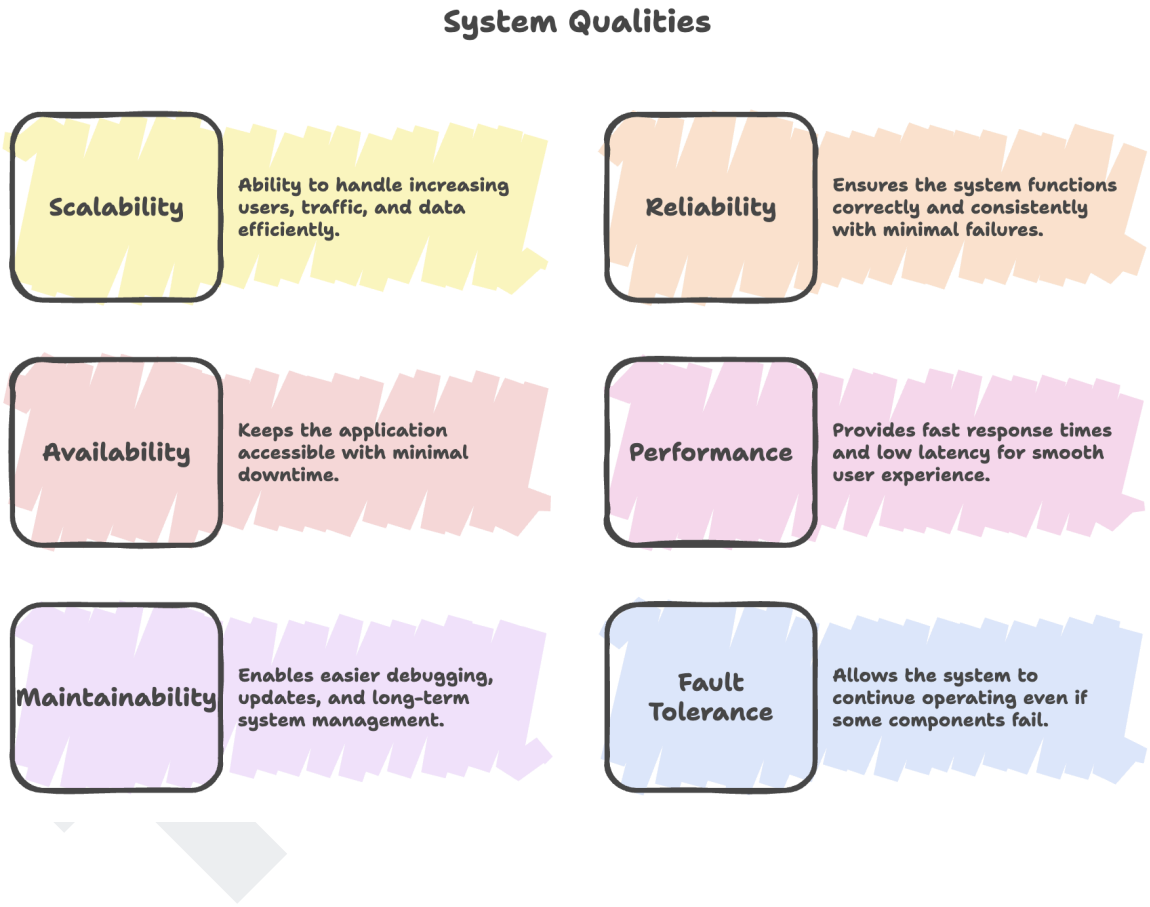
Introduction to System Design

System Design is the process of defining the architecture, components, and interactions of a system to satisfy specified requirements.

Core Concept

System = Components + Common Goal

All components within a system work together to achieve a common objective such as processing requests, managing data, or delivering services reliably at scale.



Understanding System Design Through a Banking Analogy

Consider a bank called AlienBank with a simple cash counter system.

Components in the System

- Customer

- Cash Counter
- Cashier
- Receipt System
- Database

Customer Flow Cycle

- Customer comes to the counter
- Customer requests withdrawal or deposit
- Cashier processes the request
- Receipt is generated
- Transaction is completed

Problems and Solutions Mapped to System Design Concepts

Issue 1: Slow Processing

Problem: Each customer takes around 10 minutes due to manual work.

Solution(s):

- Train the cashier to count cash faster
- Improve typing speed and workflow

System Design Mapping:

➢ Faster cashier → Improved code quality

➢ Optimization → Low-Level Design (LLD)

Issue 2: Increasing Customers

Problem: A single counter cannot handle growing demand.

Solution(s):

- Increase counter capacity

- Use better tools, such as cash counting machines
- Reduce processing time

System Design Mapping:

➢ Better tools → Vertical Scaling

➢ Upgrading resources → Adding CPU, RAM, or storage to a single server

Issue 3: High Waiting Time

Problem: A single counter becomes a bottleneck during peak hours.

Solution(s):

- Add more counters
- Add more cashiers

System Design Mapping:

➢ Multiple counters → Horizontal Scaling

➢ More cashiers → More server instances

Issue 4: Data Inconsistency

Problem: Different counters maintain separate records, causing discrepancies.

Solution(s):

- Use a centralized database as a single source of truth

System Design Mapping:

➢ Shared records → Centralized Database

➢ Data consistency → Database synchronization

Issue 5: Underutilized Usage

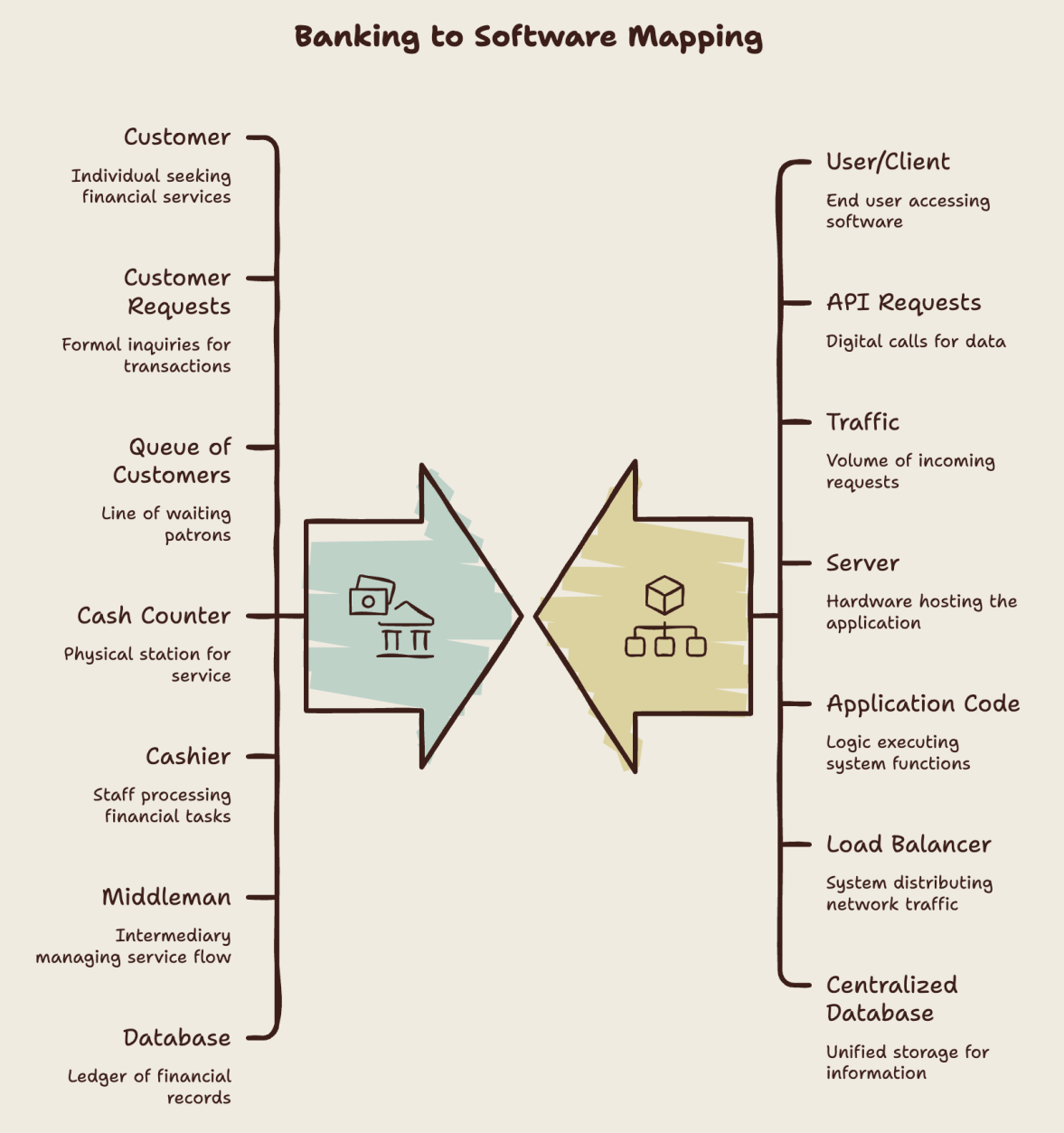
Problem: Some counters are overloaded while others remain idle.

Solution(s):

- Introduce a middleman to distribute customers evenly

System Design Mapping:

- ➢ Middleman → Load Balancer
- ➢ Customer distribution → Request distribution



Key Takeaways

- Optimizing internal processing improves system performance (LLD).
- Vertical scaling increases the capacity of a single server.

- Horizontal scaling increases capacity by adding more servers.
- Centralized databases help maintain data consistency.
- Load balancers distribute traffic efficiently across servers.
- A good system should be scalable, reliable, maintainable, efficient, and fault-tolerant.

Summary

System design focuses on creating scalable, reliable, and efficient systems. As demand grows, systems must handle more users, larger workloads, and increased complexity. Concepts such as low-level design, vertical scaling, horizontal scaling, centralized databases, and load balancing form the foundation of modern distributed systems.

System Design Components

At the heart of every software system lies data. Applications such as WhatsApp, YouTube, LinkedIn, and Instagram store, process, and deliver data in different formats including text, images, audio, and video. The primary objective of system design is to ensure that this data is stored efficiently, processed correctly, and delivered quickly to users.

Core Idea of System Design

A software system consists of multiple interconnected components. Each component performs a specific responsibility such as storing data, processing requests, handling traffic, improving performance, managing communication, or monitoring system health.

1. Database

- The database is the persistent storage layer of a system.
- It stores and retrieves application data reliably.
- Two major categories are **SQL (Relational) and NoSQL** (Non-Relational).
- Examples:
- SQL (Relational) - MySQL, PostgreSQL
- NoSQL (Non-Relational) - MongoDB, Cassandra
- The choice depends on scalability, structure, and consistency requirements.

2. Application Code (Backend / Server)

- Acts as the **middle layer between users and the database**.
- Exposes **APIs that allow users to interact with the system.**
- Handles Insert, Retrieve, Update, and Delete (CRUD) operations.
- Communicates with databases using network protocols.
- Hides database complexity from end users.

How does it connect to the database?

➢ Through **IP addresses**

- Using protocols like **TCP** or **HTTP**

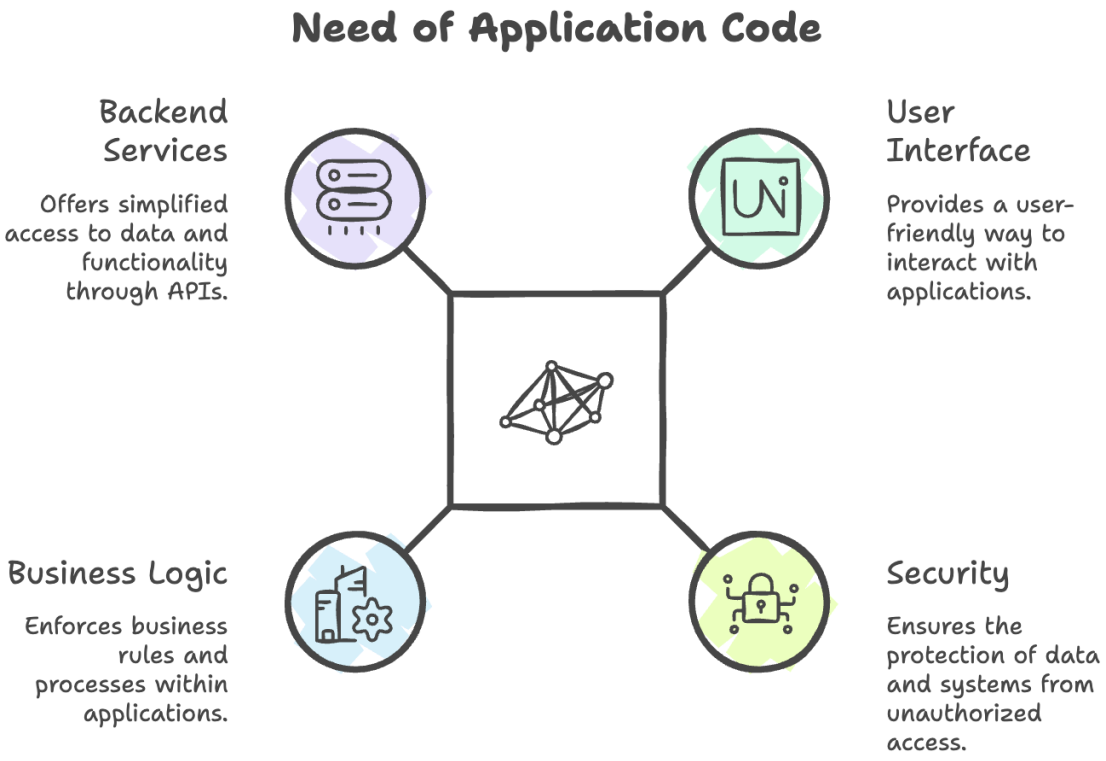
APIs (Application Programming Interfaces):

APIs are endpoints through which clients communicate with the backend application.

Operation	Example
Get user data	/getUser
Upload video	/uploadVideo
Create order	/createOrder
Delete post	/deletePost

Data Format - JSON:

- APIs send and receive data in JSON (JavaScript Object Notation) format.
- JSON provides a standardized, lightweight structure for data exchange between client and server.



3. Client Application (Frontend)

- The user-facing interface of the system.
- Examples include mobile apps, web applications, and desktop software.
- Displays data, collects user input, and sends requests to backend APIs.
- Communicates with servers through HTTP, HTTPS, or TCP/IP.

Responsibilities of Client Applications:

➢ Display data to users

➢ Take user input

- ➢ Send requests to backend APIs
- ➢ Show responses from servers

Communication with Backend:

Client applications communicate with backend systems using:

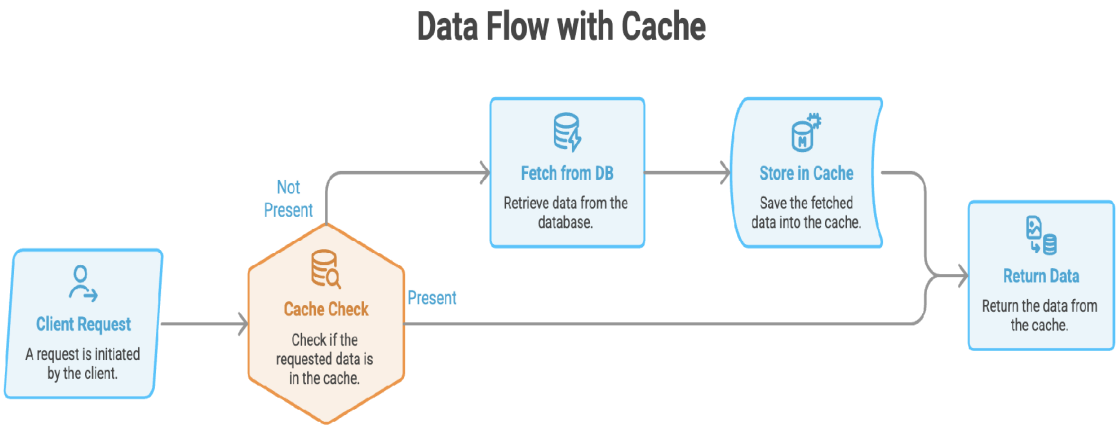
- ➢ HTTP
- ➢ HTTPS
- ➢ TCP/IP

4. Cache

- A cache **stores frequently accessed data in a fast-access layer**.
- Reduces the number of database queries.
- Improves response time and overall performance.
- Commonly used for frequently requested information.

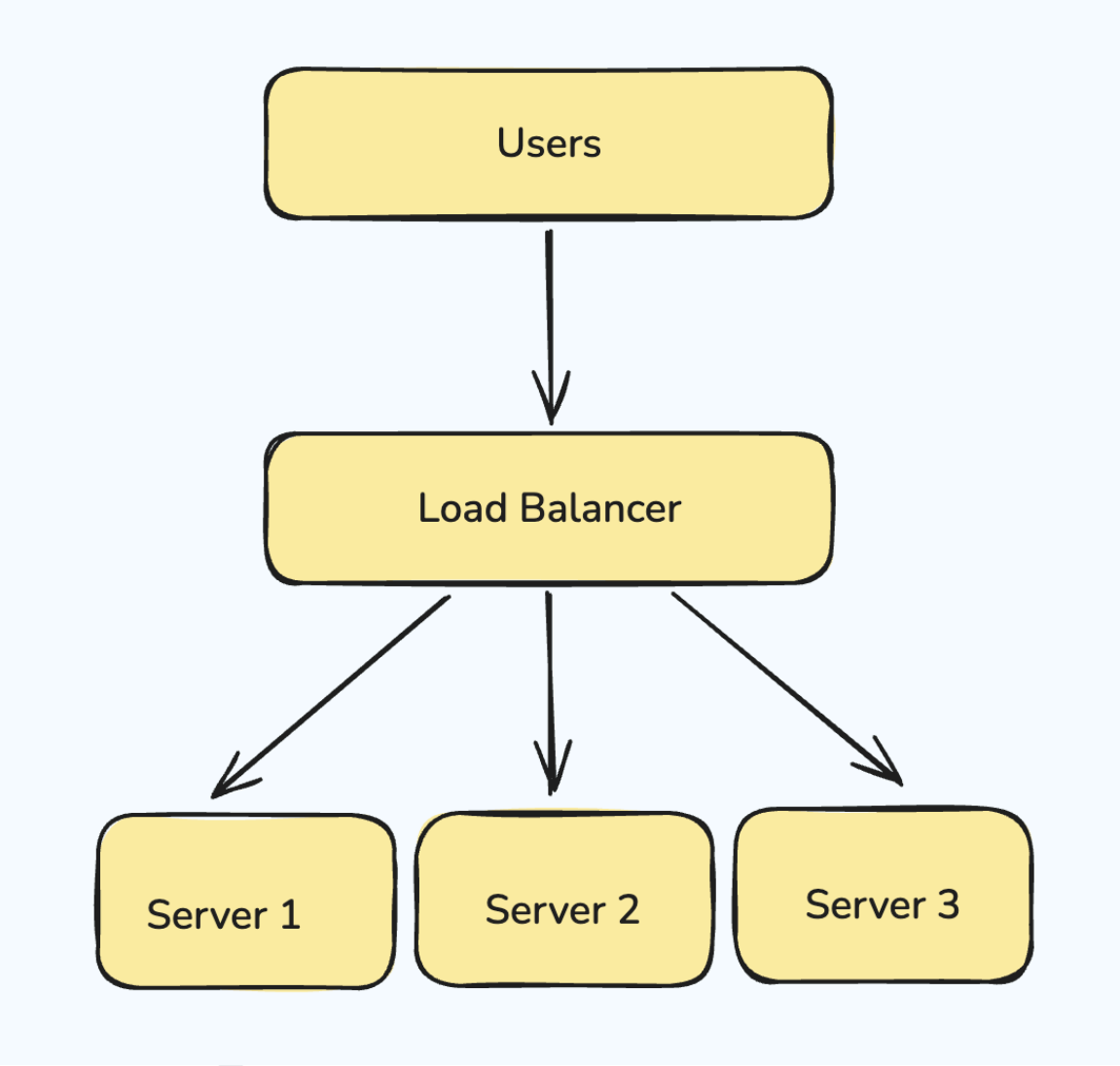
Example:

If thousands of users request the same homepage feed, the cache serves it directly without hitting the database every time.



5. Load Balancer

- **Distributes incoming requests across multiple servers.**
- Prevents a single server from becoming overloaded.
- Performs health checks to ensure servers are available.
- Improves scalability, reliability, and resource utilization.



6. Message Queue

- Enables **asynchronous communication** between services.
- Stores requests and processes them independently.
- Improves system responsiveness and reliability.

- Handles retries when processing failures occur.
- Examples include sending email or SMS notifications after an order is placed.

Real-World Example - Order Service:

When an order is placed, multiple tasks follow:

- ➢ Inventory must be updated
- ➢ Delivery service must be notified
- ➢ Email/SMS confirmation must be sent

If all tasks happen immediately in sequence, the response becomes slow.

Synchronous vs Asynchronous Communication:

- **Synchronous communication** requires the caller to wait for a response before proceeding.

  Example: Payment confirmation before order completion

- **Asynchronous communication** allows the caller to continue processing without waiting for a response.

  Example: Sending an SMS notification after order placement

How Message Queues Work?

- ➢ The order service pushes requests into the message queue
- ➢ The message queue stores and manages these requests independently
- ➢ Connected services (SMS, Email) pick up and process requests one by one
- ➢ The queue handles **retries** if a request fails
- ➢ It may send a **batch summary** (fulfilled/unfulfilled requests) back to the order service at the end of the day.

In this way, the order service is no longer responsible for tracking individual notifications, the message queue takes full ownership.

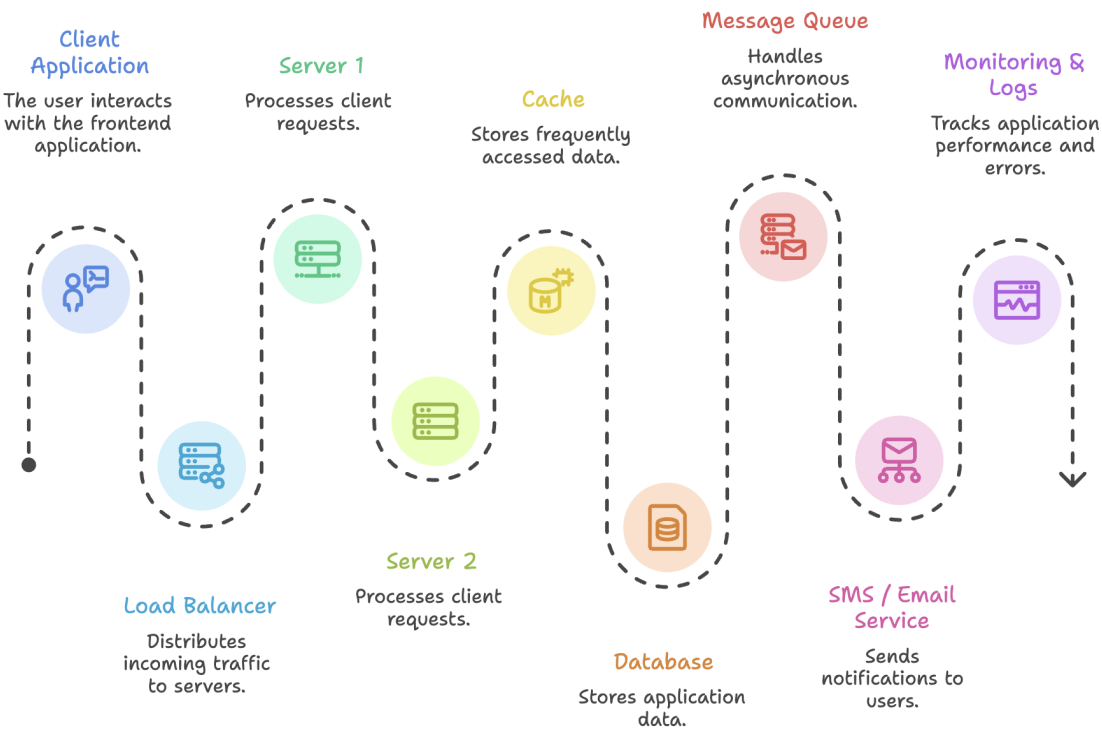
7. Monitoring and Logs

- Monitoring tracks the health and performance of system components.
- Logs record events and errors for debugging and analysis.

- Help identify issues quickly and maintain system stability.
- Support performance tracking, alerting, and troubleshooting.

Why is Monitoring Critical?

- ➢ Verify that the load balancer is distributing traffic properly
- ➢ Detect errors in application code
- ➢ Access the complete **stack trace** of errors for debugging
- ➢ Reproduce issues in local environments for fixing
- ➢ Ensure new feature deployments don't break existing functionality



Summary

- Modern systems focus on efficient data storage, processing, and delivery.

- Application code and APIs simplify communication between users and databases.
- Caching improves performance by reducing repeated database access.
- Load balancers distribute traffic efficiently across servers.
- Message queues support asynchronous communication and fault tolerance.
- Monitoring and logging are essential for maintaining system reliability and performance.

Data-Intensive and Compute-Intensive Applications

What is Latency?

Latency is the **time taken by a system to respond to a request**. It represents response delay, waiting time, or processing delay.

Examples:

- Opening an Instagram feed
- Playing a YouTube video
- Sending a WhatsApp message

The time between the action and the response is called latency.

Data-Intensive Applications

Data-intensive applications primarily deal with storing, retrieving, transferring, updating, and serving large volumes of data efficiently.

Examples:

- **Instagram Feed** - Millions of posts served to millions of users simultaneously
- **WhatsApp Messaging** - Constant read/write of messages across devices
- **Banking Systems** - High volume of transactional reads and writes
- **Analytics Dashboards** - Aggregating and displaying large datasets in real time
- **Log Processing Systems** - Ingesting, storing, and querying massive log files

Key Challenges:

- ➢ **Slow database response:** Queries take too long to return results
- ➢ **Large network transfers:** Large amounts of data travel across networks that need optimization

- **Insufficient server resources:** Server hardware not capable of serving data at required speed

Key Concerns:

- How fast can data be read?
- How safely can data be stored?
- How many users can access it simultaneously?
- What happens if a server fails?
- What happens if network communication breaks?

Common Solutions:

- **Caching:** Store frequently accessed data closer to the user for faster reads
- **Sharding:** Split data across multiple databases to distribute load
- **Replication:** Maintain copies of data across servers for fault tolerance
- **Consistency Models:** Ensure data accuracy across distributed systems
- **Database Optimization:** Indexing, query tuning, and choosing the right database type

Compute-Intensive Applications

Compute-intensive applications are limited mainly by CPU or GPU processing power rather than data storage or transfer.

Examples:

- **Image Processing:** Pixel-level transformations, filters, resizing
- **Video Rendering:** Frame-by-frame encoding and decoding
- **Machine Learning Training:** Iterative mathematical computations over large parameter spaces
- **Simulations:** Physics, weather, or financial modeling
  Cryptography: Complex encryption/decryption algorithms |

Key Concerns:

- How fast can computations be performed?
- Can the workload be parallelized?
- Can algorithm complexity be reduced?
- Can GPUs be used for acceleration?

Common Solutions:

- **Parallel Processing:** Divide tasks into smaller parts and execute them simultaneously to improve performance.
- **GPU Acceleration:** Performs large-scale computations much faster than CPUs
- **Distributed Computing:** Spread workloads across multiple machines to increase processing capacity and scalability
  Algorithm Optimization: Improve algorithms to reduce execution time and resource consumption

Data-Intensive vs Compute-Intensive

Aspect	Data-Intensive	Compute-Intensive
Bottleneck	Data movement & storage	Processing power
Focus	Fast data retrieval & delivery	Fast computation
Hardware Concern	Disk I/O, Network, Memory	CPU/GPU
Data Volume	Very high	Relatively low
Processing Complexity	Low to moderate	Very high
Solutions	Caching, Sharding & Replication	Parallelism, GPU, Acceleration, Algorithm optimization

Modern Applications Use Both Types

- **YouTube:**
  Data-Intensive → Video storage, streaming, retrieval, and delivery.
  Compute-Intensive → Recommendations, analytics, and personalization.

- **Instagram:**
  Data-Intensive → Feed loading, messaging, media delivery.
  Compute-Intensive → Image filters, recommendations, face detection.

Rule of Thumb

- If performance issues are caused by data movement, storage, or network communication, the application is Data-Intensive.

  If time is lost in **data movement** → Data-Intensive Application

- If performance issues are caused by heavy calculations and processing, the application is Compute-Intensive.

  If time is lost in **computation** → Compute-Intensive Application

Summary

- Data-Intensive applications focus on efficient data storage, retrieval, consistency, and transfer.
- Compute-Intensive applications focus on maximizing processing performance.
- Most modern systems contain both types of workloads.
- Identifying the actual bottleneck helps in selecting the correct architecture and scaling strategy.

Functional and Non-Functional Requirements

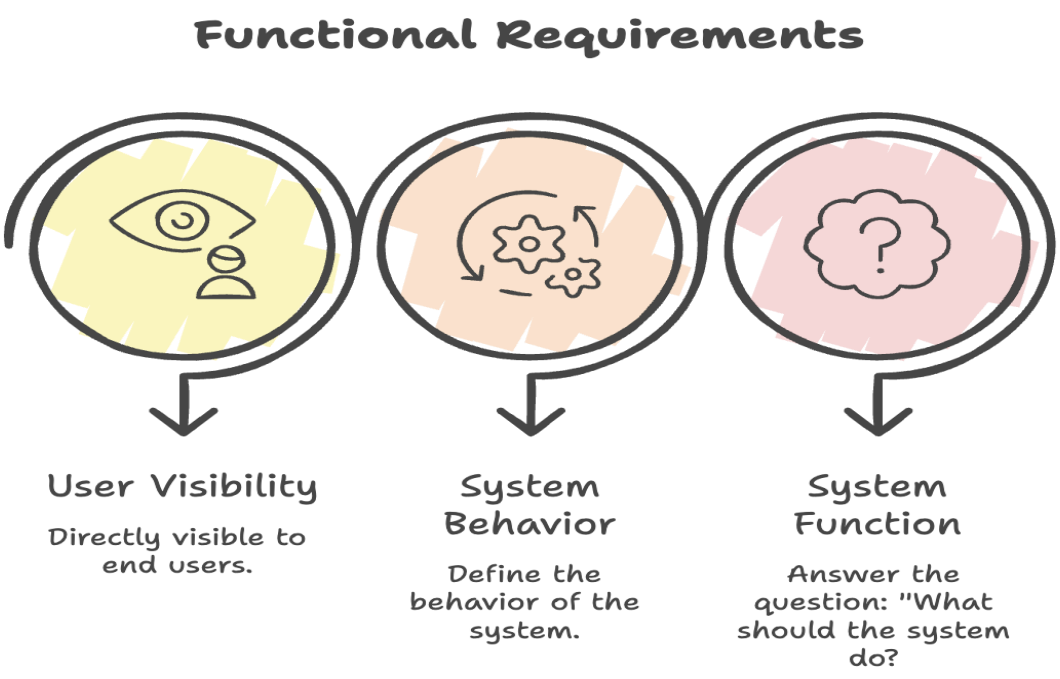
Every software system is built around two important types of requirements:

- **Functional Requirements** – Define WHAT the system should do.
- **Non-Functional Requirements** – Define HOW WELL the system should perform.

Both are essential for building a successful, scalable, and reliable application.

Functional Requirements

Functional requirements describe **the features and capabilities** of a system. They define the actions users can perform and how the system should respond.



E-Commerce Examples for Functional Requirements:

1. User Registration
2. User Login

3. User can browse products
4. User can search products
5. User can filter products
6. User can do add to cart
7. User can apply coupons
8. User can place orders
9. User can make payments
10. User can track orders

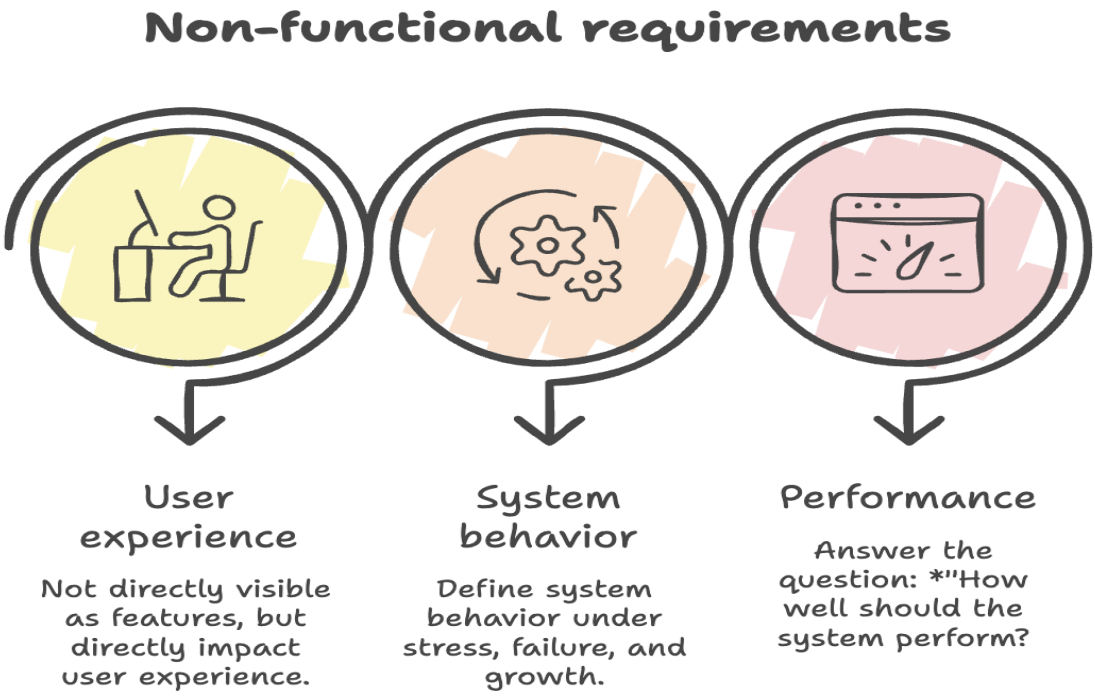
Every feature that a user interacts with in an application is a functional requirement.

Why are functional requirements important?

- Defines what needs to be built.
- Helps identify APIs and database operations.
- Describes user interactions and business logic.

Non-Functional Requirements

Non-functional requirements describe the quality, performance, and constraints of a system.



E-Commerce Examples for Non-Functional Requirements:

1. Handle 1 million concurrent users
2. Response time less than 200ms
3. 99.9% Availability
4. Data Encryption
5. Scale up to 10x traffic
6. Fault tolerance

These requirements ensure the system performs reliably under real-world conditions.

Key Non-Functional Attributes

1. **Scalability:**

   Ability to handle increasing traffic and workload.

   - ➢ Vertical Scaling – Upgrade existing server resources.

- ➢ Horizontal Scaling – Add more servers.

2. **Availability:**

- Measures how often the system remains operational and accessible.
- Measured through **SLA** (Service Level Agreement) and **SLO** (Service Level Objective)
- Expressed as uptime percentage (e.g., 99.9% = ~8.7 hours downtime/year)
  99% Availability → ~3.65 days downtime/year
  99.9% Availability → ~8.76 hours downtime/year
  99.99% Availability → ~52.6 minutes downtime/year
  99.999% Availability → ~5.26 minutes downtime/year

3. **Reliability:**

   Ensures data is not lost and results remain accurate during failures.

4. **Performance (Latency):**

   Measures how quickly the system responds.

   - ➢ P90 – 90% requests complete within this time.
   - ➢ P95 – 95% requests complete within this time.
   - ➢ P99 – 99% requests complete within this time.

5. **Security:**

   Protects users and data.

   - ➢ Authentication – Verify user identity (login credentials)
   - ➢ Authorization – Verify user permissions (role-based access)
   - ➢ Encryption – Protect data in transit and storage.

6. **Maintainability:**

   Ability to modify, update, and extend the system easily.

7. **Observability**:

Ability to monitor and understand the internal state of a system from its external outputs.

- ➢ Monitoring – Track system health.
- ➢ Logging – Record events and errors.

Functional vs Non-Functional Requirements

Functional Requirements	Non-Functional Requirements
Define what the system does	Define how the system performs
Feature-oriented	Quality-oriented
User actions and business logic	Performance and operational behavior
Visible to users	Mostly internal/System-level
Example: Add to card	Example: Response time < 200ms

Summary

- Functional requirements define system features and business operations.
- Non-functional requirements define quality attributes and performance standards.
- A successful system must balance both functionality and performance to deliver a reliable and efficient user experience.
- Non-functional requirements help create scalable and production-ready systems.

DNS (Domain Network System)

Whenever we access a website such as **telusko.com**, our browser needs to know the server's IP address to establish a connection. Since remembering numerical IP addresses is difficult, the internet uses the **Domain Name System (DNS)** to convert human-readable domain names into machine-readable IP addresses.

DNS (Domain Name System) is a distributed naming system that **translates domain names into IP addresses**, allowing users to access websites using easy-to-remember names instead of numerical addresses.

DNS is often referred to as the "Phonebook of the Internet."

Key Terminology

Term	Description
Domain	A website address such as telusko.com
Subdomain	A domain created under a parent domain, such as docs.telusko.com
IP Address	A unique numerical identifier assigned to a device or server on a network

Examples

Domain:

telusko.com

Subdomains:

docs.telusko.com

courses.telusko.com

IP Address:

192.168.1.1

Why Do We Need DNS?

Today, there are hundreds of millions of registered domains on the internet. Without DNS, users would need to remember IP addresses for every website they visit. Maintaining domain-to-IP mappings inside every browser is impractical because:

1. **Storage Limitation**

   Browsers would need massive storage to maintain a mapping of all domains to IP addresses, degrading performance.

2. **Dynamic IP Changes**

   Website IP addresses can change frequently because of:

   - Server migrations
   - Cloud deployments
   - Infrastructure upgrades
   - Load balancing

   Updating every browser whenever an IP change would be impossible.

   Solution

   DNS acts as a centralized lookup system that dynamically resolves domain names to their latest IP addresses whenever required.

How DNS Resolution Works?

When a user enters a URL such as: telusko.com, the browser performs a DNS lookup to find the corresponding IP address.

The DNS resolution process follows a hierarchical structure.

Step 1: DNS Resolver

The browser first contacts a **DNS Resolver**, which is usually provided by the user's Internet Service Provider (ISP).

Responsibilities

- Receives DNS queries from users
- Finds the correct IP address
- Returns the final result to the browser

Step 2: Root Server

- The DNS Resolver first contacts a **Root Server**.
- There are **13 root servers** worldwide, named '*a.root-server.net*' through '*m.root-server.net*'.
- These 13 servers are owned and operated by various organizations.
- The root server does not know the final IP address, it directs the resolver to the appropriate **TLD server.**

Step 3: TLD (Top-Level Domain) Server

TLD servers manage domains based on their extensions.

The TLD server directs the resolver to the **Authoritative Name Server** responsible for the requested domain.

Common TLDs

TLD	Category

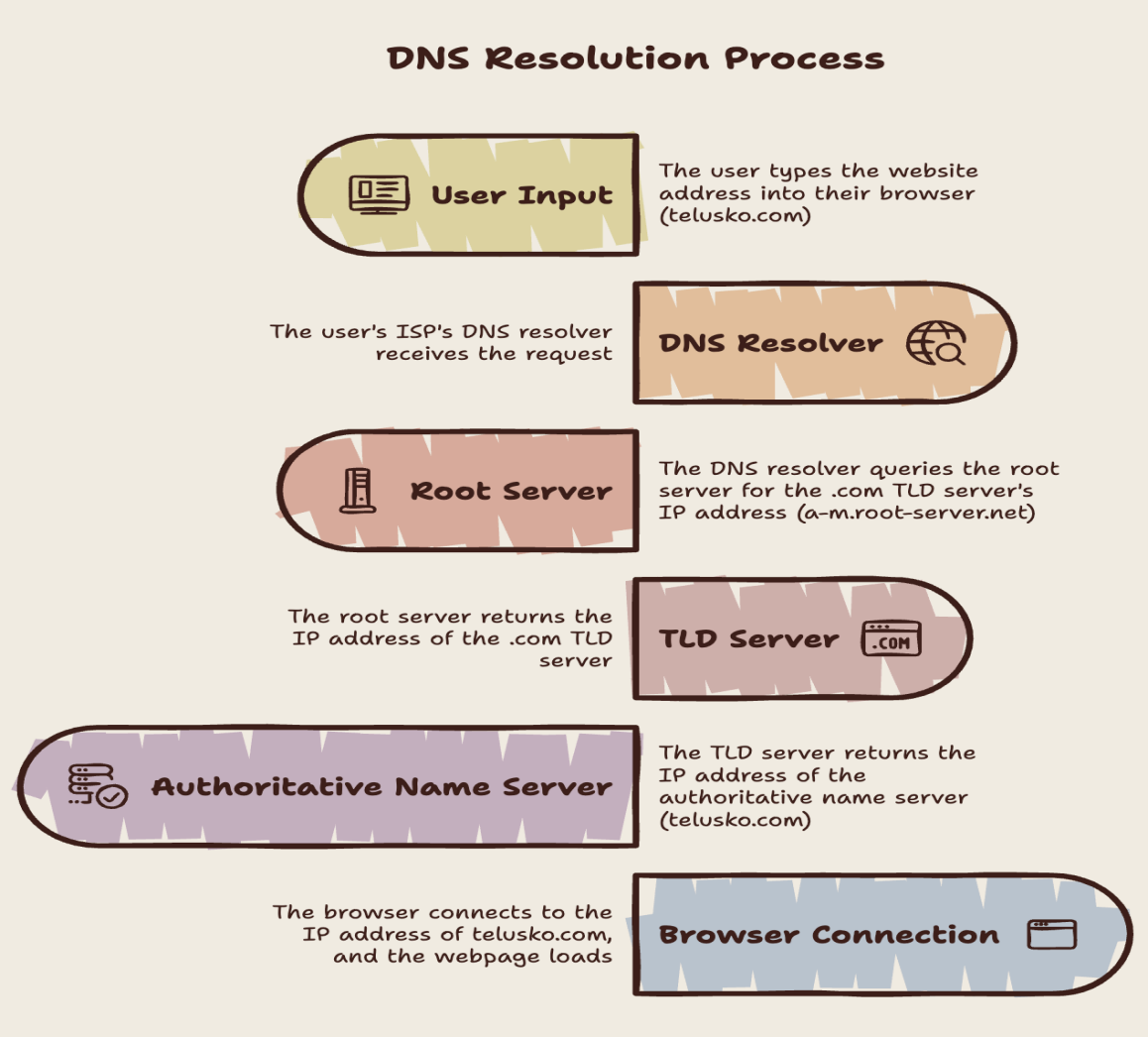
.com	Commercial
.net	Network
.gov	Government
.edu	Education
.in	India (country-specific)

Step 4: Authoritative Name Server

- The Authoritative Name Server contains the **actual DNS records** for a domain.
- It contains zones, each zone manages a specific domain and all its subdomains.
- Returns the final IP address of the requested domain to the DNS Resolver.

Step 5: Browser Connects to the Server

- After receiving the IP address:
- DNS Resolver sends the IP to the browser.
- Browser establishes a connection to that IP.
- The requested webpage is loaded.



DNS Caching

Performing the full DNS lookup process for every request would be slow and inefficient.

To improve performance, DNS responses are cached.

DNS Caching stores previously resolved IP addresses so that future requests can be answered quickly without repeating the entire lookup process.

Levels of DNS Cache

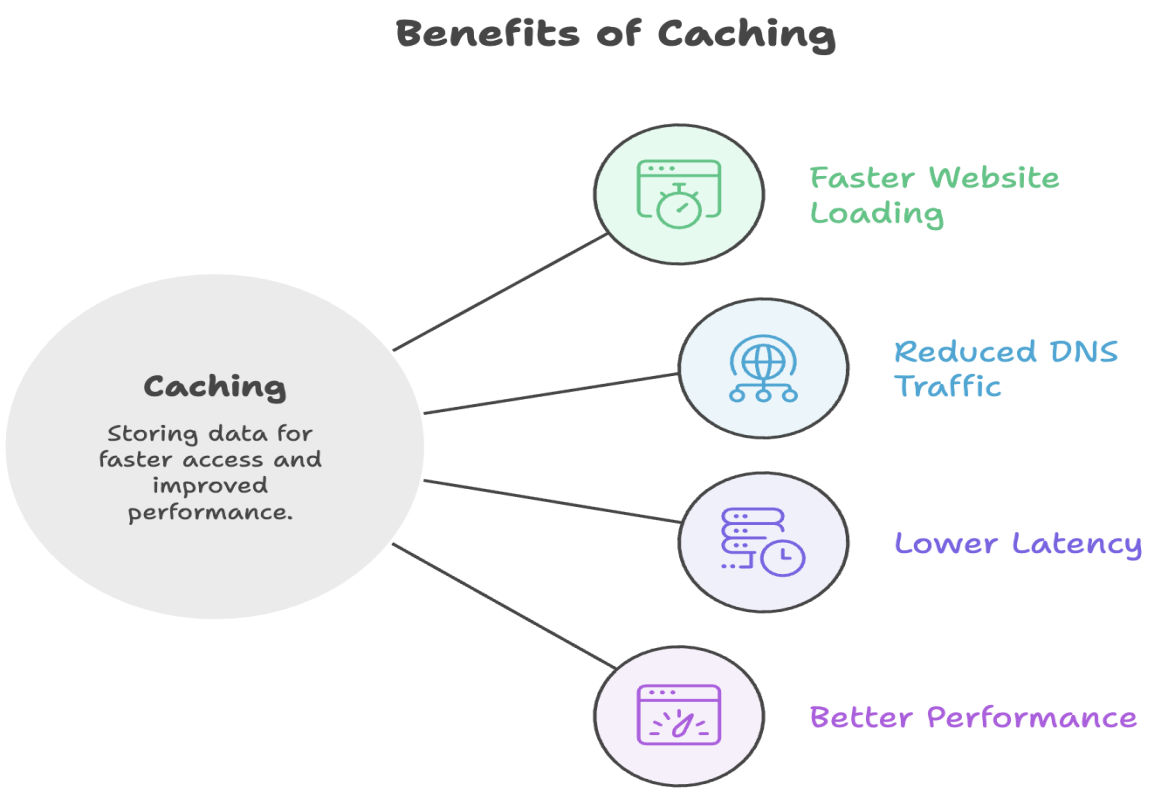
1. Browser Cache: Stores recently visited domains.

Example:

telusko.com → 93.184.216.34

2. Operating System Cache: The operating system maintains its own DNS cache and can provide previously resolved addresses.

3. DNS Resolver Cache: The ISP's DNS resolver stores commonly requested domains for all users.



Zones and Subdomains

An Authoritative Name Server manages one or more **DNS Zones**.

Each zone is dedicated to a specific domain.

A zone contains DNS records for:

- The main domain
- All associated subdomains

Example: Zone for telusko.com

telusko.com → 93.184.216.34

docs.telusko.com → 93.184.216.35

courses.telusko.com → 93.184.216.36

Key DNS Components

Component	Purpose
DNS Resolver	Initiates DNS lookups and returns the final IP address
Root Server	Directs requests to the appropriate TLD server
TLD Server	Manages top-level domains and points to authoritative servers
Authoritative Name Server	Stores actual DNS records and domain mappings
Cache	Stores previously resolved addresses for faster lookups

Summary

- DNS (Domain Name System) translates domain names into IP addresses.
- It allows users to access websites using simple names rather than numerical IP addresses.
- Root servers direct requests to TLD servers. TLD servers identify the correct authoritative server.
- Authoritative servers store actual domain records.
- DNS caching occurs at the browser, operating system, and resolver levels to improve performance.
- DNS zones organize records for domains and their subdomains.
- DNS is a foundational internet service that enables fast, scalable, and reliable website access worldwide.

REST API Requests

REST (**Representational State Transfer**) is one of the most widely used architectural styles for building web services and APIs. It provides a standard way for clients and servers to communicate over HTTP.

REST APIs use **JSON (JavaScript Object Notation)** as the primary format for exchanging data because it is lightweight, easy to read, and supported by almost all programming languages.

REST APIs enable applications to communicate with each other using HTTP methods and resource-based URLs.

Understanding JSON

What is JSON?

JSON (JavaScript Object Notation) is a lightweight data format used to exchange information between clients and servers.

JSON stores data as **key-value pairs** and supports objects, arrays, and multiple data types.

Common JSON Data Types

Data Type	Example
String	"name": "Akshay"
Number	"age": 29
Boolean	"active": true
Null	"address": null
Object	"profile": { "city": "Delhi" }
Array	"skills": ["Java", "Python"]

Example JSON Object:

```
{
  "id": 1,
  "name": "Muskan",
  "username": "muskan",
  "age": 20
}
```

Anatomy of a REST Endpoint

An **endpoint** is a URL through which an API can be accessed.

Every REST endpoint consists of two parts:

Endpoint = HTTP Method + Resource Path

Endpoint Structure:
METHOD /resource

Example:
GET /users

In this example:

- **GET** = Operation to perform
- **/users** = Resource being accessed

Components of a REST API Request

A REST request contains several important components.

1. HTTP Method

The HTTP method defines the action that should be performed on a resource.

Common HTTP Methods

Method	Endpoint	Purpose
GET	/users	Retrieve all users
GET	/users/{id}	Retrieve a specific user
POST	/users	Create a new user
PUT	/users/{id}	Replace an existing user
PATCH	/users/{id}	Partially update a user
DELETE	/users/{id}	Delete a user

CRUD Mapping

Operation	HTTP Method
Create	POST
Read	GET
Update	PUT / PATCH
Delete	DELETE

2. Resource Path (URL)

The path identifies the resource that the client wants to access.

Examples:
/users

/products

/orders

/blogs

Each path represents a collection of resources within the application.

Examples:
GET /users

GET /products

DELETE /orders/101

3. Headers

Headers provide additional information (metadata) about the request.

Example:
Content-Type: application/json

This tells the server that the request body contains JSON data.

Common Headers

Header	Purpose
Content-Type	Specifies request format
Accept	Specifies expected response format
Authorization	Sends authentication credentials
User-Agent	Identifies the client application

4. Request Body

The request body contains the actual data sent to the server.

It is commonly used with:

- POST
- PUT
- PATCH

Example:

```
{
  "name": "Muskan",
  "username": "muskan",
  "age": 20
}
```

The server uses this data to create or update resources.

PUT vs PATCH

One of the most commonly asked REST API interview questions is the difference between PUT and PATCH.

1. PUT – Full Resource Replacement

PUT replaces the entire resource.

If some fields are not included in the request, they may be overwritten with default or null values.

Existing User

```
{
  "id": 1,
  "name": "Muskan",
  "username": "muskan",
  "age": 20
```

}

PUT Request

{

"username": "muskan_new"

}

Result

{

"id": 1,

"name": null,

"username": "muskan_new",

"age": null

}

Since only the username was provided, other fields are replaced.

When to Use PUT?

Use PUT when you want to replace the complete resource with a new version.

2. PATCH – Partial Resource Update

PATCH updates only the specified fields.

All other fields remain unchanged.

PATCH Request

{

"username": "muskan_new"

}

Result

{

"id": 1,

"name": "Muskan",

"username": "muskan_new",

"age": 20

}

Only the username changes.

When to Use PATCH?

Use PATCH when updating specific fields of an existing resource.

Nested APIs

In real-world systems, resources often have relationships with one another.

Nested routes help represent these relationships clearly.

Example: Blog Application

User APIs

Endpoint	Purpose
GET /users	Get all users
POST /users	Create a user
GET /users/{id}	Get specific user
PATCH /users/{id}	Update specific user
DELETE /users/{id}	Delete specific user

Comment APIs

Endpoint	Purpose
GET /blogs/{id}/comments	Get comments for a specific blog
GET /users/{id}/comments	Get comments by a specific user

PATCH /comments/{id}	Update a specific comment
DELETE /comments/{id}	Delete a specific comment

Examples
/blogs/101/comments

/users/20/comments

These URLs clearly indicate the relationship between resources.

Nesting vs Filtering

Both approaches are commonly used to retrieve related data.

Nested Routes

Used when there is a clear parent-child relationship.

Example
/blogs/101/comments

Meaning:

"Get all comments for blog 101."

Filtering Using Query Parameters

Used when more flexible searches are required.

Example
/comments?post_id=101

Meaning:

"Get comments whose post ID is 101."

Filtering is especially useful for:

- Searching
- Sorting
- Pagination
- Complex queries

Ways to Pass Data in REST APIs

REST APIs support three primary ways of sending data.

1. Path Parameters

Used for resource identifiers.

Example
/users/42

Here, **42** is a path parameter.

Use Cases

- User IDs
- Product IDs
- Order IDs

2. Query Parameters

Used for filtering, sorting, and pagination.

Example
/users?age=20&sort=name

Common Uses

- Filtering records
- Sorting results
- Searching
- Pagination

3. Request Body

Used when sending complex or sensitive data.

Example

```
{
  "username": "muskan",
  "password": "********"
}
```

The request body is the preferred location for large payloads and confidential information.

Security Considerations

Path parameters and query parameters are visible in the URL.

Avoid Sending Sensitive Information Through URLs

Bad Example:

/login?username=muskan&password=123456

URLs can be stored in:

- Browser history
- Server logs
- Proxy logs

Recommended Approach

Use Request Body:

```
{
  "username": "muskan",
  "password": "123456"
}
```

This provides better security and privacy.

Summary

- REST APIs provide a standard and scalable way for applications to communicate through HTTP methods. Every REST endpoint consists of an HTTP method and a resource path.
- JSON is the most commonly used data format because it is lightweight and easy to understand.
- HTTP methods such as GET, POST, PUT, PATCH, and DELETE are used to perform CRUD operations.
- PUT replaces an entire resource, whereas PATCH updates only selected fields.
- Nested routes help represent relationships between resources clearly while **query parameters** are ideal for filtering, sorting, searching, and pagination.

- Data can be passed using path parameters, query parameters, or the request body.
- Sensitive information should always be sent in the request body rather than the URL.

REST API Response

In a REST API-based system, communication happens between a **client** (web application, mobile application, or another service) and a **server**.

Whenever a client sends a request, the server processes it and sends back a **response**.

A response informs the client:

- Whether the request was successful
- Whether an error occurred
- What data should be displayed or processed

Understanding API responses is important in:

- System Design
- Backend Development
- Distributed Systems
- Client-Server Architecture

A well-designed response improves:

- Communication between systems
- Frontend integration
- Debugging and troubleshooting
- Scalability
- Overall user experience

What is a REST API Response?

A REST API response consists of two main parts:

1. Status Code

A numeric code that indicates the outcome of the request.

2. Response Body

The actual data returned by the server, usually in JSON format.

Example:

HTTP/1.1 200 OK

{

"id": 1,

"name": "Muskan"

}

In this example:

- 200 is the status code.
- The JSON object is the response body.

Understanding HTTP Status Codes

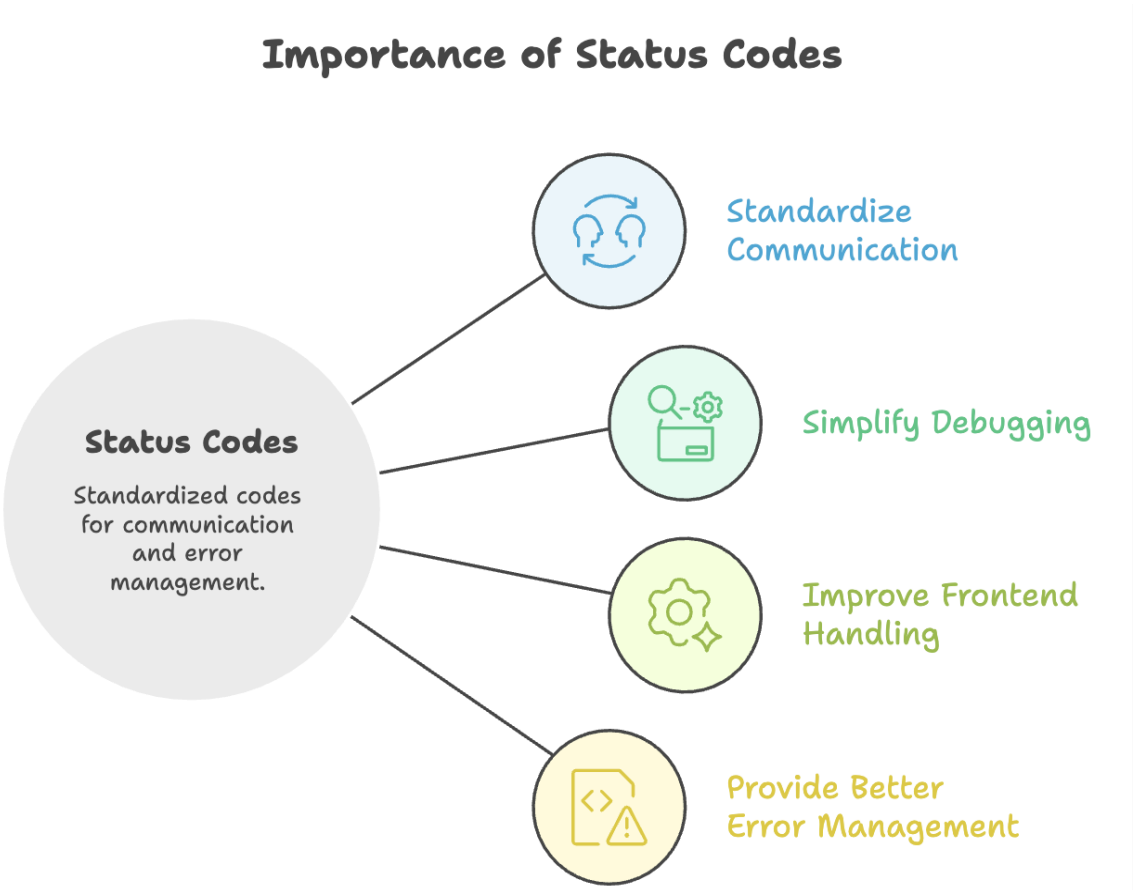
HTTP status codes are standardized codes used by servers to indicate the result of a request:

- request success,
- client-side errors,
- server-side errors,
- or redirections.

They help the client understand:

- What happened
- Whether the request succeeded
- Whether corrective action is required

Status codes are grouped based on their first digit.



HTTP Status Codes

Status codes are standardized indicators that tell the client whether a request succeeded, failed, or requires further action. They are grouped into categories by their first digit.

2xx - Success Responses

These status codes indicate that the request was successfully processed.

Code	Name	Meaning
200	OK	Request completed successfully
201	Created	New resource created successfully

204	No Content	Request succeeded but no data needs to be returned

- **200 OK**

  Used when data is successfully retrieved.

  Example
  GET /users/1

  Response:

  200 OK

  ```
  {
      "id": 1,
      "name": "Muskan"
  }
  ```

- **201 Created**

  Used when a new resource is successfully created.

  Example
  POST /users

  Response:

  201 Created

  ```
  {
      "id": 10,
      "name": "Akshay"
  }
  ```

- **204 No Content**

  Used when the operation succeeds but there is nothing to return.

  Example
  DELETE /users/10

  Response:

204 No Content

No response body is returned.

3xx - Redirection Responses

These codes indicate that the requested resource has moved or is temporarily available at another location.

Code	Name	Meaning
301	Permanent Redirect	Resource permanently moved
302	Temporary Redirect	Resource temporarily moved

- **301 Permanent Redirect**

  The requested URL has permanently changed.

  Example:

  oldsite.com → newsite.com

  Clients should update references to the new URL.

- **302 Temporary Redirect**

  The resource is temporarily available at another location.

  The original URL may become valid again later.

4xx - Client Error Responses

These status codes indicate that the client sent an invalid request.

Code	Name	Meaning
400	Bad Request	Invalid request data

401	Unauthorized	Authentication required
403	Forbidden	Permission denied
404	Not Found	Resource does not exist

- **400 Bad Request**

  The request contains invalid or malformed data.

  Example

  ```
  {
          "email": "invalid-email"
  }
  ```

  The server rejects the request because the format is incorrect.

  Response:

  400 Bad Request

- **401 Unauthorized**

  The user has not been authenticated.

  Think of it as:

  "Who are you?"

  Example

  - Missing authentication token
  - Invalid login credentials

  Response:

  401 Unauthorized

- **403 Forbidden**

  The user is authenticated but lacks permission.

  Think of it as:

  "I know who you are, but you are not allowed to access this resource."

  Example

  A normal user attempts to access an administrator endpoint.

  Response:

  403 Forbidden

401 vs 403

Status Code	Meaning
401 Unauthorized	Identity not verified
403 Forbidden	Identity verified, but permission denied

- **404 Not Found**

  The requested resource does not exist.

  Example
  GET /users/99999

  If user 99999 does not exist:

  404 Not Found

5xx - Server Error Responses

These status codes indicate problems on the server side.

Code	Name	Meaning
500	Internal Server Error	Unexpected server failure

- **500 Internal Server Error**

  Occurs when the server encounters an unexpected issue.

  Possible causes:

  - Database connection failure
  - Null pointer exceptions
  - Service crashes
  - Infrastructure failures

  Response:

  500 Internal Server Error

Important Best Practice

Never expose internal implementation details to users.

❌ Bad Response

```
{
  "error": "Database connection failed at db-server-01"
}
```

✅ Better Response

```
{
  "message": "Something went wrong. Please try again later."
}
```

Detailed information should be stored in logs instead.

Understanding Response Bodies

The response body contains the actual data returned by the API.

REST APIs usually return data in JSON format.

A response body can take several forms:

1. Array
2. Object
3. Nested Object

1. Array Response

Used when multiple records are returned.

Example

```
[
  {
        "id": 1,
        "name": "Muskan"
  },
  {
   "id": 2,
        "name": "Akshay"
  }
]
```

Typical use case:

GET /users

Returns a collection of users.

2. Object Response

Used when a single resource is returned.

Example

```
{
  "id": 1,
  "name": "Muskan"
}
```

Typical use case:

GET /users/1

Returns one specific user.

3. Nested Object Response

Used when related data is returned together.

Example

```
{
 "id": 1,
 "name": "Muskan",
 "address": {
        "city": "Chennai",
        "country": "India"
 }
}
```

This structure keeps related information grouped together.

Wrap Responses in an Object

It is recommended to return the response wrapped inside an **object** rather than a raw array.

Raw Array:

```
[
 {
        "id": 1
 },
 {
        "id": 2
 }
```

]

Wrapped Response:


```
{
  "data": [
        {
        "id": 1
        },
        {
        "id": 2
        }
  ]
}
```


Benefits of Object-Wrapped Responses

1. Easy Extensibility

Additional information can be added without changing the structure.

Example


```
{
  "data": [
        {
        "id": 1,
        "name": "Muskan"
        }
  ],
  "total": 1,
  "page": 1
}
```

2. Supports Pagination

Useful when returning large datasets.

```
{
 "data": [...],
 "page": 1,
 "size": 20,
 "totalPages": 10
}
```

3. Supports Metadata

Additional information can be included.

```
{
 "data": [...],
 "timestamp": "2026-06-14T10:30:00Z"
}
```

4. Supports Error Information

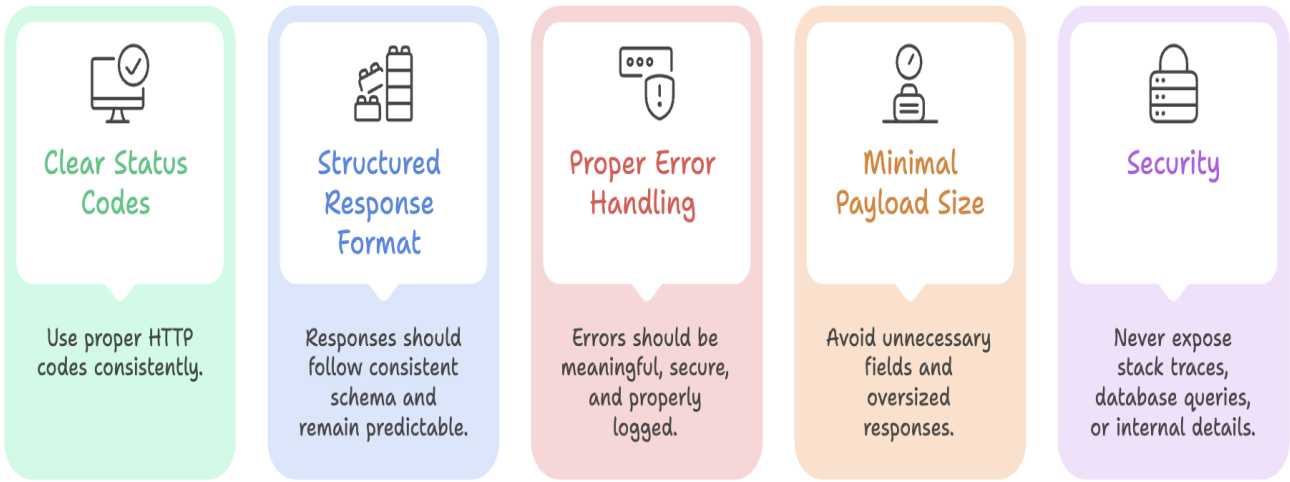
```
{
 "error": true,
 "message": "User not found"
}
```

Recommended Response Structure

```
{
 "data": [
        {
        "id": 1,
        "name": "Muskan"
        },
```

```
        {
        "id": 2,
        "name": "Akshay"
        }
  ],
  "total": 2,
  "page": 1
}
```


This approach is flexible and backward-compatible.



Summary

- A REST API response consists of a **status code** and a **response body**.
- Status codes communicate whether a request succeeded, failed, or requires further action.
- **2xx** indicates success, **4xx** indicates client-side errors, and **5xx** indicates server-side failures.

- **401 Unauthorized** means authentication is missing or invalid, while **403 Forbidden** means the user lacks permission.
- Response bodies are typically returned in **JSON** format and may contain arrays, objects, or nested objects.
- Wrapping responses inside an object improves flexibility, scalability, and backward compatibility.
- Sensitive server details should never be exposed in API responses and should instead be captured through logging and monitoring systems.

Relational Databases (RDBMS)

Data is one of the most valuable assets in modern software applications. Whether it is a social media platform, e-commerce website, banking system, or learning management platform, every application requires a reliable way to store, organize, and retrieve data efficiently.

Databases are broadly classified into two major categories:

Approach	Database Type	Examples
SQL	Relational Databases (RDBMS)	PostgreSQL, MySQL, Oracle, SQL Server
NoSQL	Non-Relational Databases	MongoDB, Cassandra, Redis, Neo4j

Relational databases are among the most widely used database systems because they provide a structured way to organize data and establish relationships between different entities.

What is a Relational Database?

A **Relational Database Management System (RDBMS)** is a database system that stores data in the form of **tables**, where data is organized into rows and columns.

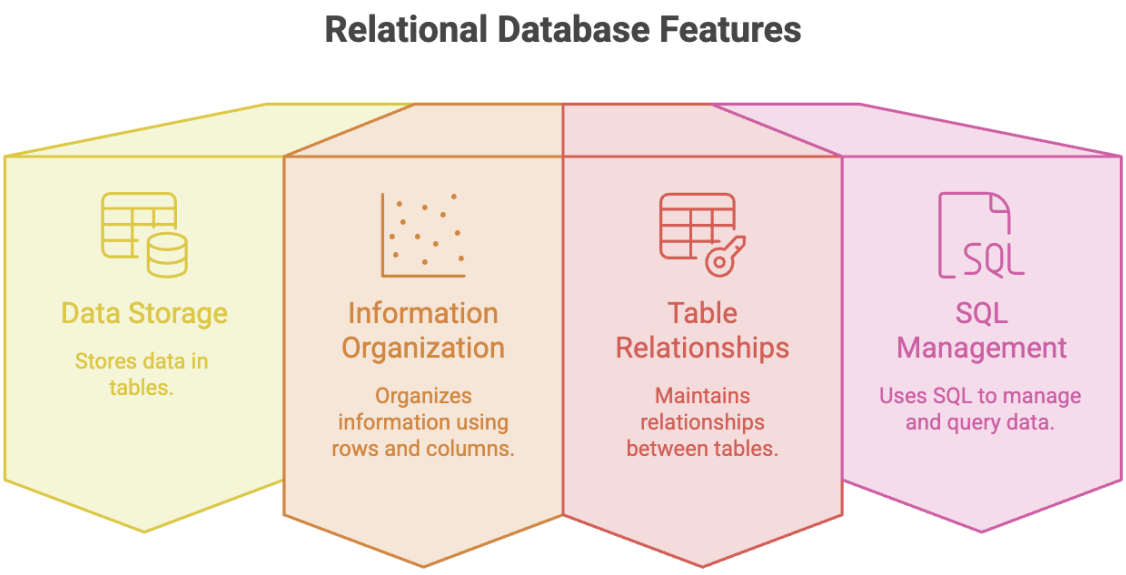
The term **relational** refers to the ability to establish relationships between different tables.

Example:

Consider a blogging platform:

- Users create posts.
- Posts receive comments.
- Comments are written by users.

All these entities are related to one another and can be connected using relational database concepts.



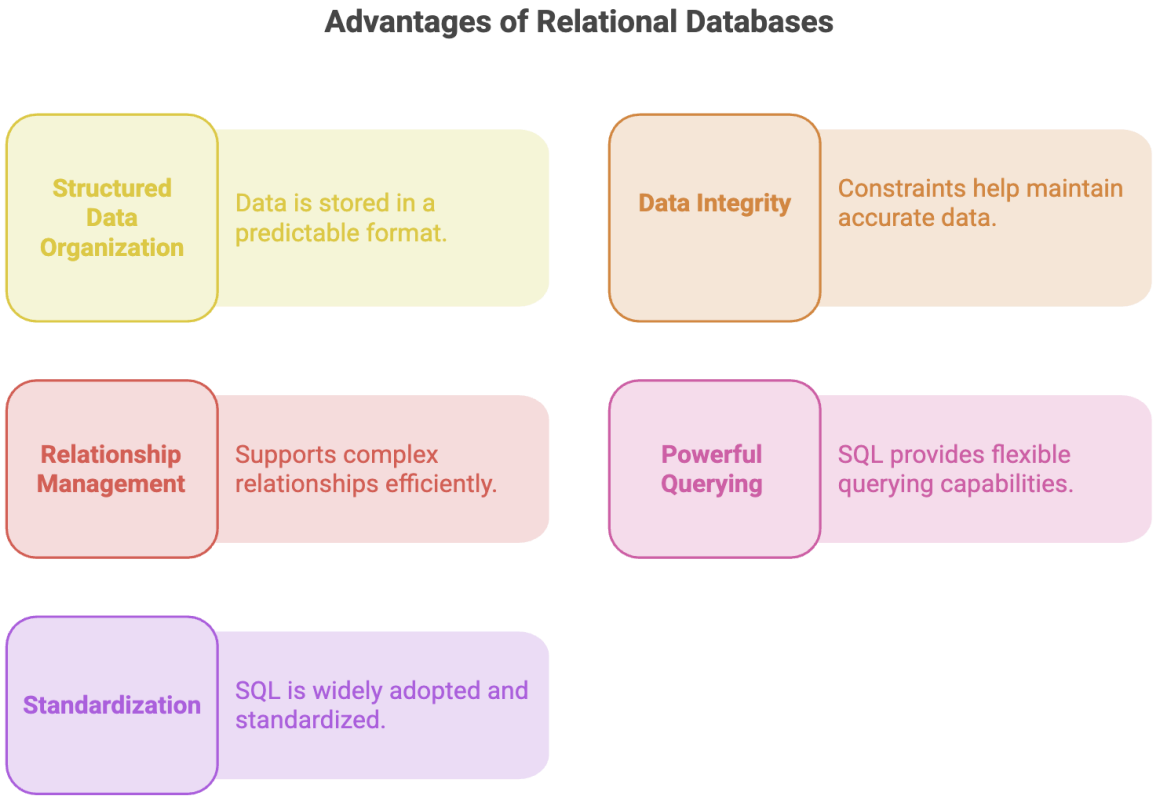
Why Use Relational Databases?

Relational databases are designed to solve several common data management challenges:

- Organizing large volumes of structured data
- Eliminating unnecessary data duplication
- Maintaining data consistency
- Supporting relationships between entities
- Enabling efficient data retrieval and reporting

Relational databases are particularly suitable when:

- Data follows a predefined structure
- Relationships between data are important
- Data accuracy and consistency are critical



Core Building Blocks of Relational Databases

Understanding relational databases begins with understanding their fundamental components.

1. Entity

An **Entity** represents a real-world object or concept about which information needs to be stored.

Examples

- User
- Product
- Order
- Employee
- Student
- Course
- Video
- Comment

In relational databases, each entity is typically represented as a table.

Example: Social Media Application

Possible entities include:

- Users
- Posts
- Comments
- Likes
- Messages

Each entity stores a specific type of information.

2. Table

A **Table** is a collection of related data organized into rows and columns.

Each table represents a single entity.

Example: Users Table

id	first_name	last_name	mobile
1	Rahul	Sharma	9876543210
2	Priya	Verma	9123456789
3	Amit	Patel	9988776655

This table stores information about users.

3. Attributes (Columns)

Attributes represent the properties of an entity.

In a database table, attributes are represented as columns.

Example: User Entity

Attribute
id
first_nam
e
last_name
mobile

Each attribute describes a specific characteristic of a user.

Real-World Analogy

Consider a student.

A student may have:

- Student ID
- Name
- Email Address
- Phone Number
- Date of Birth

These properties become columns in a database table.

4. Records (Rows)

A **Record** represents a complete instance of an entity.

Each row contains values for all attributes of a particular entity.

Example

id	first_name	last_name	mobile
1	Rahul	Sharma	9876543210

This row represents one user record.

Example: Blog Application Database

A blogging platform may contain multiple related tables.

Users Table

id	first_name
1	Rahul

2	Priya

Posts Table

id	user_id	title
1	1	System Design Basics
2	2	SQL vs NoSQL

Comments Table

id	post_id	user_id	comment_text
1	1	2	Great explanation!
2	1	1	Thank you!

Each table stores a different type of information while maintaining logical relationships with other tables.

Naming Conventions in Relational Databases

Following standard naming conventions improves readability, maintainability, and consistency.

Table Naming

Table names are generally written in plural form because they store multiple records.

Examples

users

posts

comments

orders

products

Why Use Plural Names?

A table contains multiple records rather than a single record.

Column Naming

Column names should:

- Be descriptive
- Use singular form
- Clearly indicate the stored information

Examples

first_name

last_name

mobile

created_at

updated_at

Primary Key

Most tables contain a unique identifier called a **Primary Key**.

Example
id

A primary key uniquely identifies every row in a table.

Benefits of a Primary Key

- Prevents duplicate records
- Enables faster lookups
- Supports relationships between tables

Relationships in Relational Databases

One of the most powerful features of relational databases is the ability to create relationships between entities.

Relationships connect tables and allow data to be linked logically.

Why Are Relationships Important?

Consider a blogging platform.

We may need answers to questions such as:

- Which user created a particular post?
- Which comments belong to a specific post?
- Which user wrote a specific comment?

Relationships help answer these questions efficiently.

Common Types of Relationships

One-to-One (1:1): One record in Table A is associated with one record in Table B.

Example:

- User ↔ User Profile

One-to-Many (1:N): One record in Table A can be associated with many records in Table B.

Example:

- One User can create many Posts.

Many-to-Many (M:N): Multiple records in Table A can be associated with multiple records in Table B.

Example:

- Students ↔ Courses

A student can enroll in multiple courses, and a course can have multiple students.

SQL and Relational Databases

Relational databases use **SQL (Structured Query Language)** to interact with data.

SQL provides a standardized way to manage and manipulate database records.

SQL Capabilities

- Create tables
- Insert records
- Retrieve data
- Update existing records
- Delete records

Common SQL Operations

Operation	Purpose
CREATE	Create database objects such as tables
INSERT	Add new records
SELECT	Retrieve data

UPDATE	Modify existing records
DELETE	Remove records

These operations form the foundation of database interactions.

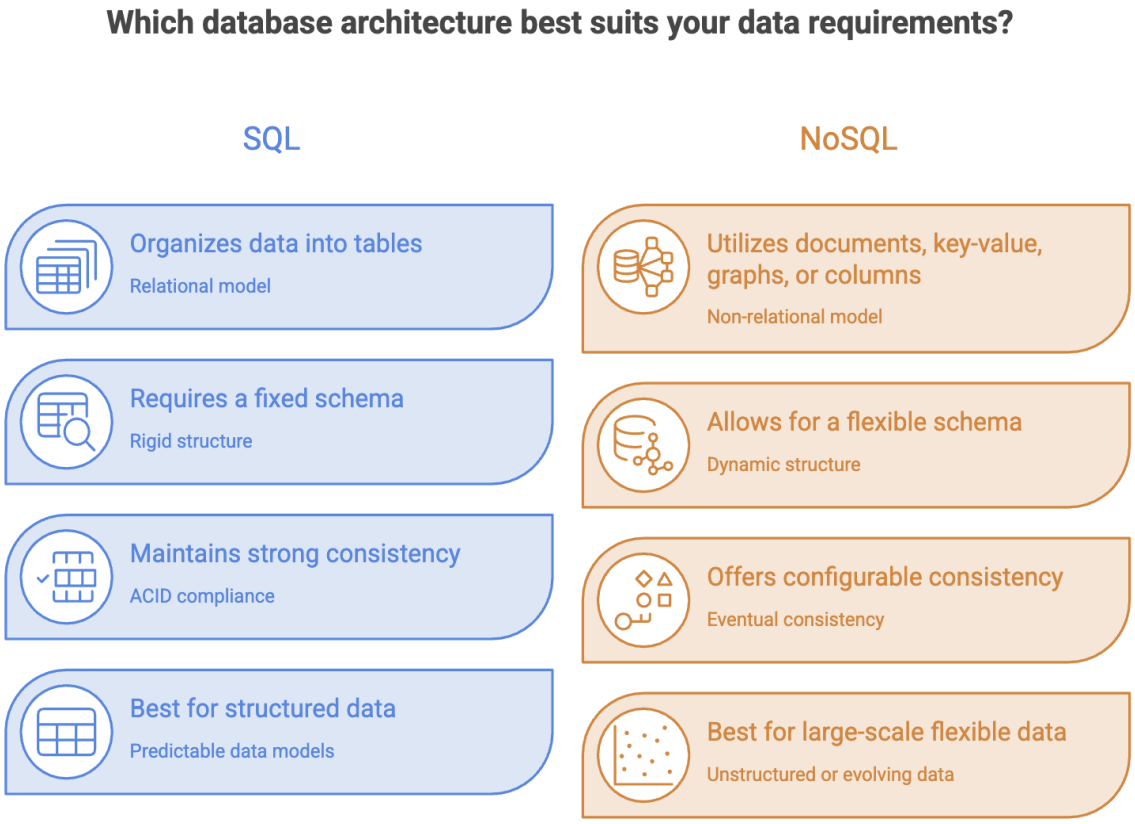
When Should You Use Relational Databases?

Relational databases are the preferred choice when:

- Data is highly structured
- Relationships are important
- Data consistency is critical
- Transactions must be reliable
- Business rules require strict enforcement

Common Use Cases

- Banking Systems
- E-Commerce Platforms
- ERP Systems
- Student Management Systems
- Inventory Management Applications
- Hospital Management Systems
- Human Resource Management Systems



Summary

- Relational databases store data in structured tables consisting of rows and columns.
- An **entity** represents a real-world object and is typically mapped to a database table.
- **Attributes** define the properties of an entity, while records store actual instances of data.
- Tables are generally named in plural form, while columns describe individual properties of an entity.
- Relationships allow tables to be connected and are commonly categorized as one-to-one, one-to-many, and many-to-many.
- Foreign keys establish relationships between tables and help maintain data integrity.
- SQL is the standard language used to create, query, update, and manage relational data.

- Relational databases are best suited for applications that require structured data, strong consistency, reliable transactions, and complex relationships.

Database Constraints

In any relational database, simply storing data is not enough. The data must remain **accurate, consistent, and reliable** throughout the lifetime of an application.

Consider the following situations:

- Multiple users register with the same email address.
- A post is created for a user who does not exist.
- Important fields such as name or email are left empty.
- Invalid values such as negative account balances are stored.

Such issues can lead to:

- Data inconsistency
- Application errors
- Unreliable business operations

To prevent these problems, relational databases provide **constraints**.

Database constraints are rules that control what data can be stored in a database, ensuring that only valid and meaningful data is maintained.

What are Database Constraints?

Database constraints are rules enforced on tables and columns to maintain:

- Data integrity
- Data consistency
- Data reliability

They define which values are allowed and prevent invalid operations that could compromise data quality.

A database constraint is a rule that restricts the values that can be inserted, updated, or deleted in a table to ensure the accuracy and reliability of stored data.

Unlike application-level validations, constraints operate directly at the database level and protect data regardless of which application accesses the database.

Why are Constraints Important?

Modern systems often have:

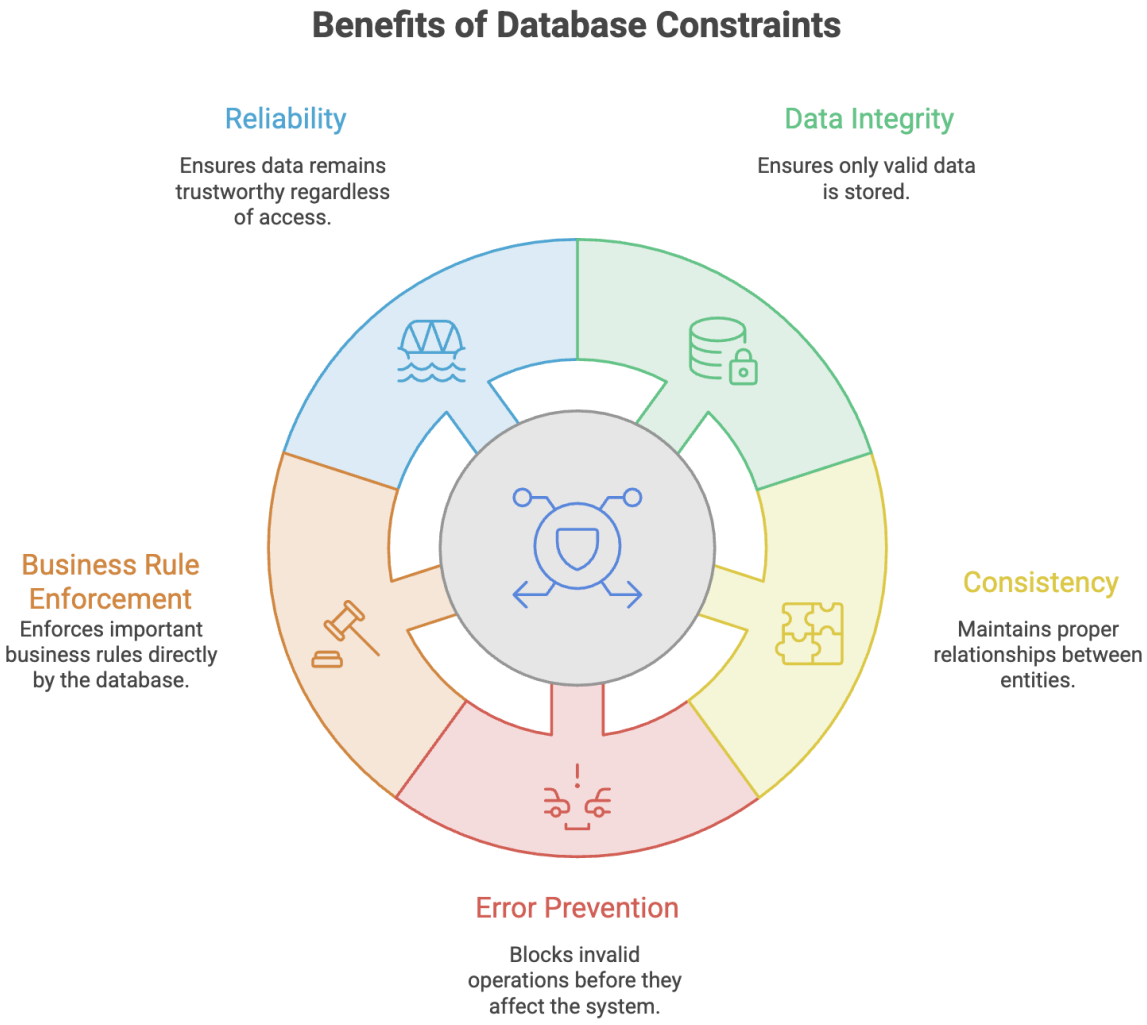
- Multiple applications
- APIs
- Microservices
- Database administrators
- End users

interacting with the same database.

If validation exists only in application code:

- Different applications may implement different rules.
- Software bugs may bypass validations.
- Direct database operations may introduce invalid data.

Database constraints act as the **final line of defense** against data corruption.



Types of Database Constraints

Constraint	Purpose	Example
PRIMARY KEY	Uniquely identifies each record	User ID
FOREIGN KEY	Creates relationships between tables	user_id in posts
NOT NULL	Prevents empty values	User Name
UNIQUE	Prevents duplicate values	Email Address

CHECK	Validates data against conditions	Age ≥ 18
DEFAULT	Assigns automatic values	Status = Active

1. PRIMARY KEY Constraint

What is a Primary Key?

A Primary Key uniquely identifies each record in a table.

Characteristics:

- Must be unique
- Cannot contain NULL values
- Each table can have only one primary key
- Commonly implemented using auto-incrementing IDs

Example

Users Table

id	first_name	last_name
1	Muskan	Garg
2	Harsh	Rawal
3	Navin	Reddy

Here, the `id` column acts as the primary key.

SQL Example

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50)
);
```

Why Primary Keys Matter

Without a primary key:

- Duplicate records become difficult to identify.
- Updates may affect incorrect rows.
- Relationships between tables cannot be created reliably.

Primary keys form the foundation of relational database design.

2. FOREIGN KEY Constraint

What is a Foreign Key?

A foreign key creates a relationship between two tables.

It references the primary key of another table and ensures that the referenced record exists.

Example

Users Table

id	first_name
1	Muskan
2	Harsh

Posts Table

id	user_id	title
1	1	System Design Basics
2	1	SQL Fundamentals
3	2	NoSQL Guide

Here:

posts.user_id

references:

users.id

SQL Example

```
CREATE TABLE posts (
        id SERIAL PRIMARY KEY,
        user_id INTEGER NOT NULL,
        title VARCHAR(200),

        FOREIGN KEY (user_id)
        REFERENCES users(id)
);
```

Benefits of Foreign Keys

- **Maintains Relationships** : Ensures valid connections between entities
- **Prevents Invalid References** : A post cannot reference a user that does not exist
- **Improves Data Consistency** : Related records remain synchronized

Invalid Scenario

If the Users table contains:

id
1
2

Attempting to insert:

user_id = 99

will fail because no user with ID 99 exists.

3. NOT NULL Constraint

What is NOT NULL?

The NOT NULL constraint ensures that a column always contains a value.

Mandatory fields cannot be left empty.

Example : Users Table If the mobile column allows NULL values, the second row is valid.

id	first_name	last_name	mobile
1	Muskan	Garg	9876543210
2	Harsh	Rawal	NULL

However, fields such as, first_name and last_name may be mandatory.

Example

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        first_name VARCHAR(50) NOT NULL,
        last_name VARCHAR(50) NOT NULL
);
```

Common Use Cases

NOT NULL is commonly applied to:

- First Name
- Last Name
- Email Address
- Password
- Product Name
- Order ID

These fields are essential and should never be empty.

4. UNIQUE Constraint

What is UNIQUE?

The UNIQUE constraint ensures that values in a column cannot be duplicated.

Every value must be distinct.

Example

id	first_name	email
1	Muskan	muskan@email.com
2	Harsh	harsh@email.com

Trying to insert another record with:

muskan@email.com

will be rejected.

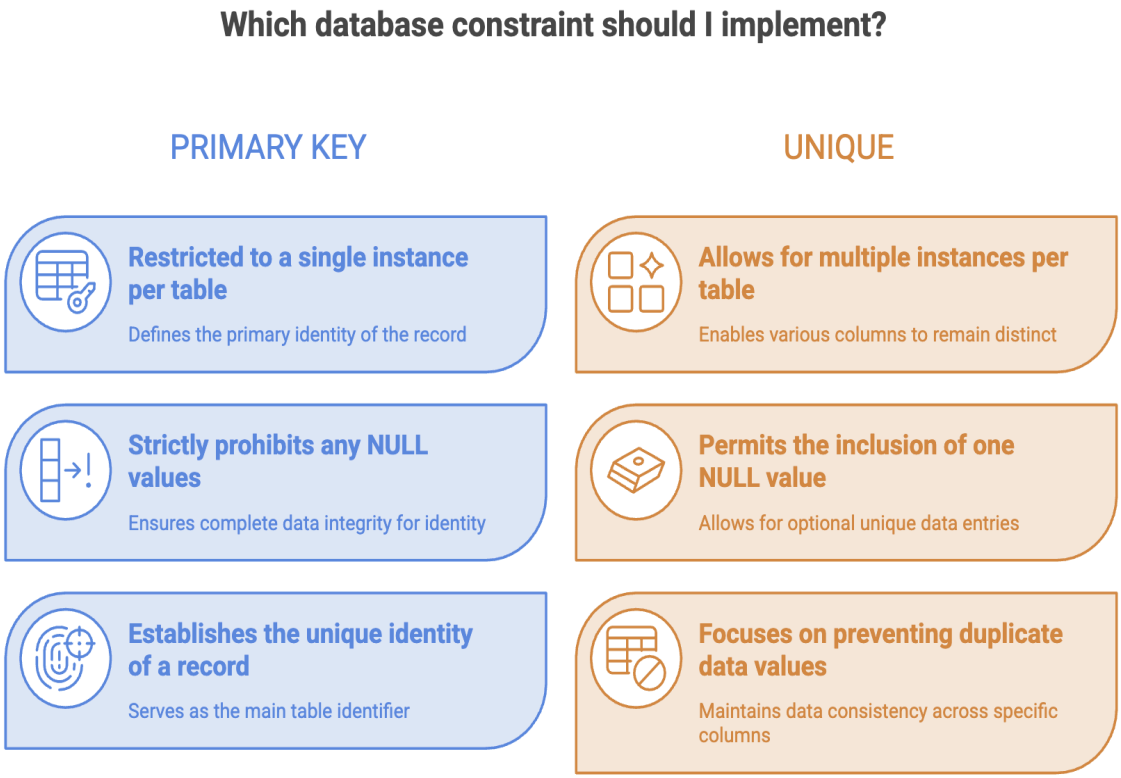
SQL Example

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        email VARCHAR(100) UNIQUE
);
```

Common Use Cases

UNIQUE is often used for:

- Email Addresses
- Usernames
- Mobile Numbers
- Employee IDs
- Account Numbers



5. CHECK Constraint

What is CHECK?

The CHECK constraint validates data against a condition before it is stored.

If the condition evaluates to false, the operation is rejected.

Example

Users must be at least 18 years old.

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        age INTEGER CHECK (age >= 18)
);
```

Valid Values

age

18
25
40

Invalid Value

age
16

Insertion fails because the condition is violated.

Common Use Cases

- Age restrictions
- Positive account balances
- Rating ranges
- Product quantities
- Percentage limits

6. DEFAULT Constraint

What is DEFAULT?

The DEFAULT constraint automatically assigns a predefined value when no value is provided.

Example

Every new user should be active by default.

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        status VARCHAR(20) DEFAULT 'active'
);
```

Insert Statement

```
INSERT INTO users(id)
VALUES (1);
```

Stored Result

id	status
1	active

The database automatically inserts the default value.

Common Use Cases

- User status
- User roles
- Boolean flags
- Default preferences
- Creation timestamps

Timestamp Example
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

Automatically stores the creation time of the record.

Combining Multiple Constraints

In real-world applications, multiple constraints are used together.

Example

```
CREATE TABLE users (
        id SERIAL PRIMARY KEY,
        first_name VARCHAR(50) NOT NULL,
        email VARCHAR(100) UNIQUE NOT NULL,
        age INTEGER CHECK(age >= 13),
        status VARCHAR(20) DEFAULT 'active'
);
```

Applied Constraints

Column	Constraints
id	PRIMARY KEY
first_name	NOT NULL

email	UNIQUE, NOT NULL
age	CHECK
status	DEFAULT

Benefits

This ensures:

- Every user has a unique identity.
- Names cannot be empty.
- Duplicate emails are prevented.
- Invalid ages are rejected.
- Default status is assigned automatically.

Foreign Key Delete Behaviors

When a parent record is deleted, the foreign key determines what happens to related child records.

1. CASCADE

Automatically deletes all related child records.

ON DELETE CASCADE

Example: Deleting a user automatically deletes all associated posts and comments.

2. SET NULL

Sets the foreign key value in child records to NULL.

ON DELETE SET NULL

The child record remains, but the relationship is removed.

3. RESTRICT

Prevents deletion of the parent record if dependent records exist.

ON DELETE RESTRICT

Useful when accidental deletion must be avoided.

4. NO ACTION

Behaves similarly to RESTRICT and is often the default behavior in many databases.

Real-World Example: Blogging Platform

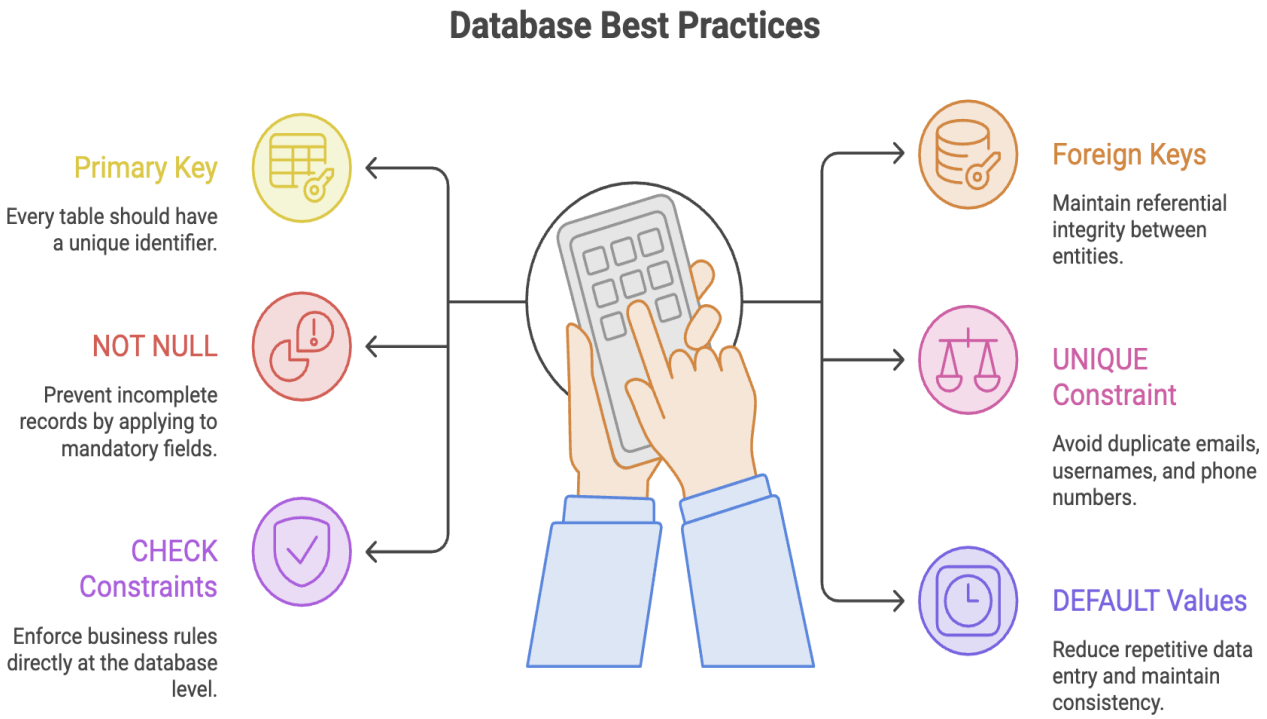
Consider a blogging application with:

- Users
- Posts
- Comments

Constraints ensure:

- Every user has a unique email address.
- Every post belongs to a valid user.
- Every comment belongs to a valid post.
- Required fields are always provided.
- Invalid data never enters the system.

Without constraints, maintaining data quality becomes increasingly difficult as the application grows.



Summary

- Database constraints are rules that ensure **data integrity, consistency, and reliability** with relational databases.
- **PRIMARY KEY** uniquely identifies each record and serves as the foundation of table relationships.
- **FOREIGN KEY** establishes relationships between tables and maintains referential integrity.
- **NOT NULL** guarantees that mandatory fields always contain values.
- **UNIQUE** prevents duplicate values in important columns such as emails and usernames.
- **CHECK** validates data against business rules before storage.
- **DEFAULT** automatically assigns predefined values when none are provided.
- Multiple constraints are typically combined to create robust and production-ready database schemas.
- Constraints play a crucial role in system design by ensuring that data remains accurate, reliable, and consistent regardless of which application accesses the database.

Database Joins

In relational databases, data is often distributed across multiple tables to reduce duplication and maintain consistency. However, real-world applications frequently require data from multiple tables to be combined and presented together.

A **join** is a SQL operation that combines data from two or more related tables based on a common column, usually a **primary key** and **foreign key** relationship.

Joins are one of the most important concepts in database design because they enable relational databases to connect and retrieve related information efficiently.

What is a Join?

A join allows SQL queries to fetch related data stored in separate tables.

A join is a database operation that combines rows from two or more tables using a related column between them.

Why Are Joins Needed?

In a well-designed relational database:

- User information is stored in a Users table.
- Blog information is stored in a Blogs table.
- Comments are stored in a Comments table.

Instead of storing all information in a single table, data is separated into logical entities. Joins help reconnect these entities whenever related information needs to be retrieved.

Example

Suppose we want to answer the following:

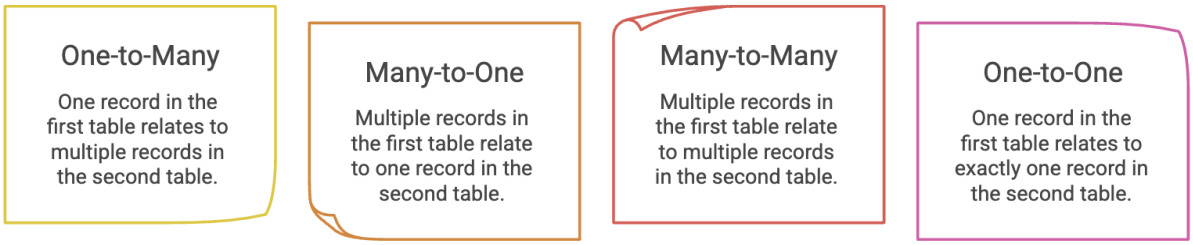
- Which user wrote a particular blog?
- Which blogs belong to a user?
- Which students are enrolled in a course?

These questions can be answered using joins.

Understanding Database Relationships

Before understanding joins, it is important to understand how tables are related.

Relational databases primarily support four types of relationships:



1. One-to-Many Relationship

A One-to-Many relationship exists when **one record in a table can be associated with multiple records** in another table.

Real-World Example

One user can create multiple blogs.

Users Table

id	first_name
1	Alice
2	Bob

Blogs Table

id	title	user_id
1	Blog on SQL	1
2	Blog on NoSQL	1
3	Blog on Caching	2

Relationship Analysis

- Alice has written two blogs.
- Bob has written one blog.
- user_id in the Blogs table acts as a **foreign key** referencing the Users table.

Common Examples

- Customer → Orders
- Department → Employees
- Category → Products
- Teacher → Students

2. Many-to-One Relationship

A Many-to-One relationship is simply the reverse view of a One-to-Many relationship.

Example

Multiple blogs belong to a single user.

Blog	User
Blog on SQL	Alice
Blog on NoSQL	Alice
Blog on Caching	Bob

From the User perspective:

One User → Many Blogs

From the Blog perspective:

Many Blogs → One User

- From the **Blogs' perspective**, many blogs point to the same user - this is Many-to-One.
- From the **Users' perspective**, one user has many blogs - this is One-to-Many.

One-to-Many and Many-to-One are the same relationship viewed from different directions.

3. Many-to-Many Relationship

A **Many-to-Many relationship** exists when multiple records from one table can relate to multiple records from another table.

Real-World Example

Students and Courses.

- One student can enroll in multiple courses.
- One course can contain multiple students.

Students Table

id	name
1	Alice
2	Bob
3	Charlie

Courses Table

id	course_name
1	Math
2	Science
3	History

Junction Table (students_courses)

id	student_id	course_id
1	1	1
2	1	2
3	2	1
4	3	2
5	3	3

Why is a Junction Table Required?

- A **junction table** (also called a bridge/mapping table) is required to maintain the Many-to-Many relationship.
- The junction table contains foreign keys referencing both related tables.
- From the above table:
    - Alice is enrolled in math and science.
    - Bob is enrolled in math.
    - Charlie is enrolled in science and history.
    - Math has Alice and Bob enrolled.
    - Science has Alice and Charlie enrolled.

Without a junction table, there is no way to represent Many-to-Many relationships in a relational database.

4. One-to-One Relationship

A **One-to-One relationship** exists when one record in a table corresponds to exactly one record in another table.

Real-World Example

Content management systems often separate metadata from actual content.

Contents Table

id	name	content_id	type
1	Intro Video	101	Video
2	Jazz Track	201	Audio
3	Tech Article	301	Blog

Videos Table

id	data
101	Video Data

Audios Table

id	data
201	Audio Data

Blogs Table

id	data
301	Blog Content

Why Use One-to-One Relationships?

Large content such as, Videos, Audio files, Images, Documents can be stored separately while keeping lightweight searchable metadata in a common table.

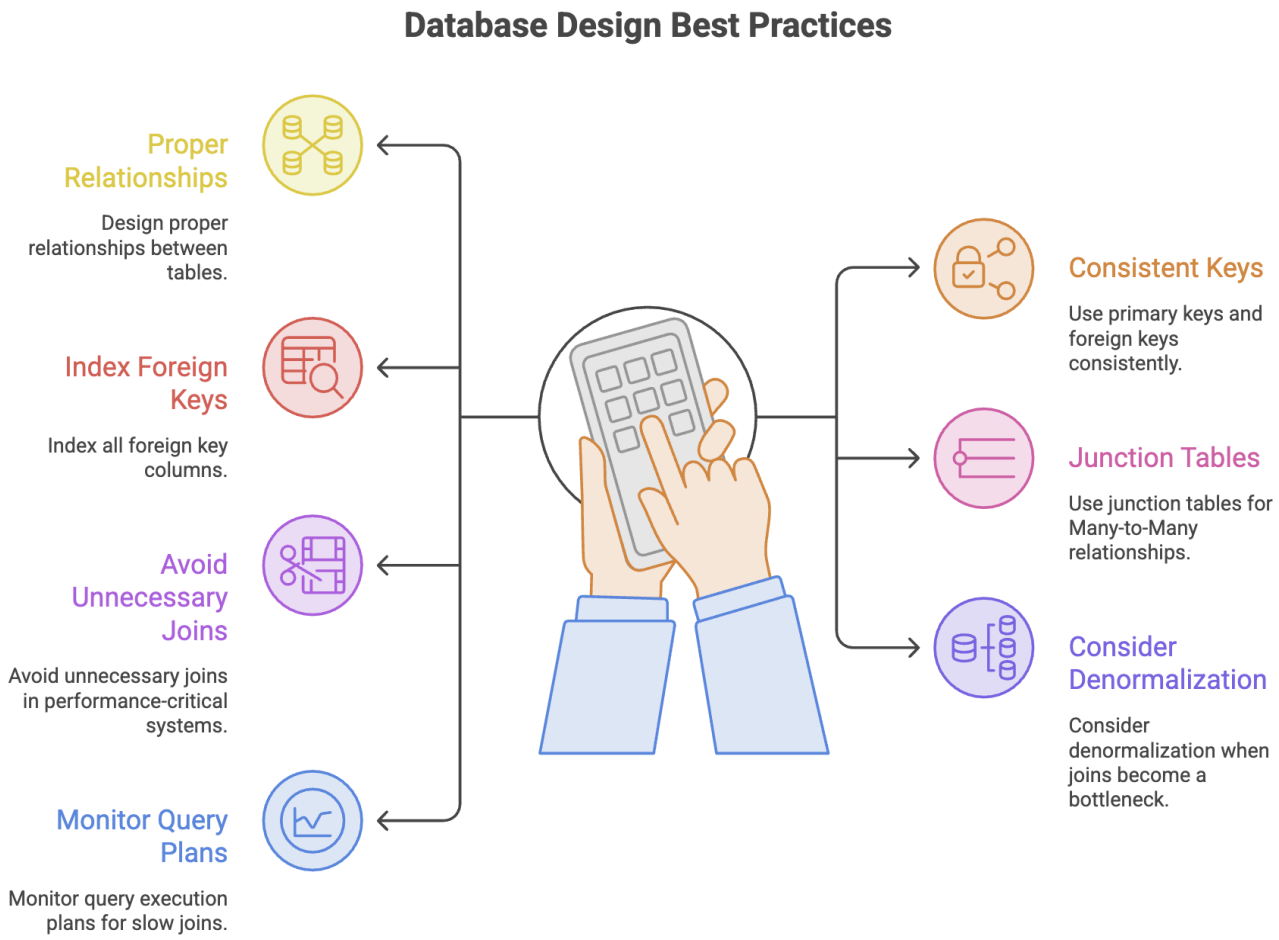
- content_id in the Contents table acts as a **foreign key** pointing to the respective data table (Videos, Audios, or Blogs).
- Each content record maps to **exactly one** record in its corresponding data table.

Advantages

- Faster searches
- Better storage management
- Improved query performance
- Cleaner database design

Relationship Summary

Relationship Type	Description	Foreign Key Location	Junction Table Required
One-to-Many	One record linked to many records	Many side	No
Many-to-One	Many records linked to one record	Many side	No
Many-to-Many	Many records linked to many records	Junction table	Yes
One-to-One	One record linked to exactly one record	Either table	No



Summary

- A **join** combines data from multiple related tables.
- Relational databases support **One-to-Many, Many-to-One, Many-to-Many, and One-to-One** relationships.
- **Foreign Keys** connect related tables and make joins possible.
- **Many-to-Many relationships** require a junction table.
- Proper indexing is essential for fast join performance.
- Excessive joins can become expensive in large-scale systems, where denormalization may be used.
- Understanding joins is fundamental for database design, backend development, and system design interviews.

NoSQL Databases

As applications grow to millions of users and handle massive amounts of data, traditional relational databases can become difficult to scale efficiently. Modern systems such as social media platforms, e-commerce websites, streaming services, and real-time applications often require databases that can scale horizontally, handle flexible data structures, and provide high performance.

This is where **NoSQL databases** come into the picture.

NoSQL databases are designed to overcome many of the scalability and flexibility challenges faced by traditional relational databases.

The Challenge with Relational Databases at Scale

Consider a Blog Application with the following relational database structure:

Table	Purpose
Posts	Stores blog posts
Content	Stores content type information
Images	Stores image data
Texts	Stores text content
Videos	Stores video content
Comments	Stores comments
Users	Stores user information

As the application grows:

- More tables are added.
- Relationships become increasingly complex.
- Queries require multiple joins.
- Scaling across multiple servers becomes difficult.

Example Challenges

- Fetching a blog post may require joins across multiple tables.
- Managing relationships across distributed servers becomes expensive.
- Query performance may degrade as data volume grows.
- For small to medium systems, relational databases work very well. However, at very large scale, alternative approaches may be needed.

Database Scaling Approaches

Databases can be scaled in two primary ways.

1. Vertical Scaling

Vertical scaling means increasing the capacity of a single server.

Examples

- More RAM
- Faster CPU
- Larger storage
- Better hardware

Advantages

- Simple to implement
- No major application changes

Limitations

- Expensive

- Physical hardware limits exist
- Eventually reaches a maximum capacity

2. Horizontal Scaling

Horizontal scaling means distributing data across multiple servers.

Examples

Instead of one server: Server 1, Server 2, Server 3, Server 4

The workload is distributed among all servers.

Advantages

- Nearly unlimited scalability
- Better fault tolerance
- Improved availability

Challenges in SQL Databases

When data is distributed across multiple servers:

- Joins become complex
- Foreign key relationships become difficult to maintain
- Cross-server queries become slower

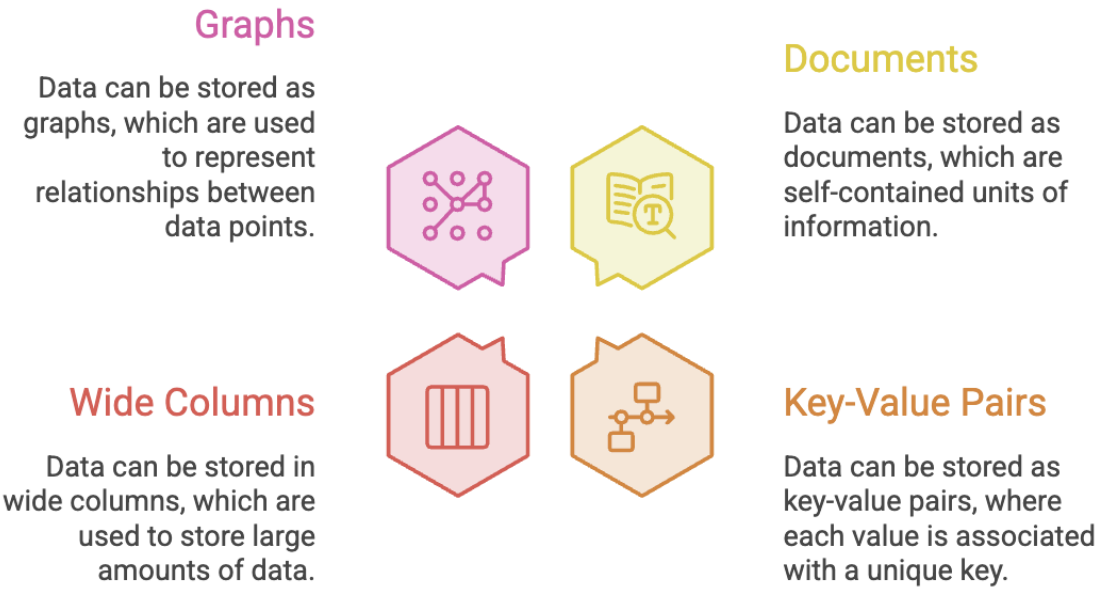
This is one of the major reasons why NoSQL databases became popular.

What is NoSQL?

NoSQL stands for **Not Only SQL**.

Unlike relational databases, NoSQL databases do not store data strictly in rows and columns.

Instead, data can be stored as:



These storage models are designed to support:

- Large-scale systems
- Flexible data structures
- High-speed operations
- Easy horizontal scaling

Document-Oriented NoSQL Example

Instead of storing related information in multiple tables, a NoSQL database can store everything inside a single document.

Blog Post Example

```
{
 "post_id": "p101",
 "author": {
        "user_id": "u1",
        "name": "Alice"
 },
```

```
 "content": {
        "type": "image",
        "data": "image_url_here"
 },
 "comments": [
        {
        "comment_id": "c1",
        "text": "Great post!"
        },
        {
        "comment_id": "c2",
        "text": "Very helpful."
        }
 ]
}
```

Benefits

- No joins required
- Faster retrieval
- Easy replication
- Easy partitioning across servers

All related data is stored together.

Advantages of NoSQL Databases

1. Easy Horizontal Scaling

NoSQL databases are designed for distributed environments.

Benefits

- Data can be split across multiple servers.
- New servers can be added easily.
- High availability can be achieved.
- Large datasets can be managed efficiently.

Common Techniques

- **Sharding:** Data is divided into smaller partitions and distributed across multiple servers.
- **Replication:** Copies of data are stored on multiple servers to improve reliability.

2. Flexible Schema (Schema-less Design)

Relational databases require every row to follow the same structure.

SQL Example

User	Type	Data	Description
User1	Image	img_url	NULL
User2	Text	text_content	NULL
User3	Code	code_snippet	Sorting Algorithm

All rows must contain the same columns.

Adding new fields often requires schema modifications.

NoSQL Solution

Each document can have a **different number and type of fields** within the same collection.

Document 1

```
{
  "user": "user1",
  "type": "image",
  "data": "img_url"
}
```

Document 2

```
{
  "user": "user2",
  "type": "text",
  "data": "text_content"
}
```

Document 3

```
{
  "user": "user3",
  "type": "java_code",
  "data": "code_snippet",
  "description": "Sorting algorithm"
}
```

Benefits

- Easy to evolve data models
- No schema migrations
- Faster development cycles

- Suitable for changing business requirements

3. Self-Contained Data

Each document contains everything required for that entity, and there is no need to maintain complex relationships.

Example: Courses Collection

Course 1

```
{
  "course_id": "c1",
  "course": "System Design"
}
```

Course 2

```
{
  "course_id": "c2",
  "course": "DSA",
  "duration": "8 weeks",
  "level": "Intermediate"
}
```

Course 3

```
{
  "course_id": "c3",
  "course": "Backend",
  "lessons": [
        "Node.js",
        "Databases",
        "APIs"
  ]
}
```

Course 4

```
{
 "course_id": "c4",
 "course": "DevOps",
 "lessons": [
        "Docker",
        "Kubernetes"
 ],
 "comments": [
        "Excellent",
        "Very Detailed"
 ]
}
```


All documents belong to the same collection despite having different structures.

Real-World Usage of NoSQL

Company	Database	Primary Use Case
Netflix	Cassandra	User activity tracking
Amazon	DynamoDB	Global e-commerce platform
Meta (Facebook)	HBase	Messaging and storage
Uber	MongoDB	Flexible real-time data
X (Twitter)	Redis	Caching and timeline rendering

These companies handle billions of requests and rely on NoSQL databases for scalability and performance.

SQL vs NoSQL

Feature	SQL (RDBMS)	NoSQL
Data Structure	Structured	Semi-structured / Unstructured
Schema	Fixed	Flexible
Relationships	Strong relationships using joins	Embedded documents
Consistency	Strong (ACID)	Usually Eventual (BASE)
Scaling	Primarily Vertical	Horizontal + Vertical
Query Complexity	Complex joins possible	Simple retrieval
Performance	Good for transactions	Good for large-scale distributed systems
Best Use Cases	Banking, Payments, ERP	Social Media, Analytics, Content Platforms

Summary

- NoSQL databases were created to solve scalability and flexibility challenges faced by traditional relational databases.
- They store data in formats such as documents, key-value pairs, wide columns, and graphs.
- NoSQL databases are designed for horizontal scaling and distributed architectures.
- They provide flexible schemas, making them ideal for rapidly changing applications.
- NoSQL databases reduce the need for joins by storing related information together.
- SQL remains the preferred choice when strong consistency and complex relationships are required.
- Modern large-scale applications often use both SQL and NoSQL databases together, selecting the best tool for each specific requirement.

Types of NoSQL Databases

As applications scale and handle increasingly diverse types of data, traditional relational databases may not always be the best fit. NoSQL databases provide alternative ways of storing and retrieving data, allowing systems to achieve better scalability, flexibility, and performance.

Unlike relational databases that store data in tables with fixed schemas, NoSQL databases use different storage models optimized for specific use cases.

NoSQL databases are generally classified into four major categories:

1. Key-Value Databases
2. Columnar Databases
3. Graph Databases
4. Document Databases

Each type solves a different set of problems and is widely used in modern system design.

1. Key-Value Databases

A Key-Value database stores data in the form of a unique **key** and its corresponding **value**.

Think of it like a dictionary:

Key → Value

The key acts as a unique identifier, while the value contains the actual data.

Structure

user_101 → Alice

user_102 → Bob

user_103 → Charlie

The value can be almost anything:

- Text
- Numbers
- JSON Objects
- Arrays
- Images
- Binary Files

Supported Value Types

Value Type	Example
String	`"Hello World"`
Number	`42`, `3.14`
JSON Object	`{"name":"Alice"}`
Array	`[1,2,3]`
Nested Object	Complex hierarchical data
BLOB	Images, videos, files
Byte Array	Raw binary data

Example: Blog Post

```
{
  "posts": [
        {
        "post_id": 1,
        "content": "Learning NoSQL",
        "comments": [
        {
        "comment_id": 1,
        "comment": "Very Helpful!"
        }
        ]
        }
  ]
}
```

All related information is stored together within a single value.

No joins are required.

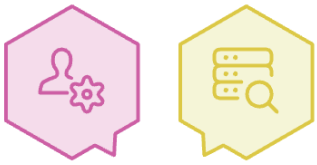
Advantages

- **Extremely Fast Access:** Data can be retrieved directly using its key.
- **Simple Data Model:** No complex relationships or joins.
- **Highly Scalable:** Easy to distribute across multiple servers.

Common Use Cases

User Preferences

Store user settings such as themes, language, and notifications. This allows users to customize their experience.



Caching

Store frequently accessed data for quick retrieval. Examples include product details, search results, and API responses.

Shopping Carts

Maintain temporary user-specific data for shopping carts. This includes items added, quantities, and prices.

Session Management

Store user login sessions to maintain user authentication. This allows users to stay logged in without re-entering credentials.

Popular Key-Value Databases

- Redis
- Amazon DynamoDB
- Memcached

2. Columnar Databases

Columnar databases store data column by column. This makes them highly efficient for analytical and reporting workloads.

Example

Students Table

id	name	marks
1	Alice	85
2	Bob	72
3	Charlie	90

Suppose we need to calculate:

Average Marks

Row-Based Database

Reads:

1, Alice, 85

2, Bob, 72

3, Charlie, 90

Even though only marks are needed, the database also reads id and name.

Columnar Database

Reads only:

85

72

90

This significantly reduces disk reads and improves performance.

Performance Characteristics

Operation	Performance
Reading	Very Fast
Writing	Slower

Why Writes Are Slower?

Data must be written separately into different column structures.

Example:

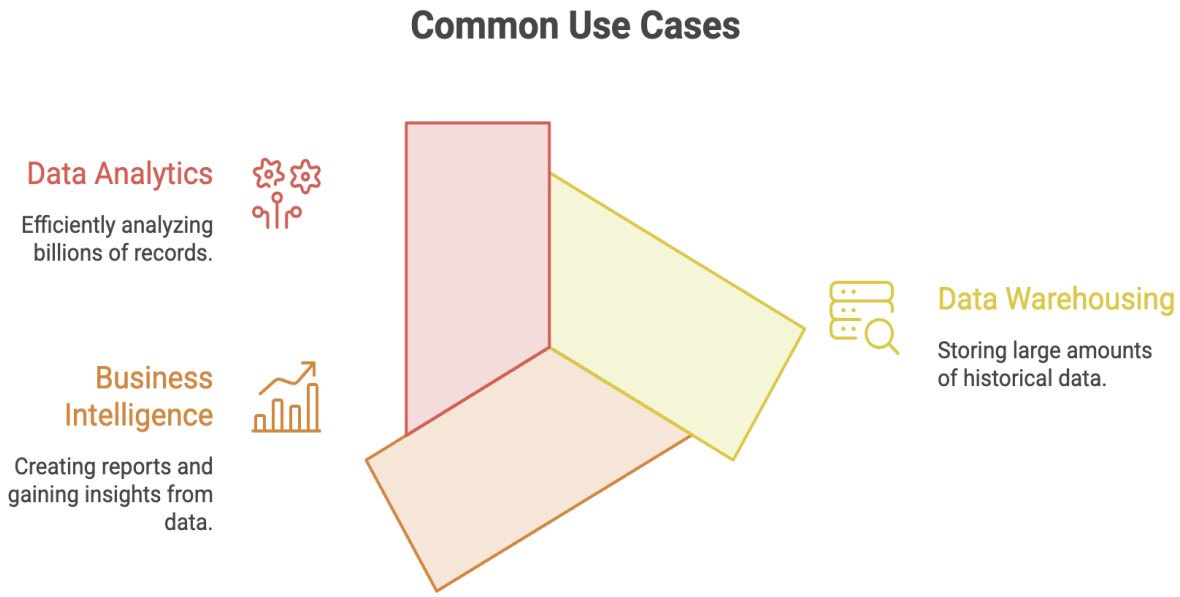
id column

name column

marks column

Advantages

- **Efficient Analytics:** Ideal for calculations such as: SUM, AVG, COUNT, MIN, MAX
- **Reduced Storage:** Similar values within a column can be compressed effectively.
- **Faster Reporting:** Perfect for dashboards and business reports.



Popular Columnar Databases

- **BigQuery** by Google
- **Redshift** by Amazon

- **Snowflake** by Snowflake Inc.

3. Graph Databases

Graph databases store data as:

- Nodes
- Edges
- Properties

This model focuses on relationships between entities.

Core Components

1. **Node:** Represents an entity.

   Examples:

   - Student
   - Course
   - College
   - User
   - Product

2. **Edge:** Represents a relationship between nodes.

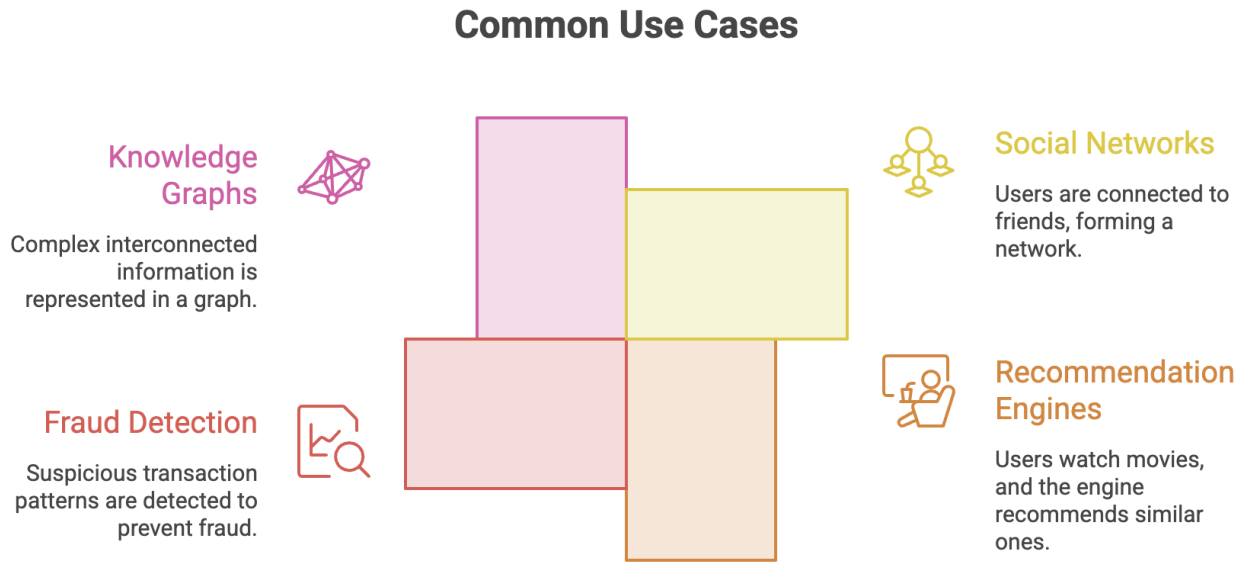
   Examples:

   Student → Enrolled In → Course

   Properties: Additional information attached to nodes or edges.

Advantages

- **Relationship-Focused:** Relationships are first-class citizens.
- **Pattern Discovery:** Can easily identify: Connections, Dependencies, Recommendations, Fraud patterns
- **Flexible Traversal:** Navigate relationships without expensive joins.



Popular Graph Technologies

- Neo4j (Cypher Query Language)
- Gremlin
- SPARQL

4. Document Databases

Document databases store data as self-contained documents.

Documents are usually stored in:

- JSON
- BSON
- JSON-like formats

Each document can have its own structure.

Example

User 1

```
{
  "user_id": "u1",
  "name": "Alice",
  "email": "alice@mail.com"
}
```

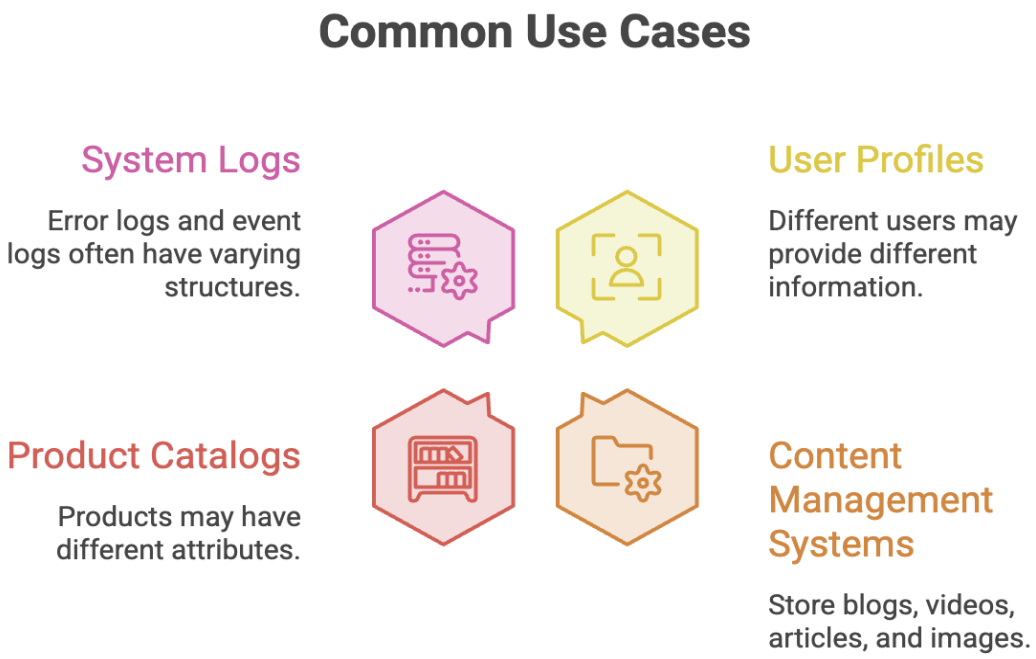
User 2

```
{
  "user_id": "u2",
  "name": "Bob",
  "email": "bob@mail.com",
  "phone": "+1234567890",
  "address": {
        "city": "New York",
        "zip": "10001"
  }
}
```

Both documents can exist in the same collection despite having different fields.

Advantages

- **Schema Flexibility:** No predefined structure is required.
- **Easy Evolution:** New fields can be added without modifying existing documents.
- **Self-Contained Data:** Related information can be stored together.
- **Fast Retrieval:** Most required data is available within a single document.



Popular Document Databases

- MongoDB
- CouchDB

Comparison of NoSQL Database Types

Type	Data Model	Read Performance	Write Performance	Best Use Cases
Key-Value	Key → Value	Very Fast	Fast	Caching, sessions, simple lookups
Columnar	Column-Oriented	Very Fast (Analytics)	Slower	Reporting, data warehousing
Graph	Nodes + Edges	Moderate	Moderate	Relationships, recommendations
Document	JSON Documents	Fast	Fast	Flexible schemas, content systems

Summary

- NoSQL databases are designed to address scalability, flexibility, and performance challenges that traditional relational databases may face at large scale.
- Key-value databases provide extremely fast access and are ideal for caching and session management.
- Columnar databases are optimized for analytical workloads and large-scale reporting systems.
- Graph databases excel at managing and analyzing relationships between interconnected entities.
- Document databases offer schema flexibility and are widely used for content platforms, user profiles, and modern web applications.
- Selecting the right NoSQL database depends on the application's access patterns, scalability requirements, and data model.

Caching

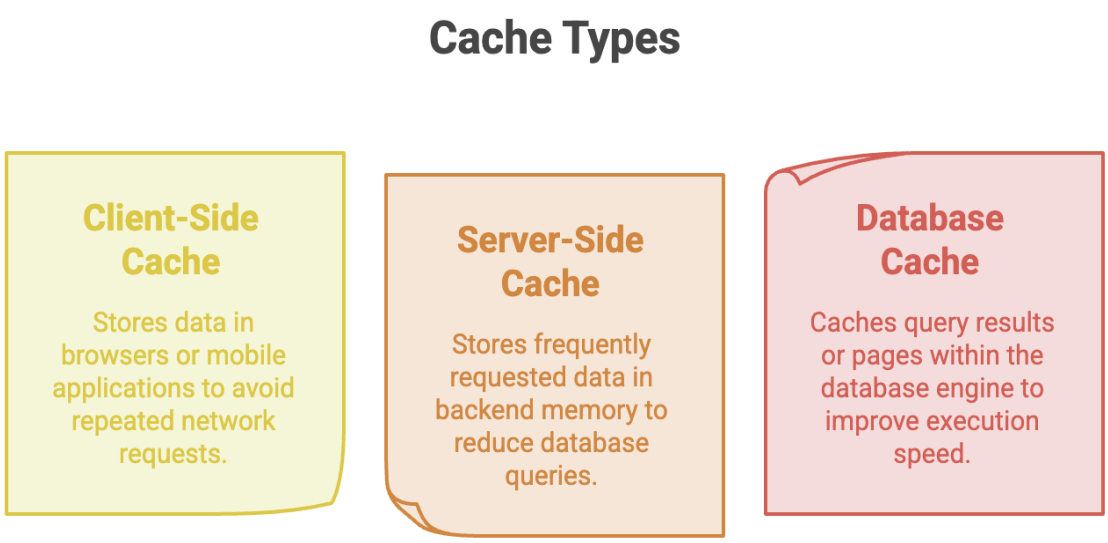
Caching is a technique used to improve application performance by storing frequently accessed data in a temporary high-speed storage layer called a cache. Instead of repeatedly querying the database for the same information, the application retrieves it directly from memory, reducing latency and improving response times.

Problem Statement

Consider a user enrolled in six courses, each containing four sections. Without caching, loading the dashboard may require 24 database requests. These repeated database operations increase response time and place unnecessary load on the database server.

Solution

After the first request, the frequently accessed course data can be stored in the cache. Future requests retrieve the data directly from the cache until it expires or is updated.



Cache Terminology

Cache Hit

The requested data is available in the cache and is returned immediately without querying the database.

Cache Miss

The requested data is not available in the cache, so the application fetches it from the database and may store it in the cache for future requests.

TTL (Time To Live)

The duration for which a cache entry remains valid. Once the TTL expires, the key-value pair is automatically removed from the cache to free up space from newer data.

How Data is Stored?

- Cache stores information as key-value pairs.
- The key uniquely identifies the cached object.
- The value contains the actual data together with metadata such as TTL.
- Expired entries are evicted automatically to free memory.

Why Not Cache All Data?

Reason	Explanation
Limited Memory	Cache memory is much smaller and more expensive than database storage.
Frequently Accessed Data	Only commonly requested data provides significant performance benefits.
Data Consistency	Cached data should be refreshed regularly to prevent stale information.

Summary

- Caching reduces latency and database load.
- Cache hits provide faster responses than cache misses.
- TTL helps remove stale data automatically.
- Caching complements the database rather than replacing it.

Caching Strategies

Caching strategies determine how data is stored, retrieved, and synchronized between the cache and the database. Choosing the correct strategy helps balance performance, consistency, scalability, and reliability within a system.

1. Read-Through Cache

A strategy designed to handle **read operations** from the cache only. Write operations always go directly to the database.

How it works:

Operation	Flow
Read (Cache Hit)	Request → Cache → Response
Read (Cache Miss)	Request → Cache (miss) → Database → Store in Cache → Response from Cache
Write	Request → Database directly (cache is not updated)

Key Points:

- Data is never written to the cache explicitly; it gets populated only on a cache miss.
- First read for any data always results in a cache miss, subsequent reads are served from cache.
- Best suited for read-heavy workloads where write data doesn't need to be cached.

Example: Product listings displayed to users are cached for fast retrieval. Order placement requests go directly to the database since they are write operations not frequently read by the user.

2. Write-Through Cache

A strategy where data is **written to the cache first**, and then synchronously written to the database.

How it works:

Operation	Flow
Write	Request → Cache (updated) → Database (updated)
Read	Request → Cache → Response

Key Points:

- Cache always holds the **latest and most consistent** data.
- Write latency is slightly higher since both cache and database are updated on every write.
- Ensures strong consistency between cache and database.

Example: Stock market values that must always reflect the latest price - every update goes to cache first, then to the database.

3. Write-Around Cache

A strategy where write operations go **directly to the database**, bypassing the cache entirely. Reads are served from the cache.

How it works:

Operation	Flow
Write	Request → Database directly (cache is not updated)
Read (Cache Hit)	Request → Cache → Response
Read (Cache Miss)	Request → Cache (miss) → Database → Store in Cache → Response from Cache

Key Points:

- Cache is not polluted with data that may not be read immediately after writing.
- Useful when written data is not frequently accessed right after the write.
- Risk of stale data in cache if the same data is updated in the database.

4. Write-Back Cache (Write-Behind)

A strategy where data is written **only to the cache** at the time of the request. The database is updated later through an **asynchronous process**.

How it works:

Operation	Flow
Write	Request → Cache (updated immediately) → Database (updated asynchronously)
Read (Cache Hit)	Request → Cache → Response
Read (Cache Miss)	Request → Cache (miss) → Database → Store in Cache → Response from Cache

Key Points:

- Prioritizes **speed over consistency**.
- Best suited for systems with **high write traffic**.
- Risk of data loss if the cache fails before the asynchronous write to the database completes.

Example: In a food delivery application, order tracking states change frequently. All state updates are written to the cache for instant retrieval by users, while the database is updated asynchronously in the background.

Strategy Comparison

Summary

- Caching strategies define how data is read from and written to the cache and database, balancing performance, consistency, and scalability.
- Read-Through Cache is optimized for read-heavy workloads, loading data into the cache only when a cache miss occurs.
- Write-Through Cache provides strong consistency by updating both the cache and database synchronously, ensuring the cache always contains the latest data.
- Write-Around Cache writes data directly to the database, preventing unnecessary cache usage for data that is unlikely to be read soon after being written.
- Write-Back Cache (Write-Behind) offers the highest write performance by updating the cache immediately and persisting data to the database asynchronously, trading immediate consistency for speed.

Cache Eviction Policies

Cache has limited memory. When the cache reaches its capacity, existing entries must be removed to make space for new data. **Eviction policies** define the rules for deciding which data to remove from the cache.

1. LRU (Least Recently Used)

Evicts the data that has **not been accessed for the longest time**.

Logic: If data hasn't been used recently, it is unlikely to be needed in the near future.

Example: After a new version of a mobile phone is released, the older version's product data is accessed less recently and gets evicted from the cache.

2. MRU (Most Recently Used)

Evicts the data that was **accessed most recently**.

Logic: If data was just used, there is a low probability the user will need it again immediately.

Example: A discount coupon that a user just applied - once used, the user won't need it again, so it is evicted first.

3. LFU (Least Frequently Used)

Evicts the data that has been **accessed the fewest number of times** overall.

Logic: Data with the lowest access count is the least valuable to keep in cache.

Example: On an e-commerce platform, the "Plants" category is searched far less frequently compared to "Gadgets" or "Clothes", so it gets evicted first when space is needed.

4. FIFO (First In First Out)

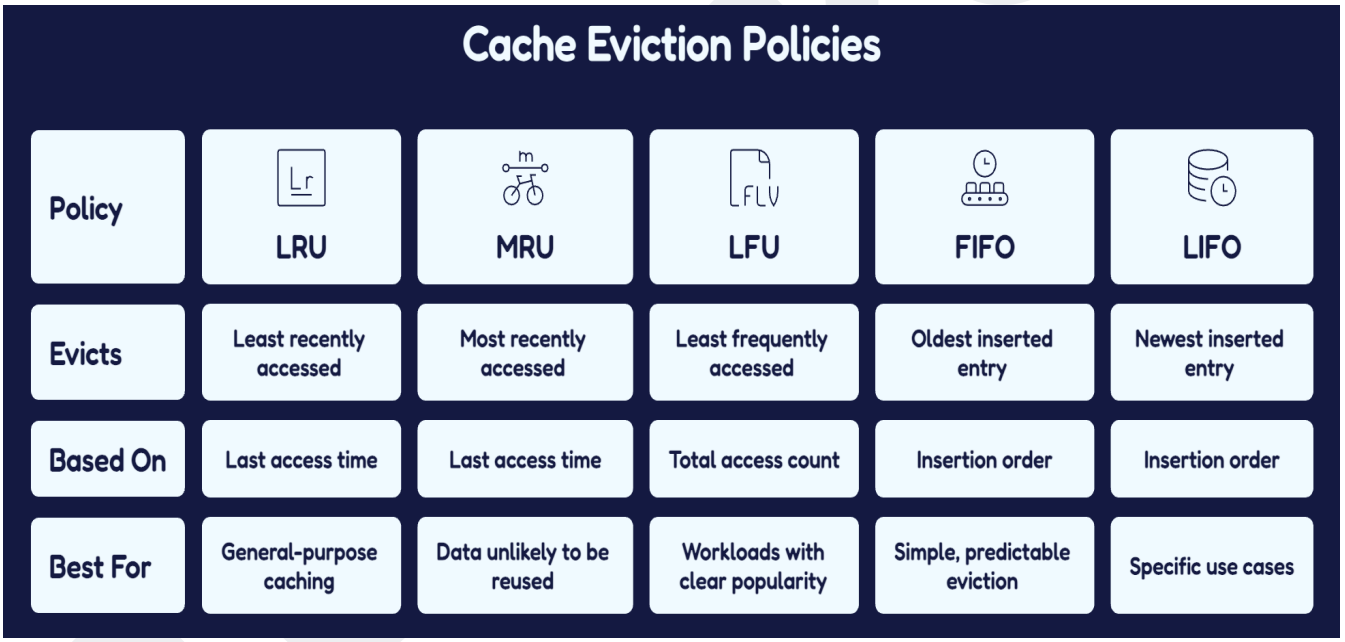
Evicts the data that was **inserted into the cache first**, regardless of how often or recently it was accessed.

Logic: Oldest entries are removed first, similar to a queue structure.

5. LIFO (Last In First Out)

Evicts the data that was **inserted into the cache most recently**, regardless of access patterns.

Logic: Newest entries are removed first, similar to a stack structure.



Summary

- Cache eviction policies determine which data should be removed when the cache reaches its storage limit, ensuring space is available for new entries.
- LRU (Least Recently Used) removes data that has not been accessed for the longest time and is one of the most commonly used eviction policies.

- LFU (Least Frequently Used) removes data with the lowest access frequency, making it suitable for workloads where some items are consistently more popular than others.
- FIFO (First In First Out) and LIFO (Last In First Out) evict entries based on insertion order rather than access patterns, making them simple but less adaptive.
- The choice of eviction policy depends on application access patterns, performance requirements, and the type of data being cached.

Load Balancer

A load balancer is a component that **distributes incoming traffic across multiple servers** to ensure no single server is overwhelmed. It sits between the client and the server pool, routing each request to an appropriate server.

Core Functions of a Load Balancer

Function	Description
Choosing the Server	Decides which server should handle the incoming request based on an algorithm
Health Checks	Continuously monitors servers to ensure they are healthy and available to handle requests

Load Balancing Algorithms

1. Round Robin

Distributes requests sequentially - first request to Server 1, second to Server 2, third to Server 3, and so on in a circular manner.

Key Points:

- Each server handles an approximately equal number of requests.
- Only a single counter is needed to track which server receives the next request.
- No server state management required.

Disadvantage: Does not account for differences in server configurations (RAM, CPU, storage). A low-capacity server receives the same load as a high-capacity server, leading to unfair distribution.

2. Weighted Round Robin

An enhancement of Round Robin where each server is assigned a **weight based on its configuration/capacity**. Servers with higher capacity receive more requests.

Key Points:

- Higher configuration → Higher weight → More requests routed to that server.
- Addresses the limitation of standard Round Robin by considering server capacity.

3. Geo-Based Algorithm

Routes requests to the server that is **geographically closest** to the user making the request.

Key Points:

- Load balancer maintains information about request origin and server locations.
- Reduces latency by minimizing physical distance between client and server.
- Database replication and partitioning must be maintained across geographic regions.
- Data should be replicated to nearby servers as failover in case of server crashes.

Disadvantage: VPNs and proxies can mislead the traffic by masking the user's actual location.

4. Least Connection Algorithm

Routes the request to the server that currently has the **fewest active connections**.

Key Points:

- Only the number of active connections is considered.
- Connection counts need to be continuously updated.

Disadvantage: Does not check server metrics like response time or server configuration. A server may have few connections but still be under heavy processing load.

5. Least Time Algorithm

Routes the request to the server with the **lowest average response time**.

Key Points:

- Accounts for server configuration - higher-capacity servers naturally have lower response times.
- Load balancer continuously calculates the average response time for each server.
- Additional information maintained: number of requests per server and average response time.
- Ideal for latency-sensitive applications (e.g., stock trading platforms).

6. IP Hash Algorithm

The client's IP address is processed through a **hash function** to generate a number, which determines the target server.

Key Points:

- IP ranges are mapped to specific servers (e.g., IPs hashing to 1–10 → Server 1, 11–20 → Server 2).
- Ensures the same client always reaches the same server (session persistence).
- The server must be healthy to maintain proper sessions; if it crashes, data must be replicated to another server.

Disadvantage:

- Scaling is challenging - adding a new server requires rehashing and reassignment of all values.
- Not ideal for microservice architectures where session continuity is critical.

Algorithm Comparison

Algorithm	Basis of Decision	State Maintained	Best For
Round Robin	Sequential order	Counter only	Equal-capacity servers
Weighted Round Robin	Server capacity	Weights	Heterogeneous server configurations
Geo-Based	Geographic proximity	Location data	Globally distributed users
Least Connection	Active connections	Connection count	Variable request durations
Least Time	Response time	Avg response time	Latency-sensitive applications
IP Hash	Client IP hash	Hash mappings	Session persistence

Hybrid Approach

In real-world applications, a **hybrid approach** is used where the load balancer combines multiple algorithms. For example, IP Hash can be combined with Round Robin to maintain session persistence while distributing load evenly across servers.

Summary

- A Load Balancer distributes incoming requests across multiple servers, improving system scalability, availability, fault tolerance, and overall performance.
- The two primary responsibilities of a load balancer are **server selection** (using routing algorithms) and **health monitoring** (ensuring traffic is sent only to healthy servers).
- Different algorithms optimize load distribution based on different factors, such as request order (Round Robin), server capacity (Weighted Round Robin), geographic

location (Geo-Based), active connections (Least Connection), response time (Least Time), or client identity (IP Hash).

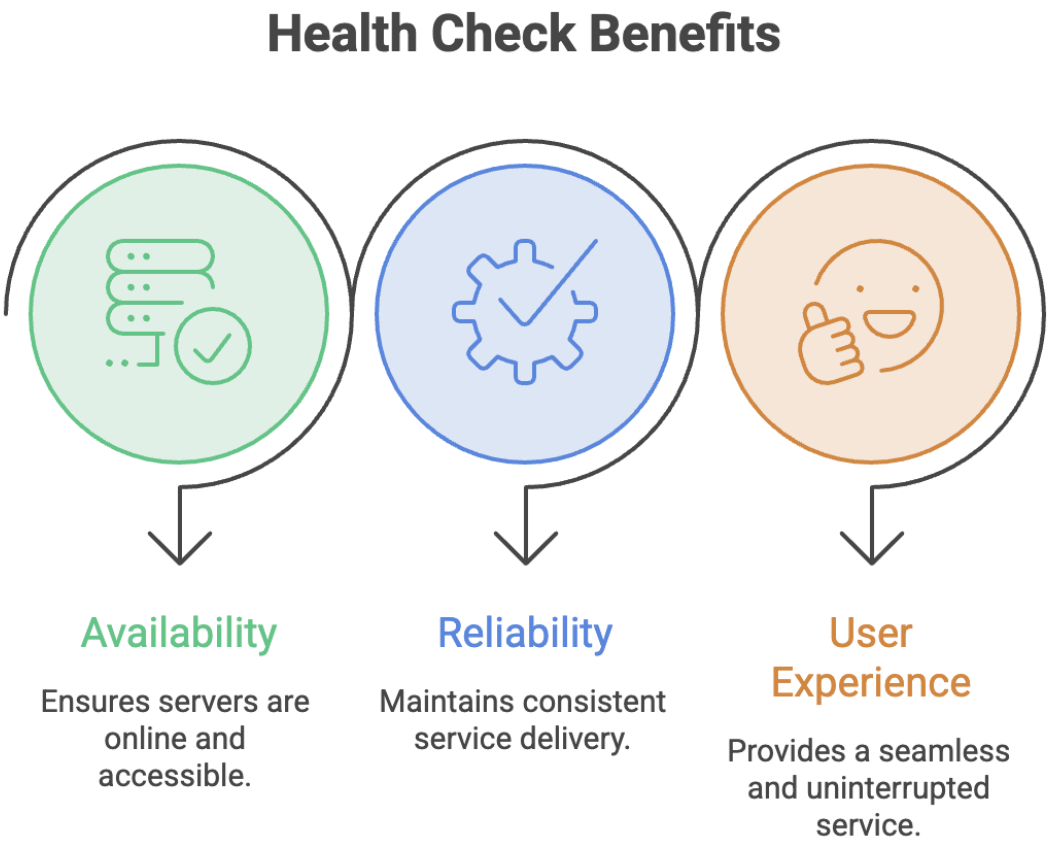
- **Weighted Round Robin**, **Least Connection**, and **Least Time** provide more intelligent traffic distribution than basic Round Robin by considering server capacity, workload, or performance metrics.
- **Geo-Based** and **IP Hash** algorithms are useful for reducing latency and maintaining session persistence, respectively, but introduce additional challenges such as location accuracy and scaling complexity.
- Modern distributed systems typically use a **hybrid load-balancing approach**, combining multiple algorithms to achieve optimal performance, reliability, and user experience.

Load Balancer Health Checks

Health checks enable a load balancer to continuously monitor the availability and performance of backend servers. Based on these checks, traffic is routed only to healthy servers, ensuring high availability, reliability, and better user experience.

Why Health Checks?

- Detect unhealthy or degraded servers before routing requests.
- Prevent failed or slow servers from receiving new traffic.
- Provide metrics for scaling decisions (scale up or scale down).
- Improve application availability and reliability.



Types of Health Checks

1. Passive Health Check

The load balancer monitors live production traffic without sending additional requests.

- No extra load on servers.
- Observes response time, failures, and error rates from real requests.
- Useful for identifying issues under actual workload.

2. Active Health Check

The load balancer periodically sends probe requests to verify server health.

- Dedicated health-check requests are sent at regular intervals.
- Provides faster and more accurate health detection.
- Introduces a small amount of additional traffic.

Health Check Parameters

Parameter	Description	Example
Interval	Frequency of health checks.	Every 5 seconds.
Timeout	Maximum wait time for a response.	Expected 50 ms–5 s; 10 s indicates failure.
Threshold	Consecutive successes/failures before changing server status.	20 failures → Unhealthy; 100 successes → Healthy.

Health Check Workflow

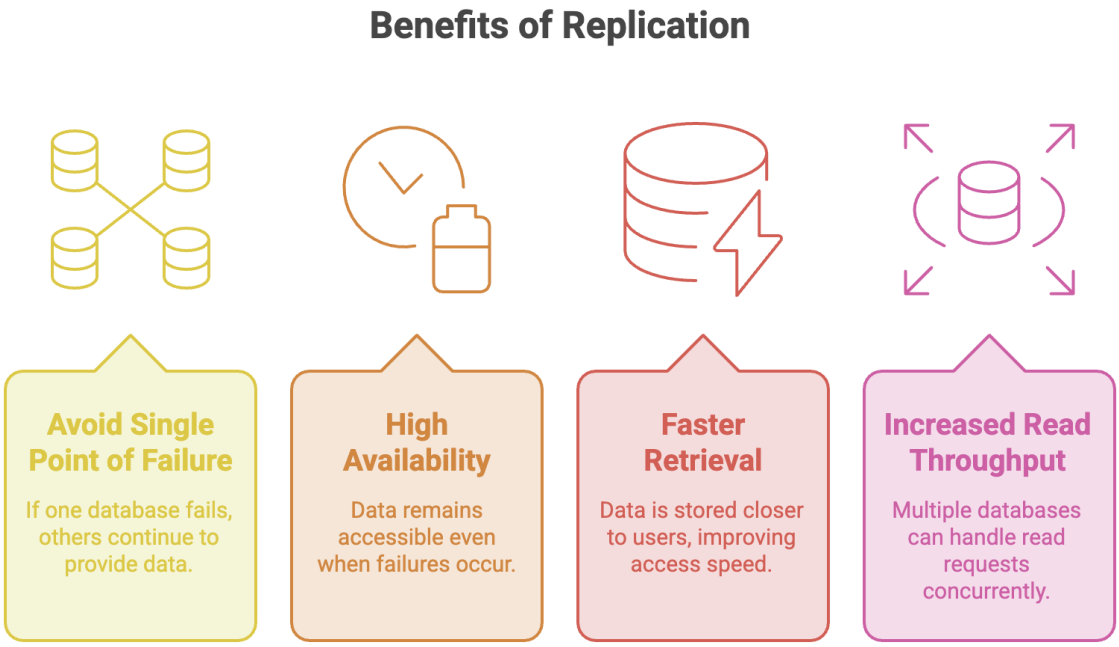
1. At every configured interval, the load balancer checks each server.
2. If the server fails to respond within the timeout, it is counted as a failed check.
3. Once the unhealthy threshold is reached, the server is marked unhealthy and removed from the load balancing pool.
4. When the server successfully responds enough times to meet the healthy threshold, it is added back into the pool.

Summary

- Health checks enable load balancers to continuously monitor server availability and performance, ensuring that traffic is routed only to healthy servers while also supporting scaling decisions.
- Passive health checks analyze real user traffic and server responses without generating additional requests, making them lightweight and efficient.
- Active health checks send periodic probe requests to servers, providing more accurate and real-time health monitoring at the cost of a small amount of additional overhead.
- Health check parameters such as interval, timeout, and threshold determine how server health is evaluated, allowing unhealthy servers to be removed from the routing pool and automatically reintroduced once they recover.

Replication

Replication is the process of **copying and maintaining data across multiple databases or nodes**. It ensures that the same data exists in more than one location.



1. Single Leader Replication

One database is designated as the **Leader** (primary), and all other databases are **Followers** (replicas). All write operations go to the leader, which then replicates the data to the followers.

Flow: User → Server → Leader → Followers

Asynchronous Replication

The leader updates the data and **immediately sends a success response** to the user without waiting for followers to confirm.

How it works:

- The leader completes the write and responds to the user.
- Followers update in the background.
- If followers fail, they sync again once recovered.

Advantage: Faster response, no resource blocking.

Disadvantage: Consistency is not guaranteed; data can be temporarily inconsistent or corrupted if followers lag behind.

Synchronous Replication

The leader waits for **confirmation from all follower nodes** before sending a success response to the user.

How it works:

- The leader sends a write request to all followers.
- Leader waits until all followers return an OK response.
- Only then does it confirm success to the user.

Aspect	Advantage	Disadvantage
Data consistency	Followers are always up-to-date	Longer response time
Failover	Any follower can become leader (all have latest data)	Blocks resources while waiting for confirmations

In practice: Asynchronous replication is preferred for most use cases due to faster responses and no resource blocking.

Adding a New Follower Node

1. Create a **snapshot** of the data from the leader (or an existing follower).
2. Load the snapshot into the new node.
3. Establish a connection between the leader and the new node for ongoing replication.

Leader Failure - Electing a New Leader

When the leader fails, a new leader must be elected:

1. **Identify the most up-to-date follower** - check timestamps to determine which follower has the latest data.
2. **Inform all other nodes** about the newly elected leader.

Tools like FSImage (full metadata snapshot) and EditLogs (incremental change logs) are used for data replication and recovery.

2. Multi-Leader Replication

Multiple data centers each have their own **leader and follower nodes**. Leaders communicate with each other across data centers in addition to their own followers.

Key Points:

- Each leader communicates with its followers and with leaders of other data centers.
- Improves performance and fault tolerance, if one leader fails, others continue to serve.
- Enables parallel writes from multiple leaders.

Conflict Problem

When two leaders write to the same data simultaneously, conflicts arise.

Example: User1 renames a file to "A" through Leader1, User2 renames the same file to "B" through Leader2. Both receive success responses. When leaders sync, a conflict is detected.

Conflict Resolution Strategies

Strategy	Description
Last Write Wins	The most recent written (based on timestamp) becomes the final value
Leader Priority	Leaders are assigned IDs, the value from the higher-priority leader is kept
User Resolution	The conflict is surfaced to the user, who decides which value to keep

3. Leaderless Replication

There is **no designated leader**. A write is considered successful only when **all (or a majority of) nodes** return an OK response.

Quorum

A quorum defines the **minimum number of nodes** that must respond for an operation to be considered successful.

Operation	Quorum Required
Read	Greater than n/2 nodes must respond
Write	Greater than n/2 nodes must confirm

Where **n** = total number of nodes.

This ensures that at least one node in both read and write operations has the latest data, maintaining consistency across the system.

Summary

- **Replication** is the process of maintaining multiple copies of data across different nodes or databases to improve availability, fault tolerance, read performance, and disaster recovery.
- **Single Leader Replication** uses one primary node to handle all write operations, while follower nodes replicate the data; it commonly employs asynchronous or synchronous replication depending on consistency requirements.
- **Multi-Leader Replication** allows multiple leaders to accept write requests across different locations, improving availability and write scalability but introducing potential data conflicts that must be resolved.
- **Leaderless Replication** removes the concept of a primary node and relies on **quorum-based reads and writes** to maintain data consistency across distributed nodes.

- The choice of replication strategy involves balancing **consistency, availability, latency, fault tolerance, and scalability** based on the application's requirements.

Partition

Partitioning is the process of **splitting a large dataset into smaller, manageable chunks** called partitions, which are then distributed across multiple nodes. Each partition holds a subset of the data, and combining all partitions gives the complete dataset.

Why Partitioning?

- Replicating an entire massive dataset across nodes is expensive and slow.
- As data grows, search and query performance degrades.
- Partitioning enables parallel processing and faster data retrieval.

Key Rule: Data should be **evenly distributed** across all partitions to avoid imbalance.

Hotspot

A **hotspot** is a partition that receives significantly more requests compared to others, leading to:

- Overutilization of the hotspot partition.
- Underutilization of other partitions.
- Potential node failure due to excessive load.

Hotspots must always be avoided to maintain system stability.

Partitioning Strategies

1. Partition by Key (Range-Based)

Data is distributed based on key ranges, for example, keys A–M go to Partition 1, and keys N–Z go to Partition 2.

Problem: If traffic is unevenly distributed across key ranges, one partition becomes a hotspot.

Example: Partition 1 stores India's data, Partition 2 stores US data. If the application is more popular in the US, Partition 2 becomes a hotspot while Partition 1 remains underutilized.

2. Partition by Hash of Key

A hash function converts each key into a unique number, and data is distributed to partitions based on the hash value. This provides more uniform distribution than range-based partitioning.

Handling Hotspots:

- Attach timestamps with requests.
- Monitor partition loads and **rearrange data** if one partition receives disproportionately more traffic.

3. Partition by Local Secondary Index

Each partition maintains its own **secondary index** for faster lookups on non-primary key attributes.

Example (Car Database):

Partition 1	Partition 2
Blue: 209, 305	Blue: 509, 609
Red: 350, 359	Red: 610, 690

- Query for "**Red cars**" → check secondary index in each partition to find relevant records.
- Query for "**Silver cars**" → secondary index confirms no records exist without scanning actual data.

Used by: Databases like Cassandra, Elasticsearch

Disadvantage: Every partition must be queried (scatter-gather), even if a partition has no relevant data , since each partition maintains its own local index independently.

4. Partition by Global Secondary Index

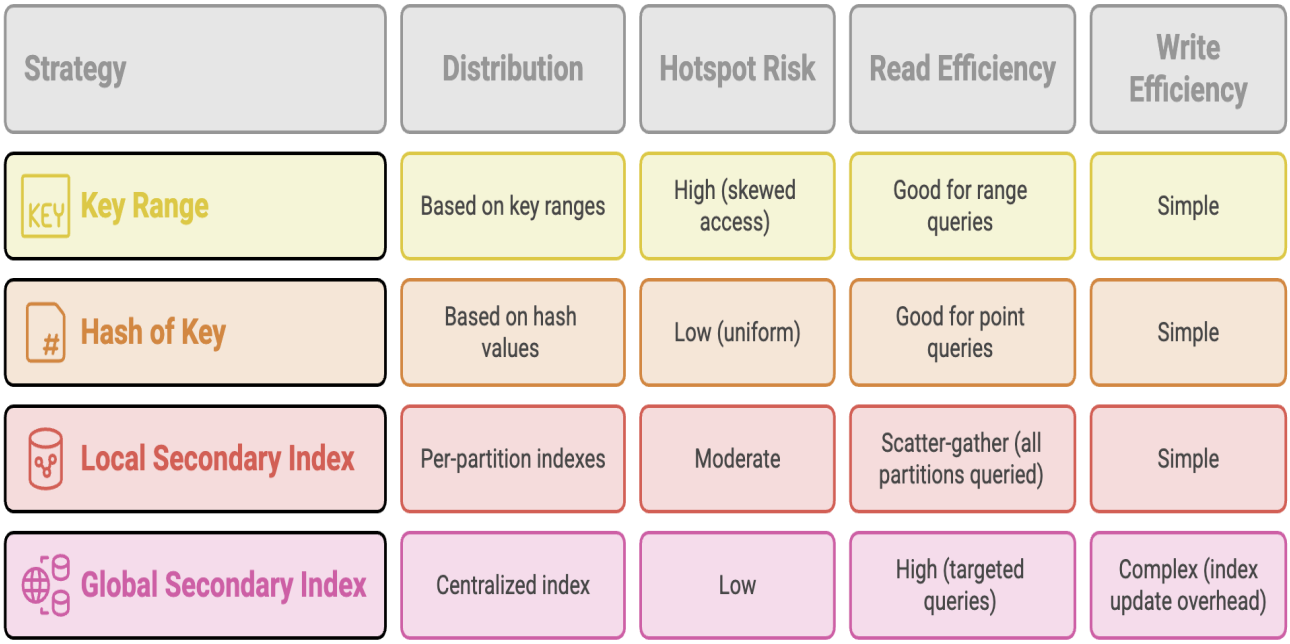
A **single global index** is maintained in a dedicated partition or data center that tracks which partitions contain specific values.

Example:

- ➢ **Global Index:** Blue cars → P1, P2 | Red cars → P1, P2 | Silver cars → P3
- ➢ Query for "Blue cars" → Global index directs the request only to P1 and P2, skipping all other partitions.

Aspect	**Advantage**	**Disadvantage**
Read	Efficient, only relevant partitions are queried	NA
Write	NA	Tedious, global index must be updated on every write across partitions

Strategy Comparison



Summary

- Partitioning is the process of dividing a large dataset into smaller partitions and distributing them across multiple nodes to improve scalability, performance, and parallel processing capabilities.
- The primary goal of partitioning is to ensure **even data and traffic distribution**, preventing hotspots where a single partition becomes overloaded while others remain underutilized.
- **Range-Based Partitioning** is simple and efficient for range queries but can create hotspots when data access patterns are uneven.
- **Hash-Based Partitioning** provides more uniform data distribution and reduces hotspot risks, making it suitable for high-scale distributed systems.
- **Local Secondary Indexes** simplify write operations but require querying multiple partitions, whereas **Global Secondary Indexes** enable targeted and efficient reads at the cost of additional index maintenance overhead.

CAP Theorem

The CAP Theorem states that in a distributed system, it is **impossible to simultaneously guarantee all three** of the following properties. You can only achieve **two out of three** at any given time.

The Three Properties

Property	Full Form	Meaning
C	Consistency	Every user always receives the **latest and most up-to-date** information
A	Availability	The system **always responds** to every request (no downtime)
P	Partition Tolerance	The system continues to operate even when data is **divided across multiple nodes** and network failures occur between them

Why Can't We Have All Three?

When data is partitioned across multiple nodes, syncing data between them takes time. During that sync window:

- If we **show the data** (even if outdated) → Availability is maintained, but **Consistency is compromised**.
- If we **block the request** until sync completes → Consistency is maintained, but **Availability is compromised**.

This forces a trade-off.

CAP Combinations

CA (Consistency + Availability)

- Data is both consistent and always available.

- **Trade-off:** No partition tolerance - works only with a **single database**.
- If multiple nodes exist, we cannot guarantee both consistency and availability simultaneously during sync delays.

Use case: Small-scale applications with a single database.

CP (Consistency + Partition Tolerance)

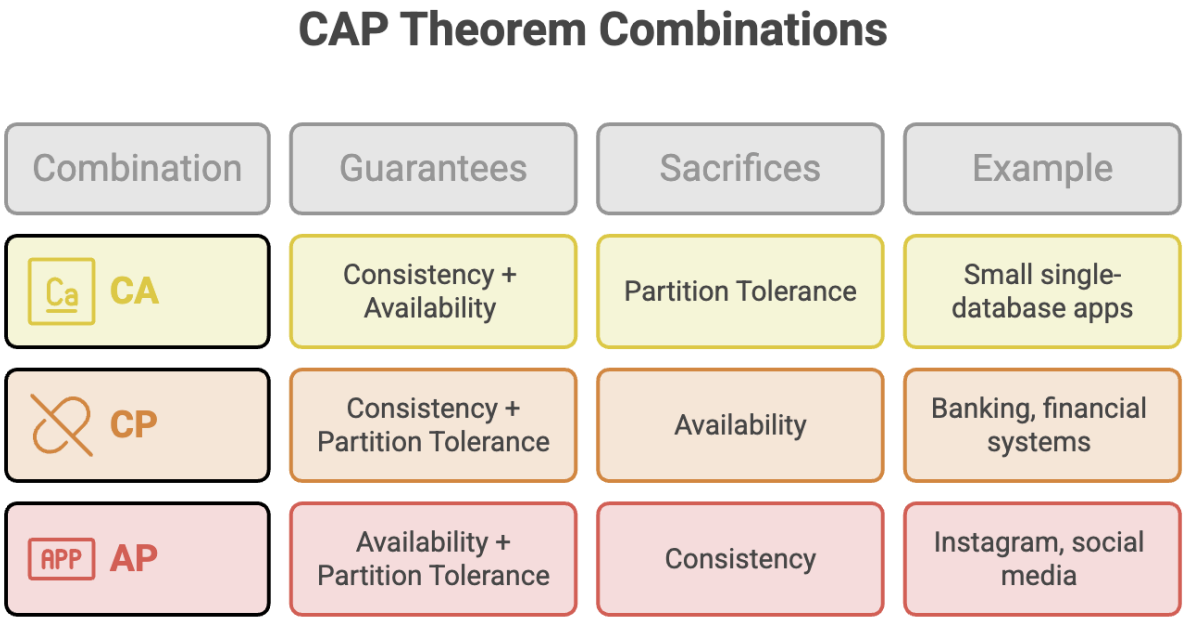
- Data is consistent across partitions, and the system tolerates network splits.
- **Trade-off:** Availability is sacrificed - requests may be blocked or rejected until data is fully synced.

Use case: Banking and financial applications where showing incorrect data is unacceptable.

AP (Availability + Partition Tolerance)

- The system is always available and tolerates network partitions.
- **Trade-off:** Consistency is sacrificed - stale or outdated data may be shown during sync.

Use case: Social media platforms like Instagram where availability matters more than showing the absolute latest data.



Summary

- **CAP Theorem** states that a distributed system can guarantee only **two out of three properties** at a time: **Consistency (C), Availability (A), and Partition Tolerance (P)**.
- During network partitions or communication failures, systems must choose between maintaining Consistency (always returning the latest data) or Availability (always responding to requests).
- **CP systems** prioritize data accuracy and consistency over availability, making them suitable for critical applications such as banking and financial systems, while **AP systems** prioritize availability and fault tolerance, even if some data becomes temporarily stale.
- The CAP Theorem highlights the fundamental trade-offs in distributed system design, requiring architects to select the most appropriate balance based on business and application requirements.

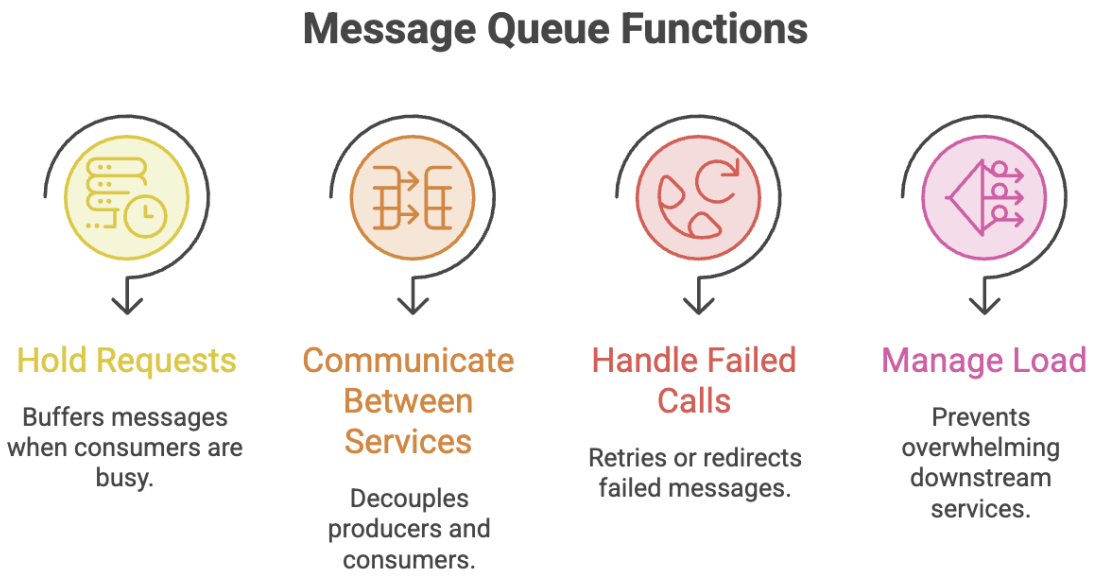
Message Queues

A message queue is a **buffer between two applications or services** that enables asynchronous communication. It temporarily holds requests and delivers them to the appropriate service for processing.

Key Terminology

Term	Also Called	Role
Producer	Publisher	Service that puts/publishes messages into the queue
Consumer	Subscriber	Service that takes/subscribes to messages from the queue

Example: An e-commerce app publishes email requests to a message queue. The queue holds the requests and forwards them to the email service, handling any failures along the way.



Synchronous vs Asynchronous Processes

Type	Description	Example
Synchronous	Request requires an immediate response for the system to continue working	Updating inventory after a purchase
Asynchronous	Request is made but response is not needed immediately (or not at all)	Sending confirmation email, triggering delivery

When to use Sync:

- Financial applications
- Security-critical operations
- Tasks with dependencies on one another

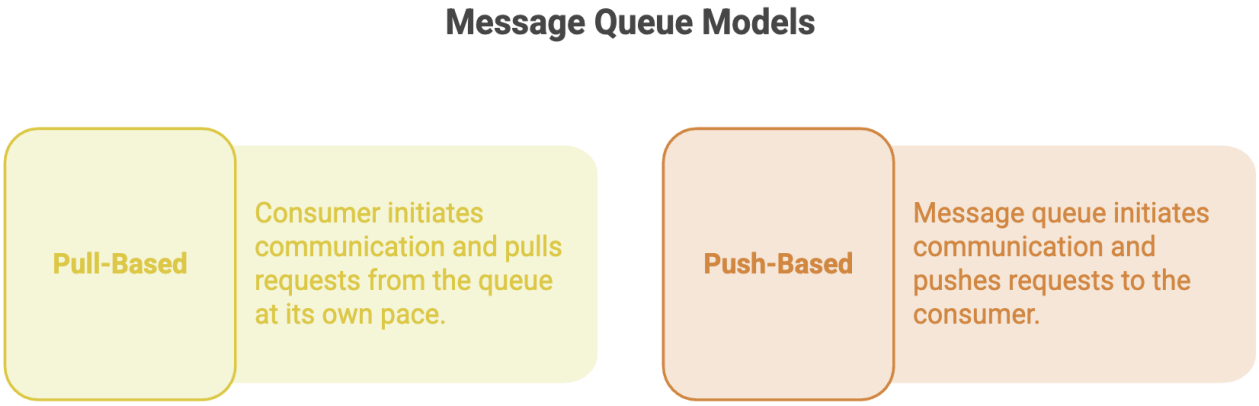
When to use Async:

- Tasks that don't critically impact the application flow
- Broadcasting, emails, SMS, OTPs, messaging

Processing Approaches (FIFO-Based)

Message queues follow the **FIFO (First In First Out)** principle by default, with the following variations:

Approach	Behavior
Strict Order	If a request fails, the queue halts, it does not process subsequent requests until the failed one is resolved
Unordered Queue	If a request fails, the queue skips it and moves to the next request
Priority Queue	Each request has a priority, higher-priority requests are processed first



Pub-Sub Model

The **Publisher-Subscriber** model involves multiple publishers, a message queue, and multiple subscribers.

How it works?

- Publishers send messages to the queue.
- Queue holds messages and distributes them to subscribers.
- Queue monitors subscriber health and redirects requests if a subscriber is down.
- Queue handles failures and retries.

Key Factors in Pub-Sub

Factor	Description
Message Order	Maintained through priority queues and asynchronous calls
Message Consumption	Consumption by subscribers can be random
Poison Message	Messages that repeatedly fail at the consumer side (bad requests, invalid data, subscriber issues)

Duplicacy	Once a message is processed by one subscriber, it must be deleted from the queue to prevent reprocessing by another subscriber

Dead Letter Queue (DLQ)

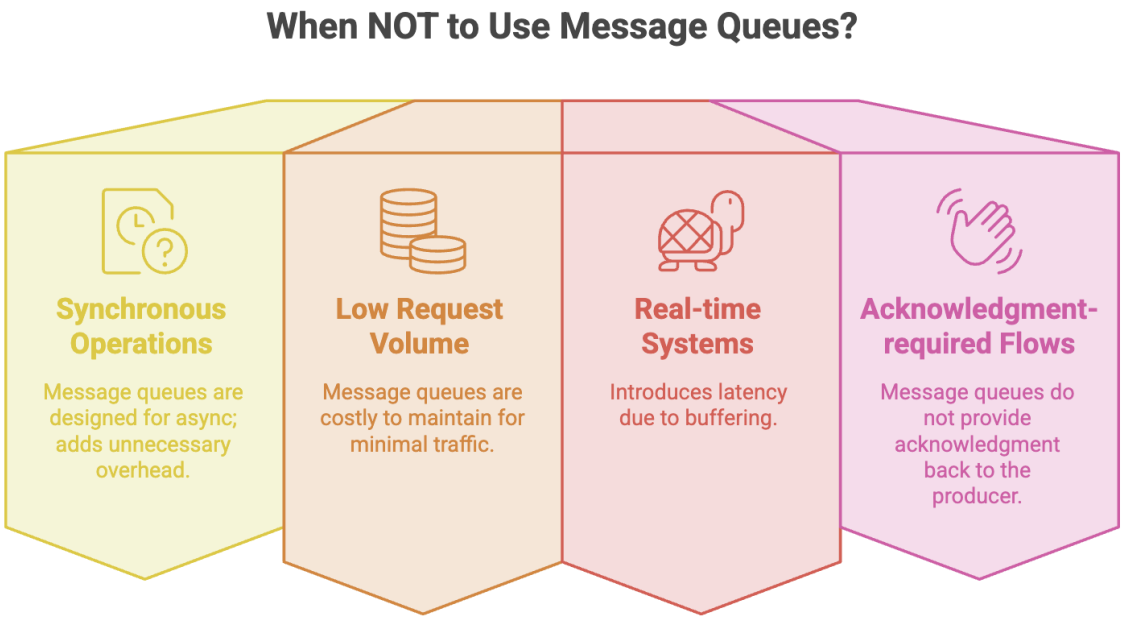
A **Dead Letter Queue** is a secondary queue maintained by the message queue to store messages that have **failed processing** after multiple attempts.

Purpose:

- Keeps track of all failed requests.
- Allows later inspection, debugging, or retry of failed messages.
- Prevents poison messages from blocking the main queue.

Use Cases

- Asynchronous operations (emails, notifications, reports)
- Decoupling services (analytics pipelines)
- Load balancing between services
- Deferred systems (scheduled processes, periodic report generation)



Summary

- **Message Queues** enable asynchronous communication between services by acting as a buffer that stores and delivers messages, improving scalability, reliability, and system decoupling.
- The core components of a message queue are the **Producer (Publisher)**, which sends messages, and the **Consumer (Subscriber)**, which processes them independently.
- Message queues help manage traffic spikes, handle failures through retries, distribute workloads efficiently, and prevent downstream services from being overwhelmed.
- Advanced messaging patterns such as **Pub-Sub**, **Priority Queues**, and **Dead Letter Queues (DLQ)** support scalable event-driven architectures, message prioritization, and failed message handling.
- Message queues are ideal for asynchronous tasks such as notifications, emails, analytics, and background processing but are generally unsuitable for synchronous, low-latency, or acknowledgment-dependent workflows.

Faults & Errors

A fault or error occurs when a system **does not perform as expected** or deviates from its intended behavior. Understanding fault categories helps in designing resilient systems.

Categories of Faults

1. Hardware Failures

Failures related to **physical infrastructure**, often random and unpredictable.

Causes:

- Disk full or disk crash
- Server down
- Network cable damage
- Power supply failure
- Overheating
- Database compromise

Characteristics: Random, difficult to predict, mitigated through redundancy and replication.

2. Software Failures

Failures occurring at the **application or software level** are generally preventable and fixable.

Causes:

- Bad or unoptimized code
- Unhandled exceptions
- Missed edge cases
- Configuration issues
- Deployment failures

- Merge conflicts
- Performance bottlenecks

Mitigation: Code reviews, testing, proper exception handling, and CI/CD best practices.

3. Human Errors

Failures caused by **human actions**, the most unpredictable and unreliable category.

Characteristics: Cannot be fully eliminated; mitigated through automation, access controls, proper training, and well-designed interfaces that minimize the scope for mistakes.

Category	Nature	Predictability	Mitigation
Hardware	Physical infrastructure	Random	Redundancy, replication, monitoring
Software	Application level	Identifiable	Code reviews, testing, proper handling
Human	Human actions	Unpredictable	Automation, access controls, training

Summary

- **Faults and errors** occur when a system deviates from its expected behavior, potentially impacting reliability, availability, and performance.
- **Hardware failures** are infrastructure-related issues such as server crashes, disk failures, or network outages and are typically mitigated through redundancy, replication, and monitoring.

- **Software failures** arise from defects in code, configuration, deployment, or application logic and can be reduced through rigorous testing, code reviews, and robust development practices.
- **Human errors** result from incorrect actions or decisions and are best minimized through automation, access controls, standardized processes, and continuous training.

Monitoring and Observability

Monitoring and observability involve continuously tracking the health, performance, and behavior of a system to ensure all components are functioning correctly. It answers critical questions:

- Are there any errors in the system?
- Is every component working as expected?
- What is the overall health of the system?
- Are all edge cases and scenarios being handled?
- Are proper alerts triggered when a component goes down?

Monitoring vs Observability

Although often used together, Monitoring and Observability are not identical concepts.

Monitoring

Monitoring focuses on collecting and tracking predefined metrics, logs, and alerts to determine whether a system is operating within expected limits.

Examples:

- CPU utilization
- Memory consumption
- API response times
- Error rates
- Server availability

Monitoring answers the question: "Is **something wrong?"**

Observability

Observability is the ability to understand the internal state of a system by analyzing its outputs, such as metrics, logs, and traces.

Observability answers the question: **"Why is something wrong?"**

In complex distributed systems, observability enables engineers to investigate failures, identify bottlenecks, and trace requests across multiple services.

API Monitoring

API Monitoring focuses on tracking the performance, availability, and reliability of application endpoints.

Since APIs serve as the primary communication layer between clients and services, monitoring them is critical for maintaining system health.

1. Throughput

Throughput measures the number of requests processed by the system within a specific period of time.

Importance

- Indicates the load handled by the application.
- Helps determine system capacity.
- Assists in scaling decisions.

Example

If a service processes 10,000 requests per minute, its throughput is 10,000 RPM.

2. Error Monitoring

Error monitoring tracks unsuccessful requests and response codes.

Common HTTP status code categories include:

2xx	**Successful requests**
3xx	Redirections
4xx	Client-side errors
5xx	Server-side errors

Importance

- Detects application failures.
- Identifies deployment issues.
- Helps maintain service reliability.

Alerts should be triggered when error rates exceed predefined thresholds.

3. Health Checks

Health checks continuously verify whether an application or service is operational.

Purpose

- Confirms service availability.
- Detects failures quickly.
- Supports automated recovery and load balancing decisions.

A healthy service typically returns a successful response such as HTTP 200.

4. Latency Monitoring

Latency measures the time required for a system to respond to a request.

Importance

- Directly impacts user experience.
- Helps identify performance bottlenecks.
- Assists in capacity planning.

Lower latency generally indicates better system performance.

Understanding Latency Percentiles

Average latency alone does not provide a complete picture because a small number of slow requests can significantly impact users.

Percentiles provide a more accurate measurement.

P90 (90th Percentile)

A P90 latency of 200 ms means:

- 90% of requests complete within 200 ms.
- 10% of requests take longer than 200 ms.

P95 (95th Percentile)

- 95% of requests complete within the specified time.
- Only 5% are slower.

P99 (99th Percentile)

- 99% of requests complete within the specified time.
- Represents worst-case user experience for most users.

Percentile-based monitoring is widely used in production systems because it highlights performance issues that averages often hide.

System Monitoring (Infrastructure Monitoring)

System monitoring focuses on the health and utilization of the underlying infrastructure.

Monitoring infrastructure resources helps prevent outages and performance degradation.

1. CPU Usage

CPU monitoring tracks processor utilization across servers.

Importance

- Identifies overloaded systems.
- Helps determine when scaling is required.
- Prevents performance degradation.

Alerts are commonly configured when CPU usage exceeds a threshold such as 80%.

2. Memory Usage

Memory monitoring tracks RAM consumption.

Importance

- Prevents out-of-memory failures.
- Detects memory leaks.
- Helps optimize resource allocation.

High memory usage over extended periods may indicate application inefficiencies.

3. Disk I/O

Disk Input/Output monitoring measures read and write operations performed on storage devices.

Importance

- Detects storage bottlenecks.
- Identifies slow database operations.
- Helps maintain application performance.

High disk utilization can significantly increase response times.

4. Network I/O

Network monitoring measures incoming and outgoing network traffic.

Importance

- Detects traffic spikes.
- Identifies network bottlenecks.
- Helps troubleshoot communication issues between services.

Abnormal traffic patterns may indicate failures, attacks, or misconfigurations.

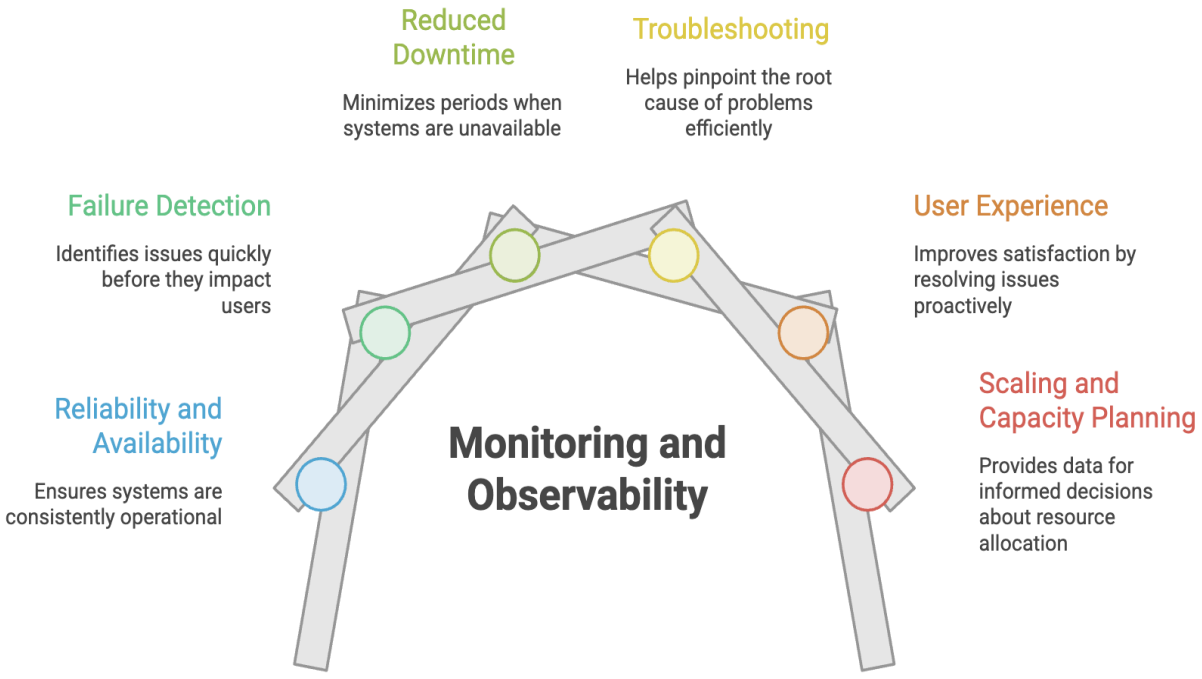
Alerting and Incident Detection

Monitoring is effective only when combined with proper alerting mechanisms.

Alerts should be configured for critical metrics such as:

- High CPU utilization
- High memory consumption
- Increased error rates
- Service downtime
- Excessive latency
- Network anomalies

Well-designed alerts allow teams to respond quickly before users experience service disruptions.



Summary

- Monitoring tracks system health, performance, and availability using predefined metrics and alerts.

- Observability helps engineers understand and diagnose the root causes of issues using metrics, logs, and traces.
- API monitoring focuses on throughput, error rates, health checks, and latency.
- Infrastructure monitoring focuses on CPU, memory, disk I/O, and network I/O.
- Percentile-based latency measurements (P90, P95, P99) provide more meaningful performance insights than averages.
- Effective monitoring combined with proactive alerting enables rapid issue detection, faster recovery, and improved system reliability.