The complete guide to System Design in 2024

32 min read

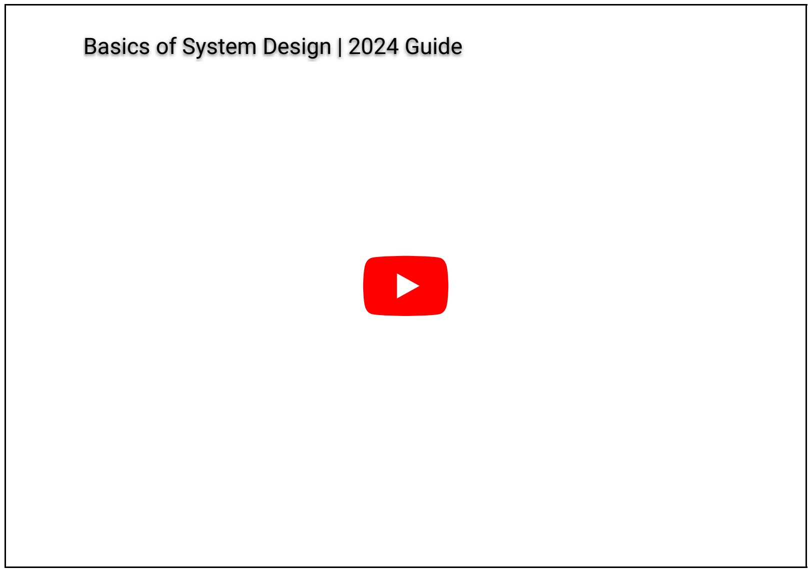
Erin Schaffer

Why should I learn system design?

Over the last two decades, there have been a lot of advancements in large-scale web applications. These advancements have redefined the way we think about software development. All of the apps and services that we use daily, like Facebook, Instagram, and Twitter, are scalable systems. Billions of people worldwide access these systems concurrently, so they need to be designed to handle large amounts of traffic and data. This is where system design comes in.

As a software developer, you'll be increasingly expected to understand system design concepts and how to apply them. In the early stages of your career, learning system design will allow you to tackle software design problems with more confidence and apply design principles to your everyday work. As you progress through your career and begin interviewing for higher-level positions, system design will become a larger part of your interview process. So, no matter what your level is, system design matters to you.

Because of its increasing importance, we wanted to create a resource to help you navigate the world of system design. This guide details the fundamental concepts of system design and also links you to relevant resources to help you gain a deeper understanding and acquire real-world, hands-on experience.



This guide covers:

> For a guide specific to the System Design Interview, check out our popular course: Grokking the Modern System Design Interview for Engineers & Managers.

What is System Design?

System design is the **process of defining the architecture, interfaces, and data for a system** that satisfies specific requirements. System design meets the needs of your business or organization through coherent and efficient systems. Once your business or organization determines its requirements, you can begin to build them into a physical system design that addresses the needs of your customers. The way you design your system will depend on whether you want to go for custom development, commercial solutions, or a combination of the two.

System design requires a systematic approach to building and engineering systems. A good system design requires you to think about everything in an infrastructure, from the hardware and software, all the way down to the data and how it's stored.

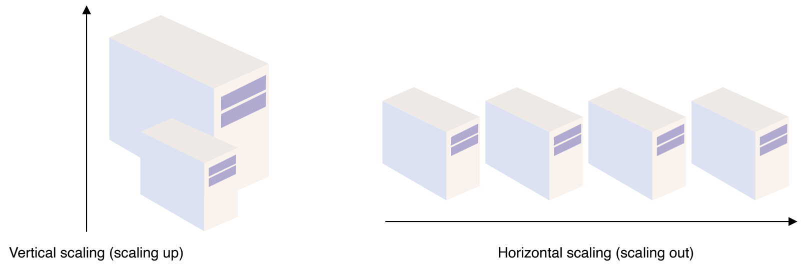
Learn more about the basics of system design.

System Design fundamentals

Horizontal and vertical scaling

Scalability refers to an application's ability to **handle and withstand an increased workload without sacrificing latency**. An application needs solid computing power to scale well. The servers should be powerful enough to handle increased traffic loads. There are two main ways to scale an application: horizontally and vertically.

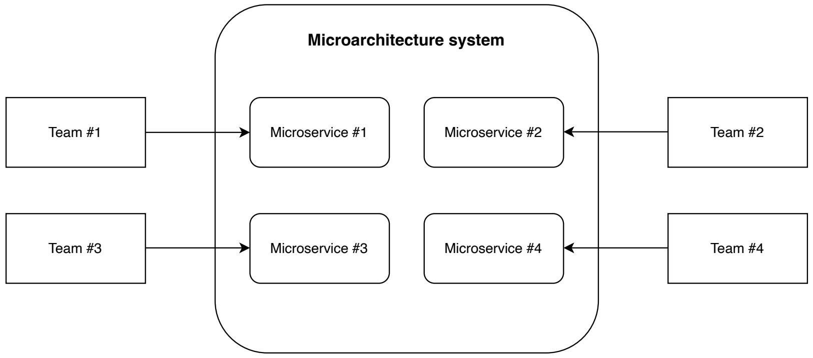
Horizontal scaling, or *scaling out*, means **adding more hardware** to the existing hardware resource pool. It increases the computational power of the system as a whole. Vertical scaling, or *scaling up*, means **adding more power** to your server. It increases the power of the hardware running the application.


Vertical scaling (scaling up)
Horizontal scaling (scaling out)

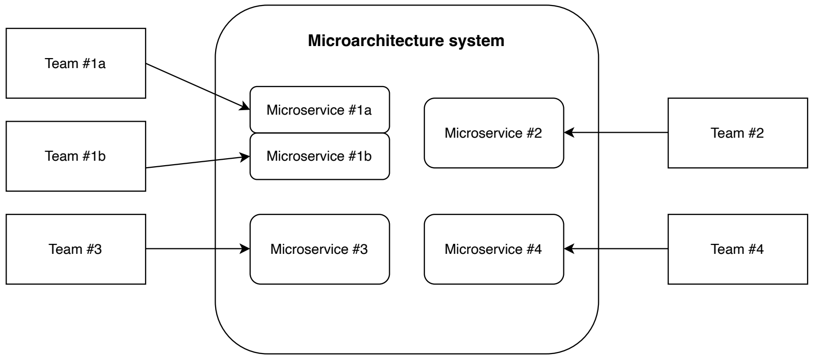
There are pros and cons to both types of scaling. You'll come across scenarios where you need to consider the tradeoffs and decide which type of scaling is best for your use case. To dive deeper into scaling, check out the scalability module in our hands-on course Web Application and Software Architecture 101. You'll learn about the benefits and risks of scaling, primary bottlenecks that hurt application scalability, and more.

Microservices

Microservices, or *microservice architecture*, is an architectural style that **structures an application using loosely coupled services**. It divides a large application into a collection of separate, modular services. These modules can be independently developed, deployed, and maintained.



In a microarchitecture system, each microservice has a team working on it.



As Team #1 outgrows the limits of a single microservice, we can easily scale up the microservices to meet the changing needs of the team.

Microservices operate at a much faster and more reliable speed than traditional monolithic applications. Since the application is broken down into independent services, every service has its own logic and codebase. These services can communicate with one another through Application Programming Interfaces (APIs).

Microservices are ideal if you want to develop a more scalable application. With microservices, it's much **easier to scale your applications** because of their modern capabilities and modules. If you work with a large or growing organization, microservices are great for your team because they're easier to scale and customize over time. To learn more about microservices and their benefits, drawbacks, technology stacks, and more, check out this microservices architecture tutorial.

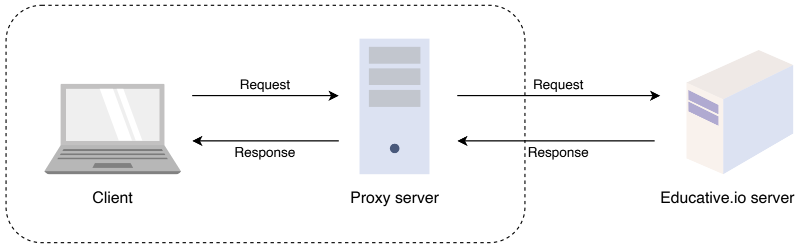
Proxy servers

A proxy server, or *forward proxy*, acts as a **channel between a user and the internet**. It separates the end-user from the website they're browsing.

Proxy servers not only forward user requests but also provide many benefits, such as:

- Improved security
- Improved privacy
- Access to blocked resources
- Control of the internet usage of employees and children
- Cache data to speed up requests

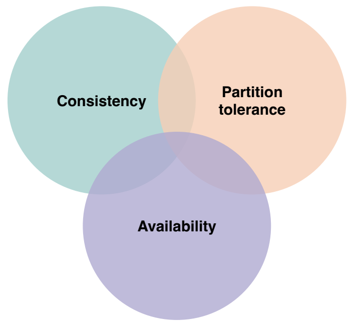
Whenever a user sends a request for an address from the end server, the traffic flows through a proxy server on its way to the address. When the request comes back to the user, it flows back through the same proxy server which then forwards it to the user.



Forward proxy

CAP theorem

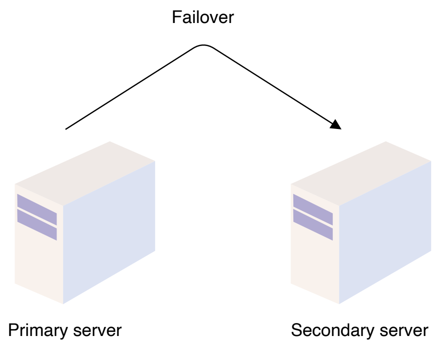
The CAP theorem is a **fundamental theorem** within the field of system design. It states that a distributed system can only provide two of three properties simultaneously: consistency, availability, and partition tolerance. The theorem formalizes the tradeoff between consistency and availability when there's a partition.



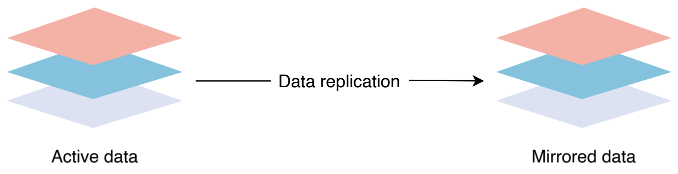
Learn more about CAP theorem.

Redundancy and replication

Redundancy is the process of **duplicating critical components of a system** with the intention of increasing a system's reliability or overall performance. It usually comes in the form of a backup or fail-safe. Redundancy plays a critical role in removing single points of failure in a system and providing backups when needed. For example, if we have two instances of a service running in production and one of those instances fails, the system can fail over to another one.



Replication is the process of **sharing information to ensure consistency between redundant resources**. You can replicate software or hardware components to improve reliability, fault-tolerance, or accessibility. Replication is used in many database management systems (DBMS), typically with a primary-replica relationship between the original and its copies. The primary server receives all of the updates, and those updates pass through the replica servers. Each replica server outputs a message when it successfully receives the update.



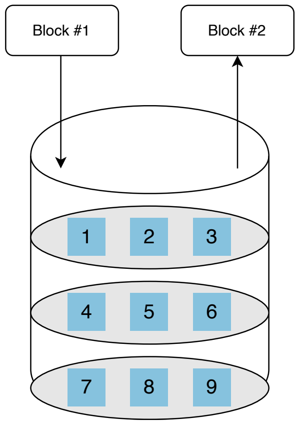
Learn more about redundancy and replication in the Web Application & Software Architecture module of Scalability & System Design for Developers.

Storage

Data is at the center of every system. When designing a system, we need to consider how we're going to store our data. There are various storage techniques that we can implement depending on the needs of our system.

Block storage

Block storage is a data storage technique where **data is broken down into blocks of equal sizes**, and each individual block is given a unique identifier for easy accessibility. These blocks are stored in physical storage. As opposed to adhering to a fixed path, blocks can be stored anywhere in the system, making more efficient use of the resources.



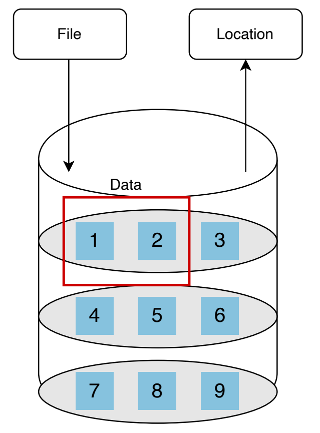
Fixed-size blocks in specific locations

Learn more about block storage.

File storage

File storage is a **hierarchical storage methodology**. With this method, the data is stored in files. The files are stored in folders, which are then stored in directories. This storage method is only good for a limited amount of data, primarily structured data.

> As the size of the data grows beyond a certain point, this data storage method can become a hassle.

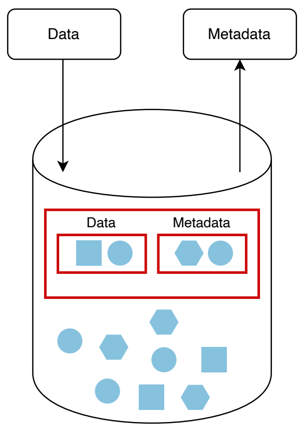


Specific folders in a fixed, logical order

Learn more about file storage.

Object storage

Object storage is the storage **designed to handle large amounts of unstructured data**. Object storage is the preferred data storage method for data archiving and data backups because it offers **dynamic scalability**. Object storage isn't directly accessible to an operating system. Communication happens through RESTful APIs at the application level. This type of storage provides immense flexibility and value to systems, because backups, unstructured data, and log files are important to any system. If you're designing a system with large datasets, object storage would work well for your organization.

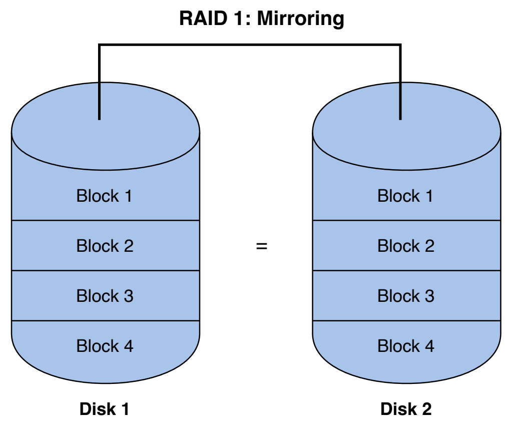


Scalable storage driven by metadata

Learn more about object storage.

Redundant Disk Arrays (RAID)

A redundant array of inexpensive disks, or RAID, is a **technique to use multiple disks** in concert to build a faster, bigger, and more reliable disk system. Externally, a RAID looks like a disk. Internally, it's a complex tool, consisting of multiple disks, memory, and one or more processors to manage the system. A hardware RAID is similar to a computer system but is specialized for the task of managing a group of disks. There are different levels of RAID, all of which offer different functionalities. When designing a complex system, you may want to implement RAID storage techniques.



RAID 1 technique: Mirroring

Learn more about the advantages and disadvantages of RAID.

Message queues

A message queue is a queue that **routes messages from a source to a destination**, or from the sender to the receiver. It follows the FIFO (first in first out) policy. The message that is sent first is delivered first. Message queues **facilitate asynchronous behavior**, which allows modules to communicate with each other in the background without hindering primary tasks. They also facilitate cross-module communication and provide temporary storage for messages until they are processed and consumed by the consumer.

Learn more about message queues in the Web Application & Software Architecture module of Scalability & System Design for Developers.

Kafka

Apache Kafka started in 2011 as a messaging system for LinkedIn but has since grown to become a popular **distributed event streaming platform**. The platform is capable of handling trillions of records per day. Kafka is a distributed system consisting of servers and clients that communicate through a TCP network protocol. The system allows us to read, write, store, and process events. Kafka is primarily used for building data pipelines and implementing streaming solutions.

While Kafka is a popular messaging queue option, there are other popular options as well. To learn more about which messaging queue to use, we recommend the following resources:

- Kafka vs RabbitMQ
- Kafka vs Kinesis
- Kafka vs Flink

File systems

File systems are processes that **manage how and where data on a storage disk is stored**. It manages the internal operations of the storage disk and explains how users or applications can access disk data. File systems manage multiple operations, such as:

- File naming
- Storage management
- Directories
- Folders
- Access rules

Without file systems, it would be hard to identify files, retrieve files, or manage authorizations for individual files.

Google File System (GFS)

Google File System (GFS) is a **scalable distributed file system designed for large data-intensive applications**, like Gmail or YouTube. It was built to handle batch processing on large data sets. GFS is designed for system-to-system interaction, rather than user-to-user interaction. It's scalable and fault-tolerant. The architecture consists of GFS clusters, which contain a single master and multiple ChunkServers that can be accessed by multiple clients.

It's common to be asked to design a distributed file system, such as GFS, in system design interviews. To prepare for this xview question, check out the System Design Interview resources in Grokking Modern System Design for Engineers & Managers.

Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS) is a **distributed file system** that handles large sets of data and runs on commodity hardware. It was built to store unstructured data. HDFS is a more simplified version of GFS. A lot of its architectural decisions are inspired by the GFS design. HDFS is built around the idea that the most efficient data processing pattern is a "write once, read many times" pattern.

It's common to be asked to design a distributed file storage system, such as HDFS, in system design interviews. To prepare for this system design interview question, check out Grokking Modern System Design for Engineers and Managers.

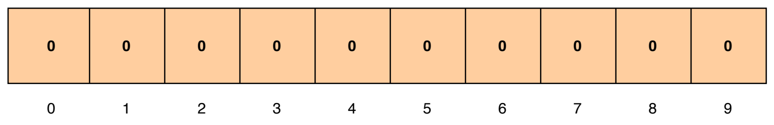
System Design patterns

Knowing system design patterns is very important because they can be applied to all types of distributed systems. They also play a major role in system design interviews. System design patterns refer to common design problems related to distributed systems and their solutions. Let's take a look at some commonly used patterns.

Bloom filters

Bloom filters are **probabilistic data structures** designed to answer the set membership question: *Is this element present in the set?*

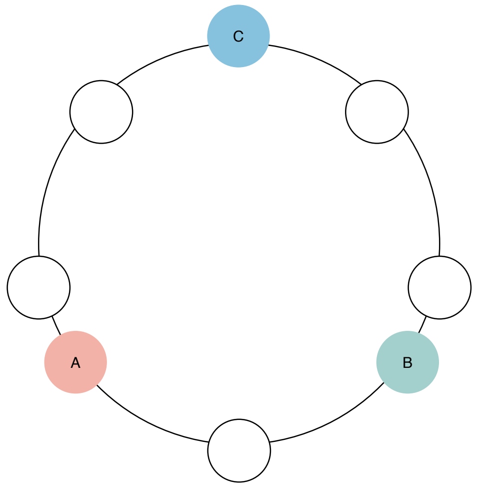
Bloom filters are highly space-efficient and do not store actual items. They determine whether an item *does not exist* in a set or if an item *might exist* in a set. They can't tell if an item is *definitely present* in a set. An empty Bloom filter is a bit vector with all bits set to zero. In the graphic below, each cell represents a bit. The number below the bit is its index in a 10-bit vector.



An empty Bloom filter

Consistent hashing

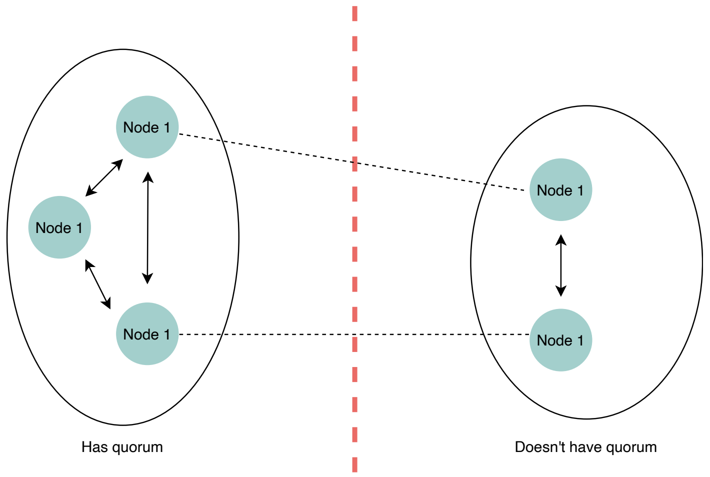
Consistent hashing **maps data to physical nodes** and ensures that only a small set of keys move when servers are added or removed. Consistent hashing stores the data managed by a distributed system in a ring. Each node in the ring is assigned a range of data. This concept is important within distributed systems and works closely with data partitioning and data replication. Consistent hashing also comes up in system design interviews.



Learn more about consistent hashing.

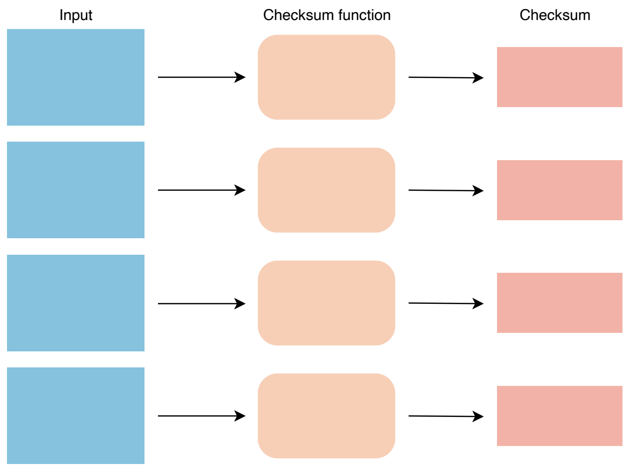
Quorum

A quorum is the minimum number of servers on which a distributed operation needs to be performed successfully before declaring the operation's overall success.



Checksum

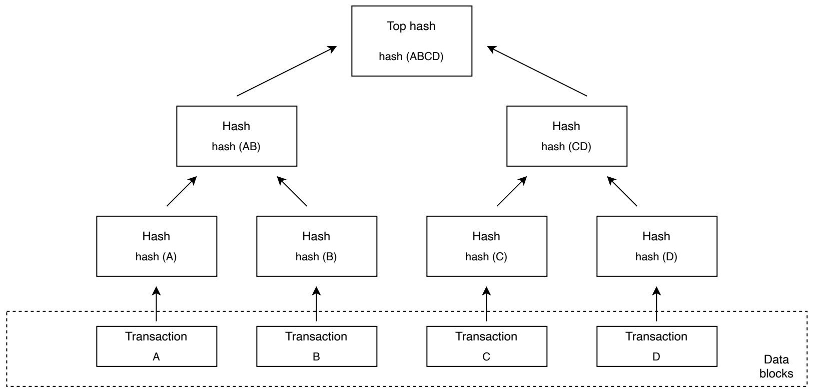
When moving data between components in a distributed system, it's possible that the data fetched from a node may arrive corrupted. This corruption occurs because of faults in storage devices, networks, software, etc. When a system is storing data, it computes a checksum of the data and stores the checksum with the data. When a client retrieves data, it verifies that the data received from the server matches the stored checksum.



Learn more about Checksum.

Merkle trees

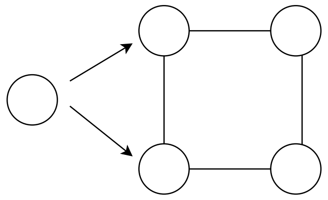
A Merkle tree is a **binary tree of hashes**, in which each internal node is the hash of its two children, and each leaf node is a hash of a portion of the original data. Replicas can contain a lot of data. Splitting up the entire range to calculate checksums for comparison is not very feasible, because there's so much data to be transferred. Merkle trees enable us to easily compare replicas of a range.



Learn more about Merkle trees.

Leader election

Leader election is the process of **designating a single process as the organizer of tasks** distributed across several computers. It's an algorithm that enables each node throughout the network to recognize a particular node as the task leader. Network nodes communicate with each other to determine which of them will be the leader. Leader election improves efficiency, simplifies architectures, and reduces operations.



Learn more about leader election.

Databases

Relational databases

Relational databases, or *SQL databases*, are **structured**. They have **predefined schemas**, just like phone books that store numbers and addresses. SQL databases store data in rows and columns. Each row contains all of the information available about a single entity, and each column holds all of the separate data points. Popular SQL databases include:

- MySQL
- Oracle
- MS SQL Server
- SQLite
- PostgreSQL
- MariaDB

MySQL

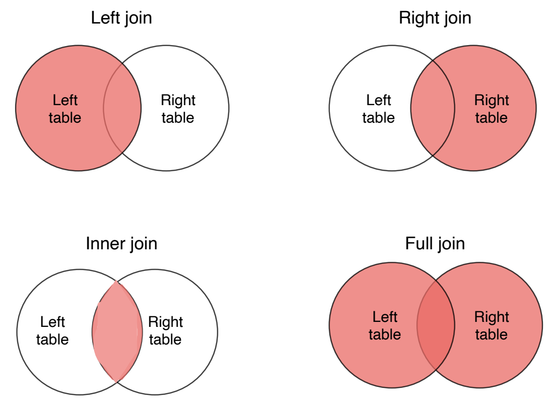
MySQL is an open-source relational database management system (RDBMS) that stores data in tables and rows. It uses SQL (structured query language) to transfer and access data, and it uses SQL joins to simplify querying and correlation. It follows client-server architecture and supports multithreading.

PostgreSQL

PostgreSQL, also known as Postgres, is an open-source RDBMS that emphasizes extensibility and SQL compliance. Postgres employs SQL to access and manipulate the database. It uses its own version of SQL called PL/pgSQL, which can perform more complex queries than SQL. Postgres transactions follow the ACID principle. Because it has a relational structure, the whole schema needs to be designed and configured at the time of creation. Postgres databases use foreign keys, which allow us to keep our data normalized.

SQL joins

SQL joins allow us to **access information from two or more tables at once**. They also keep our databases normalized, which ensures that data redundancy is low. When data redundancy is low, we can decrease the amount of data anomalies in our application when we delete or update records.



Non-relational databases

shopping preferences. There are different types of NoSQL. The most common types include:

- Key-value stores, such as Redis and DynamoDB
- Document databases, such as MongoDB and CouchDB
- Wide-column databases, such as Cassandra and HBase
- Graph databases, such as Neo4J and InfiniteGraph

MongoDB

MongoDB is a NoSQL, non-relational database management system (DBMS) that uses documents instead of tables or rows for data storage. This data model makes it possible to manipulate related data in a single database operation. MongoDB documents use JSON-like documents and files that are JavaScript supported. The document fields can vary, making it easy to change the structure over time.

How to choose a database

Databases are a basic foundation of software development. They serve many different purposes for building projects of all sizes and types. When choosing your database structure, it's important to **factor in speed, reliability, and accuracy**. We have relational databases that can guarantee data validity, and we have non-relational databases that can guarantee eventual consistency. When choosing your database structure, it's important to factor in database fundamentals, such as:

- ACID
- BASE
- SQL joins
- Normalization
- Persistence
- Etc.

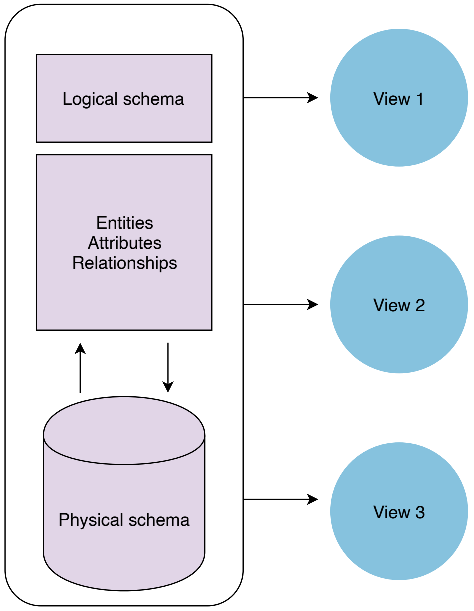
Database decisions are an important part of system design interviews, so it's important to get comfortable with making decisions based on unique use cases. The database you choose will depend upon your project. To learn more about how to choose the right database for your project, we recommend the following resources:

- MongoDB vs PostgreSQL
- MongoDB vs MySQL
- SQL vs. NoSQL

Database schemas

Database schemas are **abstract designs that represent the storage of the data in a database**. They describe the organization of data and the relationships between tables in a given database. You plan database schemas in advance so you know what components are necessary and how they'll connect to each other. A database schema doesn't hold data but instead describes the shape of the data and how it relates to other tables or models. An entry in a database is an instance of a database schema.

There are two main database schema types that define different parts of the schema: **logical and physical**.



Database schemas include:

- All important or relevant data
- Consistent formatting for all data entries
- Unique keys for all entries and database objects
- Each column in a table has a name and a data type

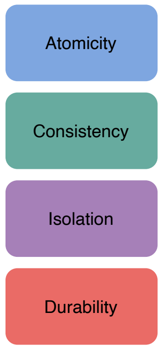
The size and complexity of a database schema depend on the size of the project. The visual style of database schemas allows you to properly structure your database and its relationships before jumping into the code. The process of planning a database design is called **data modeling**. Database schemas are important for designing DBMS and RDBMS.

Database queries

A database query is a **request to access data from a database to manipulate or retrieve it**. It's most closely associated with CRUD operations. Database queries allow us to perform logic with the information we get in response to the query. There are many different approaches to queries, from using query strings to writing with a query language, to using a QBE (Query by Example) like GraphQL.

ACID properties

To **maintain the integrity of a database**, all transactions must obey ACID properties. ACID is an acronym that stands for atomicity, consistency, isolation, and durability.



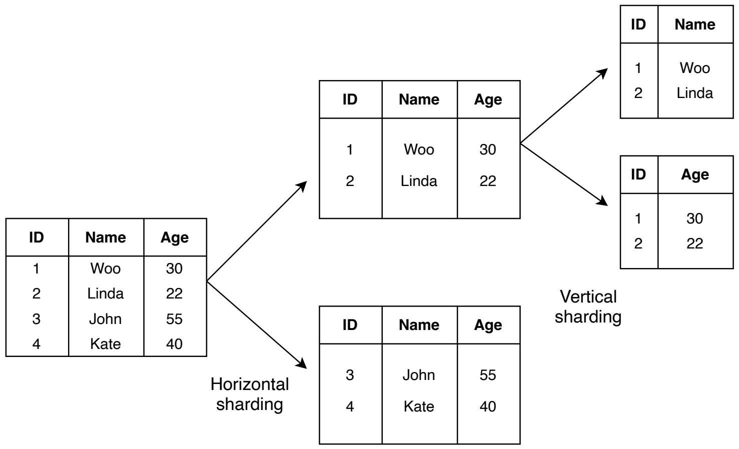
- **Atomicity**: A transaction is an atomic unit. All of the instructions within a transaction will successfully execute, or none of them will execute.
- **Consistency**: A database is initially in a consistent state, and it should remain consistent after every transaction.
- **Isolation**: If multiple transactions are running concurrently, they shouldn't be affected by each other, meaning that the result should be

the same as the result obtained if the transactions were running sequentially.

- **Durability**: Changes that have been committed to the database should remain, even in the event of a software or system failure.

Database sharding and partitioning

When sharding a database, you make partitions of data so that the **data is divided into various smaller, distinct chunks** called shards. Each shard could be a table, a Postgres schema, or a different physical database held on a separate database server instance. Some data within the database remains present in all shards, while some only appear in single shards. These two situations can be referred to as vertical sharding and horizontal sharding. Let's take a look at a visual:



To shard your data, you need to determine a **sharding key** to partition your data. The sharding key can either be an indexed field or indexed compound fields that exist in every document in the collection. There's no general rule for determining your sharding key. It all depends on your application.

Sharding allows your application to make fewer queries. When it receives a request, the application knows where to route the request. This means that it has to look through less data rather than going through the entire database. Sharding improves your application's overall performance and scalability.

Data partitioning is a technique that breaks up a big database into smaller parts. This process allows us to split our database across multiple machines to improve our application's performance, availability, load balancing, and manageability.

Database indexing

Database indexing allows you to make it **faster and easier to search through your tables** and find the rows or columns that you want. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered information. While indexes dramatically speed up data retrieval, they typically slow down data insertion and updates because of their size.

Learn more about databases for system design in Grokking Modern System Design for Engineers and Managers.

What are distributed systems?

Distributed systems make it easier for us to **scale our applications at exponential rates**. Many top tech companies use complex distributed systems to handle billions of requests and perform updates without downtime. A distributed system is a collection of computers that work together to form a single computer for the end user. All of the computers in the collection share the same state and operate concurrently. These machines can also fail independently without affecting the entire system.

Distributed systems can be difficult to deploy and maintain, but they provide many benefits, including:

- **Scaling**: Distributed systems allow you to scale horizontally to account for more traffic.
- **Modular growth**: There's almost no cap on how much you can scale.
- **Fault tolerance**: Distributed systems are more fault-tolerant than single machines.
- **Cost-effective**: The initial cost is higher than traditional systems, but because of their capacity to scale, they quickly become more cost-effective.
- **Low latency**: You can have a node in multiple locations, so traffic will hit the closest node.
- **Efficiency**: Distributed systems break complex data into smaller pieces.
- **Parallelism**: Distributed systems can be designed for parallelism, where multiple processors divide up a complex problem into smaller chunks.

Distributed system failures

A distributed system can encounter several types of failures. Four basic types of failures include:

System failure

System failures occur because of software or hardware failures. System failures usually result in the loss of the contents of the primary memory, but the secondary memory remains safe. Whenever there's a system failure, the processor fails to perform the execution, and the system may reboot or freeze.

Communication medium failure

Communication medium failures occur as a result of communication link failures or the shifting of nodes.

Secondary storage failure

A secondary storage failure occurs when the information on the secondary storage device is inaccessible. It can be the result of many different things, including node crashing, dirt on the medium, and parity errors.

Method failure

Method failures usually halt the distributed system and make it unable to perform any executions at all. A system may enter a deadlock state or do protection violations during method failures.

Distributed system fundamentals

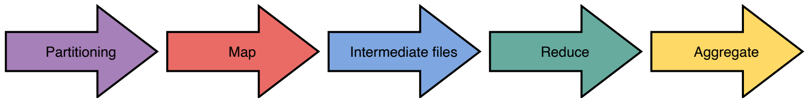
In this section, we'll cover some fundamental concepts within the field of distributed systems.

For a more comprehensive list, we recommend the following resource: Distributed Systems for Practitioners. This course establishes the basic, fundamental principles of distributed systems, and it provides many additional resources for those who want to invest more time in gaining a deeper understanding.

MapReduce

MapReduce is a framework developed by Google to **handle large amounts of data** in an efficient manner. MapReduce uses numerous servers for data management and distribution. The framework provides abstractions to underlying processes happening during the execution of user commands. A few of these processes include fault tolerance, partitioning data, and aggregating data. The abstractions **allow the user to focus on the higher-level logic** of their programs while trusting the framework to smoothly continue underlying processes.

The MapReduce workflow is as follows:



- **Partitioning**: The data is usually in the form of a big chunk. It's necessary to begin by partitioning the data into smaller, more manageable pieces that can be efficiently handled by the map workers.
- **Map**: Map workers receive the data in the form of a key-value pair. This data is processed by the map workers, according to the user-defined map function, to generate intermediate key-value pairs.
- **Intermediate files**: The data is partitioned into *R* partitions (with *R* being the number of reduce workers). These files are buffered in the memory until the primary node forwards them to the reduce workers.
- **Reduce**: As soon as the reduce workers get the data stored in the buffer, they sort it accordingly and group data with the same keys.
- **Aggregate**: The primary node is notified when the reduce workers are done with their tasks. In the end, the sorted data is aggregated together and *R* output files are generated for the user.

Learn more about MapReduce.

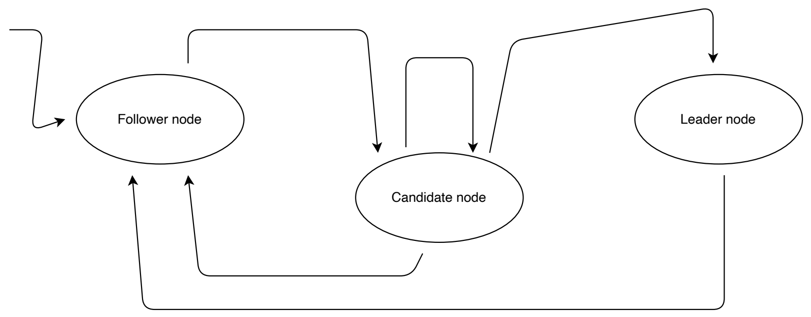
Stateless and stateful systems

Stateless and stateful systems are important concepts in the world of distributed systems. A system is either stateless or stateful. A stateless system **maintains no state of past events**. It executes based on the inputs we provide to it. Stateful systems are responsible for **maintaining and mutating a state**. To learn more about stateless vs. stateful systems and the advantages of stateless systems, check out the following resource: Distributed Systems for Practitioners.

Raft

Raft **establishes the concept of a replicated state machine and the associated replicated log of commands** as first-class citizens and supports multiple consecutive rounds of consensus by default. It requires a set of nodes that form a consensus group, or a Raft cluster. Each of these can be in one of these three states:

- Leader
- Follower
- Candidate



Learn more about Raft nodes and the implementation of Raft.

Distributed system design patterns

Design patterns give us ways to build systems that fit particular use cases. They are like **building blocks** that allow us to pull from existing knowledge rather than start every system from scratch. They also create a set of standard models for system design that help other developers see how their projects can interface with a given system.

Creational design patterns provide a baseline when building new objects. Structural patterns define the overall structure of a solution. Behavioral patterns describe objects and how they communicate with each other. Distributed system design patterns outline a software architecture for how different nodes communicate with each other, which nodes handle particular tasks, and what the process flows should look like for various tasks.

Most distributed system design patterns **fall into one of three categories** based on the functionality they work with:

- **Object communication**: Describes the messaging protocols and permissions for different components of the system to communicate
- **Security**: Handles confidentiality, integrity, and availability concerns to ensure the system is secure from unauthorized access
- **Event-driven**: Describes the production, detection, consumption, and response to system events

Learn the top 5 distributed system design patterns.

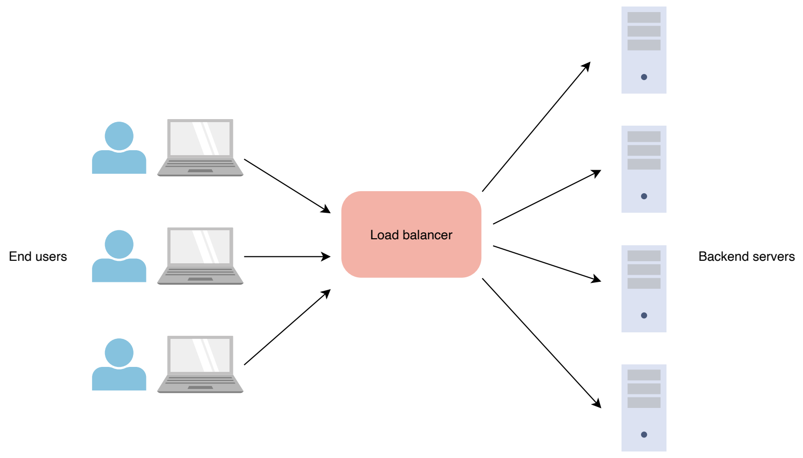
Scalable web applications

DNS and load balancing

DNS, or *domain name system*, averts the need to remember long IP addresses to visit websites by **mapping simple domain names to IP addresses**. You can set up DNS load balancing for large-scale applications and systems that need to spread user traffic across different clusters in different data centers.

Load balancing is very important to our scaling efforts. It allows us to **scale effectively with increases in traffic and stay highly available**. Load balancing is executed by load balancers, which are devices that act as reverse proxies. They’re responsible for distributing network traffic across multiple servers using different algorithms. The distribution of traffic helps avert the risks of all the traffic converging to a single machine or just a couple of machines in the cluster. If the traffic converges to only a couple of machines, this will overload them and bring them down.

Load balancing helps us avoid these problems. If a server goes down while the application is processing a user request, the load balancer automatically routes future requests to servers that are functioning.



Learn more about load balancing.

N-tier applications

N-tier applications, or *distributed applications*, are **applications that have more than three components involved**. Those components can be:

- Caches
- Message queues
- Load balancers
- Search servers
- Components involved in processing large amounts of data
- Components running heterogeneous tech, commonly known as web services

Large applications, such as Instagram, Facebook, and Uber, are n-tier applications.

Learn more about tiered applications.

HTTP and REST

HTTP stands for HyperText Transfer Protocol. This protocol **dictates the format of messages, how and when messages are sent, appropriate responses, and how messages are interpreted**. HTTP messages can be either *requests* or *responses*. HTTP APIs expose endpoints as API gateways for HTTP requests to have access to servers. They come in various forms based on their target use cases, and they can be further categorized by the architectural design principles used when they're created.

REST stands for Representational State Transfer. It's a software architectural style for implementing web services. REST is a **ruleset that defines best practices for sharing data between clients and servers**, and it emphasizes the scalability of components and the simplicity of interfaces. REST applications use HTTP methods, such as `GET`, `POST`, `DELETE`, and `PUT`.

REST APIs are API implementations that adhere to REST architectural principles. They act as **interfaces where the communication between clients and servers happens over HTTP**. REST APIs take advantage of HTTP methodologies to establish communication between clients and servers. REST also enables servers to cache responses that improve application performance.

HTTP and REST are important concepts and considerations for client-server communication in system design.

Learn more about HTTP and REST.

Stream processing

Stream processing refers to a computer programming architecture that focuses on the **real-time processing of continuous streams of data**. Popular stream processing tools include Kafka, Storm, and Flink.

Learn more about stream processing.

Caching

A cache is hardware or software that you use to **temporarily store data so it can be accessed quickly**. Caches are typically very small, which makes them cost-effective and efficient. They're used by cache clients, such as web browsers, CPUs, operating systems, and DNS servers. Accessing data from a cache is a lot faster than accessing it from the main memory or any other type of storage.

What is caching? How does it work?

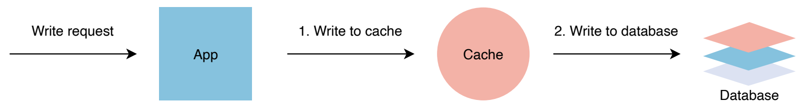
Let's say that a client wants to access some data. First, the client can check if the data is stored in the cache. If they find the data, it will immediately be returned to the client. This is called a **cache hit**. If the data isn't stored in the cache, a **cache miss** occurs. When this happens, the client obtains data from the main memory and stores it in the cache.

There are different types of caching strategies:

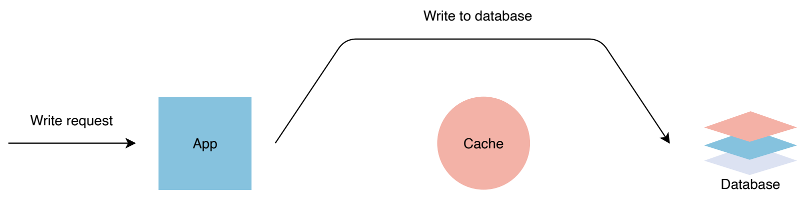
Cache invalidation

Cache invalidation is a process where a computer system declares cache entries as "invalid" and either removes or replaces them. The basic objective of this process is to ensure that when the client requests the affected content, the latest version is returned. There are three defined cache invalidation schemes:

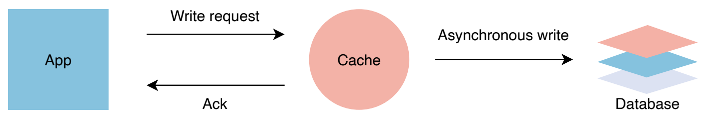
Write-through cache



Write-around cache



Write-back cache



Learn more about cache invalidation.

Cache eviction

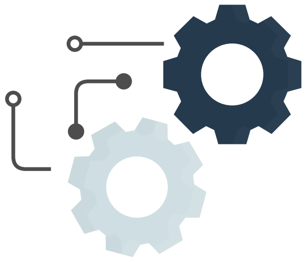
If a cache has space, data will be easily inserted. If a cache is full, some data will be evicted. What gets evicted, and why, depends on the eviction policy used. Some commonly used cache eviction policies include:

- **First in first out (FIFO)**: The cache evicts the first block accessed first

without any regard to how often or how many times it was accessed before.
- **Last in first out (LIFO)**: The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
- **Least recently used (LRU)**: The cache evicts the least recently used items first.
- **Most recently used (MRU)**: The cache evicts the most recently used items first.
- **Least frequently used (LFU)**: The cache counts how often an item is needed. The items that are used least frequently are evicted first.
- **Random replacement (RR)**: The cache randomly selects a candidate and evicts it.

Learn more about cache eviction.

Machine learning and System Design

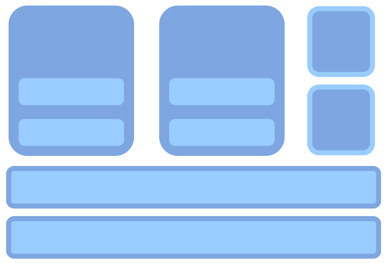


Machine learning (ML) applications and systems are increasing in popularity and are becoming more widely used throughout various industries. As these ML applications and systems continue to mature and expand, we need to begin thinking more deeply about how we design and build them. Machine learning system design is the process of **defining the software architecture, algorithms, infrastructure, and data for machine learning systems** to satisfy specific requirements.

If you want to be a machine learning engineer, you'll be expected to have solid engineering foundations and hands-on ML experiences. ML interviews share similar components to traditional coding interviews. You'll go through a similar method of problem-solving to answer questions about system design, machine learning, and machine learning system design.

The standard development cycle of machine learning includes data collection, problem, formulation, model creation, implementation of models, and enhancement of models. There are no common, standardized guidelines for approaching machine learning system design from end to end. However, we have a resource to help you approach ML system design with a top-down approach: Machine Learning System Design.

Containerization and System Design



Containerization is the **packaging of software code with its dependencies** to create a “container” that can run on any infrastructure.

We can think of containers as more lightweight versions of virtual machines (VMs) that don't need their own operating system. All containers on a host share that host's operating system, which frees up a lot of system resources. Containerization wasn't very accessible until Docker came along. Docker is an open-source containerization platform that we can use to build and run containers. Docker containers create an abstraction layer at the application layer.

Docker often gets confused with Kubernetes, which is another popular containerization tool. The two technologies complement each other and are frequently used together. While Docker is a containerization platform, Kubernetes is a containerization software that allows us to control and manage containers and VMs. With Kubernetes, you can run Docker containers and manage your containerized applications. Containers are grouped into pods, and those pods can be scaled and managed however you want. To dive deeper into these two technologies, we recommend this resource: Docker vs Kubernetes.

Similar to ML technologies, containerization technologies are also growing in popularity and are **becoming more widespread** across various industries. Because of this, the design and implementation of containerization systems are also gaining ground.

Here are a few containerization resources to help you further your learning:

- Docker for Developers
- DevOps for Developers
- Quick Start Kubernetes

The Cloud and System Design

Cloud computing **allows access to services like storage or development platforms on-demand via internet-connected offsite data centers**. Data centers are managed by third-party companies, or *cloud providers*. The cloud computing model not only addresses the problems that occur with on-premises systems, but also is more cost-effective, scalable, and convenient.

Different cloud providers offer different cloud services, such as storage, security, access management, and more. Cloud services give you the tools to be able to design and implement flexible and efficient systems. Cloud services vary in size and complexity, and there are various cloud deployment models that you can leverage. Different cloud system architectures include:

- Multi-cloud
- Hybrid cloud
- Single cloud
- Public cloud
- Private cloud

Cloud computing continues to grow in popularity. It can **play a major role in system design and architecture**. It's important to know about these services and models when designing a system.

Here's are a few cloud resources to help further your learning:

- Cloud Architecture: A Guide to Design & Architect your Cloud
- Cloud Computing 101: Master the Fundamentals
- Scalability & System Design for Developers

Continue learning about System Design

- Grokking Modern System Design for Engineers & Managers
- The complete guide to the System Design Interview
- How to prepare for the System Design Interview

WRITTEN BY
Erin Schaffer

LEGAL

Privacy Policy

Cookie Policy

Terms of Service

Business Terms of Service

Data Processing Agreement