# Fundamentals of Data Engineering - Reis & Housley

## Part I. Foundation and Building Blocks

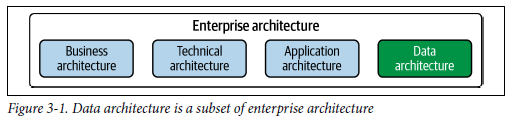
### Chapter 3 – Designing Good Data Architecture

#### What is Data Architecture?

* **Good data architecture provides seamless capabilities across every step of the data lifecycle + undercurrents**
* **Successful data engineering is built upon rock-solid data architecture**
* *What is data architecture?* When you stop to unpack it, the topic becomes a bit murky; researching data architecture yields many inconsistent + often outdated definitions
* It’s a lot like defining data engineering*,* there’s no consensus (In a field that is constantly changing, this is to be expected)

##### Enterprise Architecture Defined

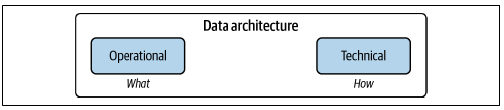
* **Enterprise architecture** has **many subsets, including business, technical, application, and data**



* As such, many frameworks + resources are devoted to enterprise architecture
* In truth, architecture is a surprisingly controversial topic
* The term “*enterprise”* gets mixed reactions 🡪 brings to mind sterile corporate offices, command-and-control/waterfall planning, stagnant business cultures, + empty catchphrases
* Even so, we can learn some things here.
* Before we define + describe enterprise architecture, let’s unpack this term + look at how enterprise architecture is defined by some significant thought leaders: TOGAF, Gartner, and EABOK
* TOGAF’s definition
* The Open Group Architecture Framework, a standard of The Open Group 🡪 touted as the most widely-used architecture framework today
* TOGAF enterprise architecture definition: *The term “enterprise” in the context of “enterprise architecture” can denote an entire enterprise (encompassing all of its information + technology services, processes, + infrastructure) or a specific domain w/in the enterprise. In both cases, the architecture crosses multiple systems, + multiple functional groups w/in the enterprise*
* Gartner’s definition
* Gartner = a global research + advisory company that produces research articles + reports on trends related to enterprises
* Among other things, it is responsible for the (in)famous Gartner Hype Cycle
* Gartner’s definition: *Enterprise architecture (EA) is a discipline for proactively + holistically leading enterprise responses to disruptive forces by identifying + analyzing the execution of change toward desired business vision + outcomes. EA delivers value by presenting business + IT leaders with signature-ready recommendations for adjusting policies + projects to achieve targeted business outcomes that capitalize on relevant business disruptions*
* EABOK’s definition
* Enterprise Architecture Book of Knowledge, an enterprise architecture reference produced by the MITRE Corporation, released as an incomplete draft in 2004 + has not been updated since
* Though seemingly obsolete, EABOK is frequently referenced in descriptions of enterprise architecture
* EABOK definition: *Enterprise Architecture (EA) is an organizational model; an abstract representation of an Enterprise that aligns strategy, operations, + technology to create a roadmap for success*
* Book’s definition
* We extract **a few common threads** in these definitions of enterprise architecture: **change, alignment, organization, opportunities, problem-solving, + migration**
* A more relevant to today’s fast-moving data landscape definition: ***Enterprise architecture is the design of systems to support change in the enterprise, achieved by flexible and reversible decisions reached through careful evaluation of trade-offs***
* Here, we touch on some key areas we’ll return to as we go on: **flexible + reversible decisions**, **change management**, + **evaluation of trade-offs**:
* **Flexible and reversible decisions** are **essential for 2 reasons**
* 1) The world is constantly changing, + predicting the future is impossible
* **Reversible decisions allow you to adjust course as the world changes + you gather new information**
* 2) There is a natural tendency toward enterprise ossification as organizations grow
* Adopting **a culture of reversible decisions** helps overcome this tendency by **reducing the risk attached to a decision**
* Bezos is credited w/ the idea of one-way and two-way doors
* **One-way door = a decision that’s almost impossible to reverse**
* Ex: Amazon could have decided to sell AWS or shut it down, + it would be nearly impossible for them to rebuild a public cloud w/ the same market position after such an action
* **Two-way door = an easily reversible decision** 🡪 walk through + proceed if you like what you see in the room or step back through the door if you don’t
* Ex: Amazon might decide to require the use of DynamoDB for a new microservices database
* If this policy doesn’t work, Amazon has the option of reversing it + refactoring some services to use other databases
* **Since the stakes attached to each reversible decision (two-way door) are low, organizations can make more decisions, iterating, improving, + collecting data rapidly**
* **Change management** is **closely related to reversible decisions** + is a central theme of enterprise architecture frameworks
* **Even w/ an emphasis on reversible decisions, enterprises often need to undertake large initiatives**
* These are **ideally broken into smaller changes, each one a reversible decision in itself**
* Returning to Amazon, note a 5-year gap (2007-2012) from the publication of a paper on the DynamoDB concept to Werner Vogels’s announcement of the DynamoDB service on AWS
* Behind the scenes, teams took numerous small actions to make DynamoDB a concrete reality for AWS customers
* **Managing such small actions is at the heart of change management.**
* **Architects** are not simply mapping out IT processes + vaguely looking toward a distant, utopian future 🡪 they ***actively* solve business problems + create new opportunities**
* **Technical solutions exist not for their own sake but in support of business goals**
* **Architects identify problems in the *current* state** (poor data quality, scalability limits, money-losing lines of business), **define *desired future* states** (agile data-quality improvement, scalable cloud data solutions, improved business processes), **+ realize initiatives through execution of small, concrete steps**
* It bears repeating: ***Technical solutions exist not for their own sake but in support of business goals.***
* For **tradeoff’s**, there’s significant inspiration in Fundamentals of Software Architectureby Mark Richards and Neal Ford (O’Reilly)
* They emphasize that **trade-offs are inevitable + ubiquitous in the engineering space**
* Sometimes the relatively fluid nature of software + data leads us to believe that we are freed from the constraints that engineers face in the hard, cold physical world
* Indeed, this is partially true; patching a software bug is much easier than redesigning + replacing an airplane wing
* However, **digital systems are ultimately constrained by physical limits such as latency, reliability, density, + energy consumption**
* **Engineers also confront various nonphysical limits, such as characteristics of programming languages + frameworks, and practical constraints in managing complexity, budgets, etc.**
* **Magical thinking culminates in poor engineering**
* **DE’s *must* account for trade-offs at *every* step to design an optimal system while minimizing high-interest tech debt**
* Let’s reiterate 1 central point in our enterprise architecture definition: **enterprise architecture balances flexibility and trade-offs**
* This **isn’t always an easy balance**, + architects must constantly assess + reevaluate with the recognition that the world is dynamic
* Given the pace of change that enterprises are faced with, organizations (+ their architecture) cannot afford to stand still

##### Data Architecture Defined

* Now that you understand *enterprise* architecture, let’s dive into **data architecture** by establishing a working definition that will set the stage for the rest of the book
* **Data architecture** = a **subset of enterprise architecture, inheriting its properties: processes, strategy, change management, and evaluating trade-offs**
* There are a couple of definitions of data architecture that influence our definition
* TOGAF’s data architecture definition:
* *A description of the* ***structure + interaction*** *of the enterprise’s major types + sources of data, logical data assets, physical data assets, + data management resources*
* DAMA’s DMBOKdefinition:
* *Identifying the data needs of the enterprise (regardless of structure) and designing + maintaining the* ***master blueprints*** *to meet those needs + to guide data integration, control data assets, + align data investments with business strategy*
* Book definition
* Considering the preceding two definitions and todays’ experience: ***Data architecture is the design of systems to support the evolving data needs of an enterprise, achieved by flexible + reversible decisions reached through a careful evaluation of trade-offs***
* *How does data architecture fit into DE?*
* Just as the DE lifecycle is a subset of the data lifecycle, **DE architecture** is a **subset of general data architecture**
* **Data engineering (DE) architecture** = **the systems + frameworks that make up the key sections of the DE lifecycle**
* We’ll use data architectureinterchangeably with data engineering architectureas we go on
* Other aspects of data architecture that you should be aware of are **operational + technical**



* **Operational architecture**encompasses the **functional requirements of what needs to happen related to people, processes, + tech**
* Ex: What business processes does the data serve? How does the organization manage data quality? What is the latency requirement from when the data is produced to when it becomes available to query?
* **Technical architecture**outlines **how data is ingested, stored, transformed, + served along the DE lifecycle**
* Ex: How will you move 10 TB of data every hour from a source database to your data lake?
* In short, **operational architecture describes *what* needs to be done, + technical architecture details *how* it will happen**

##### “Good” Data Architecture

* According to Grady Booch, “Architecture represents the significant design decisions that shape a system, where *significant* is measured by **cost of change**”
* **Data architects aim to make significant decisions that will lead to “good” architecture at a basic level**
* What do we mean by “good” data architecture? To paraphrase an old cliche, *you know good when you see it*
* **Good data architectureserves business requirements w/ a common, widely reusable set of building blocks while maintaining flexibility + making appropriate trade-offs**
* **Agility** is the **foundation for good data architecture**, as it acknowledges that the world is fluid
* **Good data architecture is flexible and easily maintainable**
* It **evolves in response to changes w/in the business + new tech + practices that may unlock even more value in the future**
* Businesses + their use cases for data are *always* evolving, the world is dynamic, + the pace of change in the data space is accelerating
* Last year’s data architecture that served you well might not be sufficient for today, let alone next year
* ***Bad* data architecture is authoritarian + tries to cram a bunch of one-size-fits-all decisions into a big ball of mud** *(a casually, even haphazardly, structured system*, <http://www.laputan.org/mud/>)
* Bad data architecture is tightly coupled, rigid, overly centralized, or uses the wrong tools for the job, hampering development + change management
* **Ideally, by designing architecture with reversibility in mind, changes will be less costly.**
* **The undercurrents of the DE lifecycle form the foundation of good data architecture for companies at *any* stage of data maturity**
* Again, these undercurrents are: security, data management, DataOps, data architecture, orchestration, + SWE
* Good data architecture is a living, breathing thing, + it’s **never finished**
* In fact, **change + evolution are *central* to the meaning + purpose of data architecture**

#### Principles of Good Data Architecture

* This section takes a 10,000-foot view of good architecture by focusing on **principles** = **key ideas useful in evaluating major architectural decisions and practices**
* We borrow inspiration for these architecture principles from several sources, especially the AWS Well-Architected Framework and Google Cloud’s Five Principles for Cloud-Native Architecture
* The **AWS Well-Architected Framework** consists of 6 pillars:
* **1) Operational excellence**
* **2) Security**
* **3) Reliability**
* **4) Performance efficiency**
* **5) Cost optimization**
* **6) Sustainability**
* <https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html>
* **Google Cloud’s Five Principles for Cloud-Native Architecture** are as follows:
* **1) Design for automation**
* **2) Be smart with state**
* **3) Favor managed services**
* **4) Practice defense in depth**
* **5) *Always* be architecting**
* <https://cloud.google.com/blog/products/application-development/5-principles-for-cloud-native-architecture-what-it-is-and-how-to-master-it>
* Carefully study both frameworks, identify valuable ideas, + determine points of disagreement
* We now will expand/elaborate on these pillars with these **Principles of DE architecture**:
* **1. Choose common components wisely**
* **2. Plan for failure**
* **3. Architect for scalability**
* **4. Architecture is leadership**
* **5. *Always* be architecting**
* **6. Build loosely coupled systems**
* **7. Make reversible decisions**
* **8. Prioritize security**
* **9. Embrace FinOps**

##### Choose Common Component Wisely

* **One of the primary jobs of a DE is to choose common components + practices that can be used widely across an organization**
* When architects choose well + lead effectively, **common components become a fabric facilitating team collaboration + breaking down silos**
* Common components **enable agility w/in + across teams in conjunction w/ shared knowledge + skills**
* Common components **can be anything that has broad applicability w/in an organization**
* Including **object storage, version-control systems, observability, monitoring + orchestration systems, + processing engines**
* Common components **should be accessible to everyone w/ an appropriate use case, + teams are encouraged to rely on common components already in use rather than reinventing the wheel**
* Common components **must support robust permissions + security to enable sharing of assets among teams while preventing unauthorized access**
* **Cloud platforms are an ideal place to adopt common components**
* Ex: Compute + storage separation in cloud data systems allows users to access a shared storage layer (most commonly object storage) using specialized tools to access + query the data needed for specific use cases
* **Choosing common components is a balancing act**
* On the one hand, you **need to focus on needs across the DE lifecycle + teams, utilize common components that will be useful for individual projects, + simultaneously facilitate interoperation + collaboration**
* On the other hand, **architects should *avoid* decisions that will hamper the productivity of engineers working on domain-specific problems by forcing them into one-size-fits-all tech solutions**

##### Plan For Failure

* “Everything fails, all the time” —Werner Vogels, CTO of AWS
* Modern hardware is highly robust + durable, but even so, **any hardware component will fail, given enough time**
* **To build highly robust data systems, you *must* consider failures in your designs**
* Here are a few key terms for evaluating failure scenarios;
* **Availability** = % of time an IT service or component is in an operable state.
* **Reliability** = system’s probability of meeting defined standards in performing its intended function during a specified interval
* **Recovery time objective** = maximum acceptable time for a service or system outage
* The recovery time objective (RTO) is generally set by determining the business impact of an outage
* An RTO of 1 day might be fine for an internal reporting system
* A website outage of just 5 minutes could have a significant adverse business impact on an online retailer
* **Recovery point objective** = The acceptable state after recovery
* In data systems, data is often lost during an outage
* In this setting, the recovery point objective (RPO) refers to maximum acceptable data loss
* **Engineers need to consider acceptable reliability, availability, RTO, and RPO in designing for failure**, as these will guide their architecture decisions as they assess possible failure scenarios.

##### Architect For Scalability

* **Scalability** **in data systems encompasses 2 main capabilities**
* **1) Scalable systems can scale upto handle significant quantities of data**
* Might need to spin up a large cluster to train a model on a PB of customer data or scale out a streaming ingestion system to handle a transient load spike
* Our ability to scale up allows us to handle extreme loads temporarily
* **2) Scalable systems can scale down**
* Once the load spike ebbs, we should automatically remove capacity to cut costs (This is related to principle 9)
* An **elastic systemcan scale dynamically in response to load, ideally in an automated fashion**
* Some scalable systems can also **scale to zero**: they **shut down completely when not in use**
* Once a large model-training job completes, we can delete the cluster
* **Many serverless systems** (e.g., serverless functions + serverless OLAP databases) **can automatically scale to zero**
* Note that **deploying inappropriate scaling strategies can result in overcomplicated systems + high costs**
* A straightforward RDB w/ one failover node may be appropriate for an application instead of a complex cluster arrangement
* **Measure your current load, approximate load spikes, and estimate load over the next several years to determine if your database architecture is appropriate**
* If your startup grows much faster than anticipated, this growth should also lead to more available resources to rearchitect for scalability

##### Architecture is Leadership

* **Data architects are responsible for technology decisions + architecture descriptions + disseminating these choices through effective leadership + training**
* Data architects **should be highly technically competent but delegate most individual contributor (IC) work to others**
* **Strong leadership skills combined w/ high technical competence are rare + extremely valuable**, + the best data architects take this duality seriously
* Note that leadership **does** **NOT imply a command-and-control approach to tech**
* It was not uncommon in the past for architects to choose one proprietary database technology + force *every* team to house their data there
* Oppose this approach because **it can significantly hinder current data projects**
* **Cloud environments allow architects to balance common component choices w/ flexibility that enables innovation w/in projects**
* Returning to the notion of technical leadership, Martin Fowler describes a specific archetype of an ideal software architect, well embodied in his colleague Dave Rice:
* *“In many ways, the most important activity of Architectus Oryzus is to* ***mentor the development team, to raise their level so they can take on more complex issues****. Improving the development team’s* ***ability gives an architect much greater leverage than being the sole decision-maker + thus running the risk of being an architectural bottleneck****”*
* An **ideal data architect** manifests similar characteristics:
* **Possess the tech skills of a DE but no longer practice data engineering day to day**
* **Mentor current DE’s, make careful tech choices in consultation w/ their organization, + disseminate expertise through training + leadership**
* **Train engineers in best practices + bring the company’s engineering resources together to pursue common goals in both tech + business.**
* **As a DE, practice architecture leadership + seek mentorship from architects**
* *Eventually, you may well occupy the architect role yourself*

##### Always Be Architecting

* Borrowed directly from Google Cloud’s Five Principles for Cloud-Native Architecture
* **Data architects** don’t serve in their role simply to maintain the existing state; instead, they **constantly design new and exciting things in response to changes in business + tech**
* Per the EABOK, **an architect’s job is to develop deep knowledge of the baseline architecture(current state), develop a target architecture, and map out a sequencing planto determine priorities + the order of architecture changes**
* We add that **modern architecture should not be command-and-control or waterfall but collaborative and agile**
* The data architect maintains a target architecture + sequencing plans that change over time
* The **target architecture becomes a *moving* target, adjusted in response to business + tech changes internally and worldwide**
* The **sequencing plan determines immediate priorities for delivery**

##### Build Loosely-Coupled Systems

* *When the architecture of the system is designed to* ***enable teams to test, deploy, + change systems w/out dependencies on other teams****, teams* ***require little communication to get work done****. In other words, both the architecture + the teams are* ***loosely-coupled*** — Google DevOps tech architecture guide (<https://cloud.google.com/architecture/devops/devops-tech-architecture>)
* In 2002, Bezos wrote an email to Amazon employees that became known as the **Bezos API Mandate**:
* 1. **All teams** will henceforth **expose their data + functionality through service interfaces**
* 2. Teams **must communicate with each other through these interfaces**
* 3. There will be **no other form of interprocess communication allowed**: no direct linking, no direct reads of another team’s data store, no shared-memory model, no back-doors whatsoever
* The **only communication allowed is via service interface calls over the network**
* 4. It **doesn’t matter what tech they use**
* HTTP, Corba, Pubsub, custom protocols, doesn’t matter
* 5. **All service interfaces, w/out exception, *must* be designed from the ground up to be externalizable**
* That is to say, the team must plan + design to be able to expose the interface to developers in the outside world. No exceptions.
* The advent of Bezos’s API Mandate is widely viewed as a watershed moment for Amazon, as putting data + services behind APIs enabled loose coupling + eventually resulted in AWS as we know it now
* Google’s pursuit of loose coupling allowed it to grow its systems to an extraordinary scale
* **For software architecture, a loosely coupled system has the following properties**:
* **1. Systems are broken into many small components**
* **2.** These **systems interface with other services through abstraction layers**, such as a messaging bus or an API
* These **abstraction layers hide + protect internal details of the service**, such as a database backend or internal classes + method calls
* **3.** As a consequence of property 2, **internal changes to a system component don’t require changes in other parts**
* **Details** of code updates are **hidden behind stable APIs**
* **Each piece can evolve + improve separately**
* **4.** As a consequence of property 3, **there is no waterfall, global release cycle for the whole system**
* Instead, **each component is updated separately as changes and improvements are made**
* Notice that we are talking about *technical systems,* butwe need to think bigger
* Let’s **translate these technical characteristics into organizational characteristics**:
* **1. Many small teams engineer a large, complex system**
* Each team is tasked w/ engineering, maintaining, + improving *some* system components
* **2. These teams publish the abstract details of their components to other teams** via API definitions, message schemas, etc.
* Teams need not concern themselves w/ other teams’ components; they simply use the published API or message specifications to call these components
* They **iterate *their* part to improve *their* performance + capabilities over time** + might also publish new capabilities as they are added or request new stuff from other teams
* Again, the latter happens w/out teams needing to worry about the internal technical details of the requested features
* Teams work together through **loosely coupled communication**.
* **3.** As a consequence of characteristic 2, **each team can rapidly evolve and improve its component independently of the work of other teams**
* **4.** Specifically, characteristic 3 implies that **teams can release updates to their components w/ minimal downtime**
* Teams release continuously during regular working hours to make code changes + test them
* **Loose coupling of both tech + human systems will allow your DE teams to more efficiently collaborate w/ one another + w/ other parts of the company**
* This principle also directly facilitates principle 7 below

##### Make Reversible Decisions

* The data landscape is changing rapidly 🡪 Today’s hot tech or stack is tomorrow’s afterthought, + popular opinion shifts quickly
* **Aim for reversible decisions, as these tend to simplify your architecture + keep it agile**
* As Fowler wrote, “One of an architect’s most important tasks is to *remove* architecture by finding ways to eliminate irreversibility in software designs”
* What was true when Fowler wrote this in 2003 is just as accurate today
* Bezos refers to reversible decisions as “two-way doors” 🡪 “If you walk through and don’t like what you see on the other side, you can’t get back to before. We can call these Type 1 decisions. But most decisions aren’t like that, they are changeable, reversible, they’re two-way doors”
* **Aim for two-way doors whenever possible**
* **Given the pace of change (+the decoupling/modularization of technologies across your data architecture), always strive to pick the best-of-breed solutions that work for *today***
* Also, **be prepared to upgrade or adopt better practices as the landscape evolves**

##### Prioritize Security

* **Every DE must assume responsibility for the security of the systems they build + maintain**
* **2 main ideas:** **zero-trust security** and the **shared responsibility security model**, which both align closely to a cloud-native architecture

###### A) Hardened-Perimeter and Zero-Trust Security Models

* To define **zero-trust security**, it’s helpful to **start by understanding the traditional**

**hard-perimeter security model + its limitations**, as detailed in Google Cloud’s Five Principles:

* *Traditional architectures place a lot of faith in* ***perimeter security****, crudely a hardened network perimeter with “trusted things” inside and “untrusted things” outside. Unfortunately, this approach has always been vulnerable to insider attacks, as well as external threats such as spear phishing*
* The 1996 film *Mission Impossible* presents a perfect example of the hard-perimeter security model and its limitations:
* The CIA hosts highly sensitive data on a storage system inside a room w/ extremely tight physical security
* Ethan Hunt infiltrates CIA headquarters + exploits a human target to gain physical access to the storage system
* Once inside the secure room, he can exfiltrate data with relative ease
* For at least a decade, alarming media reports have made us aware of the growing menace of security breaches that exploit human targets inside hardened organizational security perimeters
* Even as employees work on highly secure corporate networks, they remain connected to the outside world through email + mobile devices
* External threats effectively become internal threats
* **In a cloud-native environment, the notion of a hardened perimeter erodes further**
* ***ALL* assets are connected to the outside world to *some* degree**
* While **virtual private cloud (VPC) networks *can* be defined with no external connectivity, the API control plane that engineers use to define these networks still faces the internet**

###### B) The Shared Responsibility Model

* Amazon emphasizes the **shared responsibility model**, which **divides security into the security *of* the cloud and security *in* the cloud**, where AWS is responsible for the security *of*the cloud:
* *“AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you w/ services that you can use securely”*
* AWS users are responsible for security *in* the cloud:
* *“Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your organization’s requirements, + applicable laws + regulations”*
* **In general, all cloud providers operate on some form of this shared responsibility model**
* They **secure their services according to published specifications**
* Still, it is **ultimately the user’s responsibility to design a security model for their applications + data and leverage cloud capabilities to realize this model**

###### C) Data Engineers as Security Engineers

* In the **corporate world today, a command-and-control approach to security is quite common, wherein security + networking teams manage perimeters + general security practices**
* The **cloud pushes this responsibility out to engineers who are *not* explicitly in security roles**
* Because of this responsibility, in conjunction with more general erosion of the hard security perimeter, ***all* DE’s should consider themselves security engineers.**
* Failure to assume these new implicit responsibilities can lead to dire consequences
* Numerous data breaches have resulted from the simple error of configuring Amazon S3 buckets w/ public access
* **Those who handle data must assume that they are ultimately responsible for securing it**

##### Embrace FinOps

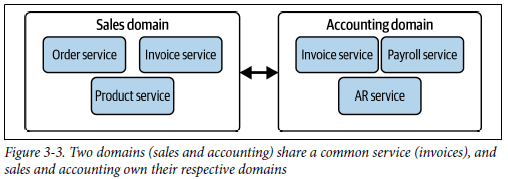
* Let’s start by considering a couple of definitions of **FinOps**
* First, the **FinOps Foundation** offers this:
* *“FinOps is an evolving* ***cloud financial management discipline + cultural practice*** *that enables organizations to get maximum business value by helping engineering, finance, tech, + business teams to collaborate on data-driven spending decisions”*
* In addition, J. R. Sorment and Mike Fuller provide the following definition in “*Cloud FinOps”:*
* *“The term “FinOps” typically refers to the emerging professional movement that advocates a* ***collaborative working relationship between DevOps + Finance****, resulting in* ***an iterative, data-driven management of infrastructure spending*** *(i.e., lowering the unit economics of cloud)* ***while simultaneously increasing the cost efficiency and, ultimately, the profitability of the cloud environment”***
* The **cost structure of data has evolved dramatically during the cloud era**
* In an **on-prem** setting, data systems are generally acquired w/ a **capital expenditure** (see Chapter 4) for a new system **every few years**
* Responsible parties have to **balance their budget against desired compute + storage capacity**
* *Overbuying* entails wasted money, while *underbuying* means hampering future data projects + driving significant personnel time to control system load + data size (+ may require faster technology refresh cycles, w/ associated extra costs)
* **In the cloud era, most data systems are pay-as-you-go and readily scalable**
* Systems can run on a **cost-per-query model**, **cost-per-processing-capacity model**, or **another variant of a pay-as-you-go model**
* This approach **can be far more efficient than the capital expenditure approach**
* It is now **possible to scale up for high performance, then scale down to save money**
* However, the **pay-as-you-go approach makes spending far more dynamic**
* **New challenge for data leaders = to manage *budgets, priorities, + efficiency***
* **Cloud tooling necessitates a set of processes for managing spending and resources**
* In the past, DE’s thought in terms of **performance engineering = maximizing the performance for data processes on a *fixed* set of resources and buying adequate resources for future needs**
* **W/ FinOps, engineers need to learn to think about the cost structures of *cloud systems***
* Ex:What is the appropriate mix of AWS spot instances when running a distributed cluster? What is the most appropriate approach for running a sizable daily job in terms of cost-effectiveness + performance? When should the company switch from a pay-per-query model to reserved capacity?
* **FinOps evolves the operational monitoring model to monitor spending on an ongoing basis**
* Rather than simply monitor requests + CPU utilization for a web server, **FinOps might monitor the ongoing cost of serverless functions handling traffic, as well as spikes in spending trigger alerts**
* **Just as systems are designed to fail gracefully in excessive traffic, companies may consider adopting hard limits for spending, w/ graceful failure modes in response to spending spikes**
* **Ops teams should also think in terms of cost attacks**
* Just as a distributed denial-of-service (DDoS) attack can block access to a web server, many companies have discovered to their chagrin that excessive downloads from S3 buckets can drive spending through the roof + threaten a small startup w/ bankruptcy
* **When sharing data publicly, data teams can address these issues by setting requester-pays policies, or simply monitoring for excessive data access spending + quickly removing access if spending begins to rise to unacceptable levels**
* As of **2022-2023, FinOps is a recently formalized practice**
* The FinOps Foundation was started only in 2019
* However, **start thinking about FinOps early, before you encounter high cloud bills**
* Can start your journey with the FinOps Foundation (<https://oreil.ly/4EOIB>) and O’Reilly’s *Cloud FinOps* (<https://www.oreilly.com/library/view/cloud-finops/9781492054610/>)
* Also, **DE’s should involve themselves in the community process of creating FinOps practices for DE (in such a new practice area, a good deal of territory is yet to be mapped out)**

#### Major Architecture Concepts

* If you follow the current trends in data, it seems like new types of data tools + architectures are arriving on the scene every week
* Amidst this flurry of activity, one must not lose sight of the **main goal of all of such architectures: to take data + transform it into something useful for downstream consumption**

##### Domains and Services

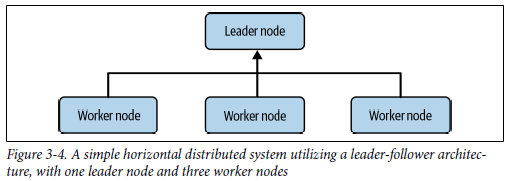
* **“Domain: A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software”** — Eric Evans (<https://oreil.ly/pQ9oq>)
* Before diving into the *components* of the architecture, let’s briefly cover 2 common terms: **domain** + **services**
* A **domain** = the **real-world subject area for which you’re architecting**
* A **service** = a **set of functionality whose goal is to accomplish a task**
* Ex: Might have a sales order-processing service whose task is to process orders as they are created
* The sales order-processing service’s *only* job is to process orders; it doesn’t provide other functionality, such as inventory management or updating user profiles
* **A** **domain can contain *multiple* services**
* Ex: Might have a sales domain with 3 services: orders, invoicing, + products
* **Each service has particular tasks that support the sales domain**
* Other domains may also share services



* In this case, the accounting domain is responsible for basic accounting functions: invoicing, payroll, + accounts receivable (AR)
* Notice the **accounting domain *shares* the invoice service with the sales domain** since a sale generates an invoice, + accounting must keep track of invoices to ensure payment is received
* **Sales and accounting *own* their respective domains**
* When thinking about **what constitutes a domain, focus on what the domain represents in the real world and work backward**
* In the preceding example, the sales domain should represent what happens w/ the sales function in your company
* When architecting the sales domain, **avoid cookie-cutter copying + pasting from what other companies do**
* Your company’s sales function likely has unique aspects that require specific services to make it work the way your sales team expects
* **Identify what should go in the domain**
* When determining what the domain should encompass + what services to include, the best advice is to **simply go + talk w/ users + stakeholders**, listen to what they’re saying, + build the services that will help them do their job
* **Avoid the classic trap of architecting in a vacuum**

##### Distributed Systems, Scalability, and Designing for the Future

* The next discussion is related to our 2nd and 3rd principles of DE architecture: **plan for failure** and **architect for scalability**.
* **As DE’s, we’re interested in 4 closely related characteristics of data systems**
* **Scalability**: **Allows us to increase the capacity of a system to improve performance + handle demand**
* Ex: Might want to scale a system to handle a high rate of queries or process a huge data set
* **Elasticity**: **The ability of a scalable system to scale dynamically; a highly elastic system can automatically scale *up* + *down* based on the current workload**
* **Scaling up = critical as demand increases, while scaling down saves money in a cloud environment**
* Modern systems sometimes **scale to zero** = can **automatically shut down when idle**
* **Availability**: The **% of time an IT service or component is in an operable state**
* **Reliability**: The **system’s probability of meeting defined standards in performing its intended function during a specified interval**
* *How are these characteristics related?*
* If a system fails to meet performance requirements during a specified interval, it may become unresponsive
* **Thus, low reliability can lead to low availability**
* On the other hand, dynamic scaling helps ensure adequate performance w/out manual intervention from engineers (**elasticity improves reliability**)
* **Scalability can be realized in a variety of ways**
* For your services + domains, does a *single machine* handle everything?
* A single machine can be scaled vertically 🡪 can increase resources (CPU, disk, memory, I/O)
* But there are **hard limits to possible resources on a single machine**
* *Also, what happens if this machine dies?*
* **Given enough time, some components will eventually fail**
* **What’s your plan for backup + failover?**
* **Single machines generally can’t offer high availability and reliability.**
* We **utilize a** **distributed system to realize higher overall scaling capacity + increased availability + reliability**
* **Horizontal scaling**allows you to add more machines to satisfy load + resource requirements



* **Common horizontally-scaled systems have a leader node that acts as the main point of contact for the instantiation, progress, + completion of workloads**
* When a workload is started, the **leader node distributes tasks to the worker nodes w/in its system, completing the tasks, + returning the results to the leader node**
* **Typical modern distributed architectures also build in redundancy**
* **Data is replicated so that if a machine dies, the other machines can pick up where the missing server left off; the cluster may add more machines to restore capacity**
* Distributed systems are widespread in the various data technologies you’ll use across your architecture
* **Almost every cloud DW object storage system you use has some notion of distribution under the hood**
* **Management details of the distributed system are typically abstracted away, allowing you to focus on high-level architecture instead of low-level plumbing**
* ***However*, it’s highly recommended that you learn more about distributed systems because these details can be extremely helpful in understanding + improving the performance of pipelines**
* **Martin Kleppmann’s *Designing Data-Intensive Applications* (O’Reilly) = excellent resource**

##### Tight vs. Loose Coupling: Tiers, Monoliths, and Microservices

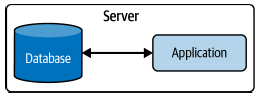
* **When designing a data architecture, you choose how much interdependence you want to include w/in your various domains, services, + resources**
* On one end of the spectrum, you can choose to have **extremely centralized dependencies + workflows (tightly coupled)**
* **Every part of a domain + service is vitally dependent upon *every other* domain + service**
* On the other end of the spectrum, you have **decentralized domains + services that do NOT have strict dependence on each other**, in a pattern known as **loose coupling**
* In a **loosely coupled** scenario, it’s **easy for decentralized teams to build systems whose data may not be usable by their peers**
* **Be sure to assign common standards, ownership, responsibility, + accountability to the teams owning their respective domains + services**
* **Designing “good” data architecture relies on trade-offs between the tight + loose coupling of domains and services**
* It’s worth noting that **many of the ideas in this section originate in SWE**.

###### a) Architecture tiers

* As you develop your architecture, it helps to be aware of **architecture tiers**
* An **architecture has layers (data, application, business logic, presentation, + so forth), + you need to know how to decouple these layers**
* **Because tight coupling of modalities presents obvious vulnerabilities, keep in mind how you structure the layers of your architecture to achieve maximum reliability + flexibility**

*i)* Single tier

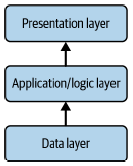
* **Single-tier architecture: a database and application = tightly coupled, residing on a single server**



* This server could be your laptop or a single VM in the cloud
* The **tightly coupled nature means if the server, database, *or* application fails, the entire architecture fails**
* While **single-tier architectures = good for prototyping + development, they are NOT advised for production environments because of the obvious failure risks**
* Even when single-tier architectures build in redundancy (for example, a failover replica), they present significant limitations in other ways
* Ex: It is often impractical (+ not advisable) to run analytics queries against production application databases
* Doing so risks overwhelming the database and causing the application to become unavailable
* **A single-tier architecture is fine for testing systems on a local machine but is not advised for production uses**

ii) Multitier

* The challenges of a tightly coupled single-tier architecture are solved by **decoupling the data and application**
* A **multitier(AKA n-tier) architecture** is **composed of separate layers: data, application, business logic, presentation, etc.**
* These layers are **bottom-up** + **hierarchical**, meaning the **lower layer isn’t necessarily dependent on the upper layers; the upper layers depend on the lower layers**
* The **notion is to separate data from the application, + application from the presentation**
* A **common multitier architecture is a 3-tier architecture, a widely used client-server design**
* A **3-tier architecture**consists of **data**, **application** **logic**, and **presentation tiers**



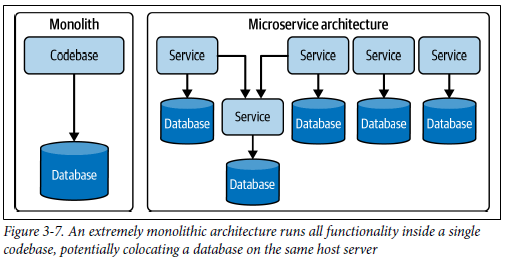
* **Each tier is isolated from the other, allowing for separation of concerns**
* **W/ a three-tier architecture, you’re free to use whatever tech you prefer w/in each tier without the need to be monolithically focused**
* We’ve seen many **single-tier architectures in production**, which **offer simplicity but also severe limitations**
* Eventually, an organization or application outgrows this arrangement; it works well until it doesn’t
* Ex: In a **single-tier architecture**, the **data + logic layers share + compete for resources** (disk, CPU, + memory) in ways that are **simply avoided in a multitier architecture**, where **resources are spread across various tiers**
* **DE’s should use tiers to evaluate their layered architecture + the way dependencies are handled**
* Again, **start simple + bake in evolution to additional tiers as your architecture becomes more complex**
* **In a multitier architecture, you need to consider separating your layers + the way resources are shared w/in layers when working with a distributed system**
* **Under the hood, distributed systems power many technologies you’ll encounter across the DE lifecycle**
* First, think about whether you want **resource contention** w/ your nodes
* If *not*, exercise a **shared-nothing architecture**: a **single node handles each request, meaning other nodes do *not* share resources (memory, disk, CPU, etc. ) w/ this node or w/ each other**
* **Data and resources are isolated to the node**
* *Alternatively*, **various nodes can handle multiple requests + share resources *but at the risk of resource contention***
* Another consideration is whether **nodes should share the same disk + memory accessible by all nodes** 🡪 a **shared disk architecture** = **common when you want shared resources if a random node failure occurs**

iii) Monoliths

* The general notion of a **monolith** includes **as much as possible under one roof**
* In its ***most extreme version***, a **monolith consists of a *single* codebase running on a *single* machine that provides both the application logic + the UI**
* **Coupling w/in monoliths can be viewed in 2 ways**: **technical coupling** and **domain coupling**.
* **Technical coupling**refers to **architectural tiers**, while **domain coupling**refers to **the way domains are coupled together**
* **A monolith has varying degrees of coupling among technologies and domains**
* **Could have an application w/ various layers decoupled in a multitier architecture but still share multiple domains**
* **Or could have a single-tier architecture serving a single domain**
* The **tight coupling of a monolith implies a lack of modularity of its components**
* Swapping out or upgrading components in a monolith is often an exercise in trading one pain for another
* **Because of the tightly coupled nature, reusing components across the architecture is difficult or impossible**
* When evaluating how to improve a monolithic architecture, it often becomes a game of whack-a-mole: **one component is improved, often at the expense of unknown consequences w/ other areas of the monolith**
* Data teams will often ignore solving the growing complexity of their monolith, letting it devolve into a **big ball of mud** *(a casually, even haphazardly, structured system*, <http://www.laputan.org/mud/>)
* NOTE: Chapter 4 provides a more extensive discussion comparing monoliths to distributed technologies
* We also discuss the **distributed monolith**, a strange hybrid that emerges when engineers build distributed systems w/ excessive tight coupling

iv) Microservices

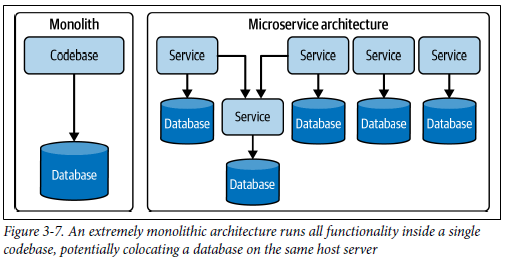
* Compared w/ the **attributes of a monolith (interwoven services, centralization, + tight coupling among services),** **microservices** are the **polar opposite**
* **Microservices architecture** comprises **separate, decentralized, + loosely coupled services**
* **Each service has a specific function + is decoupled from other services operating w/in its domain**
* **If one service temporarily goes down, it won’t affect the ability of other services to continue functioning**
* A question that comes up often is *how to convert your monolith into many microservices*



* **This completely depends on how complex your monolith is + how much effort it will be to start extracting services out of it**
* It’s **entirely possible that a monolith *cannot* be broken apart**, in which case, you **start creating a new parallel architecture w/ the services decoupled in a microservices-friendly manner**
* **It’s not suggested to do an entire refactor but instead break out services**
* The monolith didn’t arrive overnight + is a **technology issue as well as an organizational one**
* **Be sure you get buy-in from stakeholders of the monolith if you plan to break it apart**
* To learn more about breaking apart a monolith, check out the fantastic, pragmatic guide *Software Architecture: The Hard Parts* by Neal Ford et al. (O’Reilly)
* <https://www.oreilly.com/library/view/software-architecture-the/9781492086888/>

###### b) Considerations for data architecture

* As mentioned, the concepts of tight vs. loose coupling stem from SWE, w/ some of these concepts dating back over 20 years
* **Though architectural practices in data are now adopting those from SWE, it’s still common to see very monolithic, tightly-coupled data architectures**
* Some of this is due to the nature of existing data technologies + the way they integrate
* **Ex: Data pipelines might consume data from many sources ingested into a central DW, which is inherently monolithic**
* A move toward a microservices equivalent with a DW is to decouple the workflow w/ domain-specific data pipelines connecting to corresponding domain-specific DW’s
* Ex: The sales data pipeline connects to the sales-specific DW, + the inventory and product domains follow a similar pattern
* **Rather than dogmatically preach microservices over monoliths** (among other arguments), it’s best to **pragmatically use loose coupling as an ideal, while recognizing the state + limitations of the data technologies you’re using w/in your data architecture**
* **Incorporate reversible technology choices that allow for modularity + loose coupling whenever possible**
* As you can see below, you separate the components of your architecture into different layers of concern in a **vertical fashion**



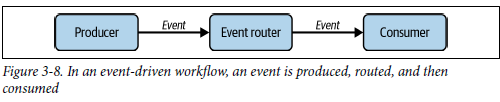
* **While a multitier architecture solves the technical challenges of decoupling shared resources, it does NOT address the complexity of sharing domains**
* **Along the lines of single vs. multitiered architecture, also consider how you separate the domains of your data architecture**
* Ex: Your analyst team might rely on data from sales + inventory
* The sales + inventory domains are different + **should be viewed as separate**
* One approach to this problem is **centralization**: a **single team is responsible for gathering data from all domains + reconciling it for consumption across the organization** (a common approach in traditional data warehousing)
* Another approach is the **data mesh**: **each software team is responsible for preparing its data for consumption across the rest of the organization** (see later in the chapter)
* So, **monoliths aren’t necessarily *bad*, + it might make sense to start w/ one, under certain conditions**
* Sometimes you **need to move fast, + it’s much simpler to start with a monolith**
* **Just be prepared to break it into smaller pieces eventually; don’t get too comfortable**

##### User Access: Single vs. Multitenant

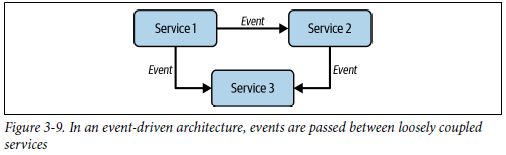
* As a DE, you have to make decisions about **sharing systems across multiple teams, organizations, + customers**
* **In some sense, *all* cloud services are multitenant, although this multitenancy occurs at various grains**
* Ex: A **cloud compute instance is usually on a shared server**, but the **VM** itself provides some degree of **isolation**
* **Object storage is a multitenant system**, but cloud vendors guarantee **security + isolation** so long as customers **configure their permissions correctly**
* **Engineers frequently need to make decisions about multitenancy at a much smaller scale**
* Ex: Do multiple departments in a large company share the same DW?
* Ex: Does the organization share data for multiple large customers w/in the same table?
* We have **2 factors to consider in multitenancy**: **performance** and **security**
* W/ multiple large tenants w/in a cloud system, will the system support consistent performance for all tenants, or will there be a noisy neighbor problem? (That is, **will high usage from one tenant degrade performance for other tenants?**)
* Regarding security, **data from different tenants *must* be properly isolated**
* When a company has multiple external customer tenants, these **tenants should not be aware of one another, + engineers must prevent data leakage**
* **Strategies for data isolation vary by system**
* Ex: It is often perfectly acceptable to use multitenant tables + isolate data through database views
* **However, you must make certain that these views cannot leak data**
* Read vendor or project documentation to understand appropriate strategies + risks

##### Event-Driven Architecture

* Your **business is rarely static** 🡪things often happen in your business, such as getting a new customer, a new order from a customer, or an order for a product or service
* These are all examples of **events**, **broadly defined as something that happened, typically a change in the stateof something**
* Ex: A new order might be created by a customer, or a customer might later make an update to this order
* An **event-driven workflow** encompasses the **ability to create, update, + asynchronously move events across various parts of the DE lifecycle**



* An **event-driven workflow boils down to 3 main areas: event production, routing, + consumption**
* An event must be produced + routed to something that consumes it **w/out tightly-coupled dependencies among the producer, event router, + consumer**
* An **event-driven architecture** **embraces the event-driven workflow + uses this to communicate across various services**



* The **advantage of an event-driven architecture = it distributes the state of an event across multiple services**
* This is helpful if a service goes offline, a node fails in a distributed system, or you’d like multiple consumers or services to access the same events
* **Anytime you have loosely-coupled services, this is a candidate for event-driven architecture**
* Many examples later in this chapter incorporate some form of event-driven architecture
* Note: Learn more about **event-driven streaming** and **messaging** **systems** in Chapter 5

##### Brownfield vs. Greenfield Projects

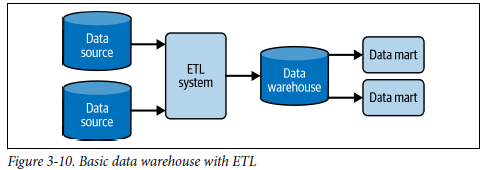
* **Before you design a data architecture project, you need to know whether you’re starting w/ a clean slate or redesigning an existing architecture**
* Each type of project requires assessing trade-offs, albeit w/ different considerations + approaches
* Projects roughly fall into two buckets: **brownfield** and **greenfield**.
* **Brownfield projects**
* Often involve **refactoring** + **reorganizing** an existing architecture + are **constrained by the choices of the present + past**
* B/c **a key part of architecture is change management**, you **must figure out a way around these limitations** + **design a path forward to achieve new business + technical objectives**
* Such projects **require a thorough understanding of the legacy architecture + the interplay of various old and new technologies**
* It’s easy to criticize a prior team’s work + decisions, but it is **far better to dig deep, ask questions, and understand WHY decisions were made**
* Empathy + context go a long way in helping diagnose problems w/ the existing architecture, identify opportunities, + recognize pitfalls
* **Need to introduce your new architecture and technologies + deprecate the old stuff at some point**, so let’s look at a couple of popular approaches
* Many teams jump headfirst into an **all-at-once or big-bang overhaul** of the old architecture, often **figuring out deprecation as they go**
* Though popular, this approach is not advised b/c of the associated **risks + lack of a plan**
* This path **often leads to disaster, w/ many irreversible + costly decisions**, while a **DE’s job is to make reversible, high-ROI decisions**
* A popular alternative to a direct rewrite is the **strangler pattern: new systems slowly + incrementally replace a legacy architecture’s components**
* Eventually, the legacy architecture is completely replaced
* The **attraction to the strangler pattern is its targeted + surgical approach of deprecating 1 piece of a system at a time**
* This allows for **flexible + reversible decisions while assessing the impact of the deprecation on dependent systems**
* It’s **important to note that deprecation might be “ivory tower” advice + not practical or achievable**
* Eradicating legacy technology or architecture might be impossible if you’re at a large organization, since someone, somewhere, is using these legacy components.
* As someone once said, “Legacy is a condescending way to describe something that makes money.”
* **If you *can* deprecate, understand there are numerous ways to deprecate your old architecture**
* **It is critical to demonstrate value on the new platform by gradually increasing its maturity to show evidence of success + then follow an exit plan to shut down old systems**
* **Greenfield projects**
* Allows you to **pioneer a fresh start, unconstrained by the history/legacy of a prior architecture**
* Tend to be easier than brownfield projects, and many data architects + engineers find them more fun
* You have the **opportunity to try the newest + coolest tools + architectural patterns.**
* You should **watch out for some things before getting too carried away**
* Teams can get overly exuberant with shiny object syndrome 🡪 they feel compelled to reach for the latest + greatest tech fad w/out understanding how it will impact the value of the project
* There’s also a temptation to do **resume-driven development** = stacking up impressive new technologies without prioritizing the project’s ultimate goals
* ***Always prioritize requirements over building something cool***
* ***Whether working on a brownfield or greenfield project, always focus on the tenets of “good” data architecture***
* ***Assess trade-offs, make flexible and reversible decisions, + strive for positive ROI***

#### Examples and Types of Data Architecture

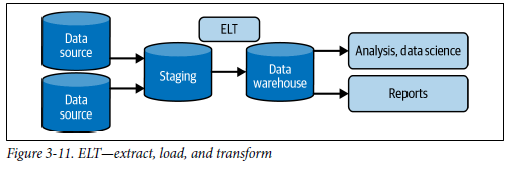
* Because data architecture is an abstract discipline, it helps to reason by example
* We outline prominent examples + types of data architecture popular today
* *Though this set of examples is by no means exhaustive*, the intention is to expose you to some of the most common data architecture patterns + to get you thinking about the requisite flexibility + trade-off analysis needed when designing a good architecture for your use case

##### 1) Data Warehouse

* **Data Warehouse (DW)** = **a central data hub used for reporting + analysis**
* Data in a DW = typically **highly formatted + structured for analytics use cases**
* Among the oldest + most well-established data architectures.
* 1989 🡪 Bill Inmon originated the notion of the DW, described as **“a subject-oriented, integrated, non-volatile, + time-variant collection of data in support of management’s decisions.”**
* Though technical aspects of the DW have evolved significantly, this original definition still holds its weight today
* In the past, DW’s were widely used at enterprises w/ significant budgets (often in the millions of dollars) to acquire data systems + pay internal teams to provide ongoing support to maintain them
* This was **expensive + labor-intensive**
* **Since then, scalable, pay-as-you-go model has made cloud DW’s accessible even to tiny companies**
* Because a 3rd-party provider manages the DW infrastructure, companies can do a lot more w/ fewer people, even as the complexity of their data grows
* 2 types of DW architecture: **organizational** and **technical**
* **Organizational DW Architecture****organizes data associated w/ certain business team structures + processes**
* Organizational DW architecture has 2 main characteristics:
* **Separates online analytical processing (OLAP) from production databases (online transaction processing/OLTP)**
* This separation is **critical as businesses grow**
* **Moving data into a separate physical system directs load away from production systems + improves analytics performance**
* **Centralizes and organizes data**
* Traditionally, a DW **pulls data from application systems by using ETL**
* The extract phase pulls data from source systems, the transformation phase cleans + standardizes data, organizing + imposing business logic in a highly modeled form (see Chapter 8), + the load phase pushes data into the DW target database system
* **Data is loaded into multiple data marts that serve the analytical needs for specific lines or business and departments**
* Here is the general workflow:



* The **DW + ETL go hand in hand w/ specific business structures**, including DBA and ETL developer teams that implement the direction of business leaders to ensure that data for reporting + analytics corresponds to business processes
* **Technical DW Architecture** **reflects the technical nature of the DW, such as MPP (massively parallel processing)**
* A company *can* have a DW *without* an MPP system, *or* run an MPP system that is *not* organized as a DW
* However, the technical + organizational architectures have existed in a virtuous cycle + are frequently identified with each other
* The 1st MPP systems in the late ‘70s became popular in the ‘80s
* **MPPs support essentially the same SQL semantics used in relational application databases**
* Still, **MPPs are optimized to scan massive amounts of data in parallel + thus allow high-performance aggregation + statistical calculations**
* In recent years, **MPP systems have increasingly shifted from a row-based to a columnar architecture to facilitate even *larger* data + queries, especially in cloud DW’s**
* **MPPs = indispensable for running performant queries for large enterprises as data + reporting needs grow**
* One variation on ETL is **ELT**
* W/ the **ELT DW architecture**, **data gets moved more or less directly from production systems into a staging area in the DW**
* **“Staging” in this setting indicates that the data is in a raw form**
* Rather than using an external system, **transformations are handled directly in the DW**
* Intention = **take advantage of the massive computational power of cloud DW’s + data processing tools**
* **Data is processed in batches, + transformed output is written into tables + views for analytics**
* Here is the general process:



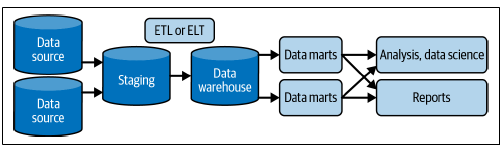
* **ELT is also popular in a streaming arrangement, as events are streamed from a CDC process, stored in a staging area, + then subsequently transformed w/in the DW**
* A 2nd version of ELT was popularized during big data growth in the Hadoop ecosystem: **transform-on-read ELT** (discussed during the “Data Lake” section later)

###### The Cloud Data Warehouse

* **Cloud data warehouses**represent a *significant* evolution of the on-prem DW architecture + have thus led to significant changes to the *organizational* architecture
* Amazon Redshift kicked off the cloud DW revolution.
* Instead of needing to appropriately size an MPP system for the next several years + sign a multimillion-dollar contract to procure the system, companies had the option of **spinning up a Redshift cluster on demand, scaling it up over time as data + analytics demand grew**
* **Could even spin up new Redshift clusters on demand to serve specific workloads + quickly delete clusters when they were no longer needed**
* Google BigQuery, Snowflake, + others popularized the idea of **separating compute from storage**
* In this architecture, **data is housed in object storage, allowing virtually limitless storage**
* This also **gives users the option to spin up computing power on demand, providing ad hoc big data capabilities w/out the long-term cost of thousands of nodes**
* Cloud DW’s expand the capabilities of MPP systems to cover many big data use cases that required a Hadoop cluster in the very recent past
* **Can readily process PB’s of data in a *single* query**
* Typically support data structures that allow the storage of tens of MB of raw text data per row or extremely rich + complex JSON documents
* **As cloud DW’s (and data lakes) mature, the line between the DW and the data lake will continue to blur**
* So significant is the impact of the new capabilities offered by cloud DW’s that we might consider jettisoning the term “data warehouse” altogether
* Instead, **these services are evolving into a new data platform with much broader capabilities than those offered by a traditional MPP system**

###### Data Marts

* A **data mart**is **a more refined subset of a warehouse designed to serve analytics + reporting, focused on a single suborganization, department, or line of business**
* **Every department has its own data mart, specific to its needs**
* This is in **contrast to the *full* DW that serves the broader organization or business**
* Data marts exist for 2 reasons
* 1) A data mart **makes data more easily accessible to analysts + report developers**
* 2) Data marts **provide an additional stage of transformation beyond that provided by the initial ETL/ELT pipelines**
* This **can significantly improve performance if reports or analytics queries require complex joins + aggregations of data, especially when the raw data is large**
* Transform processes can populate the data mart w/ joined and aggregated data to improve performance for live queries
* Below is the general workflow:



* See more in Chapter 8

##### 2) Data Lake

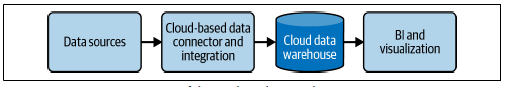
* Among the most popular architectures that appeared during the big data era is the **data lake**, where **instead of imposing tight structural limitations on data, simply dump all data (structured *and* unstructured) into a central location**
* The data lake promised to be a democratizing force, liberating the business to drink from a fountain of limitless data
* The first-generation data lake, **“data lake 1.0,” made solid contributions but generally failed to deliver on its promise**
* Data lake 1.0 started w/ **HDFS**
* **As the cloud grew in popularity, these data lakes moved to cloud-based object storage, w/ extremely cheap storage costs + virtually limitless storage capacity**
* **Instead of relying on a monolithic DW where storage + compute are tightly coupled, the data lake allows an immense amount of data of any size + type to be stored**
* **When this data needs to be queried or transformed, you have access to nearly unlimited computing power by spinning up a cluster on demand, + you can pick your favorite data-processing technology for the task at hand (MapReduce, Spark, Ray, Presto, Hive, etc.)**
* Despite the promise + hype, data lake 1.0 had serious shortcomings
* The **data lake became a dumping ground**; terms such as ”data swamp”, ”dark data”, and ”WORN”were coined as once-promising data projects failed
* **Data grew to unmanageable sizes, w/ little in the way of schema management, data cataloging, + discovery tools**
* In addition, the original data lake concept was essentially write-only, creating huge headaches w/ the arrival of regulations such as GDPR that required targeted deletion of user records
* **Processing data was also challenging**
* Relatively banal data transformations such as joins were a huge headache to code as MapReduce jobs
* Later frameworks such as Pig + Hive somewhat improved the situation for data processing but did little to address the basic problems of data management
* Simple **data manipulation language (DML)** operations common in SQL (deleting or updating rows) were painful to implement, generally achieved by creating entirely new tables
* While big data engineers radiated a particular disdain for their counterparts in data warehousing, the latter could point out that **DW’s provided basic data management capabilities out of the box, + that SQL was an efficient tool for writing complex, performant queries + transformations**
* Data lake 1.0 also failed to deliver on another core promise of the big data movement:
* OSS in the Apache ecosystem was touted as a means to avoid multimillion-dollar contracts for proprietary MPP systems
* Cheap, off-the-shelf hardware would replace custom vendor solutions
* In reality, **big data costs ballooned as the complexities of managing Hadoop clusters forced companies to hire large teams of engineers at high salaries**
* Companies often chose to purchase licensed, customized versions of Hadoop from vendors to avoid the exposed wires + sharp edges of the raw Apache codebase + acquire a set of scaffolding tools to make Hadoop more user-friendly
* Even companies that avoided managing Hadoop clusters using cloud storage had to spend big on talent to write MapReduce jobs
* **Be careful not to understate the utility + power of first-generation data lakes**
* **Many organizations found significant value in data lakes, especially huge, heavily data-focused Silicon Valley tech companies** like Netflix and Facebook
* These companies **had the resources to build successful data practices and create their custom Hadoop-based tools + enhancements**
* **But for many organizations, data lakes turned into an internal superfund site of waste, disappointment, + spiraling costs**

##### 3) Convergence, Next-Generation Data Lakes, and the Data Platform

* In response to the limitations of first-generation data lakes, **various players have sought to enhance the concept to fully realize its promise**
* Ex: **Databricks introduced the notion of a data lakehouse** = incorporates the **controls, data management, + data structures found in a DW while still housing data in object storage + supporting a variety of query + transformation engines**
* In particular, a data lakehouse **supports atomicity, consistency, isolation, + durability (ACID) transactions, a big departure from the original data lake, where you simply pour in data + never update or delete it**
* The term “data lakehouse*”* suggests a convergence between data lakes + DW’s
* The technical architecture of cloud DW’s has evolved to be very similar to a data lake architecture
* Cloud DW’s separate compute from storage, support PB-scale queries, store a variety of unstructured data + semi-structured objects, + integrate w/ advanced processing technologies such as Spark or Beam.
* We believe that the **trend of convergence will only continue**
* The data lake + DW will still exist as different architectures, while in practice, their capabilities will converge so that few users will notice a boundary between them in their day-today work
* We now see several vendors offering **data platforms**that **combine data lake + DW capabilities**
* 2022-2023: AWS, Azure, Google Cloud, Snowflake, + Databricks are class leaders, each offering a constellation of tightly integrated tools for working w/ data, running the gamut from relational to completely unstructured
* **Instead of choosing between a data lake or DW architecture, future DE’s will have the option to choose a converged data platform based on a variety of factors, including vendor, ecosystem, + relative openness**

##### 4) Modern Data Stack

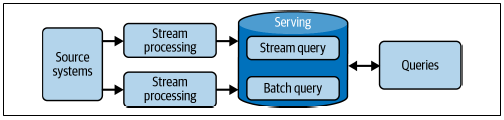
* The **modern data stack**(basic components below) is currently a **trendy analytics architecture that highlights the type of abstraction expected to be sees more widely used over the next several years**



* Whereas **past data stacks relied on expensive, monolithic toolsets**, **the main objective of the modern data stack is to use *cloud-based, plug-and-play, easy-to-use, off-the-shelf components to create a modular + cost-effective data architecture***
* **Components include: data pipelines, storage, transformation, data management/governance, monitoring, visualization, + exploration**
* **The domain is still in flux, + the specific tools are changing + evolving rapidly, but the core aim will remain the same: to reduce complexity and increase modularization**
* Note that the notion of a modern data stack integrates nicely w/ the converged data platform idea from above
* **Key outcomes of the modern data stack = self-service (analytics + pipelines), agile data management, + using open-source tools or simple proprietary tools w/ clear pricing structures**
* **Community is a central aspect of the modern data stack as well**
* Unlike products of the past that had releases + roadmaps largely hidden from users, projects + companies operating in the modern data stack space typically have strong user bases + active communities that participate in development by using the product early, suggesting features, + submitting pull requests to improve code
* Regardless of where “modern” goes (see Chapter 11), the **key concept of plug-and-play modularity with easy-to-understand pricing + implementation is the way of the future**
* Especially in **analytics engineering**, the **modern data stack is + will continue to be the default choice of data architecture**

##### 5) Lambda Architecture

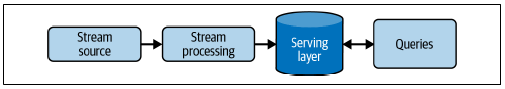
* In the “old days” (early to mid-2010s), the **popularity of working w/ streaming data exploded w/ the emergence of Kafka as a highly scalable message queue + w/ frameworks such as Apache Storm and Samza for streaming/real-time analytics**
* These technologies **allowed companies to perform *new* types of analytics + modeling on large amounts of data, user aggregation + ranking, + product recommendations**
* **DE’s needed to figure out how to reconcile batch and streaming data into a *single* architecture**
* The **Lambda architecture** was one of the early popular responses to this problem: you **have systems operating independently of each other (batch, streaming, + serving)**



* The **source system is ideally immutable + *append-only*, sending data to *two* destinations for processing: stream and batch**
* **In-stream processing** **intends to serve the data w/ the lowest possible latency in a “speed” layer, usually a NoSQL database**
* In the **batch layer, data is processed + transformed in a system such as a DW, creating precomputed + aggregated views of the data**
* The **serving layer** **provides a combined view by aggregating query results from the 2 layers**
* Lambda architecture has its **share of challenges + criticisms**
* **Managing multiple systems w/ different codebases is as difficult as it sounds, creating error-prone systems w/ code + data that are extremely difficult to reconcile**
* Lambda architecture still gets attention + is popular in search-engine results for data architecture
* **Lambda *ISN’T* a first recommendation if trying to combine streaming + batch data for analytics, as tech + practices have moved on**
* The **reaction to Lambda architecture was the Kappa architecture**

##### 6) Kappa Architecture

* As a response to the shortcomings of Lambda architecture, Jay Kreps proposed an alternative called **Kappa architecture**w/ a central thesis of: why not **just use a stream-processing platform as the backbone for *all* data handling (ingestion, storage, + serving)**, which **facilitates a true event-based architecture**
* **Real-time *and* batch processing can be applied seamlessly to the *same* data by reading the live event stream directly + replaying large chunks of data for batch processing**



* Though the original Kappa architecture article came out in 2014, it hasn’t been widely adopted
* There may be a couple of reasons for this
* **1) Streaming itself is still a bit of a mystery for many companies; it’s easy to talk about, but harder than expected to execute**
* 2) Kappa architecture turns out to be complicated + expensive in practice
* **While some streaming systems can scale to huge data volumes, they are complex + expensive**
* **Batch storage + processing remain much more efficient + cost-effective for enormous historical datasets**

##### 7) The Dataflow Model and Unified Batch and Streaming

* Both Lambda + Kappa sought to address limitations of the Hadoop ecosystem of the 2010s by trying to duct-tape together complicated tools that were likely not natural fits in the first place
* **The central challenge of unifying batch and streaming data remained**, + Lambda and Kappa both provided inspiration and groundwork for continued progress in this pursuit
* **1 of the central problems of managing batch and stream processing = unifying multiple code paths**
* While the Kappa architecture relies on a unified queuing + storage layer, one still has to confront using different tools for collecting real-time statistics or running batch aggregation jobs
* Today, engineers seek to solve this in several ways
* Google made its mark by developing the **Dataflow model** (<https://research.google/pubs/the-dataflow-model-a-practical-approach-to-balancing-correctness-latency-and-cost-in-massive-scale-unbounded-out-of-order-data-processing/>) and the **Apache Beam framework** that implements this model
* **Core idea in the Dataflow model = to view all data as events, as the aggregation is performed over various types of windows**
* **Ongoing real-time event streams are unbounded data, + data batches are simply bounded event streams, + the boundaries provide a natural window**
* **Engineers can choose from various windows for real-time aggregation, such as sliding or tumbling**
* **Real-time and batch processing happens in the same system using nearly identical code**
* **The philosophy of “batch as a special case of streaming” is now more pervasive**
* Various frameworks such as Flink and Spark have adopted a similar approach

##### 8) Architecture for IoT

* The **Internet of Things (IoT)** is the **distributed collection of devices** (AKA “things”, like computers, sensors, mobile devices, smart home devices, + anything else w/ an internet connection)
* Rather than generating data from direct human input (likw data entry from a keyboard), **IoT data is generated from devices that collect data periodically or continuously from the surrounding environment + transmit it to a destination**
* IoT devices are often low-powered and operate in low-resource/low bandwidth environments
* While the concept of IoT devices dates back at least a few decades, the smartphone revolution created a massive IoT swarm virtually overnight
* Since then, numerous new IoT categories have emerged, such as smart thermostats, car entertainment systems, smart TVs, and smart speakers
* **The IoT has evolved from a futurist fantasy to a massive DE domain**
* **It’s expected that IoT will become one of the dominant ways data is generated and consumed**
* Having a cursory understanding of IoT architecture will help understand broader data architecture trends. Let’s briefly look at some IoT architecture concepts.

###### a) Devices

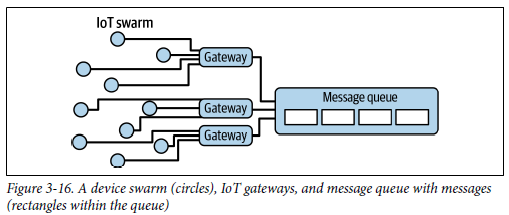
* The physical hardware connected to the internet, sensing the environment around them + collecting + transmitting data to a downstream destination
* Might be used in consumer applications like a doorbell camera, smartwatch, or thermostat
* Might be an AI-powered camera that monitors an assembly line for defective components, a GPS tracker to record vehicle locations, or a Raspberry Pi programmed to download the latest tweets + brew your coffee
* **Any device capable of collecting data from its environment is an IoT device**
* Devices **should be minimally capable of collecting and transmitting data**.
* However, the device **might also crunch data or run ML on the data it collects before sending it downstream (edge computing and edge ML, respectively)**
* A **DE doesn’t necessarily need to know the inner details of IoT devices but should know what the device does, the data it collects, any edge computations or ML it runs before transmitting the data, + how often it sends data**
* Also **helps to know the consequences of a device or internet outage, environmental or other external factors affecting data collection, + how these may impact the downstream collection of data from the device**

###### b) Interfacing with Devices

* **A device isn’t beneficial unless you can get its data**
* This section covers some of the *key components necessary to interface with IoT devices in the wild*:

i) IoT Gateway

* An **IoT gateway**is a **hub for connecting devices + securely routing devices to the appropriate destinations on the internet**
* While you can connect a device *directly* to the internet *w/out* an IoT gateway, the **gateway allows devices to connect using extremely little power**
* It **acts as a way station for data retention + manages an internet connection to the final data destination**
* New low-power WiFi standards are designed to make IoT gateways less critical in the future, but these are just rolling out now
* **Typically, a swarm of devices will utilize *many* IoT gateways, one at each physical location where devices are present (see below)**



ii) Ingestion

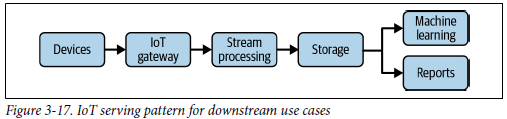
* **Ingestionbegins with an IoT gateway + from there, events + measurements can flow into an event ingestion architecture**
* Of course, other patterns are possible
* Ex: The gateway may accumulate data + upload it in batches for later analytics processing
* Ex: In remote physical environments, gateways may often not have connectivity to a network
* May upload all data only when they are brought into the range of a cellular or WiFi network
* The point is that **the diversity of IoT systems and environments presents complications** (e.g., late-arriving data, data structure + schema disparities, data corruption, + connection disruption) **that DE’s must account for in their architectures + downstream analytics**

iii) Storage

* **Storage requirements will depend a great deal on the latency requirement for the IoT devices in the system** (see Chapter 6)
* Ex: For remote sensors collecting scientific data for analysis at a later time, batch object storage may be perfectly acceptable.
* However, near real-time responses may be expected from a system backend that constantly analyzes data in a home monitoring and automation solution
* In this case, a message queue or time-series database is more appropriate

iv) Serving

* **Serving patterns are incredibly diverse**
* In a **batch** scientific application, data might be **analyzed using a cloud DW** + then **served in a** **report**
* Data will be presented + served in numerous ways in a home-monitoring application
* Data will be analyzed in the near time using a stream-processing engine or queries in a time-series database to look for critical events such as a fire, electrical outage, or break-in
* Detection of an anomaly will trigger alerts to the homeowner, fire department, or other entity
* A batch analytics component also exists (Ex: a monthly report on the state of the home)
* **One significant serving pattern for IoT looks like reverse ETL** (see below), although it’s not usual to use this term in the IoT context



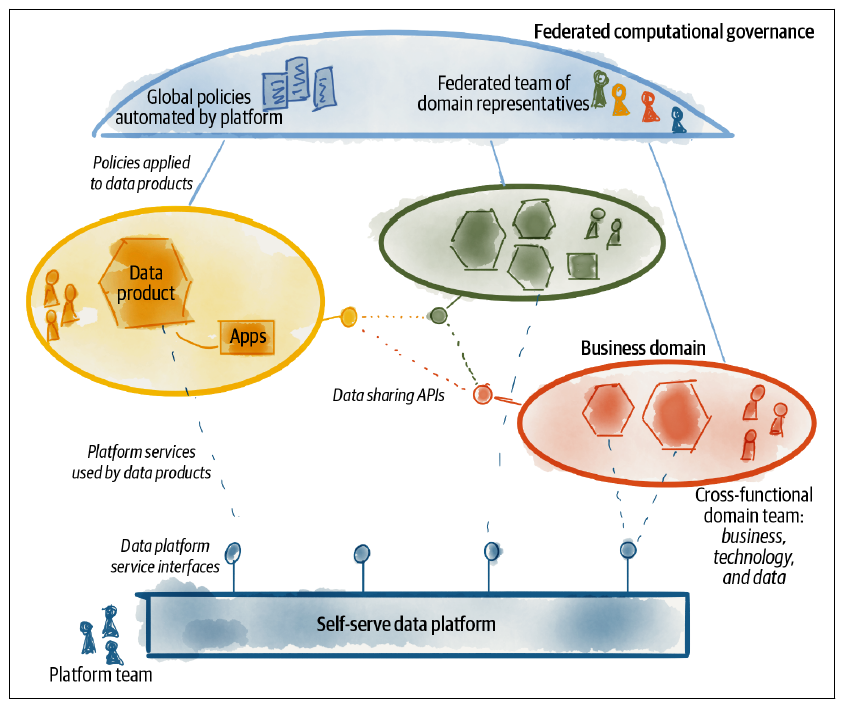
* Think of this scenario: data from sensors on manufacturing devices is collected + analyzed
* The results of these measurements are processed to look for optimizations that will allow equipment to operate more efficiently
* Data is sent back to reconfigure the devices + optimize them

###### c) Scratching the Surface of the IoT

* IoT scenarios are incredibly complex, and IoT architecture + systems are also less familiar to DE’s who may have spent their careers working w/ business data
* This intro above should encourage interested DE’s to learn more about this fascinating + rapidly evolving specialization

##### 9) Data Mesh

* The **data mesh**is a **recent response to sprawling monolithic data platforms, such as centralized data lakes and DW’s, and “the great divide of data,” wherein the landscape is divided between operational data and analytical data**
* The data mesh **attempts to invert the challenges of centralized data architecture, taking the concepts of domain-driven design (commonly used in software architectures) + applying them to data architecture**
* **Because the data mesh has captured much recent attention, be aware of it**
* A **big part of the data mesh is decentralization**, as Zhamak Dehghani noted in her groundbreaking article on the topic (<https://martinfowler.com/articles/data-mesh-principles.html>):
* *“In order to decentralize the monolithic data platform, we need to reverse how we think about data, its locality, + ownership. Instead of flowing the data from domains into a centrally-owned data lake or platform, domains need to host + serve their domain datasets in an easily consumable way*
* Dehghani later identified **4 key components of the data mesh**:
* **Domain-oriented decentralized data ownership + architecture**
* **Data as a product**
* **Self-serve data infrastructure as a platform**
* **Federated computational governance**
* Below shows a simplified version of a data mesh architecture:



* You can learn more about data mesh in Dehghani’s book *Data Mesh* (O’Reilly)
* <https://www.oreilly.com/library/view/data-mesh/9781492092384/>

##### Other Data Architecture Examples

* Data architectures have countless other variations, such as data fabric, data hub, scaled architecture, metadata-first architecture, event-driven architecture, live data stack (Chapter 11), + many more
* And **new architectures will continue to emerge as practices consolidate and mature, and tooling simplifies and improves**
* We’ve focused on a handful of the most critical data architecture patterns that are extremely well-established, evolving rapidly, or both
* **As a DE, pay attention to how new architectures may help your organization**
* **Stay abreast of new developments by cultivating a high-level awareness of the DE ecosystem developments**
* **Be open-minded and don’t get emotionally attached to one approach**
* **Once you’ve identified potential value, deepen your learning + make concrete decisions**
* ***When done right,* minor tweaks (or major overhaul) in your data architecture can positively impact the business**

#### Who’s Involved with Designing a Data Architecture?

* **Data architecture isn’t designed in a vacuum**
* Bigger companies may still employ data architects, but those **architects will need to be heavily in tune + current w/ the state of tech and data,** b/c gone are the days of ivory tower data architecture
* In the past, architecture was largely orthogonal to engineering, + it’s expected that **this distinction will disappear as DE, + engineering in general, quickly evolves, becoming more agile, w/ less separation between engineering and architecture**
* **Ideally, a DE will work *alongside* a dedicated data architect**
* However, **if a company is small or low in its level of data maturity, a DE might work double duty as an architect**
* **Because data architecture is an undercurrent of the DE lifecycle, a DE should understand “good” architecture + the various types of data architecture**
* **When designing architecture, you’ll work alongside business stakeholders to evaluate trade-offs**
* What are the trade-offs inherent in adopting a cloud DW vs. a data lake?
* What are the trade-offs of various cloud platforms?
* When might a unified batch/streaming framework (Beam, Flink) be an appropriate choice?
* **Studying these choices in the abstract will prepare you to make concrete, valuable decisions**

#### Conclusion

* **You’ve learned how data architecture fits into the DE lifecycle + what makes for “good” data architecture, + you’ve seen several examples of data architectures**
* **Because architecture is such a key foundation for success, invest the time to study it deeply + understand the trade-offs inherent in any architecture**
* **Be prepared to map out architecture that corresponds to your organization’s unique requirements**