SYSTEM DESIGN

PART I – FOUNDATIONS OF SYSTEM DESIGN

1. **Introduction to System Design**
   1.1 Why Study System Design
   1.2 Real-World vs College Projects
   1.3 Functional vs Non-Functional Requirements
2. **Basic System Architecture**
   2.1 Client and Server
   2.2 Backend and Database Interaction
   2.3 Deployment and Cloud Overview
   2.4 DNS and Domain Resolution
3. **Performance Fundamentals**
   3.1 Latency
   3.2 Throughput
   3.3 Round Trip Time (RTT)

PART II – SCALING AND AVAILABILITY

4. **Scaling Concepts**
   4.1 Vertical Scaling (Scale Up / Down)
   4.2 Horizontal Scaling (Scale Out / In)
   4.3 When to Scale
5. **Load Balancing**
   5.1 Why Load Balancers are Needed
   5.2 Load Balancer Architecture
   5.3 Load Balancer Algorithms
   - Round Robin
   - Weighted Round Robin
   - Least Connections
   - Hash-Based Routing
6. **Auto Scaling**
   6.1 Dynamic Traffic Handling
   6.2 Auto Scaling Groups
   6.3 Cost Optimization

PART III – SYSTEM CAPACITY ESTIMATION

7. **Back-of-the-Envelope Estimation**
   7.1 Load Estimation
   7.2 Storage Estimation
   7.3 Resource Estimation
   7.4 Real-World Example (Twitter-like System)

PART IV – DATABASE DESIGN AND SCALING

8. **Databases and State Management**
   8.1 Stateful vs Stateless Systems
   8.2 Role of Databases in System Design
9. **Database Indexing**
   9.1 Full Table Scan
   9.2 Indexes and B-Trees
   9.3 Performance Trade-offs
10. **Database Partitioning**
    10.1 Table Partitioning Concepts
    10.2 Partition Pruning
    10.3 Application vs Database-Level Partitioning
11. **Master-Slave Architecture**
    11.1 Read and Write Separation
    11.2 Replication Strategies
    11.3 Read Scalability
12. **Multi-Master Architecture**
    12.1 Write Scalability
    12.2 Conflict Resolution
    12.3 Geo-Distributed Databases
13. **Database Sharding**
    13.1 Sharding Concepts
    13.2 Sharding Keys
    13.3 Sharding Strategies
    - Range-Based
    - Hash-Based
    - Geographic
    - Directory-Based

    13.4 Challenges of Sharding

PART V – DISTRIBUTED SYSTEMS

14. **Distributed Systems Fundamentals**
    14.1 What is a Distributed System
    14.2 Advantages and Challenges
    14.3 Horizontal Scaling as a Distributed System
15. **Leader–Follower Architecture**
    15.1 Role of Leader
    15.2 Role of Followers
    15.3 Leader Election Overview
16. **Auto-Recoverable Systems**
    16.1 Orchestrators
    16.2 Failure Detection
    16.3 Leader Election for Recovery

PART VI – CONSISTENCY AND RELIABILITY

17. **CAP Theorem**
    - 17.1 Consistency
    - 17.2 Availability
    - 17.3 Partition Tolerance
    - 17.4 CP vs AP Systems
18. **Consistency Models**
    - 18.1 Strong Consistency
    - 18.2 Eventual Consistency
    - 18.3 Trade-offs and Use Cases
19. **Consistency Mechanisms**
    - 19.1 Synchronous Replication
    - 19.2 Asynchronous Replication
    - 19.3 Quorum-Based Protocols
    - 19.4 Consensus Algorithms
    - 19.5 Gossip Protocol
20. **Consistent Hashing**
    - 20.1 Problem with Modulo Hashing
    - 20.2 Hash Ring
    - 20.3 Minimal Data Movement
    - 20.4 Real-World Usage

PART VII – DATA SAFETY AND RECOVERY

21. **Data Redundancy**
    - 21.1 Importance of Redundancy
    - 21.2 Backup Strategies
22. **Data Recovery and Failover**
    - 22.1 Primary-Replica Architecture
    - 22.2 Synchronous vs Asynchronous Replication
    - 22.3 Disaster Recovery

PART VIII – CACHING AND CONTENT DELIVERY

23. **Caching Fundamentals**
    - 23.1 Cache Concepts
    - 23.2 Cache Hit and Miss
    - 23.3 Cache Invalidation
24. **Redis**
    - 24.1 In-Memory Storage
    - 24.2 Redis Data Types
    - 24.3 Redis as Cache
    - 24.4 Redis Pub/Sub

25. **Blob Storage**
    25.1 Binary Large Objects
    25.2 Object Storage Systems
26. **Content Delivery Network (CDN)**
    26.1 Edge Servers
    26.2 Origin Servers
    26.3 TTL and GeoDNS

PART IX – MICROSERVICES AND COMMUNICATION

27. **Monolithic Architecture**
28. **Microservices Architecture**
29. **API Gateway**
30. **Message Brokers**
    30.1 Synchronous vs Asynchronous Communication
31. **Message Queues**
    31.1 Producers and Consumers
    31.2 Parallel Processing
32. **Message Streams**
    32.1 Stream Processing
    32.2 Multi-Consumer Systems
33. **Apache Kafka**
    33.1 Topics and Partitions
    33.2 Consumer Groups
    33.3 High Throughput Systems
34. **Real-Time Pub/Sub Systems**
    34.1 Push-Based Messaging
    34.2 Chat Applications
35. **Event-Driven Architecture (EDA)**
    35.1 Simple Event Notification
    35.2 Event-Carried State Transfer

PART X – PROXIES AND INFRASTRUCTURE

36. **Proxy Servers**
    36.1 Forward Proxy
    36.2 Reverse Proxy
37. **Reverse Proxy in Practice**
    37.1 Load Balancers as Reverse Proxies
    37.2 SSL Termination
    37.3 Security and Caching

PART XI – BIG DATA SYSTEMS

38. **Big Data Fundamentals**
39. **Distributed Data Processing**
40. **Apache Spark Overview**
41. **Use Cases of Big Data Tools**

CHECKLIST

Topic	**Completion**
Introduction to System Design	☐ ☐
Why Study System Design	☐ ☐
Client–Server Architecture	☐ ☐
Backend and Database Basics	☐ ☐
Deployment and Cloud Basics	☐ ☐
DNS and Domain Resolution	☐ ☐
Latency	☐ ☐
Throughput	☐ ☐
Scaling Concepts	☐ ☐
Vertical Scaling	☐ ☐
Horizontal Scaling	☐ ☐
Load Balancer	☐ ☐
Load Balancer Algorithms	☐ ☐
Auto Scaling	☐ ☐
Back-of-the-Envelope Estimation	☐ ☐
Stateful vs Stateless Systems	☐ ☐
Database Basics	☐ ☐
Indexing	☐ ☐
Database Partitioning	☐ ☐
Master–Slave Architecture	☐ ☐
Multi-Master Architecture	☐ ☐
Database Sharding	☐ ☐

Topic	Completion
Distributed Systems	☐ ☐
Leader–Follower Architecture	☐ ☐
Auto-Recoverable Systems	☐ ☐
CAP Theorem	☐ ☐
Strong Consistency	☐ ☐
Eventual Consistency	☐ ☐
Quorum-Based Consistency	☐ ☐
Consensus Algorithms	☐ ☐
Consistent Hashing	☐ ☐
Data Redundancy	☐ ☐
Data Recovery	☐ ☐
Primary–Replica Databases	☐ ☐
SQL Databases	☐ ☐
NoSQL Databases	☐ ☐
SQL vs NoSQL Decision	☐ ☐
Caching Fundamentals	☐ ☐
Cache Invalidation	☐ ☐
Redis	☐ ☐
Redis Pub/Sub	☐ ☐
Blob Storage	☐ ☐
Content Delivery Network (CDN)	☐ ☐
Monolithic Architecture	☐ ☐
Microservices Architecture	☐ ☐
API Gateway	☐ ☐
Message Broker	☐ ☐

Topic	Completion
Message Queue	☐ ☐
Message Stream	☐ ☐
Apache Kafka	☐ ☐
Real-Time Pub/Sub	☐ ☐
Event-Driven Architecture	☐ ☐
Proxy	☐ ☐
Forward Proxy	☐ ☐
Reverse Proxy	☐ ☐
Big Data Fundamentals	☐ ☐
Distributed Data Processing	☐ ☐
Apache Spark	☐ ☐
System Design Best Practices	☐ ☐
System Design Trade-offs	☐ ☐

Why study System Design? (Easy explanation)

1️⃣ What you usually build in college (Simple setup)

In college or personal projects, you usually build something like this:

User → Backend (NodeJS) → Database → Response

- User clicks a button
- Request goes to your backend
- Backend does some logic + CRUD in DB
- Response comes back

✅ This is **perfect for learning and prototypes**
❌ But it **breaks in the real world**

2️⃣ Real-life problem with this approach

Now imagine your app becomes popular.

Real-life example: Food delivery app 🍔

- 10 users → works fine
- 1,000 users → still okay
- **1 million users at dinner time 😨**

What happens?

- Server becomes **slow**
- App **hangs or crashes**
- Payments **fail**
- Users **leave and uninstall**

💥 One small server + one database **cannot handle millions of users**

3️⃣ Why simple architecture fails

Because in real companies, you must handle:

Problem	Real-life meaning
Scalability	Can system handle more users?

Problem	Real-life meaning
Fault tolerance	What if one server crashes?
Security	Can hackers steal data?
Performance	Is response fast or slow?
Monitoring	Do we know when something breaks?

👉 College projects **don't think about these**
👉 **System Design does**

4️⃣ Real-life analogy (Very simple 🚦)

Think of a **small tea stall vs big restaurant**:

- Tea stall:
  - 1 person
  - 10 customers
  - Simple system works
- Big restaurant:
  - Many chefs
  - Multiple counters
  - Order management
  - Backup staff
  - CCTV & security

📌 **System Design = Designing the restaurant, not just cooking food**

5️⃣ Who is the "Client"?

When system design diagrams show **Client**, it can be:

- React web app
- Android app
- iOS app
- Laptop / Mobile browser
- Any device used by real users

📱💻 Anything that sends a request = **Client**

6️⃣ Why companies care about System Design

Big companies handle:

- Millions of users
- Billions of requests
- Zero downtime expectations

If system design is bad:

- App crashes
- Company loses money
- Users lose trust

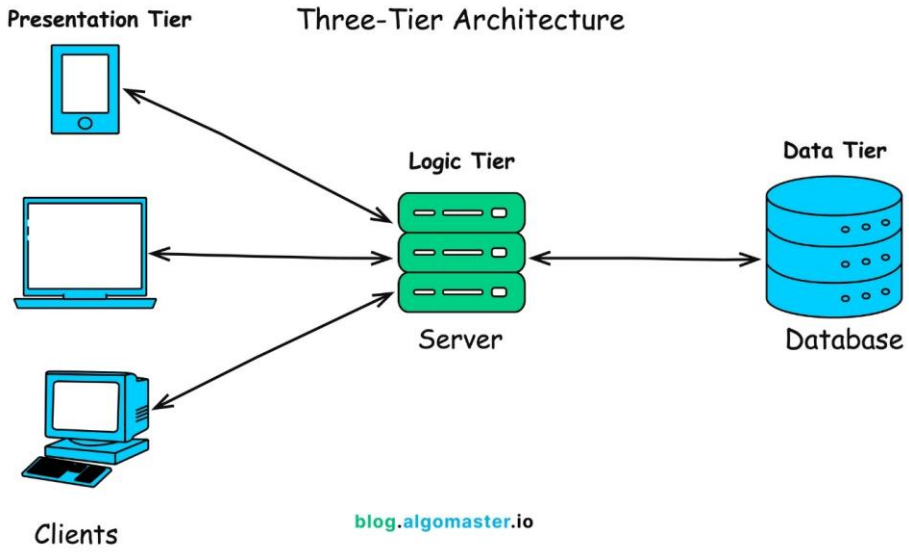
7️⃣ Why YOU should study System Design (as a fresher)

- Helps you **think like a real engineer**
- Makes your backend code **production-ready**
- Required for **product-based interviews**
- Helps you design **scalable Spring Boot / NodeJS apps**

One-line summary (Interview ready ✅)

We study system design to build applications that can scale to millions of users, stay reliable during failures, remain secure, and perform efficiently in real-world conditions.

What is a Server? (Beginner-friendly explanation)



1️⃣ Simple definition

A **server** is just a **computer (physical or virtual)** that runs your application **24×7** and **listens to requests** from users, then sends responses back.

👉 Your **laptop can also act as a server** when you run an app locally.

2️⃣ Localhost example (college project level)

When you run a React / Node / Spring Boot app locally, you see something like:

http://localhost:8080

What this means:

- **localhost** → your own computer
- **127.0.0.1** → IP address of your own machine
- **8080** → port where your app is running

📌 At this stage, **only you** can access the app.

3️⃣ Real-life analogy 🏠

Think of:

- **IP Address** = Home address
- **Port** = Door number (main door, back door, etc.)

Your house can have:

- Door 80 → Web app
- Door 8080 → Backend API
- Door 3306 → Database

Same house (server), **multiple services**, different doors (ports).

4️⃣ What happens when you open a real website?

When you type:

https://abc.com

Behind the scenes:

1️⃣ DNS lookup

- abc.com is sent to DNS

- DNS returns something like: 35.154.33.64

2️⃣ **Browser sends request**

- Browser requests: 35.154.33.64:443
- (443 = default HTTPS port)

3️⃣ **Server receives request**

- Server checks **which application** is listening on port 443
- That application processes request
- Sends response (HTML / JSON / Image)

📌

https://abc.com = https://35.154.33.64:443

5️⃣ **Why do we use domain names?**

Because:

- Remembering **IP addresses is hard**
- Domains are **human-friendly**

So we buy a domain and **map it to server IP**.

Example:

google.com → 142.250.xxx.xxx

6️⃣ **Why not use your laptop as a server?**

In theory, you can:

https://<your-laptop-ip>:8080

But problems:

- Laptop must stay ON 24×7 🔌
- Internet may disconnect
- Security risks
- Not scalable
- Not reliable

❌ **Not practical for real users**

7️⃣ Cloud servers (Real-world solution ☁️)

Instead of managing all this yourself, companies **rent servers** from cloud providers like:

- **Amazon Web Services**
- **Microsoft Azure**
- **Google Cloud Platform**

They give you:

- A **virtual machine**
- Public IP address
- 24×7 availability
- Security & backups

📌 In AWS, this virtual machine is commonly known as an **EC2 instance**.

8️⃣ What is Deployment?

Deployment =

Copying your application code from your laptop → cloud server and running it there.

Example:

- Your app runs on 8080
- Cloud server has public IP
- Users access:

http://<public-ip>:8080

Now **anyone in the world** can use your app 🌍

9️⃣ Real-world application (important 💡)

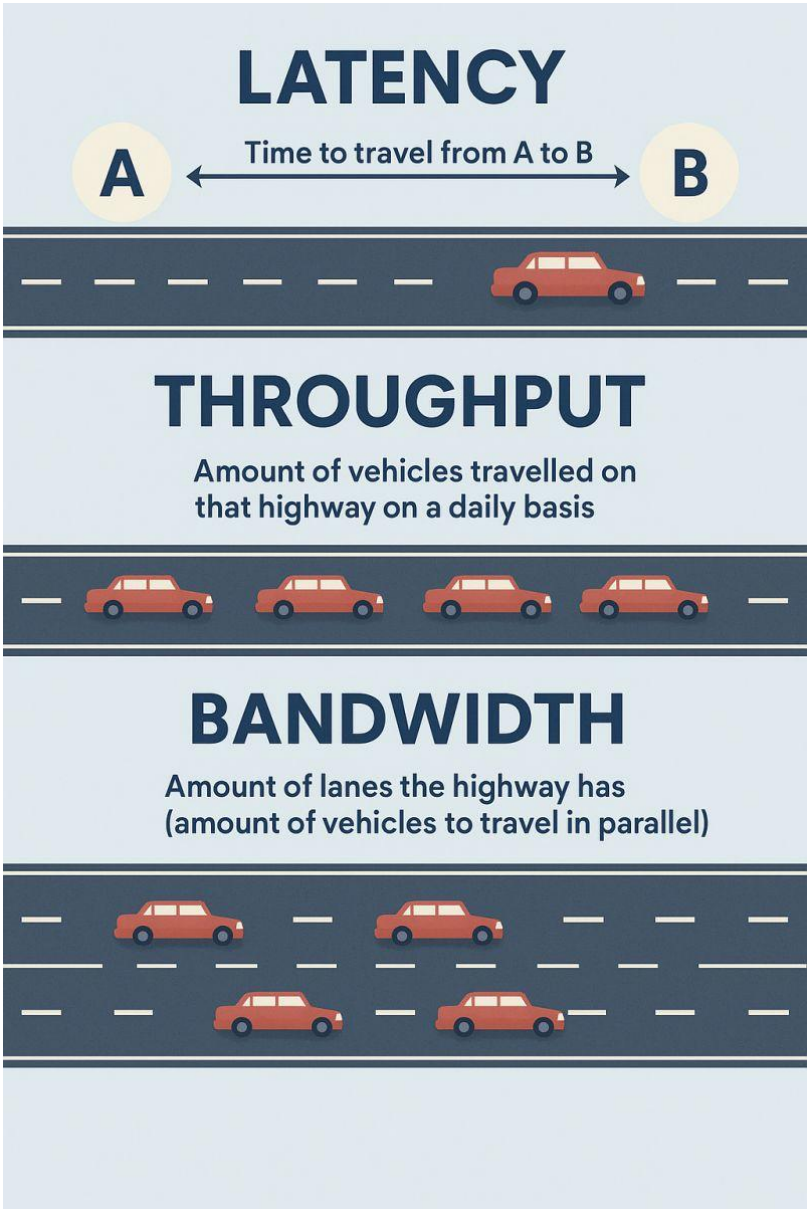
- Banking apps
- Food delivery apps
- E-commerce websites
- Social media apps

All of them run on **multiple servers**, not just one laptop.

🔑 Interview-ready one-liner

A server is a physical or virtual machine that runs applications, listens to client requests on specific ports using an IP address, and sends responses back reliably over the internet.

Latency and Throughput (Very easy explanation)



1️⃣ Latency (How fast is ONE request?)

Simple meaning

Latency = time taken for one request to go to the server and come back with a response.

- Measured in **milliseconds (ms)**
- Focuses on **speed**

Real-life example 🌐

You open a website:

- Page loads in **200 ms** → low latency (fast)
- Page loads in **3 seconds** → high latency (slow)

👉 Faster response = **lower latency**

Round Trip Time (RTT)

RTT =

Time taken for request → server → response back to client

📌 RTT is often used **interchangeably with latency**.

Daily-life analogy 🛵

Ordering tea from a nearby shop:

- Shop next door → 2 minutes → low latency
- Shop 5 km away → 20 minutes → high latency

2️⃣ Throughput (How much work at the same time?)

Simple meaning

Throughput = number of requests the system can handle per second.

- Measured in:
  - **RPS** (Requests Per Second)
  - **TPS** (Transactions Per Second)
- Focuses on **capacity**

Real-life example 🍕

A pizza shop:

- 1 chef → 10 pizzas/hour → low throughput

- 10 chefs → 200 pizzas/hour → high throughput

📌 Throughput = **how much work can be done simultaneously**

3️⃣ Server limitation (Very important 💡)

Every server has a **limit**.

If:

- Server handles **1,000 RPS**
- You send **5,000 RPS**

👉 Server will:

- Slow down
- Drop requests
- Or crash ❌

This is why **scaling** is needed.

4️⃣ Traffic analogy (Best way to remember 🚗)

Concept	Meaning
Latency	Time for one car to reach destination
Throughput	Number of cars passing per hour

- Wide highway → high throughput
- Short distance → low latency

5️⃣ Important relationship

✔ **Low latency ≠ high throughput**
✔ **High throughput ≠ low latency**

Examples:

- Single fast road → low latency, low throughput
- Very wide road but long distance → high throughput, high latency

📌 **Ideal system = low latency + high throughput**

6️⃣ Real-world applications

- Payment systems (need low latency 💳)
- Stock trading platforms
- Video streaming
- Banking systems
- Large APIs

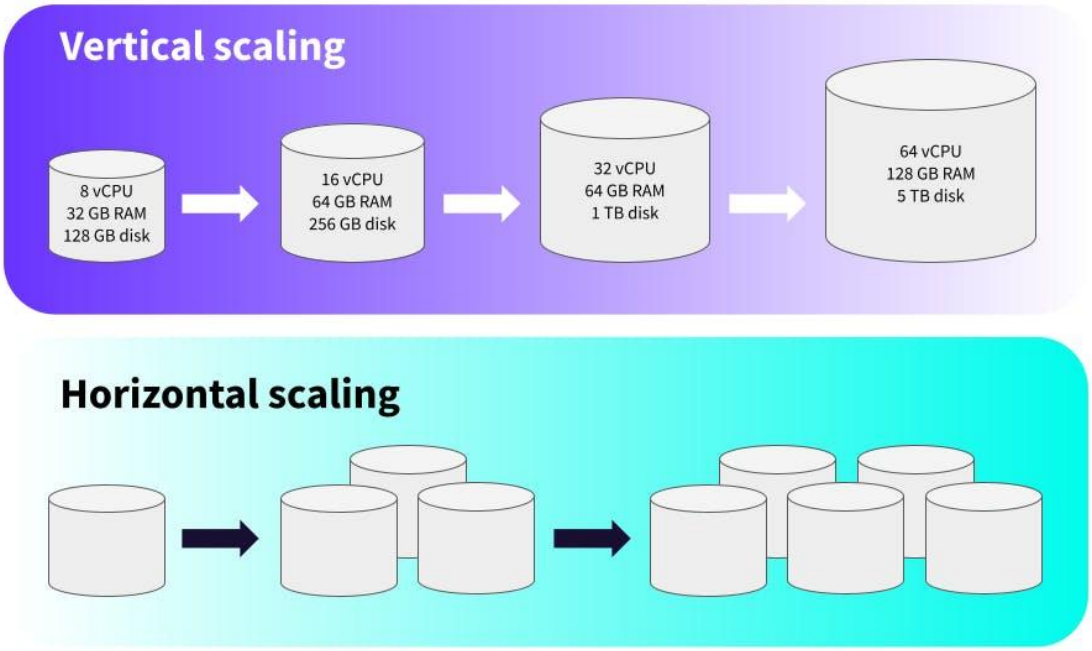
7️⃣ Interview-ready summary 🧠

Latency measures the time taken to complete a single request, while throughput measures how many requests a system can process per second. A well-designed system aims for low latency and high throughput.

8️⃣ One-line memory trick

👉**Latency=Speed**
👉 **Throughput = Capacity**

Scaling and its Types (Easy + Real-life explanation)



1️⃣ Why do websites crash when traffic increases?

When a website suddenly gets **too many users at the same time**, the server becomes **overloaded** and:

- Response becomes slow
- Requests fail

- Server crashes ❌

👉 To avoid this, we **scale** the system.

2️⃣ What is Scaling?

Scaling means increasing your system's capacity so it can handle more users.

There are **two ways** to do this:

1. Increase power of the **same machine**
2. Add **more machines**

3️⃣ Real-life analogy 📱 (Mobile phone example)

- Cheap phone (2GB RAM) → hangs with heavy games
- Better phone (8GB RAM) → runs smoothly

Same thing happens with servers:

- Small server → crashes under heavy traffic
- Bigger or more servers → handles traffic smoothly

4️⃣ Types of Scaling

There are **two main types**:

🔼 Vertical Scaling (Scale Up / Scale Down)

Simple meaning

Increase the power of the same machine

- More RAM
- More CPU
- More storage

Example:

- 4GB RAM → 16GB RAM
- 2 CPU → 8 CPU

📌 Same server, just stronger.

Real-life example 🏋️

One delivery boy:

- Can deliver 10 orders/day ❌
  Train him + give bike:
- Can deliver 30 orders/day ✅

Still **one person**, just more powerful.

Where vertical scaling is used?

- **SQL Databases**
- **Stateful applications** (where data is stored in memory)

Why?

- Data consistency is easier with one machine
- No sync issues

Limitations ❌

- You **can't upgrade forever**
- Hardware limit exists
- Very expensive after a point

👉 That's why vertical scaling alone is **not enough**

➡️ **Horizontal Scaling (Scale Out / Scale In)**

Simple meaning

Add more machines and distribute the load

Instead of:

- 1 powerful server ❌

Use:

- 5 normal servers ✅

Real-life example 🍽️

Restaurant:

- 1 chef → slow service
- 5 chefs → fast service

Same kitchen, **more people working in parallel**

5️⃣ Problem: How do users know which server to hit?

Clients (browser / mobile app):

- Are **not smart**
- Don't know server-1 or server-2 IP

❌ We cannot expose multiple server IPs to users.

6️⃣ Solution: Load Balancer ⚖️

We place a **Load Balancer** in between.

How it works:

1. All clients send requests to **Load Balancer**
2. Load Balancer checks:
    - Which server is free
    - Which server is least busy
3. Forwards request to that server

📌 Clients **never talk directly to servers**

Example flow:

```
Client → Load Balancer → Server 1 / Server 2 / Server 3
```

Load Balancer = Traffic Police 🚦

7️⃣ Real-world setup (Very common)

- Multiple EC2 instances
- One Load Balancer
- Traffic evenly distributed

This is how:

- E-commerce
- Banking apps
- Food delivery apps
- Social media platforms

work in real life.

8️⃣ Why horizontal scaling is preferred?

✅ No hard hardware limit
✅ High availability
✅ Fault tolerant (if one server dies, others work)
✅ Cheaper in long run

👉 Most real-world systems use horizontal scaling

9️⃣ Vertical vs Horizontal (Quick comparison)

Feature	Vertical Scaling	Horizontal Scaling
Machines	One	Many
Cost	Expensive	Flexible
Limit	Hardware bound	Almost unlimited
Failure	Single point	Fault tolerant
Real use	Databases	Web apps / APIs

🔑 Interview-ready one-liner

Scaling is the process of increasing system capacity to handle more load, either by upgrading the same machine (vertical scaling) or by adding more machines and distributing traffic using a load balancer (horizontal scaling).

🧠 **Memory trick**

- **Vertical** → *Stronger machine*
- **Horizontal** → *More machines*

Auto Scaling (Easy explanation with real-life example)

1️⃣ Problem first (why Auto Scaling is needed)

You launched your website on the internet using **one EC2 server**.

- Few users → website works fine ✅
- Suddenly many users → server CPU/RAM becomes full → website crashes ❌

👉 Because **every server has limits**.

So you do **horizontal scaling**:

- Add more servers
- Put a load balancer in front

But now comes a **new problem**…

2️⃣ The money-wasting problem 💸

Traffic is **not constant**.

Example:

- Normal days → **10,000 users** → need **10 servers**
- Festival / sale day → **100,000 users** → need **100 servers**

❌ Bad solution:

- Keep **100 servers running all the time**
- Works technically, but you **waste money** during low traffic

---

3️⃣ Best solution: Auto Scaling 🚀

Auto Scaling means:

Automatically increase or decrease the number of servers based on traffic.

You **don't do it manually**.

The system decides:

- When to **add servers**
- When to **remove servers**

4️⃣ How Auto Scaling works (step by step)

Let's say:

- One EC2 can handle **1,000 users**
- CPU threshold = **90%**

Flow:

1. Users increase
2. CPU usage goes above 90%
3. New EC2 instance is **automatically created**
4. Load balancer starts sending traffic to it
5. Website stays fast and alive ✅

When traffic decreases:

1. CPU usage drops
2. Extra servers are **terminated**
3. You stop paying for them 💰

📌 This automatic up & down is **Auto Scaling**

5️⃣ Real-life analogy 🍕

Pizza shop:

- Normal day → 2 cooks
- Weekend rush → hire 10 cooks
- Late night → send extra cooks home

👉 You **don't keep 10 cooks all day**
👉 You adjust based on demand

That's **Auto Scaling**

6️⃣ Who manages this in real systems?

In **Amazon Web Services**, this is handled using:

- EC2 instances
- Load Balancer
- **Auto Scaling Group (ASG)**

📌 You define rules like:

- CPU > 80% → add server
- CPU < 30% → remove server

7️⃣ Why Auto Scaling is important

✅ Handles traffic spikes
✅ Saves money
✅ No manual intervention
✅ Improves availability
✅ Used by almost all real apps

Examples:

- E-commerce sale
- Cricket match live scores
- Payment apps
- Food delivery apps

8️⃣ Important note (from your text)

The numbers like:

- 1000 users per server
- 90% CPU threshold

are **not fixed**.

👉 In real projects, we find them using **load testing**.

9️⃣ Interview-ready one-liner 🧠

Auto Scaling is a mechanism where the number of servers is automatically increased or decreased based on system load (like CPU usage or traffic), ensuring high availability while optimizing cost.

🔑 **Memory trick**

- **Scaling** → Add/remove servers
- **Auto Scaling** → System does it automatically

Back-of-the-Envelope Estimation (Super easy + interview-ready)

1️⃣ What is Back-of-the-Envelope Estimation?

It is a **quick, rough calculation** to estimate:

- How many **users**
- How much **traffic**
- How much **storage**
- How many **servers**

👉 We **don't aim for 100% accuracy**
👉 We aim for **reasonable approximation**

📌 In **system design interviews**, spend **~5 minutes max** on this.

2️⃣ Why do interviewers like this?

Because it shows:

- You can **think practically**
- You understand **scale**
- You can make **engineering trade-offs**

Real engineers do this **before building anything**.

3️⃣ Golden table you MUST remember 🧠

This helps you calculate fast:

Power of 2	Approx value	Power of 10	Full name	Short
10^{20}	1 Thousand	10^{3}	Kilobyte	KB

Power of 2	Approx value	Power of 10	Full name	Short
10^{20}	1 Million	10^{6}	Megabyte	MB
10^{30}	1 Billion	10^{9}	Gigabyte	GB
10^{40}	1 Trillion	10^{12}	Terabyte	TB
10^{50}	1 Quadrillion	10^{15}	Petabyte	PB

📌 **Powers of 10 = your best friend**

4️⃣ What do we usually estimate?

You correctly narrowed it down to **3 things** 👌

1. **Load Estimation** → traffic
2. **Storage Estimation** → data size
3. **Resource Estimation** → servers & CPU

5️⃣ Example: Twitter

We'll **approximate**, not be perfect.

🔹 A. Load Estimation (Reads & Writes)

Given:

- DAU = **100 million users**
- Tweets per user per day = **10**

Writes per day:

100M users × 10 tweets

= 1 Billion tweets/day

✅ **Writes = 1B/day**

Reads per day:

- Each user reads **1000 tweets/day**

100M users × 1000 reads

= 100 Billion reads/day

📌 **Reads are MUCH higher than writes**
(very common in real systems)

🔹 **B. Storage Estimation**

Tweet types:

- 90% normal tweets
- 10% tweets with photos

Size calculation:

- 1 character = 2 bytes
- 1 tweet = 200 characters

200 × 2 = 400 bytes ≈ 500 bytes

Photo size:

- 1 photo ≈ **2 MB**

Tweets per day:

- Total tweets = **1 Billion**
- Tweets with photos = **10% → 100 Million**

Total storage per day:

(500 bytes × 1B tweets)

+ (2 MB × 100M photos)

Approximation:

≈ (1 KB × 1B) + (2 MB × 500M)

≈ 1 TB + 1 PB

≈ 1 PB/day

📌 Ignore **1 TB** because **1 PB is much larger**

✅ **Storage needed ≈ 1 PB per day**

🔹 **C. Resource Estimation (Servers & CPU)**

Given:

- Requests = **10,000 requests/sec**
- CPU time per request = **10 ms**

Total CPU time needed per second:

10,000 × 10 ms

= 100,000 ms CPU / second

CPU capacity:

- 1 core = **1000 ms / second**

100,000 / 1000 = 100 CPU cores

Servers required:

- 1 server = **4 CPU cores**

100 / 4 = 25 servers

✅ **25 servers + Load Balancer**

6️⃣ Real-life analogy 🏭

Think of a factory:

- **Load estimation** → how many orders/day
- **Storage estimation** → warehouse size
- **Resource estimation** → workers & machines

You **estimate first**, then build.

7️⃣ Why approximation is OK?

Because:

- Real numbers change
- Traffic fluctuates
- Systems auto-scale

📌 What matters is **directionally correct thinking**

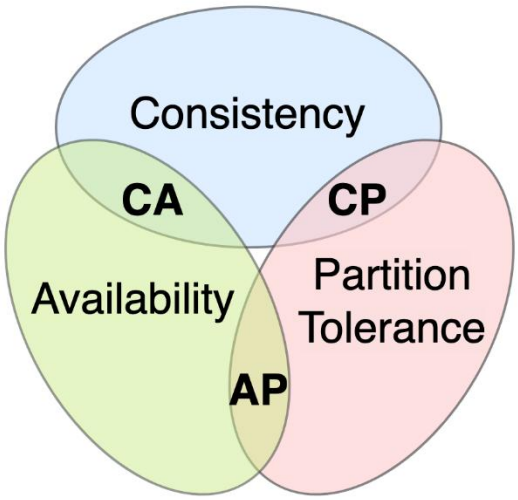
8️⃣ Interview-ready summary (🔥 important)

Back-of-the-envelope estimation is a quick approximation technique used in system design to estimate load, storage, and resources required, helping engineers design scalable systems without over-engineering.

🧠 Memory shortcut

- **Users → Load**
- **Data → Storage**
- **Requests → CPU → Servers**

CAP Theorem (Explained very simply with real-life examples)



1️⃣ First, what is a Distributed System?

A **distributed system** means:

- Data is stored on **multiple servers (nodes)**, not just one
- Servers can be in **different locations**
- Same data is **replicated** across nodes

👉 Each server is called a **Node**

Why do we use distributed systems?

- Handle **more users** (scaling)
- Faster access by serving data from **nearest location**
- **Fault tolerance** (one server down ≠ whole system down)

2️⃣ What is CAP Theorem?

CAP Theorem says:

In a distributed system, you can **only guarantee 2 out of these 3** at the same time.

CAP stands for:

- **C – Consistency**
- **A – Availability**
- **P – Partition Tolerance**

👉 **All 3 together (CAP) is impossible**

3️⃣ Let's understand each term (easy)

🔹 C — Consistency

Every read returns the latest, same data, no matter which node you read from.

📌 All nodes always have **exactly the same data**.

Example (Banking 💰):

- Balance updated to ₹10,000
- Every server must show ₹10,000
- Showing ₹9,000 anywhere is ❌ unacceptable

🔹 A — Availability

The system **always responds**, even if some nodes fail.

📌 You may get a response, but **data might be stale**.

Example (Social media 👍):

- Like count is 101 for you
- Your friend sees 102
- That's OK ✅

🔹 **P — Partition Tolerance**

The system **continues working even if network breaks** between nodes.

📌 Nodes can't talk to each other, but they still serve requests.

Example 🌐:

- Data center in Mumbai can't connect to Delhi
- Both still serve users independently

4️⃣ **Simple 3-node example (A, B, C)**

Normal state:

- A, B, C have same data
- Everything works fine

Case 1: Consistency

- Update happens on **B**
- Update is replicated to **A & C**
- All nodes have same data ✅

Case 2: Availability

- Node **B crashes**
- A & C are still running
- System is still usable ✅

Case 3: Partition Tolerance

- Network breaks
- **B can't talk to A & C**
- B still serves users
- A & C still serve users

👉 This is **network partition**

5️⃣ **The core rule of CAP Theorem**

You can only choose **2 out of 3**:

Combination	Possible?	Meaning
CA	❌ Practically No	Needs no network failure
CP	✅ Yes	Consistency + Partition tolerance
AP	✅ Yes	Availability + Partition tolerance
CAP	❌ Impossible	All 3 together

6️⃣ Why is "P" always required?

Because in real life:

- Network failures **will happen**
- Data centers **will disconnect**
- Packets **will drop**

👉 So **Partition Tolerance is mandatory**

That leaves us with a choice:

- **CP** or **AP**

7️⃣ CP vs AP (most important part 🔥)

🔹 CP (Consistency + Partition Tolerance)

- Stop serving requests during network failure
- Ensure data is always correct

❌ System may be temporarily unavailable
✅ Data is always correct

Used in:

- Banking
- Payments
- Stock trading
- Wallets

🔹 AP (Availability + Partition Tolerance)

- Always serve requests
- Data may be slightly inconsistent

✅ System always works
❌ Data may differ temporarily

Used in:

- Social media
- Likes, views, comments
- Feed systems

8️⃣ Real-life analogy 🏧 vs 👍

ATM (CP)

- Network issue → transaction blocked
- Better to stop than show wrong balance

Instagram Likes (AP)

- Like count mismatch → no big issue
- App should stay online

9️⃣ Interview-ready summary 🧠

CAP Theorem states that in a distributed system, it is impossible to simultaneously guarantee Consistency, Availability, and Partition Tolerance. Since network partitions are unavoidable, systems must choose between Consistency and Availability.

🔑 Memory trick

- **Money apps → CP**
- **Social apps → AP**
- **Distributed system → P is mandatory**

Scaling of Database (Beginner-friendly, step by step)

1️⃣ The starting point (simple setup)

Initially, most applications look like this:

Client → Application Server → Single Database

- One **database server**
- Application queries it
- Database returns data

✅ Works perfectly for:

- College projects
- 5k–10k users
- Low traffic

2️⃣ The problem at scale 😬

As users increase:

- More **read requests** (SELECT queries)
- More **write requests** (INSERT/UPDATE)
- Database CPU, RAM, disk get overloaded

Results:

- Slow queries 🐌
- Timeouts
- Database crash ❌

👉 Database usually becomes the **biggest bottleneck** in a system.

3️⃣ Important principle: Avoid over-engineering ⚠️

You mentioned a **very important system design rule** 👌

If you have 10k users, don't design for 10 million.

Why?

- Extra cost 💸
- More complexity
- Harder to maintain

📌 **Scale gradually, only when needed**

4️⃣ What exactly are we scaling?

You have:

- One database server
- A **User table**
- Application frequently asks:
- Get user by user_id

These are **READ-heavy queries**.

So the **first focus is: make READs faster**.

5️⃣ Step 1: Make reads faster (inside the same DB)

🔹 Use Indexing (VERY important)

If you are frequently querying like:

SELECT * FROM users WHERE user_id = ?

Then:

- Add an **index** on user_id

📌 Index = **book index**

- Without index → scan every row ❌
- With index → jump directly to data ✅

✅ Faster reads
✅ No extra servers
✅ Cheapest optimization

👉 **Always do this first**

6️⃣ Real-life analogy 📚

Imagine a phone directory:

- No index → read every name
- Alphabetical index → direct access

Indexing = **zero cost scaling**

7️⃣ Step 2: Reduce unnecessary DB hits

Many times:

- Same user data is requested again & again
- Database is doing repeated work

Solution:

- Cache frequently accessed data

(We'll go deep into caching later, but idea is simple)

App → Cache → DB

- If data exists in cache → return immediately
- If not → fetch from DB and store in cache

📌 This **reduces DB load drastically**

8️⃣ Step 3: When single DB is still not enough

At some point:

- Even after indexing + caching
- Reads are still too many

Then:

- We **scale the database**

But **not all at once**.

👉 We go step by step:

1. Optimize queries
2. Add indexes
3. Add caching
4. Then move to DB scaling techniques

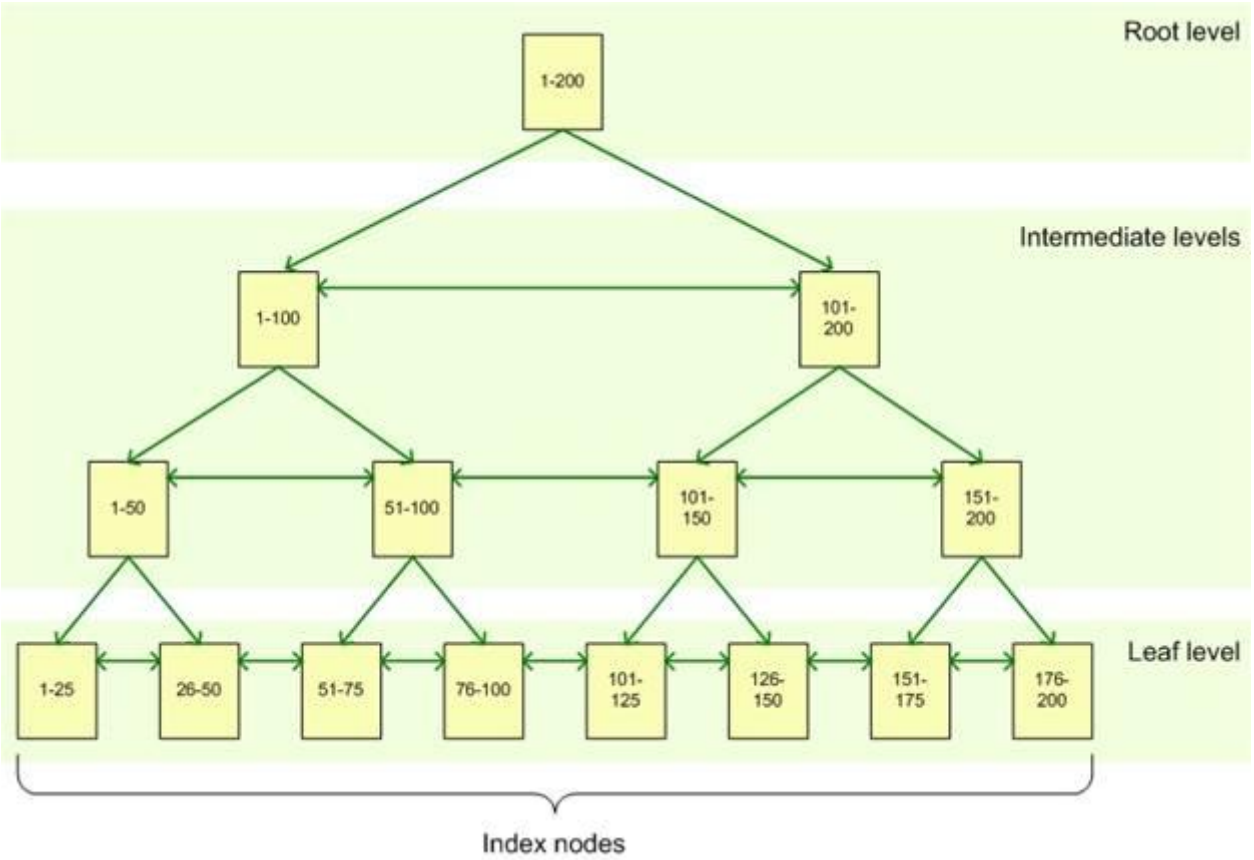
9️⃣ Key takeaway (very important 🧠)

- Database scaling is **not first step**
- Always:
    1. Optimize
    2. Measure
    3. Then scale

🔑 Interview-ready summary

Database scaling is the process of handling increasing load on the database by first optimizing queries and reads, and then gradually introducing techniques like indexing, caching, and replication as traffic grows—avoiding over-engineering.

Indexing



1️⃣ What happens WITHOUT indexing? (Full Table Scan)

When there is **no index**, the database does this:

- Start from **row 1**
- Check row 2
- Check row 3
- …
- Until it finds the required id

This is called a **Full Table Scan**.

Time complexity:

- **O(N)** → if table has 1 crore rows, DB may check all ❌

📌 This becomes **very slow** as data grows.

2️⃣ Real-life analogy 📖 (Best way to remember)

Think of a **big textbook**:

❌ Without index:

- You read **every page** to find a topic

✅ With index:

- Go to **index page**
- Jump directly to the page number

👉 Database indexing works **exactly the same way**

3️⃣ What is Indexing?

Indexing means:

Creating a special data structure on a column so the database can find data faster.

Usually used on:

- id
- user_id
- email
- phone
- Any frequently searched column

4️⃣ How indexing works internally (easy)

When you add an index on id:

- Database creates a **copy of the id column**
- Stores it in a **B-Tree** data structure
- Values are stored in **sorted order**

📌 You don’t manage this manually
📌 Database handles everything

5️⃣ Why B-Tree?

Because:

- Data is **sorted**
- Search is very fast
- Similar to **binary search**

Time complexity:

- **O(log N)** ✅ (much faster than O(N))

Example:

- 1,000,000 rows
- Full scan → 1,000,000 checks
- Index search → ~20 checks 🤯

6️⃣ Simple comparison

Without Index	With Index
Full table scan	Direct jump
O(N)	O(log N)
Slow	Fast
OK for small data	Mandatory for large data

7️⃣ How hard is it to add an index? (Very easy 😄)

You only write **one line of SQL**:

CREATE INDEX idx_user_id ON users(id);

That’s it ✅

- DB creates B-Tree

- DB maintains it
- DB updates it automatically

👉 **Zero headache for developer**

8️⃣ Important trade-off ⚖️

Indexing is great, but:

✅ Faster READs
❌ Slightly slower WRITEs

Why?

- Every INSERT/UPDATE must also update the index

📌 That's why:

- Index only **frequently searched columns**
- Don't index everything blindly

9️⃣ Where indexing is MOST useful?

- Login (email / phone search)
- Fetch user by ID
- Order history by user_id
- Any **read-heavy system**

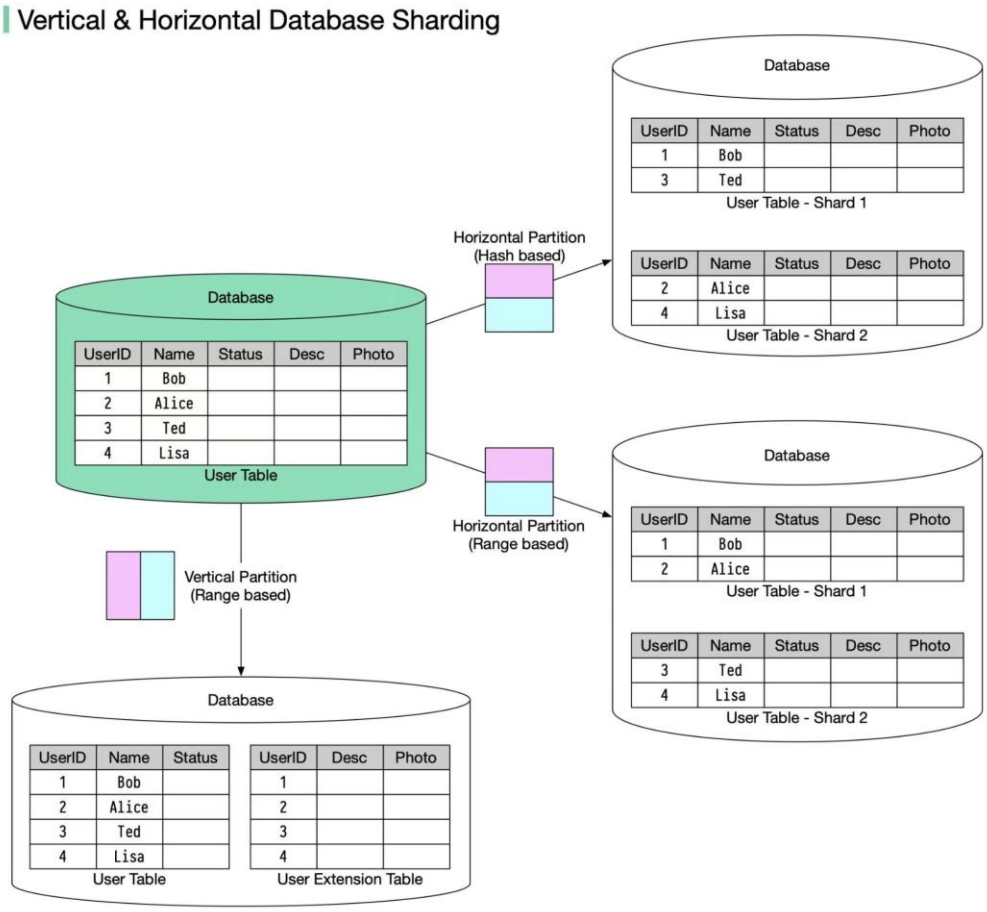
🔑 Interview-ready summary

Indexing is a database optimization technique where a separate data structure like a B-tree is created on a column to allow faster searches, reducing lookup time from O(N) to O(log N).

🧠 Memory trick

- **No index → scan everything**
- **Index → jump directly**

Partitioning



1️⃣ What is Partitioning?

Partitioning means:

Breaking **one very large table** into **multiple smaller tables (partitions)**.

Important point 👉

📌 **All partitions stay on the same database server**

Example:
Instead of:

users

We have:

user_table_1

user_table_2

user_table_3

2️⃣ Why do we need partitioning?

As your application grows:

- users table becomes **huge**
- Index on users table also becomes **huge**
- Searching in a very large index becomes **slow**

❌ Big table → Big index → Slower performance

3️⃣ How partitioning helps 🚀

After partitioning:

- Each small table has its **own small index**
- Searching in a small index is **faster**
- Queries become **more efficient**

📌 Same server, but **less work per query**

4️⃣ Real-life analogy 📚

Think of a **library**:

❌ One giant book with 10,000 pages

- Finding a topic is slow

✅ Split into 10 smaller books

- Each book has its own index
- Searching is much faster

👉 Partitioning = **splitting one giant book into smaller books**

5️⃣ Common doubt: "How do we know which table to query?" 🤔

Earlier, you wrote:

SELECT * FROM users WHERE id = 4;

After partitioning, you **still write the same query** ✅

Why?

Modern databases like **PostgreSQL** are **smart**:

- They know partition rules
- They automatically route the query to:
  - user_table_1 or
  - user_table_2 or
  - user_table_3

📌 This is called **partition pruning**

6️⃣ Do developers need to worry?

Two options:

Option 1️⃣ DB-managed (most common)

- Database handles everything
- App code remains unchanged
- Best and safest option ✅

Option 2️⃣ App-managed

- Application decides which table to query
- More control, but more complexity ❌

📌 Beginners should always prefer **DB-managed partitioning**

7️⃣ Types of Partitioning (just names for now)

You'll see these terms later:

- **Range partitioning** (by date, id range)
- **List partitioning** (country, category)
- **Hash partitioning** (mod(id))

(No need to deep dive now 👍)

8️⃣ Partitioning vs Indexing (very important)

Feature	Indexing	Partitioning
Purpose	Faster search	Manage huge tables
Level	Inside table	Table level
Index size	Still large	Smaller indexes
Used when	Medium data	Very large data

📌 **Partitioning is used AFTER indexing**

🚫 **What partitioning does NOT do ❌**

- Does NOT add new servers
- Does NOT make DB distributed
- Does NOT handle unlimited scale

👉 It is **vertical optimization inside one DB**

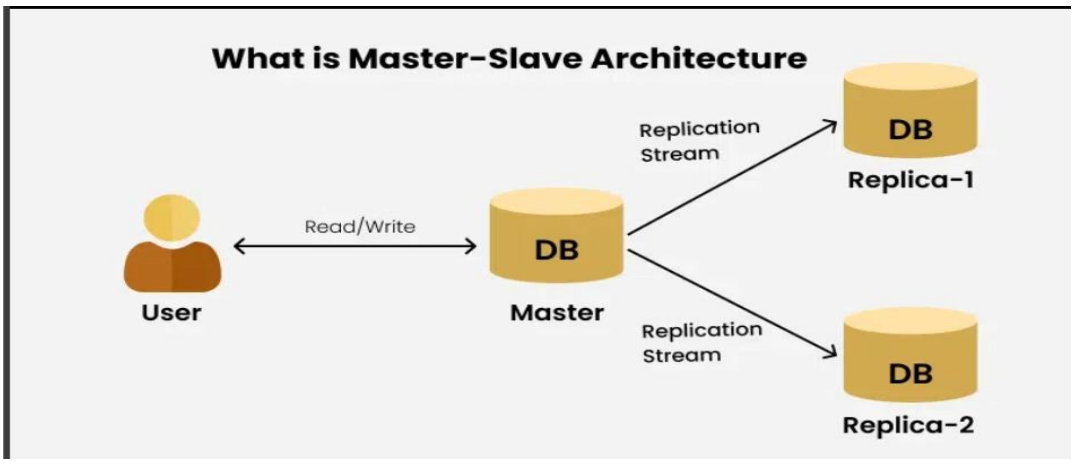
🔑 **Interview-ready summary**

Partitioning is a database optimization technique where a large table is split into smaller tables within the same database server, improving query performance by reducing index size and search space.

🧠 **Memory trick**

- **Indexing** → Faster lookup
- **Partitioning** → Smaller tables

Master–Slave Architecture



1️⃣ When do we need Master–Slave?

You use **Master–Slave architecture** when:

- Indexing ❌ not enough
- Partitioning ❌ not enough
- Vertical scaling ❌ hit hardware limit
- Database is still **slow** or **overloaded**

👉 This usually happens because **READ traffic is very high**.

2️⃣ Core idea (one line)

Separate READs and WRITEs so the database can handle more load.

3️⃣ What is Master–Slave Architecture?

Instead of **one database server**, we use **multiple database servers**:

- **1 Master Node** → handles **WRITE** operations
- **Multiple Slave Nodes** → handle **READ** operations

📌 Same data is **replicated** across all nodes.

4️⃣ Roles explained clearly

🔴 Master Node

- Handles:
  - INSERT
  - UPDATE
  - DELETE
- Source of **truth**
- Writes data first

🟢 Slave Nodes

- Handle:
  - SELECT queries

- Used only for **reading**
- Data comes from master via replication

5️⃣ How data flows (step by step)

✍️ Write request

App → Master DB

1. Application sends write request
2. Master saves data
3. Master **replicates data** to slave nodes
    - Can be **synchronous** or **asynchronous**

📖 Read request

App → Slave DB (least busy)

1. Application sends read request
2. Request is routed to a **slave**
3. Faster response because load is distributed

6️⃣ Real-life analogy 🏢

Think of a **company office**:

- **Manager (Master)**:
    - Makes final decisions
    - Updates official records
- **Employees (Slaves)**:
    - Answer customer questions
    - Refer to copied records

👉 One decision maker, many helpers

7️⃣ Why does this improve performance?

Before:

- One DB handling **all reads + writes**

After:

- Master → only writes
- Slaves → all reads

📌 Since **reads are usually much higher than writes**, performance improves massively 🚀

8️⃣ Replication types (simple)

- **Synchronous**:
  - Master waits till slaves update
  - Strong consistency
  - Slower writes
- **Asynchronous** (most common):
  - Master writes immediately
  - Slaves update later
  - Faster, but slight delay (eventual consistency)

9️⃣ Important limitations ⚠️

❌ Master is still a **single point of failure**
❌ Writes cannot be scaled horizontally
❌ Possible **stale reads** from slaves

👉 That's why:

- Monitoring is critical
- Failover mechanisms are added later

🔁 Where Master–Slave fits in scaling journey

1. Indexing
2. Partitioning
3. Vertical scaling
4. **Master–Slave replication** ✅
5. Sharding (next level)

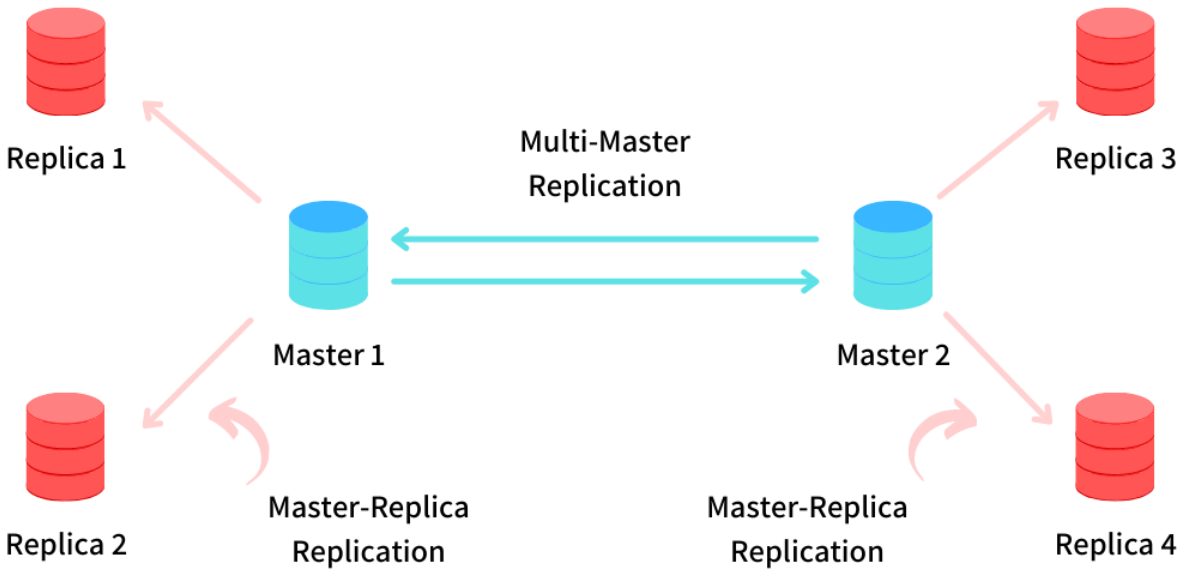
🔑 Interview-ready summary

Master–Slave architecture is a database replication strategy where a single master node handles all write operations, and multiple slave nodes handle read operations, allowing the system to scale read traffic efficiently.

🧠 Memory trick

- **Master → Writes**
- **Slave → Reads**
- **Many readers, one writer**

Multi-Master Setup



Multi-Master Replication

1️⃣ When do we need Multi-Master?

You move to **Multi-Master** when:

- One **master DB** cannot handle all **write requests**
- Writes become slow
- You need **low latency writes from different regions**

- High availability is required for writes

👉 Master–Slave helps **READ** **scaling**

👉 **Multi-Master helps WRITE scaling**

2️⃣ Core idea (one line)

Instead of one master handling all writes, **multiple masters handle writes in parallel**.

3️⃣ What is a Multi-Master Setup?

In **Multi-Master architecture**:

- There are **multiple master databases**
- Each master can:
  - INSERT
  - UPDATE
  - DELETE
- Masters **replicate data among themselves**

📌 No single "write-only" master

4️⃣ Real-world example (India IN)

Let's take your example:

- **North-India-DB**
  - Handles all write requests from North India
- **South-India-DB**
  - Handles all write requests from South India

Flow:

North users → North DB (write)

South users → South DB (write)

North DB ↔ South DB (data sync)

✅ Faster writes

✅ Reduced network latency

✅ No single write bottleneck

5️⃣ Why this improves performance 🚀

Before:

- All writes → one master ❌
- High load → slow writes

After:

- Writes distributed across regions
- Each master handles **local traffic**
- Parallel processing

📌 **Write throughput increases**

6️⃣ Biggest challenge: Conflict Handling ⚠️

This is the **hardest part** of Multi-Master.

What is a conflict?

Same data updated on **two masters at the same time**.

Example:

- User ID = 101
- North DB → name = “Rahul”
- South DB → name = “Rohit”

Now during sync:
❓ Which one is correct?

7️⃣ How conflicts are handled?

There is **NO universal rule** ❌
It depends on **business logic**.

Common strategies:

Strategy	Meaning
Last write wins	Latest timestamp overwrites
First write wins	Ignore later update
Merge	Combine both values

Strategy	Meaning
Manual resolution	Human/admin decision
Business rule	Custom logic (most common)

📌 **You must write conflict-resolution logic yourself**

8️⃣ Real-life analogy ✍️

Google Docs (offline mode):

- Two people edit same line offline
- When internet comes back → conflict
- System asks or auto-merges

👉 Multi-Master conflicts are similar

9️⃣ Trade-offs of Multi-Master

✅ Scales writes
✅ High availability
✅ Low regional latency

❌ Very complex
❌ Conflict resolution is hard
❌ Debugging is difficult
❌ Not suitable for strict consistency systems

1️⃣0️⃣ Where Multi-Master is used?

Used when:

- Write traffic is huge
- Data is geo-distributed

Examples:

- Messaging systems
- Collaborative tools
- Social platforms
- Global SaaS products

❌ **Not ideal for banking / payments**

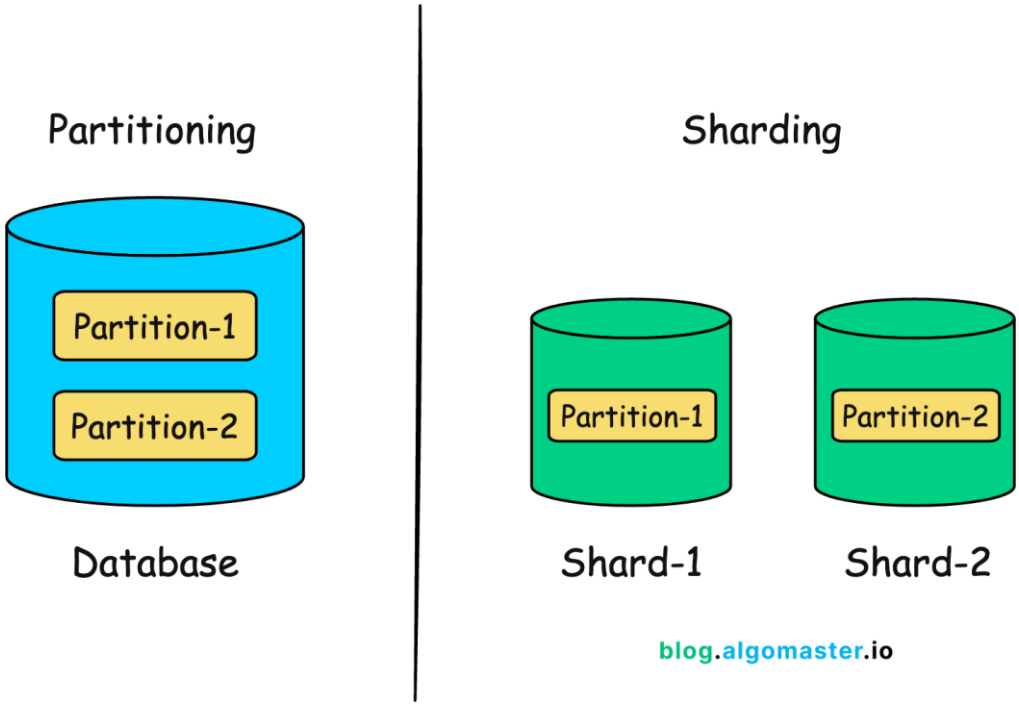
🔑 **Interview-ready summary**

Multi-Master architecture allows multiple database nodes to handle write operations simultaneously, improving write scalability and availability, but introduces complexity in conflict resolution, which must be handled using business-specific logic.

🧠 **Memory trick**

- **Master–Slave** → Scale READs
- **Multi-Master** → Scale WRITEs
- **Complexity** → Very High

Database Sharding



1️⃣ What is Database Sharding?

Sharding means:

Splitting a **very large table** into smaller parts and storing each part on a **different database server**.

- Each server is called a **Shard**
- Each shard holds **only a subset of data**

📌 This is **true horizontal database scaling**.

2️⃣ Partitioning vs Sharding (clear difference)

Partitioning	Sharding
Same server	Different servers
DB handles routing	App handles routing
Easier	Very complex
Vertical optimization	Horizontal scaling

👉 Sharding = **Partitioning + Different servers**

3️⃣ Simple example (Users table)

Suppose you have a huge users table.

Instead of:

One DB → users (1 to 1 billion)

You do:

Shard 1 (DB-1): users 1 – 1M

Shard 2 (DB-2): users 1M – 2M

Shard 3 (DB-3): users 2M – 3M

Each shard:

- Has its own CPU, RAM, disk
- Can be scaled independently

4️⃣ What is a Sharding Key?

A **sharding key** is:

The column used to decide **which shard** data goes into.

Common sharding keys:

- user_id
- order_id
- email
- region

📌 **Good sharding key = evenly distributes data**

❌ Bad key → one shard overloaded (hotspot)

5️⃣ Why sharding is powerful 🚀

- Data no longer fits in one machine ❌
- Writes become too heavy ❌

Sharding helps:

- Unlimited data growth
- High write throughput
- Independent scaling per shard

6️⃣ Why sharding is DIFFICULT ⚠️

This is the most important part.

❌ **App-level complexity**

- DB does NOT know where data lives
- Application must decide:
  - Which shard to READ from
  - Which shard to WRITE to

Example:

user_id = 5 → DB-3

user_id = 1200 → DB-1

You must **code this logic yourself**.

❌ **Joins become expensive**

If:

- users table is in shard-1
- orders table is in shard-3

Then:

- You must fetch data from **multiple DBs**
- Join in application memory

📌 This is slow and costly.

❌ **Consistency becomes hard**

- Data spread across many servers
- Transactions across shards are complex
- Strong consistency is difficult

7️⃣ **Sharding Strategies (explained simply)**

🔹 **1. Range-Based Sharding**

Data split by value ranges.

Example:

Shard 1: user_id 1–1000

Shard 2: user_id 1001–2000

✅ Simple
❌ Uneven load (some ranges very hot)

🔹 **2. Hash-Based Sharding**

Shard decided using hash function.

Example:

HASH(user_id) % number_of_shards

✅ Even data distribution
❌ Very hard to rebalance when adding shards

🔹 **3. Geographic / Entity-Based Sharding**

Split by region or business logic.

Example:

US users → Shard 1

EU users → Shard 2

✅ Low latency, logical grouping
❌ Traffic imbalance (hot regions)

🔹 4. Directory-Based Sharding

A lookup service maps data → shard.

Example:

user_id → shard_id

✅ Flexible, easy to move data
❌ Directory itself can become a bottleneck

8️⃣ Disadvantages of Sharding (summary)

❌ Very hard to implement
❌ App logic becomes complex
❌ Cross-shard joins are expensive
❌ Consistency is difficult
❌ Rebalancing is painful

👉 **Avoid unless absolutely necessary**

9️⃣ When SHOULD you use sharding? (Rules of thumb)

Follow this order 👇

✅ Database Scaling Strategy (Golden Rules)

1. **Vertical scaling first** (simple & cheap)
2. **Indexing + Partitioning**
3. **Master–Slave** → read-heavy systems
4. **Multi-Master** → write-heavy (regional)
5. **Sharding** → when data & writes exceed one machine

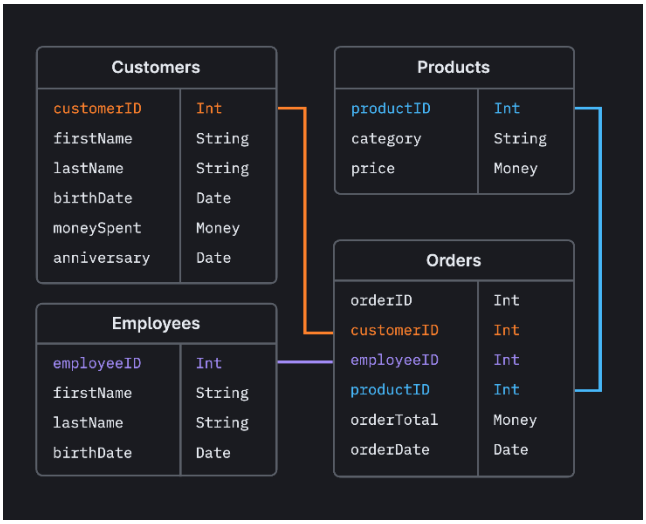
1️⃣0️⃣ Final summary (Interview-ready 🧠)

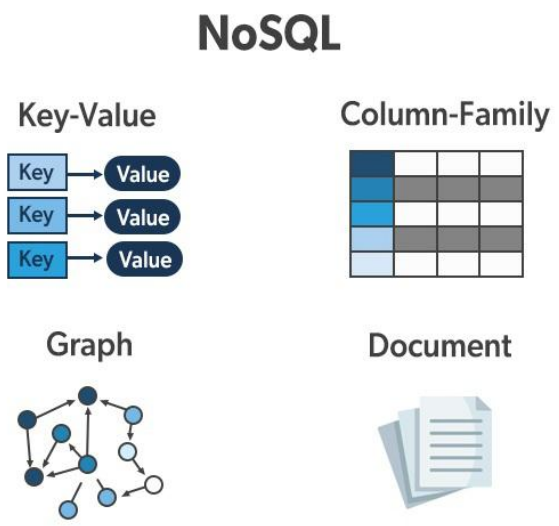
Database sharding is a horizontal scaling technique where a large dataset is split across multiple database servers using a sharding key. While it enables massive scalability, it introduces significant complexity in routing, joins, and consistency, so it should be used only as a last resort.

🧠 Memory tricks

- **Partitioning** → same server
- **Replication** → same data, many servers
- **Sharding** → different data, many servers
- **Avoid sharding until forced**

SQL vs NoSQL Databases





1️⃣ SQL Databases (Relational Databases)

What are SQL databases?

- Data is stored in **tables (rows & columns)**
- **Fixed schema** (structure decided before inserting data)
- Follows **ACID** properties → strong data correctness

Key characteristics

- Tables with relations (foreign keys)
- Strong consistency
- Supports **JOINs**, **complex queries**, **transactions**

Examples

- MySQL
- PostgreSQL
- Oracle
- SQL Server
- SQLite

Real-life example 🏦

Banking system

- Accounts table
- Transactions table
- Balance must always be correct

❌ Wrong balance = disaster
✅ SQL ensures correctness

ACID (in simple words)

Property	Meaning
Atomicity	All or nothing
Consistency	Data always valid
Isolation	Parallel transactions don't conflict
Durability	Data is not lost

📌 This is why **SQL is trusted for money-related systems**

2️⃣ NoSQL Databases (Non-relational)

What are NoSQL databases?

- Data is **not stored in tables**
- **Flexible schema**

- Designed for **scalability & speed**
- Trades strict consistency for performance

3️⃣ Types of NoSQL Databases (very important)

🔹 1. Document-based

- Data stored as **JSON-like documents**
- Flexible fields

Example use case:

- Product reviews
- User profiles

🔹 2. Key–Value Stores

- Data stored as key → value
- Extremely fast

Example use case:

- Caching
- Session storage
- Counters

🔹 3. Column-family Stores

- Data stored column-wise
- Great for massive data & analytics

Example use case:

- Time-series data
- Logs
- IoT data

🔹 4. Graph Databases

- Data stored as **nodes & relationships**
- Focus on connections

Example use case:

- Friends of friends
- Mutual connections
- Recommendations

4️⃣ Schema: Fixed vs Flexible

SQL

Table structure decided first

→ Then data is inserted

NoSQL

Insert data anytime

→ Fields can change anytime

📌 This makes NoSQL **developer-friendly** for evolving products.

5️⃣ Scaling: SQL vs NoSQL (CRUCIAL 🔥)

SQL Scaling

- Primarily **Vertical Scaling**
- Bigger machine (CPU, RAM, Disk)

❌ Hardware limit
❌ Sharding is hard & risky

NoSQL Scaling

- Designed for **Horizontal Scaling**
- Add more servers easily

✅ Sharding is natural
✅ Handles massive data
✅ High availability

6️⃣ Why sharding is avoided in SQL?

Because:

- SQL relies on **ACID**

- Sharding breaks easy transactions
- JOINs across shards are expensive
- Consistency becomes difficult

📌 SQL = correctness first
📌 NoSQL = scale first

7️⃣ When to use SQL Database ✅

Use **SQL** when:

✓ Data is structured

- Fixed fields
- Clear relationships

Examples:

- Users table
- Orders table
- Payments table

✓ Data integrity is critical

- Banking
- Payments
- Stock trading
- Wallet balances

✓ Complex queries needed

- JOINs
- Aggregations
- Analytics
- Reports

📌 **Money + Reports = SQL**

8️⃣ When to use NoSQL Database ✅

Use **NoSQL** when:

✓ **Data is unstructured / semi-structured**

- Reviews
- Comments
- Posts
- Messages

✓ **Massive scale & high traffic**

- Social media feeds
- Likes & views
- Chat systems

✓ **Low latency is important**

- Real-time location tracking
- Gaming
- Live updates

✓ **Data doesn't fit on one server**

- Logs
- Events
- Telemetry
- Analytics pipelines

📌 **Speed + Scale = NoSQL**

👣 **Real-world mixed usage (MOST COMMON 💡)**

Big companies **never use only one DB**.

Example: **E-commerce App**

Feature	**Database**
Users	SQL

Feature	Database
Orders	SQL
Payments	SQL
Product reviews	NoSQL
Search results	NoSQL
Caching	Key–Value (NoSQL)

👉 This is called **polyglot persistence**

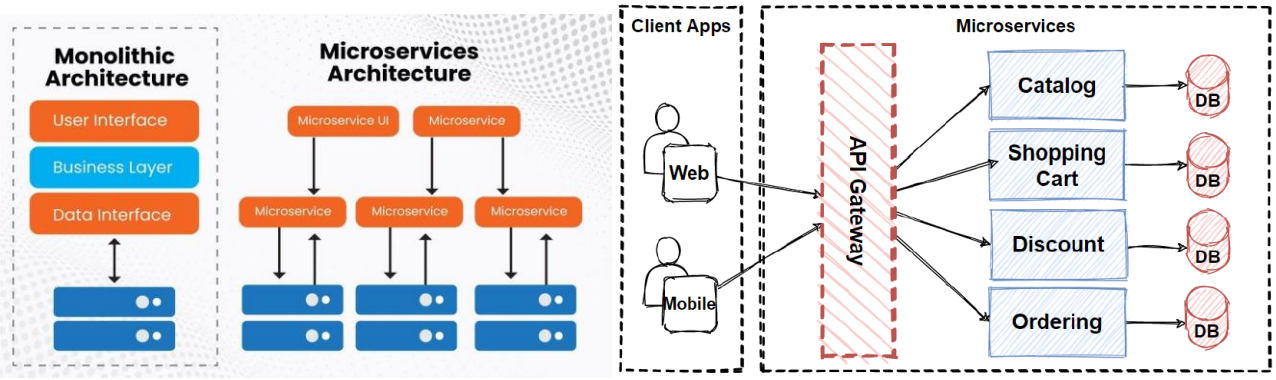
🔑 Interview-ready summary

SQL databases provide strong consistency, fixed schema, and support complex queries, making them ideal for structured and transactional data. NoSQL databases offer flexible schemas, horizontal scalability, and high availability, making them suitable for large-scale, high-performance systems with unstructured data.

🧠 Memory shortcut

- **Money → SQL**
- **Scale → NoSQL**
- **Joins → SQL**
- **Speed → NoSQL**
- **Real apps → Both**

Microservices



1️⃣ Monolith vs Microservices (basic idea)

🔹 Monolith

Monolith = one big application

- Single backend
- All features inside one codebase
- One deployment unit

E-commerce example (Monolith):

- User management
- Product listing
- Orders
- Payments

👉 All written **inside one backend app**

✅ Easy to start
❌ Hard to scale, risky at large size

🔹 **Microservices**

Microservices = many small applications

- Each service does **one business job**
- Each service is **independent**
- Each service is **separately deployable**

E-commerce example (Microservices):

- User Service
- Product Service
- Order Service
- Payment Service

👉 Each is a **separate backend app**

2️⃣ **Real-life analogy** 🏢

Monolith = Small shop

- One person handles:
  - Billing
  - Packing
  - Customer support

- If that person falls sick → shop closes ❌

Microservices = Big company

- Separate teams:
  - Accounts
  - Sales
  - Warehouse
- One team down ≠ company stops ✅

3️⃣ Why do we break Monolith into Microservices?

✅ 1. Independent scaling

Suppose:

- Product service → very high traffic
- Payment service → low traffic

Monolith:

- Scale entire app ❌ (waste of money)

Microservices:

- Scale **only Product Service** ✅

✅ 2. Tech stack flexibility

- User Service → Node.js
- Order Service → Java / Spring Boot
- Payment Service → Go

📌 Each service can use **best tech for its job**

✅ 3. Fault isolation

Monolith:

- Payment crash → entire app down ❌

Microservices:

- Payment crash →

- Users can still browse products
- Login still works ✅

✅ 4. Team scalability (VERY important 💡)

Microservices reflect team structure

- 3 teams → 3 services
- 10 teams → 10 services

Each team:

- Owns one service
- Deploys independently
- Moves faster 🚀

4️⃣ When should you use Microservices?

❌ Don't start with Microservices if:

- Small team (2–3 devs)
- Early-stage startup
- Product requirements changing fast

👉 Start with **Monolith**

✅ Use Microservices when:

- Teams grow
- App becomes complex
- Need independent scaling
- Want to avoid single-point failure

📌 Most companies:

Monolith → Microservices (gradually)

5️⃣ Problem in Microservices: Too many endpoints 😵

Each service has:

- Different IP
- Different domain

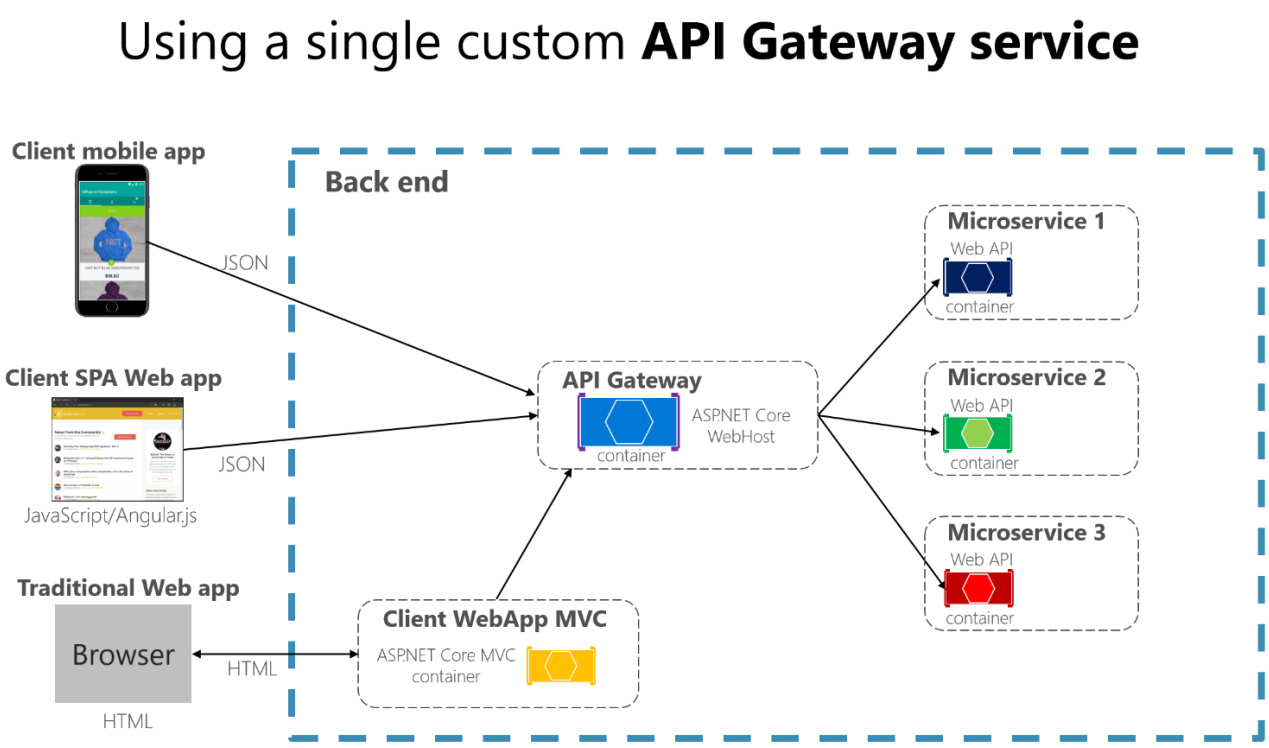
- Different deployment

Example:

- User Service → 192.168.24.32
- Product Service → 192.168.24.38
- Order Service → another IP

❌ Client cannot manage this complexity

6️⃣ Solution: API Gateway 🚪



What is API Gateway?

API Gateway = single entry point for clients.

Flow:

Client → API Gateway → Correct Microservice

Client always hits:

https://api.myapp.com

API Gateway decides:

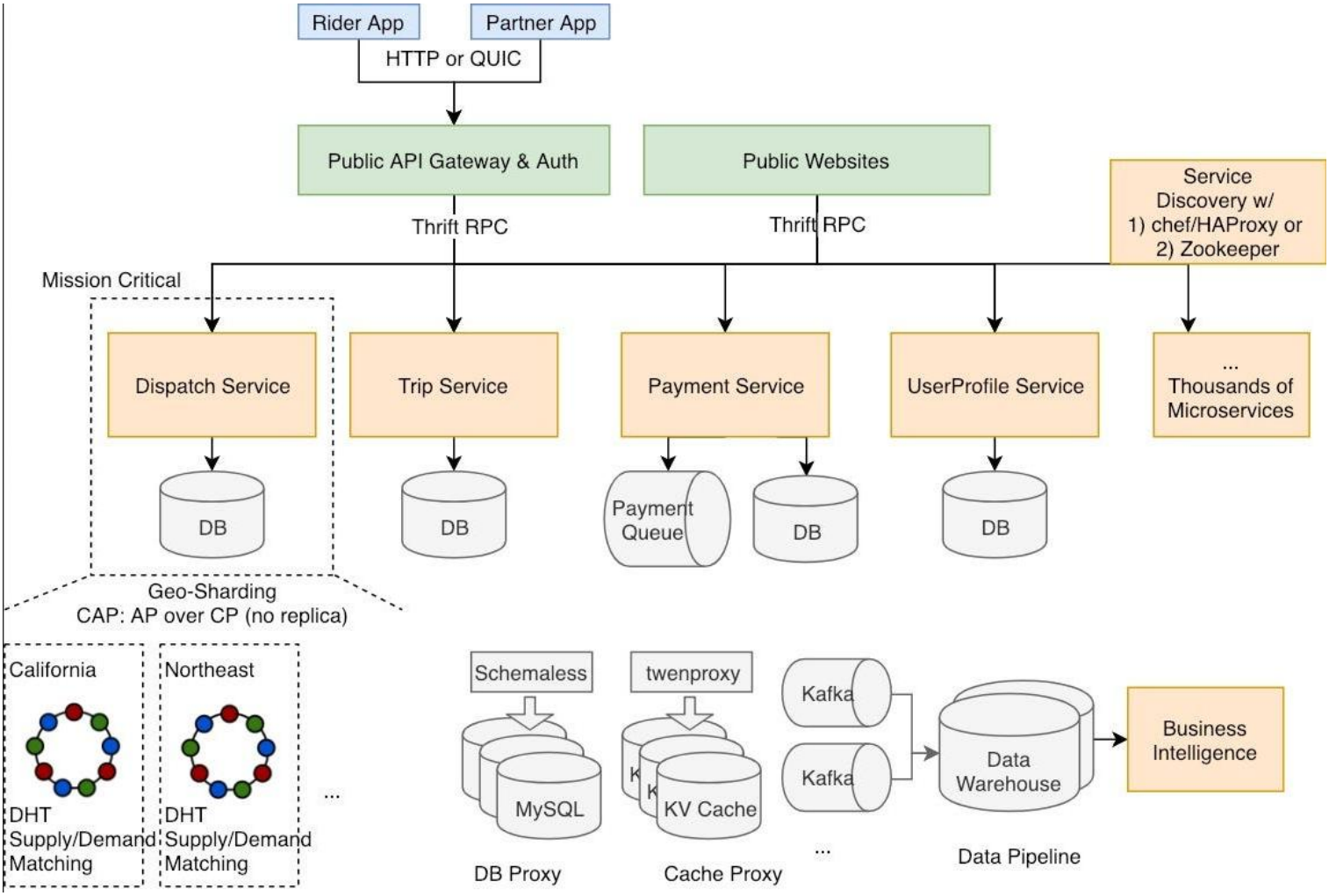
- /users → User Service
- /products → Product Service
- /orders → Order Service

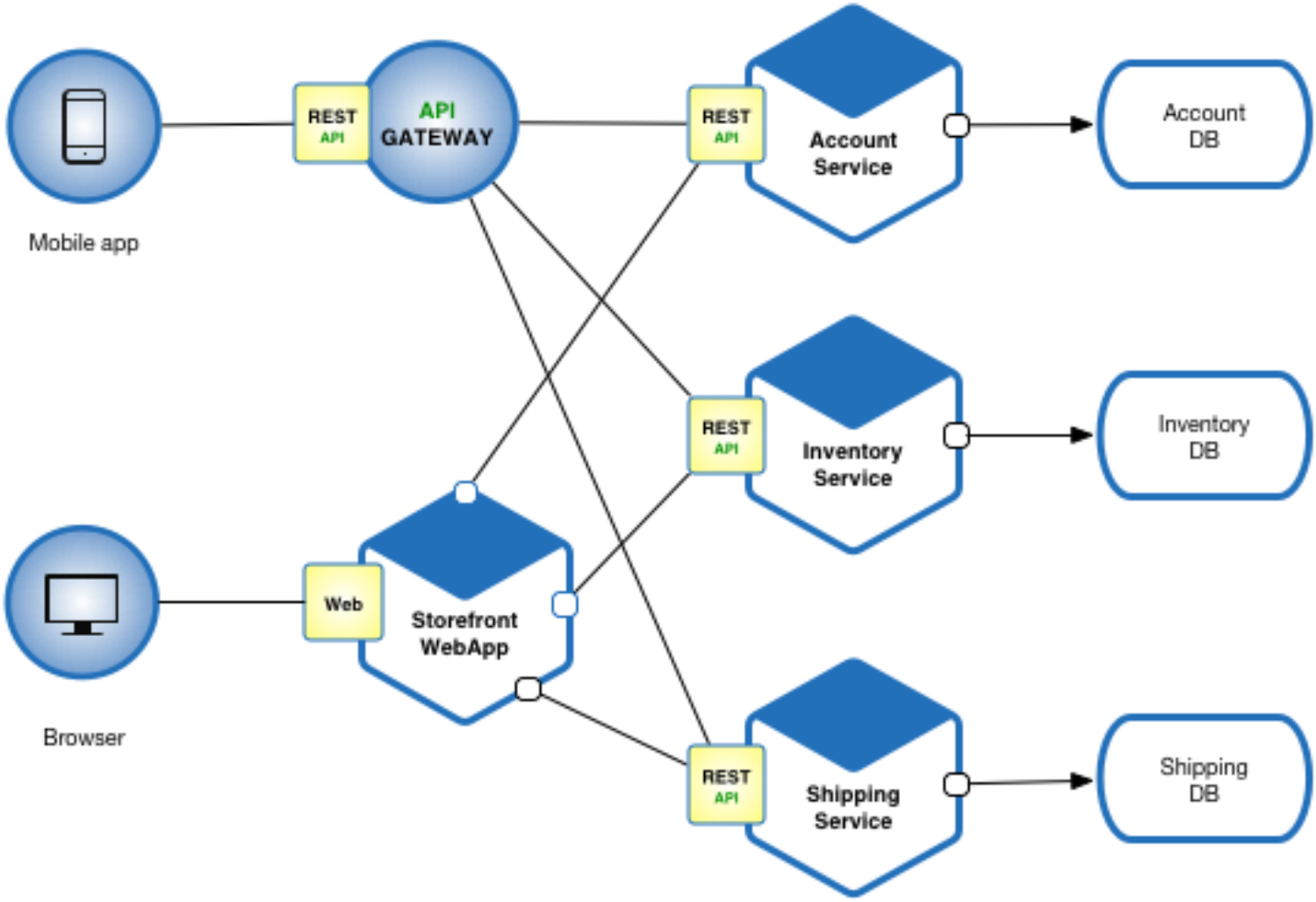
7️⃣ Extra benefits of API Gateway

API Gateway also provides:

- ✅ **Rate Limiting**
  (protect from abuse)
- ✅ **Caching**
  (faster responses)
- ✅ **Security**
  (authentication & authorization)
- ✅ **Monitoring & logging**

8️⃣ Independent scaling in Microservices





Example:

- Product Service → 3 machines
- User Service → 2 machines
- Payment Service → 1 machine

👉 Each service scales **based on its own traffic**

🚮 Trade-offs (important ⚠️)

Microservices are powerful, but:

❌ Complex infrastructure
❌ Network calls between services
❌ Monitoring & debugging harder
❌ Needs DevOps maturity

📌 **Microservices ≠ default choice**

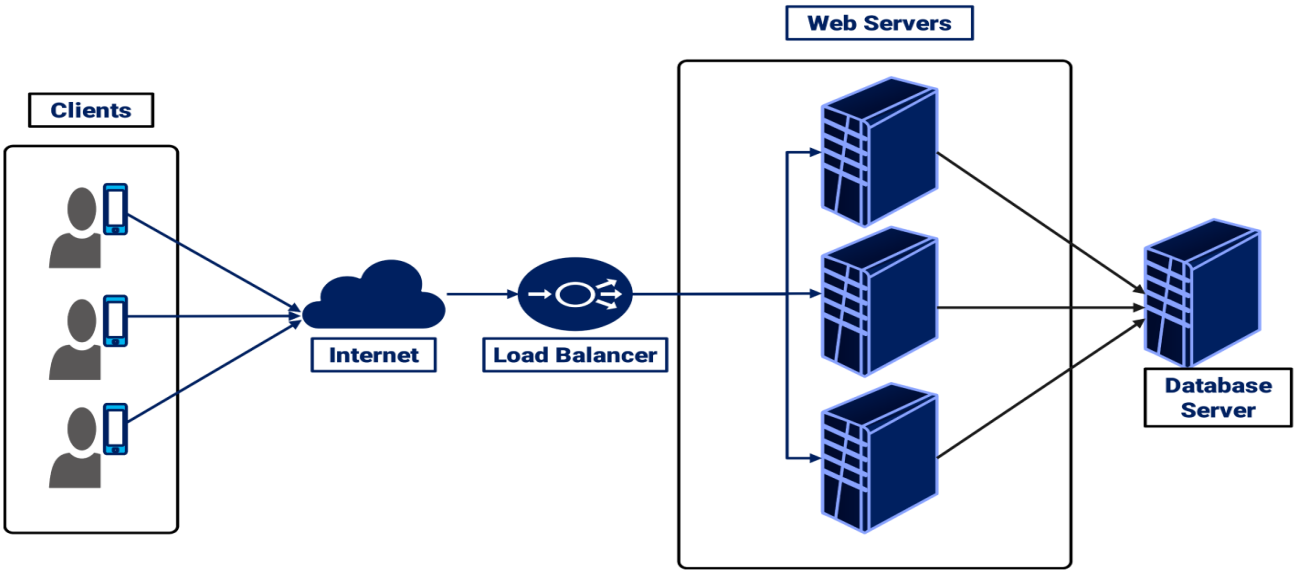
🔑 Interview-ready summary

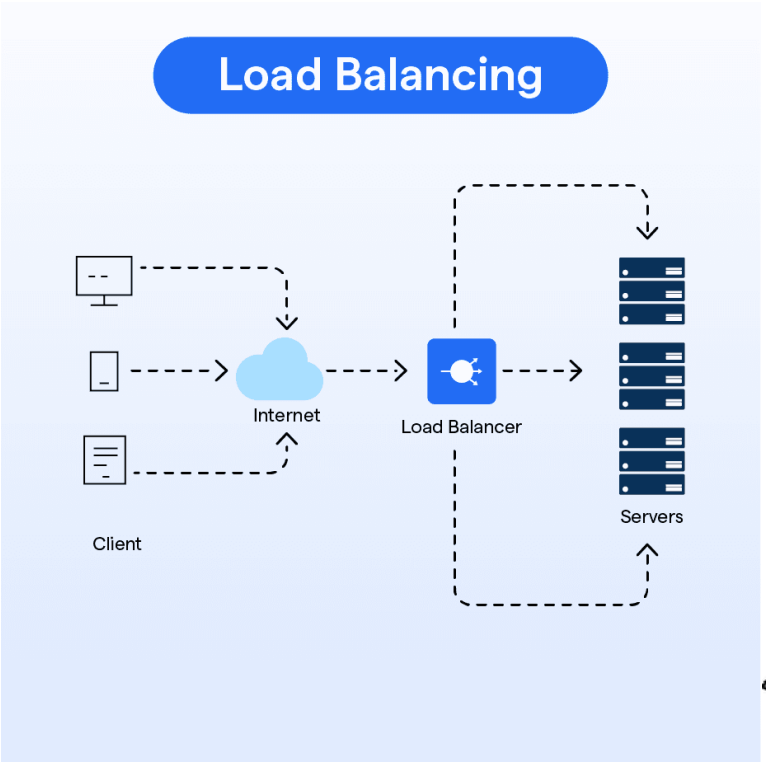
Microservices architecture breaks a large application into small, independent services that can be developed, deployed, and scaled separately, improving scalability, fault isolation, and team productivity at the cost of increased system complexity.

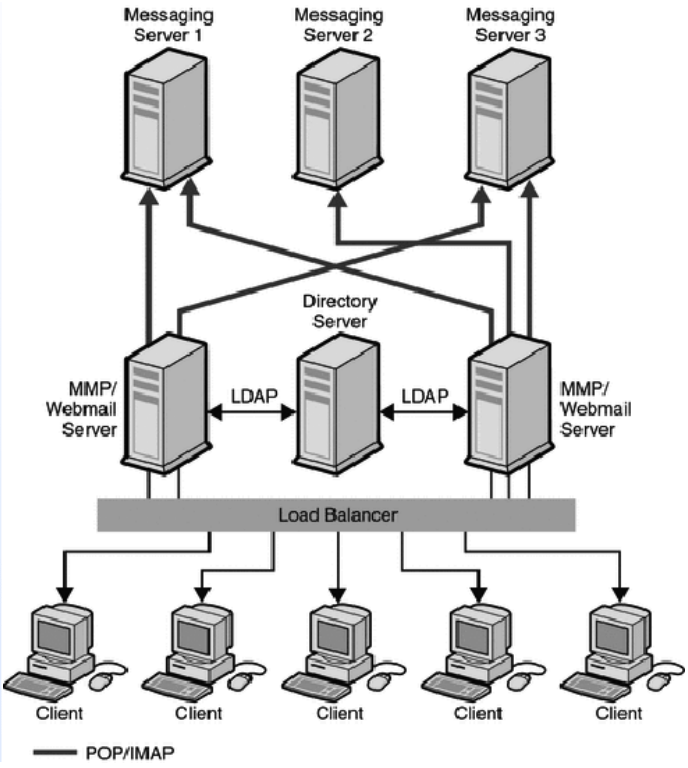
🧠 **Memory tricks**

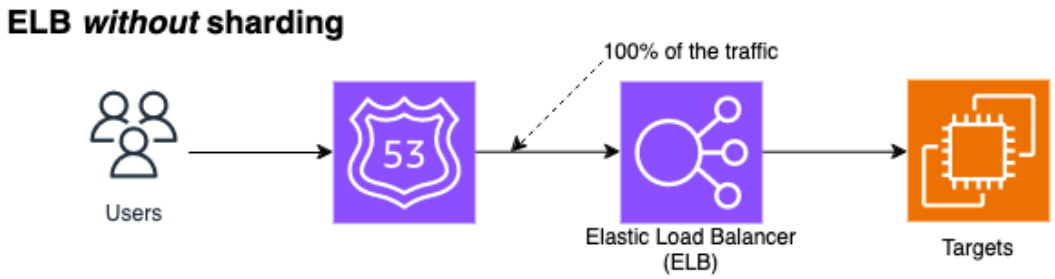
- **Monolith** → One big app
- **Microservices** → Many small apps
- **Scaling problem** → Microservices
- **Early startup** → Monolith

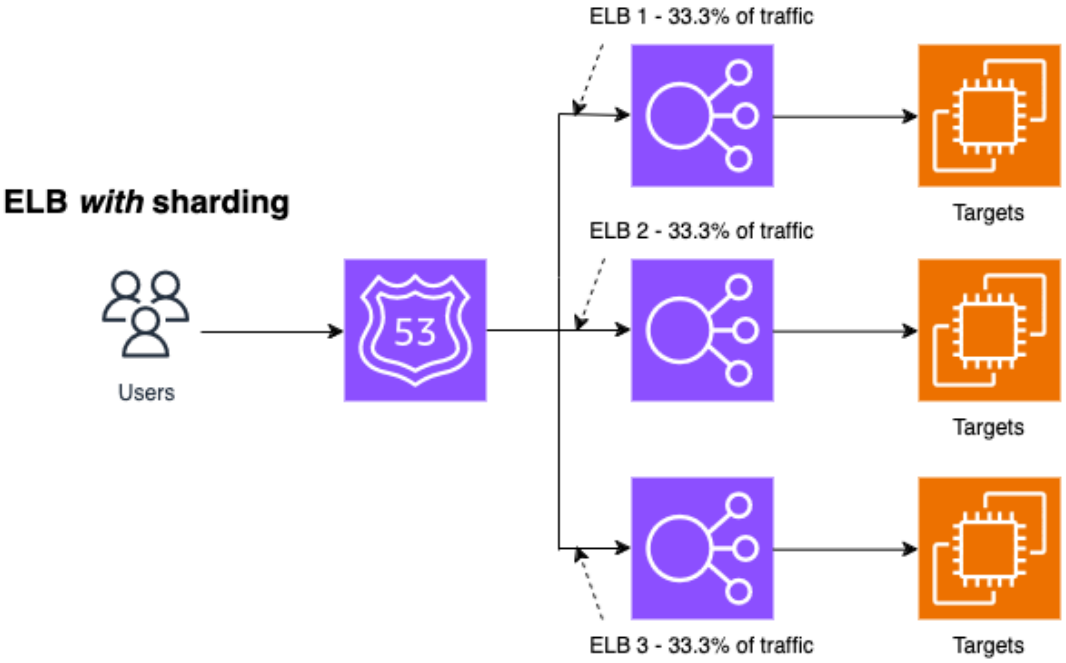
Load Balancer – Deep Dive (Easy explanation + real-life examples)




Load Balancing
Internet
Load Balancer
Servers
Client


Messaging Server 1
Messaging Server 2
Messaging Server 3
Directory Server
MMP/ Webmail Server
LDAP
LDAP
MMP/ Webmail Server
Load Balancer
Client
Client
Client
Client
Client
POP/IMAP


ELB without sharding
100% of the traffic
Users
53
Elastic Load Balancer (ELB)
Targets


ELB with sharding
ELB 1 - 33.3% of traffic
Targets
ELB 2 - 33.3% of traffic
Users
53
Targets
ELB 3 - 33.3% of traffic
Targets

1️⃣ Why do we need a Load Balancer?

When we do **horizontal scaling**, we have **multiple servers** running the same application.

Example:

- Server 1 → IP1
- Server 2 → IP2
- Server 3 → IP3

❌ We **cannot** give all these IPs to the client and say:

"You decide which server to hit."

Why?

- Clients are **not smart**
- They don't know which server is busy or free
- Servers may go up/down anytime

👉 This is where a **Load Balancer (LB)** comes in.

2️⃣ What is a Load Balancer?

A **Load Balancer** is:

A component that sits between clients and servers and distributes incoming requests across multiple servers intelligently.

📌 It is the single point of contact for clients.

Client only knows:

https://api.myapp.com

They **don't know** how many servers exist behind it.

3️⃣ How Load Balancer works (step by step)

Flow:

Client → Load Balancer → Server (least busy)

1. Client sends request to LB's domain
2. Load Balancer:
    - Checks server health
    - Checks current load
3. Forwards request to a suitable server

4. Server responds → LB → Client

✅ Simple for client
✅ Smart routing internally

4️⃣ Real-life analogy 🚦 (Best way to remember)

Think of a **traffic police at a junction**:

- Many roads (servers)
- Many cars (requests)
- Police sends cars to **less crowded roads**

👉 Traffic police = **Load Balancer**

5️⃣ Why Load Balancer is critical 🔥

✅ 1. Handles traffic spikes

- Sudden user surge
- LB spreads traffic evenly
- No single server crashes

✅ 2. Fault tolerance

If:

- Server 2 goes down ❌

Load Balancer:

- Stops sending traffic to it
- Sends traffic only to healthy servers

📌 Client never notices the failure

✅ 3. Enables horizontal scaling

- Add new servers anytime
- LB automatically includes them

👉 **Scale without downtime**

6️⃣ Load Balancer as a single entry point

Clients only hit:

api.myapp.com

Behind the scenes:

- 2 servers today
- 10 servers tomorrow
- 100 servers during sale day

📌 Client code never changes

7️⃣ Load Balancer in the real world

Cloud providers give managed LBs:

- **Amazon Web Services** → Elastic Load Balancer (ELB)
- **Google Cloud Platform** → Cloud Load Balancing
- **Microsoft Azure** → Azure Load Balancer

These handle:

- Health checks
- SSL termination
- Auto scaling integration

8️⃣ What Load Balancer does NOT do ❌

- Does not process business logic
- Does not store data
- Does not replace servers

👉 It only **routes traffic intelligently**

9️⃣ What decides "which server to send traffic to"?

Load Balancers use **algorithms** like:

- Round Robin
- Least Connections
- Weighted routing

- Hash-based routing

📌 We'll study these **next** as *Load Balancer Algorithms*.

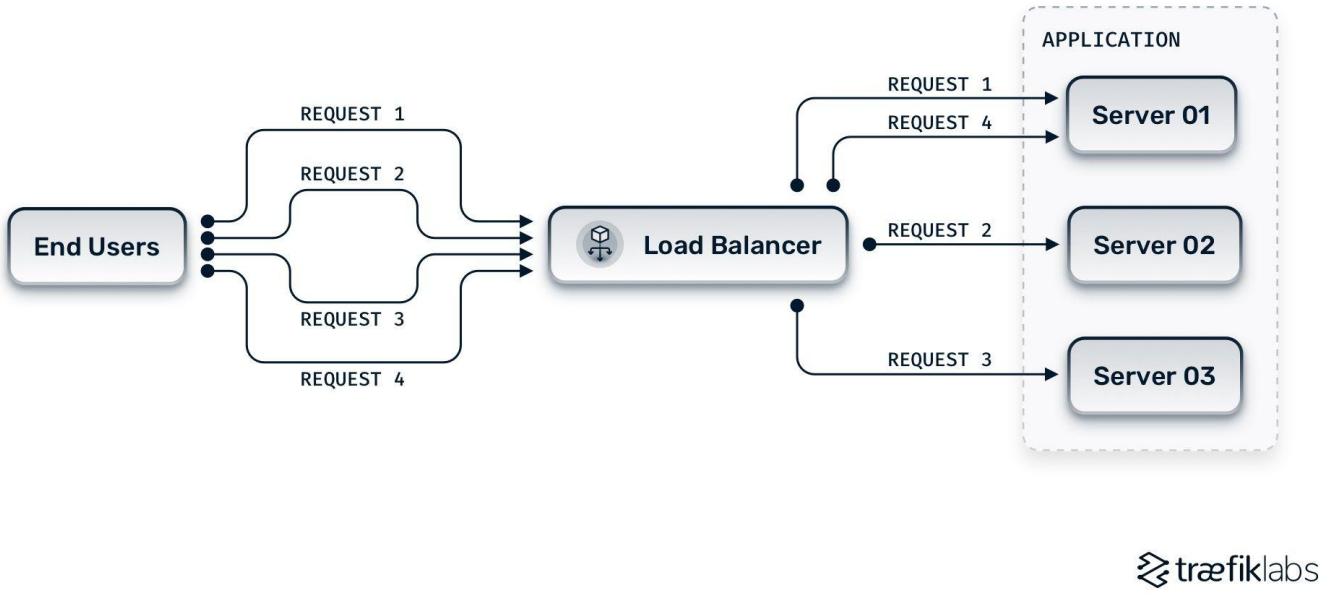
🔑 Interview-ready summary

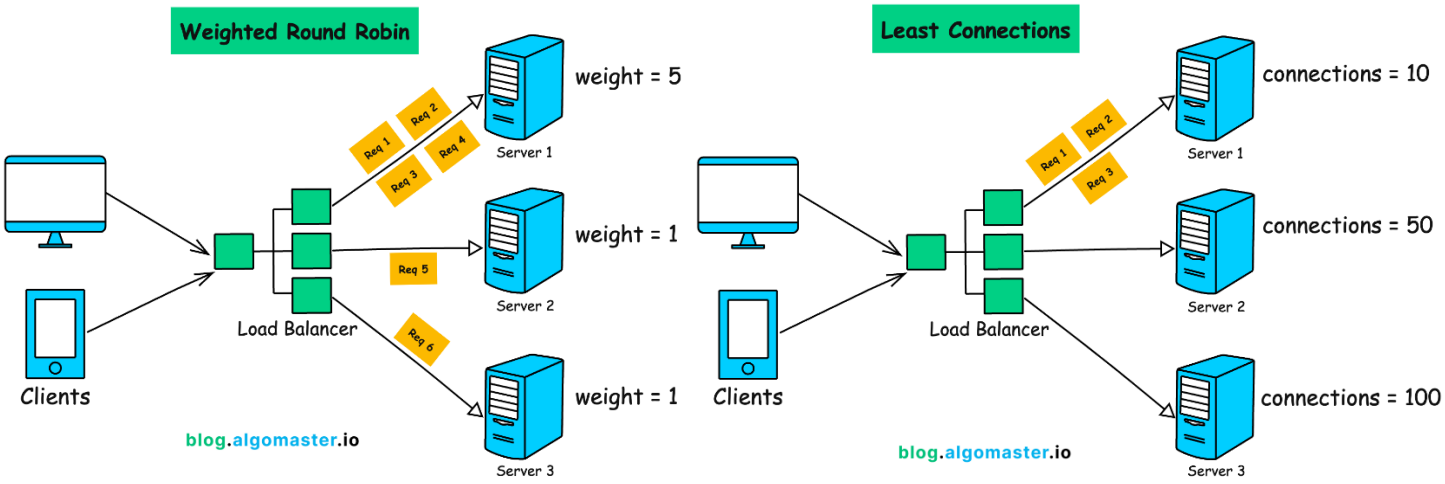
A load balancer acts as a single entry point for clients and distributes incoming traffic across multiple backend servers, improving scalability, availability, and fault tolerance in horizontally scaled systems.

🧠 Memory trick

- **Client knows LB**
- **LB knows servers**
- **Servers don't know clients**

Load Balancer Algorithms (Explained simply with examples)





Load balancer algorithms answer **one simple question**:

"Which server should handle this request?"

Let's understand the **most common algorithms** one by one, in very easy language.

1️⃣ Round Robin Algorithm

How it works

Requests are sent to servers **one by one in a circular order**.

If you have:

- Server-1
- Server-2
- Server-3

Then traffic goes like this:

Request 1 → Server-1

Request 2 → Server-2

Request 3 → Server-3

Request 4 → Server-1

Request 5 → Server-2

Request 6 → Server-3

🔁 Repeat forever.

Real-life analogy 🍕

3 pizza counters:

- First customer → Counter 1
- Second → Counter 2
- Third → Counter 3
- Fourth → Counter 1 again

✅ **Advantages**

- Very **simple**
- Easy to implement
- Works well when **all servers are identical**

❌ **Disadvantages**

- Ignores:
  - Server load
  - Server health
- If one server is slow, it still gets equal traffic ❌

2️⃣ **Weighted Round Robin Algorithm**

How it works

Same as Round Robin, but **powerful servers get more requests**.

Example:

- Server-1 → weight 1
- Server-2 → weight 1
- Server-3 → weight 2 (bigger machine)

Traffic distribution:

Server-1 → 1 request

Server-2 → 1 request

Server-3 → 2 requests

Real-life analogy 🏭

- Small shop → 1 worker

- Big shop → 2 workers

More workers = more customers handled.

✅ **Advantages**

- Handles **unequal server capacity**
- Better than simple Round Robin

❌ **Disadvantages**

- Weights are **static**
- Real-time CPU/RAM load is ignored

3️⃣ **Least Connections Algorithm**

How it works

The load balancer checks:

Which server currently has the fewest active connections?

Then it sends the request to that server.

Connections can be:

- HTTP
- TCP
- WebSocket
- Long-running API calls

Real-life analogy ☎️

Call center:

- Operator with **least calls** gets the next customer

✅ **Advantages**

- **Dynamic load balancing**
- Adapts to real-time traffic
- Very effective for uneven workloads

❌ **Disadvantages**

- Not ideal if:
    - Some connections are very long
    - Some are very short

(One long connection can block a server)

🔽 Hash-Based Algorithm

How it works

Load balancer:

1. Takes an input (IP / user_id / session_id)
2. Applies a **hash function**
3. Hash decides the server

Example:

hash(user_id) → Server-2

Same user → same server every time.

Real-life analogy 📜

Customers always go to:

- Their **assigned relationship manager**

✅ Advantages

- **Session persistence**
- Same user → same server
- Useful when server keeps session data

❌ Disadvantages

- Adding/removing servers:
    - Hash mapping breaks
    - Users may go to different servers
- Needs extra care (consistent hashing)

5️⃣ Quick comparison table 🧠

Algorithm	Best for	Problem
Round Robin	Equal servers	No load awareness
Weighted RR	Unequal servers	Static weights
Least Connections	Real-time balance	Long connections
Hash-Based	Sticky sessions	Server changes

6️⃣ Which one is used in real life?

- **Round Robin** → simple apps
- **Weighted RR** → mixed server sizes
- **Least Connections** → APIs, web apps
- **Hash-Based** → login sessions, carts

📌 Often combined with **health checks**

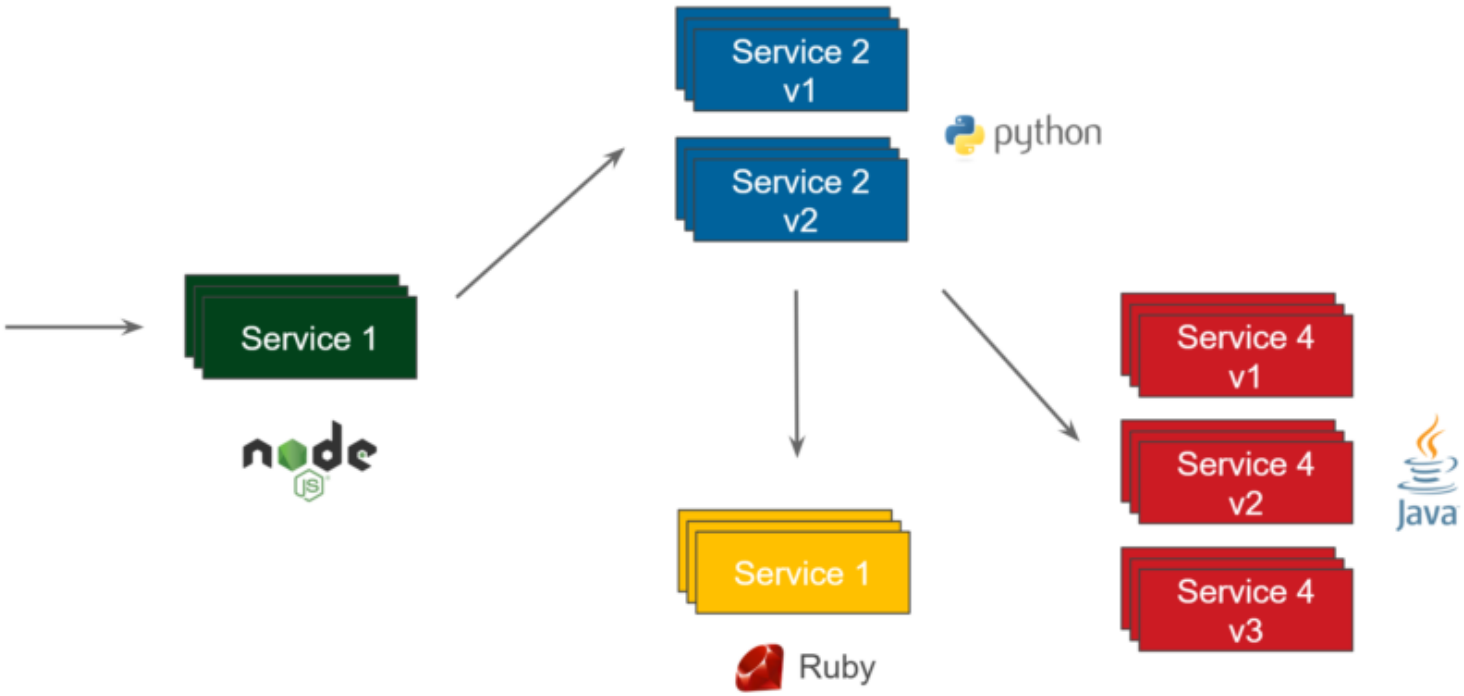
🔑 Interview-ready summary

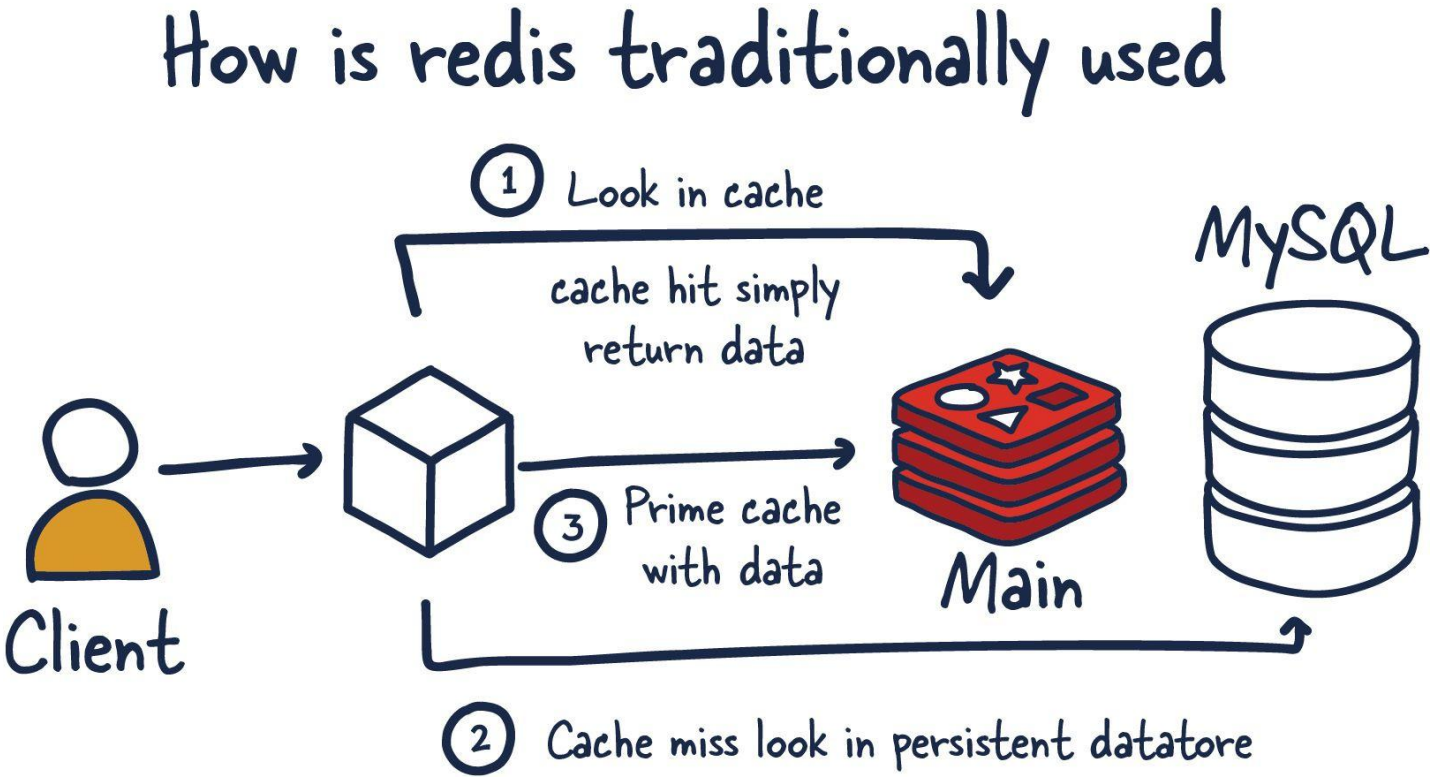
Load balancer algorithms decide how incoming requests are distributed among backend servers. Common strategies include Round Robin, Weighted Round Robin, Least Connections, and Hash-Based routing, each suited for different traffic and workload patterns.

🧠 Memory tricks

- **Round Robin** → Turn by turn
- **Weighted** → Bigger server, more work
- **Least Connections** → Least busy wins
- **Hash-Based** → Same user, same server

Caching (Beginner-friendly + real-life examples)





1️⃣ What is Caching? (in very simple words)

Caching means:

Storing **frequently used data** in a **very fast storage** so next time we don't recompute or re-fetch it.

📌 Instead of going to a **slow database**, we serve data from a **fast cache**.

2️⃣ Why caching is so powerful 🔥

Without cache:

- DB fetch → 500 ms
- Backend processing → 100 ms
- Total → **600 ms**

With cache:

- Redis fetch → **20–60 ms**

👉 **10x faster response**

3️⃣ Real-life analogy 🧠

Think of your brain vs notebook:

- First time → read from book (slow)
- Next time → remember from brain (fast)
- 👉 Brain = **Cache**
  👉 Book = **Database**

4️⃣ Simple caching flow (blog example)

Route: /blogs

First request (Cache MISS ❌)

1. Request comes
2. Cache is empty
3. Fetch from database (800 ms)
4. Store result in cache
5. Return response

Next requests (Cache HIT ✅)

1. Request comes
2. Data found in cache

3. Return in **20 ms**

📌 **Database is not touched**

5️⃣ What is Cache Invalidation? (very important ⚠️)

When data changes, cache becomes **stale**.

Example:

- New blog added
- Old cached /blogs is wrong ❌

👉 We must **invalidate or update cache**

6️⃣ Common Cache Invalidation strategies

🔹 TTL (Time To Live)

- Cache expires after fixed time
- Example: 24 hours

Cache → auto delete after 24 hrs

Simple and widely used ✅

🔹 Write-through / Write-behind (later topic)

- Update cache whenever DB updates

🔹 Manual eviction

- Delete cache explicitly when data changes

📌 **Cache invalidation is one of the hardest problems in system design**

7️⃣ Benefits of Caching (why everyone uses it)

- 🚀 Faster response time
- 📉 Reduced DB load
- 💸 Lower infra cost
- 📈 Better scalability

- 😊 Happy users

8️⃣ Types of Caches (overview)

🔹 1. Client-side Cache

Stored on **user device**.

Examples:

- Browser cache
- HTML, CSS, JS files

✅ Reduces server load
❌ Not good for dynamic data

🔹 2. Server-side Cache (Most important 🔥)

Stored on **server**.

Examples:

- **Redis**
- Memcached

Used for:

- API responses
- DB query results
- Session data

🔹 3. CDN Cache

Used for **static content**.

Examples:

- Images
- Videos
- CSS / JS

Popular CDNs:

- AWS CloudFront

- Cloudflare

📌 Cached closer to user → low latency

🔹 4. Application-level Cache

Inside application memory.

Examples:

- Java HashMap
- Guava Cache
- Caffeine

📌 Fastest, but limited size & not shared across servers

🌍 Where caching fits in system design

Caching is used:

- Before database
- Before expensive computation
- At multiple layers (client, CDN, server)

Client → CDN → App Cache → DB

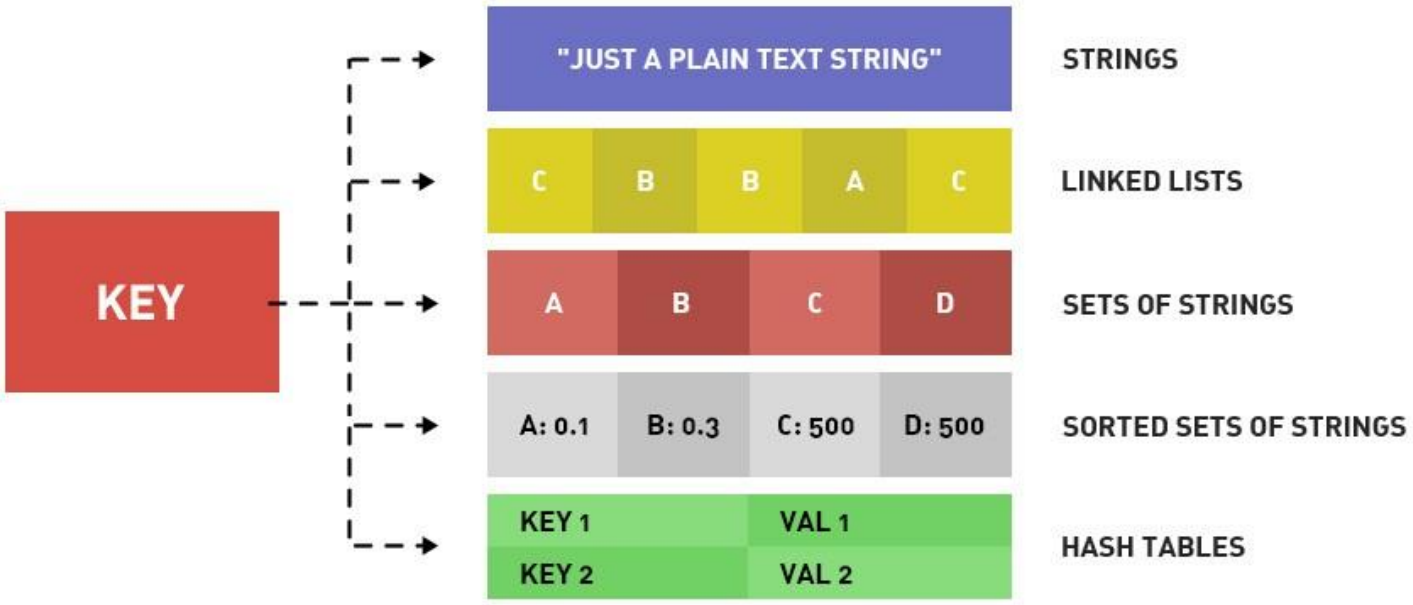
🔑 Interview-ready summary

Caching is the technique of storing frequently accessed or pre-computed data in a fast storage layer like Redis or CDN to reduce latency, lower backend load, and improve system scalability.

🧠 Memory tricks

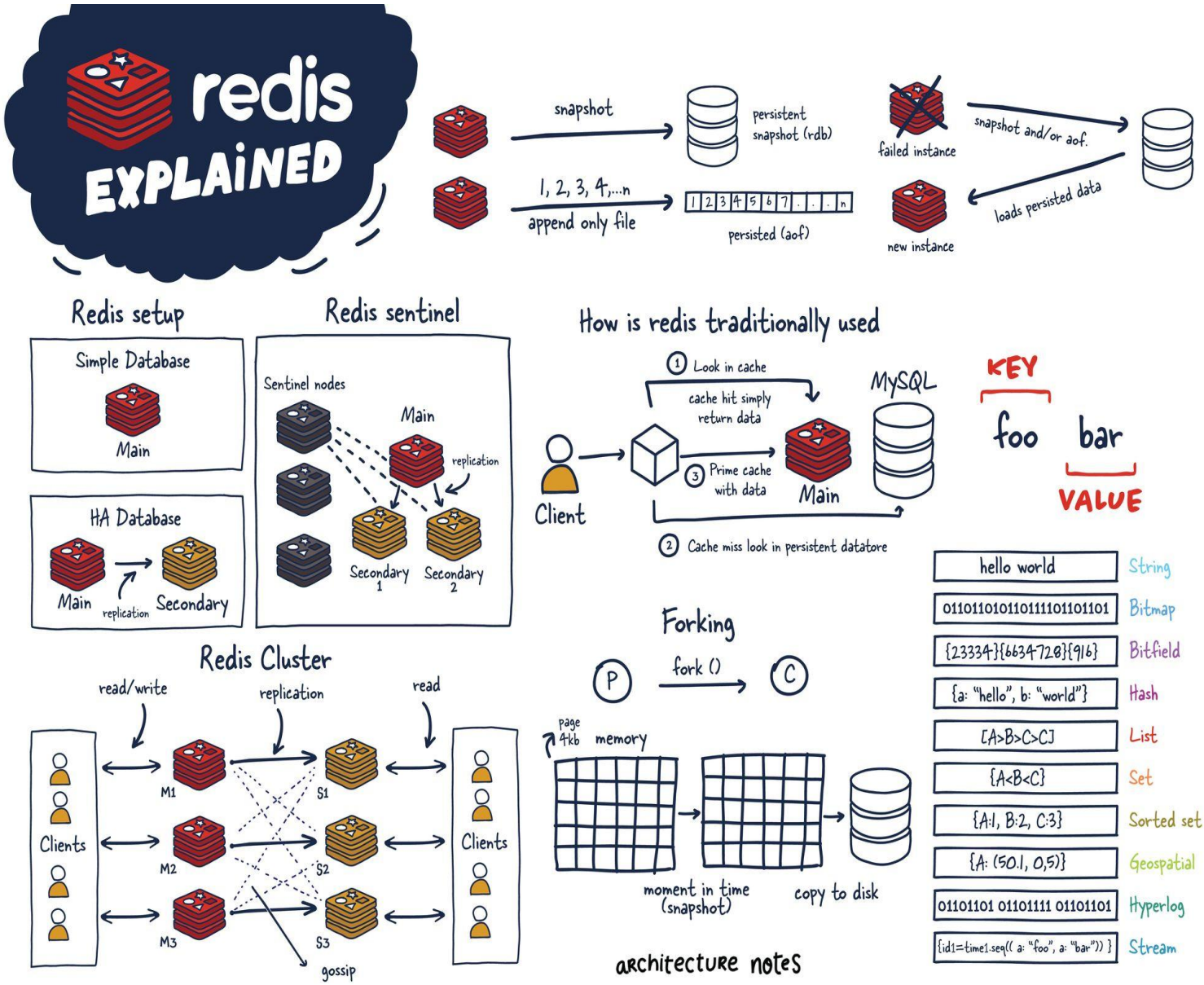
- **Cache hit = fast**
- **Cache miss = slow**
- **TTL = auto cleanup**
- **Invalidate when data changes**

Redis – Deep Dive (Easy explanation + real-life examples)



Redis vs Memcached

Feature	Redis	Memcached
Supported Data Types	strings, lists, sets, sorted sets, hashes, bit arrays, geospatial, and hyper logs	strings
Memory Management	Can store the details to disk when the physical memory is fully occupied.	in memory only. Supports saving into drive using an extension extstore
Data Size Limits	512 MB (for string values)	1 MB
Sub-milliseconds latency	Supports it	Supports it
Data Persistence	Supports using RDB snapshot and AOF Log persistence policies	Doesn't support it
Cluster Mode (Distributed caching)	Supports it	doesn't support tit. Can be achieved on the client-side using a ***consistent hashing***
Multi-Threading	Supports very well	Doesn't support multi-threading
Scaling	Supports horizontal scaling	Supports vertical scaling only (Horizontal scaling from the client side)
Data replication	Supports data replication out of the box.	Doesn't support
Supported Eviction Policies	Supports various types of policies https://docs.redislabs.com/latest/rc/concepts/data-eviction-policies/	Least recently Used (LRU)
Transaction Management	Supports it	Doesn't support
Pub/Sub	Supports it	Doesn't support



1️⃣ What is Redis?

Redis is an **in-memory data structure store**.

👉 **In-memory** means:

- Data is stored in **RAM**
- Not on disk (like traditional databases)

📌 RAM is **extremely fast** compared to disk.

2️⃣ Why Redis is so fast 🚀

Let's compare:

Storage	Speed
Disk (DB)	Slow
RAM (Redis)	Very fast

That's why:

- Database query → 300–800 ms
- Redis fetch → **5–20 ms**

👉 Redis is mainly used for **caching**

3️⃣ Big question ❓

"If Redis is so fast, why not store everything in Redis?"

Answer:

Because **RAM is limited and expensive**.

- Redis stores data in RAM
- RAM size is much smaller than disk
- If Redis stores too much data → **Out of Memory (OOM)**

💡 Similar to:

- LeetCode error: **Memory Limit Exceeded**

👉 That's why:

- **Database = source of truth**
- **Redis = temporary fast storage**

4️⃣ Redis vs Database (simple comparison)

Feature	Redis	Database
Storage	RAM	Disk
Speed	Very fast	Slower
Cost	Expensive	Cheaper
Data size	Small	Huge

Feature	Redis	Database
Use case	Cache	Permanent storage

5️⃣ Redis data model (Key–Value)

Redis stores data as:

key → value

Just like:

- DB → table → row
- Redis → key → value

📌 Values can be **different data types**

6️⃣ Redis key naming convention 🧠

Industry-standard way:

user:1

user:2:email

blog:all

blog:123

👉 Use : to separate hierarchy
👉 Makes debugging & maintenance easier

7️⃣ Common Redis data types (MOST used)

🔹 1. String

Used for:

- Cached API responses
- Counters
- Flags

Commands:

- SET key value
- GET key

- SET key value NX → only if key doesn't exist
- MGET key1 key2

📌 Most commonly used data type

🔹 2. List

Used for:

- Queues
- Stacks
- Streams

Commands:

- LPUSH → push left
- RPUSH → push right
- LPOP → pop left
- RPOP → pop right

Queue (FIFO)

LPUSH + RPOP

Stack (LIFO)

LPUSH + LPOP

🔹 Other data types (just awareness)

- Hash
- Set
- Sorted Set

(You'll learn these when needed)

8️⃣ Redis caching flow (Blog example)

First request → Cache MISS ❌

1. /blogs request comes
2. Redis has no data
3. Fetch from DB / API

4. Store result in Redis (TTL = 24 hrs)
5. Return response

Next requests → Cache HIT ✅

1. /blogs request comes
2. Redis returns data instantly
3. DB is not touched

📌 Massive performance boost 🚀

🧠 TTL (Time To Live) ⏰

TTL means:

How long data stays in Redis

Example:

- TTL = 24 hours
- Redis auto-deletes data after that

👉 Prevents:

- Stale data
- Memory overflow

🔁 Cache Update Strategy (important)

Another common approach:

Write DB and Redis at the same time

Example:

- Codeforces ranking
- User submits solution
- Update:
  - Database ✅
  - Redis ✅

📌 Ensures users always see latest data

1️⃣0️⃣ Cache Hit vs Cache Miss

- **Cache Hit** → data found in Redis → fast
- **Cache Miss** → data not in Redis → go to DB

Goal:

Maximize cache hits

1️⃣1️⃣ Where Redis is used in real systems

- API response caching
- Session storage
- Rate limiting
- Leaderboards
- Queues
- Counters (likes, views)

1️⃣2️⃣ Redis is language-agnostic

You can use Redis with:

- Node.js
- Spring Boot
- Django
- Go

👉 Redis logic stays same, only client library changes.

🔑 Interview-ready summary

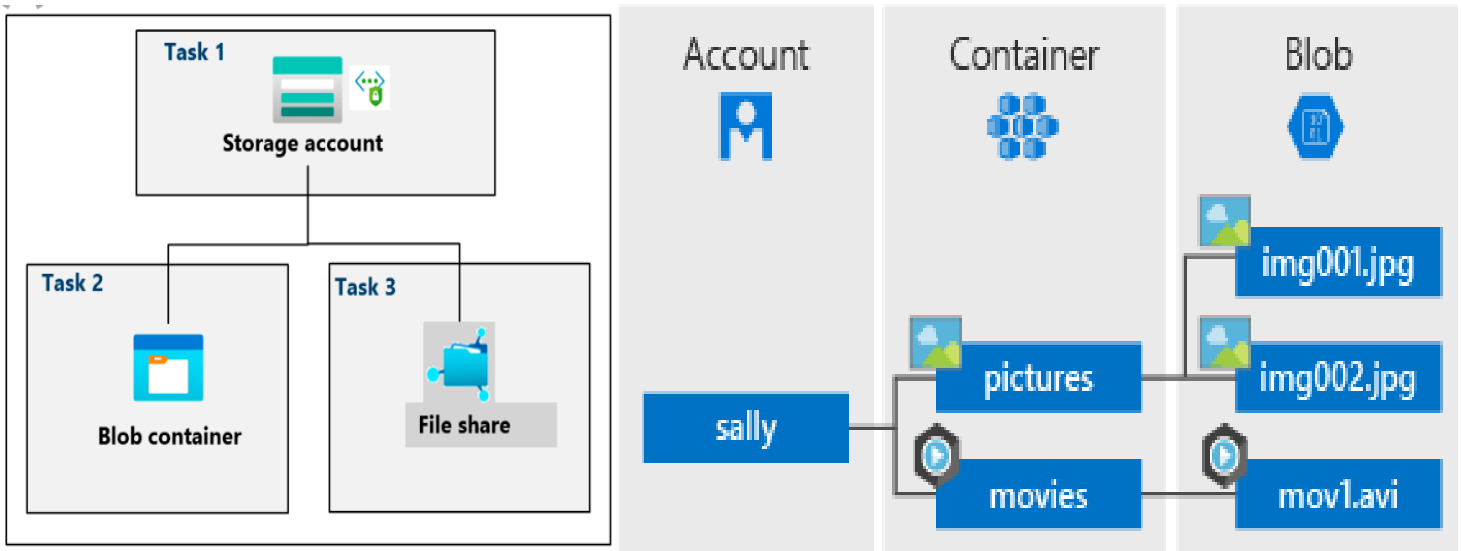
Redis is an in-memory key–value data store used primarily for caching and fast data access. It significantly reduces latency and backend load but cannot replace databases due to limited memory and volatile storage.

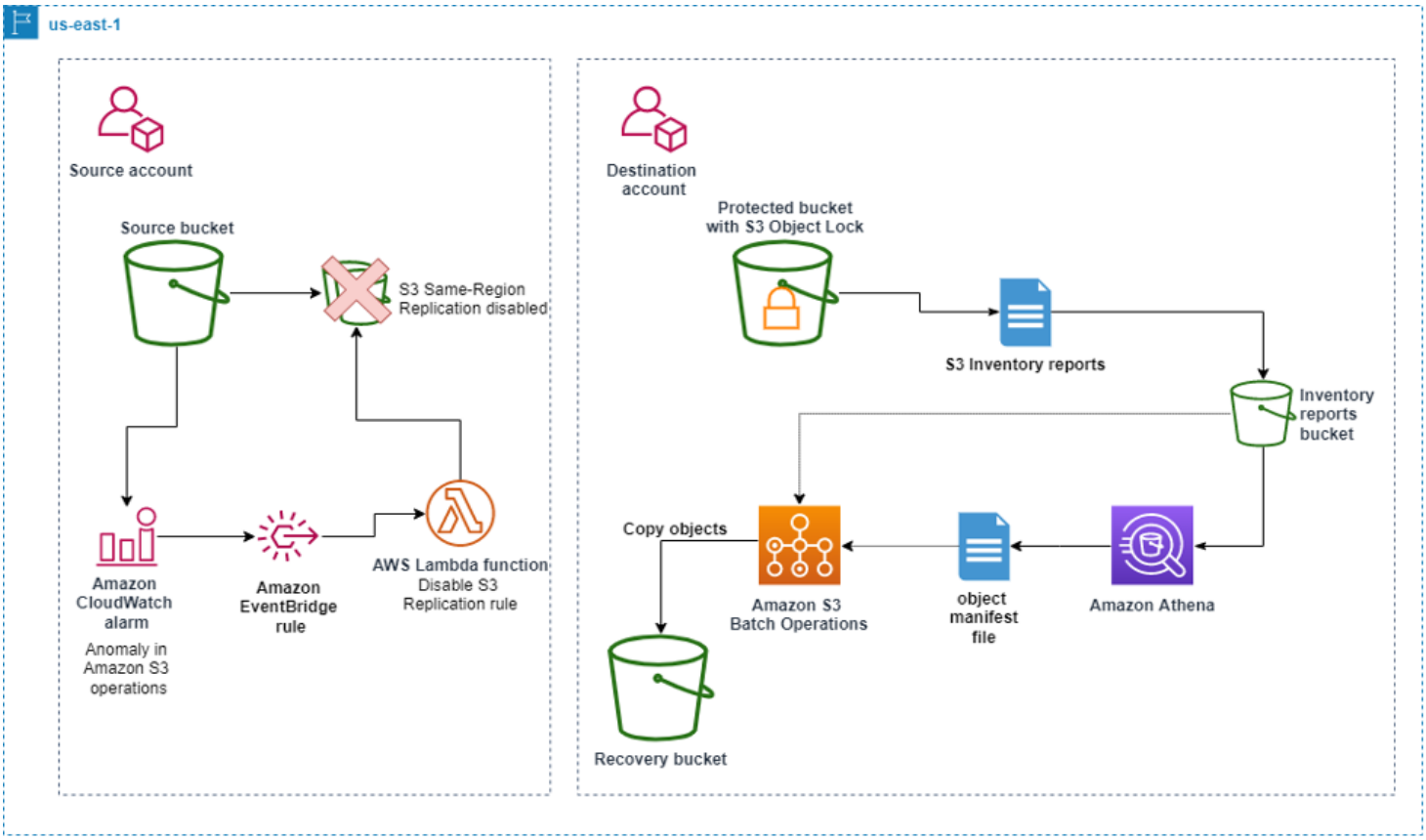
🧠 Memory tricks

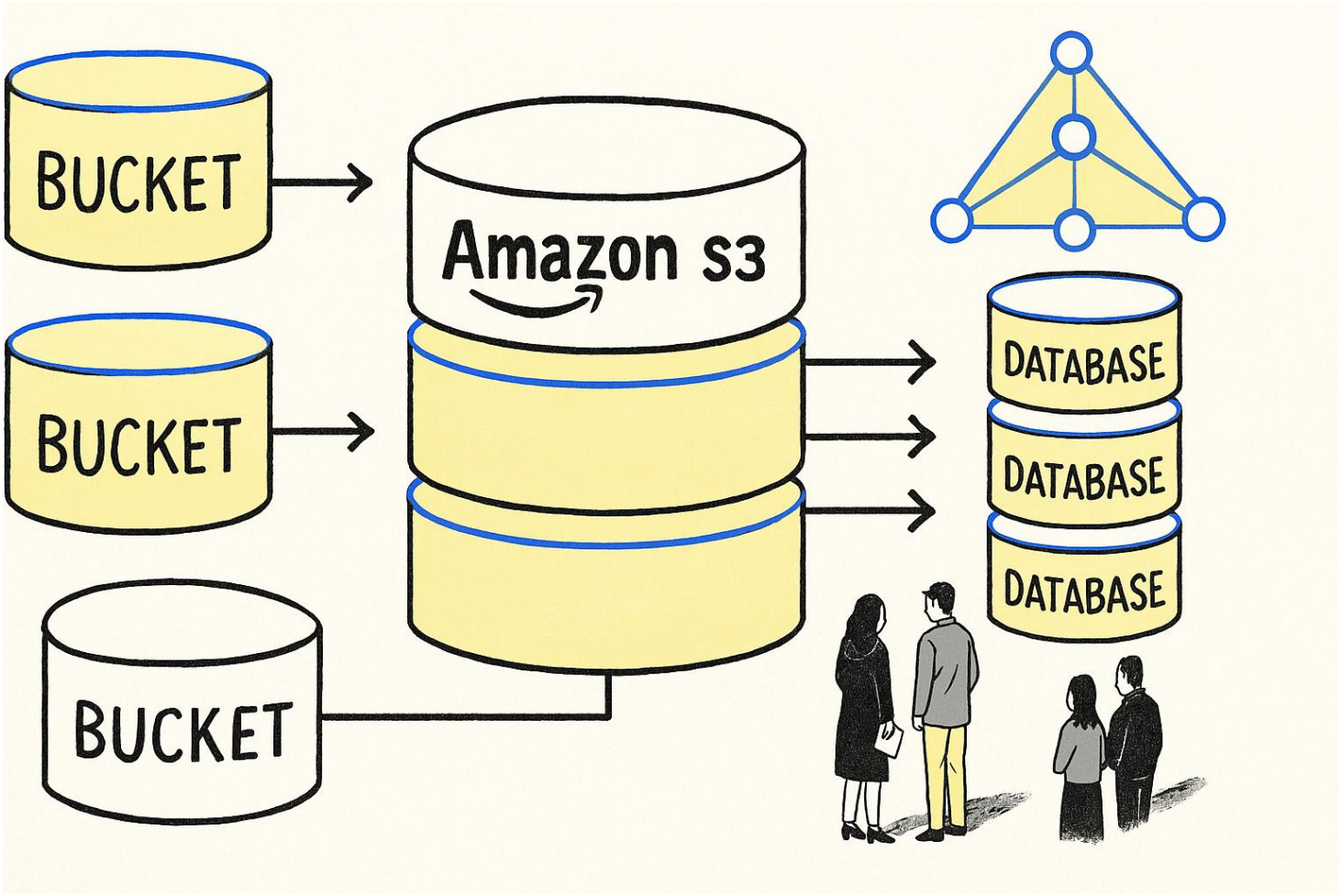
- **Redis = RAM = Fast**
- **DB = Disk = Slow**
- **Redis caches, DB stores**

- **TTL avoids stale data**

Blob Storage & CDN (Easy explanation with real-life examples)







1️⃣ What is a Blob, and why do we need Blob Storage?

Blob = **Binary Large Object**
Files like **mp4, png, jpeg, pdf** are stored as **0s and 1s** (binary). These files can be **very large** (MBs/GBs).

Why NOT store blobs in databases?

- Databases are optimized for **rows & columns** (text, numbers)
- Large blobs make queries **slow**
- Harder scaling, backups, and availability
- Much **more expensive** than object storage

👉 So we store blobs in **Blob Storage** (managed services that handle scaling, durability, security).

2️⃣ What is Blob Storage?

Blob Storage (aka **Object Storage**) stores files as **objects** with:

- **Key (path/name)**
- **Data (binary)**

- **Metadata**

Think of it like **Google Drive** for your app.

Popular Blob Storage services

- **Amazon S3**
- **Cloudflare R2**

3️⃣ Why S3 is widely used (high-level)

Amazon S3 is used to store **images, videos, PDFs, backups**, etc.

Key benefits

- **Scalability**: auto-scales
- **Durability**: **11 nines** (99.999999999%)
- **Availability**: high SLAs
- **Cost-effective**: cheaper than DB storage
- **Security**: encryption, IAM, bucket policies
- **Access control**: pre-signed URLs, fine-grained permissions

📌 Typical pattern:

- **DB** stores metadata (file name, URL, owner)
- **S3** stores the actual file

4️⃣ Typical upload/download flow

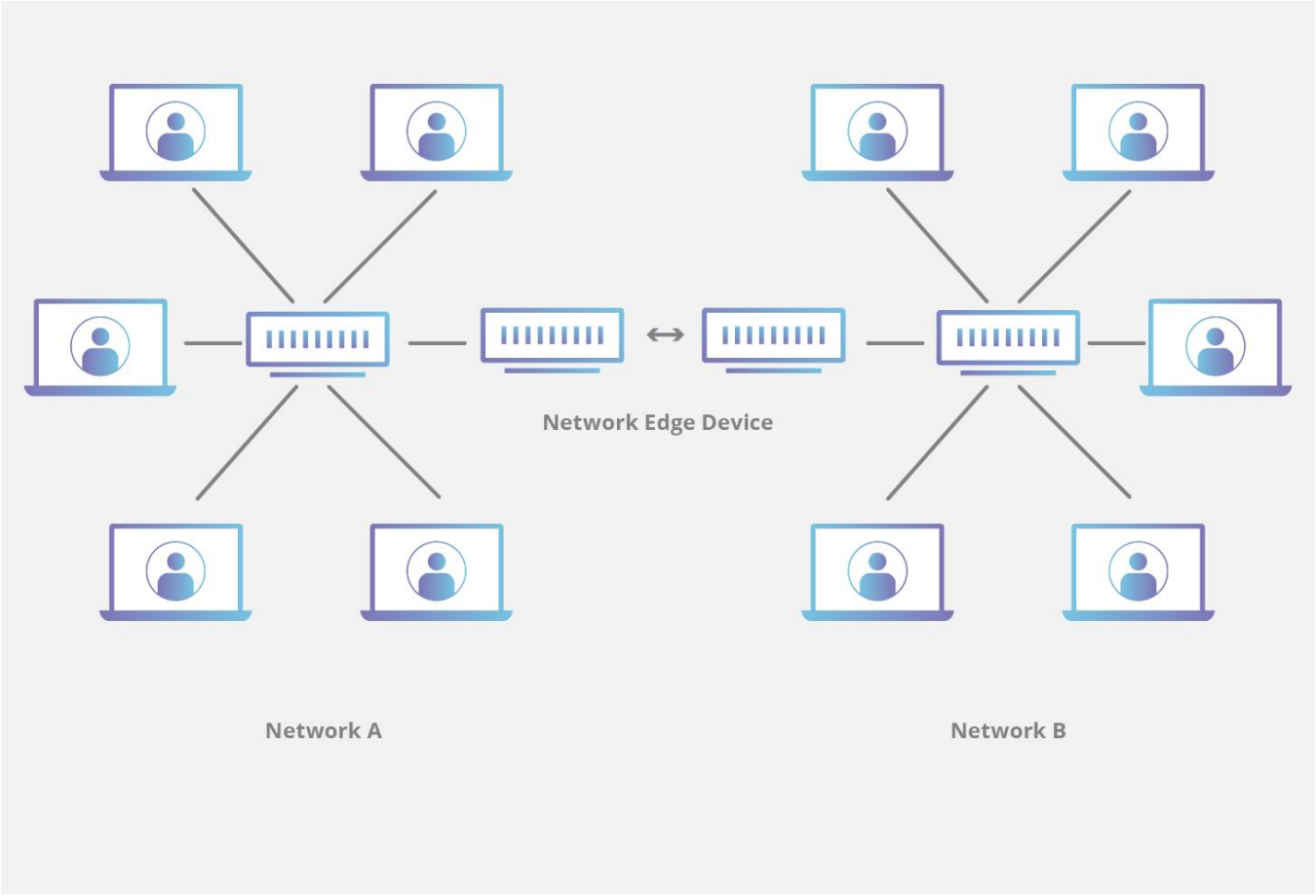
1. Client uploads file
2. Backend generates **pre-signed URL**
3. Client uploads **directly to S3**
4. Backend stores **S3 URL** in DB
5. Clients fetch file using the URL

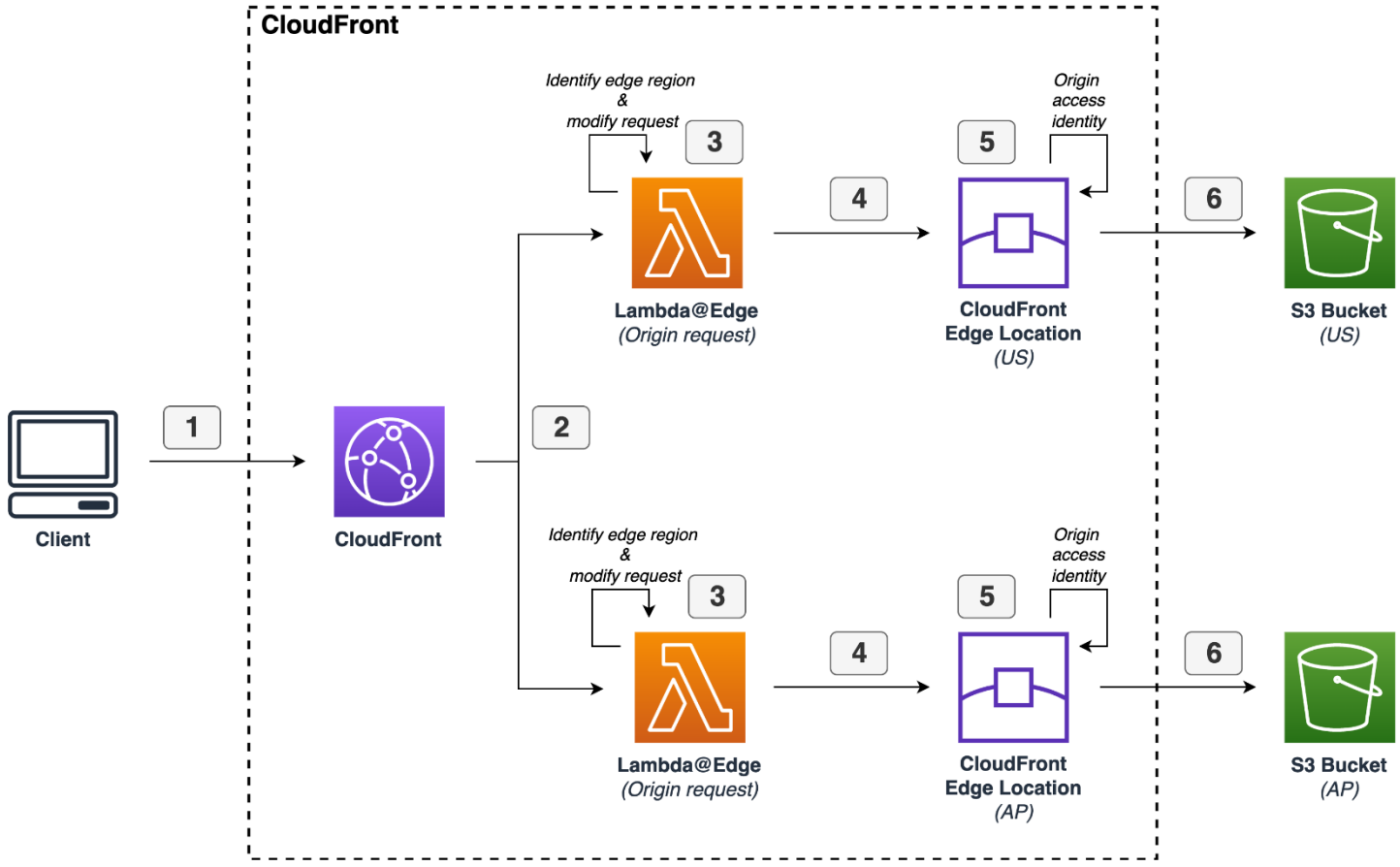
✅ Fast
✅ Secure
✅ Scales easily

5️⃣ Why we still need a CDN?

If your S3 bucket is in **India** and a user is in **USA**, fetching files from far away is **slow**.

Solution → **Content Delivery Network (CDN)**





6️⃣ What is a CDN?

A **CDN** is a network of **edge servers** distributed globally that **cache and serve static files** close to users.

Popular CDNs

- **AWS CloudFront**
- **Cloudflare**

7️⃣ How CDN works (step by step)

1. User requests a file
2. Request goes to **nearest edge server**
3. **Cache HIT** → file served instantly
4. **Cache MISS** → edge fetches from **Origin (S3)**, caches it, then serves
5. Next users get it from the edge (fast)

📌 Result: **low latency**, **lower origin load**, **better UX**

8️⃣ Key CDN concepts (simple)

- **Edge Server**: nearest server to the user
- **Origin Server**: your main storage (e.g., S3)
- **Caching**: store copies at the edge
- **TTL**: how long a file stays cached
- **GeoDNS**: routes users to nearest edge

9️⃣ Blob Storage vs CDN (clear roles)

Component	What it does
Blob Storage (S3)	Stores files reliably
CDN (CloudFront)	Delivers files fast globally

👉 **Use both together** for best results.

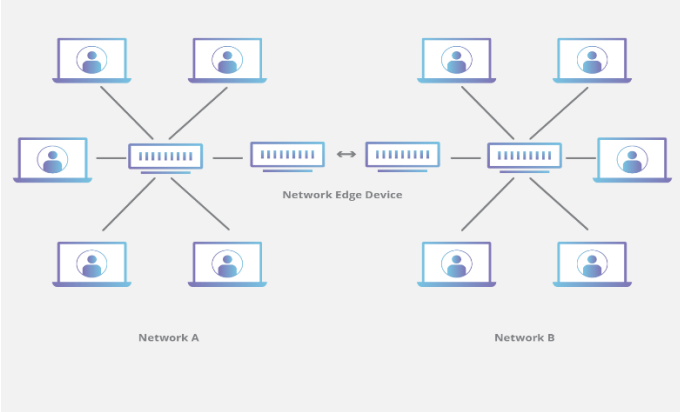
🔑 Interview-ready summary

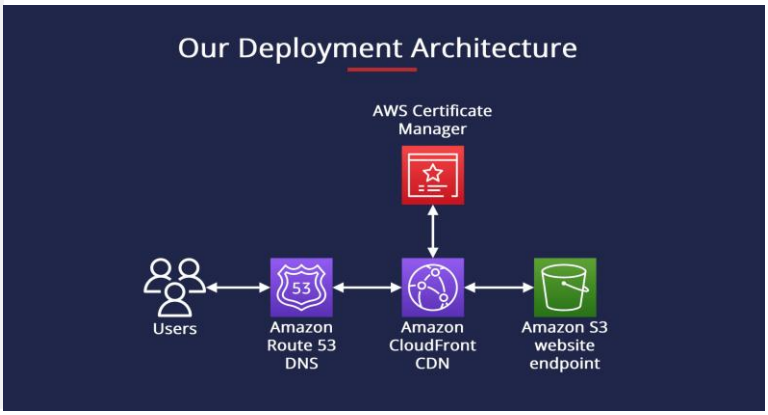
Blob storage stores large binary files efficiently outside databases, while CDNs cache and deliver those files from edge locations close to users, reducing latency and improving scalability.

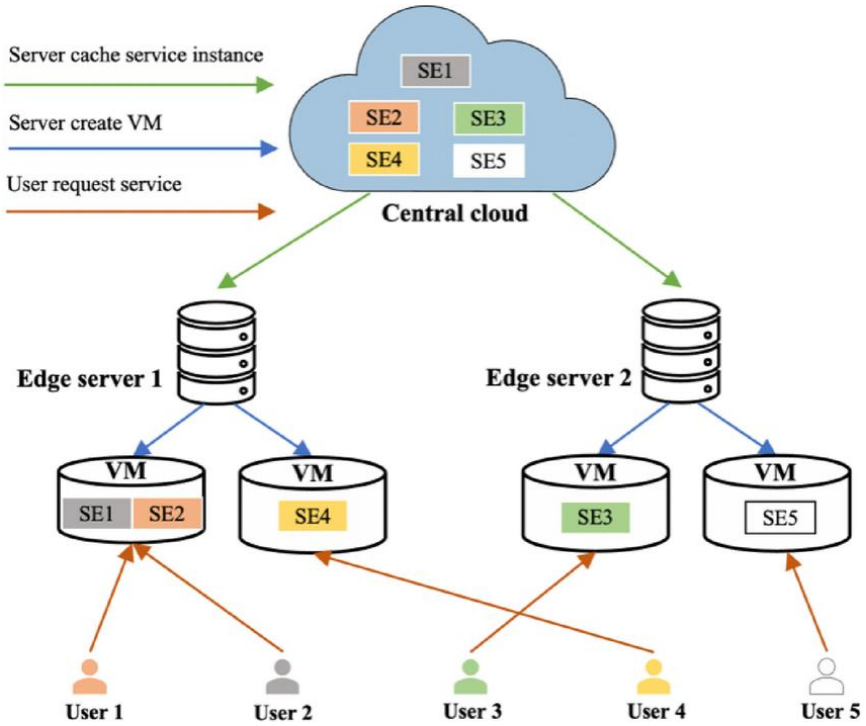
🧠 **Memory tricks**

- **DB = structured data**
- **Blob Storage = files**
- **CDN = speed near users**
- **S3 + CDN = best practice**

Content Delivery Network (CDN) — Easy explanation with real-life examples







1️⃣ What is a CDN? (in very simple words)

A **CDN (Content Delivery Network)** is:

A network of **servers placed all over the world** that **cache and deliver static files** (images, videos, PDFs, CSS, JS) **from the location closest to the user**.

📌 Closer server = **faster response**

2️⃣ Why do we need a CDN?

Imagine:

- Your files are stored in **India (S3)**
- User is in **USA or Australia**

Without CDN:

- Request travels across continents 🌍
- High latency (slow load)

With CDN:

- File is served from a **nearby server**
- Low latency (fast load)

👉 **Distance kills performance; CDN fixes it**

3️⃣ Real-life analogy 🏪

Think of a **warehouse**:

- One warehouse in Delhi ❌ (slow for US users)
- Many warehouses worldwide ✅

Customers get items from the **nearest warehouse**.

👉 Warehouse = **CDN Edge Server**

4️⃣ Popular CDN providers

- **AWS CloudFront**
- **Cloudflare**

(Used by almost every large website)

5️⃣ How does a CDN work? (step by step)

1️⃣ User requests a file (image/video)
2️⃣ Request goes to the **nearest edge server**
3️⃣ **Cache HIT**:

- File already present → served immediately ⚡

4️⃣ **Cache MISS**:

- Edge server fetches from **Origin Server** (e.g., S3)
- Caches it locally
- Returns file to user

5️⃣ Next users get it from the edge (fast)

6️⃣ Key CDN concepts (must know)

🔹 1. Edge Servers

- CDN servers distributed globally
- Cache your content
- Serve users from nearest location

🔹 2. Origin Server

- The **source of truth**
- Usually **S3**, web server, or blob storage
- Edge fetches data from here on cache miss

🔹 3. Caching

- Copies of files stored at edge servers
- Reduces:
  - Latency
  - Origin load
  - Bandwidth cost

🔹 4. TTL (Time To Live)

- How long a file stays cached at edge

Example:

- TTL = 24 hours
- After 24 hrs → edge refreshes file from origin

📌 Helps balance **freshness vs speed**

🔹 **5. GeoDNS**

- Routes users to the **nearest edge server**
- Based on geographic location

👉 User in USA → USA edge
👉 User in India → India edge

7️⃣ **What should be served via CDN?**

✅ Images (png, jpeg)
✅ Videos (mp4)
✅ PDFs
✅ CSS / JS
✅ HTML (static pages)

❌ Not ideal for highly dynamic, personalized data

8️⃣ **Benefits of using CDN**

- 🚀 Faster load times
- 🌍 Global reach
- 📉 Reduced server load
- 💸 Lower bandwidth cost
- 📈 Better scalability
- 😊 Better user experience

9️⃣ **CDN in a typical system design**

User → CDN (Edge) → Origin (S3)

- Most requests end at **CDN**
- Origin is hit only on cache miss

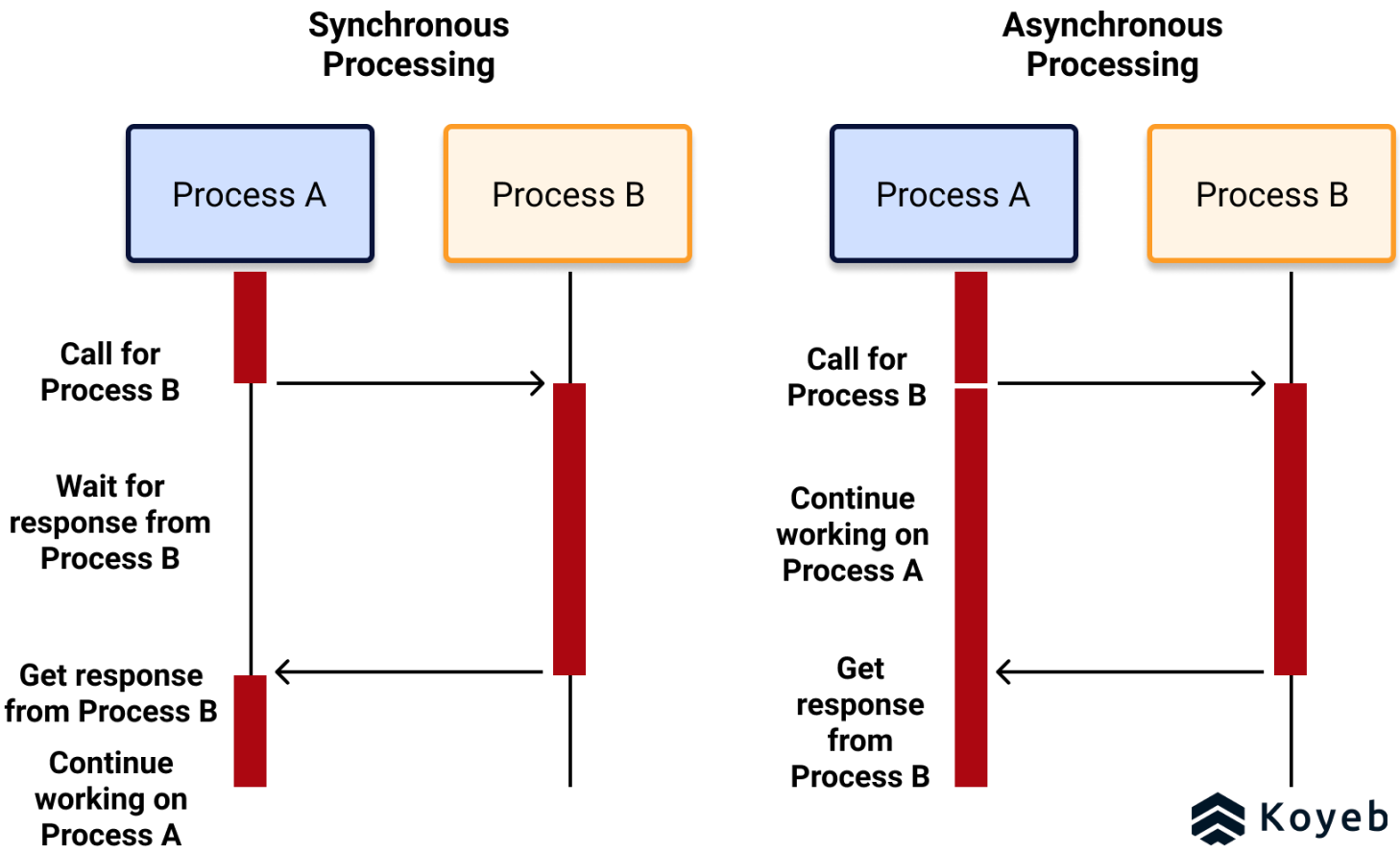
🔑 Interview-ready summary

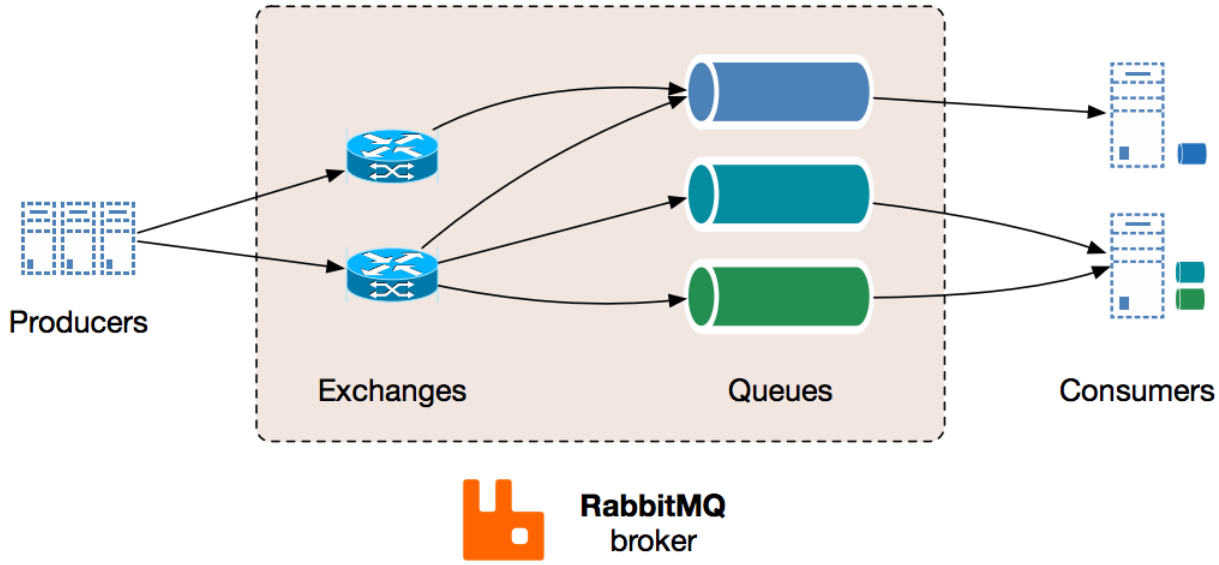
A CDN is a globally distributed network of edge servers that cache and deliver static content from locations closest to users, reducing latency, improving performance, and lowering origin server load.

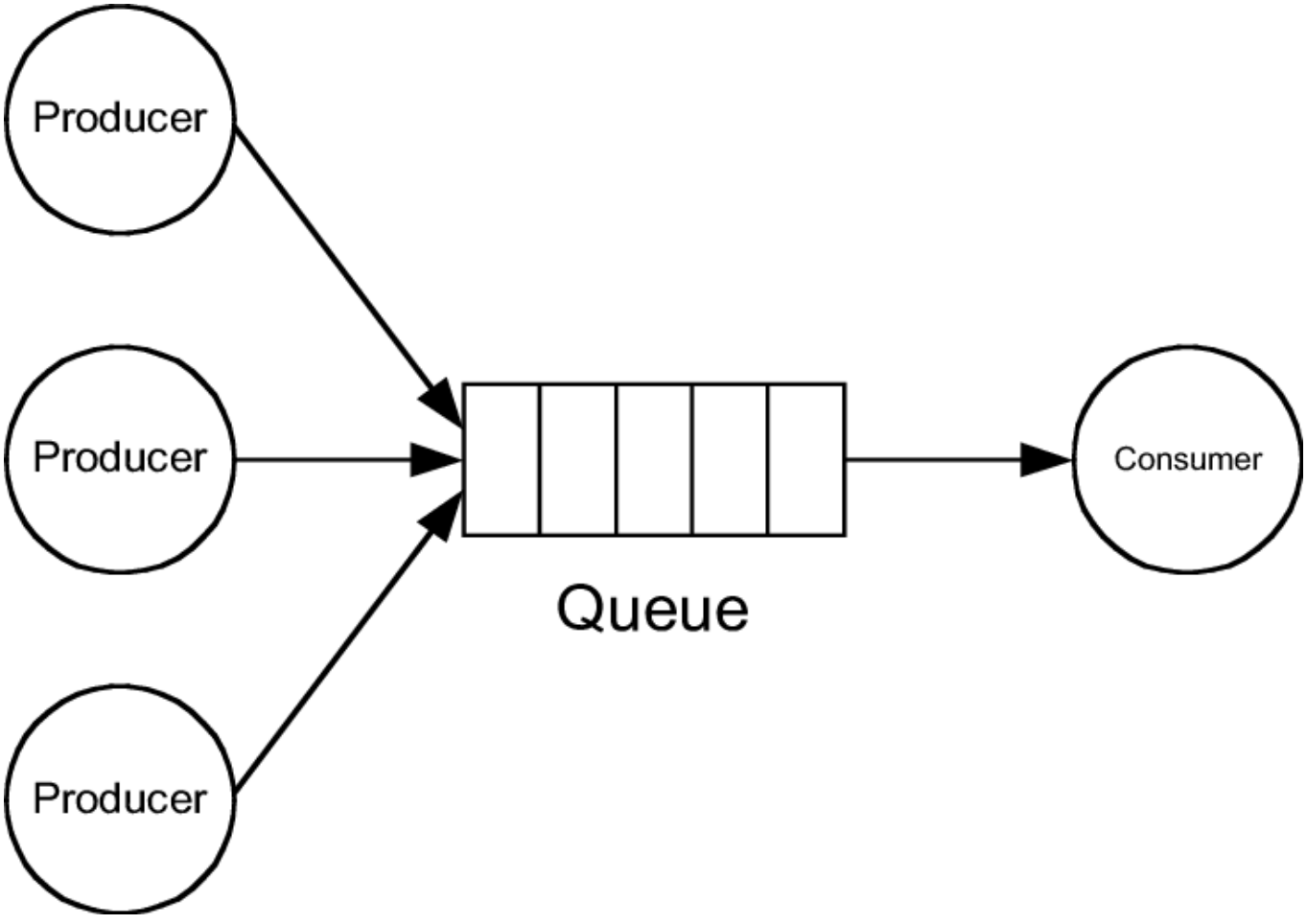
🧠 Memory tricks

- **CDN = speed**
- **Edge = nearest server**
- **Origin = S3**
- **TTL = cache lifetime**

Message Broker & Asynchronous Programming (Easy explanation + real-life examples)







---

1 Synchronous Programming (what you already know)

Synchronous means:

Client sends a request → server processes it → server sends the response **immediately**.

Example

- Login API
- Fetch profile
- Get product list

📌 These are **fast operations** (milliseconds).

2️⃣ Why synchronous fails for long tasks ❌

Suppose a task takes **10 minutes**:

- Video processing
- Sending thousands of emails
- Generating reports
- Image compression

Problems with sync:

- Client waits too long 😴
- HTTP request times out
- Server threads are blocked

👉 **Bad user experience + poor scalability**

3️⃣ Asynchronous Programming (the solution ✅)

Asynchronous means:

Client does **not wait** for the task to finish.

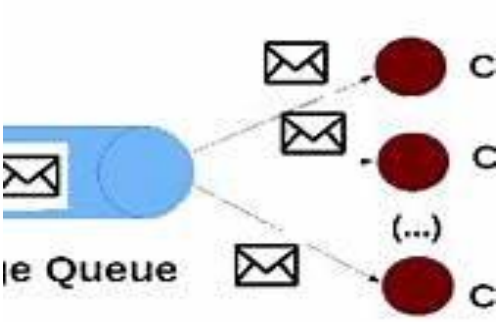
Flow:

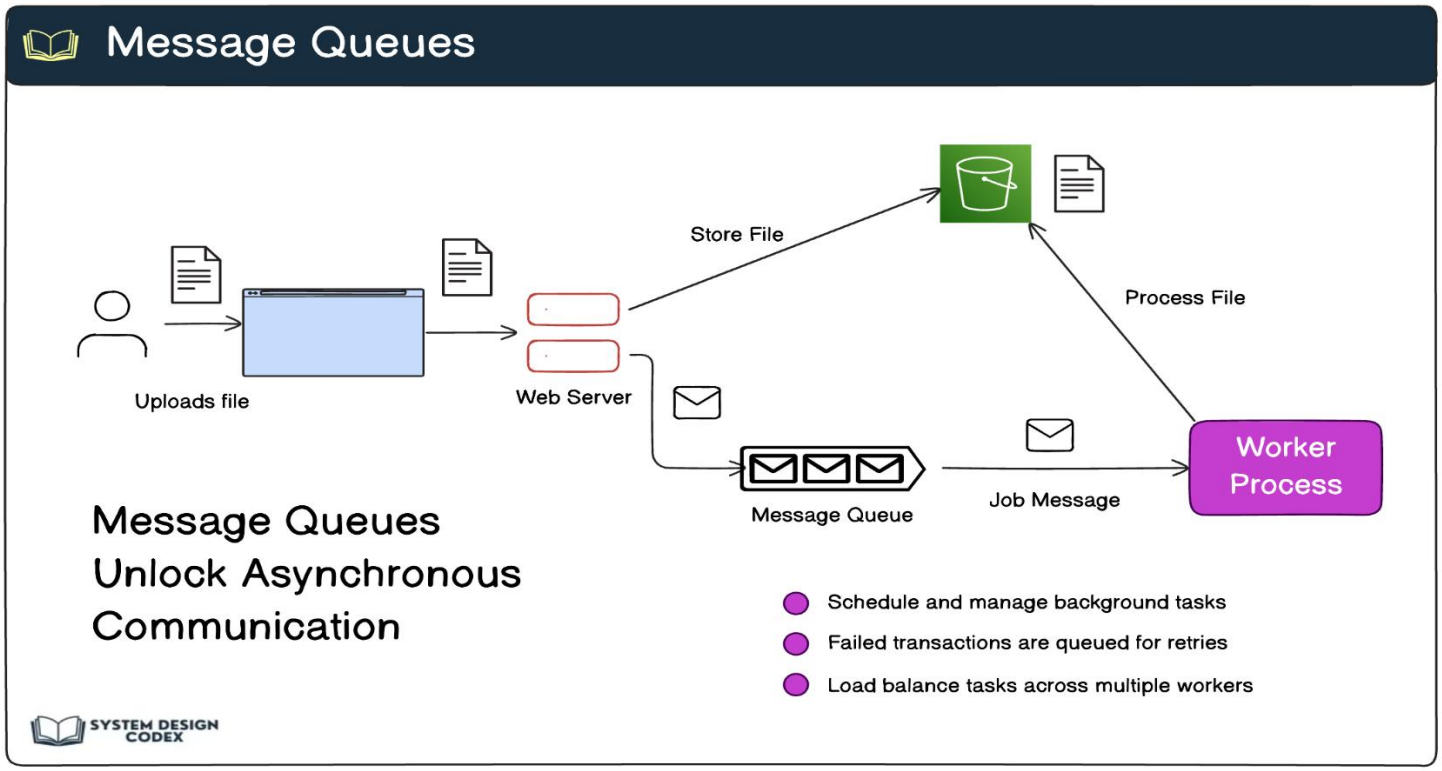
1. Client sends request
2. Server replies immediately:
3. "Your task is being processed"
4. Task runs **in background**
5. Client is notified later (email / notification / webhook)

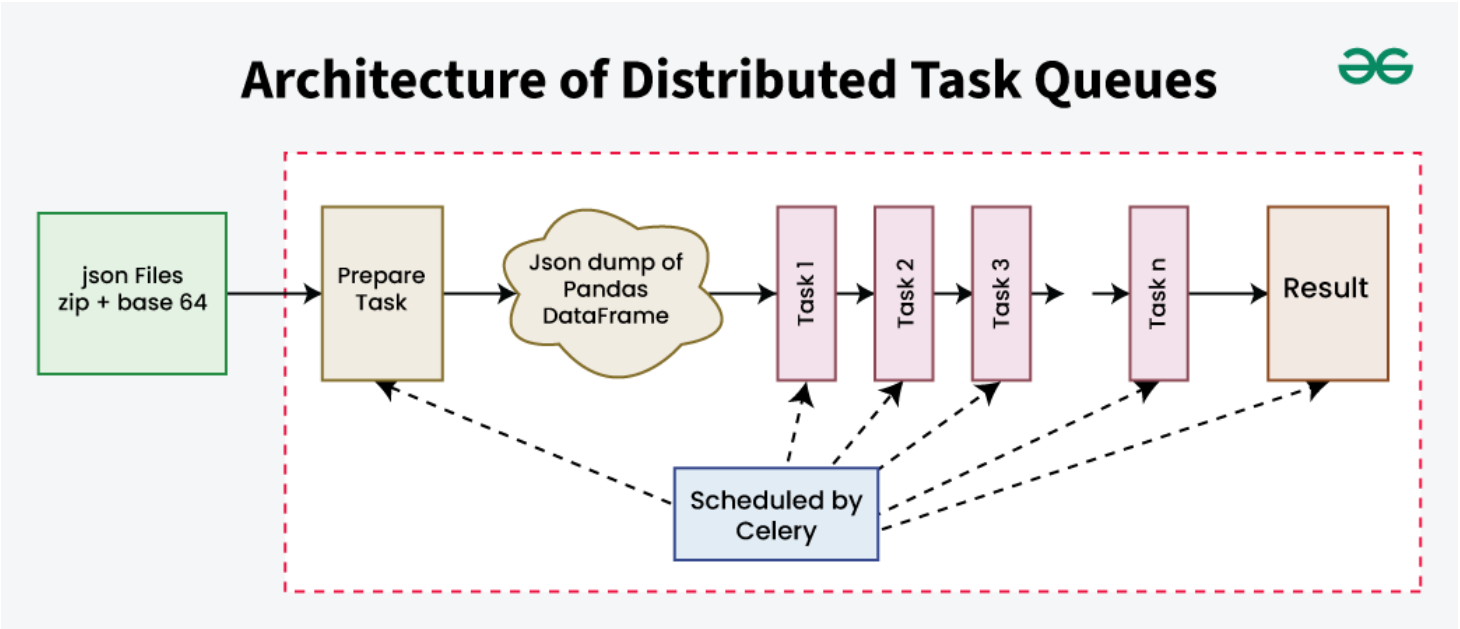
📌 Client is free, server is safe.

4️⃣ Why we need a Message Broker

Instead of directly sending tasks to workers, we add a **Message Broker** in between.







Components

- **Producer** → puts task/message into broker

- **Message Broker** → stores tasks reliably
- **Consumer (Worker)** → pulls tasks and processes them

5️⃣ Simple real-life analogy 🧻

Think of a **restaurant**:

- Customer places order → **Producer**
- Order slip on counter → **Message Broker**
- Chef picks slip and cooks → **Consumer**

If chef is busy:

- Orders wait safely on counter
- No order is lost ✅

6️⃣ Why put Message Broker in between? (VERY important 🔥)

✅ 1. Reliability

- Producer can crash
- Messages stay safe in broker
- Workers continue processing

✅ 2. Retry mechanism

- Worker fails mid-task ❌
- Message is **not deleted**
- Worker retries later

📌 Very useful for unstable tasks

✅ 3. Decoupling

- Producer doesn't care **who** processes the task
- Consumer doesn't care **who** created the task
- Both work independently

👉 Loose coupling = scalable systems

7️⃣ Producer vs Consumer (clear definition)

- **Producer**: creates & sends messages
- **Consumer / Worker**: reads & processes messages

They communicate **only through the broker**.

8️⃣ Types of Message Brokers

🔹 1. Message Queues

(Messages are processed **once**)

Examples:

- **RabbitMQ**
- **Amazon SQS**

Use cases

- Background jobs
- Email sending
- Payment processing
- Order fulfillment

📌 FIFO / at-least-once processing

🔹 2. Message Streams

(Messages are **stored & replayable**)

Examples:

- **Apache Kafka**
- **Amazon Kinesis**

Use cases

- Event streaming
- Analytics
- Logs
- Real-time dashboards

📌 Multiple consumers can read same message

9️⃣ Queue vs Stream (quick comparison)

Feature	Message Queue	Message Stream
Message usage	Once	Replayable
Consumers	One	Many
Order	Optional	Strict
Use case	Tasks	Events

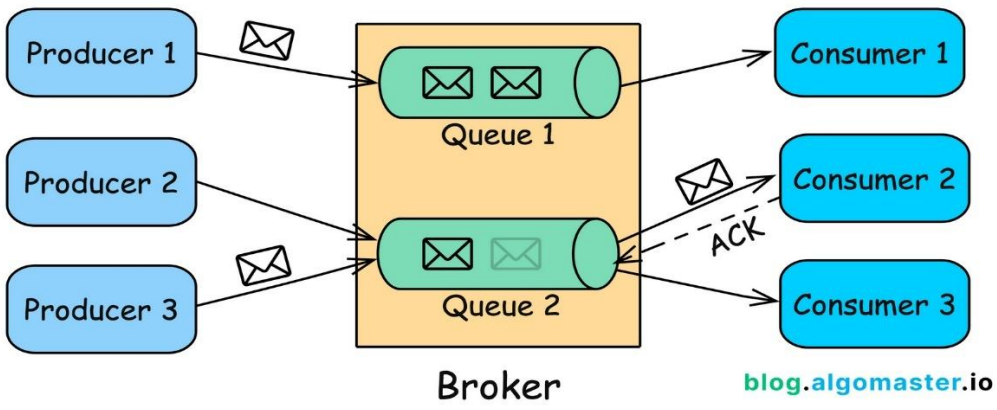
🔑 Interview-ready summary

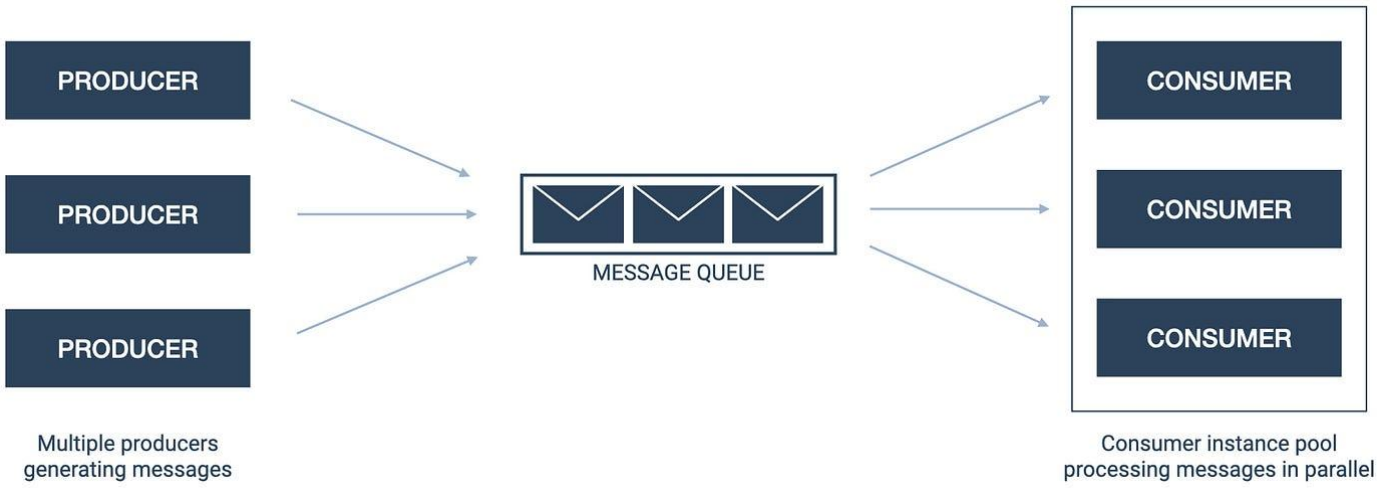
A message broker enables asynchronous communication by decoupling producers and consumers. It allows long-running tasks to be processed in the background with reliability, retries, and scalability using queues or streams.

🧠 Memory tricks

- **Sync** → wait & block
- **Async** → fire & forget
- **Producer** → puts message
- **Consumer** → processes message
- **Queue** → jobs
- **Stream** → events

Message Queue (Easy explanation with real-life examples)





1️⃣ What is a Message Queue?

A **Message Queue** is exactly what the name says:

A **queue** where a **producer** puts messages, and **consumers (workers)** pull messages and process them.

- Producer → puts message
- Queue → stores message safely
- Consumer → pulls, processes, deletes message

📌 Messages are usually processed **once**.

2️⃣ Message Queue vs Message Stream (quick clarity)

The **main difference** you mentioned is correct:

- **Message Queue** → one message is processed by **one consumer**
- **Message Stream** → same message can be read by **multiple consumers**

👉 Message Queue = **job/task processing**

3️⃣ Real-world example 🎥 (Video Transcoding)

Imagine a **video platform**:

Step-by-step flow:

1. User uploads a video
2. Backend creates **metadata** (video path, format, user ID)

3. Backend puts this metadata into **Message Queue**
4. **Video Transcoder Service (Consumer)**:
    - Pulls message
    - Fetches video
    - Converts it to 480p, 720p, 1080p
5. After completion → **message is deleted**

- 📌 Message deletion means:
  ✓ Task completed successfully

4️⃣ Why delete messages?

Message queues like:

- **RabbitMQ**
- **Amazon SQS**

provide APIs to:

- Acknowledge message
- Delete message after processing

If message is **not deleted**:

- Queue assumes task failed
- Message can be **retried**

5️⃣ What if one consumer is not enough? 🤔

Then we **scale consumers horizontally**.

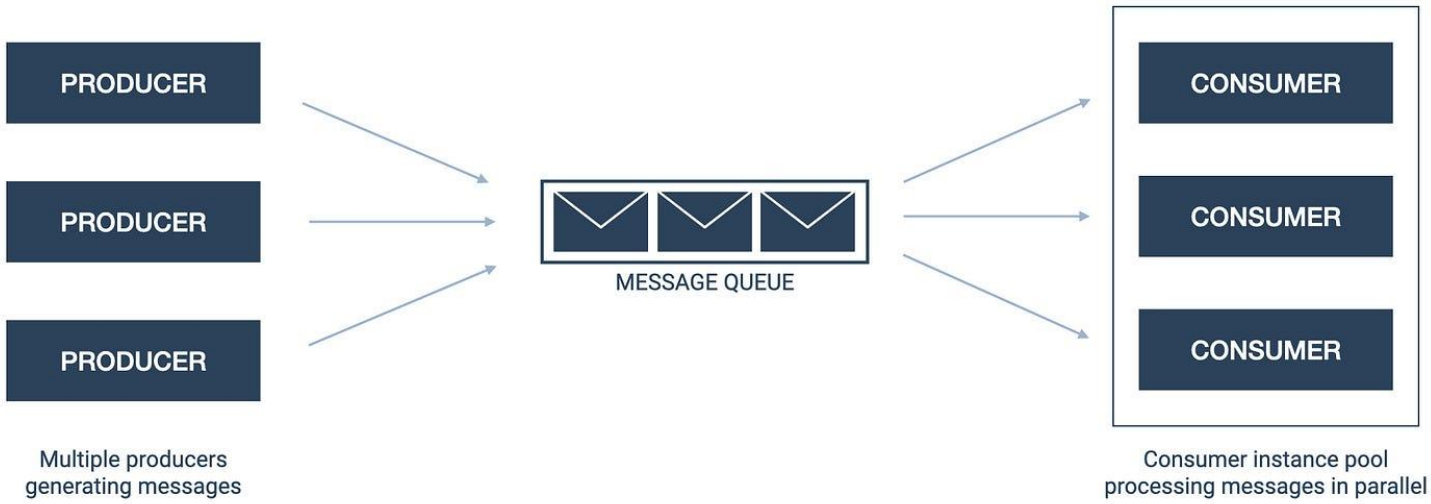
Instead of:

- 1 transcoder ❌

We run:

- 3 transcoders ✅
- 10 transcoders during peak load 🚀

6️⃣ Parallel processing (big advantage 🔥)



Example:

- Queue has 100 messages
- 3 consumers running

Processing:

- Consumer 1 → Message A
- Consumer 2 → Message B
- Consumer 3 → Message C

📌 All run **in parallel**
📌 Faster processing
📌 Better throughput

7️⃣ Important behavior of Message Queue

- Each message is processed by **only one consumer**
- Consumers **compete** for messages
- Free consumer picks next message

👉 This is called **Competing Consumers Pattern**

8️⃣ Why Message Queue is powerful 💪

✅ Reliability

- Messages are stored safely

- If consumer crashes → message is retried

✅ **Scalability**

- Add more consumers anytime
- No change needed in producer

✅ **Decoupling**

- Producer doesn't care who processes
- Consumer doesn't care who produced

🚦 **Typical use cases of Message Queues**

- Video processing
- Email sending
- Payment processing
- Order fulfillment
- Background jobs
- Image compression

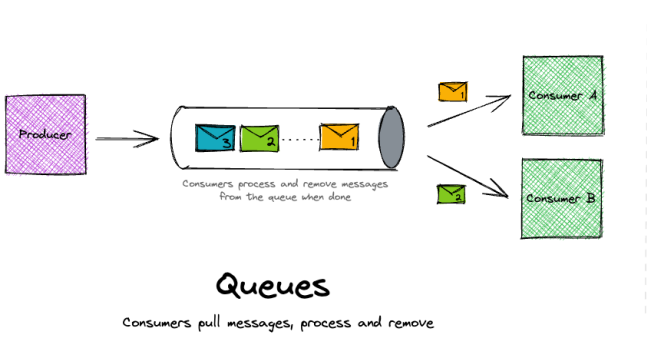
🔑 **Interview-ready summary**

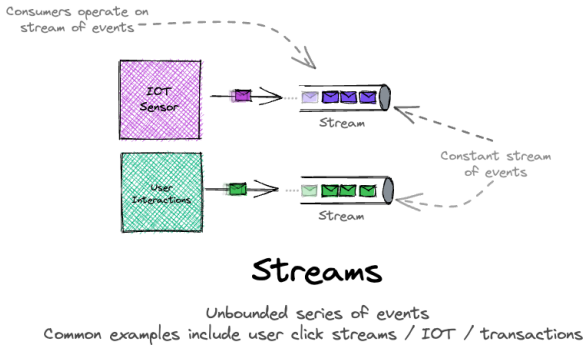
A message queue is a communication mechanism where producers send tasks as messages to a queue and consumers process them asynchronously. Each message is handled by only one consumer, enabling reliable, parallel, and scalable background processing.

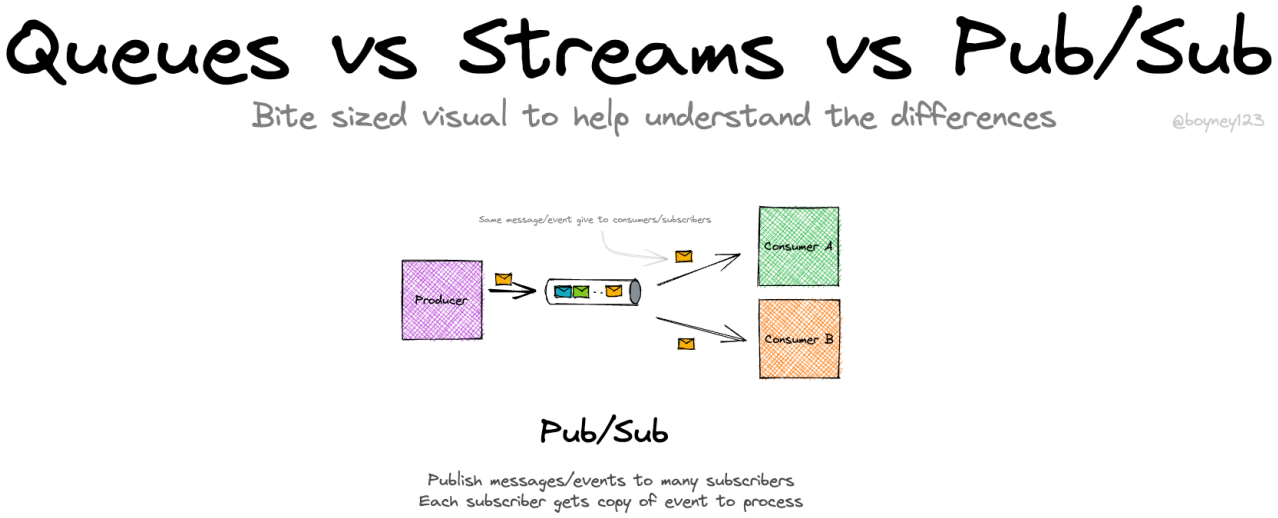
🧠 **Memory tricks**

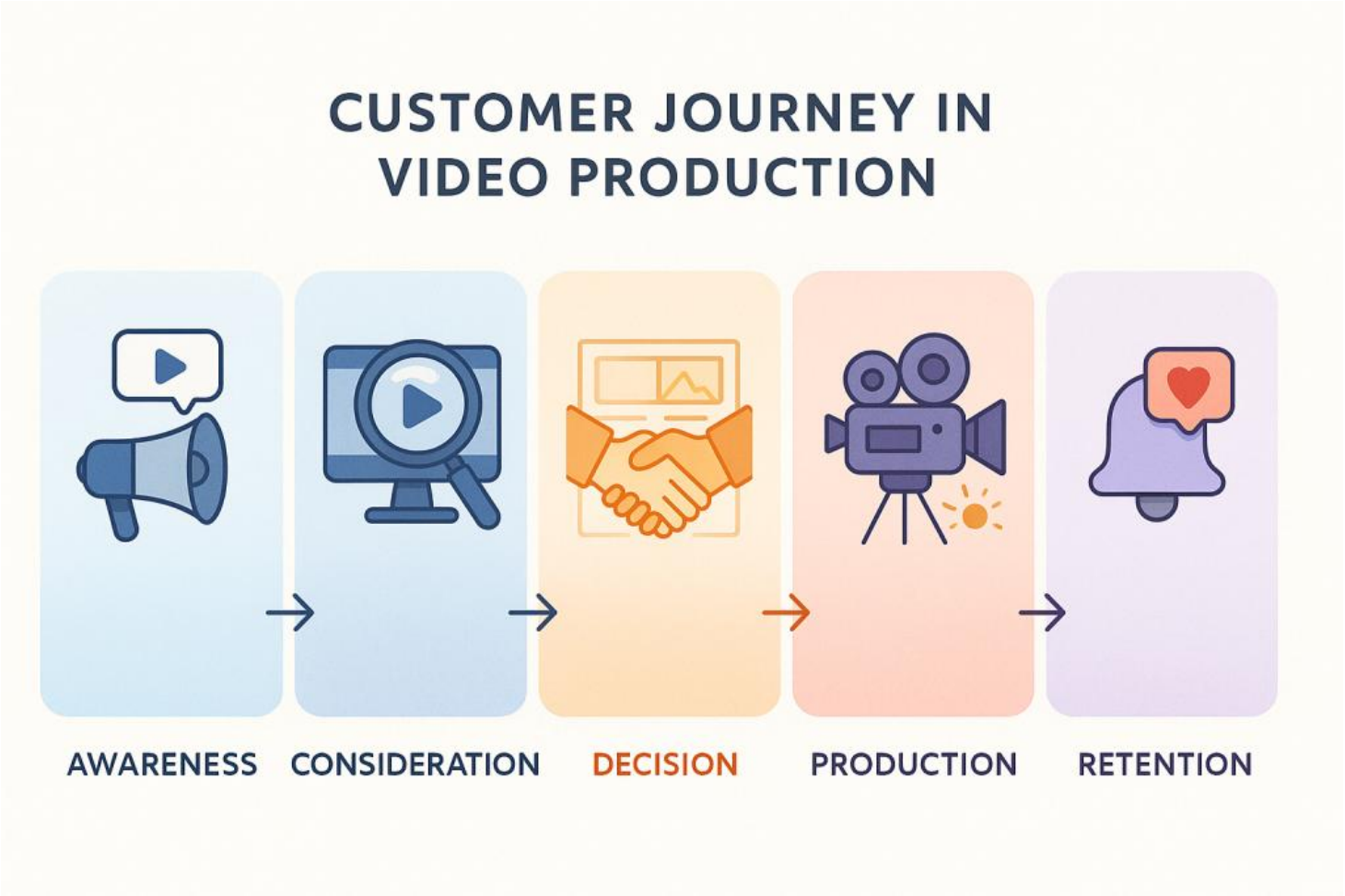
- **Queue = tasks**
- **One message → one consumer**
- **Delete message = task done**
- **More consumers = faster processing**

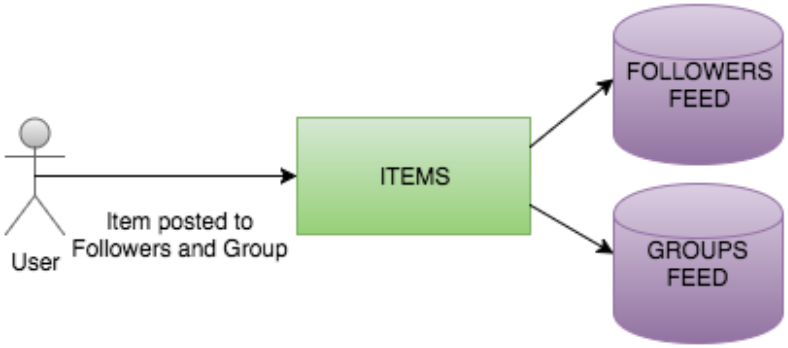
Why do we need a Message Stream? (Easy explanation with real-life examples)

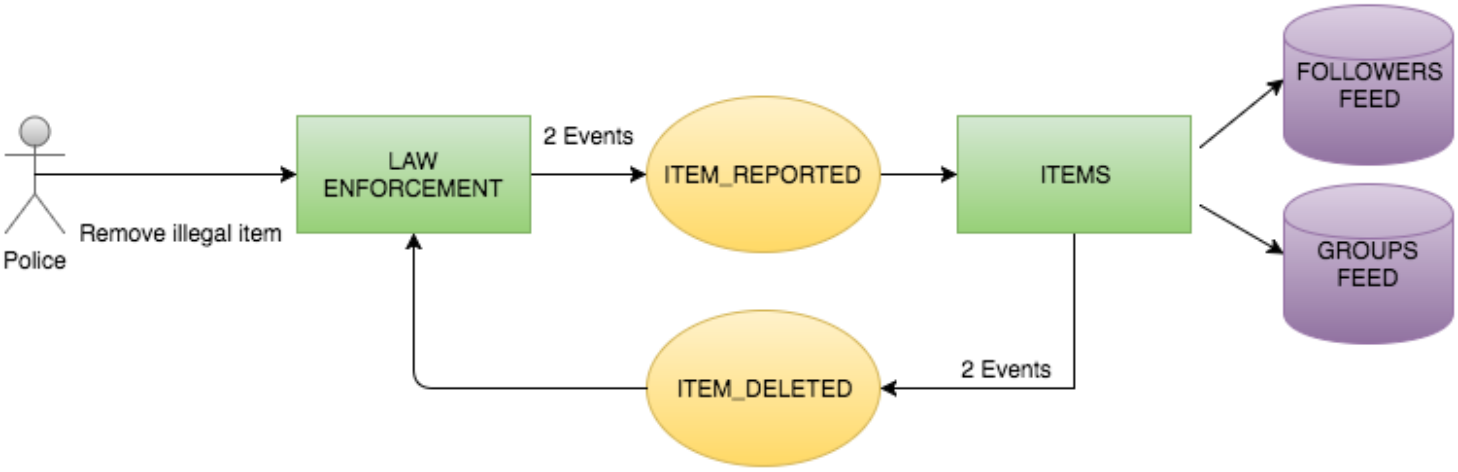


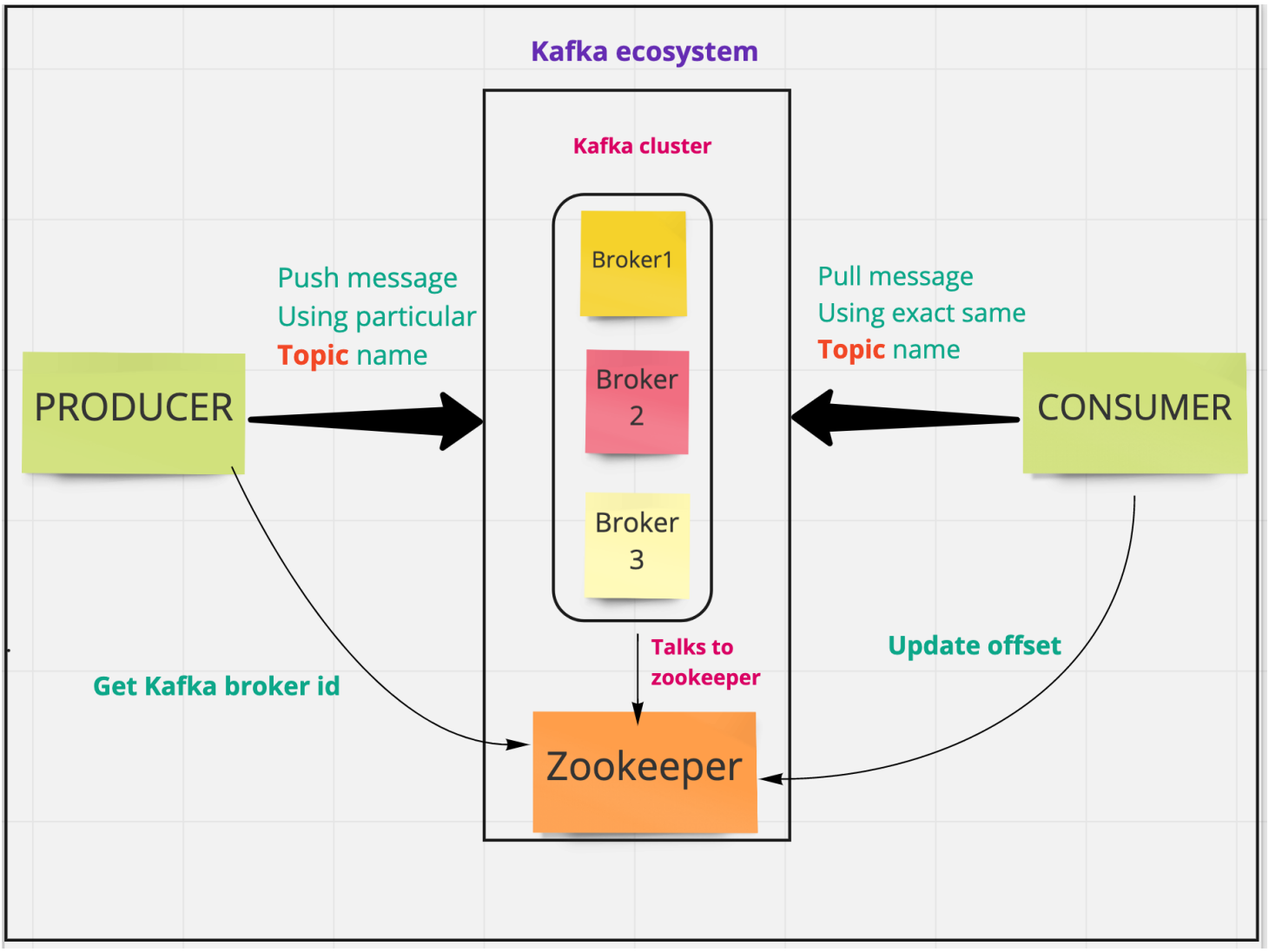







FOLLOWERS FEED
ITEMS
GROUPS FEED
User
Item posted to Followers and Group


FOLLOWERS FEED
LAW ENFORCEMENT
2 Events
ITEM_REPORTED
ITEMS
GROUPS FEED
Police
Remove illegal item
ITEM_DELETED
2 Events


Kafka ecosystem
Kafka cluster
Broker1
Broker 2
Broker 3
Push message
Using particular
Topic name
Pull message
Using exact same
Topic name
PRODUCER
CONSUMER
Talks to zookeeper
Update offset
Get Kafka broker id
Zookeeper

1️⃣ The problem with Message Queues (your video example)

You already understood **message queues** well. Now let's see **where they fail**.

Scenario 🎥

After a video upload, you want to:

- ✅ Transcode video (480p, 720p)
- ✅ Generate captions
- ✅ (Maybe later) Generate thumbnails

Naive idea ❌

Use **two message queues**:

- Queue A → Video Transcoder
- Queue B → Caption Generator

Problem:

- Producer writes to Queue A
- Before writing to Queue B → producer crashes 💥

👉 Result:

- Video is transcoded
- Captions are **never generated**
- ❌ **Inconsistent system state**

This is **very dangerous** in distributed systems.

2️⃣ Why Message Streams solve this problem

A **Message Stream** follows this principle:

Write once → Read by many

Instead of:

- One message → one consumer (queue)

We have:

- One message → **multiple independent consumers**

3️⃣ What is a Message Stream?

A **Message Stream** is:

An append-only log of events where **multiple consumers can independently read the same message**.

Examples:

- **Apache Kafka**
- **Amazon Kinesis**

4️⃣ How Message Stream works (step by step)

Video upload example 🎬

1. Video Upload Service writes **video metadata** to the stream
2. Stream stores it as:
3. video-1 metadata
4. video-2 metadata
5. video-3 metadata

Consumers:

- Video Transcoder Service
- Caption Generator Service
- Thumbnail Generator Service

Each service:

- Reads messages **at its own pace**
- Maintains its **own offset (position)**

📌 **No one deletes the message**

5️⃣ Important confusion: "If messages aren't deleted, won't they be processed again?" 🤔

No ❌

Because:

- Each consumer keeps track of **where it last read**
- This position is called an **offset**

Example:

- Transcoder read till video-5
- Caption service read till video-3

👉 Both are independent
👉 No duplicate processing per consumer

6️⃣ Key difference: Queue vs Stream (CRUCIAL 🔥)

Feature	Message Queue	Message Stream
Message usage	One consumer	Many consumers
Deletion	Yes (after processing)	No
Replay	No	Yes
Consumers	Competing	Independent
Best for	Tasks	Events

7️⃣ Real-life analogy 📺

Message Queue

- One TV
- One remote
- One person watches at a time

Message Stream

- YouTube live stream
- Millions watch **same event**
- Everyone joins independently

8️⃣ When should you use Message Streams?

Use **Message Streams** when:

- Multiple services need **same event**
- You want **fan-out** behavior
- You need **event replay**
- You want strong decoupling

Examples:

- Video processing pipelines
- User activity tracking

- Analytics & logging
- Event-driven microservices

9️⃣ When do we use Message Brokers in general?

Between microservices, you usually choose between:

- **REST APIs** (synchronous)
- **Message Brokers** (asynchronous)

Use Message Brokers when:

✅ Task is **non-critical**

- Email notifications
- Logs

✅ Task is **time-consuming**

- Video transcoding
- PDF generation
- ML processing

1️⃣0️⃣ Queue vs Stream (final decision rule 🧠)

- **One job → one worker** → Message Queue
- **One event → many consumers** → Message Stream

🔑 Interview-ready summary

Message streams are used when a single event must be consumed by multiple independent services. Unlike message queues, messages are not deleted after consumption, enabling fan-out, replayability, and strong decoupling across distributed systems.

🧠 Memory tricks

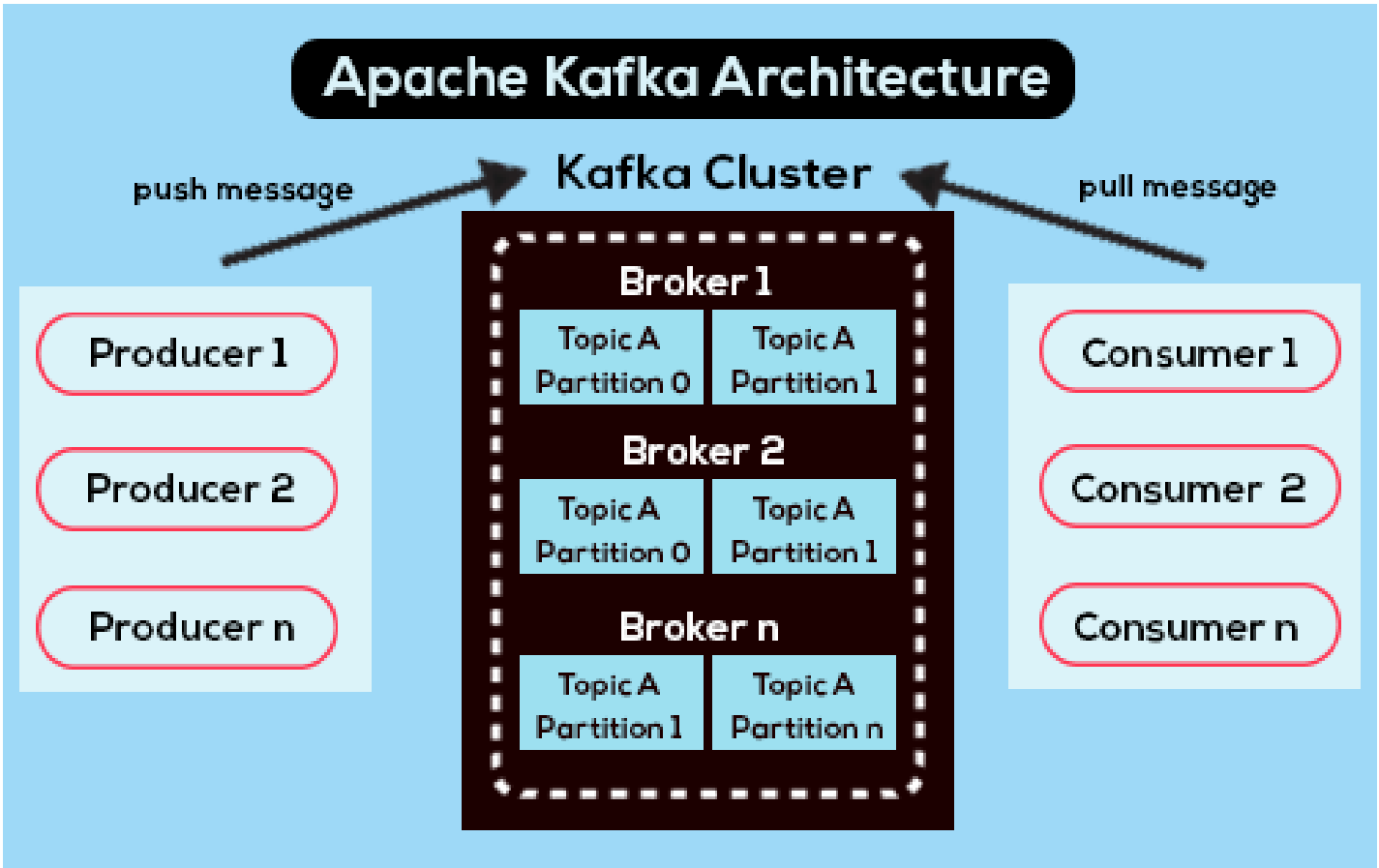
- **Queue → Jobs**
- **Stream → Events**
- **Delete → Queue**
- **Replay → Stream**
- **Write once, read many → Stream**

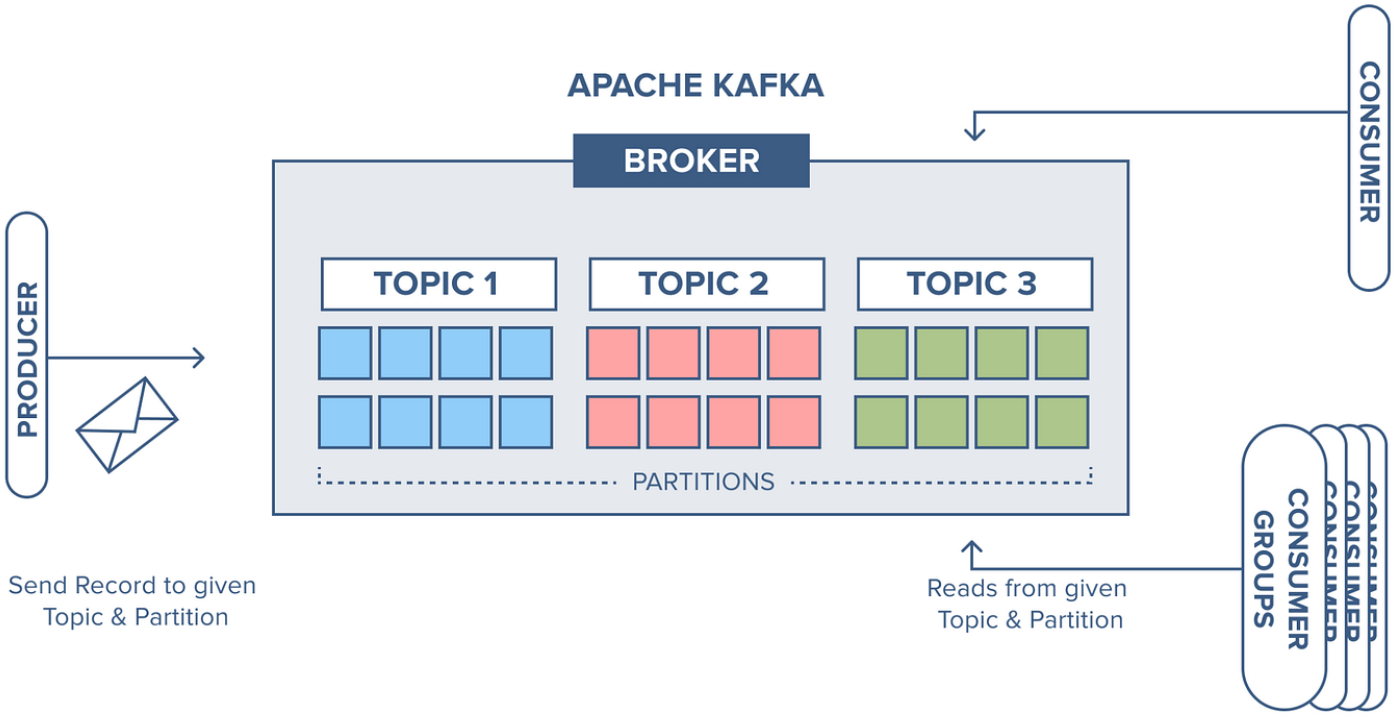
🚀 **Practice exercise (highly recommended)**

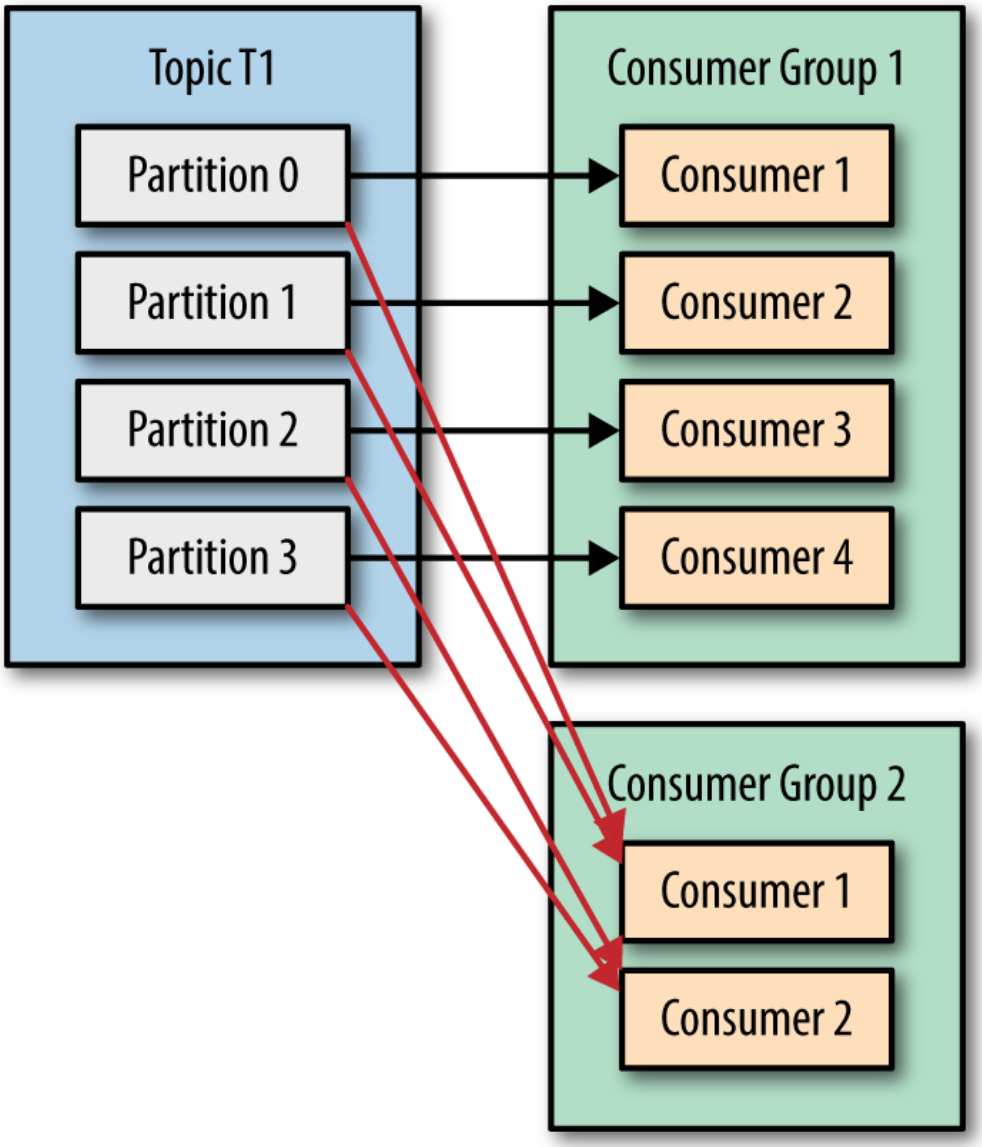
- Use **Apache Kafka** or **Amazon Kinesis**
- Build:
  - Video upload → publish event
  - Two consumers → transcoder + caption service

This will **cement your understanding**.

Apache Kafka – Deep Dive (Easy explanation + real-life examples)




APACHE KAFKA
BROKER
TOPIC 1
TOPIC 2
TOPIC 3
PARTITIONS
PRODUCER
CONSUMER
CONSUMER GROUPS
Send Record to given Topic & Partition
Reads from given Topic & Partition


Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1
Consumer 2
Consumer 3
Consumer 4
Consumer Group 2
Consumer 1
Consumer 2

1️⃣ When do we use Apache Kafka?

Apache Kafka is used as a **Message Stream**.

You use Kafka when:

- You need **write once, read by many**
- You have **very high throughput**
- You want to **decouple producers and consumers**
- You want to **store events and replay them**

2️⃣ Why Kafka is famous: High Throughput 🚀

Kafka can handle **millions of events per second**.

Real-life example: Uber 🚗

- Every driver sends location **every 2 seconds**
- Thousands of drivers → thousands of writes every 2 seconds

❌ Writing directly to DB:

- DB throughput is limited
- DB may crash

✅ Kafka approach:

1. Push driver locations to Kafka every 2 sec
2. Kafka handles massive writes easily
3. Consumer batches data
4. Writes to DB every **10 minutes**

📌 **DB load is reduced**
📌 **System becomes stable**

3️⃣ Kafka basic building blocks (very important)

🔹 Producer

- Sends messages (events) to Kafka
- Example message:

{ "email": "user@gmail.com", "message": "Welcome!" }

🔹 **Consumer**

- Reads messages from Kafka
- Processes them
- Maintains its own **offset**

🔹 **Broker**

- A Kafka **server**
- Stores messages
- Manages topics and partitions

📌 Kafka cluster = multiple brokers

🔹 **Topic**

- A **logical stream of events**
- Similar to a **table** in DB

Examples:

- sendEmail
- writeLocationToDB
- videoUploaded

4️⃣ **Kafka ≈ Database analogy** 🧠

Kafka	Database
Broker	DB Server
Topic	Table
Partition	Shard
Message	Row

This analogy helps you **remember easily**.

5️⃣ **Partitions (core of Kafka scaling 🔥)**

Each **topic** is divided into **partitions**.

Why partitions?

- Parallelism
- High throughput
- Horizontal scaling

📌 Partitioning logic is decided by **you (developer)**.

Example: Partition by location

- North India → Partition 0
- South India → Partition 1

👉 Similar to **DB sharding**

6️⃣ Consumer Groups (MOST IMPORTANT CONCEPT)

A **Consumer Group**:

- A group of consumers doing **same type of work**
- Each partition is assigned to **only one consumer per group**

Example setup

Topic has **4 partitions**:

P0, P1, P2, P3

Consumer Group has **3 consumers**:

C1, C2, C3

Kafka automatically rebalances:

- C1 → P0
- C2 → P1, P2
- C3 → P3

📌 **Kafka handles rebalancing automatically**

7️⃣ Important Kafka rule ⚠️

One partition can be consumed by only ONE consumer in a consumer group

But:

- Same partition can be read by **different consumer groups**

8️⃣ What if consumers > partitions?

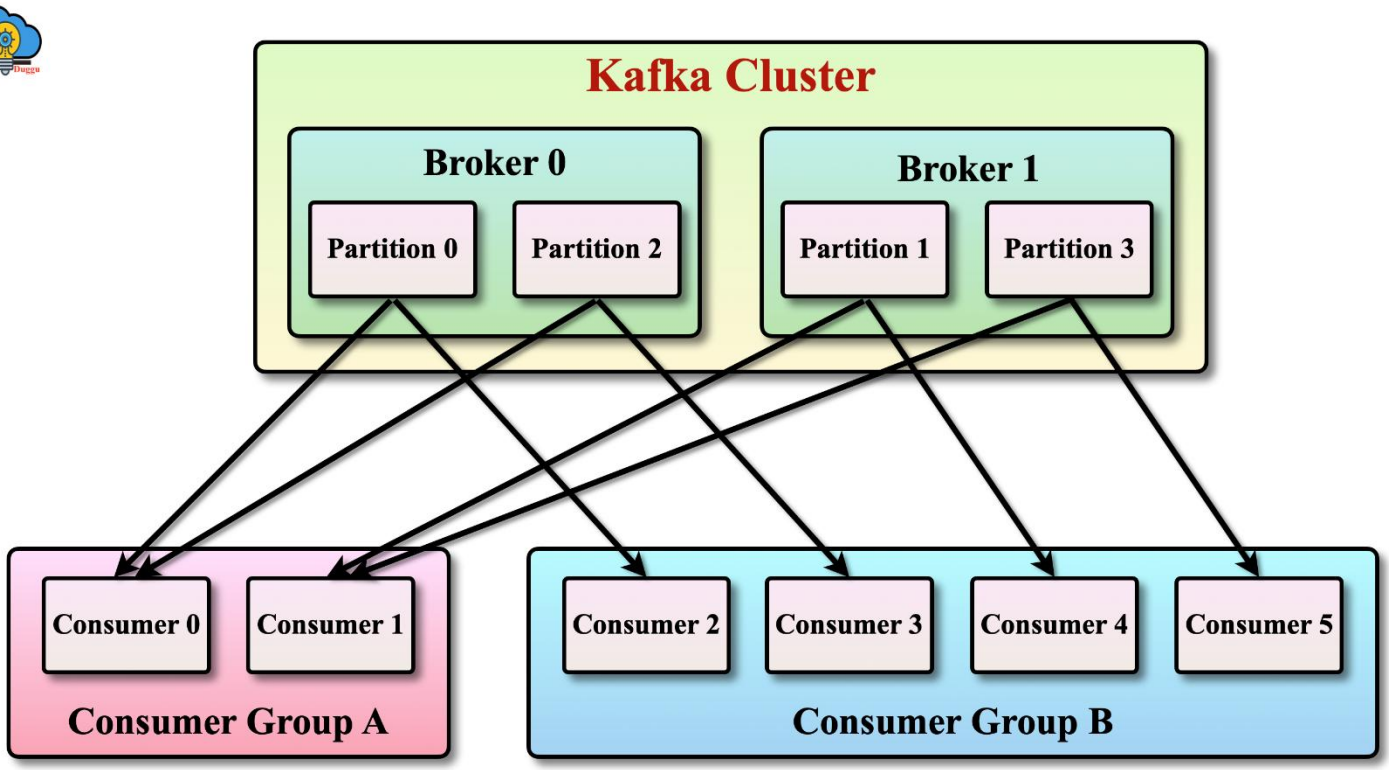
- Extra consumers stay **idle**
- They do nothing

👉 To scale consumers:

- Increase **number of partitions**

📌 **Consumers ≤ Partitions**

9️⃣ Multiple consumer groups (fan-out pattern)



Example:

- Topic: videoUploaded

Consumer Groups:

- Group 1 → Video Transcoding
- Group 2 → Caption Generator

📌 Both read **same messages**
📌 Both maintain **independent offsets**
📌 No conflict

1️⃣0️⃣ Why Kafka messages are not deleted?

- Kafka keeps messages for:
  - Fixed time (e.g., 7 days)
  - Or size-based retention

Consumers:

- Track offsets
- Replay messages if needed

👉 This enables:

- Debugging
- Reprocessing
- Analytics

1️⃣1️⃣ Where Kafka is used in real life

- Uber driver tracking
- Netflix activity logs
- LinkedIn feeds
- Payment event processing
- Analytics pipelines
- Event-driven microservices

🔑 Interview-ready summary

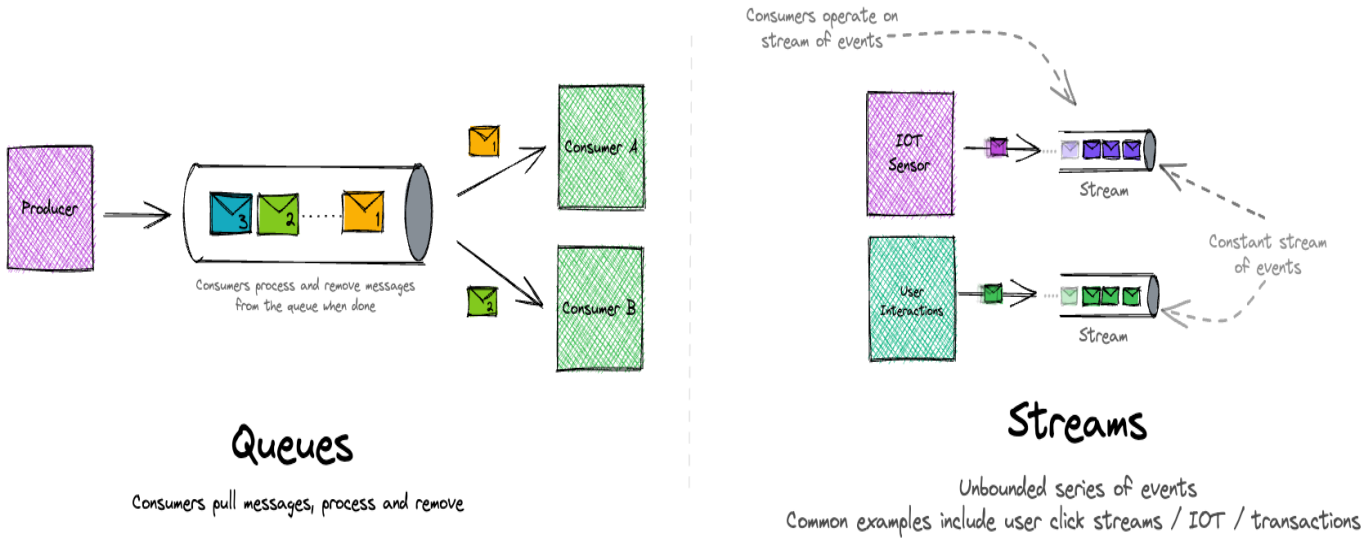
Apache Kafka is a high-throughput distributed event streaming platform used to build real-time data pipelines and event-driven systems. It enables producers to write events once and allows multiple consumer groups to read them independently with strong scalability and fault tolerance.

🧠 Memory tricks

- **Kafka = stream**
- **Topic = table**
- **Partition = shard**
- **Consumer group = scaling unit**

- **Consumers ≤ partitions**
- **Write once, read many**

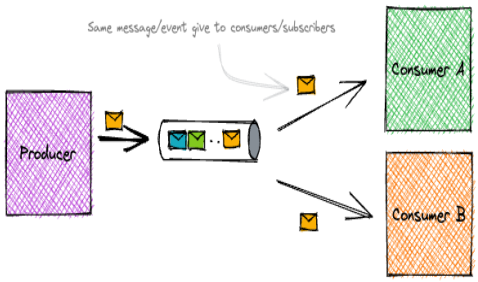
Realtime Pub/Sub (Push-based messaging, explained simply)



Queues vs Streams vs Pub/Sub

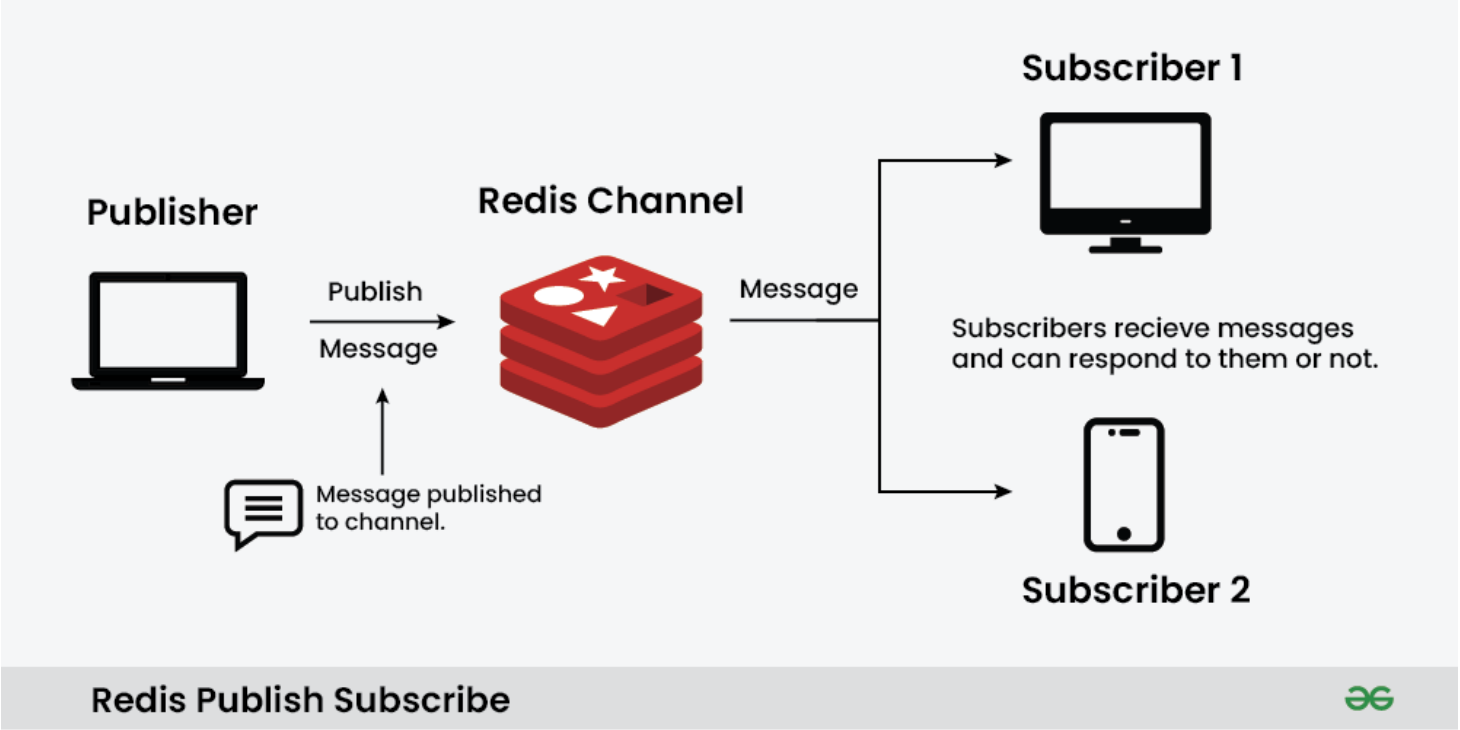
Bite sized visual to help understand the differences

@boyney123



Pub/Sub

Publish messages/events to many subscribers
Each subscriber gets copy of event to process



1️⃣ Message Broker vs Realtime Pub/Sub (core difference)

You already know **message brokers** (queues/streams). The key difference here is **who initiates delivery**.

Message Broker (Queue / Stream)

- **Consumer pulls** messages
- Messages are **stored/retained**
- Consumer uses **API/SDK** to fetch

Realtime Pub/Sub

- **Broker pushes** messages
- Messages are **not stored**
- Delivery is **instant (realtime)**

📌 **Pull vs Push** is the entire story.

2️⃣ What is Realtime Pub/Sub?

Realtime Pub/Sub means:

As soon as a publisher sends a message, the broker **immediately delivers it** to all subscribed consumers.

- No polling
- No waiting
- Ultra-low latency

3️⃣ Important property (don't miss this ⚠️)

❗ **Messages are NOT stored**

- If a subscriber is offline → **message is lost**
- Pub/Sub is for **live data**, not guaranteed delivery

👉 That's why Pub/Sub ≠ Queue ≠ Stream

4️⃣ Real-life analogy 📣

- **Pub/Sub** → Live TV broadcast
  Missed it? You can't rewind.
- **Queue** → Courier parcel
  Stored until someone picks it.
- **Stream** → Recorded show
  You can replay later.

5️⃣ Redis as a Realtime Pub/Sub Broker

Redis supports **Pub/Sub** in addition to caching.

How Redis Pub/Sub works

- **Publisher** publishes to a **channel**
- **Subscribers** listen to that channel
- Redis pushes messages instantly to all subscribers

📌 No persistence, no retries, no offsets.

6️⃣ Where do we use Realtime Pub/Sub? 🚀

Use it when you need:

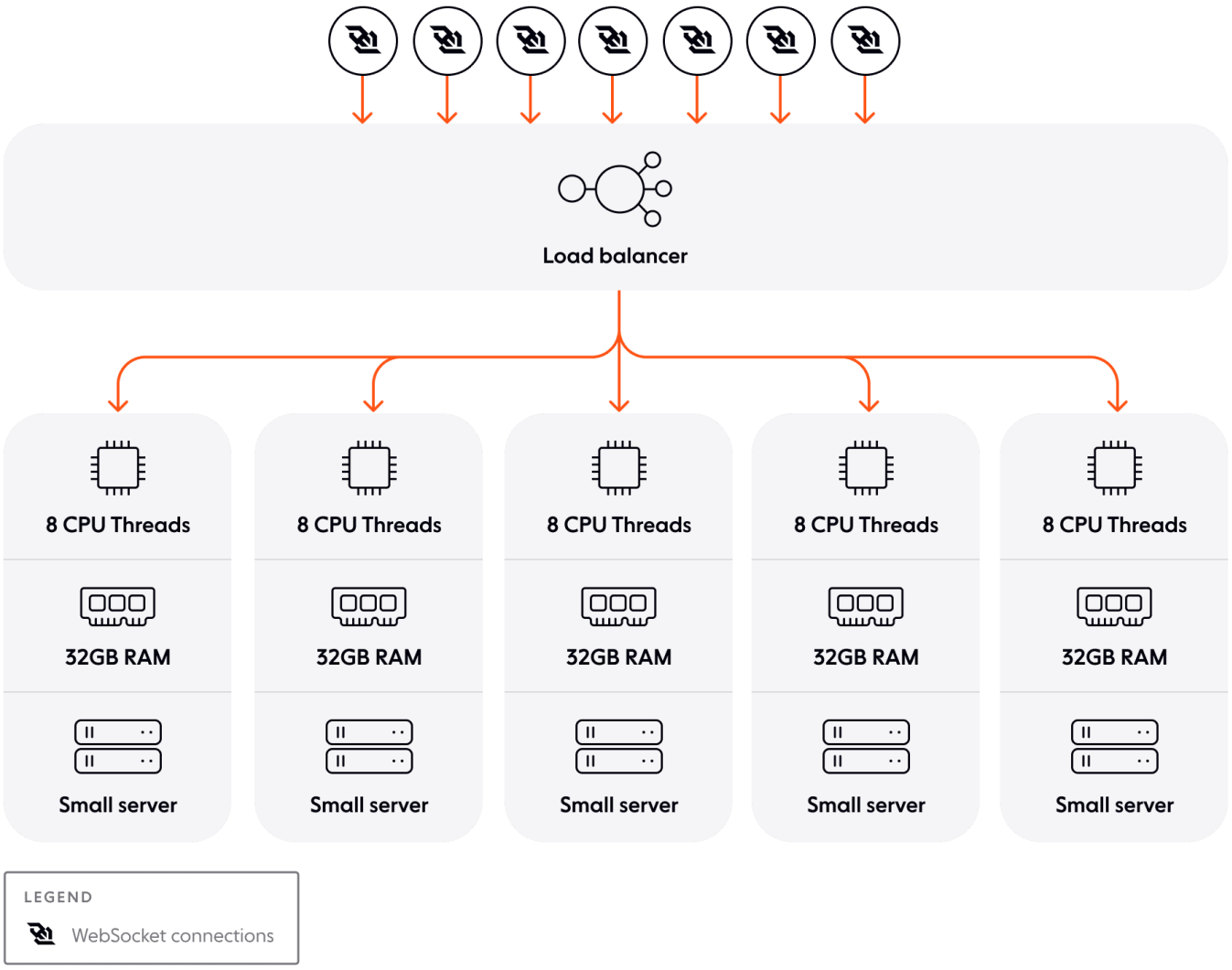
- **Very low latency**
- **Live updates**
- **Best-effort delivery**

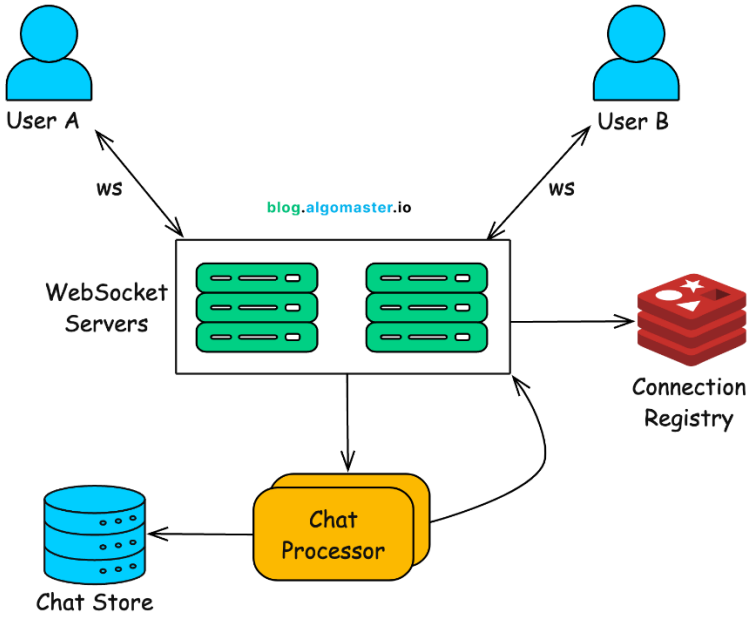
Common use cases

- Chat messages
- Live notifications
- Online presence (user typing…)
- Live dashboards
- Multiplayer game updates

7️⃣ Chat application problem (horizontal scaling)

Let’s say you have a **WebSocket chat app**.



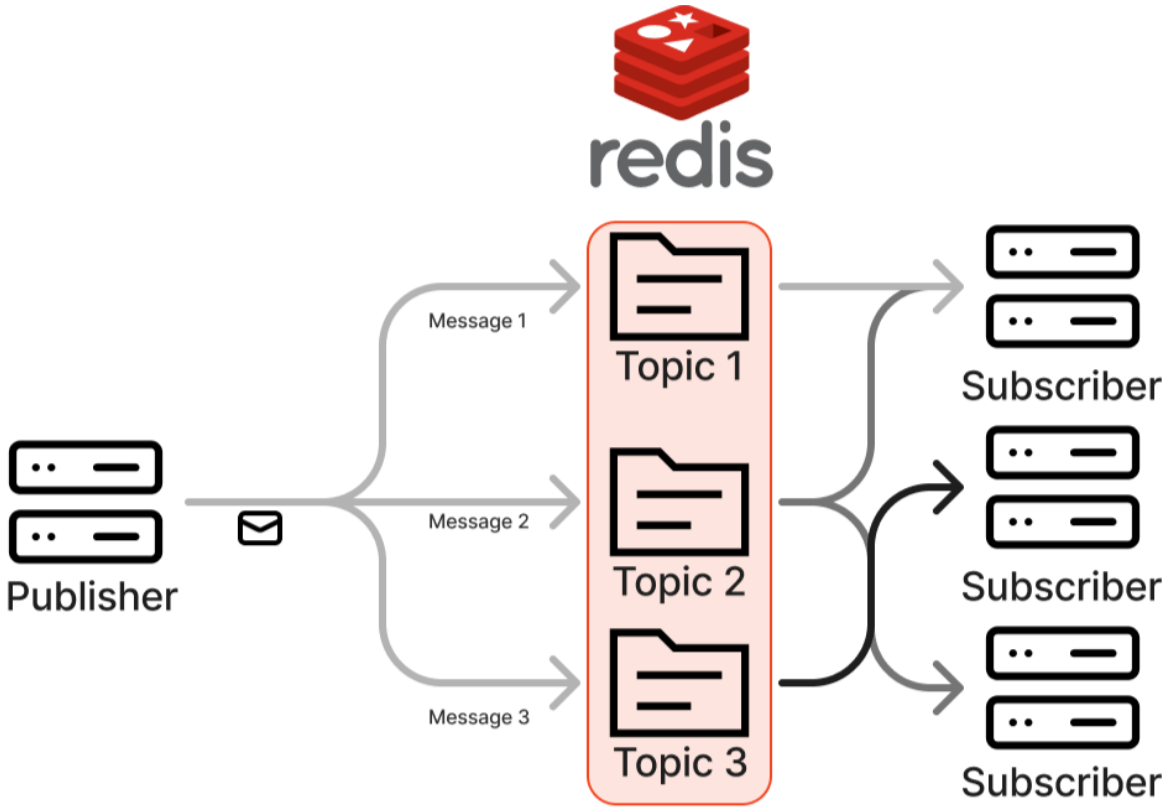


- Client-1 → connected to **Server-1**
- Client-3 → connected to **Server-2**

❌ Problem:

- Client-1 sends message to Server-1
- Server-1 **cannot reach** Client-3 directly

8️⃣ Solution: Redis Pub/Sub 🧠



Flow:

1. Client-1 sends message → Server-1
2. Server-1 **publishes** message to Redis channel
3. Redis **pushes** message to:
    - Server-1
    - Server-2
4. Server-2 sends message to Client-3 via WebSocket

📌 Redis becomes the **realtime bridge** between servers.

9️⃣ Why Redis Pub/Sub works well here

✅ Extremely fast (in-memory)
✅ Push-based (no polling)
✅ Simple to use
❌ No message durability

👉 Perfect for **live chat**, not for **financial transactions**

1️⃣0️⃣ Pub/Sub vs Queue vs Stream (final clarity table)

Feature	Pub/Sub	Queue	Stream
Delivery	Push	Pull	Pull
Storage	❌ No	✅ Yes	✅ Yes
Replay	❌ No	❌ No	✅ Yes
Latency	Ultra-low	Low	Medium
Use case	Live updates	Background jobs	Events & analytics

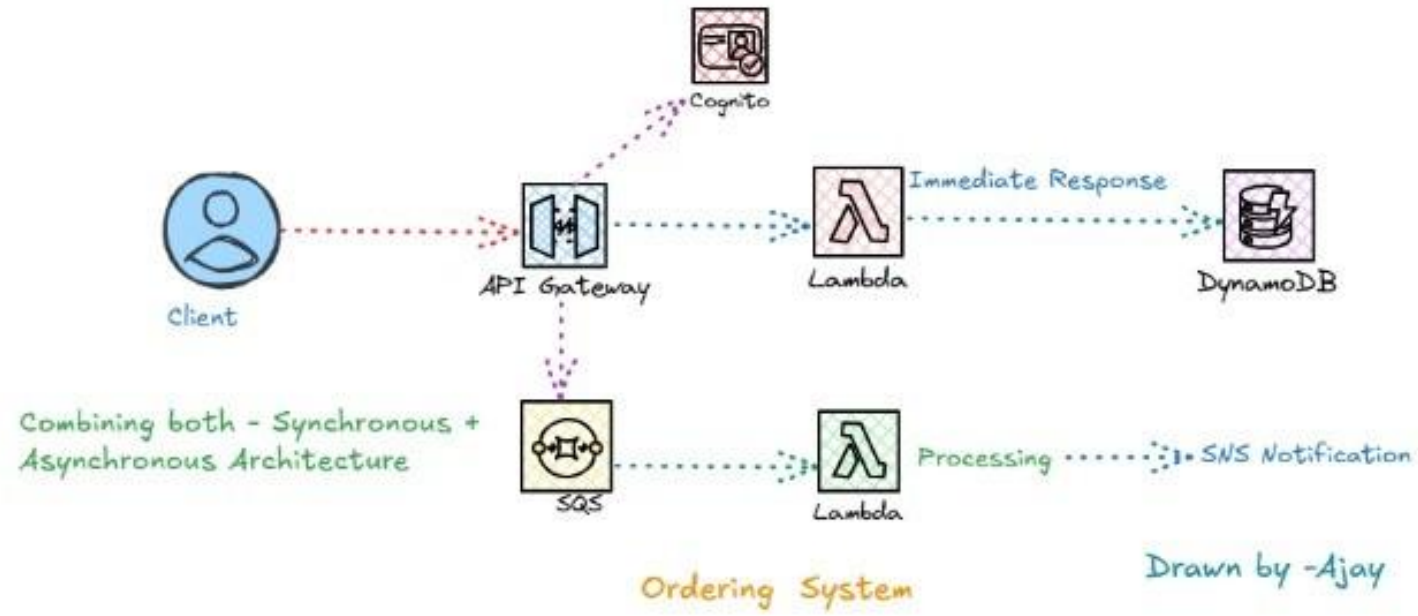
🔑 Interview-ready summary

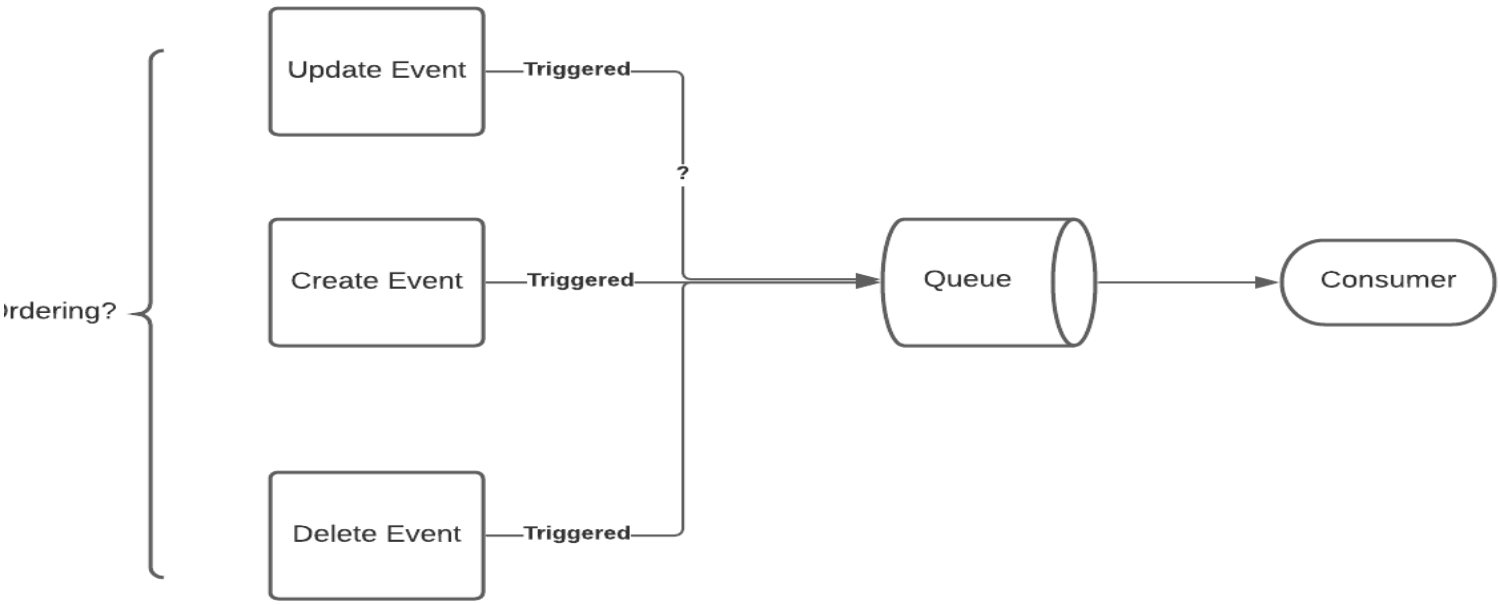
Realtime Pub/Sub is a push-based messaging model where messages are delivered instantly to all active subscribers without persistence. It is ideal for low-latency, live communication use cases such as chat and notifications, but not for guaranteed delivery workflows.

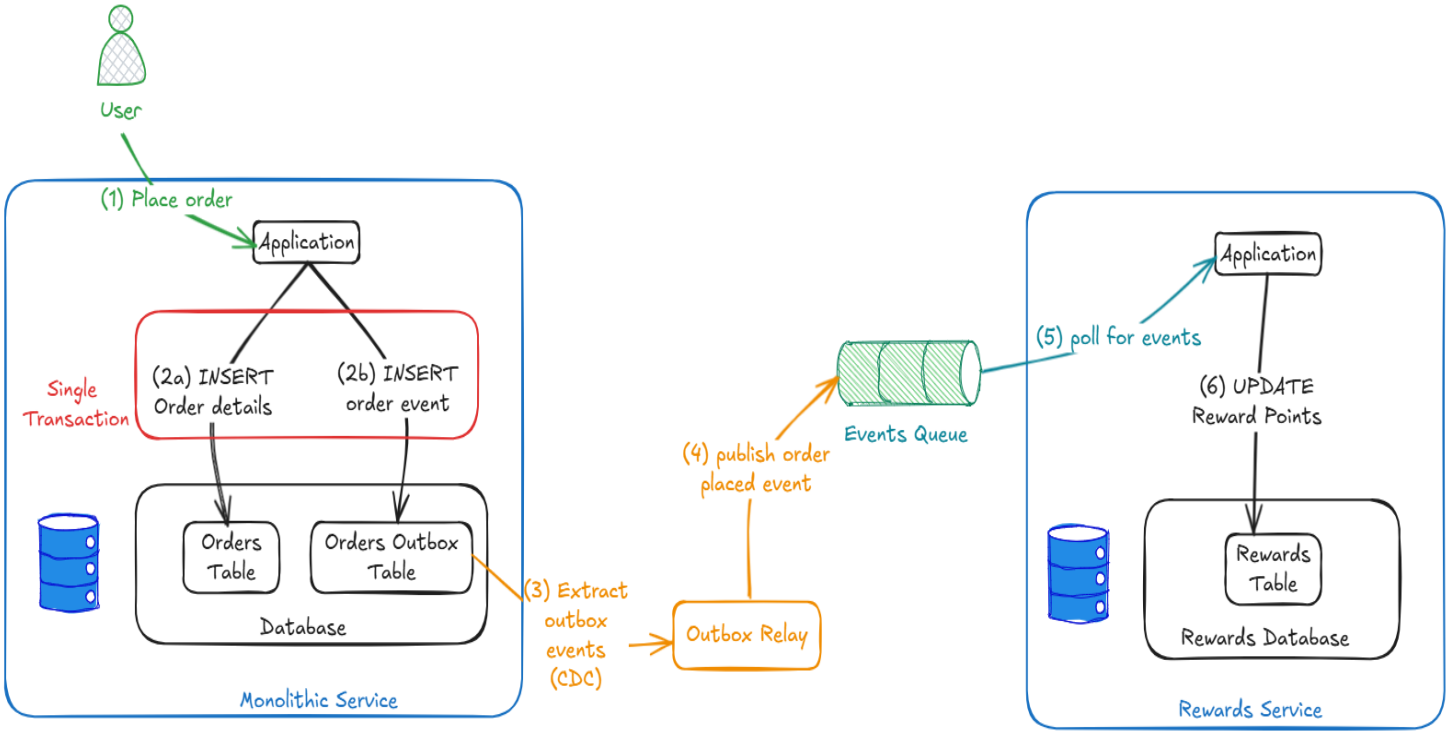
🧠 Memory tricks

- **Pub/Sub → Push**
- **Queue → Pull once**
- **Stream → Pull & replay**
- **Redis Pub/Sub → Live only**

Event-Driven Architecture (EDA) — easy explanation with real-life examples







1️⃣ What is Event-Driven Architecture?

EDA (Event-Driven Architecture) means:

Instead of services calling each other directly, they **emit events** and **react to events**.

- **Producer** → creates an **event**
- **Event Broker** → delivers the event
- **Consumers** → react asynchronously

📌 Producer **does not wait** for consumers.

2️⃣ Why do we need EDA? (Problem first)

Let's take an **e-commerce app like Amazon**.

Traditional (synchronous) flow ❌

When user places an order:

1. Order Service
2. Calls Payment Service
3. Calls Inventory Service
4. Calls Notification Service
5. Finally responds to client

Problems:

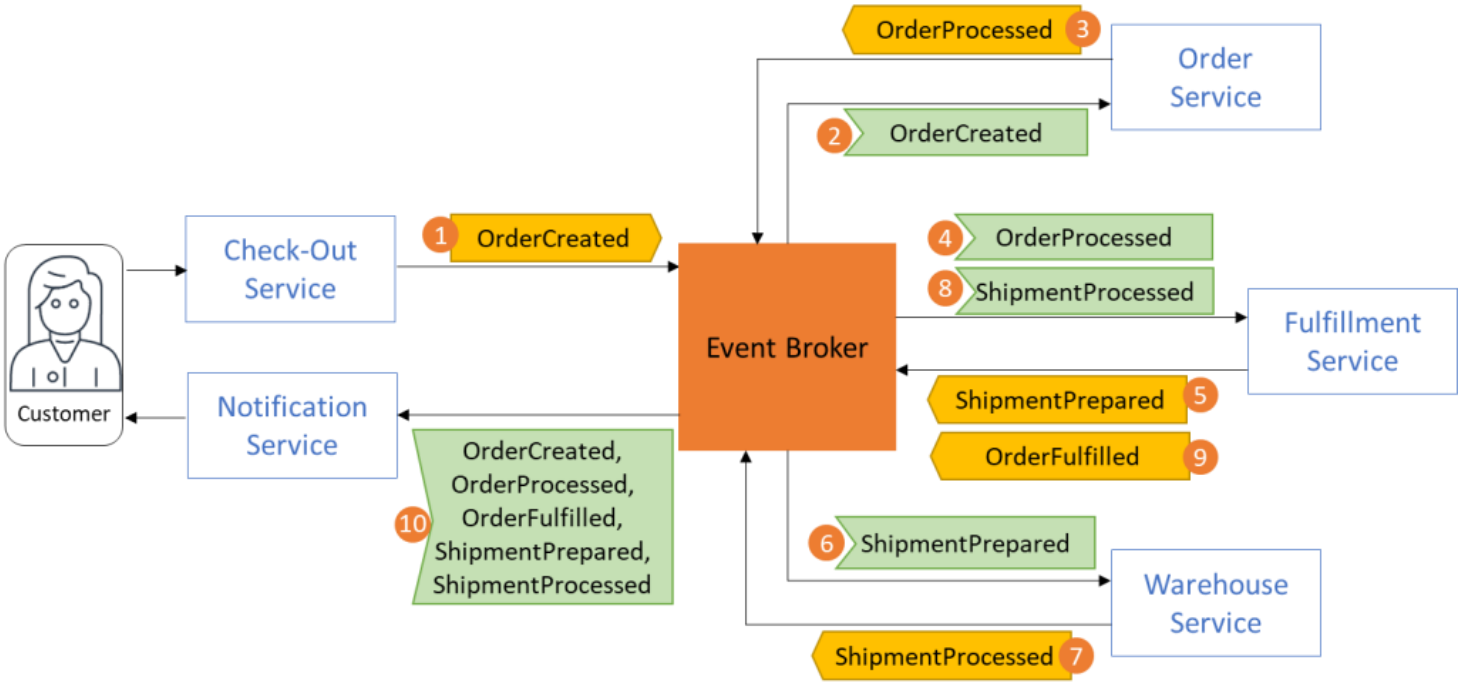
- Client waits unnecessarily ⏳
- Tight coupling
- If **Inventory Service is down**, order fails ❌
- Hard to scale

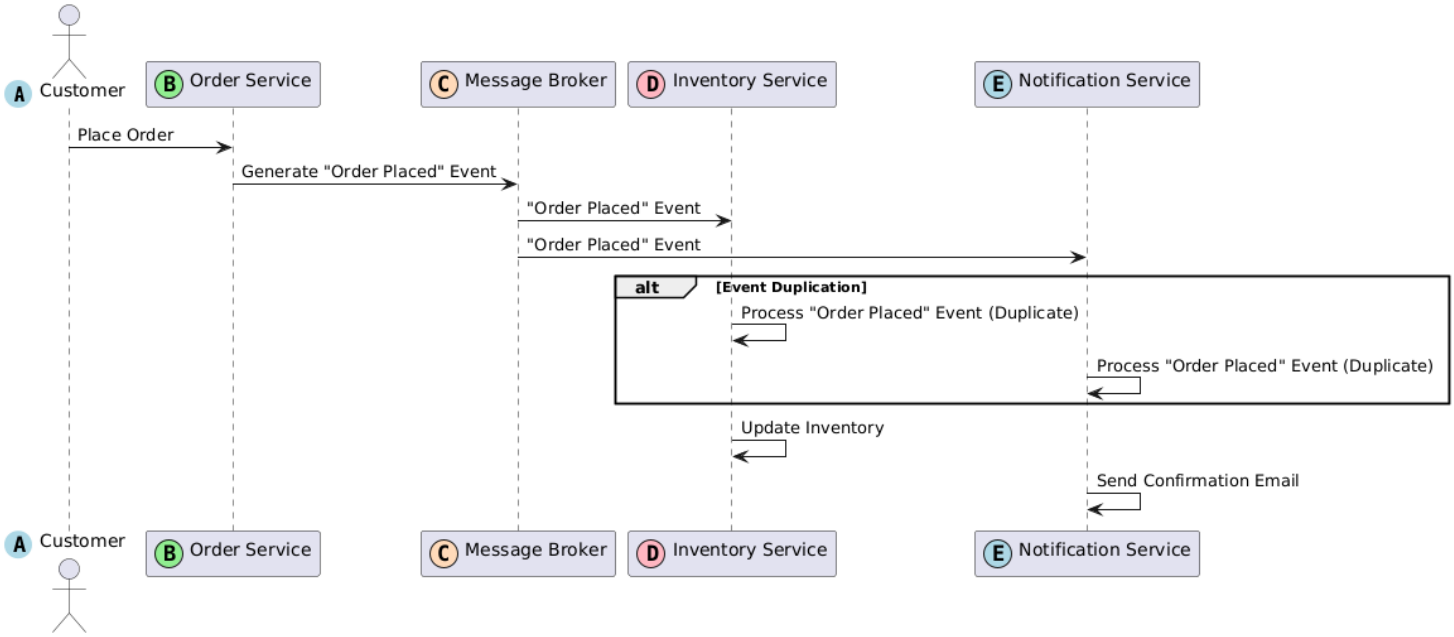
📌 Inventory & Notification **do not affect success page**

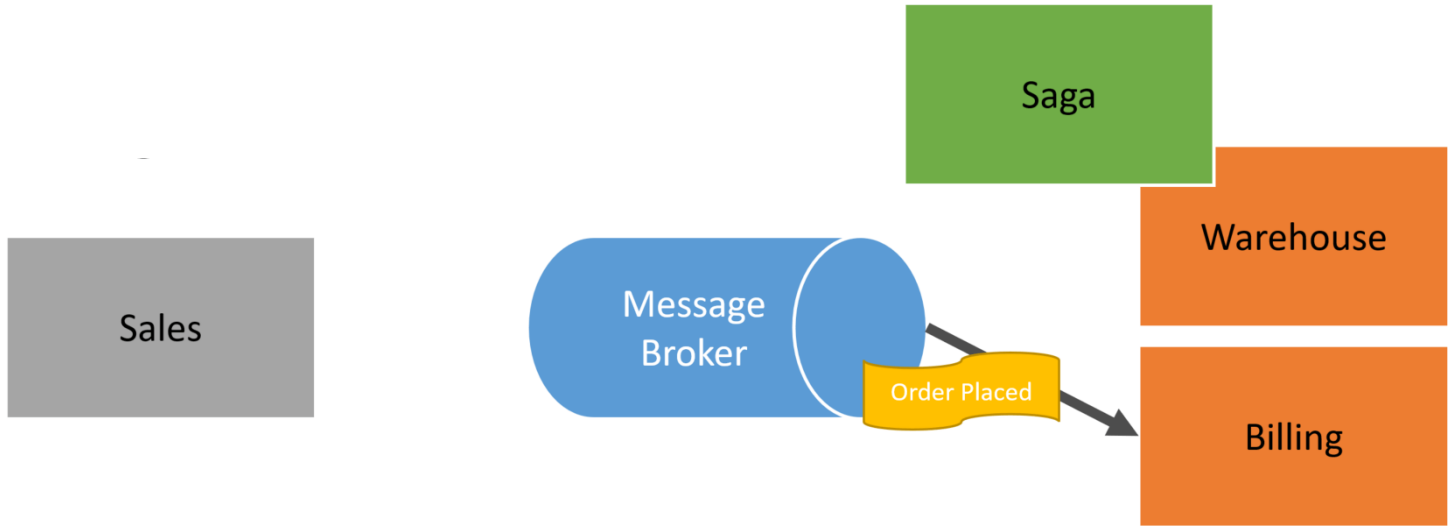
3️⃣ EDA solution (asynchronous flow) ✅

Instead:

1. Order Service completes payment
2. Sends response to client immediately
3. Publishes **OrderPlaced event**
4. Other services react in background







Flow:

Order Service → Event Broker → Inventory Service

→ Notification Service

📌 Order Service **forgets about consumers**

4️⃣ What is an Event?

An **event** is:

A fact that something **already happened**

Examples:

- OrderPlaced
- PaymentCompleted
- UserRegistered

- VideoUploaded

📌 Events are **immutable** (cannot be changed)

5️⃣ Why EDA is powerful 🔥

✅ 1. Decoupling

- Producer doesn't know consumers
- Add new consumers **without changing producer**

Example:

- Later add **Analytics Service**
- No change in Order Service

✅ 2. Resilience

- Inventory Service down?
  → Orders still work
- Notification fails?
  → Order still succeeds

📌 Failures don't cascade

✅ 3. Scalability

- Inventory Service scales independently
- Notification Service scales independently

📌 Each service grows at its own pace

6️⃣ Real-life analogy 📰

Think of a **newspaper**:

- Publisher prints news
- Readers read independently
- Publisher doesn't care **who reads**

👉 Newspaper = **Event**
👉 Readers = **Consumers**

7️⃣ Event-Driven vs API-Driven (quick compare)

API-Driven	Event-Driven
Tight coupling	Loose coupling
Sync calls	Async
Failure cascades	Failure isolated
Hard to extend	Easy to extend

8️⃣ Types of Event-Driven Patterns

There are **4 EDA patterns**, but we focus on **first two** (most used).

🔹 1. Simple Event Notification (MOST COMMON)

Producer emits event with **minimal data**.

Example:

```
{
  "event": "OrderPlaced",
  "orderId": 123
}
```

Consumers:

- Inventory Service → fetch order details
- Notification Service → fetch order details

📌 Event says *what happened*, not *all details*

🔹 2. Event-Carried State Transfer

Event carries **all required data**.

Example:

```
{
  "event": "OrderPlaced",
  "orderId": 123,
  "userId": 45,
```

"items": [...],

"total": 999

}

Consumers:

- No extra DB calls
- Faster processing

📌 Larger event, but fewer dependencies

🚫 Not covering (advanced / niche)

- Event Sourcing
- Event Sourcing + CQRS

(Used in very specific systems like banking ledgers)

🌍 Where EDA is commonly used

- E-commerce (orders, inventory)
- Notifications & emails
- Analytics & logging
- Microservices communication
- Video processing pipelines

🔑 Interview-ready summary

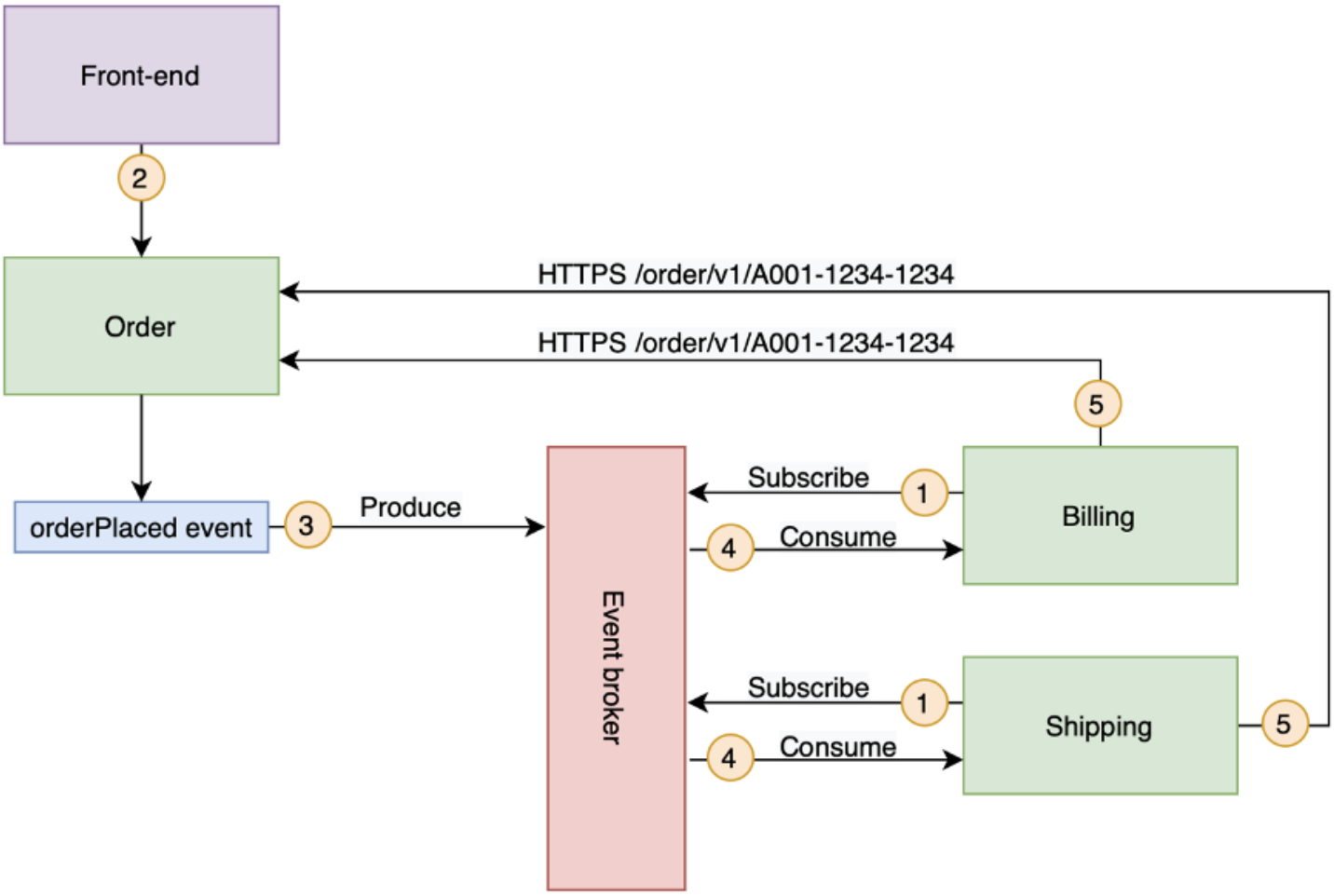
Event-Driven Architecture is a design approach where services communicate by emitting and reacting to events asynchronously. It improves decoupling, scalability, and resilience by removing direct service-to-service dependencies.

🧠 Memory tricks

- **API call → wait**
- **Event → fire & forget**
- **Non-critical work → EDA**
- **Loose coupling → Events**

Simple Event Notification vs Event-Carried State Transfer

(Easy explanation + real-life examples)



1️⃣ Simple Event Notification (SEN)

What it means

In **Simple Event Notification**, the producer sends a **lightweight event** that only says **"something happened"**.

- Event contains **minimal data**
- Consumers fetch extra details **themselves** (usually from DB or another service)

Example (E-commerce order)

Event published:

{

"event": "OrderPlaced",

"orderId": 123

}

What consumers do

- **Inventory Service**
  - Receives orderId
  - Queries DB to get items & quantity
- **Notification Service**
  - Receives orderId
  - Queries DB to get user email

📌 Producer doesn't care who consumes or how they fetch data.

Real-life analogy 📣

Someone shouts:

"Order #123 is placed!"

Anyone interested:

- Goes and checks the **order register** for details

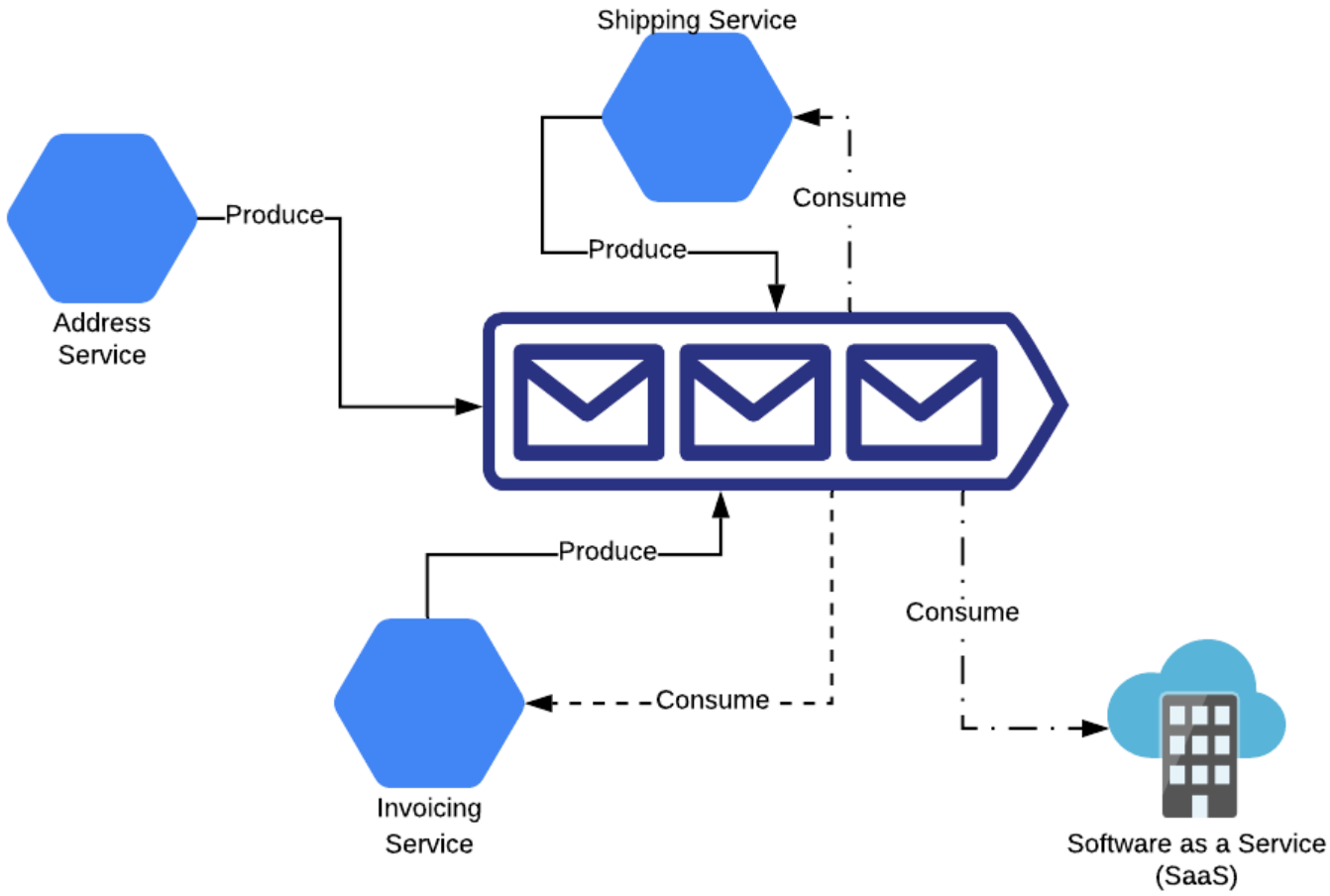
✅ **Advantages**

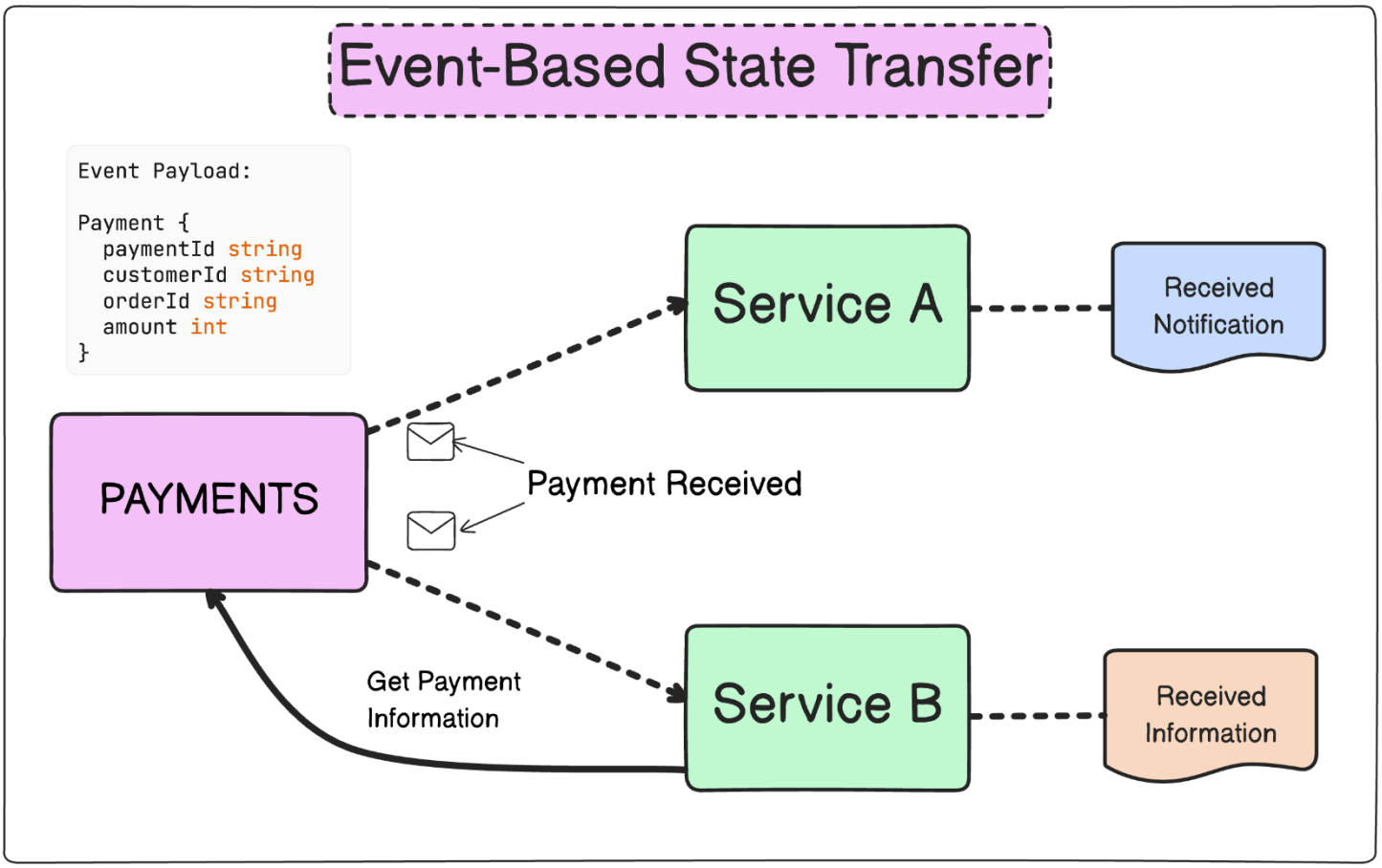
- Events are **small & cheap**
- Less bandwidth & broker storage
- Producer logic is **simple**

❌ **Disadvantages**

- Consumers make **extra DB/network calls**
- Higher end-to-end latency
- More load on database

2️⃣ **Event-Carried State Transfer (ECST)**





What it means

In **Event-Carried State Transfer**, the producer sends **all required data** inside the event.

- Event is **self-contained**
- Consumers **don't query DB**

Example (E-commerce order)

Event published:

```
{
 "event": "OrderPlaced",
 "orderId": 123,
 "userId": 45,
 "email": "user@gmail.com",
 "items": [
  {"productId": 7, "qty": 2}
 ],
 "totalAmount": 999
}
```

What consumers do

- Inventory Service → updates stock directly
- Notification Service → sends email immediately

📌 No extra network calls.

Real-life analogy 📦

Someone hands you a **complete order slip**:

- Customer details
- Items
- Address

You don't need to ask anyone else.

✅ **Advantages**

- **Lower latency**
- Faster consumer processing

- Fewer DB calls
- Better for high-throughput systems

❌ **Disadvantages**

- Larger event size
- More bandwidth & broker cost
- Tighter coupling to event schema
- Versioning becomes important

3️⃣ **Side-by-side comparison** 🧠

Aspect	Simple Event Notification	Event-Carried State Transfer
Event size	Small	Large
DB calls by consumer	Yes	No
Latency	Higher	Lower
Broker cost	Lower	Higher
Coupling	Loose	Medium
Complexity	Low	Medium

4️⃣ **When to use which? (Rule of thumb)**

Use Simple Event Notification when:

- Data is large or sensitive
- Consumers already need DB access
- You want **loose coupling**
- Traffic is moderate

Examples:

- Order placed → inventory update
- User signup → analytics tracking

Use Event-Carried State Transfer when:

- You need **very low latency**

- Consumers should be independent
- DB load must be minimized
- Events are processed at massive scale

Examples:

- Payment success notifications
- Real-time order fulfillment
- Streaming pipelines

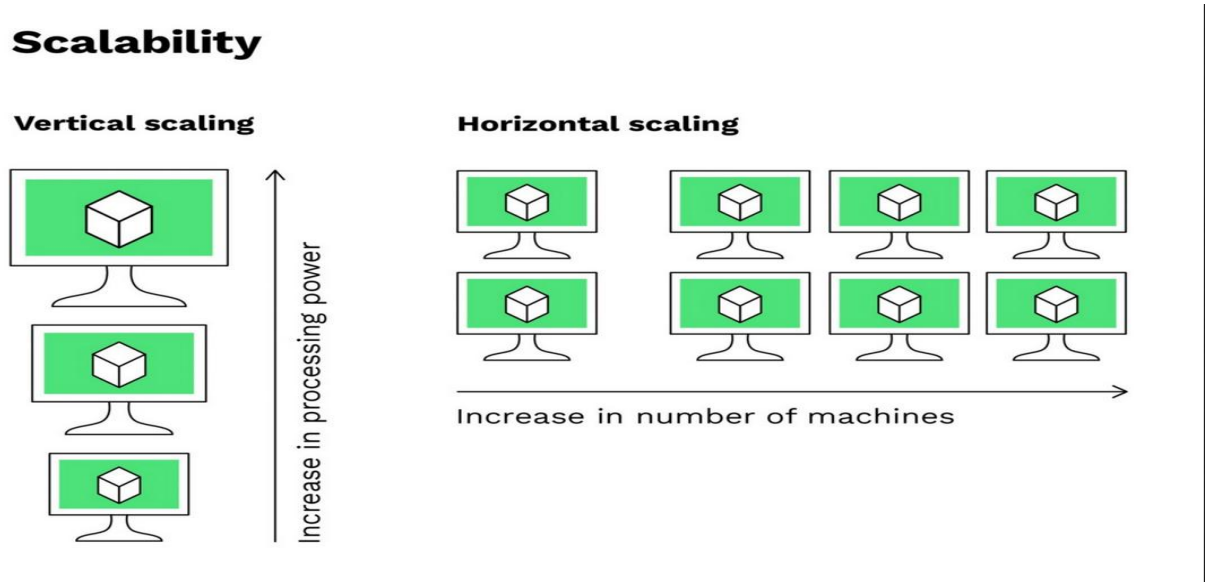
🔑 Interview-ready summary

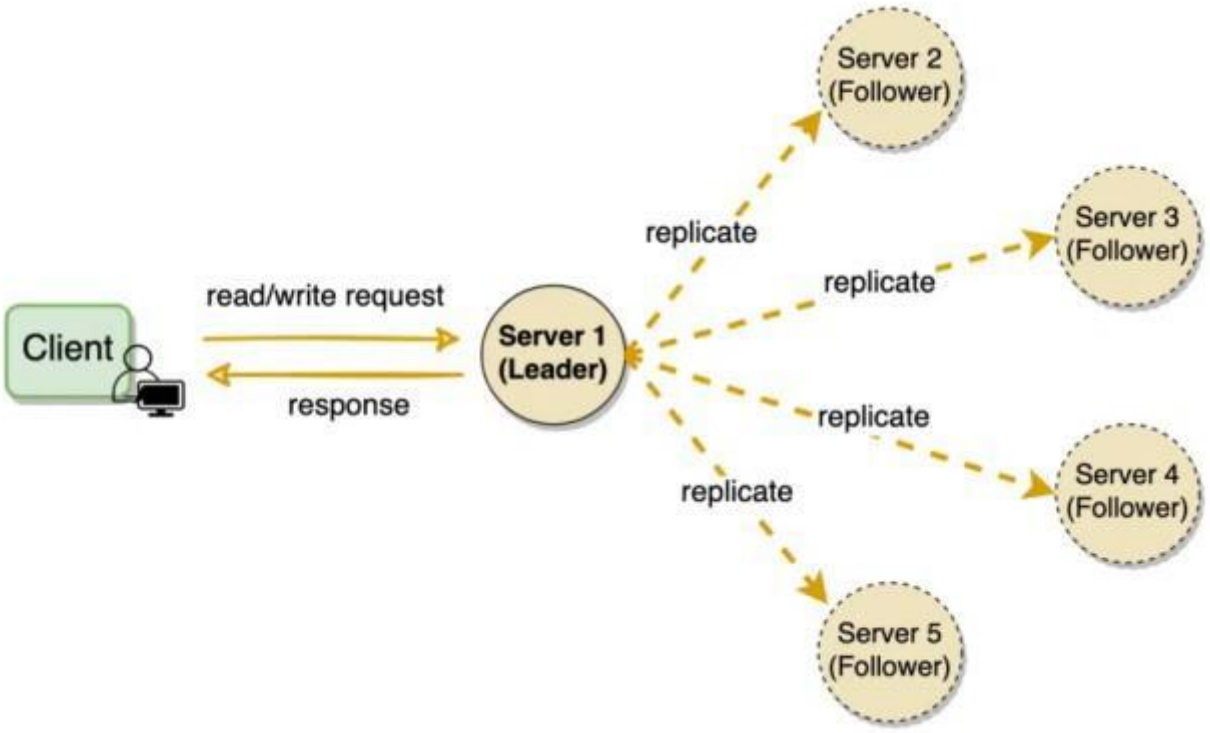
Simple Event Notification sends minimal event data and lets consumers fetch details themselves, while Event-Carried State Transfer embeds all required data in the event, reducing latency at the cost of larger event payloads and tighter coupling.

🧠 Memory tricks

- **SEN** → "Something happened"
- **ECST** → "Here is everything"
- **Small event → DB call**
- **Big event → No DB call**

Distributed Systems (Easy explanation with real-life examples)





1️⃣ What is a Distributed System?

A **Distributed System** is:

A system where **multiple computers (machines)** work together to complete a task instead of relying on a single machine.

From the **user's point of view**, it should feel like:

"I am talking to **one system**, not many machines."

2️⃣ Why do we need Distributed Systems?

Let's start with your example 👇

Small problem (single machine works)

- Find prime numbers between **0 and 10,000**
- One computer + one loop = fine ✅

Huge problem (single machine fails ❌)

- Find primes between **0 and 10^{100}**
- One machine:
    - Takes forever

- Runs out of CPU / memory
- Chokes ❌

3️⃣ How Distributed Systems solve this

We **divide the work** and **do it in parallel**.

Example with 10 machines:

- Machine 1 → 0 to 10^{10}
- Machine 2 → 10^{10}+1 to 10^{20}
- …
- Machine 10 → last range

Each machine:

- Works independently
- Computes partial result

Finally:

- Results are **combined**
- Final answer is returned to client

📌 **Parallelism = speed + scalability**

4️⃣ Simple definition (interview-friendly)

A distributed system is a collection of independent machines that appear to the user as a single coherent system.

5️⃣ Real-world examples you already know

Distributed systems are everywhere 👇

🔹 Horizontal Scaling

- Multiple servers handle user requests
- Load balancer distributes traffic
  ✓ Example: Web applications

🔹 Database Sharding

- Data split across multiple machines
- Each machine stores part of the data
  ✓ Example: User table split by user_id

🔹 **Distributed Databases**

Examples:

- **Apache Cassandra**
- **Amazon DynamoDB**
- **MongoDB**
- **Google Spanner**
- **Redis**

📌 All use **multiple machines** internally.

6️⃣ **Biggest challenge in Distributed Systems** ⚠️

The hardest part is:

- **Coordination**
- **Communication**
- **Failure handling**

Questions the system must answer:

- Who does which work?
- What if one machine fails?
- How do machines agree on decisions?

7️⃣ **Client perspective (VERY important)**

Even though internally:

- 10 servers
- 100 servers
- 1000 servers

From client's view:

Client → One system → Response

📌 **Abstraction hides complexity**

8️⃣ Leader–Follower model (most common approach)

How it works:

- One server → **Leader**
- Others → **Followers**

Responsibilities:

- **Leader**
  - Receives client requests
  - Divides work
  - Assigns tasks to followers
  - Collects results
- **Followers**
  - Do assigned work
  - Send results back

Leader merges results → sends response to client.

9️⃣ Leader election (conceptual understanding)

The system must decide:

1. **Who is leader at startup?**
2. **What if the leader crashes?**

Solution:

- **Leader Election Algorithm**
- All servers participate
- One becomes leader
- Others accept it

📌 If leader dies:

- Followers detect failure
- Run election again
- New leader is chosen automatically

You can think of this as a **black box** for system design interviews.

1️⃣0️⃣ Leader election algorithms (just awareness)

You mentioned these correctly:

Algorithm	Time Complexity
LCR	$O(N^2)$
HS	O(N log N)
Bully	O(N)
Gossip Protocol	O(log N)

📌 **Not required to memorize** for system design

📌 Just know **leader election exists**

1️⃣1️⃣ CAP Theorem reminder 🧠

Every distributed system must deal with:

- **Consistency**
- **Availability**
- **Partition Tolerance**

👉 You can only fully guarantee **two** at a time

👉 This applies to **all distributed systems**

1️⃣2️⃣ Real-life analogy 🏗️

Think of building a bridge:

- One worker → too slow
- 100 workers → fast
- One supervisor (leader)
- Many workers (followers)

If supervisor leaves:

- New supervisor is selected

👉 That's a distributed system.

🔑 Interview-ready summary

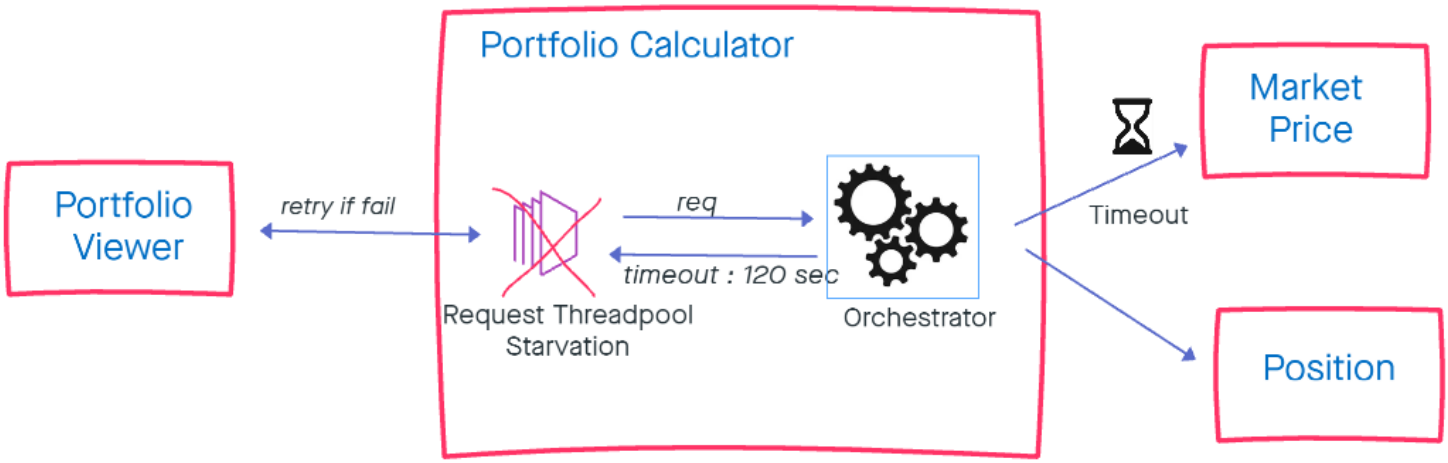
A distributed system is a collection of multiple independent machines that work together to solve a problem and appear as a single system to the client. It enables scalability, fault tolerance, and parallelism but introduces challenges like coordination, failure handling, and consistency.

🧠 **Memory tricks**

- **One machine fails → system survives**
- **Leader assigns work**
- **Followers execute**
- **Client sees one system**
- **CAP applies everywhere**

Auto-Recoverable System using Leader Election

(Easy explanation + real-life analogy + how it works end-to-end)



1️⃣ **The problem we are solving**

You have a **horizontally scaled system**:

- Multiple servers behind a **load balancer**
- You want **at least 4 servers always running**

❌ **Manual way (bad)**

- A server crashes
- You notice it
- You restart it manually

This is:

- Tedious 😴
- Error-prone
- Not scalable

👉 We want the **system to heal itself automatically**

2️⃣ Solution: Orchestrator (Auto-Healing)

An **Orchestrator** is a system whose job is:

Continuously monitor servers and restart them if they crash

What it does:

- Checks server health (heartbeat)
- If a server goes down → **restart it**
- Ensures desired number of servers is always running

📌 This is called **self-healing / auto-recovery**

3️⃣ New problem ❓

"Who watches the orchestrator?"

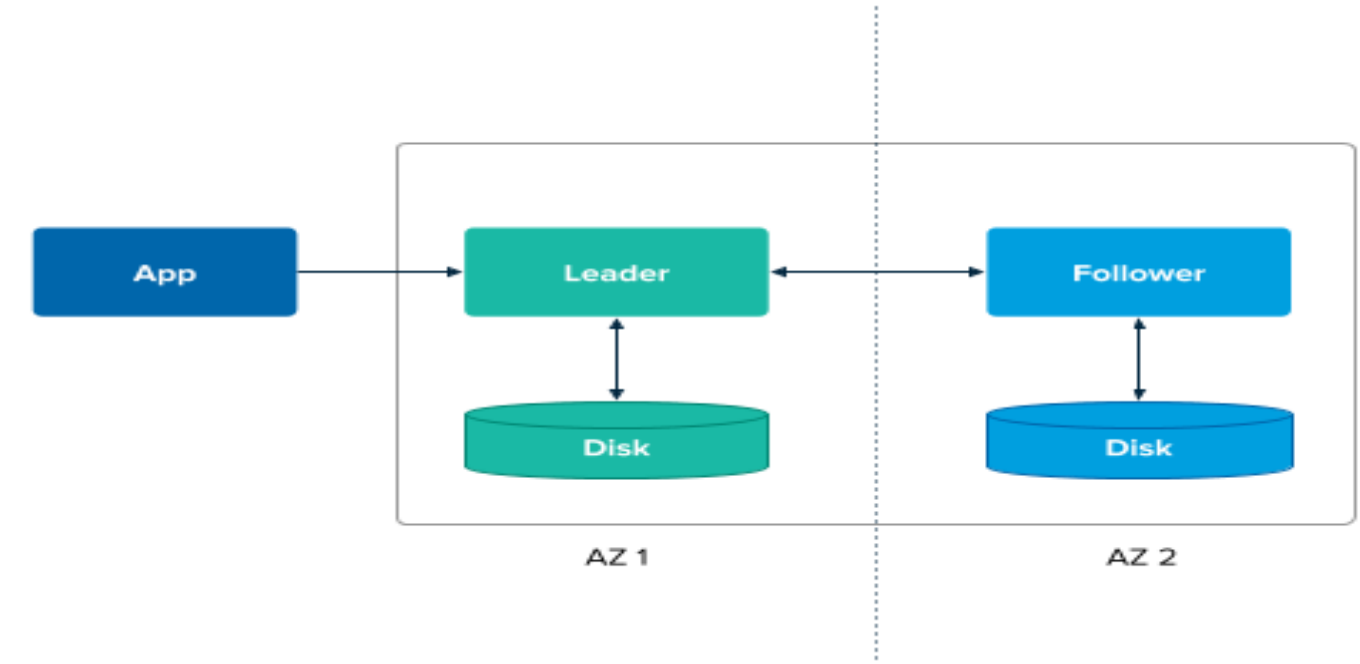
If:

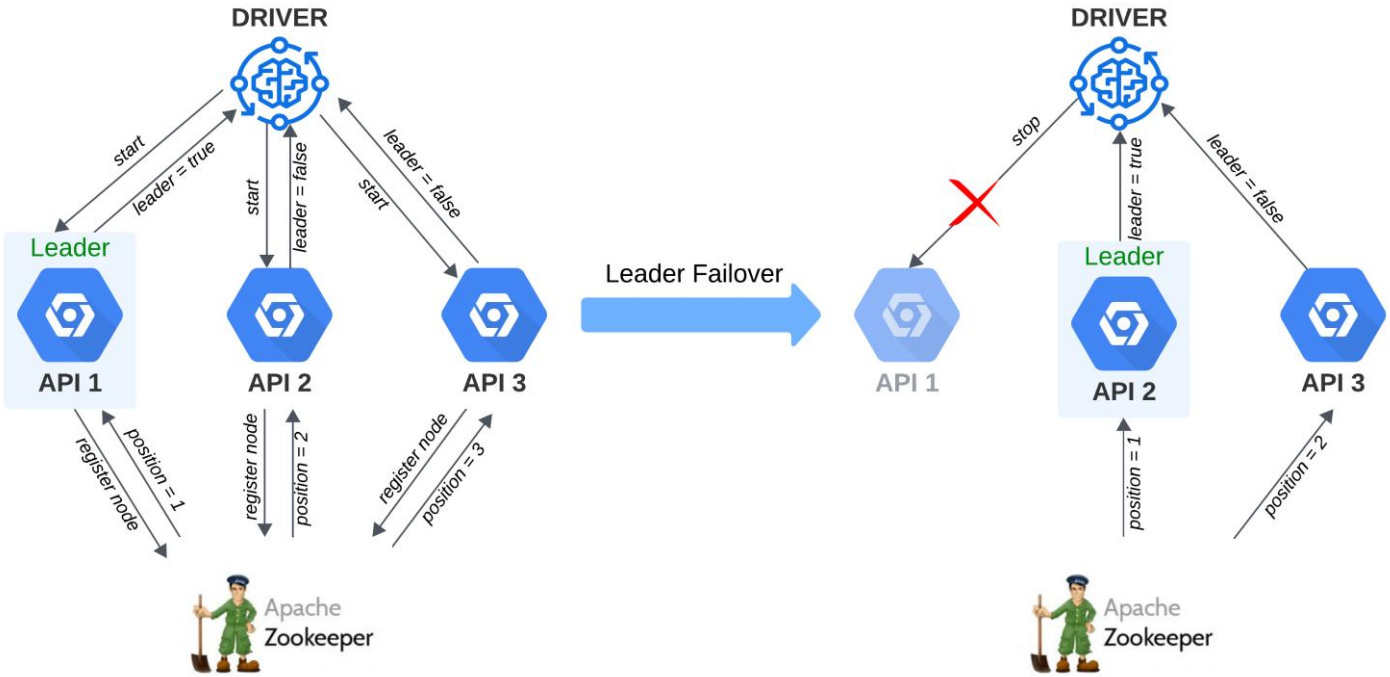
- Orchestrator crashes 💥
  Then:
- No one restarts servers
- System is broken again ❌

👉 **Single orchestrator = single point of failure**

4️⃣ Final solution: Multiple Orchestrators + Leader Election

Instead of **one orchestrator**, we run **many orchestrators**.





Roles:

- **Leader-Orchestrator**
  - Supervises everything
  - Monitors worker-orchestrators
- **Worker-Orchestrators**
  - Monitor application servers
  - Restart servers if needed

5️⃣ How Leader Election fits here 🧠

Among orchestrators:

- One is elected as **Leader**
- Others are **Followers**

Leader Election ensures:

- Only **one leader at a time**
- Automatic leader replacement on failure

📌 You don't manually choose the leader
📌 System decides automatically

6️⃣ Failure scenarios (IMPORTANT 🔥)

🔹 Case 1: Application Server crashes

Worker-Orchestrator → detects failure

→ Restarts the server

✅ No human needed

🔹 Case 2: Worker-Orchestrator crashes

Leader-Orchestrator → detects failure

→ Restarts worker-orchestrator

✅ System still stable

🔹 Case 3: Leader-Orchestrator crashes (most critical)

Followers detect leader is down

→ Run leader-election algorithm

→ One follower becomes new leader

✅ Zero downtime leadership

7️⃣ Why this system is AUTO-RECOVERABLE

Because:

- Every component is monitored
- Every failure has a recovery path
- Leadership is never lost

📌 **No single point of failure**
📌 **No manual intervention**

8️⃣ Real-life analogy 🏭

Think of a factory:

- Workers → Machines
- Supervisors → Worker-Orchestrators
- Manager → Leader-Orchestrator

If:

- A machine breaks → supervisor fixes it
- Supervisor leaves → manager replaces
- Manager leaves → senior supervisor becomes manager

👉 Factory never stops running

9️⃣ Real-world example (industry)

This exact model is used by **Kubernetes**:

- Pods → Application servers
- Nodes → Machines
- Controllers → Orchestrators
- Control Plane → Leader-elected

📌 Kubernetes is the **best real-world example** of this concept.

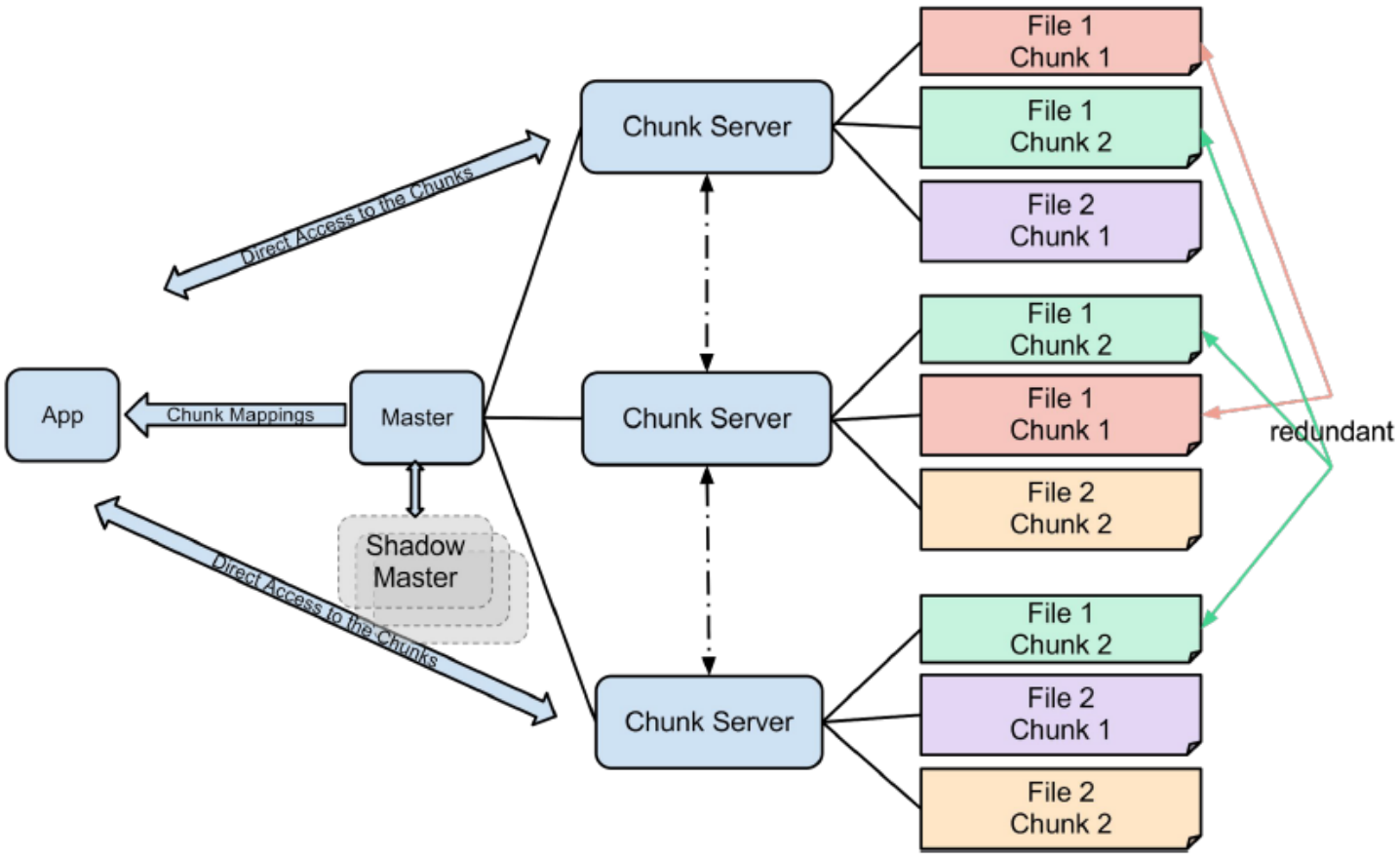
🔑 Interview-ready summary

An auto-recoverable system uses orchestrators with leader election to continuously monitor and restart failed components. By running multiple orchestrators and electing a leader dynamically, the system eliminates single points of failure and achieves self-healing without human intervention.

🧠 **Memory tricks**

- **Server dies → Worker restarts**
- **Worker dies → Leader restarts**
- **Leader dies → Election happens**
- **No humans → No downtime**

Big Data Tools (Easy explanation with real-life examples)



1️⃣ What are Big Data Tools?

Big Data tools are systems used to **process extremely large amounts of data** that **cannot be handled by a single machine**.

Example tools:

- **Apache Spark**
- **Apache Flink**

📌 These tools internally use **distributed systems**.

2️⃣ Why single machine is not enough ❌

Suppose:

- You have **TBs or PBs of data**
- You try to process it on one EC2 instance

Problems:

- CPU overload
- Memory overflow
- Disk I/O bottleneck
- Extremely slow execution

👉 The machine will **choke or take days**

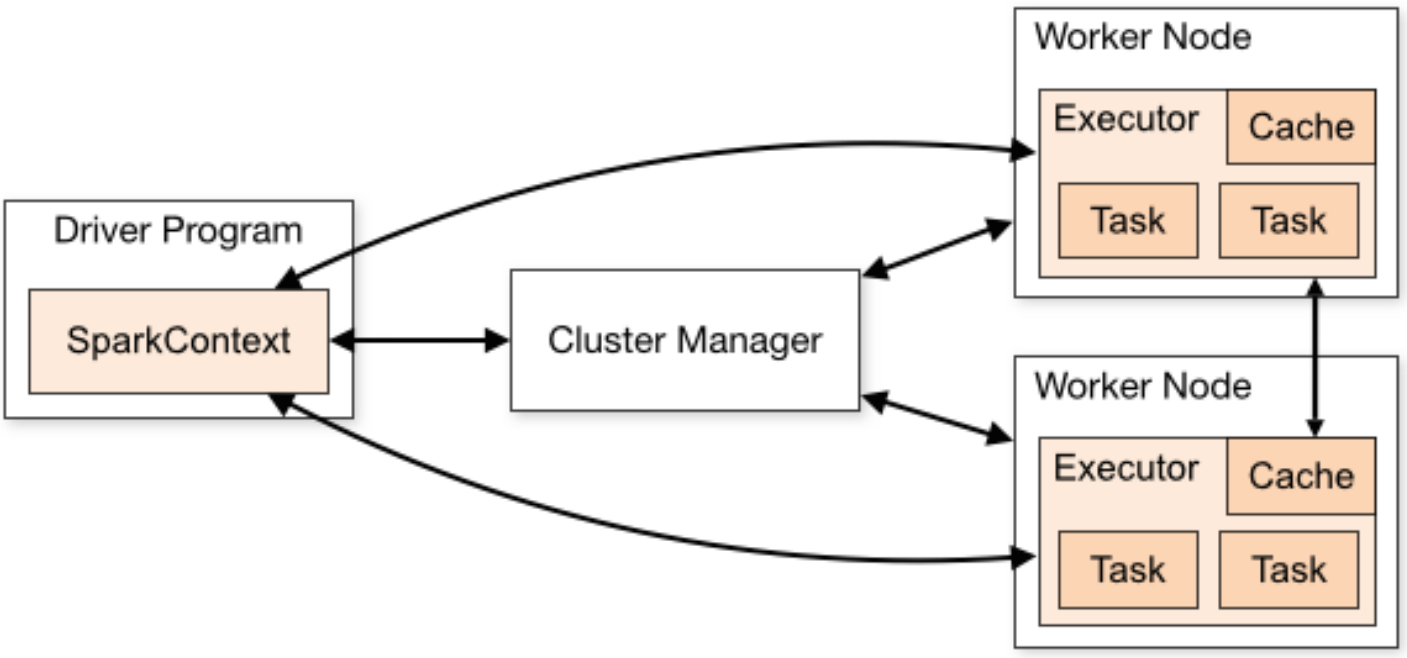
3️⃣ How Big Data tools solve this 🧠

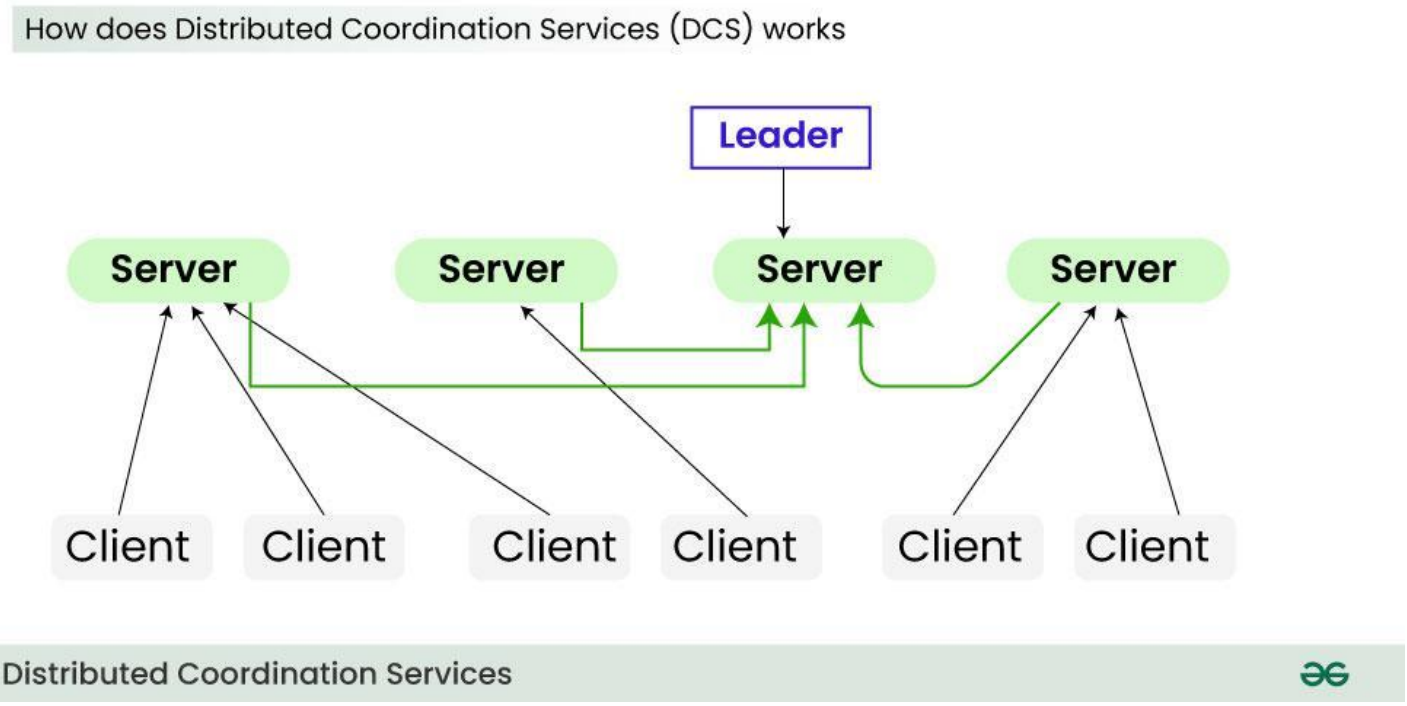
They use **many machines together**.

Core idea:

Divide the data → process in parallel → combine results

4️⃣ Coordinator–Worker model (core architecture)





Roles:

- **Coordinator (Master / Driver)**
  - Receives request from client
  - Splits large data into smaller chunks
  - Assigns work to workers
  - Collects results
  - Returns final output
- **Workers (Executors)**
  - Process assigned data chunks
  - Perform computation
  - Send results back

📌 This is very similar to **leader–follower distributed systems**

5️⃣ Example: Apache Spark (simple view)

In **Apache Spark**:

- Driver = Coordinator
- Executors = Workers
- Jobs = Business logic you write (ML, analytics, ETL)

You write:

- Python / Java / Scala code

Spark handles:

- Parallel execution
- Distribution
- Failures
- Recovery

👉 You focus on **business logic**, Spark handles **infrastructure**

6️⃣ What coordinator must handle (behind the scenes ⚙️)

Big data tools automatically manage:

- ✅ **Failure handling**
  - If worker crashes → reassign task
- ✅ **Recovery**
  - Restart failed workers
- ✅ **Result aggregation**
  - Combine partial outputs
- ✅ **Scaling**
  - Add/remove workers dynamically
- ✅ **Data redistribution**
- ✅ **Logging & monitoring**

📌 Writing this yourself is **extremely hard**

7️⃣ When should you use Big Data tools?

Use them when:

A **single EC2 instance is not sufficient**

8️⃣ Common real-world use cases 🌍

- Training **Machine Learning models**
- Processing **clickstream / logs**
- Recommendation systems

- Social network analysis
- ETL pipelines (data → warehouse)
- Analytics on massive datasets

9️⃣ Why we don't build this ourselves ❌

Because:

- Distributed coordination is complex
- Failure handling is hard
- Scaling logic is tricky
- Re-inventing the wheel is costly

👉 That's why we use:

- **Spark**
- **Flink**
- **Hadoop (older)**

as **black boxes**

1️⃣0️⃣ What is expected in System Design interviews?

You are **NOT expected** to:

- Design Spark internals
- Implement distributed schedulers

You **ARE expected** to:

- Know **when** big data tools are needed
- Understand **high-level architecture**
- Explain **why single machine won't work**

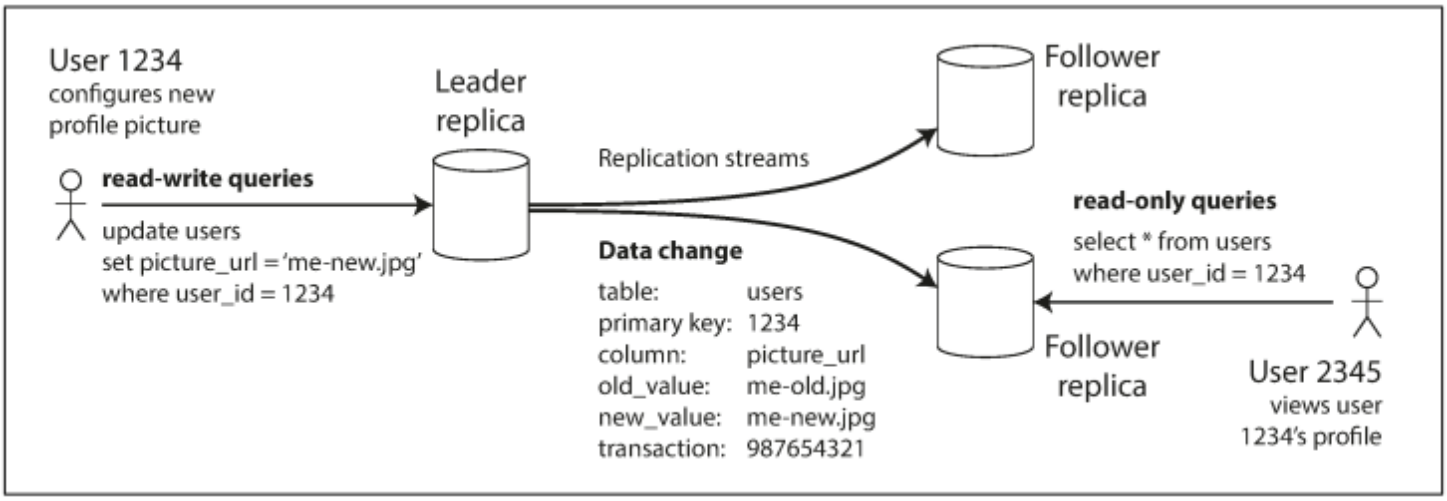
🔑 Interview-ready summary

Big data tools like Apache Spark process massive datasets using distributed systems. They divide data across multiple machines, process it in parallel using a coordinator–worker model, handle failures automatically, and allow developers to focus only on business logic instead of infrastructure.

🧠 **Memory tricks**

- **Big data ≠ single machine**
- **Coordinator divides work**
- **Workers compute**
- **Results are merged**
- **Spark = distributed engine**

Consistency — Deep Dive (easy language, real-life examples, when to choose what)



1️⃣ When does consistency even matter?

Consistency matters **only** when a system is **distributed AND stateful**.

🔹 **Stateful vs Stateless**

- **Stateless system**: doesn't store data
  👉 Most **application servers** (they just process requests)
- **Stateful system**: stores data for future use
  👉 **Databases**, caches with persistence, file systems

🔹 **Distributed**

- Data is stored on **multiple machines (nodes)**

- Usually for scaling, availability, or fault tolerance

📌 **Consistency = "Is the data the same on all nodes?"**
That's why we mostly talk about consistency for **databases**, not app servers.

2️⃣ What is Consistency? (simple definition)

Consistency means that every read returns the same value, no matter which replica (node) you read from.

3️⃣ Two main types of consistency

1. **Strong Consistency**
2. **Eventual Consistency**

Let's break them down 👇

4️⃣ Strong Consistency (strict & correct)

What it guarantees

- After a **write**, **all future reads** return the **latest value**
- All replicas **agree before** acknowledging the write
- Feels like there is **only one copy of data**

📌 Read-after-write is always correct.

Real-life example 🏦

Banking app

- You transfer ₹10,000
- Your balance must update **immediately**
- Showing old balance = ❌ unacceptable

When to choose Strong Consistency

✅ Banking systems
✅ Payment systems
✅ Stock trading apps
✅ Account balances

Money + correctness = Strong Consistency

5️⃣ Eventual Consistency (fast & available)

What it guarantees

- After a write, **replicas may differ temporarily**
- After some time, **all replicas converge**
- Prioritizes **availability & performance**

📌 Temporary inconsistency is allowed.

Real-life example 🧑‍💻

Social media

- You like a post
- Like count shows:
  - 101 on your phone
  - 100 on your friend's phone
- After some time → both show 101

👉 Totally acceptable ✅

When to choose Eventual Consistency

✅ Social media likes/comments
✅ Product catalogs
✅ Recommendations
✅ Analytics data

Scale + speed > immediate correctness

6️⃣ Strong vs Eventual (quick comparison)

Aspect	Strong Consistency	Eventual Consistency
Read correctness	Immediate	Delayed
Availability	Lower	Higher
Latency	Higher	Lower

Aspect	Strong Consistency	Eventual Consistency
Complexity	Higher	Lower
Use cases	Finance	Social apps

7️⃣ How do systems achieve Strong Consistency?

🔹 1. Synchronous Replication

- Write goes to **all replicas**
- ACK sent **only after all replicas update**

📌 Slower but safest

Example: **Google Spanner**

🔹 2. Quorum-Based Protocols (Strict)

In a system with **N nodes**:

- **W** = write quorum
- **R** = read quorum

For **strong consistency**:

$W + R > N$

👉 Ensures overlap between reads & writes

Examples:

- **Amazon DynamoDB** (configurable)
- **Apache Cassandra**

🔹 3. Consensus Algorithms

- Nodes **vote**
- Operation succeeds when **>50% nodes agree**

Common algorithm:

- **Raft** (easy to understand)

Used in:

- **Docker Swarm**

📌 Heavy but very reliable

8️⃣ How do systems achieve Eventual Consistency?

🔹 1. Asynchronous Replication

- Write is acknowledged immediately
- Replicas update **in background**

📌 Fast, but temporarily inconsistent

Examples:

- Cassandra
- MongoDB
- DynamoDB (default)

🔹 2. Relaxed Quorum

For eventual consistency:

$W + R \leq N$

👉 Reads may miss latest writes

🔹 3. Gossip Protocol

- Nodes periodically exchange updates
- Spread data like **word-of-mouth**

Used for:

- Replica sync
- Failure detection

Examples:

- Cassandra
- DynamoDB

🔹 4. Vector Clocks (awareness)

- Track **causal history**

- Help resolve conflicts

📌 Advanced topic (not interview-critical)

9️⃣ CAP theorem connection 🧠

In a distributed system:

- Network partitions **will happen**
- So you choose between:
  - **CP** → Strong Consistency
  - **AP** → Eventual Consistency

📌 You can't have **CAP** together.

1️⃣0️⃣ Choosing consistency (rule of thumb)

Ask yourself:

- ❓ Is wrong data acceptable temporarily?
- ❓ Is availability more important than correctness?

Decision:

- **Money / correctness** → Strong Consistency
- **User experience / scale** → Eventual Consistency

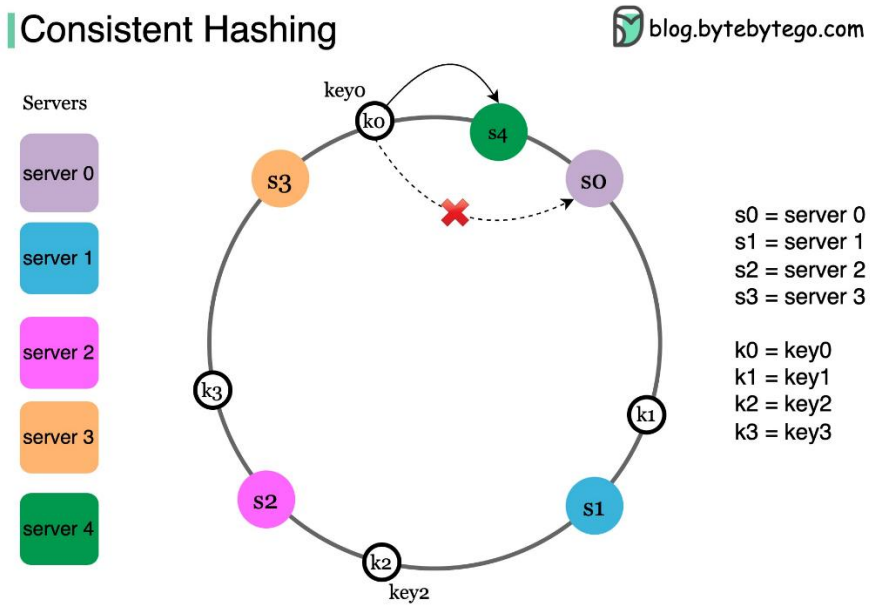
🔑 Interview-ready summary

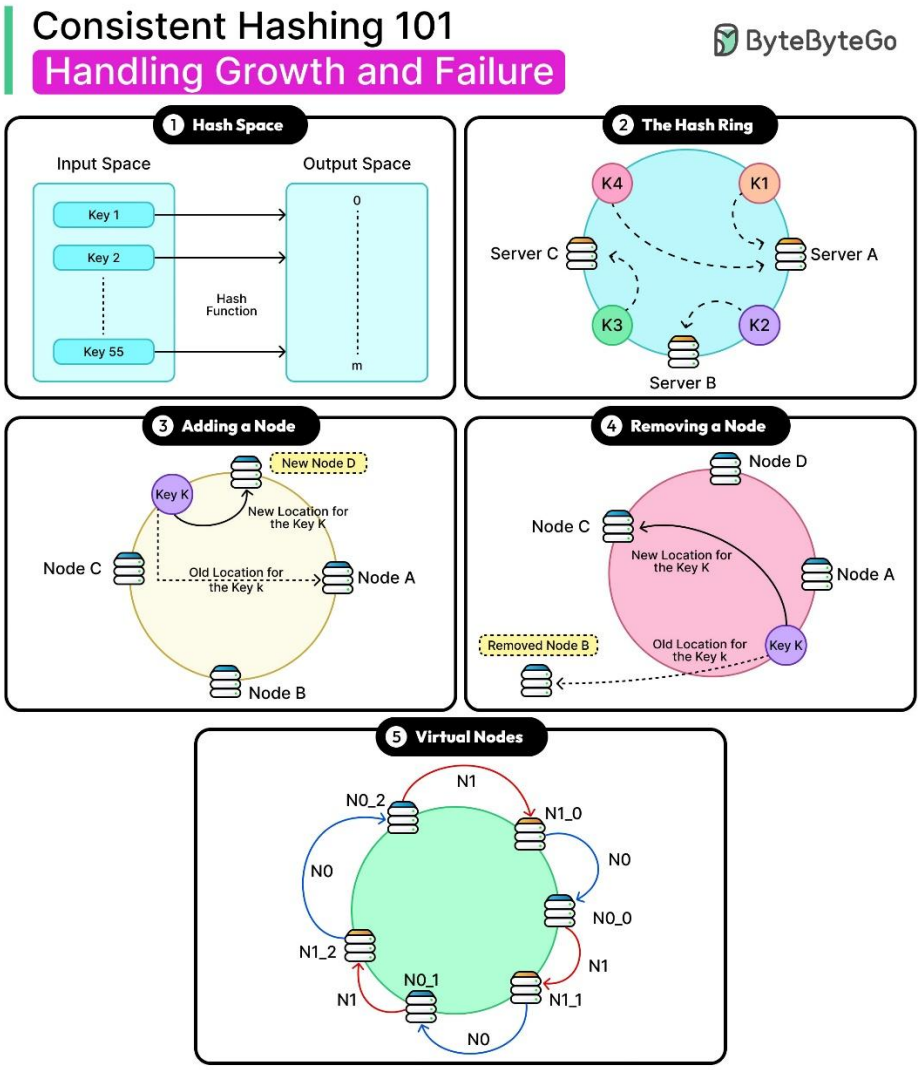
Consistency applies to distributed stateful systems and defines whether all replicas return the same data at any time. Strong consistency guarantees immediate correctness after writes, while eventual consistency allows temporary divergence in favor of higher availability and performance.

🧠 Memory tricks

- **DB = stateful**
- **Distributed + stateful = consistency problem**
- **Strong = correct now**
- **Eventual = correct later**
- **Money → strong**
- **Social → eventual**

Consistent Hashing — Deep Dive (easy language + intuition)





1️⃣ What is Consistent Hashing?

Consistent hashing is **just an algorithm** that answers one question:

"Given a key, which node (server) should store it?"

It's mainly used in **distributed, stateful systems** (like databases), where:

- Data is spread across **multiple machines**
- Machines can be **added or removed** dynamically

📌 App servers are usually **stateless**, so they don't need this.
📌 Databases are **stateful**, so they do.

2️⃣ Why the naive hashing approach breaks ❌

A common (but flawed) approach:

serverIndex = hash(key) % numberOfServers

Works only when servers are fixed

Example:

- Servers = 3
- hash(key1) % 3 = 1 → Server-1

Now autoscaling happens:

- Servers drop from **3 → 2**
- hash(key1) % 2 = 0 → Server-0

🚨 **Problem**:

- key1 moves to a different server
- This happens for **most keys**
- Causes **massive data movement**

👉 Not acceptable in large systems.

3️⃣ What Consistent Hashing fixes ✅

Consistent hashing ensures:

When servers change, only a small fraction of keys move.

That's the **entire goal**.

4️⃣ Core idea: Hash Ring 🧠

We imagine a **circular ring** representing the hash space.

Step-by-step:

1. Pick a hash function (e.g., SHA-128)
2. Hash outputs range:
3. [0, 2^128)
4. Place this range on a **circle (ring)**

5️⃣ Placing servers on the ring

- Take each **server's ID** (IP / name)
- Hash it
- Place the server at that position on the ring

Example:

- Node-1 → position A
- Node-2 → position B
- Node-3 → position C

Servers are **evenly scattered** around the ring.

6️⃣ Placing keys on the ring

- Take the **key**
- Hash it
- Place the key on the **same ring**

7️⃣ Rule to assign a key to a server

A key is stored on the nearest server in the clockwise direction.

That's it. That's the rule.

8️⃣ Example mapping (from your explanation)

Let's say the ring looks like this:

- **Node-1**
- **Node-2**
- **Node-3**

Keys:

- key-1, key-4 → Node-1
- key-3 → Node-2
- key-2, key-5 → Node-3

9️⃣ What happens when a node is removed? 🔥

Suppose **Node-2 is removed**.

Only:

- Keys that belonged to **Node-2** move
- key-3 moves to **Node-1**

All other keys:

- Stay where they are
- ❌ No mass reshuffling

📌 **Minimal data movement** — mission accomplished.

1️⃣0️⃣ Important clarification ⚠️

Consistent hashing:

- ❌ **Does NOT move data**
- ✅ **Only tells you where data should go**

You must:

- Copy data
- Rebalance manually
- Or let your DB do it internally

1️⃣1️⃣ Why this is perfect for autoscaling

When:

- Adding a node → only nearby keys move

- Removing a node → only that node's keys move

👉 This makes:

- Autoscaling cheap
- Failures manageable
- Rebalancing fast

1️⃣2️⃣ Where Consistent Hashing is used (real life)

It's used internally by distributed databases like:

- **Amazon DynamoDB**
- **Apache Cassandra**
- **Riak**

You don't usually code this yourself unless:

- You're building a database
- Or a distributed cache/router

1️⃣3️⃣ Key advantages (exam/interview points)

✅ Minimal key movement
✅ Handles dynamic servers
✅ Scales horizontally
✅ Fault-tolerant

🔑 Interview-ready summary

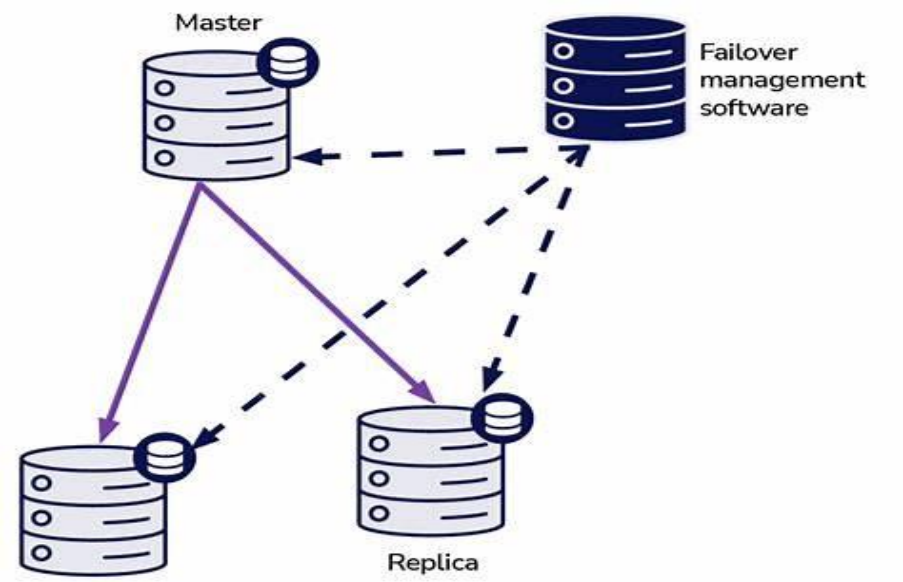
Consistent hashing is an algorithm used in distributed stateful systems to map keys to nodes such that when nodes are added or removed, only a minimal number of keys need to be reassigned, avoiding expensive rebalancing.

🧠 Memory tricks

- **Modulo hashing → bad for scaling**
- **Ring hashing → good for scaling**
- **Clockwise server wins**
- **Server change → minimal key movement**

Data Redundancy & Data Recovery

(Easy language + real-life examples + how it works in practice)



1️⃣ What does Data Redundancy mean?

Redundancy = multiple copies of the same data

Instead of keeping your data in **one database server**, you store **copies in multiple servers**.

📌 This is done mainly for **safety and reliability**.

2️⃣ Why do we need Data Redundancy?

Because **failures are guaranteed**. Not *if*, but *when*.

Real-life problems that can happen:

- 🌊 **Natural disasters** (flood, fire, earthquake)
- 💽 **Disk crash or corruption**
- ⚡ **Power failure**
- 👨‍💻 **Human mistakes** (accidental delete)
- 🐞 **Software bugs**

If data exists in **only one place**, then:
👉 Failure = **permanent data loss** ❌

Redundancy ensures:
👉 Failure = **system survives** ✅

3️⃣ Old-school backup methods (still valid)

Earlier, companies used **periodic backups**.

Common strategies:

- 🕛 **Daily backup** (night snapshots)
- 🗓️ **Weekly backup**
- 🗓️ **Monthly backup**

How it works:

- Take a snapshot of DB
- Store it on another server / storage

Problem:

- Data loss between backups
  (Example: backup at night, crash in evening)

📌 This is called **Recovery Point Objective (RPO)** gap.

4️⃣ Modern approach: Continuous Redundancy 🚀

Today, most systems use **continuous replication**.

Instead of:

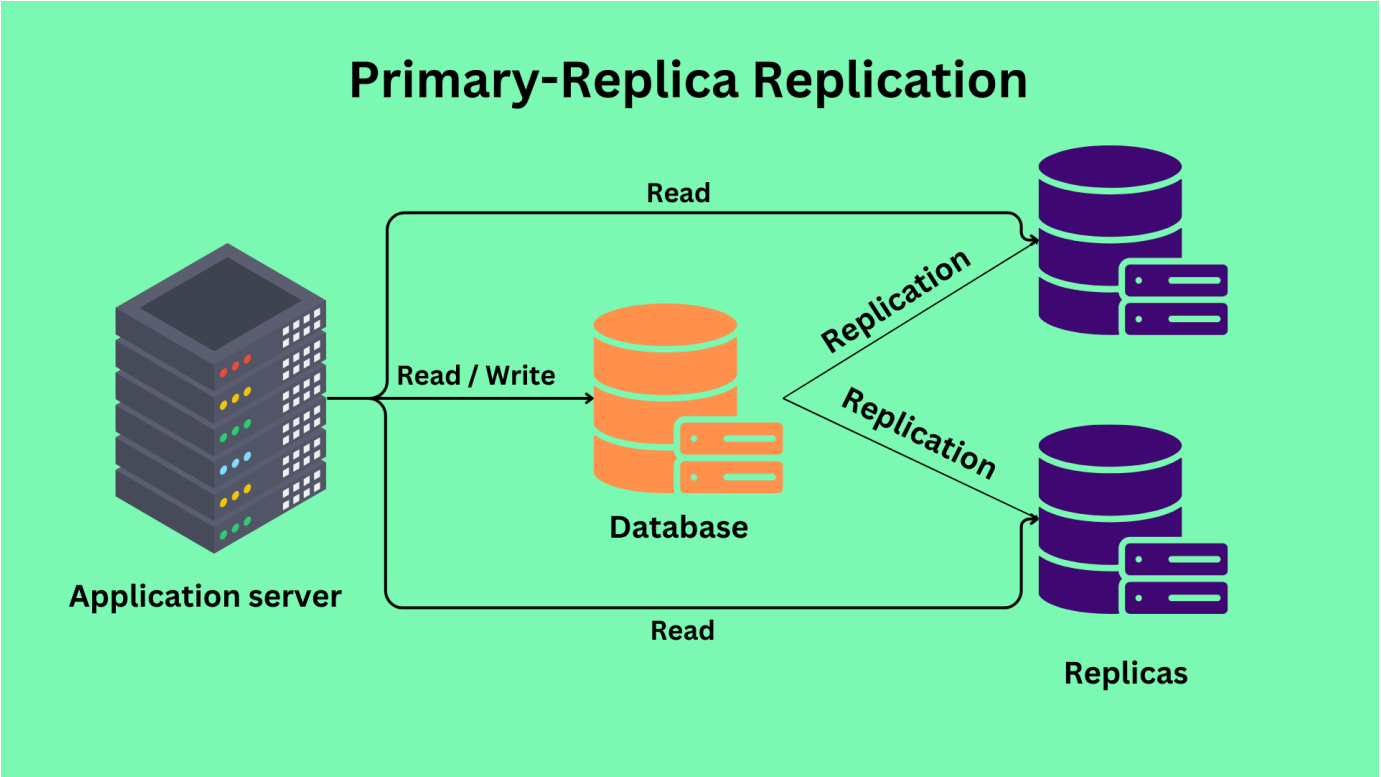
- Backup → wait → restore

We do:

- **Write once → copy immediately**

---

5 Primary–Replica (Main–Standby) model



Components:

- **Primary (Main) DB**
  - Handles all client reads & writes
- **Replica (Standby) DB**
  - Keeps an exact copy
  - No direct client traffic

---

6 How replication works

1. Client writes data to **Primary DB**
2. Data is copied to **Replica DB**
   - **Synchronous** (wait for replica)
   - **Asynchronous** (copy in background)

Replica's job:

"Always stay in sync with primary"

7️⃣ What happens when Primary DB fails? 🔥

- Replica is **promoted** to Primary
- Traffic is redirected
- Application keeps running

📌 This is called **Failover**

👉 User may not even notice the failure.

8️⃣ Synchronous vs Asynchronous replication

Type	How it works	Pros	Cons
Synchronous	Waits for replica ACK	No data loss	Higher latency
Asynchronous	ACK immediately	Faster	Small data loss possible

Rule of thumb:

- **Banking / Payments** → Synchronous
- **Social media / Analytics** → Asynchronous

9️⃣ Real-life analogy 🏦

Think of a **bank locker system**:

- One key with you
- One key with bank

If one is lost:
👉 Other key saves you

Replica DB = second key 🔑

1️⃣0️⃣ Where this is used in real life

Managed databases make this very easy:

- **Amazon RDS**
- **MySQL**
- **PostgreSQL**

📌 You just enable **Read Replica / Standby**, AWS handles rest.

1️⃣1️⃣ What redundancy gives you (key benefits)

✅ High availability
✅ Disaster recovery
✅ Minimal downtime
✅ Business continuity

1️⃣2️⃣ What redundancy does NOT replace

❌ It does not replace **backups**

Why?

- If you delete data accidentally
- Replication deletes it everywhere

👉 You still need **periodic backups**

1️⃣3️⃣ Best-practice combination 🧠

Modern systems use:

- **Continuous replication** (for availability)
- **Periodic backups** (for historical recovery)

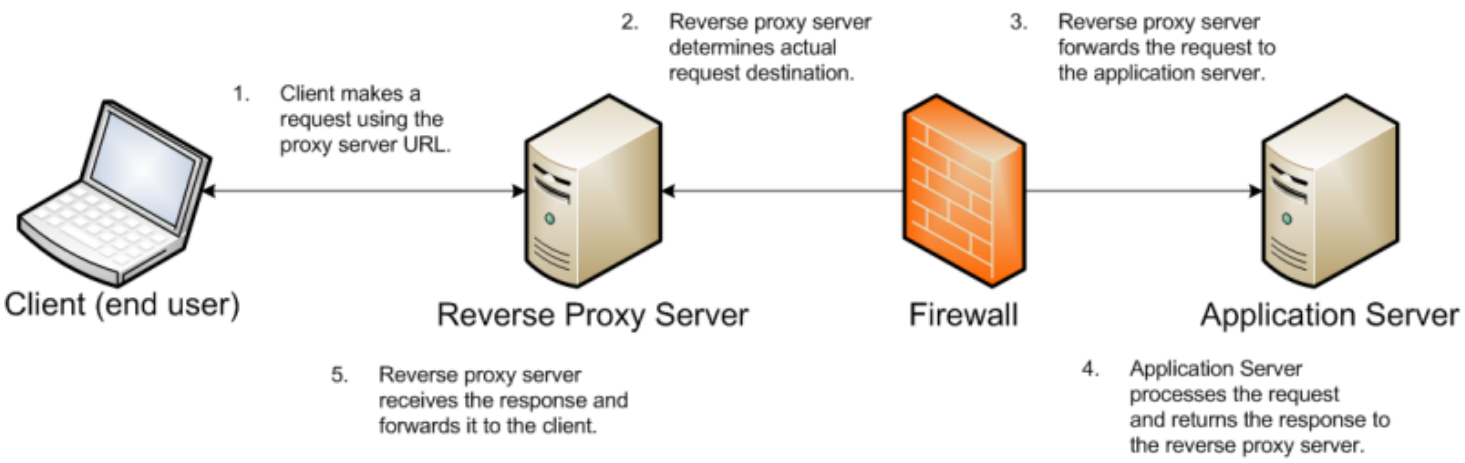
🔑 Interview-ready summary

Data redundancy means storing multiple copies of data across different machines to prevent data loss. Modern systems use continuous replication with primary–replica setups so that if the primary database fails, a replica can immediately take over, ensuring high availability and resilience.

🧠 Memory tricks

- **Redundancy = safety**
- **Primary fails → replica takes over**
- **Replication ≠ backup**
- **Money apps → synchronous**
- **Social apps → async**

Proxy (Easy explanation + real-life examples + practical view)



1️⃣ What is a Proxy?

A **proxy** is an **intermediary server** that sits **between a client and another server**.

Think of it as a **middleman**:

Client → Proxy → Server

You might wonder: *Why add an extra hop?*
Because proxies solve **privacy, performance, security, and scalability** problems.

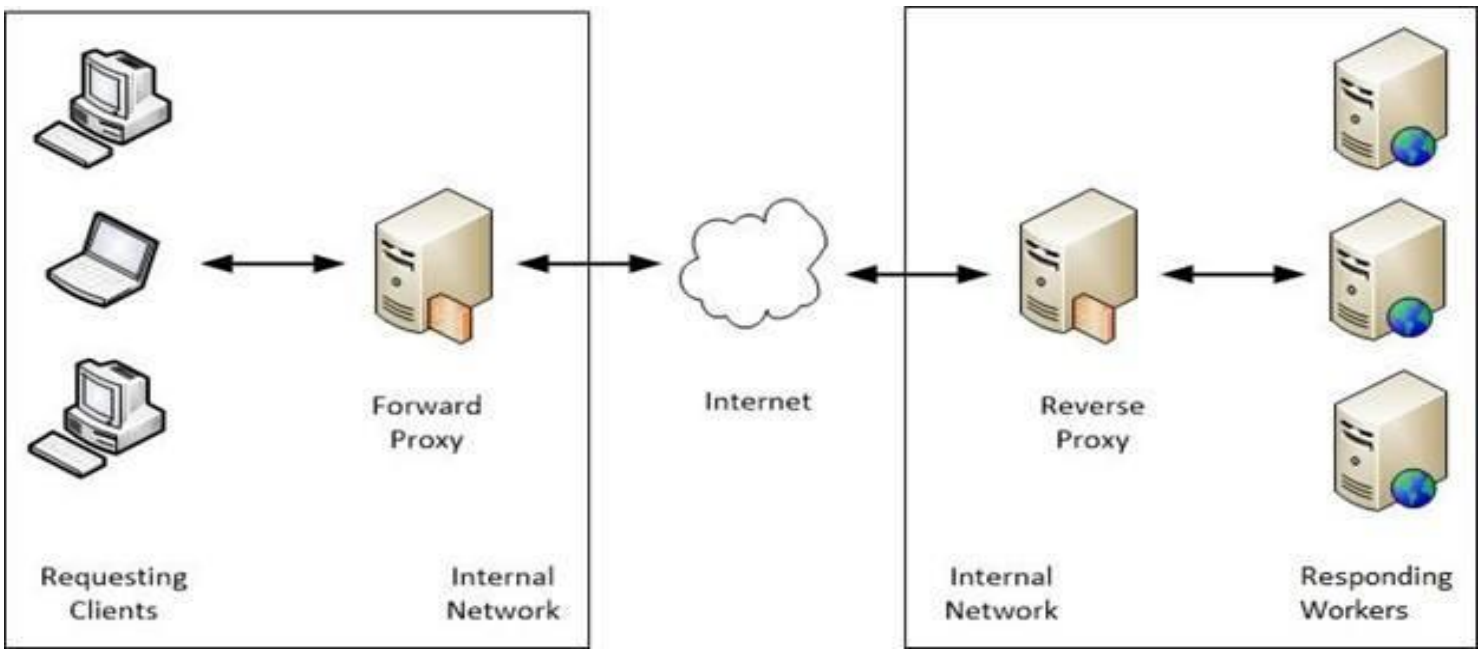
2️⃣ Types of Proxy

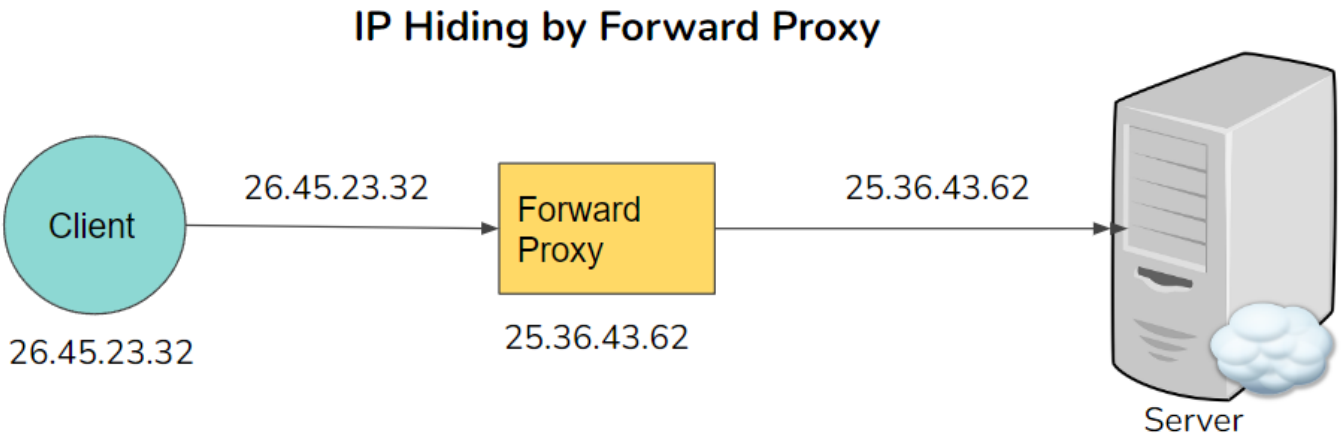
There are **two kinds**:

1. **Forward Proxy** → acts on behalf of the **client**
2. **Reverse Proxy** → acts on behalf of the **server**

Let's understand both with intuition.

3️⃣ Forward Proxy (Client-side proxy)





What it does

A **forward proxy hides the client**.

- Client sends request to proxy
- Proxy sends request to server
- Server only sees **proxy IP**, not client IP

📌 The server **does not know who the real client is**.

Real-life example 🌍

VPN

- You open a website via VPN
- Website sees VPN's IP
- Your real IP is hidden

👉 VPN = Forward Proxy

Use cases of Forward Proxy

- 🌐 Access geo-blocked content
- 🔒 Hide client identity
- ⚡ Client-side caching
- 🏢 Internet control in offices/colleges

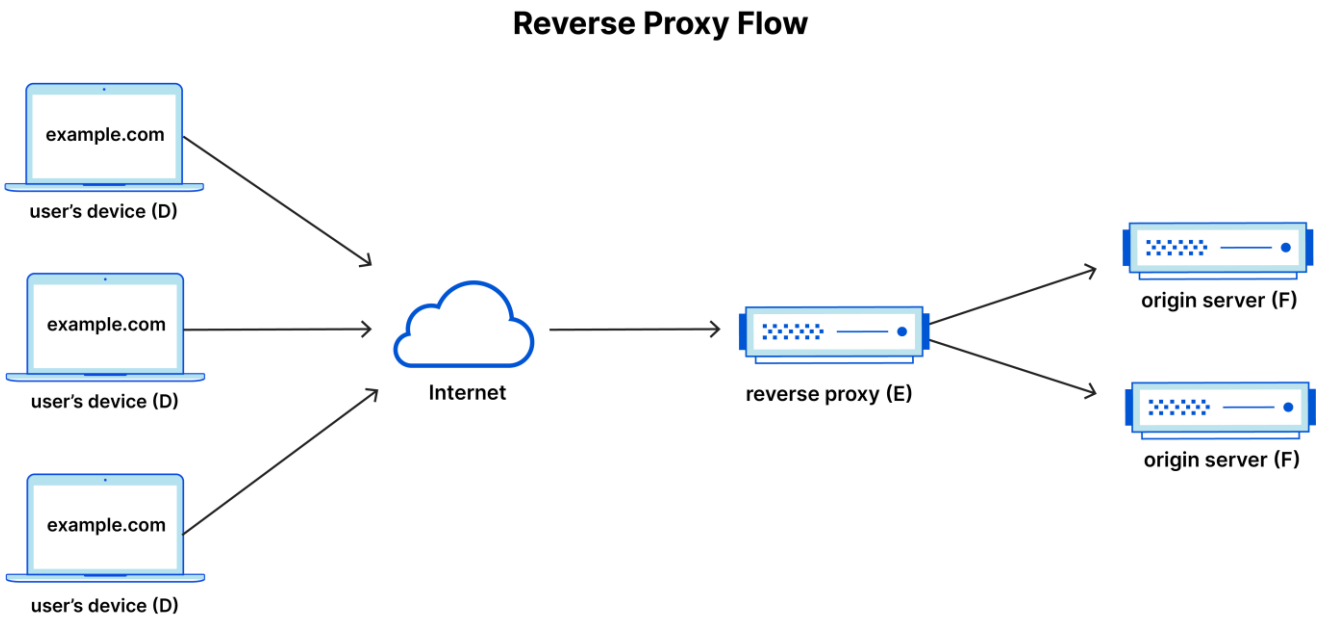
Simple flow

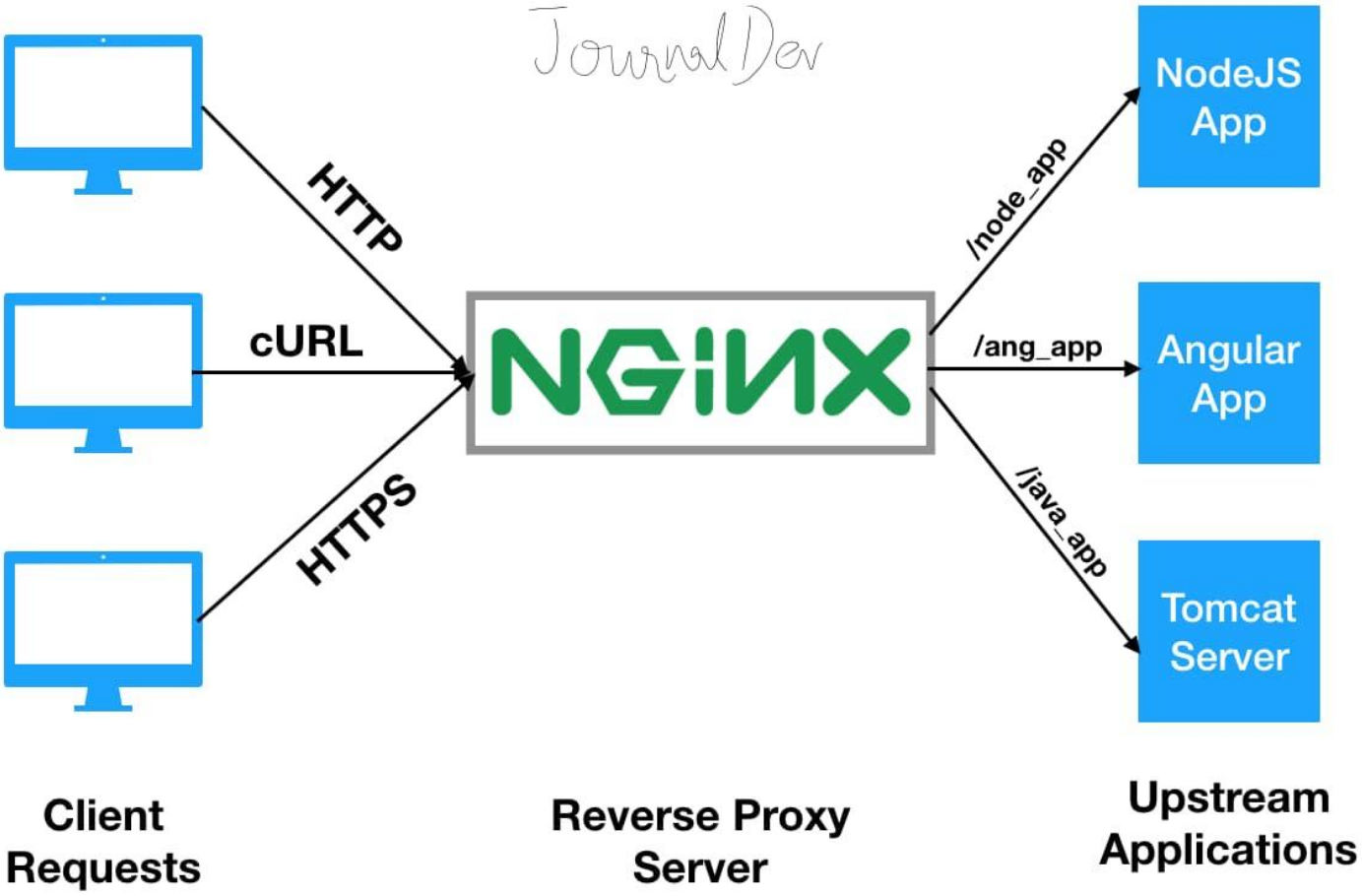
Client → Forward Proxy → Server

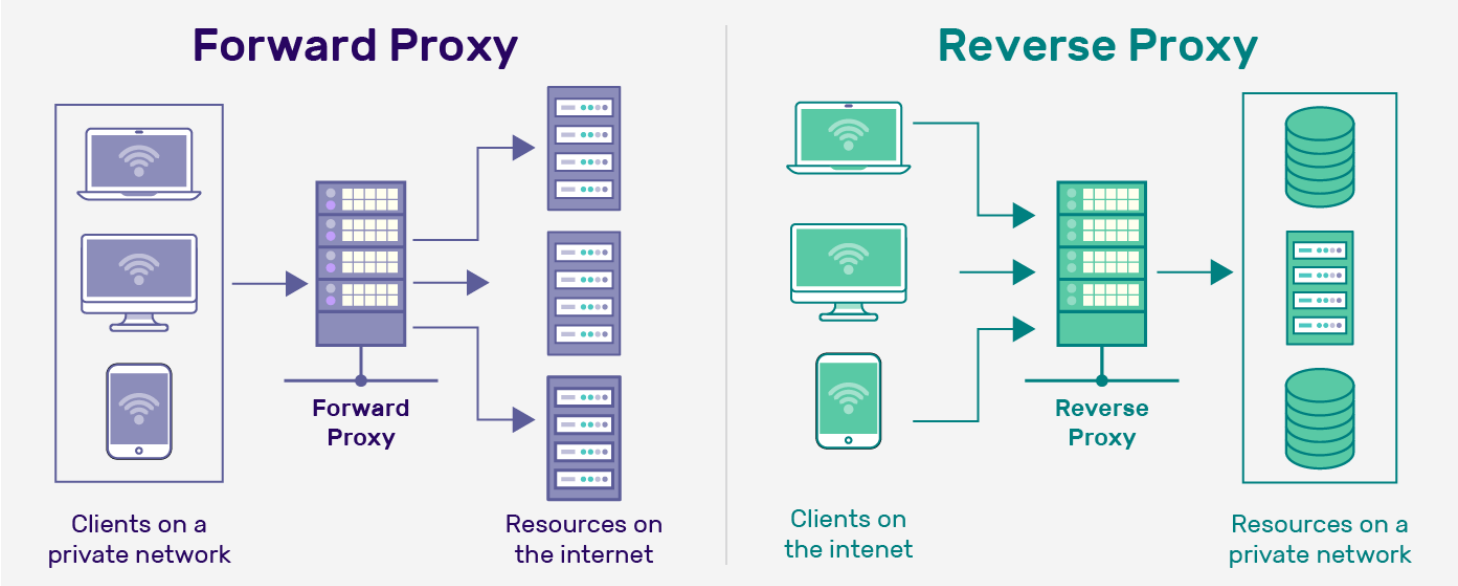
Server → Forward Proxy → Client

📌 In **system design interviews**, we usually **don't focus much on forward proxies**, because they are **client-side concerns**.

🔽 Reverse Proxy (Server-side proxy) ⭐ IMPORTANT







What it does

A **reverse proxy hides the backend servers**.

- Client sends request to proxy
- Proxy decides **which backend server** to forward to
- Client never sees real server IPs

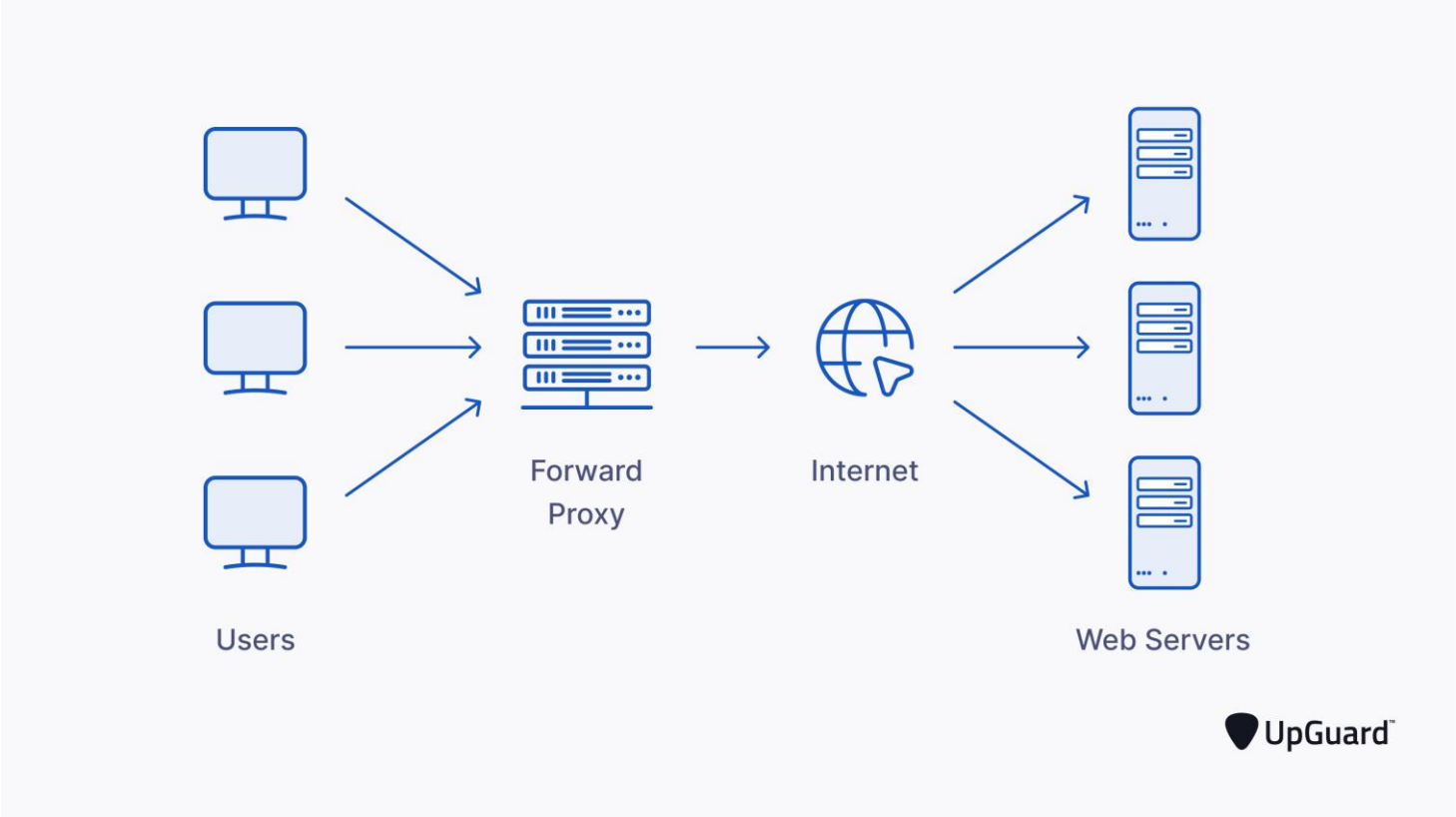
📌 The **client doesn't know which server handled the request**.

Main feature

Hides the server infrastructure

5️⃣ Load Balancer is a Reverse Proxy

A **load balancer** is actually a **specialized reverse proxy**.



- Client talks to load balancer
- Load balancer forwards request to backend servers
- Servers stay private & protected

6️⃣ Use cases of Reverse Proxy

Reverse proxies are everywhere in backend systems:

🔹 Load balancing

Distribute traffic across servers

🔹 SSL termination

- HTTPS handled at proxy
- Backend servers deal with plain HTTP

- Reduces CPU load

🔹 **Caching**

- Cache static content
- Reduce backend load

🔹 **Security**

- Backend servers not directly exposed
- Acts as first defense layer

7️⃣ Popular Reverse Proxy tools

In real production, we **never write our own proxy**.

We use battle-tested tools like:

- **Nginx**
- **HAProxy**

These tools provide:

- High performance
- Security
- SSL handling
- Caching
- Monitoring

8️⃣ Forward vs Reverse Proxy (Quick comparison)

Aspect	**Forward Proxy**	**Reverse Proxy**
Acts for	Client	Server
Hides	Client IP	Server IP
Used by	Clients	Backend systems

Aspect	Forward Proxy	Reverse Proxy
Example	VPN	Load balancer
Interview focus	Low	High ⭐

9️⃣ Building a simple Reverse Proxy (learning purpose)

To **understand the concept**, here's a **simple Node.js reverse proxy**.

⚠️ This is for learning only. In production → use **Nginx / HAProxy**.

Scenario

- /product → Product Service (:5001)
- /order → Order Service (:5002)
- Client talks only to proxy (:8080)

Install dependency

npm init -y

npm install http-proxy

Reverse Proxy Code

How it works

- Client → http://localhost:8080/product
- Proxy → forwards to http://localhost:5001
- Client never sees backend server

1️⃣0️⃣ Why we don't reinvent the wheel

Yes, you *can* code:

- Load balancers
- Reverse proxies
- Message brokers

But in real systems:

- These are **hard problems**
- Already solved optimally
- Maintained by experts

👉 We use them as **black boxes**.

🔑 Interview-ready summary

A proxy is an intermediary server. A forward proxy acts on behalf of the client and hides the client identity, while a reverse proxy acts on behalf of the server and hides backend infrastructure. Reverse proxies are widely used for load balancing, security, SSL termination, and caching in backend systems.

🧠 Memory tricks

- **Forward proxy → hides client**
- **Reverse proxy → hides server**
- **VPN = forward proxy**
- **Load balancer = reverse proxy**
- **Backend interviews → reverse proxy**

COMPLETE SYSTEM DESIGN SUMMARY (BEGINNER → INTERMEDIATE)

1️⃣ Why System Design Matters

In college projects, you usually build:

Client → Backend → Database

This works for:

- 10 users
- Prototypes
- College demos

❌ Real-world problem

When users grow to **millions or billions**:

- Server crashes
- DB becomes slow
- Data loss risk
- Security & scaling issues

✅ **System Design solves:**

- Scalability
- Reliability
- Availability
- Performance
- Fault tolerance

📌 **System design is about building systems that don't break when traffic grows.**

2️⃣ Client, Server & Deployment

Server

- A server is just a **computer** running your code
- localhost = 127.0.0.1
- Port identifies the application (:8080, :443)

DNS

- Domain name → IP address
- Easier than remembering IPs

Deployment

- Instead of your laptop → use cloud
- Cloud gives:
  - Public IP
  - Virtual machines
  - Reliability

📌 Example: AWS EC2

3️⃣ Latency & Throughput

Latency

- Time taken for **one request**
- Measured in **milliseconds**
- Lower latency = faster app

Throughput

- Number of requests handled per second
- Measured in **RPS / TPS**

Ideal system

- ✅ Low latency
- ✅ High throughput

🚗 Analogy:

- Latency = travel time of one car
- Throughput = number of cars per hour

4️⃣ Scaling

Scaling = handling more load

Vertical Scaling

- Increase CPU / RAM of same machine
- Easy
- Limited

Horizontal Scaling

- Add more machines
- Needs load balancer
- Most common in real systems

📌 **Rule**:
Always try **vertical first**, then horizontal.

5️⃣ Load Balancer

A **load balancer**:

- Sits between client & servers
- Distributes traffic

Why needed?

Clients:

- Are not smart
- Cannot choose server

Load Balancer Algorithms

- Round Robin
- Weighted Round Robin
- Least Connections
- Hash-based routing

📌 Load balancer is a **reverse proxy**

6️⃣ Auto Scaling

Traffic is not constant.

Problem

- Running max servers always = 💸 waste

Solution

- Automatically add/remove servers
- Based on CPU / memory / traffic

📌 Example: AWS Auto Scaling Group

7️⃣ Back-of-the-Envelope Estimation

Used in interviews to:

- Estimate servers
- Estimate storage
- Estimate load

What you calculate

- DAU
- Reads per day
- Writes per day
- Storage per day
- CPU & servers needed

📌 Approximation > exact numbers

8️⃣ Databases & State

Stateless

- App servers
- Can restart anytime

Stateful

- Databases
- Store data permanently

📌 **Consistency only matters for stateful systems**

9️⃣ CAP Theorem

In distributed systems, you can guarantee only **2 out of 3**:

- **Consistency**
- **Availability**
- **Partition Tolerance**

Reality

- Network failures WILL happen
- So P is mandatory

Choice

- **CP** → Banking, payments
- **AP** → Social media

🔟 Database Scaling Journey

Step-by-step (IMPORTANT)

1. Vertical scaling
2. Indexing
3. Partitioning
4. Master–Slave
5. Multi-Master

6. Sharding (last resort)

1️⃣1️⃣ Indexing

- Avoid full table scan (O(N))
- Use B-trees (O(log N))
- Faster reads

📌 Index = book index

1️⃣2️⃣ Partitioning

- Split table into smaller tables
- Same DB server
- Faster index lookup

📌 DB handles routing automatically

1️⃣3️⃣ Master–Slave Architecture

- **Master** → writes
- **Slaves** → reads

Benefit

- Read scalability

Downside

- Write bottleneck

1️⃣4️⃣ Multi-Master

- Multiple write nodes
- Used for geo-distribution

Challenge

- Conflict resolution

📌 Very complex, use carefully

1️⃣5️⃣ Sharding

- Split data across **different DB servers**
- Based on sharding key

Pros

- Massive scale

Cons

- App-level routing
- Expensive joins
- Hard consistency

📌 Avoid unless absolutely needed

1️⃣6️⃣ SQL vs NoSQL

SQL

- Structured schema
- ACID
- Strong consistency
- Complex joins

NoSQL

- Flexible schema
- Horizontal scaling
- Eventual consistency

Rule

- Money → SQL
- Scale → NoSQL

1️⃣7️⃣ Microservices

Monolith

- One big app
- Simple start
- Hard to scale teams

Microservices

- Small independent services

- Independent scaling
- Team-based architecture

📌 Use API Gateway

1️⃣8️⃣ API Gateway

- Single entry point
- Routing
- Auth
- Rate limiting
- Caching

📌 Client talks to gateway, not services

1️⃣9️⃣ Caching

Why?

DB is slow, cache is fast.

Where?

- Client-side
- Server-side (Redis)
- CDN
- Application-level

Cache concepts

- Cache hit
- Cache miss
- TTL
- Invalidation

2️⃣0️⃣ Redis

- In-memory key-value store
- Very fast
- Limited memory

Use cases

- Caching
- Session storage
- Queues
- Pub/Sub

📌 Redis ≠ database replacement

2️⃣1️⃣ Blob Storage

Why not DB?

- Files are huge
- DB becomes slow

Blob storage

- Stores binary data
- Cheap
- Durable

📌 Example: AWS S3

2️⃣2️⃣ CDN

- Caches static files near users
- Reduces latency

Flow

- User → Edge server
- Cache miss → Origin
- Cache hit → Serve fast

2️⃣3️⃣ Message Broker

Synchronous

- Client waits
- Not good for long tasks

Asynchronous

- Background processing
- Use message broker

Components

- Producer
- Broker
- Consumer

2️⃣4️⃣ Message Queue

- One message → one consumer
- Message deleted after processing

📌 Example: RabbitMQ, SQS

2️⃣5️⃣ Message Stream

- One message → many consumers
- Messages retained

📌 Example: Kafka

2️⃣6️⃣ Kafka

- High throughput
- Topics, partitions
- Consumer groups
- Horizontal scaling via partitions

📌 Used for event streaming & big data

2️⃣7️⃣ Pub/Sub (Realtime)

- Broker pushes message
- No storage

📌 Used in chat systems

2️⃣8️⃣ Event-Driven Architecture (EDA)

- Producer emits event
- Consumers react asynchronously

Patterns

- Simple Event Notification
- Event-Carried State Transfer

2️⃣9️⃣ Distributed Systems

- Multiple machines
- One logical system

Leader–Follower model

- Leader assigns work
- Followers execute

📌 Leader election required

3️⃣0️⃣ Auto-Recoverable Systems

- Orchestrators monitor servers
- Leader election for orchestrators
- Self-healing systems

📌 Example: Kubernetes

3️⃣1️⃣ Big Data Tools

- Data too large for one machine
- Distributed processing

Architecture

- Coordinator
- Workers

📌 Example: Apache Spark

3️⃣2️⃣ Consistency

Strong Consistency

- Read after write = latest data

Eventual Consistency

- Data converges later

📌 Money → strong
📌 Social → eventual

3️⃣3️⃣ Consistent Hashing

- Maps keys → nodes
- Minimal data movement
- Uses hash ring

📌 Used in DynamoDB, Cassandra

3️⃣4️⃣ Data Redundancy & Recovery

Why?

- Hardware failure
- Natural disasters

Modern approach

- Primary + Replica
- Automatic failover

📌 Replication ≠ backup

3️⃣5️⃣ Proxy

Forward Proxy

- Hides client
- VPN example

Reverse Proxy

- Hides server
- Load balancer
- SSL termination
- Security

📌 Backend focus = reverse proxy