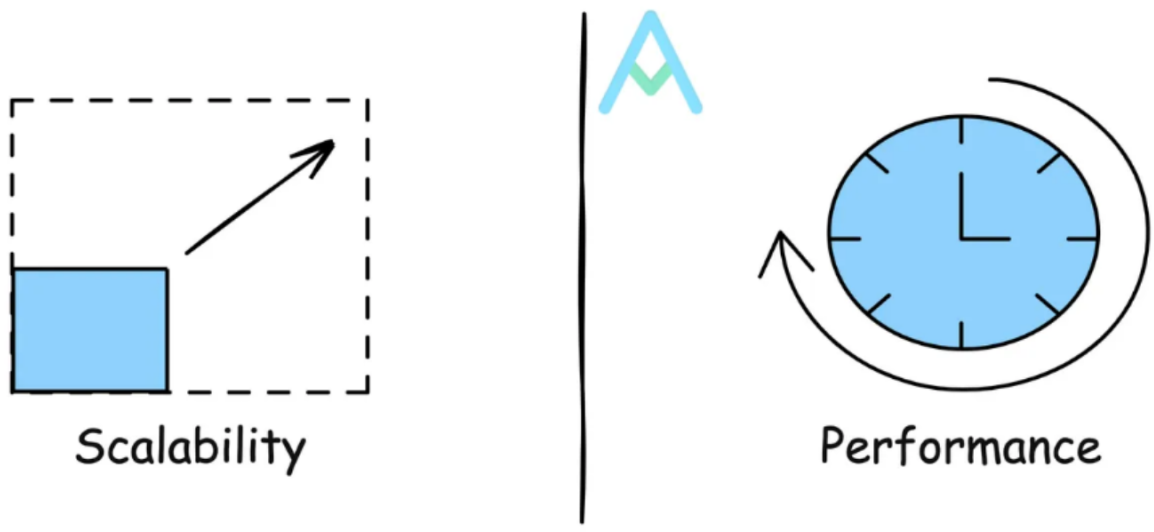
[Trade offs]

1. Scalability vs. Performance

Scalability is about size: "Can the system grow to handle more work?"

Performance is about speed: "How fast can the system complete a task?"

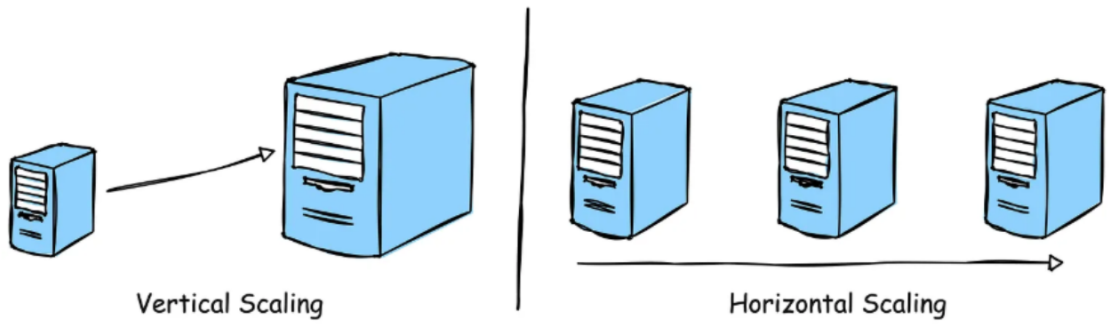


They often pull in opposite directions and improving one can sometimes impact the other.

> **Example**: Adding more machines to a system can make it more scalable, but the complexity of managing these machines and coordinating tasks across them might introduce delays affecting performance.
>
> Conversely, optimizing for performance with minimal resources may create scalability issues when user demand increases.

2. Vertical vs. Horizontal Scaling

Vertical scaling involves adding more resources to an existing server (eg. CPU, RAM), while **horizontal scaling** means adding more servers to the pool.



Scaling vertically is simpler but there's a physical limit to how much you can upgrade a single machine. It introduces a single point of failure. If the machine goes down, the entire system can become unavailable.

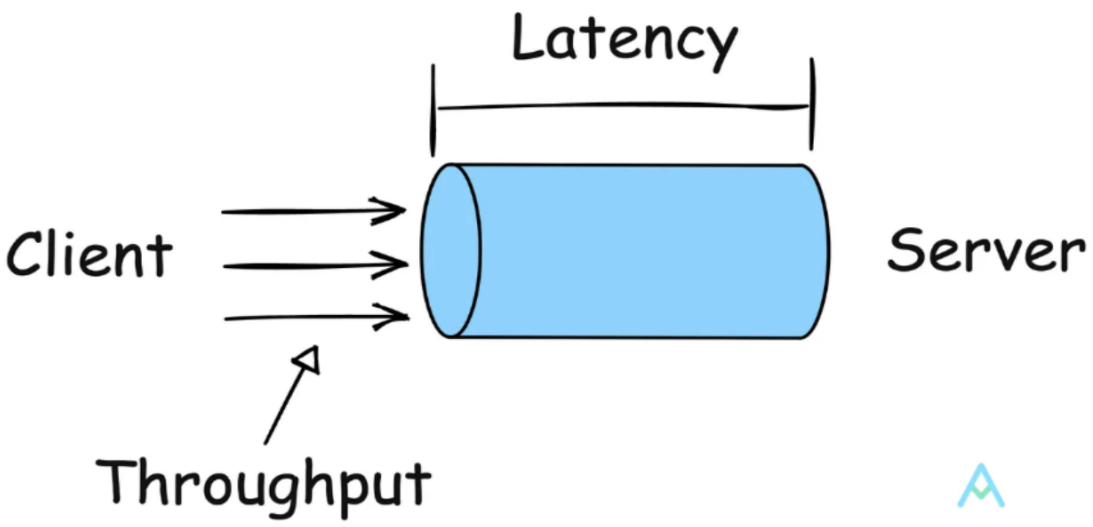
Scaling horizontally allows for almost limitless scaling but brings complexity of managing distributed systems.

> **Example:** Initially, a startup may vertically scale its server to handle increased load by adding more CPUs and RAM.
>
> As the startup grows, it might shift to horizontal scaling by adding more servers to distribute the load.

3. Latency vs. Throughput

Latency measures the time that data takes to transfer across the network.

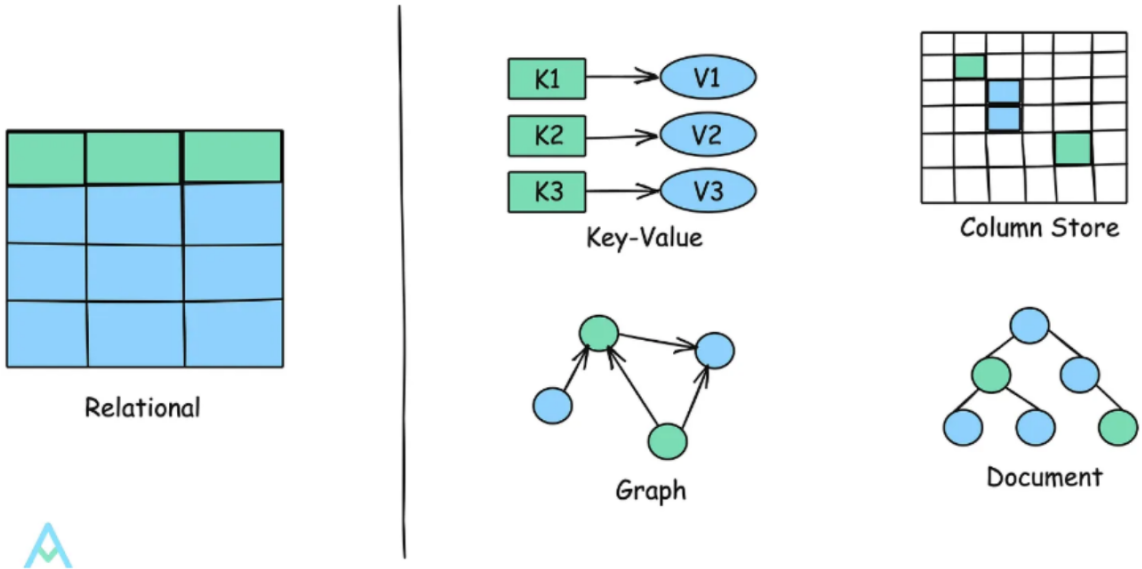
Throughput refers to the average volume of data that can actually pass through the network over a specific time.



Low-latency systems are essential for real-time applications, whereas high-throughput systems are crucial for data-intensive applications.

> **Example:** Online gaming requires low latency to provide real-time interactions among players.
>
> A data analytics service prioritizes throughput to process and analyze large datasets over time.

4. SQL vs NoSQL Databases



SQL (**relational databases**) are built on the relational model that organizes data into tables of rows and columns, with a unique key identifying each row.

These databases are highly structured and offer powerful query languages, making them ideal for complex queries and transactions. Examples: MySQL, PostgreSQL.

However, SQL databases can be challenging to scale horizontally.

> **Example:** Banks use SQL databases for transaction management. These databases ensure that all transactions are processed reliably, maintaining accurate account balances and transaction histories.

NoSQL (**non-relational databases**) offer flexibility and scale easily but might sacrifice SQL like query capabilities and ACID transactions.

5. Consistency vs. Availability (CAP Theorem)

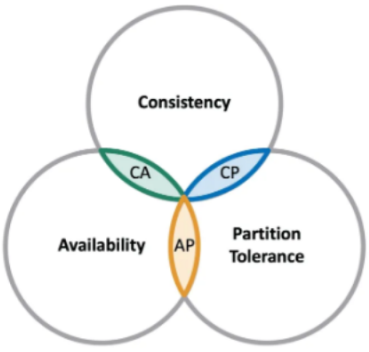
Consistency means that every time someone accesses the system, they get the most recent data.

> **Example:** In an e-commerce platform like Amazon, when a customer places an order, the system ensures that stock levels are immediately updated. This consistency prevents other customers from ordering the same item if it's unavailable.

Availability is about ensuring the system is always up and running, even if some parts of it are having problems.

> **Example:** In an online messaging service, availability ensures you can still send and receive messages even when some servers are down.

According to CAP Theorem, in a distributed system, you can only guarantee two out of the three: Consistency, Availability, and Partition Tolerance.



Credit: https://hazelcast.com/glossary/cap-theorem/

Choosing between consistency and availability depends on what's more important for the user experience you're aiming to provide.

6. Strong vs Eventual Consistency

In the world of distributed systems, where data is stored across multiple locations, ensuring that everyone sees the same data at the same time can be challenging.

This is where the concepts of strong and eventual consistency come into play.

Strong consistency means that as soon as a data update occurs, any subsequent access to that data will reflect the update.

> **Example:** In a banking system, when you transfer money from one account to another, the system updates the balances immediately.

Eventual consistency, on the other hand, means there might be a delay before an update is visible across all nodes in a system. But, it's guaranteed that if no new updates are made, eventually, all accesses to that data will return the updated value.

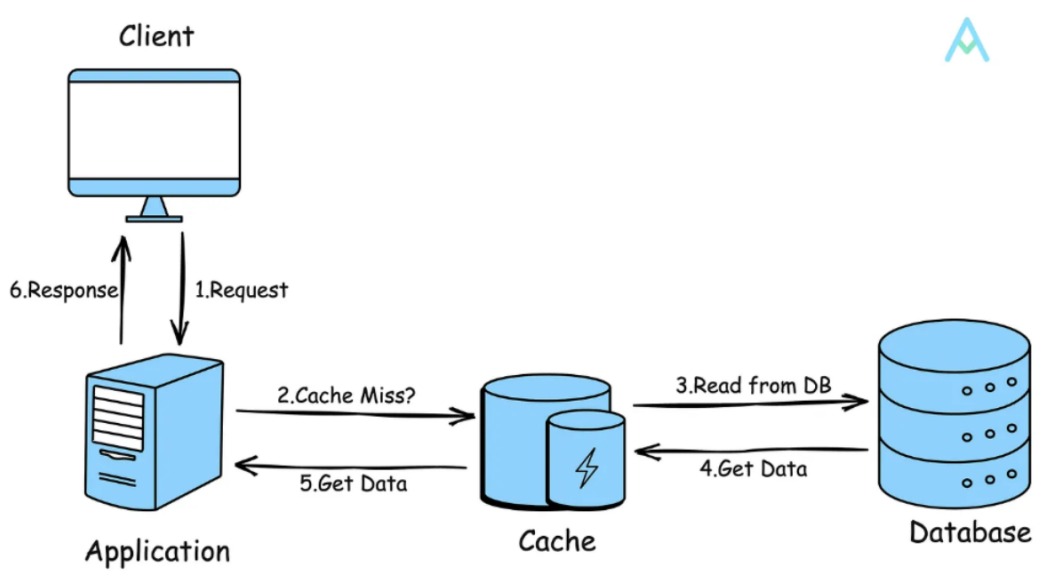
> **Example:** On a social media platforms like Instagram, when you post a new photo, it might not immediately appear on all your followers' feeds. However, after a short period, everyone will be able to see the latest updates.

7. Read-Through vs Write-Through Cache

Caching is a technique used to speed up access to data by storing a copy of frequently accessed data in a faster storage medium.

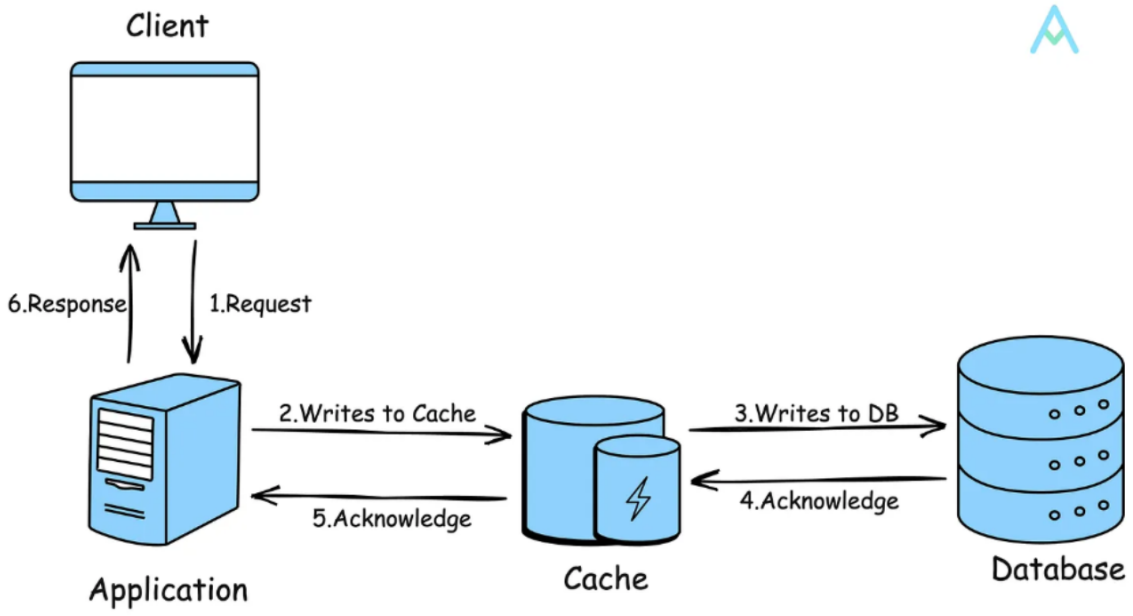
When it comes to cache strategies, "Read-Through" and "Write-Through" are two common approaches.

A **read-through** cache checks the cache first when data is requested. If the data isn't there (cache miss), it's loaded from the slower primary storage into the cache before being returned.



It is useful for *read-heavy* applications where data is frequently accessed but not often updated, reducing the load on the primary storage and improving read performance.

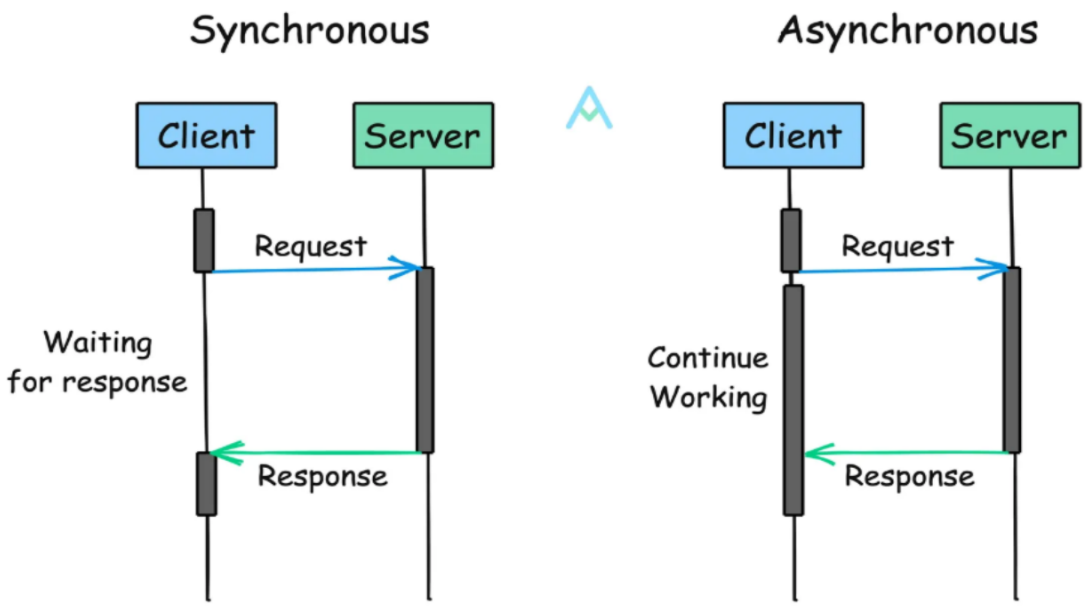
A **write-through** cache simultaneously writes data updates to the cache and primary storage, ensuring up-to-date data and reducing data loss risks.



It's beneficial for *write-heavy* applications. All writes are instantly reflected in both the cache and primary storage.

> **Example**: A movie ticket booking system uses a write-through cache to prevent overbooking by immediately recording bookings both in the cache and the database.

9. Synchronous vs. Asynchronous Processing



Synchronous processing means tasks are performed one after another. A task must be completed before the next one starts, and the system waits for the outcome before proceeding.

> **Example**: When you make an online purchase, the payment process is synchronous. After you click "Pay Now," you wait for the transaction to process. The website does not let you move forward until it confirms that your payment was successful.

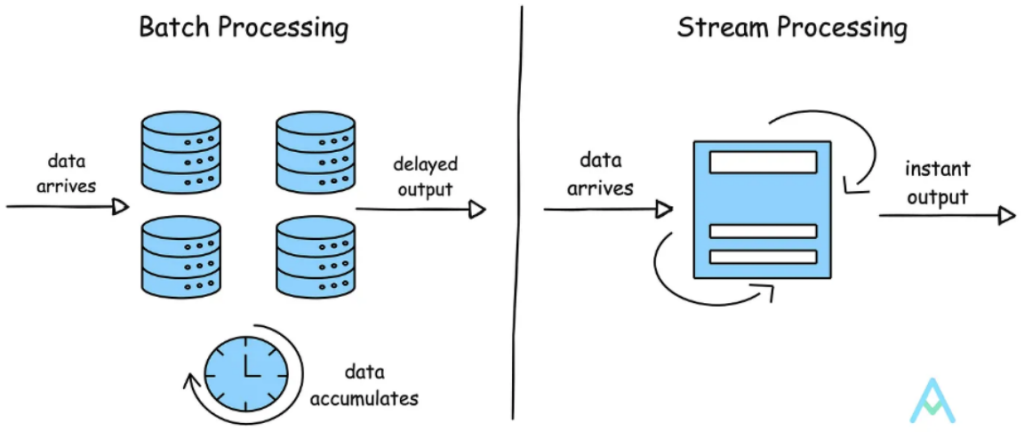
Asynchronous processing allows tasks to run in the background and doesn't need to wait for it to complete before starting another one.

> **Example**: Uploading photos on social media happens asynchronously in the background. You can keep scrolling or exit the app while the photo uploads.

8. Batch vs Stream Processing

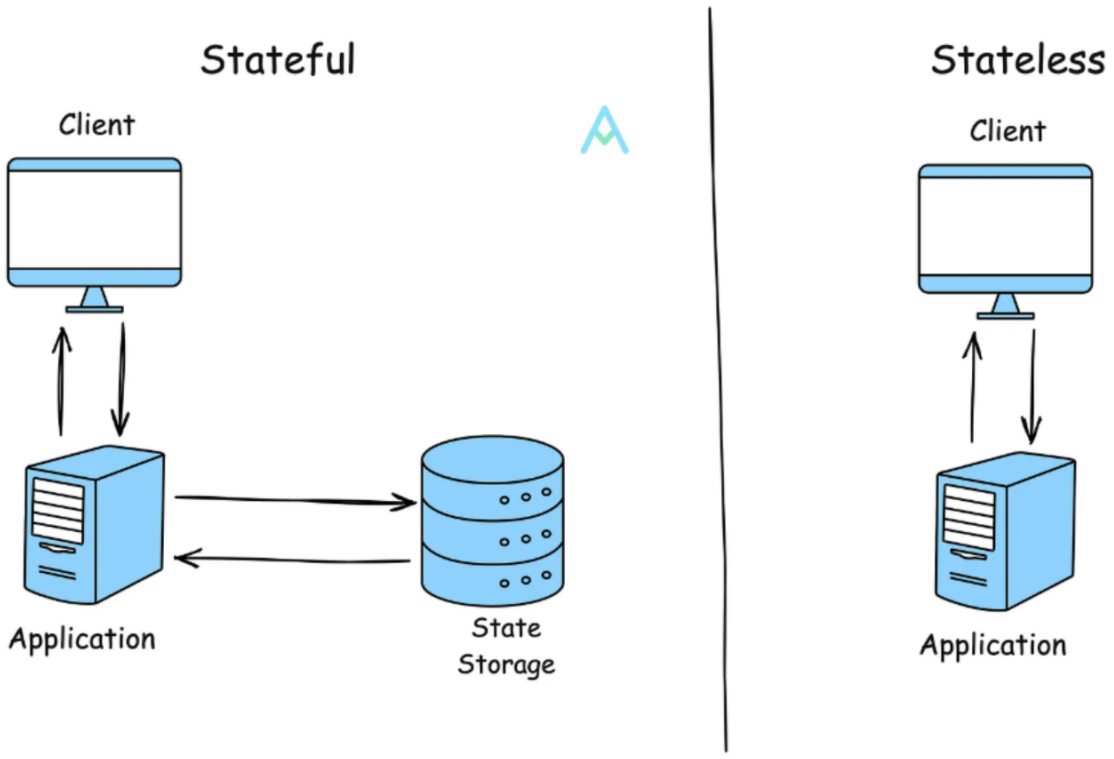
Batch processing involves collecting data over a period of time and then processing it all at once.

Stream processing, on the other hand, deals with data in real-time, processing it as soon as it arrives.



> **Example**: Credit card companies use batch processing for daily billing and statement generation.
>
> For fraud detection, they implement stream processing to analyze transactions in real-time and flag suspicious activities immediately.

10. Stateful vs Stateless Architecture



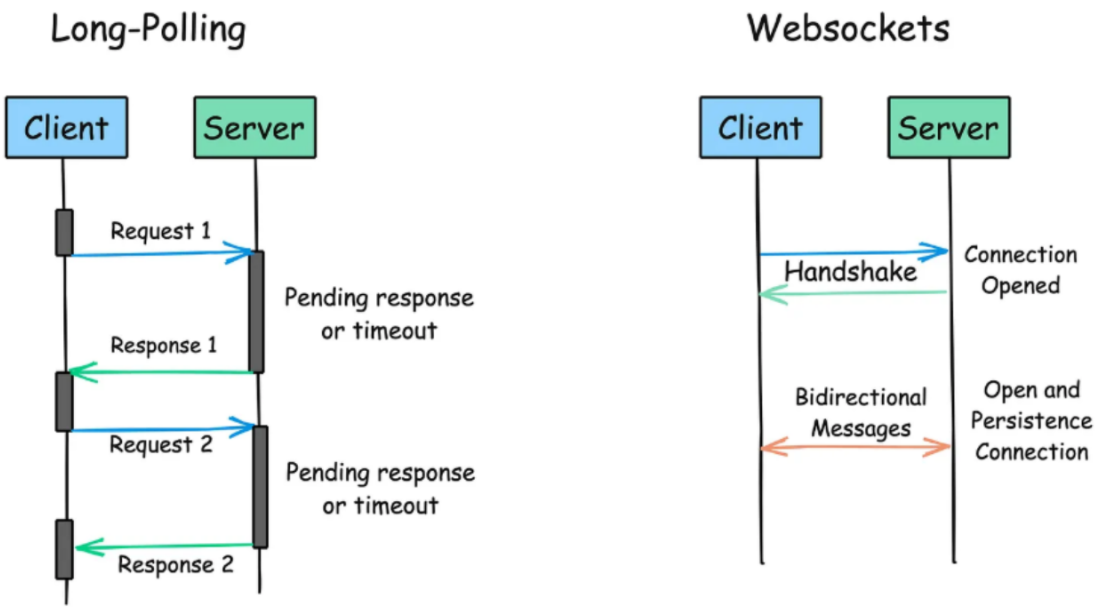
A **stateful system** remembers past interactions. It stores information about the current session, making it possible to have continuity and context in subsequent interactions without needing to start from scratch each time.

> **Example:** During online shopping, when you add items to your cart, the website remembers your selections. If you navigate away to browse more items and then return to your cart, your items are still there, waiting for you to check out.

A **stateless system** does not keep track of past interactions. Each request is treated as new, with no information retained from previous requests.

> **Example**: Many RESTful web services, operate without remembering past requests. For instance, when you make a query to a public API for weather information, you provide all necessary details (like location) with each request.

11. Long Polling vs WebSockets



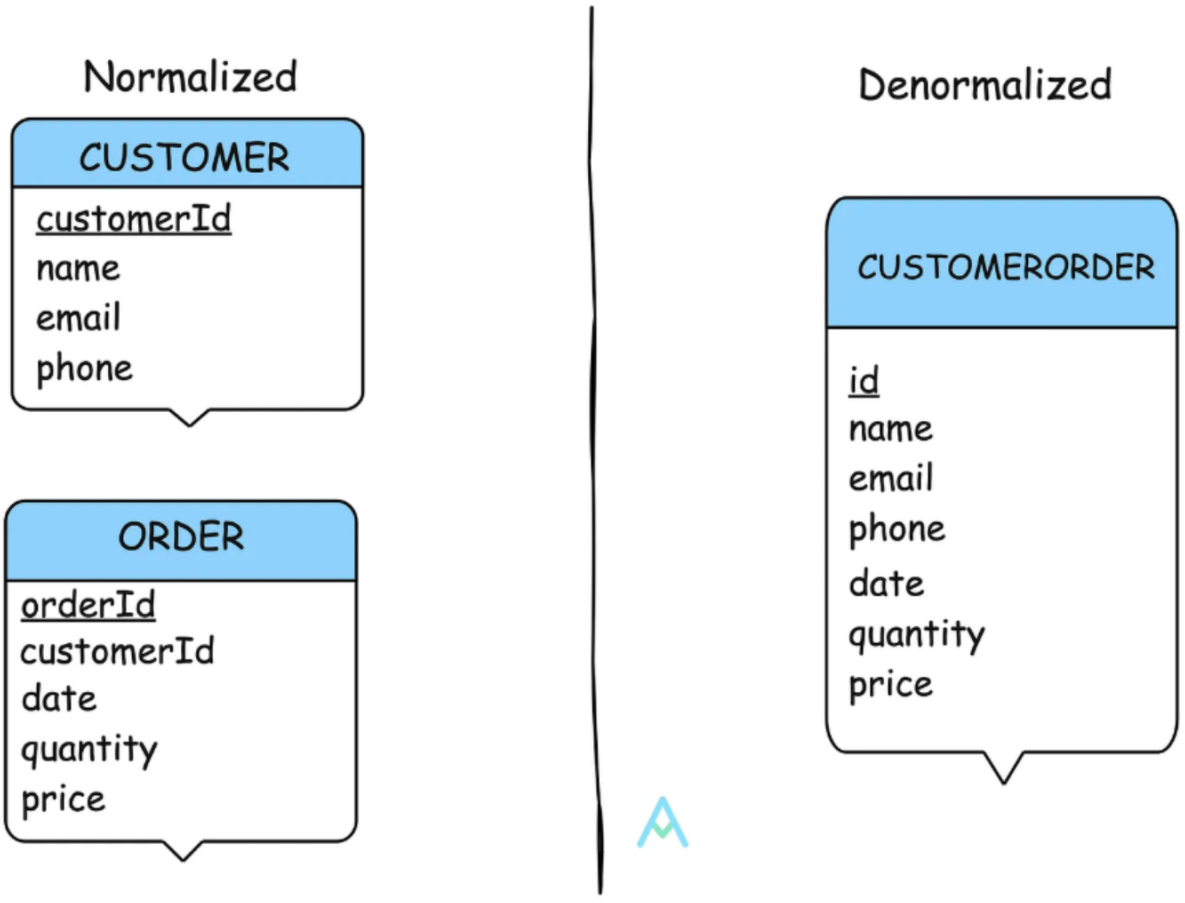
Long polling is a technique where the client requests data from the server and the server holds the request open until new information is available. After receiving the data, the client instantly sends a new request, enabling immediate updates.

> **Example:** A social media platform's notification system. The browser continuously queries the server for new notifications, and the server responds when a new notification appears or a timeout occurs.

WebSockets provide a full-duplex communication channel over a single, long-lived connection, allowing the server and client to send data back and forth as soon as it's available, without waiting for a request from the other side.

> **Example**: In a multiplayer online game, WebSockets enable real-time sharing of player actions and game updates, thanks to the persistent connection between client and server.

12. Normalization vs. Denormalization



Normalization in database design involves splitting up data into related tables to ensure each piece of information is stored only once.

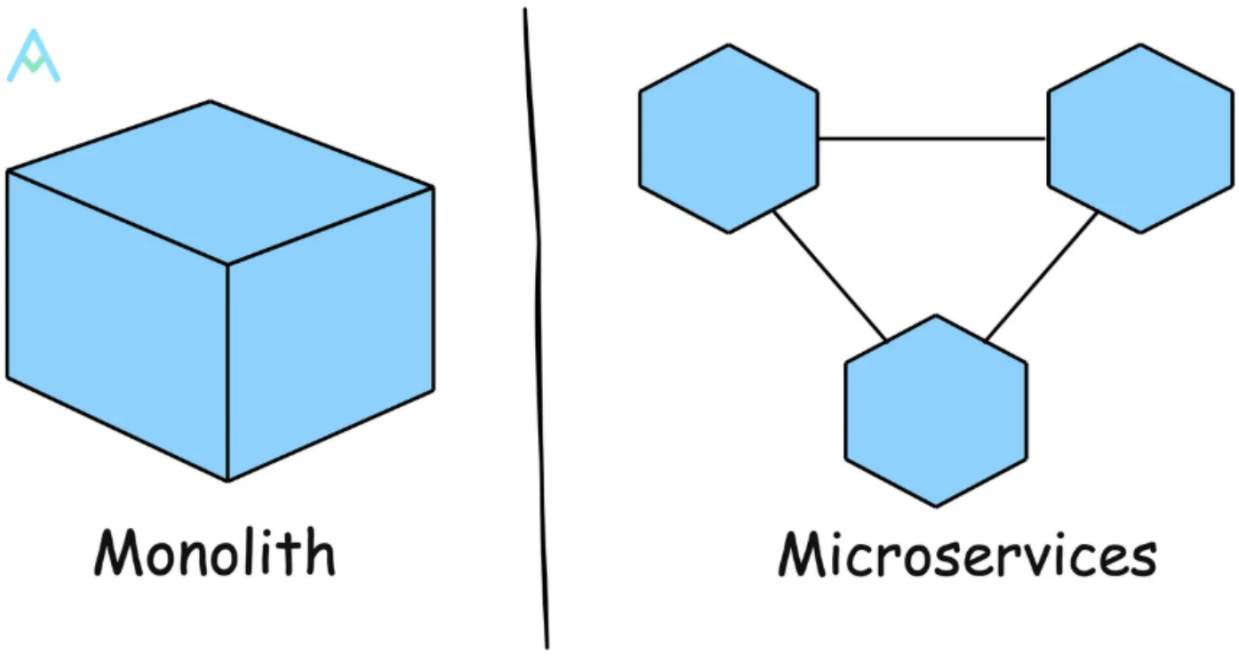
It aims to reduce redundancy and improve data integrity.

> **Example**: A customer database can have two separate tables: one for customer details and another for orders, avoiding duplication of customer information for each order.

Denormalization, on the other hand, is the process of combining data back into fewer tables to improve query performance. This often means introducing redundancy (duplicate information) back into your database.

13. Monolithic vs. Microservices Architecture

A **monolithic** architecture is built as a single unified unit where all the parts of an application are bundled together while a **microservices** architecture is a collection of smaller, independently deployable services.



Monolithic Architecture offers simplicity and ease of deployment, making it suitable for smaller applications or teams. However, it may slow down development and complicate scalability as the application grows.

Microservices Architecture improves scalability and development velocity. But, it introduces complexity in service management, data consistency, and can increase communication overhead.

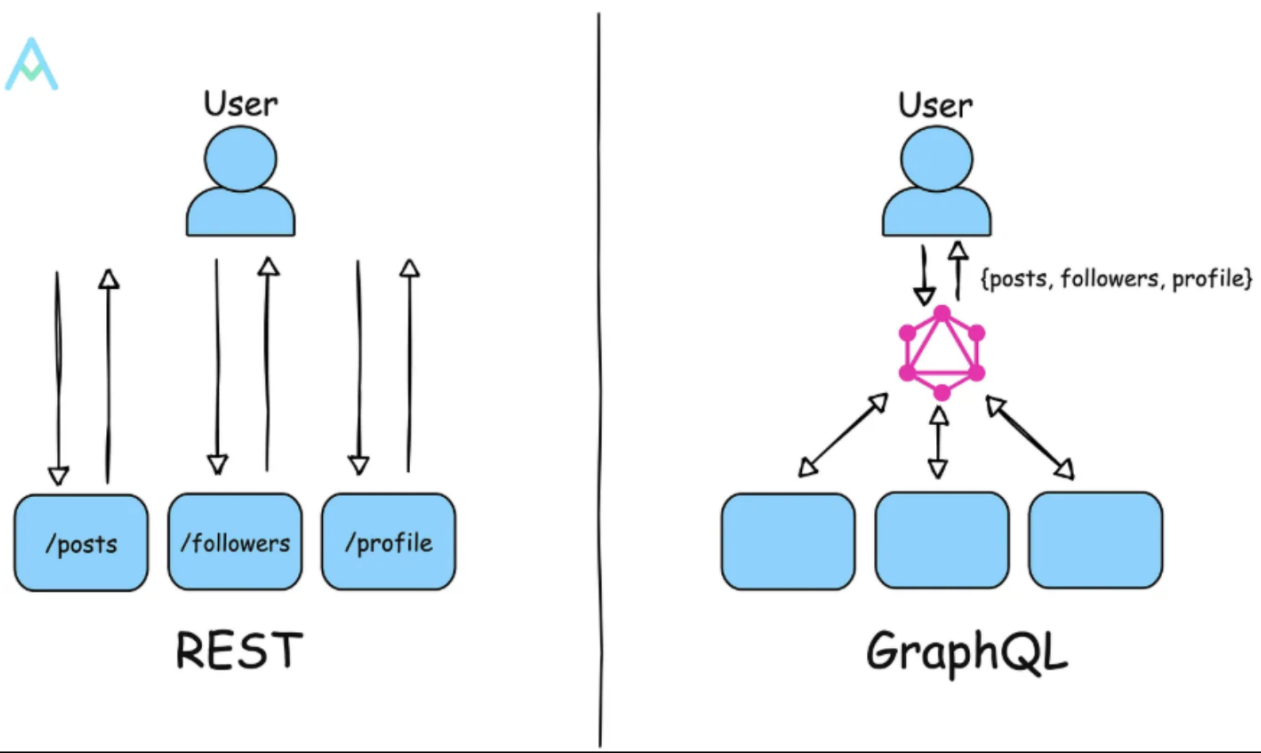
> **Example:** A small web application might start as a monolith for simplicity. As it grows, it could evolve into a microservices architecture, splitting into smaller, independently scalable services for better agility and scalability.

14. REST vs. GraphQL

REST is a well-established standard for APIs, offering simplicity and support for multiple formats. With a REST API, you gather the data by accessing multiple endpoints.

GraphQL provides more efficient data fetching with fewer requests but requires a steeper learning curve and more upfront design.

In GraphQL, you send a single query to the GraphQL server that includes the concrete data requirements. The server then responds with a JSON object where these requirements are fulfilled.



15. TCP vs. UDP

TCP and UDP are communications standard for delivering data and messages through networks.

TCP (**Transmission Control Protocol**) ensures that your message arrives intact and in the exact order you sent it. It establishes a connection between sender and receiver, checks that data is received correctly, and resends lost data.

TCP is ideal for applications like an email service where reliability is crucial.

UDP (**User Datagram Protocol**) sacrifices reliability for speed, suitable for time-sensitive applications like video streaming where it's okay if some data gets lost in transmission.

UDP sends data without establishing a connection, and there's no check to ensure the data is received or in the right sequence.

Online gaming and live streaming services might choose UDP for its lower latency, sacrificing reliability for speed.