, they may like AWS b/c it has several best-in-class services (e.g., AWS Lambda) + 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 multi-cloud more compelling.
* A **multi-cloud methodology has several disadvantages**
* As just mentioned, **data egress costs + networking bottlenecks are critical**
* Going multi-cloud **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, + multi-cloud networking can be diabolically complicated**
* A new generation of **“cloud of clouds” services** **aims to facilitate multi-cloud w/ reduced complexity by offering services *across* clouds + 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**
* W/in a few years of 2022-2023, many more of these services will be available
* DE’s + 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-prem and in the cloud, the rise of blockchain, Web 3.0, + 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 tech decisions

##### Advice

* We are still at the beginning of the transition to the cloud, thus the evidence + arguments around workload placement + 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 + more generally toward more managed service offerings such as AWS Glue, Google BigQuery, + Snowflake**
* We’ve also seen the **emergence of new workload placement abstractions**
* On-prem services are becoming more cloud-like + abstracted
* Hybrid cloud services allow customers to run fully managed services w/in their walls while facilitating tight integration between local + remote environments
* Further, the “cloud of clouds” is beginning to take shape, fueled by 3rd-party services + public cloud vendors
* **Choose technologies for the present, but look toward the future**
* As mentioned earlier, you **need to keep one eye on the present while planning for unknowns**
* Right now is a tough time to plan workload placements + migrations.
* B/c of the fast pace of competition + change in the cloud industry, the decision space will look very different in 5-10 years
* **It is tempting to take into account *every* possible *future* architecture permutation, but it is critical to avoid this endless trap of analysis**
* Instead, **plan for the present 🡪 Choose the best technologies for your *current* needs + concrete plans for the near future**
* **Choose your deployment platform based on real business needs while focusing on simplicity + flexibility**
* **In particular, don’t choose a complex multi-cloud 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 tech need for specific services on 2 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 emphasized before, **every technology (even OSS) comes w/ some degree of lock-in**
* A single-cloud strategy has significant advantages of simplicity + integration but comes w/ significant lock-in attached
* In this instance, we’re talking about mental flexibility: the flexibility to evaluate the current state of the world + 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

* Sarah Wang and Martin Casado published *“The Cost of Cloud, A Trillion Dollar Paradox”*, an article that generated significant sound + fury in the tech space
* Readers widely interpreted the article as a call for the **repatriation** of cloud workloads to on-prem servers
* They make a somewhat more subtle argument that **companies should expend significant resources to control cloud spending + should consider repatriation as a possible option**
* 1 part of their discussion = Wang + 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*
* This case study is frequently used w/out appropriate context + is a compelling example of the **false equivalence logical fallacy**
* Dropbox provides particular services where ownership of hardware + data centers can offer a competitive advantage
* Companies should not rely excessively on Dropbox’s example when assessing cloud + on-prem 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 + 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 w/ transit providers (regional + global ISPs), + hundreds of new peering partners (where we exchange traffic directly rather than through an ISP).”
* The data egress costs would be extremely high in a public cloud environment
* Third, Dropbox is essentially a cloud storage vendor, but one w/ a highly specialized storage product that combines object + 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 + 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 + hardware stack
* 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 + data synchronization while offloading software + hardware management in other areas, such as data analytics
* 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, w/ 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
* Netflix also runs its application backend + data analytics on AWS
* However, rather than using the AWS content distribution network, Netflix has built a **custom CDN** in collaboration w/ ISPs, utilizing a highly specialized combination of software + hardware
* For a company that consumes a substantial slice of all internet traffic, 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 + network connections in particular circumstances
* The biggest modern success stories of companies building + maintaining hardware involve *extraordinary scale* (exabytes of data, terabits per second of bandwidth, etc.) + 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-prem 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 + performance advantage by migrating iCloud storage to its own servers

#### Buy Vs. Build

* The **argument for building = you have end-to-end control over the solution + are not at the mercy of a vendor or OSS community**
* The **argument supporting buying comes down to resource constraints + expertise**
* Do you have the expertise to build a better solution than something already available?
* **Either decision comes down to TCO, TOCO, + whether the solution provides a competitive advantage to your organization**
* It’s so far been suggested to **be investing in building + 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 + paid services (both of which may have communities of volunteers or highly paid teams of amazing engineers), you’re **foolish to build everything yourself**
* When you need new car tires, do you get the raw materials, create the tires from scratch, + install them yourself? Like most, you’re probably buying tires + having someone install them
* *The same argument applies to build versus buy*
* Seen teams that have built their databases from scratch, when a simple OSS RDBMS would’ve 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 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 OSS 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 + 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** w/in companies is becoming an **organic, continuous process**

##### Open-Source Software

* **Open-source software (OSS)** is **a software distribution model in which software, + the underlying codebase, is made available for general use, typically under specific licensing terms**
* **Often** **created + maintained by a distributed team of collaborators**
* OSS is **free to use, change, + distribute most of the time, but *w/ 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 + maintaining OSS vary
* Sometimes OSS is organic, springing from the mind of an individual or a small team that creates a novel solution + 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 2 main flavors**: **community managed** and **commercial OSS**.
* **Community-managed OSS**
* **OSS projects succeed w/ a strong community + a vibrant user base**
* **Community-managed OSS** is a prevalent path for OSS projects
* The **community opens up high rates of innovations + contributions from developers worldwide** w/ 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 + popularity**
* Look at the number of GitHub stars, forks, and commit volume + recency
* Pay attention to community activity on related chat groups + 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 w/ the framework**
* **Maturity**: **How long has the project been around, how active is it today**, + **how usable** are people finding it **in production**?
* A project’s maturity indicates that people find it useful + 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 + the way they’re addressed**
* Are they addressed quickly? If so, what’s the process to submit an issue + get it resolved?
* **Team:**Is a **company sponsoring** the OSS project? Who are the **core contributors**?
* **Developer relations + community management**: What is the project doing to **encourage uptake + adoption?**
* Is there a vibrant chat community (e.g., in Slack) that provides encouragement + support?
* **Contributing**: Does the project **encourage + accept pull requests?**
* What are the process + timelines for pull requests to be accepted + included in main codebase?
* **Roadmap**: ***Is there* a project roadmap?**
* If so, is it clear + transparent?
* **Self-hosting and maintenance**: Do you **have the resources to host + maintain the OSS solution?**
* If so, what’s the TCO + TOCO versus buying a managed service from the OSS vendor?
* **Giving back to the community**: If you like the project + are actively using it, consider investing in it
* **Can contribute to the codebase, help fix issues, + give advice in the community forums + chats**
* If the project allows **donations**, consider making one
* Many OSS projects are essentially community-service projects, + 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 + maintain the solution in your environment**
* May be trivial or extremely complicated + cumbersome, depending on the OSS application
* **Commercial vendors try to solve this management headache by hosting + managing the OSS solution for you, typically as a cloud SaaS offering**
* Examples of such vendors: Databricks (Spark), Confluent (Kafka), DBT Labs (dbt), + 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 w/ 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 VC money to commercialize the OSS project, + the company scales as a fast-moving rocket ship**
* At this point, the DE has 2 options
* Can continue using the community-managed OSS version, which you need to continue maintaining on your own (updates, server/container maintenance, PR’s for bug fixes, etc.)
* Or pay the vendor + 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 + 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 + subsequent releases
* **Support**: Support **cannot be understated**, + 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, + what is *not* covered**
* Anything that’s not covered by support will be *your* responsibility to own + manage
* **Releases and bug fixes**: Is the **vendor transparent about the release schedule, improvements, + 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, + 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, + 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?**
* May be surprised by the number of companiesprimarily concerned w/ growing their customer count, GitHub stars, or Slack channel membership w/out 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 w/ certain vendors co-opting OSS projects + subsequently providing little value back to the community
* How likely will the product remain viable as a community-supported OSS if the company shuts down?
* Note also that **clouds offer their own managed OSS products**
* If a cloud vendor sees traction w/ a particular product or project, expect that vendor to offer its version
* Can range from simple examples (open-source Linux offered on VMs) to extremely complex managed services (fully managed Kafka)
* Motivation for these offerings is simple: **clouds make their money through consumption**
* **More offerings in a cloud ecosystem mean a greater chance of “stickiness” + 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
* 2 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
* W/ the ability to raise funds from VCs flush w/ capital, these data companies can scale + hire great engineering, sales, + marketing teams
* This presents a situation where **users have some great product choices in the marketplace while having to wade through endless sales + marketing clutter**
* **As of 2022-2023, 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 + 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, w/ 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 + 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 + post questions on social networks about users’ experiences w/ the product
* ***Make sure you know what you’re getting into***
* **Cloud platform proprietary service offerings**: **Cloud vendors develop + sell their proprietary services for storage, databases, + more**
* Many of these solutions are internal tools used by respective sibling companies
* Ex: Amazon created DynamoDB to overcome the limitations of traditional RDB’s + handle the large amounts of user + 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 + maturity levels
* **Cloud vendors will often bundle their products to work well together**
* **Each cloud can create stickiness w/ 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?

##### Advice

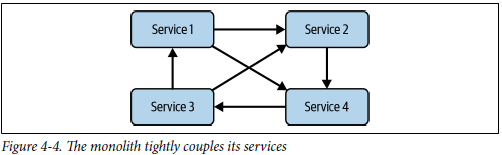
* **Build vs. buy comes back to knowing your competitive advantage + where it makes sense to invest resources toward customization**
* In general, **favor OSS + COSS by default, which frees you to focus on improving those areas where these options are insufficient**
* **Focus on a few areas where building 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-prem servers**
* In addition, think about how a company makes money, especially its sales + customer experience teams, which will generally indicate how you’re treated during the sales cycle + when you’re a paying customer
* Finally, **who is responsible for the budget at your company?**
* How does this person decide the projects + technologies that get funded?
* *Before making business cases for COSS/managed services, does it make sense to try to use OSS?*
* The last thing you want is for your tech choice to be stuck in limbo while waiting for budget approval, since, 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 + what will successfully get approved**

#### Monolith Vs. Modular

* Monoliths vs. modular systems is another longtime debate in the software architecture space
* **Monolithic systems** = **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**, + 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**

##### Monolith

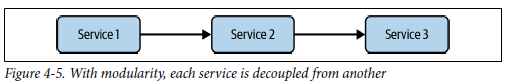
* The **monolith**(see below) has been a technology mainstay for decades



* The old days of **waterfall meant that software releases were huge, tightly coupled, + moved at a slow cadence**
* Large teams worked together to deliver a single working codebase
* Monolithic data systems continue to this day, w/ older software vendors such as Informatica + OSS frameworks such as Spark
* **Pros of the monolith = it’s easy to reason about, + it requires a lower cognitive burden + context switching since everything is self-contained**
* Instead of dealing with dozens of technologies, you deal with “one” technology + typically one principal programming language
* **Monoliths** are an **excellent option if you want simplicity in reasoning about your architecture + processes**
* Of course, the **monolith has cons**
* For one thing, monoliths are **brittle**
* B/c of the **vast number of moving parts**, **updates + 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
* Ex: 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 2 days late by default + 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 DW, one user-defined function might consume enough CPU to slow the system for other users
* Conflicts between dependencies + resource contention are frequent sources of headaches
* **Another con** of monoliths isthat **switching to a new system will be painful if the vendor or OSS project dies**
* B/c all of your processes are contained in the monolith, extracting yourself out of that system, + onto a new platform, will be costly in both time + money

##### Modularity

* **Modularity** (see below) is an old concept in SWE, 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 + 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 SWE + is increasingly seen in modern data systems.
* Major tech companies have been key drivers in the microservices movement
* The famous Bezos API mandate decreases coupling between applications, allowing refactoring + decomposition
* Bezos also imposed the two-pizza rule (no team should be so large that 2 pizzas can’t feed all)
* Effectively, this means that a team will have at most 5 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 100 people, **dividing developers into small groups of 5 requires that this application be broken into small, manageable, loosely coupled pieces**
* In a **modular microservice environment, components are swappable, + it’s possible to create a polyglot(multiprogramming language) application**; a Java service can replace a Python service
* 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 + lakehouses
* Any processing tool that supports the format can read the data + write processed results back into the lake for processing by another tool
* **Cloud DW’s support interoperation w/ object storage through import/export using standard formats and external tables (i.e., queries run directly on data in a data lake)**
* New tech arrives on the scene at a dizzying rate in today’s data ecosystem, + most get stale + outmoded quickly, **so the ability to swap out tools as technology changes is invaluable**
* **Data modularity is a more powerful paradigm than monolithic DE**
* **Modularity allows DE’s to choose the best tech 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 + operate
* **Interoperability is a potential headache**; hopefully, these systems all play nicely together
* This very problem led to breaking 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 DW space
* **But orchestrating 5 or 10 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**
* **Basic idea = one runs a distributed system w/ different services to perform different tasks**
* **Still, services + 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, a cluster runs core Hadoop components: common libraries, HDFS, YARN, + 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 + *all* jobs
* Managing upgrades + installations is a significant challenge
* Forcing jobs to upgrade dependencies risks breaking them, + maintaining 2 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 + asynchronous architecture, **every service runs the same codebase w/ 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**
* **1 solution to problems of the distributed monolith = ephemeral infrastructure in a cloud setting**
* Each job gets its own temporary server or cluster installed w/ dependencies
* Each cluster remains highly monolithic, but separating jobs dramatically reduces conflicts
* This pattern is now quite common for Spark w/ services like Amazon EMR and Google Cloud Dataproc
* **2nd solution = properly decompose a distributed monolith into multiple software environments using containers** (see later on)

##### Advice

* **While monoliths are attractive b/c of ease of understanding + reduced complexity, this comes at a high cost 🡪 the potential loss of flexibility, opportunity cost, + 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 + reversible decisions**

#### Serverless Vs. Servers

* A big trend for cloud providers is **serverless, allowing developers + DE’s to run applications w/out 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*

##### Serverless

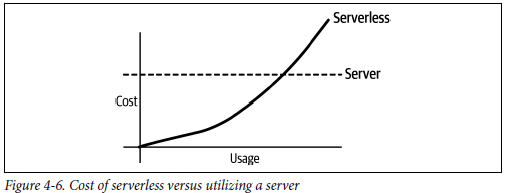
* Though serverless has been around for some time, the trend kicked off w/ **AWS Lambda** in 2014
* W/ the **promise of executing small chunks of code on an as-needed basis without having to manage a server**, serverless exploded in popularity
* The **main reasons for its popularity are cost + convenience**
* **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 AWS Lambda**
* Ex: Google Cloud’s BigQuery is “serverless” in that DE’s don’t need to manage backend infrastructure, + the system scales to zero + scales up automatically to handle large queries
* Just load data into the system, start querying, + you pay for the amount of data your query consumes + 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**, + **DE’s 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
* On the other hand, **serverless functions suffer from an inherent overhead inefficiency**
* **Handling 1 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**
* **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 w/ 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 VM’s**
* **Whereas traditional VMs wrap up an entire OS, a container packages an isolated user space (such as a filesystem + a few processes), + many such containers can coexist on a single host OS**
* This **provides some of the principal benefits of virtualization (i.e., dependency + code isolation) w/out the overhead of carrying around an entire OS kernel**
* A **single hardware node can host numerous containers w/ fine-grained resource allocations**
* As of 2022-2023, containers continue to grow in popularity, along with **K8’s, a container management system**
* **Serverless environments typically run on containers behind the scenes**
* Indeed, **K8’s is a kind of serverless environment b/c it allows developers + ops teams to deploy microservices w/out worrying about the details of the machines where they are deployed**
* **Containers provide a *partial* solution to problems of the distributed monolith mentioned earlier**
* Ex: Hadoop now supports containers, allowing each job to have its own isolated dependencies
* **NOTE: Container clusters do NOT provide the same security + 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 K8s cluster should host code *only w/in an environment of mutual trust* (e.g., inside the walls of a single company)**
* In addition, **code review processes + 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** (Examples include OpenFaaS, Knative, + Google Cloud Run)
* This gives users the simplicity of AWS Lambda w/ 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 w/out managing a compute cluster required for K8s
* These services also fully isolate containers, preventing the security issues associated w/ multitenancy.
* **Abstraction will continue working its way across the data stack**
* Consider the impact of K8s on cluster management
* While you can manage your K8s cluster (many engineering teams do so), even K8s is widely available as a managed service
* What comes after Kubernetes? We’re as excited as you to find out

##### How to Evaluate Serverless Vs. Servers

* **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 + cost exceed the ongoing cost of running + maintaining a server**



* However, **at a certain scale, the economic benefits of serverless may diminish, + running servers becomes more attractive.**
* **Customization, power, + 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 + brittle, as this introduces a glaring vulnerability in your architecture
* Instead, **treat servers as ephemeral resources that you can create as needed + 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 + shrink compute resources on demand**
* **As your application increases its usage, cluster your application servers, + use autoscaling capabilities to automatically horizontally scale (increasing the capacity of a system by adding additional machines (nodes)) your application as demand grows**
* **Treat your infrastructure as code: Automation doesn’t apply to just servers + should extend to your infrastructure whenever possible**
* Deploy your infrastructure (servers or otherwise) using deployment managers such as Terraform, AWS CloudFormation, + Google Cloud Deployment Manager
* **Use containers**
* For more sophisticated or heavy-duty workloads w/ complex installed dependencies, consider using containers on either a single server or K8s

##### Advice

* Some key considerations to help you determine whether serverless is right for you:
* **Workload size and complexity**
* **Serverless works best for simple, discrete tasks + 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 + a container workflow orchestration framework like K8s**
* **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, + duration**
* If your application can’t function neatly w/in these limits, it is time to consider a container-oriented approach
* **Requests and networking**
* **Serverless platforms often utilize some form of simplified networking + don’t support all cloud virtual networking features, such as VPCs + firewalls**
* **Language**
* If the language you typically use is **not one of the officially-supported languages** supported by the serverless platform, **consider containers instead**
* **Runtime limitations**
* **Serverless platforms DON’T give you complete OS 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, but **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** 🡪 **look at using serverless first + then servers (w/ containers + 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: a 787 Business Jet vs. a Tesla Model S Plaid
* 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 + are glad to see many database vendors finally dropping DeWitt clauses from their customer contracts
* Even so, **let the buyer beware: the data space is full of nonsensical benchmarks**
* Next up are a few common tricks used to place a thumb on the benchmark scale

##### Big Data… For the 1990’s

* Products that claim to support “big data” at PB scale will often use benchmark datasets small enough to easily fit in the storage on a smartphone
* For systems that rely on caching layers to deliver performance, test datasets fully reside in solid-state drive (SSD) or memory, + benchmarks can show ultra-high performance by repeatedly querying the same data
* A small test dataset also minimizes RAM + SSD costs when comparing pricing
* **To benchmark for *real-world* use cases, you must simulate anticipated real-world data + query size**
* **Evaluate query performance + 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 + deleted even when they reside in a cloud environment, these systems run for years on + once they’ve been configured
* Other databases support a dynamic compute model + charge either per query or per second of use
* **Comparing ephemeral + nonephemeral 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 w/ some schema changes
* Worse, vendors will juice their systems w/ an extra shot of **join optimization** (e.g., pre-indexing joins) w/out 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 + choose tech**

#### Undercurrents and Their Impacts on Choosing Technologies

* As seen in this chapter, **a** **DE has a lot to consider when evaluating technologies**
* **Whatever tech you choose, be sure to understand how it supports the undercurrents of the DE lifecycle**

##### Data Management

* Data management is a broad area, + concerning technologies, **it isn’t always apparent whether a technology adopts data management as a principal concern**
* For instance, behind the scenes, a 3rd-party vendor may use data management best practices (regulatory compliance, security, privacy, data quality, + 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 + w/in?
* What is your product’s compliance with GDPR, CCPA, + other data privacy regulations?
* Do you allow me to host my data to comply with these regulations?
* How do you ensure data quality + that I’m viewing the correct data in your solution?
* *There are many other questions to ask*, + 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/database may die, a cloud’s region may have an outage, you might deploy buggy code, bad data might be introduced into your DW, + 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, + 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, + 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, + 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 + choosing the best tools for the job while keeping your decisions reversible**
* W/ 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**

##### Orchestration Example: Airflow

* We have actively avoided discussing any particular technology too extensively, but will make an exception for orchestration because the space is currently dominated by one OSS 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 OSS marketplace
* 1) The Airflow OSS project is extremely active, w/ a high rate of commits + a quick response time for bugs + security issues, + the project recently released Airflow 2, a major refactor of the codebase
* 2) Airflow enjoys massive mindshare, w/ a vibrant, active community on many communications platforms, including Slack, Stack Overflow, + GitHub
* Users can easily find answers to questions and problems
* 3) Airflow is available commercially as a managed service or software distribution through many vendors, including GCP, AWS, + Astronomer.io
* Airflow also has some downsides
* It relies on a **few core non-scalable components** (the scheduler + backend database) that can become bottlenecks for performance, scale, + reliability (the scalable parts of Airflow still follow a distributed monolith pattern)
* Airflow **lacks support for many data-native constructs**, such as schema management, lineage, + cataloging
* Also it is **challenging to develop and test Airflow workflows**
* A couple Airflow alternatives/key orchestration contenders as of 2022-2023:
* Prefect and Dagster aim to solve some of the problems discussed previously by rethinking components of the Airflow architecture
* **Anyone choosing an orchestration technology should 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 DE, you should strive for simplification + abstraction across the data stack**
* **Buy or use pre-built OSS solutions whenever possible**
* **Eliminating undifferentiated heavy lifting should be your big goal**
* **Focus resources (custom coding + tooling) on areas that give you a solid competitive advantage**
* Ex: Is hand-coding a database connection between your production database + your cloud DW a competitive advantage for you? *Probably not*
* This is very much a solved problem, so pick an off-the-shelf solution (OSS 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 + processes, you can continue chipping away, refining, + customizing the things that move the needle for the business**

#### Conclusion

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