N.B.K.R.INSTITUTE OF SCIENCE &TECHNOLOGY,

VIDYANAGAR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DATA ENGINEERING

LECTURE NOTES

UNIT-III

DATA ENGINEERING
LECTURE NOTES

SUBJECT: DATA ENGINEERING

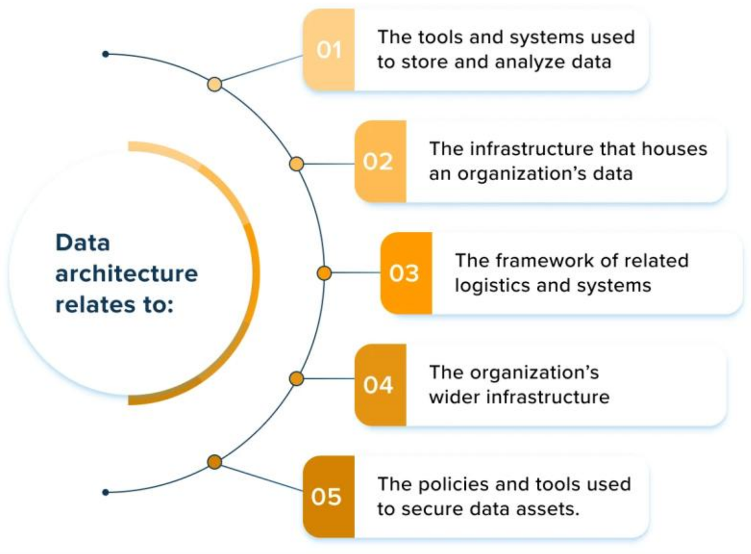
UNIT-III

Designing Good Data Architecture: Enterprise Architecture, Data Architecture**,** Principles of Good Data Architecture, Major Architecture Concepts.
Data Generation in Source Systems: Sources of Data, Files and Unstructured Data, APIs, Application Databases (OLTP), OLAP, Change Data Capture, Logs, Database Logs, CRUD, Source System Practical Details.

What is Data Architecture?

Data architecture is a framework that describes how an organization manages its data, including how it's collected, stored, and used. It's a blueprint for an organization's data assets and data management resources, and it's made up of models, rules, and standards.

Data architecture is important because it: Ensures data quality and security, Aligns data processes with business goals, Supports strategic initiatives like AI and business intelligence, Avoids redundant data storage, and Enables new applications.


Data architecture relates to:
01 The tools and systems used to store and analyze data
02 The infrastructure that houses an organization's data
03 The framework of related logistics and systems
04 The organization's wider infrastructure
05 The policies and tools used to secure data assets.

Data architects design data architectures to meet business and technology requirements, while also ensuring data security and compliance with regulations. Data architectures can vary depending on the business's goals and needs. For example, a financial data architecture might focus on collecting, storing, and managing high-velocity data, while a healthcare data architecture might focus on protecting patient health records.

3.1. Enterprise Data Architecture

Enterprise architecture has many subsets, including business, technical, application, and data (Figure below). As such, many frameworks and resources are devoted to enterprise architecture. In truth, architecture is a surprisingly controversial topic.

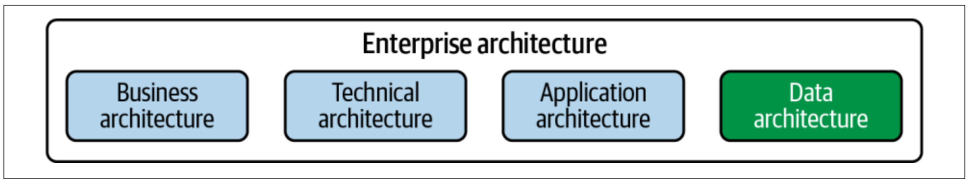


Figure. Data architecture is a subset of enterprise architecture

TOGAF's definition: TOGAF is The Open Group Architecture Framework, a standard of The Open Group. It's touted as the most widely used architecture framework today. Here's the TOGAF definition:

The term "enterprise" in the context of "enterprise architecture" can denote an entire enterprise—encompassing all of its information and technology services, processes, and infrastructure—or a specific domain within the enterprise. In both cases, the architecture crosses multiple systems, and multiple functional groups within the enterprise.

Gartner's definition: Gartner is a global research and advisory company that produces research articles and reports on trends related to enterprises. Among other things, it is responsible for the (in)famous Gartner Hype Cycle. Gartner's definition is as follows:

Enterprise architecture (EA) is a discipline for proactively and holistically leading enterprise responses to disruptive forces by identifying and analyzing the execution of change toward desired business vision and outcomes. EA delivers value by presenting business and IT leaders with signature-ready recommendations for adjusting policies and projects to achieve targeted business outcomes that capitalize on relevant business disruptions.

EABOK's definition: EABOK is the Enterprise Architecture Book of Knowledge, an enterprise architecture reference produced by the MITRE Corporation. EABOK was released as an incom- plete draft in 2004 and has not been updated since. Though seemingly obsolete, EABOK is frequently referenced in descriptions of enterprise architecture; we found many of its ideas helpful while writing this book. Here's the EABOK definition:

Enterprise Architecture (EA) is an organizational model; an abstract representation of an Enterprise that aligns strategy, operations, and technology to create a roadmap for success.

Our 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.

We discuss each theme at length in this section and then make the definition more concrete in the latter part of the chapter by giving various examples of data architecture.

Decisions must be reversible and flexible for two reasons. First of all, it is impossible to forecast the future because the world is always changing. Reversible choices enable you to modify your direction when circumstances change and you learn new facts. Second, as organisations expand, there is a natural tendency for business ossification to occur. By lowering the risk involved in a choice, a culture of reversible decisions aids in overcoming this inclination.

One-way and two-way doors are credited to Jeff Bezos.. Reversing a decision that is one-way is nearly impossible. For instance, Amazon had the option of closing AWS or selling it. After taking such a step, it would be extremely difficult for Amazon to re-establish a public cloud with the same market position. On the other hand, a two-way door is an easily reversible decision: you walk through and proceed if you like what you see in the room or step back through the door if you don’t. Amazon might decide to require the use of DynamoDB for a new micro- services database. If this policy doesn’t work, Amazon has the option of reversing it and 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, and collecting data rapidly.

Change management is closely related to reversible decisions and is a central theme of enterprise architecture frameworks. Even with an emphasis on reversible decisions, enterprises often need to undertake large initiatives.

Architects do more than just sketch up IT procedures and hazily envision a far-off, ideal future; they actively address business issues and generate new opportunities. Technical solutions exist to serve corporate objectives rather than for their own sake. Architects pinpoint issues with the current state (poor data quality, scalability constraints, financially unsuccessful business segments), specify ideal future states (agile data-quality enhancement, scalable cloud data solutions, enhanced business processes), and bring initiatives to life by carrying out tiny, tangible steps. It is worth restating:

Technical solutions exist not for their own sake but in support of business goals.

3.2. Data Architecture

Data architecture is a subset of enterprise architecture, inheriting its properties: processes, strategy, change management, and evaluating trade-offs. Here are a couple of definitions of data architecture that influence our definition.

TOGAF’s definition: A description of the structure and interaction of the enterprise’s major types and sources of data, logical data assets, physical data assets, and data management resources.

DAMA’s definition: The DAMA DMBOK defines data architecture as follows:

Identifying the data needs of the enterprise (regardless of structure) and designing and maintaining the master blueprints to meet those needs. Using master blueprints to guide data integration, control data assets, and align data investments with business strategy.

Our definition: Data architecture is the design of systems to support the evolving data needs of an enterprise, achieved by flexible and reversible decisions reached through a careful evaluation of trade-offs.

The data engineering life-cycle is a subset of the data life-cycle; data engineering architecture is a subset of general data architecture.

The systems and frameworks that comprise the essential components of the data engineering lifecycle are known as data engineering architecture. The terms data architecture and data engineering architecture will be used interchangeably.

You should also be aware of the **operational** and **technical** components of data architecture (Figure below). The functional requirements of what must occur in relation to people, processes, and technology are all included in operational architecture. What business operations, for instance, does the data support? How is data quality managed by the company? What is the minimum amount of time that must pass between the production of the data and its query ability? Throughout the data engineering lifecycle, technical architecture describes how data is ingested, stored, transformed, and served. For example, how would you transfer 10 TB of data per hour to your data lake from a source database? To put it briefly, technical architecture explains how things will be done, while operational architecture outlines what must be done.

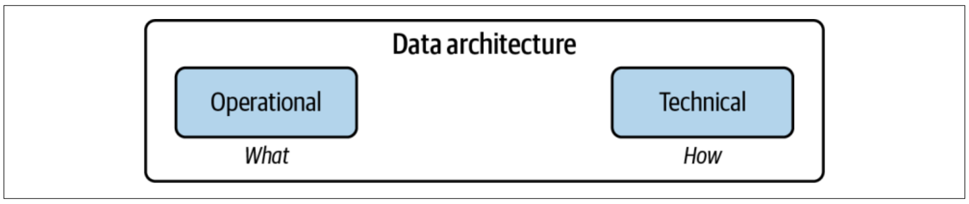


Figure. Operational and technical data architecture

"Good" Data Architecture: Never shoot for the best architecture, but rather the least worst 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.

Good data architecture serves business requirements with a common, widely reusable set of building blocks while maintaining flexibility and making appropriate trade-offs. Bad architecture is authoritarian and tries to cram a bunch of one-size-fits-all decisions into a big ball of mud.

Good data architecture is built on agility, which recognises that the world is changing. A well-designed data architecture is adaptable and simple to maintain. It varies in reaction to company shifts as well as new procedures and technology that could eventually yield even greater value. Data use cases and businesses are constantly changing. The data space is changing at an accelerated rate, and the world is dynamic. The data architecture that worked effectively for you last year might not be enough for today, much less next year.

Bad data architecture is tightly coupled, rigid, overly centralized, or uses the wrong tools for the job, hampering development and change management. Ideally, by designing architecture with reversibility in mind, changes will be less costly.

The undercurrents of the data engineering 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, and software engineering.

Good data architecture is a living, breathing thing. It's never finished. In fact, per our definition, change and evolution are central to the meaning and purpose of data architecture. Let's now look at the principles of good data architecture.

3.3. Principles of Data Architecture

The AWS Well-Architected Framework consists of six pillars:

a. Operational excellence
b. Security
c. Reliability
d. Performance efficiency
e. Cost optimization
f. Sustainability

Google Cloud's Five Principles for Cloud-Native Architecture are as follows:

a. Design for automation.
b. Be smart with state.
c. Favor managed services.
d. Practice defense in depth.
e. Always be architecting.

We advise you to carefully study both frameworks, identify valuable ideas, and deter- mine points of disagreement. We'd like to expand or elaborate on these pillars with these principles of data engineering 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.

Principle 1: Choose Common Components Wisely

Selecting common elements and procedures that can be applied broadly throughout an organisation is one of a data engineer's main responsibilities. Common elements build a fabric that promotes teamwork and dismantles silos when architects make wise decisions and exercise good leadership. Together with shared knowledge and abilities, common components allow for agility both inside and between teams.

Common components can be anything that has broad applicability within an organization. Common components include object storage, version-control systems, observability, monitoring and orchestration systems, and processing engines.

Common components should be accessible to everyone with an appropriate use case, and teams are encouraged to rely on common components already in use rather than reinventing the wheel.

Common components must support robust permissions and security to enable sharing of assets among teams while preventing unauthorized access.

Cloud platforms are an ideal place to adopt common components. For example, compute and storage separation in cloud data systems allows users to access a shared storage layer (most commonly object storage) using specialized tools to access and query the data needed for specific use cases.

Principle 2: Plan for Failure

Modern hardware is highly robust and durable. 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: The percentage 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.

Recovery time objective: The 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 one day might be fine for an internal reporting system. A website outage of just five 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 the maximum acceptable data loss.

Principle 3: Architect for Scalability

Scalability in data systems encompasses two main capabilities.

First, scalable systems can **scale up** to handle significant quantities of data. We might need to spin up a large cluster to train a model on a petabyte 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.

Second, scalable systems can **scale down**. Once the load spike ebbs, we should automatically remove capacity to cut costs. An elastic system can 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 the large model-training job completes, we can delete the cluster. Many ser- verless systems (e.g., serverless functions and serverless online analytical processing, or OLAP, databases) can automatically scale to zero.

Keep in mind that using improper scaling techniques might lead to expensive and too complex systems. An application might be better served by a simple relational database with a single failover node rather than a sophisticated cluster configuration. To find out if your database architecture is suitable, measure your current load, project load spikes, and project load over the next few years. Your start up should have additional resources available to redesign for scalability if it expands far more quickly than expected.

Principle 4: Architecture Is Leadership

Data architects are responsible for technology decisions and architecture descrip- tions and disseminating these choices through effective leadership and training. Data architects should be highly technically competent but delegate most individual contributor work to others. Strong leadership skills combined with high technical competence are rare and extremely valuable. The best data architects take this duality seriously.

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 and thus running the risk of being an architectural bottleneck.

Principle 5: Always Be Architecting

This principle 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 and technology. 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 plan to determine priorities and the order of architecture changes.

Principle 6: Build Loosely Coupled Systems

When the architecture of the system is designed to enable teams to test, deploy, and change systems without dependencies on other teams, teams require little communication to get work done. In other words, both the architecture and the teams are loosely coupled.

In 2002, Bezos wrote an email to Amazon employees that became known as the Bezos API Mandate:

a. All teams will henceforth expose their data and functionality through service interfaces.
b. Teams must communicate with each other through these interfaces.
c. 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 inter- face calls over the network.
d. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols—doesn't matter.
e. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

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 and protect internal details of the service, such as a database backend or internal classes and 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 and 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. We 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 with engineering, maintaining, and 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 with other teams' components; they simply use the published API or message specifications to call these components. They iterate their part to improve their performance and capabilities over time. They might also publish new capabilities as they are added or request new stuff from other teams. Again, the latter hap- pens without 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 with minimal downtime. Teams release continuously during regular working hours to make code changes and test them.

Principle 7: Make Reversible Decisions

The data landscape is changing rapidly. Today's hot technology or stack is tomorrow's afterthought. Popular opinion shifts quickly. You should aim for reversible decisions, as these tend to simplify your architecture and keep it agile.

Principle 8: Prioritize Security

Every data engineer must assume responsibility for the security of the systems they build and maintain. We focus now on two main ideas:

a. Zero-trust security
b. Shared responsibility security model.

These align closely to a cloud-native architecture.

Hardened-perimeter and zero-trust security model: To define zero-trust security, it's helpful to start by understanding the traditional hard-perimeter security model and its limitations, as detailed in Google Cloud's Five Principles:

Traditional architectures place a lot of faith in perimeter security, crudely a hard- ened 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 shared responsibility model: Amazon emphasizes the shared responsibility model, which divides security into the security of the cloud and security in the cloud. 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 with 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, and applicable laws and regulations.

Data engineers as security engineers: In the corporate world today, a command-and-control approach to security is quite common, wherein security and networking teams manage perimeters and 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 data engineers should consider themselves security engineers.

Principle 9: Embrace FinOps

FinOps is an evolving cloud financial management discipline and cultural practice that enables organizations to get maximum business value by helping engineering, finance, technology, and 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 advo- cates a collaborative working relationship between DevOps and 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 cloud age has seen a significant change in the cost structure of data. Data systems are typically purchased with a capital expenditure in an on-premises arrangement, with a new system being added every few years. The budget and the required computing and storage capacity must be balanced by the responsible parties. Under buying may necessitate quicker technology refresh cycles, which come with additional expenses, while overbuying results in financial waste and hinders future data initiatives and requires a large amount of staff work to monitor system load and data quantity.

The majority of data solutions in the cloud era are easily scalable and pay-as-you-go. Systems may operate using a pay-as-you-go paradigm, a cost-per-query methodology, or a cost-per-processing-capacity model. Compared to the capital expenditure approach, this strategy may be significantly more effective. Scaling down to save money and scaling up for great performance is now feasible. Pay-as-you-go, on the other hand, makes spending far more flexible. Data executives now have the problem of managing efficiency, priorities, and budgets.

Cloud tooling requires a set of procedures for resource and expense management. Data engineers used to think in terms of performance engineering, which involves purchasing enough resources for upcoming requirements and optimising data processes on a certain set of resources. Engineers working in FinOps must develop the ability to consider cloud system cost structures. For instance, while operating a distributed cluster, what combination of AWS spot instances is suitable? Which strategy is best for managing a large daily task in terms of performance and cost-effectiveness? When should the business go to reserved capacity from a pay-per-query model?

The operational monitoring methodology is evolved by FinOps to continuously monitor spending. FinOps may track the continuous cost of serverless operations managing traffic in addition to expenditure spikes that trigger warnings, rather than just tracking requests and CPU usage for a web server. Companies may think about establishing strict expenditure caps with embarrassing failure modes in reaction to spending spikes, just like systems are made to fail gracefully in the event of high traffic.

Major Architecture Concepts

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

Domains and Services

A *domain* is the real-world subject area for which you're architecting. A *service* is a set of functionality whose goal is to accomplish a task. For example, you 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. For example, you might have a sales domain with three services: orders, invoicing, and products. Each service has particular tasks that support the sales domain. Other domains may also share services (Figure 3-3). In this case, the accounting domain is responsible for basic accounting functions: invoicing, payroll, and accounts receivable (AR). Notice the accounting domain shares the invoice service with the sales domain since a sale generates an invoice,

and accounting must keep track of invoices to ensure that payment is received. Sales and accounting own their respective domains.

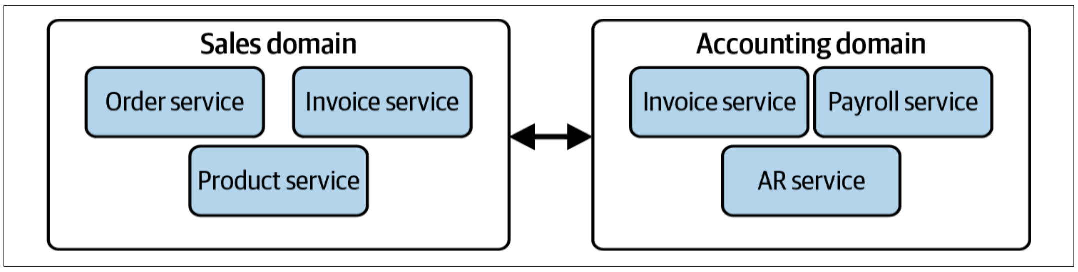


Figure 3-3. Two domains (sales and accounting) share a common service (invoices), and sales and accounting own their respective domains

When thinking about what constitutes a domain, focus on what the domain repre- sents in the real world and work backward. In the preceding example, the sales domain should represent what happens with the sales function in your company. When architecting the sales domain, avoid cookie-cutter copying and 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 and what services to include, the best advice is to simply go and talk with users and stakeholders, listen to what they're saying, and 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 Failure

The discussion in this section is related to our second and third principles of data engineering architecture discussed previously: plan for failure and architect for scala- bility. As data engineers, we're interested in four closely related characteristics of data systems (availability and reliability were mentioned previously, but we reiterate them here for completeness):

Scalability

Allows us to increase the capacity of a system to improve performance and handle the demand. For example, we 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 and down based on the current workload. Scaling up is critical as demand increases, while scaling down saves money in a cloud environ- ment. Modern systems sometimes scale to zero, meaning they can automatically shut down when idle.

Availability

The percentage 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 without manual intervention from engineers—elastic- ity improves reliability.

Scalability can be realized in a variety of ways. For your services and domains, does a single machine handle everything? A single machine can be scaled vertically; you 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 and failover? Single machines generally can't offer high availability and reliability.

We utilize a distributed system to realize higher overall scaling capacity and increased availability and reliability. *Horizontal scaling* allows you to add more machines to satisfy load and resource requirements (Figure 3-4). Common horizontally scaled systems have a leader node that acts as the main point of contact for the instantiation, progress, and completion of workloads. When a workload is started, the leader node distributes tasks to the worker nodes within its system, completing the tasks and 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 data warehouse 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.

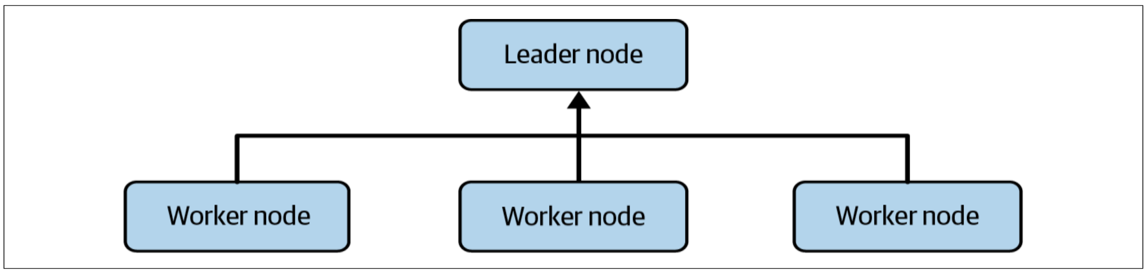


Figure 3-4. A simple horizontal distributed system utilizing a leader-follower architec- ture, with one leader node and three worker nodes

Tight Versus Loose Coupling: Tiers, Monoliths, and Microservices

When designing a data architecture, you choose how much interdependence you want to include within your various domains, services, and resources. On one end of the spectrum, you can choose to have extremely centralized dependencies and workflows. Every part of a domain and service is vitally dependent upon every other domain and service. This pattern is known as *tightly coupled*.

On the other end of the spectrum, you have decentralized domains and 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, owner- ship, responsibility, and accountability to the teams owning their respective domains and services. Designing "good" data architecture relies on trade-offs between the tight and loose coupling of domains and services.

It's worth noting that many of the ideas in this section originate in software develop- ment. We'll try to retain the context of these big ideas' original intent and spirit— keeping them agnostic of data—while later explaining some differences you should be aware of when applying these concepts to data specifically.

Architecture tiers

As you develop your architecture, it helps to be aware of architecture tiers. Your architecture has layers—data, application, business logic, presentation, and so forth
—and 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 and flexibility. Let’s look at single-tier and multitier architecture.

Single tier. In a *single-tier architecture*, your database and application are tightly coupled, residing on a single server (Figure 3-5). This server could be your laptop or a single virtual machine (VM) in the cloud. The tightly coupled nature means if the server, the database, or the application fails, the entire architecture fails. While single-tier architectures are good for prototyping and development, they are not advised for production environments because of the obvious failure risks.

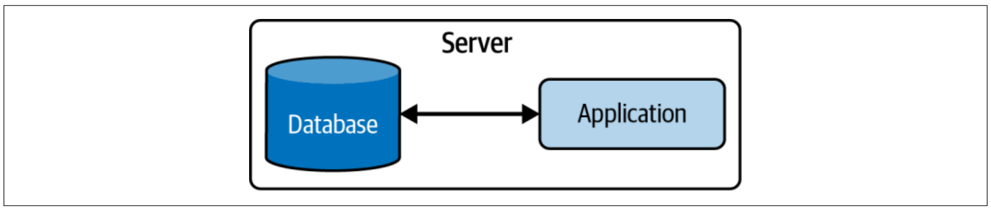


Figure 3-5. Single-tier architecture

Even when single-tier architectures build in redundancy (for example, a failover replica), they present significant limitations in other ways. For instance, it is often impractical (and not advisable) to run analytics queries against production applica- tion 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.

Multitier. The challenges of a tightly coupled single-tier architecture are solved by decoupling the data and application. A *multitier* (also known as *n-tier*) architecture is composed of separate layers: data, application, business logic, presentation, etc. These layers are bottom-up and 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, and application from the presentation.

A common multitier architecture is a three-tier architecture, a widely used client- server design. A *three-tier architecture* consists of data, application logic, and presen- tation tiers (Figure 3-6). Each tier is isolated from the other, allowing for separation of concerns. With a three-tier architecture, you’re free to use whatever technologies you prefer within each tier without the need to be monolithically focused.

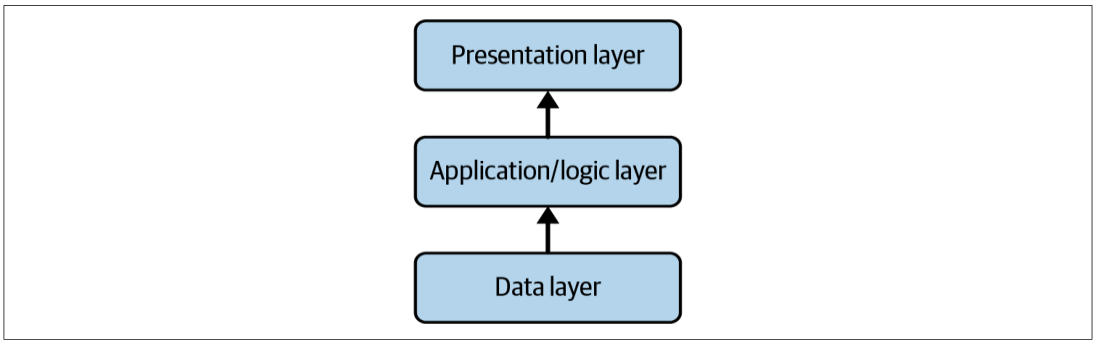


Figure 3-6. A three-tier architecture

We've seen many single-tier architectures in production. Single-tier architectures offer simplicity but also severe limitations. Eventually, an organization or application outgrows this arrangement; it works well until it doesn't. For instance, in a single-tier architecture, the data and logic layers share and compete for resources (disk, CPU, and memory) in ways that are simply avoided in a multitier architecture. Resources are spread across various tiers. Data engineers should use tiers to evaluate their layered architecture and the way dependencies are handled. Again, start simple and bake in evolution to additional tiers as your architecture becomes more complex.

In a multitier architecture, you need to consider separating your layers and the way resources are shared within layers when working with a distributed system. Distributed systems under the hood power many technologies you'll encounter across the data engineering lifecycle. First, think about whether you want resource contention with your nodes. If not, exercise a *shared-nothing architecture*: a single node handles each request, meaning other nodes do not share resources such as memory, disk, or CPU with this node or with each other. Data and resources are isolated to the node. Alternatively, various nodes can handle multiple requests and share resources but at the risk of resource contention. Another consideration is whether nodes should share the same disk and memory accessible by all nodes. This is called a *shared disk architecture* and is common when you want shared resources if a random node failure occurs.

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 and user interface.

Coupling within monoliths can be viewed in two ways: technical coupling and domain coupling. *Technical coupling* refers to architectural tiers, while *domain cou- pling* refers to the way domains are coupled together. A monolith has varying degrees of coupling among technologies and domains. You could have an application with various layers decoupled in a multitier architecture but still share multiple domains. Or, you 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's often a game of whack-a-mole: one component is improved, often at the expense of unknown consequences with 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.

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 with excessive tight coupling.

Microservices

Compared with the attributes of a monolith—interwoven services, centralization, and tight coupling among services—microservices are the polar opposite. *Microservices architecture* comprises separate, decentralized, and loosely coupled services. Each service has a specific function and is decoupled from other services operating within 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 micro- services (Figure 3-7). This completely depends on how complex your monolith is and how much effort it will be to start extracting services out of it. It's entirely possible that your monolith cannot be broken apart, in which case, you'll want to start creating a new parallel architecture that has the services decoupled in a microservices-friendly manner. We don't suggest an entire refactor but instead break out services. The monolith didn't arrive overnight and is a technology issue as an organizational one. Be sure you get buy-in from stakeholders of the monolith if you plan to break it apart.

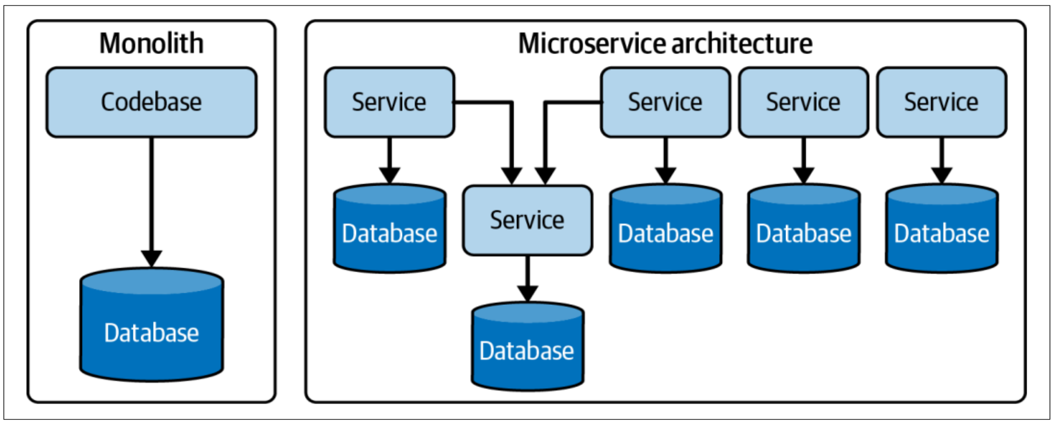


Figure 3-7. An extremely monolithic architecture runs all functionality inside a single codebase, potentially colocating a database on the same host server

Considerations for data architecture

As we mentioned at the start of this section, the concepts of tight versus loose coupling stem from software development, with some of these concepts dating back over 20 years. Though architectural practices in data are now adopting those from software development, it's still common to see very monolithic, tightly coupled data architectures. Some of this is due to the nature of existing data technologies and the way they integrate.

For example, data pipelines might consume data from many sources ingested into a central data warehouse. The central data warehouse is inherently monolithic. A move toward a microservices equivalent with a data warehouse is to decouple the workflow with domain-specific data pipelines connecting to corresponding domain-specific data warehouses. For example, the sales data pipeline connects to the sales-specific data warehouse, and the inventory and product domains follow a similar pattern.

Rather than dogmatically preach microservices over monoliths (among other argu- ments), we suggest you pragmatically use loose coupling as an ideal, while recogniz- ing the state and limitations of the data technologies you're using within your data architecture. Incorporate reversible technology choices that allow for modularity and loose coupling whenever possible.

As you can see in Figure 3-7, 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 com- plexity of sharing domains. Along the lines of single versus multitiered architecture, you should also consider how you separate the domains of your data architecture. For example, your analyst team might rely on data from sales and inventory. The sales and inventory domains are different and should be viewed as separate.

One approach to this problem is centralization: a single team is responsible for gathering data from all domains and reconciling it for consumption across the orga- nization. (This is a common approach in traditional data warehousing.) Another approach is the *data mesh*. With the data mesh, each software team is responsible for preparing its data for consumption across the rest of the organization. We'll say more about the data mesh later in this chapter.

Our advice: monoliths aren't necessarily bad, and it might make sense to start with one under certain conditions. Sometimes you need to move fast, and 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 Versus Multitenant

As a data engineer, you have to make decisions about sharing systems across mul- tiple teams, organizations, and customers. In some sense, all cloud services are multitenant, although this multitenancy occurs at various grains. For example, 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 and isolation so long as customers configure their permissions correctly.

Engineers frequently need to make decisions about multitenancy at a much smaller scale. For example, do multiple departments in a large company share the same data warehouse? Does the organization share data for multiple large customers within the same table?

We have two factors to consider in multitenancy: performance and security. With multiple large tenants within 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, and engineers must prevent data leakage. Strategies for data isolation vary by system. For instance, it is often perfectly acceptable to use multitenant tables and isolate data through views. However, you must make certain that these views cannot leak data. Read vendor or project documentation to understand appropriate strategies and 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* that are broadly defined as something that happened, typically a change in the *state* of something. For example, a new order might be created by a customer, or a customer might later make an update to this order.

An event-driven workflow (Figure 3-8) encompasses the ability to create, update, and asynchronously move events across various parts of the data engineering lifecy- cle. This workflow boils down to three main areas: event production, routing, and consumption. An event must be produced and routed to something that consumes it without tightly coupled dependencies among the producer, event router, and consumer.

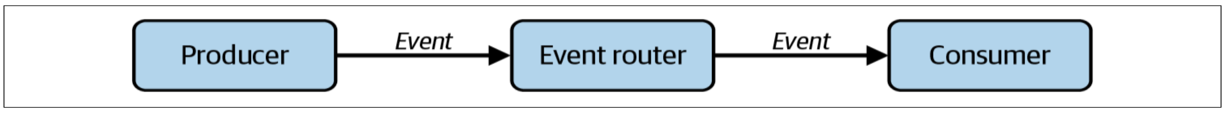


Figure 3-8. In an event-driven workflow, an event is produced, routed, and then consumed

An event-driven architecture (Figure 3-9) embraces the event-driven workflow and uses this to communicate across various services. The advantage of an event-driven architecture is that 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 of the examples we describe later in this chapter incorporate some form of event-driven architecture.

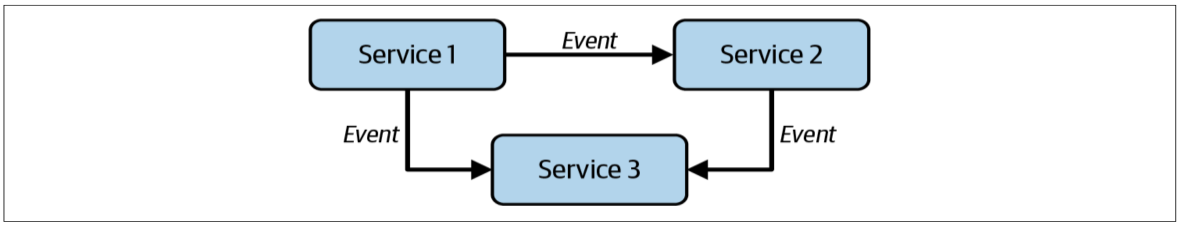


Figure 3-9. In an event-driven architecture, events are passed between loosely coupled services

Brownfield Versus Greenfield Projects

Before you design your data architecture project, you need to know whether you're starting with a clean slate or redesigning an existing architecture. Each type of project requires assessing trade-offs, albeit with different considerations and approaches. Projects roughly fall into two buckets: brownfield and greenfield.

Brownfield projects

Brownfield projects often involve refactoring and reorganizing an existing architecture and are constrained by the choices of the present and past. Because a key part of architecture is change management, you must figure out a way around these limitations and design a path forward to achieve your new business and technical objectives. Brownfield projects require a thorough understanding of the legacy archi- tecture and the interplay of various old and new technologies. All too often, it's easy to criticize a prior team's work and decisions, but it is far better to dig deep, ask questions, and understand why decisions were made. Empathy and context go a long way in helping you diagnose problems with the existing architecture, identify opportunities, and recognize pitfalls.

You'll need to introduce your new architecture and technologies and deprecate the old stuff at some point. 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, we don't advise this approach because of the associated risks and lack of a plan. This path often leads to disaster, with many irreversible and costly decisions. Your job is to make reversible, high-ROI decisions.

A popular alternative to a direct rewrite is the strangler pattern: new systems slowly and incrementally replace a legacy architecture's components. Eventually, the legacy architecture is completely replaced. The attraction to the strangler pattern is its targeted and surgical approach of deprecating one piece of a system at a time. This allows for flexible and reversible decisions while assessing the impact of the deprecation on dependent systems.

It's important to note that deprecation might be "ivory tower" advice and not practi- cal or achievable. Eradicating legacy technology or architecture might be impossible if you're at a large organization. Someone, somewhere, is using these legacy compo- nents. 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 and then follow an exit plan to shut down old systems.

Greenfield projects

On the opposite end of the spectrum, a *greenfield project* allows you to pioneer a fresh start, unconstrained by the history or legacy of a prior architecture. Greenfield projects tend to be easier than brownfield projects, and many data architects and engineers find them more fun! You have the opportunity to try the newest and coolest tools and architectural patterns. What could be more exciting?

You should watch out for some things before getting too carried away. We see teams get overly exuberant with shiny object syndrome. They feel compelled to reach for the latest and greatest technology fad without 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.[21] Always prioritize requirements over building something cool.

Whether you're working on a brownfield or greenfield project, always focus on the tenets of "good" data architecture. Assess trade-offs, make flexible and reversible decisions, and strive for positive ROI.

Now, we'll look at examples and types of architectures—some established for dec- ades (the data warehouse), some brand-new (the data lakehouse), and some that quickly came and went but still influence current architecture patterns (Lambda architecture).

Data Generation in Source Systems:

This chapter covers some popular operational source system patterns and the important types of source systems. Many source systems exist for data generation. These systems generate and things you should consider when working with source systems. We also discuss how the undercurrents of data engineering apply to this first phase of the data engineering lifecycle.

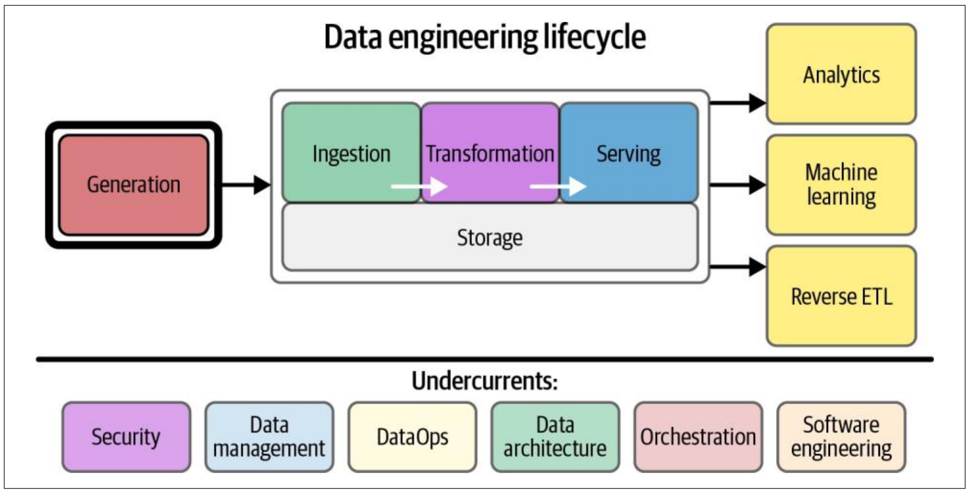


Figure 5-1. Source systems generate the data for the rest of the data engineering lifecycle

As data rapidly increasing in an organization, especially with the rise of data sharing, we expect that a data engineer's role will shift heavily toward understanding between data sources and destinations. The basic tasks of data engineering moving data from A to B will simplify dramatically. On the other hand, it will remain critical to understand the nature of data as it's created in source systems.

Sources of Data: How Is Data Created?

As we learn about the various operational patterns of the systems that generate data, it's essential to understand how data is created. Data is an unorganized, context-less collection of facts and figures. It can be created in many ways, both analog and digital.

Analog data creation occurs in the real world, such as vocal speech, sign language, writing on paper, or playing an instrument. This analog data is often transient; how often have you had a verbal conversation whose contents are lost to the either after the conversation ends?

Digital data is either created by converting analog data to digital form or is the native product of a digital system. An example of analog to digital is a mobile texting app that converts analog speech into digital text. An example of digital data creation is a credit card transaction on an ecommerce platform. A customer places an order, the transaction is charged to their credit card, and the information for the transaction is saved to various databases.

We'll utilize a few common examples in this chapter, such as data created when interacting with a website or mobile application. But in truth, data is everywhere in the world around us. We capture data from IoT devices, credit card terminals, telescope sensors, stock trades, and more.

Get familiar with your source system and how it generates data. Put in the effort to read the source system documentation and understand its patterns and quirks. If your source system is an RDBMS, learn how it operates (writes, commits, queries, etc.); learn the ins and outs of the source system that might affect your ability to ingest from it.

Source Systems: Main Ideas

Source systems produce data in various ways. This section discusses the main ideas you'll frequently encounter as you work with source systems.

Files and Unstructured Data

A *file* is a sequence of bytes, typically stored on a disk. Applications often write data to files. Files may store local parameters, events, logs, images, and audio. In addition, files are a universal medium of data exchange. As much as data engineers wish that they could get data programmatically, much of the world still sends and receives files. For example, if you're getting data from a government agency, there's an excellent chance you'll download the data as an Excel or CSV file or receive the file in an email.

The main types of source file formats you'll run into as a data engineer—files that originate either manually or as an output from a source system process—are Excel, CSV, TXT, JSON, and XML. These files have their quirks and can be structured (Excel, CSV), semi structured (JSON, XML, CSV), or unstructured (TXT, CSV). Although you'll use certain formats heavily as a data engineer (such as Parquet, ORC, and Avro), we'll cover these later and put the spotlight here on source system files.

CSV-Comma Separated Value, TXT, JSON-Java Script Object Notation, and XML-Extension Markup Language.

APIs

Application programming interfaces (APIs) are a standard way of exchanging data between systems. In theory, APIs simplify the data ingestion task for data engineers. In practice, many APIs still expose a good deal of data complexity for engineers to manage. Even with the rise of various services and frameworks, and services for automating API data ingestion, data engineers must often invest a good deal of energy into maintaining custom API connections.

Application DataBases (OLTP Systems)

An *application database* stores the state of an application. A standard example is a database that stores account balances for bank accounts. As customer transactions and payments happen, the application updates bank account balances.

Typically, an application database is an online transaction processing (OLTP) system— a database that reads and writes individual data records at a high rate. OLTP systems are often referred to as transactional databases, but this does not necessarily imply that the system in question supports atomic transactions.

More generally, OLTP databases support low latency and high concurrency. An RDBMS database can select or update a row in less than a millisecond (not accounting for network latency) and handle thousands of reads and writes per second. A document database cluster can manage even higher document commit rates at the expense of potential inconsistency. Some graph databases can also handle transactional use cases.

Fundamentally, OLTP databases work well as application backends when thousands or even millions of users might be interacting with the application simultaneously, updating and writing data concurrently. OLTP systems are less suited to use cases driven by analytics at scale, where a single query must scan a vast amount of data.

Online Analytical Processing System

An *online analytical processing* (OLAP) system is built to run large analytics queries and is typically inefficient at handling lookups of individual records. For example, modern column databases are optimized to scan large volumes of data, dispensing with indexes to improve scalability and scan performance.

Any query typically involves scanning a minimal data block, often 100 MB or more in size. Trying to look up thousands of individual items per second in such a system will bring it to its knees unless it is combined with a caching layer designed for this use case. Note that we're using the term *OLAP* to refer to any database system that supports high-scale interactive analytics queries; we are not limiting ourselves to systems that support OLAP cubes (multidimensional arrays of data). The *online* part of OLAP implies that the system constantly listens for incoming queries, making OLAP systems suitable for interactive analytics.

OLAPs are typically storage and query systems for analytics. Data engineers often need to read data from an OLAP system. For example, a data warehouse might serve data used to train an ML model. Or, an OLAP system might serve a reverse ETL workflow, where derived data in an analytics system is sent back to a source system, such as a CRM, SaaS platform, or transactional application.

Change Data Capture:

Change data capture (CDC) is a method for extracting each change event (insert, update, delete) that occurs in a database. CDC is frequently Improved to coping between databases in near real time or create an event stream for downstream processing.

CDC is handled differently depending on the database technology. Relational data bases often generate an event log stored directly on the database server that can be processed to create a stream. Many cloud NoSQL databases can send a log or event stream to a target storage location.

Logs:

A *log* captures information about events that occur in systems. For example, a log may capture traffic and usage patterns on a web server. Our PC's operating system (Windows, macOS, Linux) logs events as the system boots and when applications start or crash, for example.

Logs are a rich data source, potentially valuable for downstream data analysis, ML, and automation. Here are a few familiar sources of logs:

- Operating systems
- Applications
- Servers
- Containers
- Networks
- IoT devices

All logs track events and event metadata. At a minimum, a log should capture who, what, and when:

Who
The human, system, or service account associated with the event (e.g., a web browser user agent or a user ID)

What happened
The event and related metadata

When
The timestamp of the event

Log encoding

Logs are encoded in a few ways:

Binary-encoded logs
These encode data in a custom compact format for space efficiency and fast I/O.

Semi Structured logs
These are encoded as text in an object serialization format (JSON, more often than not). Semi structured logs are machine-readable and portable. However, they are much less efficient than binary logs. And though they are nominally machine-readable, extracting value from them often requires significant custom code.

Plain-text (unstructured) logs
These essentially store the console output from software. These logs can provide helpful information for data scientists and ML engineers, though extracting useful information from the raw text data might be complicated.

Log resolution

Logs are created at various resolutions and log levels.

The log *resolution* refers to the amount of event data captured in a log. For example, database logs capture enough information from database events to allow reconstructing the database state at any point in time.

On the other hand, capturing all data changes in logs for a big data system often isn’t practical. Instead, these logs may note only that a particular type of commit event has occurred.

The *log level* refers to the conditions required to record a log entry, specifically concerning errors and debugging. Software is often configurable to log every event or to log only errors.

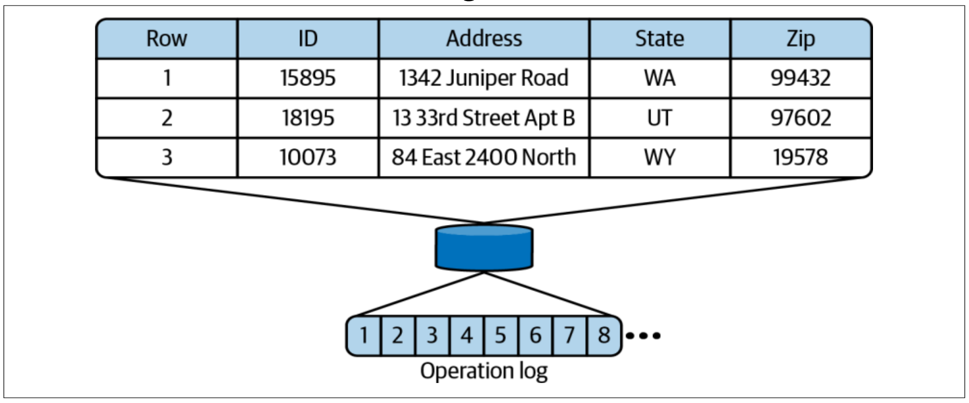
Log latency: Batch or real time

Batch logs are often written continuously to a file. Individual log entries can be written to a messaging system such as Kafka or Pulsar for real-time applications.

Database Logs:

Database logs are essential enough that they deserve more detailed coverage. Write-ahead logs—typically, binary files stored in a specific database-native format—play a crucial role in database guarantees and recoverability. The database server receives write and update requests to a database table (see Figure 5-3), storing each operation in the log before acknowledging the request. The acknowledgment comes with a log-associated guarantee: even if the server fails, it can recover its state on reboot by completing the unfinished work from the logs.

Database logs are extremely useful in data engineering, especially for CDC to generate event streams from database changes.



Row	ID	Address	State	Zip
1	15895	1342 Juniper Road	WA	99432
2	18195	13 33rd Street Apt B	UT	97602
3	10073	84 East 2400 North	WY	19578

Figure 5-3. Database logs record operations on a table

CRUD

CRUD, which stands for *create*, *read*, *update*, and *delete*, is a transactional pattern commonly used in programming and represents the four basic operations of persistent storage. CRUD is the most common pattern for storing application state in a database. A basic tenet of CRUD is that data must be created before being used. After the data has been created, the data can be read and updated. Finally, the data may need to be destroyed. CRUD guarantees these four operations will occur on data, regardless of its storage.

CRUD is a widely used pattern in software applications, and you'll commonly find CRUD used in APIs and databases. For example, a web application will make heavy use of CRUD for RESTful HTTP requests and storing and retrieving data from a database.

As with any database, we can use snapshot-based extraction to get data from a database where our application applies CRUD operations. On the other hand, event extraction with CDC gives us a complete history of operations and potentially allows for near real-time analytics.

Source System Practical Details:

This section discusses the practical details of interacting with modern source systems. We'll get into the details of commonly encountered databases, APIs, and other aspects. This information will have a shorter shelf life than the main ideas discussed previously; popular API frameworks, databases, and other details will continue to change rapidly.

1.Databases:

In this section, we'll look at common source system database technologies that we'll encounter as a data engineer and high-level considerations for working with these systems. There are as many types of databases as there are use cases for data.

Major considerations for understanding database technologies:

Database management system
A database system used to store and serve data. Abbreviated as DBMS, it consists of a storage engine, query optimizer, disaster recovery, and other key components for managing the database system.

Lookups
How does the database find and retrieve data? Indexes can help speed up lookups, but not all databases have indexes. Know whether your database uses indexes; if so, what are the best patterns for designing and maintaining them? Understand how to leverage for efficient extraction. It also helps to have a basic knowledge of the major types of indexes, including B-tree and log-structured merge-trees (LSM).

Query optimizer
Does the database utilize an optimizer? What are its characteristics? Scaling and distribution Does the database scale with demand? What scaling strategy does it deploy? Does it scale horizontally (more database nodes) or vertically (more resources on a single machine)?

Modeling patterns
What modeling patterns work best with the database.

CRUD
How is data queried, created, updated, and deleted in the database? Every type of database handles CRUD operations differently.

Consistency
Is the database fully consistent, or does it support a relaxed consistency model (e.g., eventual consistency)? Does the database support optional consistency modes for reads and writes (e.g., strongly consistent reads)?

Relational databases
A *relational database management system* (RDBMS) is one of the most common application backends. Relational databases were developed at IBM in the 1970s and popularized by Oracle in the 1980s. The growth of the internet saw the rise of the LAMP stack (Linux, Apache web server, MySQL, PHP) and an explosion of vendor and open source RDBMS options. Even with the rise of NoSQL databases, relational databases have remained extremely popular. Data is stored in a table of *relations* (rows), and each relation contains multiple *fields* (columns); see Figure 5-7. Note that we use the terms *column* and *field* interchangeably throughout this book. Each relation in the table has the same *schema* (a sequence of columns with assigned static types such as string, integer, or float). Rows are typically stored as a contiguous sequence of bytes on disk.

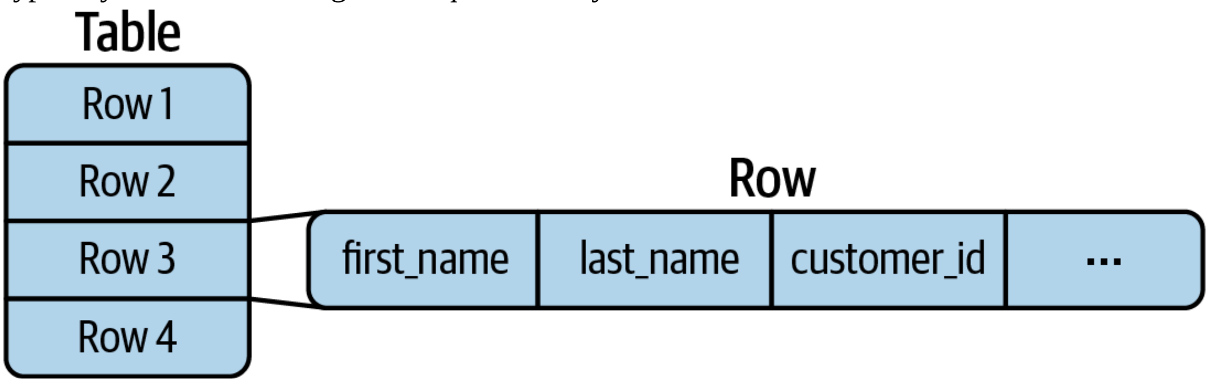


Figure 5-7. RDBMS stores and retrieves data at a row level

Nonrelational databases: NoSQL

NoSQL, or **"Not Only SQL,"** is a database management system (DBMS) designed to handle large volumes of unstructured and semi-structured data. Unlike traditional relational databases that use tables and pre-defined schemas, NoSQL databases provide **flexible data models** and support **horizontal scalability**, making them ideal for modern applications that require real-time data processing.

Types of NoSQL Databases

NoSQL databases are generally classified into four main categories based on how they store and retrieve data

Document databases

Store data in **JSON, BSON, or XML** format.

Data is stored as **documents** that can contain varying attributes.

Examples: **MongoDB, CouchDB, Cloudant**

Ideal for content management systems, user profiles, and catalogs where flexible schemas are needed.

Key-value stores

Data is stored as key-value pairs, making retrieval **extremely fast**.

Optimized for **caching and session storage**.

Examples: **Redis, Memcached, Amazon DynamoDB**

Perfect for applications requiring **session management, real-time data caching**, and leaderboards.

Column-family stores

Data is stored in **columns rather than rows**, enabling high-speed analytics and distributed computing.

Efficient for **handling large-scale data** with high write/read demands.

Examples: **Apache Cassandra, HBase, Google Bigtable**

Great for **time-series data, IoT applications**, and big data analytics.

Graph databases

Data is stored as **nodes and edges**, enabling complex relationship management.

Best suited for **social networks, fraud detection, and recommendation engines**.

Examples: **Neo4j, Amazon Neptune, ArangoDB.**

Useful for applications requiring **relationship-based queries** such as fraud detection and social network analysis.

2.APIs

APIs are the way in which we can exchange the data in the cloud, for SaaS platforms, and between internal company systems. Many types of API interfaces exist across the web, but we are principally interested in those built around HTTP, the most popular type on the web and in the cloud.

REST

REST stands for *representational state transfer*. This set of practices and philosophies for building HTTP web APIs. REST is built around HTTP verbs, such as GET and PUT; in practice, modern REST uses only a handful of the verb mappings outlined in the original dissertation.

GraphQL

GraphQL was created at **Facebook** as a query language for application data and an alternative to generic REST APIs. Whereas REST APIs generally restrict your queries to a specific data model, GraphQL opens up the possibility of retrieving multiple data models in a single request. This allows for more flexible and expressive queries than with REST. GraphQL is built around JSON and returns data in a shape resembling the JSON query.

RPC and gRPC

A *remote procedure call* (RPC) is commonly used in distributed computing.
It allows you to run a procedure on a remote system.
gRPC is a remote procedure call library developed internally at Google in 2015 and later released as an open standard. Many Google services, such as Google Ads and GCP, offer gRPC APIs. gRPC is built around the Protocol Buffers open data serialization standard, also developed by Google.

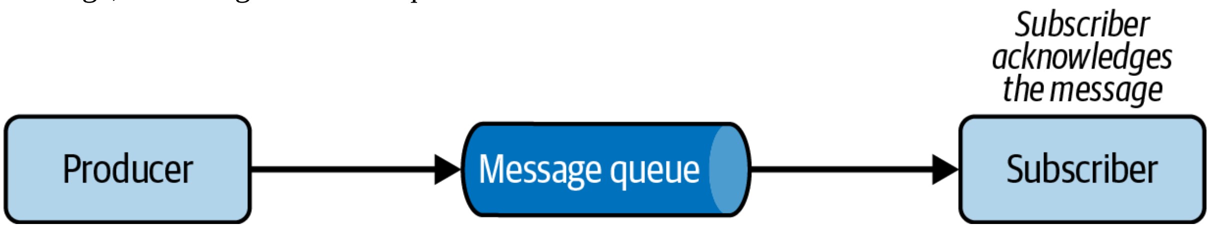
3.Data Sharing

The core concept of cloud data sharing is that a multitenant system supports security policies for sharing data among tenants. Concretely, any public cloud object storage system with a fine-grained permission system can be a platform for data sharing. Popular cloud data-warehouse platforms also support data-sharing capabilities. Of course, data can also be shared through download or exchange over email, but amultitenant system makes the process much easier.

4. Message Queues and Event-Streaming Platforms

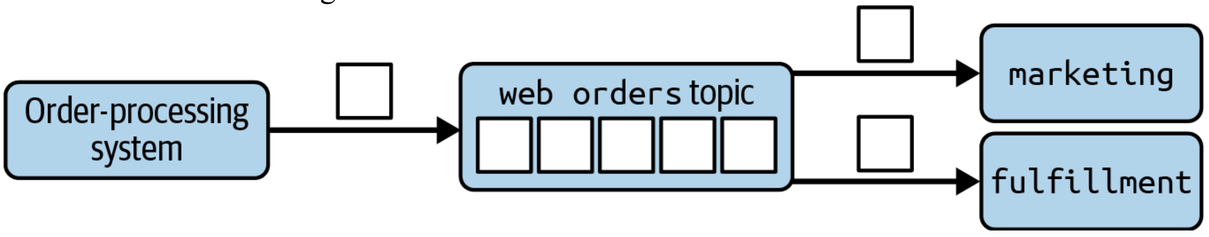
Message queues

A *message queue* is a mechanism to asynchronously send data (usually as small individual messages, in the kilobytes) between systems. Data is published to a message queue and is delivered to one or more subscribers (Figure 5-9). The subscriber acknowledges receipt of the message, removing it from the queue.



Message ordering and delivery

The order in which messages are created, sent, and received can significantly impact downstream subscribers. In general, order in distributed message queues is a tricky problem. Message queues often apply a fuzzy notion of order and first in, first out (FIFO). Strict FIFO means that if message A is ingested before message B, message A will always be delivered before message B. In practice, messages might be published and received out of order, especially in highly distributed message systems.
Delivery frequency. Messages can be sent exactly once or at least once. If a message is sent *exactly once*, then after the subscriber acknowledges the message, the message disappears and won't be delivered again.


Order-processing system
web orders topic
marketing
fulfillment