Scaling 0 to millions of User

Phase-1

Step 1: Actors involved (0 users stage)

At the very beginning, we have **4 main parts**:

1. **Client** → Browser / Mobile App
2. **DNS** → Domain Name System
3. **Server** → One backend server
4. **Database** → One database (optional at start)

Step 2: Client (User side)

Example:

- User opens browser
- Types URL:

www.myapp.com

The **client does NOT know IP address**, it only knows the domain name.

Step 3: DNS – Domain Name System

DNS's job:

Convert **domain name → IP address**

How DNS works (simple flow):

1. Client asks:
2. What is IP of www.myapp.com ?
3. DNS replies:
4. www.myapp.com → 13.235.120.45

Now client knows **where the server is**.

Step 4: Client → Server (Request)

After getting IP:

Client sends HTTP/HTTPS request:

GET /login

Host: www.myapp.com

This request goes to:

13.235.120.45

Step 5: Server (Single Server – Monolith)

At **0 users → few users**, you usually have:

- One **EC2 / VM / Physical server**
- Everything runs here:
  - Backend code (Node / Java / Python)
  - API logic
  - Authentication
  - Sometimes frontend too

Example:

Single Server

├── Backend App

├── Frontend (optional)

└── Database (sometimes same machine)

Step 6: Server → Database (if needed)

If request needs data:

Server → Database

Database → Server

Example:

- **Login**
- **Fetch user profile**
- **Save form data**

Step 7: Response back to Client

Server sends response:

200 OK

HTML / JSON / Data

Client renders:

- Web page
- App screen

Step 8: Initial Architecture Diagram (Text)

User (Browser/App)

|

v

DNS

|

v

Single Server (IP)

|

v

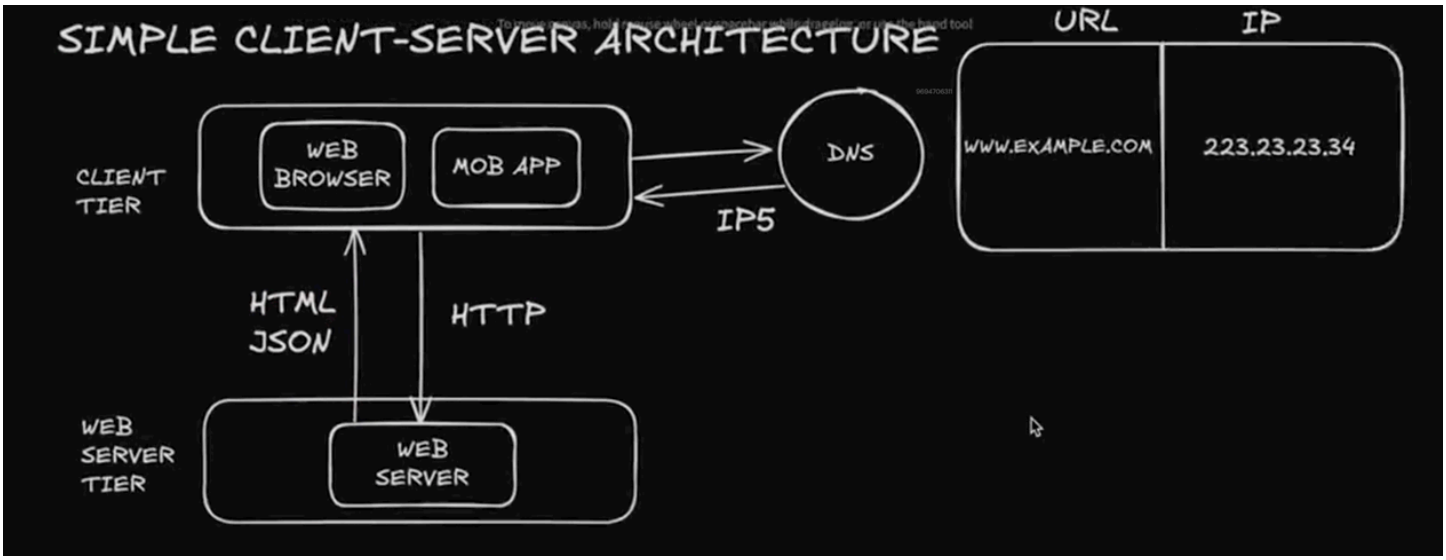
Database

Why this model works for 0 → few thousand users

1.Simple
2.Cheap
3.Easy to manage
4.Single point of failure
5.Limited scaling

What breaks first when users increase?

- Server CPU overload
- Memory issue
- Too many DB connections
- Downtime if server crashes



1. HTTP Request = What client sends to server

An HTTP request is made of **4 main parts**:

1. **Request Line**
2. **Headers**
3. **Body** (optional)
4. **Query Params / Path Params** (part of URL)

2.Request Line (Start Line)

Format:

METHOD PATH HTTP/VERSION

Example:

GET /users/42 HTTP/1.1

Contains:

- **Method** → What action you want
- **Path** → Resource on server
- **HTTP version**

Common HTTP Methods:

Method	Meaning
GET	Read data
POST	Send new data
PUT	Update full data
PATCH	Update partial data
DELETE	Remove data

3.URL Parts (Very Important)

Example URL:

https://api.myapp.com:443/users?id=10

Breakdown:

- **Protocol** → https
- **Domain** → api.myapp.com
- **Port** → 443 (default for HTTPS)
- **Path** → /users
- **Query String** → ?id=10

4.Headers (Metadata about request)

Headers are **key–value pairs**.

Example:

Host: api.myapp.com

User-Agent: Chrome/143.0

Accept: application/json

Content-Type: application/json

Authorization: Bearer <token>

Cookie: sessionId=abc123

Common Headers (with reason):

Header	Why it’s used
Host	Which domain is requested
User-Agent	Client info
Accept	Expected response format
Content-Type	Format of request body
Authorization	Auth token
Cookie	Session data
Content-Length	Size of body

5.Request Body (Optional)

Used mainly with:

- POST
- PUT
- PATCH

Example **JSON** body:

{

"email": "vijay@gmail.com",

"password": "123456"

}

GET requests usually **do NOT have body**.

6.Query Parameters

Used to send small data in URL.

Example:

GET /search?keyword=phone&page=2

Parsed as:

keyword = "phone"

page = 2

7.Path Parameters

Used to identify a specific resource.

Example:

GET /users/42

Here:

42 → userId

8.Complete Raw HTTP Request Example

POST /login HTTP/1.1

Host: api.myapp.com

Content-Type: application/json

Authorization: Bearer xyz123

Content-Length: 58

{

"email": "vijay@gmail.com",

"password": "123456"

}

9.Visual Structure

HTTP Request

├── Request Line

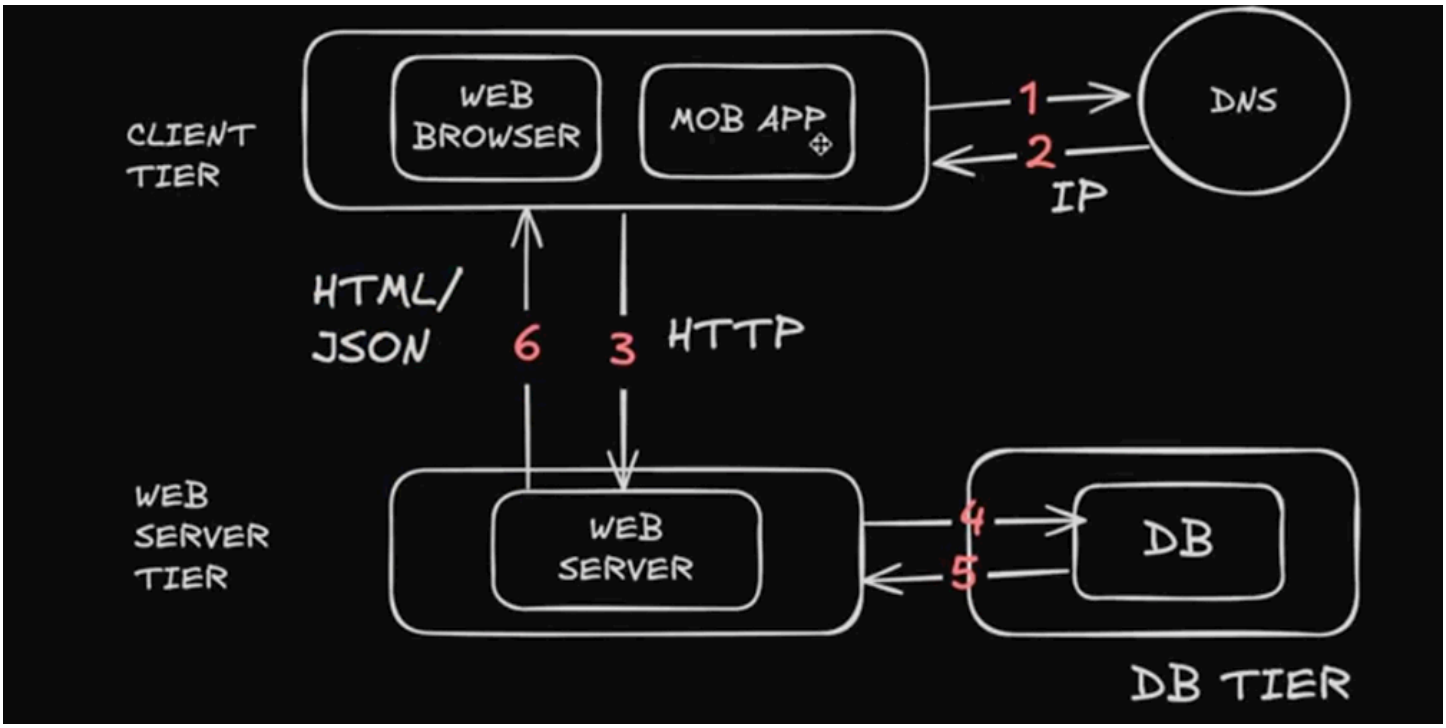
├── Headers

├── Blank Line

└── Body (optional)

10.Why this matters for scaling?

- Headers help **load balancer routing**
- Authorization helps **security**
- Body size affects **performance**
- Query params affect **cacheability**



First rule (MOST IMPORTANT)

There is **NO single “best database”**
The **best DB depends on your use-case**

Step 1: Main types of Databases

1.Relational Database (SQL)

Examples:

- **PostgreSQL** (best overall)
- MySQL
- MariaDB
- Oracle

2.NoSQL Databases

Type	Example
Key–Value	Redis, Amazon DynamoDB
Document	MongoDB
Wide-Column	Cassandra DB
Graph	Neo4j,graph Ql

Step 2: Start from 0 users (Best choice)

1.Use **PostgreSQL**

Why PostgreSQL is BEST at start?

✔ Strong data consistency
✔ ACID transactions
✔ Supports complex queries
✔ Indexing, joins, constraints
✔ Scales surprisingly well
✔ Used by big companies (Uber, Instagram, GitHub)

90% applications should start with PostgreSQL

Step 3: When PostgreSQL is PERFECT

Use PostgreSQL if you have:

- **Users**
- **Orders**
- **Payments**
- **Relationships (user ↔ order ↔ product)**
- **Financial data**
- **Login / auth**

Example schema:

Users

Orders

Payments

Products

Step 4: When NOT to use MongoDB first

Many beginners choose MongoDB — **bad idea initially**.

MongoDB problems at scale:

- **Weak joins**
- **No strong constraints**
- **Data duplication**
- **Hard migrations**
- **Eventual consistency issues**

MongoDB is good **only when schema is flexible**

Step 5: Scaling path (0 → 1 million users)

Stage 1: 0 → 50k users

PostgreSQL (Single instance)

Stage 2: 50k → 200k users

PostgreSQL

├── Read Replicas

└── Indexing + Connection Pooling

Stage 3: 200k → 1M users

PostgreSQL

├── **Primary (writes)**

├── **Read replicas (reads)**

├── **Partitioning**

└── **Cache (Redis)**

Step 6: Add Redis (NOT a replacement)

Redis is NOT a main DB

Redis is a **cache**

Use Redis for:

- **Sessions**
- **JWT blacklist**
- **OTP**
- **Rate limiting**
- **Frequently read data**

Example:

Client → Redis → DB (fallback)

Step 7: When to add MongoDB

Use MongoDB when:

- **Logs**
- **Analytics events**
- **Chat messages**
- **Schema changes often**

Example:

User Activity Logs

Clickstream Data

Step 8: High-Scale databases (Million+ users)

Use Case	Best DB
User data	PostgreSQL
Cache	Redis
Logs	MongoDB
Time-series	InfluxDB
Search	Elasticsearch
Huge writes	Cassandra

Step 9: Simple Decision Table

Requirement	DB
Strong consistency	PostgreSQL
Fast reads	Redis
Flexible schema	MongoDB
Graph relations	Neo4j
Search	Elasticsearch

Final Recommendation (Very clear)

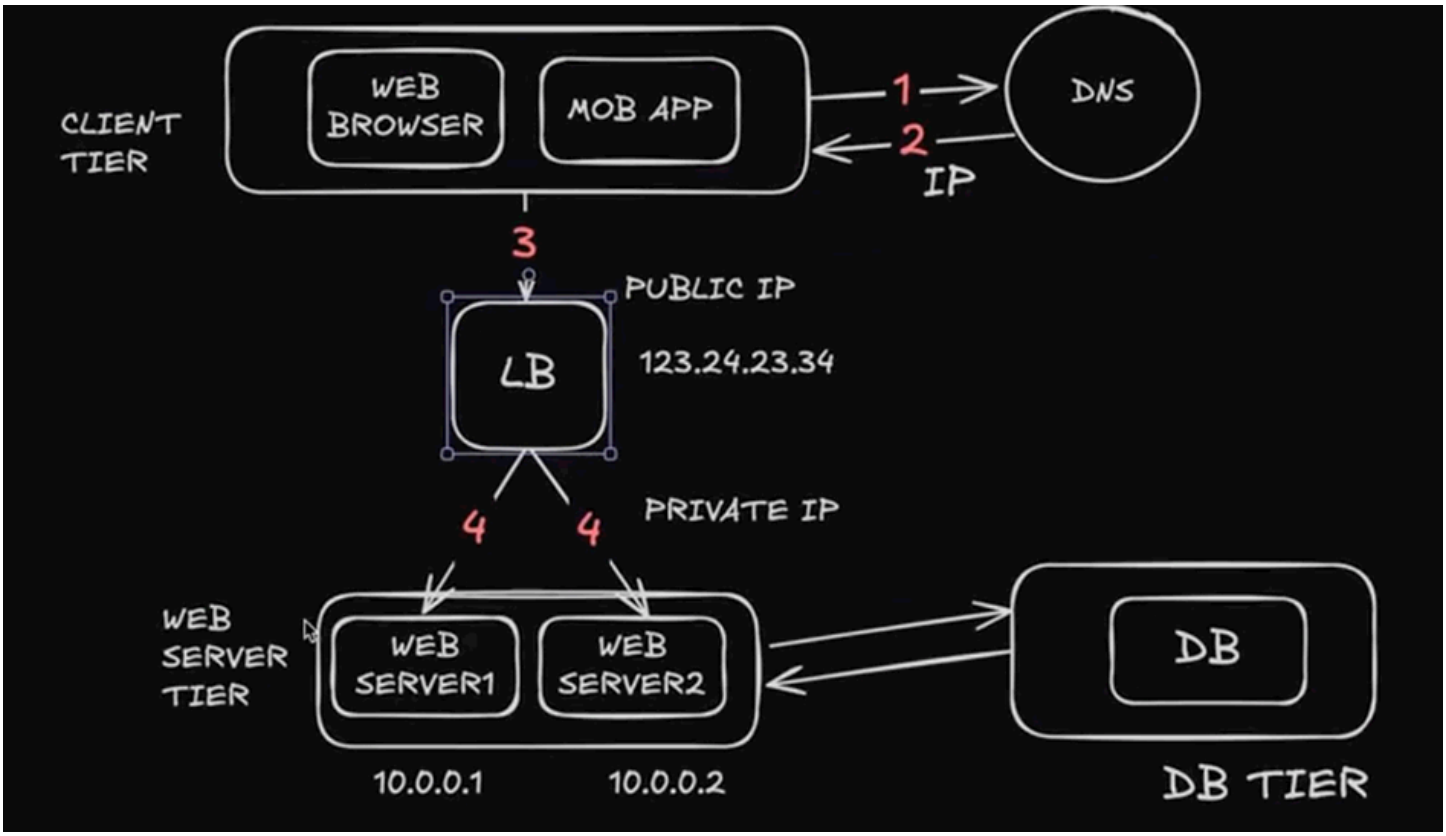
->Start with:

PostgreSQL + Redis

->Don’t over-engineer early

PHASE-2

LOAD BALANCER



1.What is a Load Balancer?

Definition (simple):

A **Load Balancer (LB)** sits between **client** and **servers** and **distributes incoming traffic** across **multiple servers**.

Instead of:

Client → Server

We get:

Client → Load Balancer → Multiple Servers

Why do we NEED a Load Balancer?

Problem without Load Balancer

Client → Single Server

Problems:

- **Server overload**
- **App crash = total downtime**
- **No scaling**
- **No high availability**

3.Problems Load Balancer Solves

1. Traffic Overload

Distributes traffic evenly

2. Single Point of Failure

If one server dies → traffic goes to others

3. Horizontal Scaling

Add/remove servers easily

4. Zero Downtime Deployment

Deploy one server at a time

5. Health Monitoring

Stops sending traffic to bad servers

4.Architecture Before vs After

Before (Phase-1)

Client

|

DNS

|

Server

After (Phase-2)

Client

|

DNS

|

Load Balancer

|

├── Server-1

├── Server-2

└── Server-3

5.What happens in real request flow?

Step-by-step:

1. Client hits:

www.myapp.com

1. DNS resolves:

www.myapp.com → Load Balancer IP

1. Request goes to **Load Balancer**
2. Load Balancer selects **healthy server**
3. Server processes request
4. Response → LB → Client

6.Types of Load Balancers (VERY IMPORTANT)

1.Layer-4 Load Balancer (Transport Layer)

Works on:

- IP
- Port
- TCP/UDP

Example:

LB sees: IP + Port only

Pros:

- Very fast
- Low latency

Cons:

- No header inspection
- No smart routing

Examples:

- AWS NLB
- HAProxy (L4 mode)

2.Layer-7 Load Balancer (Application Layer)

Works on:

- URL
- Headers
- Cookies
- HTTP method

Example:

/api → Backend

/images → CDN

/admin → Auth Server

Pros:

- Smart routing
- SSL termination

- Caching
- Auth handling

Cons:

- Slightly slower than L4

Examples:

- NGINX
- AWS ALB
- Envoy

7.Load Balancing Algorithms

How LB decides **which server** gets request:

1.Round Robin

Req1 → S1

Req2 → S2

Req3 → S3

2.Least Connections

Send to server with least active users

3.IP Hash

Same IP → same server

4.Weighted Round Robin

More powerful server → more traffic

8.Health Checks (CRITICAL)

LB continuously checks servers:

GET /health

If server returns:

- 500 / timeout → removed
- 200 → kept active

This avoids:

- Sending traffic to dead server

Sticky Sessions (Session Affinity)

Problem:

User logs in on Server-1
Next request goes to Server-2

Solution:

Sticky session via:

- Cookie
- IP hash

BUT:

For large scale → **store session in Redis**

10.SSL Termination

Without LB:

Every server handles HTTPS

With LB:

Client → HTTPS → LB → HTTP → Server

Benefits:

- Faster servers
- Easier certificate management

11.What things exist INSIDE a Load Balancer?

Internals:

- Listener (Port 80/443)
- Target groups
- Routing rules
- Health checks
- SSL certificates
- Algorithms
- Logs & metrics

Real-World Example (Instagram-like)

DNS

↓

Global Load Balancer

↓

Regional Load Balancer

↓

App Servers

↓

Cache / DB

12.Common Load Balancers

Platform	Load Balancer
AWS	ALB, NLB
GCP	Cloud Load Balancing
Azure	Azure LB
Self-hosted	NGINX, HAProxy

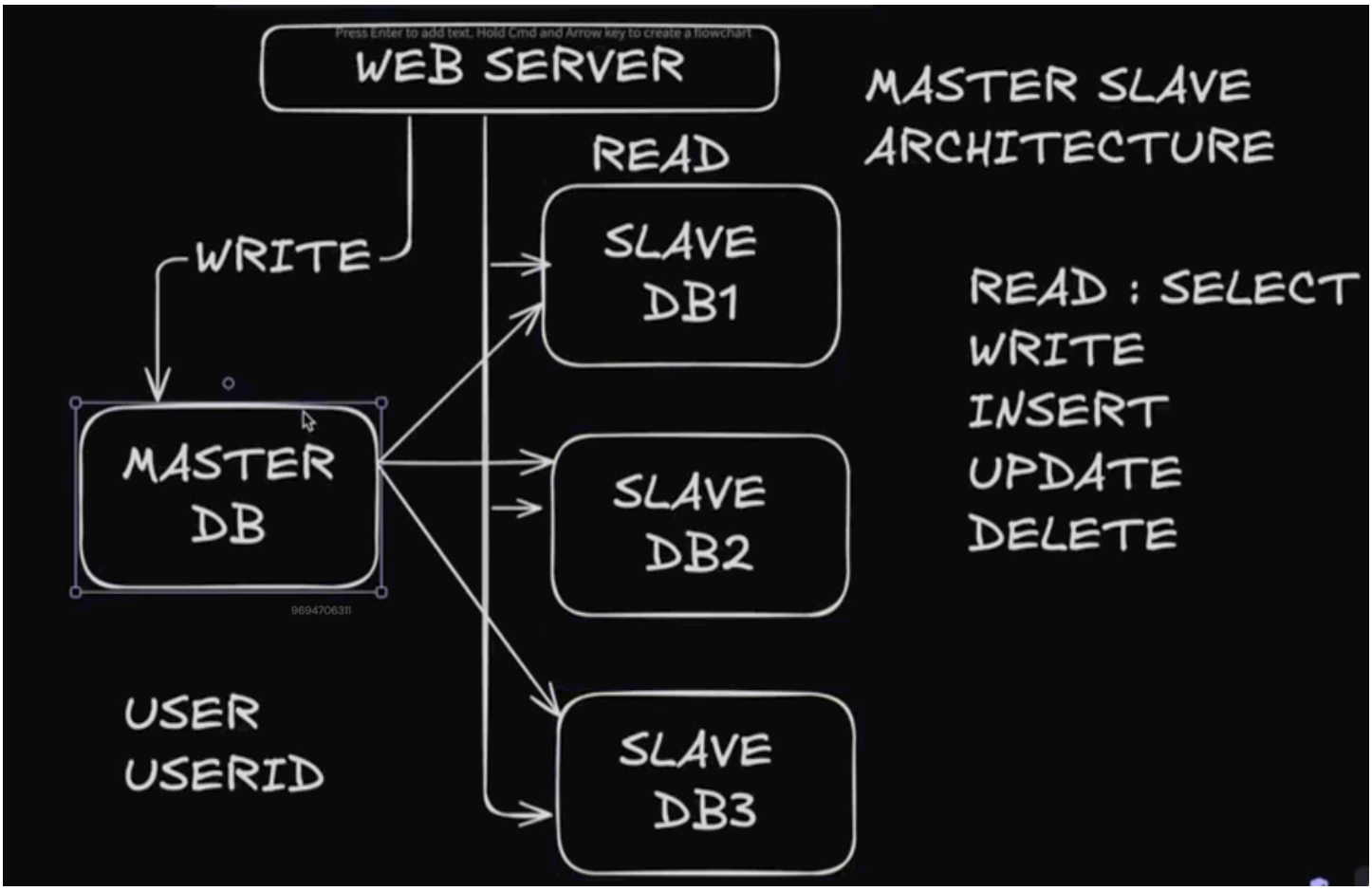
13.When should you add Load Balancer?

Add LB when:

- Traffic > single server capacity
- Need high availability
- Want zero downtime deploys

Load Balancer = Traffic Manager + Fault Tolerance + Scalability

Phase=3



1.What is Master–Slave Database Architecture?

It is a **database design** where:

- **Master Database**
  → Handles all **WRITE operations**
  (INSERT, UPDATE, DELETE)
- **Slave Databases**
  → Handle **READ operations**
  (SELECT / fetch data)

Writes go to Master

Reads go to Slaves

Why do we use Master–Slave?

Because a **single database cannot handle very large traffic**.

Most applications have:

- **Very few writes**
- **Too many reads**

2.Problems it solves

1.Too many read requests

Users mostly read data:

- **View profile**
- **View posts**
- **View products**

One database becomes slow
Slaves handle read traffic

2.Better performance

Reads and writes are separated.

Database works faster and smoother

3.Scalability

More users = more read requests

Add more slave databases
No need to change master

4.High availability

If master database has issues:

- **Slaves still have data**
- **System does not fully crash**

5.Heavy work handled by slaves

- **Backups**
- **Reports**
- **Analytics queries**

Master stays free for important write work

3.How does data stay same?

- **Data is written to master**
- **Slaves copy data from master**
- **Small delay can happen (normal)**

Simple real-life example

Teacher and students example:

- Teacher writes notes (Master)
- Students read notes (Slaves)
- Many students can read at the same time
- Only teacher can change notes

4.What happen when Master Database goes down

Problem:

- Master handles all WRITE operations
- If Master goes down → writes stop
- Reads may still work from Slaves

How do we resolve it?

1.Failover

- One Slave is promoted to Master
- Application starts sending writes to new Master

2.Automatic Monitoring

- System continuously checks Master health
- On failure, failover happens automatically

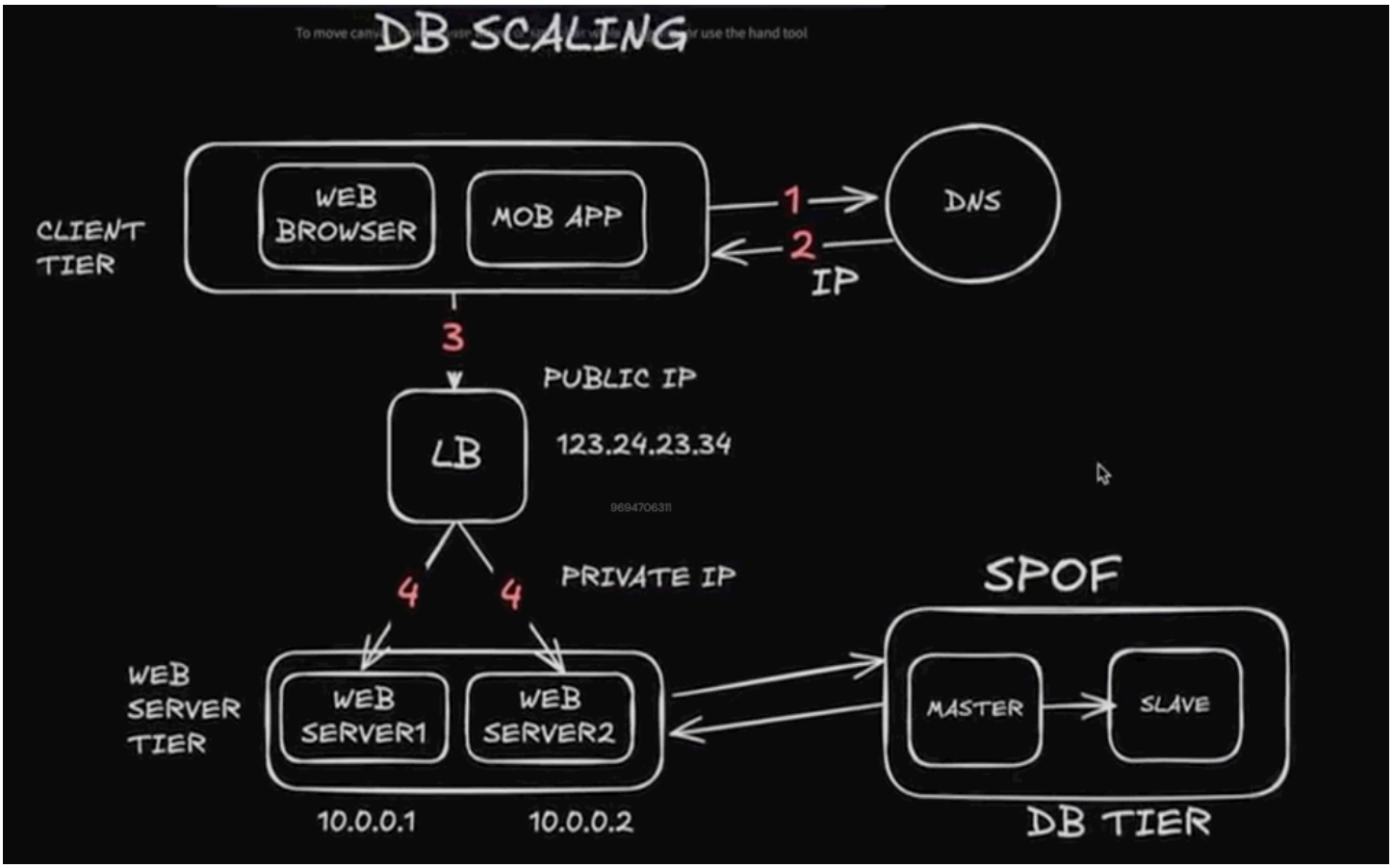
3.Reconfiguration

- App updates DB connection to new Master
- Old Master is fixed and added back as Slave

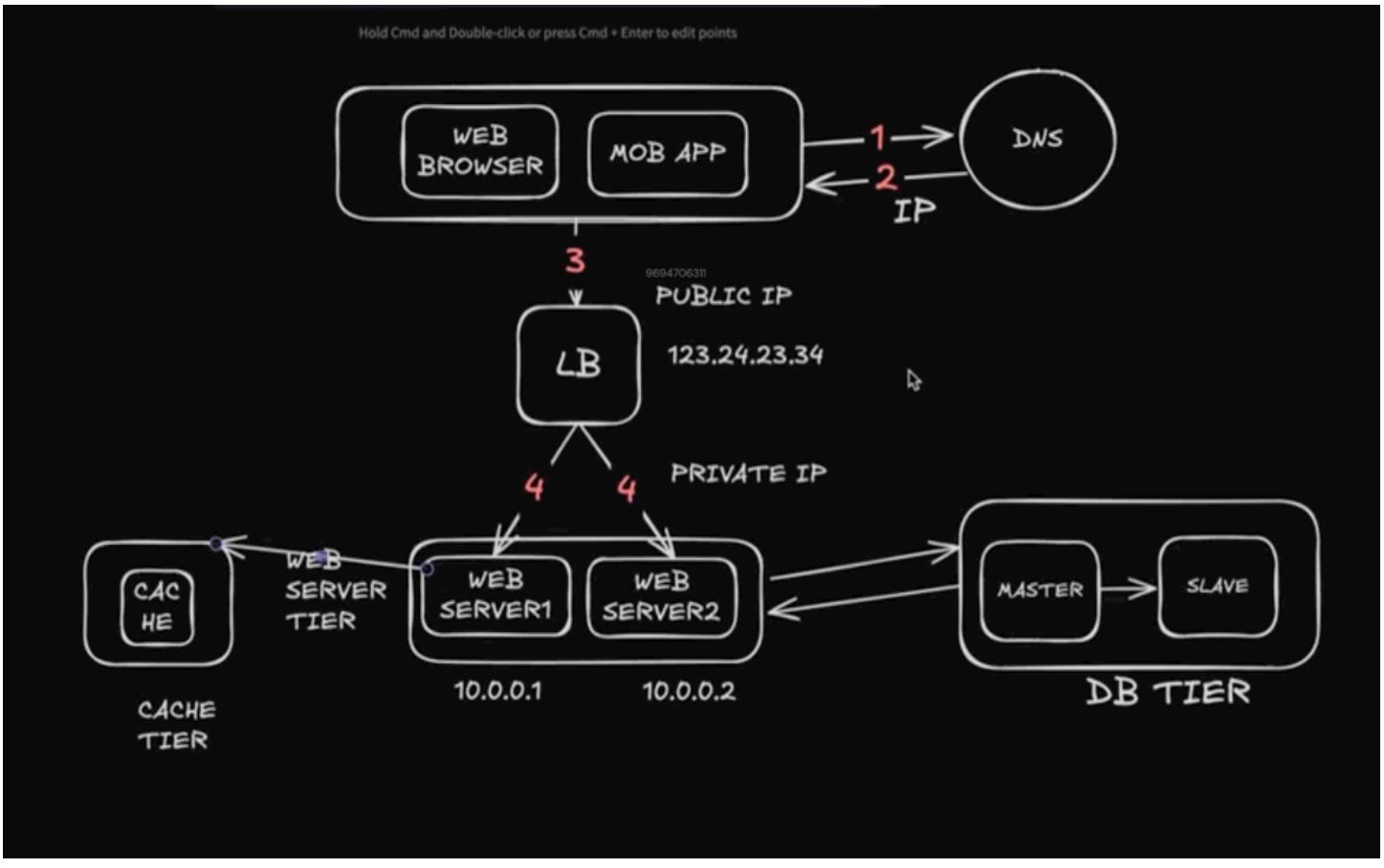
4.Data Safety

- Replication logs are used
- Slight data loss possible in async setup

When Master goes down, writes stop; the issue is resolved using failover by promoting a slave to master and redirecting write traffic.



Phase4



1.What is Cache?

Cache is a temporary fast storage that stores frequently used data so it can be returned very quickly.

Instead of going to:

Client → Server → Database (slow)

Cache allows:

Client → Server → Cache (fast)

2.Why do we use Cache?

Because database access is slow and expensive, especially when:

- **Millions of users**
- **Same data requested again and again**

Cache reduces database load and response time.

3.What problems does Cache solve?

1.Slow response time

Problem:
Database queries take time (disk I/O, locks, joins).

Solution:
Cache stores data in memory (RAM).

Memory is much faster than disk.

2.Too many repeated requests

Problem:
Same data is requested again and again:

- **User profile**
- **Product list**
- **Home page data**

Database runs same query repeatedly

Solution:
Cache stores result once and reuses it.

4.Database overload

Problem:
Too many read queries overload database.

Solution:
Cache handles most read requests.

Database stays free for writes.

5.Scalability issue

Problem:
As users increase, DB cannot handle load.

Solution:
Cache absorbs traffic.

System scales easily.

6.Cost problem

Problem:
High DB usage = expensive servers.

Solution:
Cache reduces DB usage.

Lower infrastructure cost.

4.How Cache works

1. **Client requests data**
2. **Server checks Cache**
3. **If data found → Cache Hit**
4. **If data not found → Cache Miss**
5. **On miss:**
   - **Fetch from DB**
   - **Store in Cache**
   - **Return to client**

Cache Hit vs Cache Miss

Term	Meaning
Cache Hit	Data found in cache
Cache Miss	Data not found in cache

5.Where Cache is placed?

- Between Application Server & Database
- Sometimes at CDN level

6.What data is cached?

- Read-heavy data
- Less frequently changing data
- Public data

Not suitable for frequently changing data

Cache expiry (TTL)

- Cached data is not stored forever
- TTL (Time To Live) is set
- After TTL, data expires

Common Cache Systems

- Redis
- Memcached
- CDN cache

Real-life example

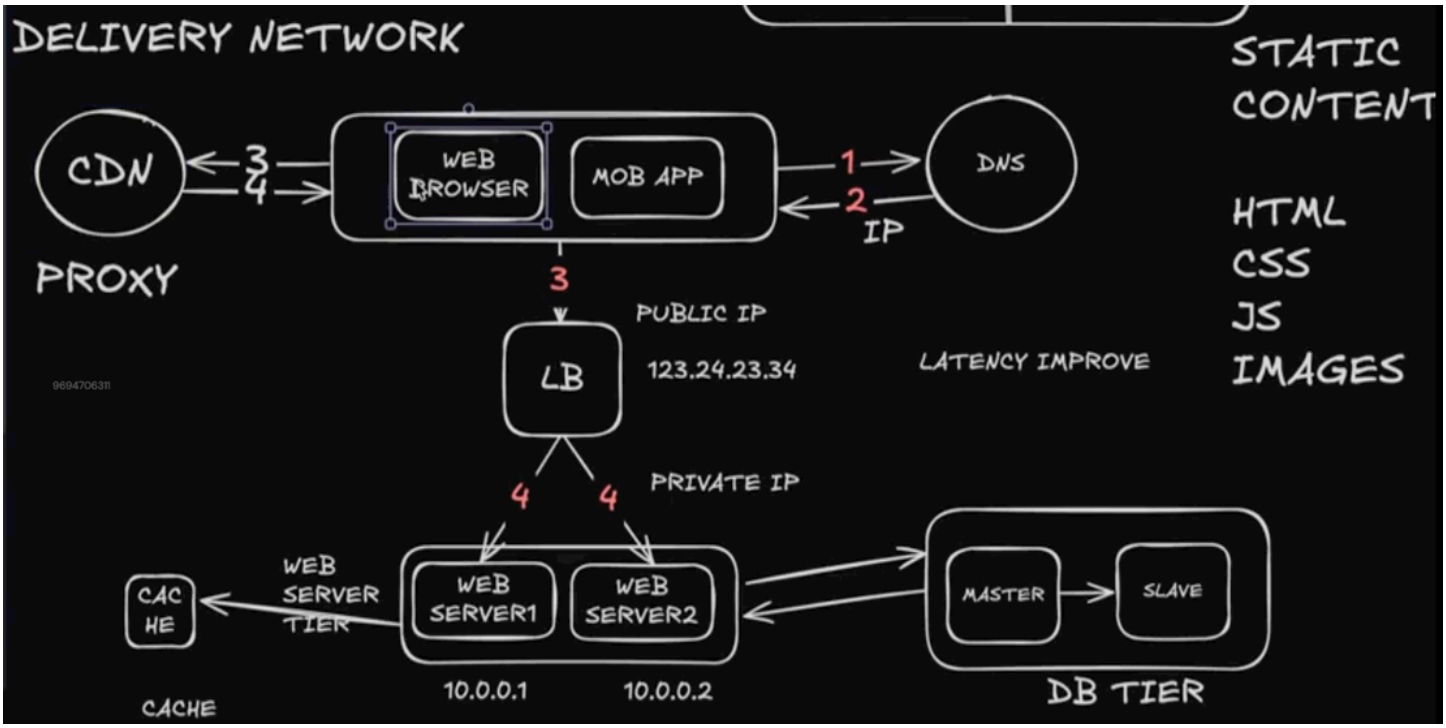
Think of notes on study table:

- Book = Database
- Notes = Cache

You check notes first because they are faster.

Cache is a fast in-memory storage used to reduce database load and improve response time by storing frequently accessed data.

Phase = 5



1.What is CDN?

CDN (Content Delivery Network) is a network of servers distributed across different locations that deliver content from the nearest server to the user.

It mainly serves:

- **Images**
- **Videos**
- **CSS, JS files**
- **Static content**

2.Why do we use CDN?

Because serving content from one central server is slow for users far away.

3.What problems does CDN solve?

1.High latency (slow loading)

Problem:
User is far from main server
Data has to travel long distance

Solution:
CDN serves data from nearest location

Faster page load

2.Heavy load on main server

Problem:
All users hit main server for images/videos

Solution:
CDN handles static content

Main server handles business logic only

3.Scalability problem

Problem:
Sudden traffic spike (viral content)

Solution:
CDN absorbs traffic

Website does not crash

4.Bandwidth cost

Problem:
Serving large files uses high bandwidth

Solution:
CDN optimizes delivery

Lower bandwidth cost

5.Better availability

Problem:
If one server/location fails

Solution:
CDN serves from another nearby server

High availability

6.Security benefits

Problem:
DDoS attacks, bots

Solution:
CDN provides:

- **DDoS protection**
- **Rate limiting**
- **Web Application Firewall (WAF)**

4.How CDN works (Simple flow)

1.User requests a file (image/video)

2.Request goes to nearest CDN server

3.If file exists → served immediately

4.If not → CDN fetches from origin server and caches it

What content is best for CDN?

Only for Static content

Real-life example

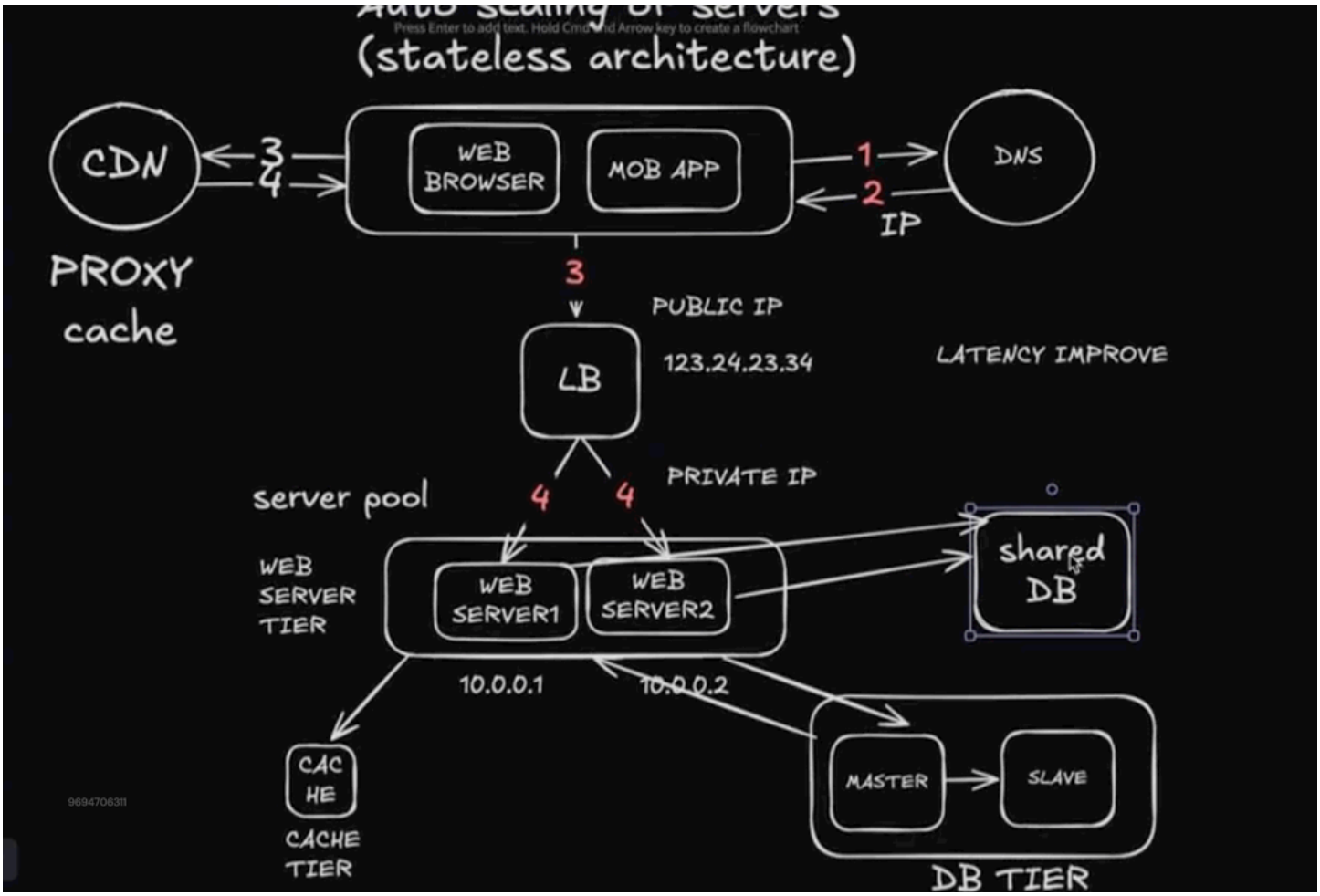
Think of movie theatres:

- One movie in one city = slow for others
- Same movie in many cities = fast access

CDN is used to deliver content faster by serving it from the nearest server, reducing latency, server load, and improving scalability.

Phase 5

Auto Scaling



Why Stateless & Shared DB are required

What is Auto Scaling?

Auto scaling means:

- **Automatically add servers when traffic increases**
- **Automatically remove servers when traffic decreases**

Why Auto Scaling fails in Stateful architecture?

Problem 1: Session stored in server memory

In stateful systems:

- **User session is stored inside one server's RAM**

User → Server A (session stored here)

If auto scaling adds Server B:

User → Server B (session not found)

User gets logged out / error

Problem 2: Load balancer sends user to any server

Auto scaling means:

- **Requests can go to any server**
- **No guarantee user hits same server**

Stateful apps depend on same server

Problem 3: Server removal breaks sessions

When traffic drops:

- **Auto scaling removes servers**
- **Session data stored in that server is lost**

Active users get logged out

1.Why Stateless / Sessionless architecture fixes this?

Sessions stored centrally

Sessions stored in:

- **Redis**
- **Database**
- **JWT token**

User → Any Server → Shared Session Store

Any server can handle request

Servers become replaceable

- **No user data inside server**
- **Server can be added or removed safely**

Auto scaling works perfectly

2.Why Shared Database is required?

Separate DB per server (bad)

- **Data inconsistency**
- **Sync issues**
- **Scaling impossible**

Shared DB (good)

- **All servers read/write same data**

- Consistent state

Multiple App Servers → One DB layer

3.Final Architecture for Auto Scaling

Load Balancer

|

Multiple Stateless App Servers (Auto Scaled)

|

Shared Cache / Session Store

|

Shared Database

One-line

Auto scaling is not possible in stateful systems because sessions are tied to servers; stateless architecture with shared DB and external session storage makes auto scaling possible.

Instagram real-world architecture

Instagram has millions of users online at the same time, so it must use auto-scaling.

1.User opens Instagram app

User → Load Balancer

- Load balancer sends request to any available server
- No fixed server for any user

2.Stateless App Servers (Auto Scaled)

Instagram servers are stateless:

- No session stored in server memory
- Any server can handle any user request

User request → Server A

Next request → Server C

User still logged in

3.How login/session works?

Instagram uses:

- **JWT / access tokens**
- **Or session stored in Redis**

So:

Server does NOT remember user

Token/Redis does

Servers can be added/removed freely

4.Auto Scaling in action (Real example)

During peak time (evening / reels viral):

- **Traffic increases**
- **Auto scaling adds more app servers**

During night:

- **Traffic decreases**
- **Extra servers are removed**

Users see no issue

5.Shared Database layer

- **All servers connect to same DB system**
- **Uses Master–Replica**
  - **Writes → Master**
  - **Reads → Replicas**

Data consistent for all users

6.Cache (Redis) in Instagram

Cached data:

- **User profile**
- **Feed metadata**
- **Like counts**

Server → Redis → DB (only if needed)

Faster feed loading

7.CDN usage (Very important)

Instagram uses CDN for:

- **Images**
- **Videos**
- **Reels**

User → Nearest CDN server → Image/Video

Fast loading worldwide

8.What if one server goes down?

- **Load balancer stops sending traffic to it**
- **Other servers handle users**
- **Auto scaling replaces it**

User never notices

9.Why Instagram CANNOT be stateful?

If Instagram stored session in server:

- **User would logout on every request**
- **Auto scaling would break login**

So stateless architecture is mandatory

Instagram uses stateless auto-scaling app servers with shared cache, master-replica databases, and CDN so millions of users can be served reliably and fast.

1.What are Cookies?

Cookies are small pieces of data stored in the user's browser by a website.

They help the website remember the user.

Why do we use Cookies?

1.Remember login

- **Keeps user logged in**
- **No need to login again and again**

2.Store user preferences

- **Language**

- Theme (dark/light)
- Settings

3.Session management

- Identify which requests belong to which user
- Used with sessions / tokens

4.Tracking & analytics

- Count visitors
- Understand user behavior

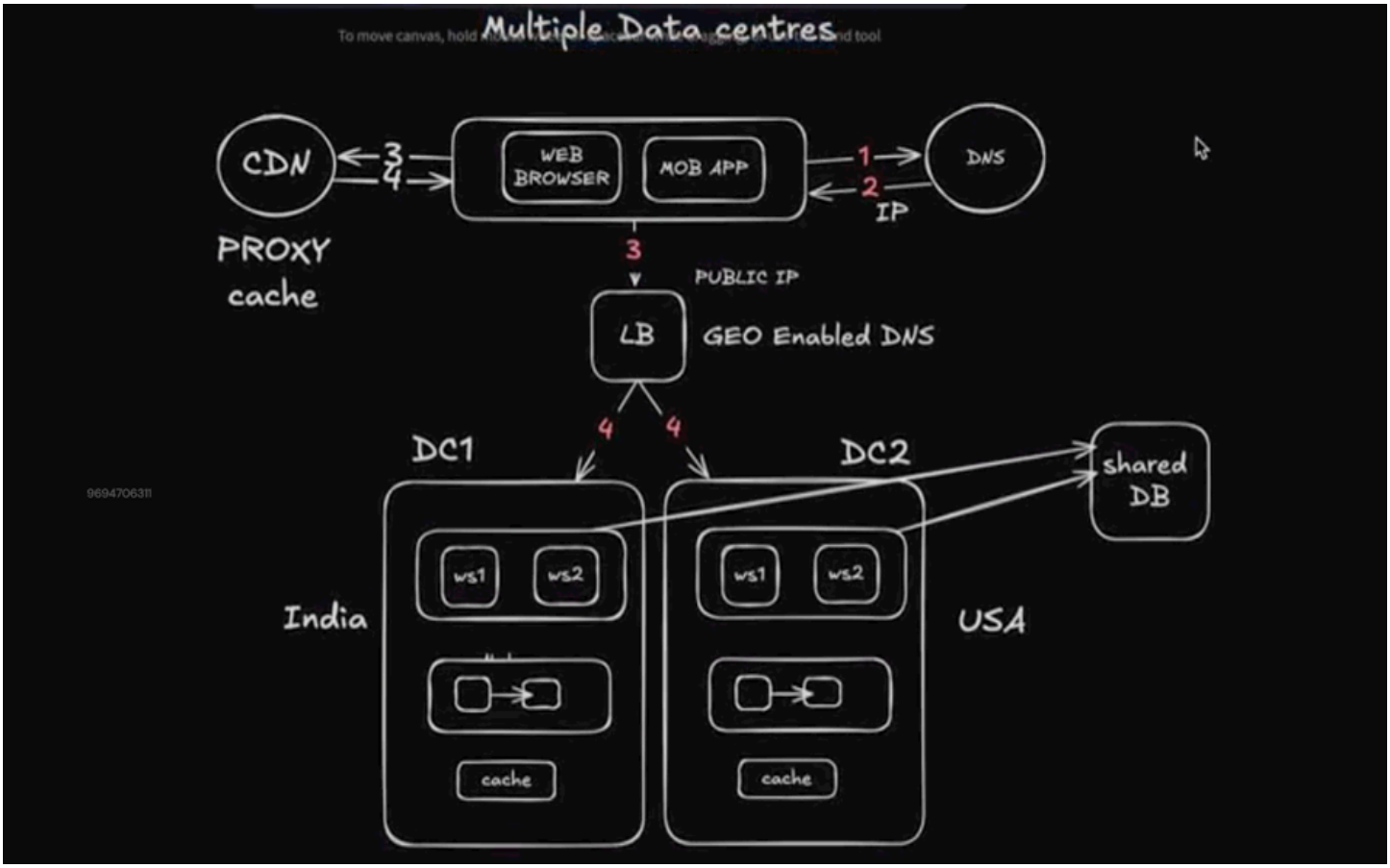
Simple example

When you login to a website:

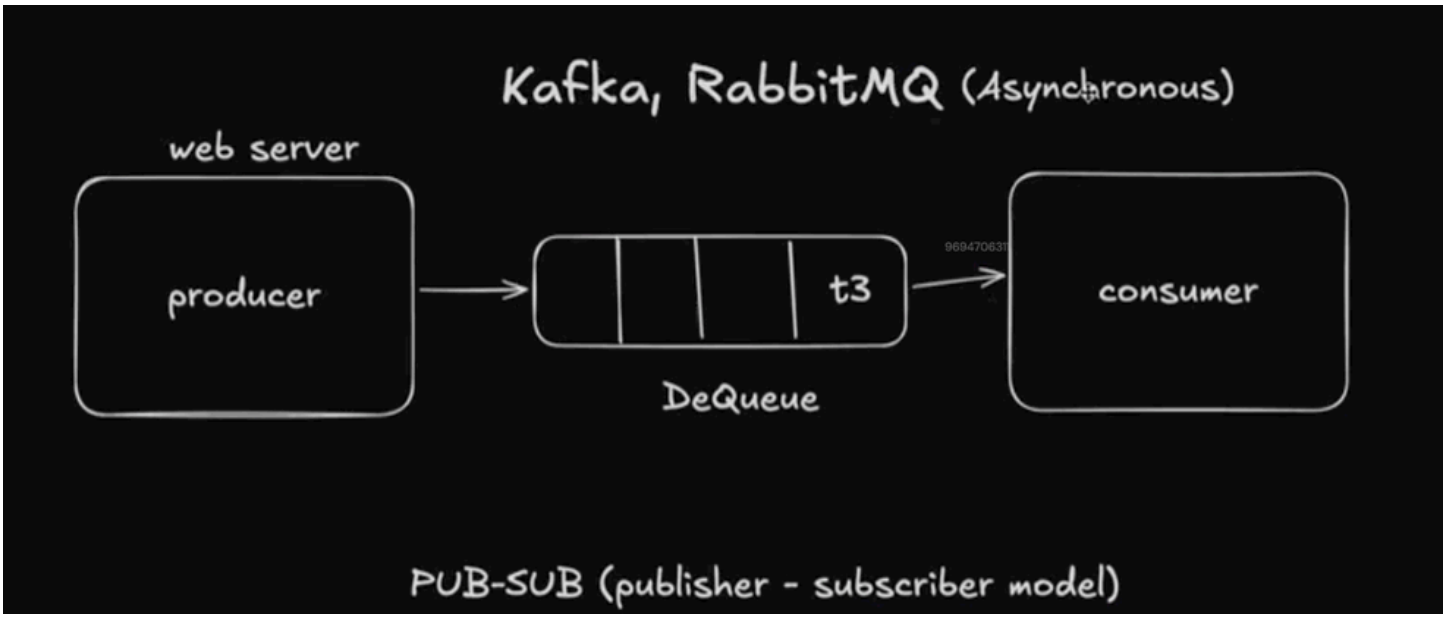
- Server stores a cookie in your browser
- Browser sends that cookie with every request
- Server knows you are the same user

Cookies are small data stored in the browser to remember user information like login and preferences.

How can we handle international client ?



Phase6



1.Pub-Sub (Publish–Subscribe) Model —Messaging Queue

The **Pub-Sub model** is a messaging pattern where **senders (publishers)** send messages **without knowing who will receive them**, and **receivers (subscribers)** get messages **without knowing who sent them**.

They communicate through a **message broker / topic**.

Core Components

1. **Publisher**
   - Sends (publishes) messages
   - Does **not care who consumes** the message
2. **Topic / Channel**
   - Logical stream where messages are published
   - Example: user-signup, order-created
3. **Subscriber**
   - Listens to a topic
   - Automatically receives messages
4. **Message Broker**
   - Manages topics and delivery
   - Examples: **Kafka, RabbitMQ, Redis Pub/Sub, Google Pub/Sub**

How It Works (Step by Step)

1. Publisher sends a message to a **topic**
2. Broker receives the message
3. Broker pushes the message to **all subscribers** of that topic
4. Each subscriber processes the message **independently**

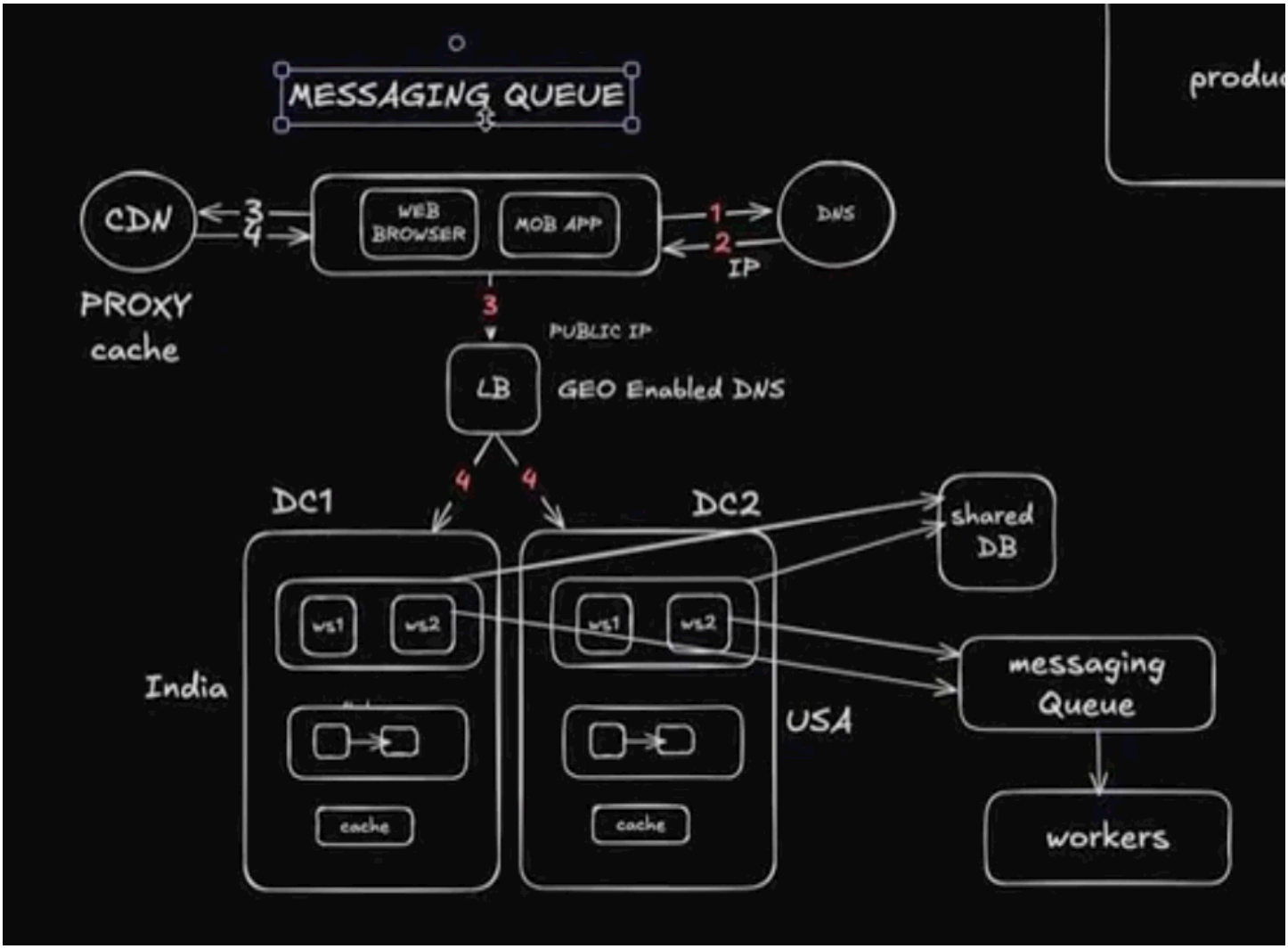
Simple Example (Real Life)

YouTube Channel

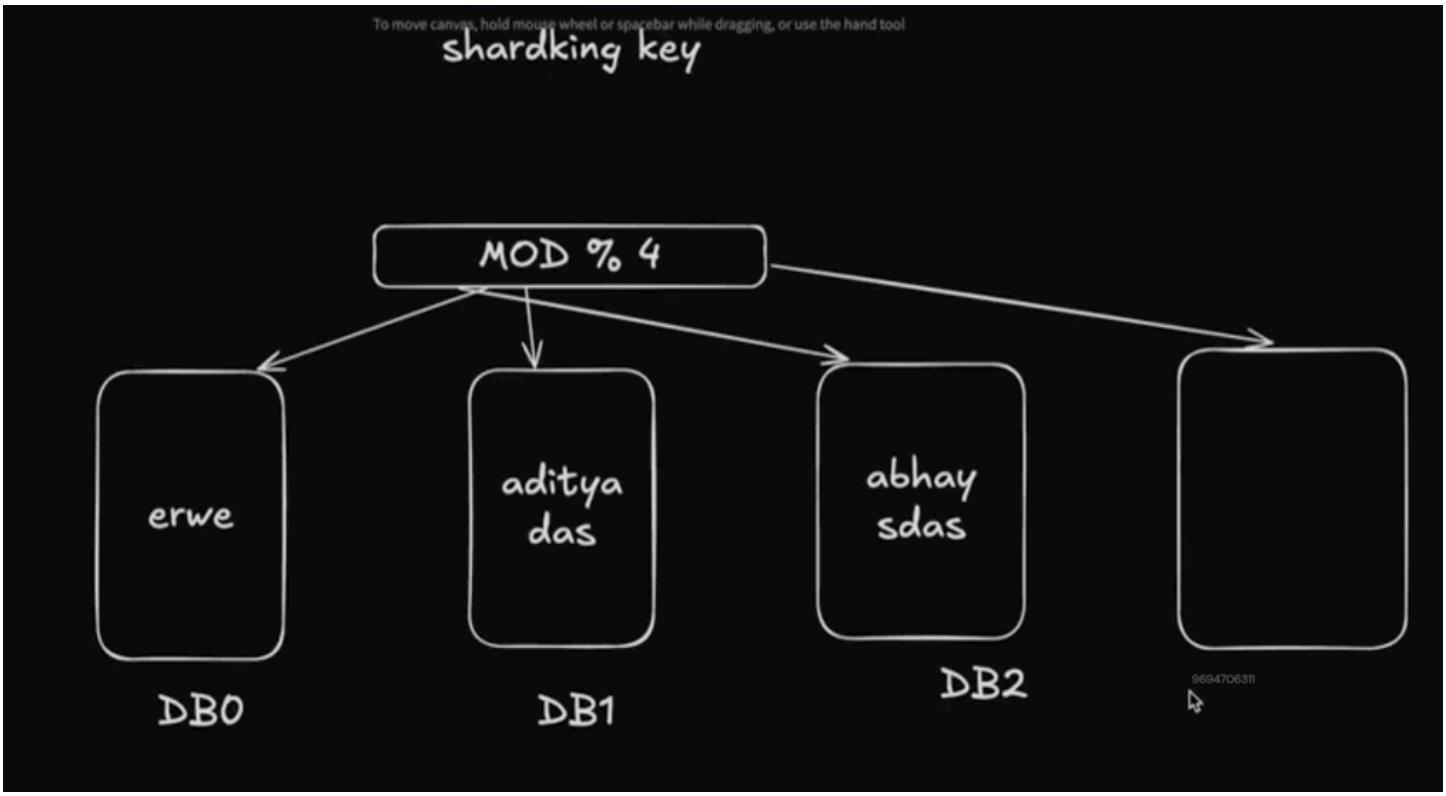
- You = **Publisher**
- Channel = **Topic**
- Subscribers = **Subscribers**
- New video uploaded → **all subscribers get notification**

You don’t send notifications individually — YouTube handles it.

Pub-Sub model allows services to communicate asynchronously by publishing events to topics that multiple subscribers can consume independently.



Phase 7



1.Sharding (DB Partitioning) using Modulus

When your application reaches **tens / hundreds of millions of users**, **one database is not enough**.
So in **Phase 7**, we **split data across multiple databases** → this is called **Sharding**.

2.What is Sharding?

Sharding = Horizontal Partitioning of data

- Same table structure
- Data divided across **multiple DB servers (shards)**
- Each shard stores **only a subset** of data

Goal: **Scale database writes + reads**

3.Why Phase 7?

Earlier phases:

- Phase 5 → Stateless servers
- Phase 6 → Pub-Sub (async processing)

Now bottleneck = **DATABASE**

- Too many writes
- Too many reads
- Big tables
- Single DB becomes slow & expensive

So we shard.

4.Modulus-Based Sharding (Most Common)

Formula

shard_id = shard_key % number_of_shards

Shard Key

A **shard key** is the value used to decide **where data will go**.

Good shard keys:

- user_id
- order_id
- customer_id

Bad shard keys:

- country
- gender
- status

(These cause uneven data)

Example (Very Important)

Suppose:

- 4 DB shards → DB0, DB1, DB2, DB3
- Shard key = user_id

Formula:

user_id % 4

user_id	Calculation	Shard
1	1 % 4 = 1	DB1
2	2 % 4 = 2	DB2
3	3 % 4 = 3	DB3
4	4 % 4 = 0	DB0
5	5 % 4 = 1	DB1
6	6 % 4 = 2	DB2

Data is **evenly distributed**

5.Read Flow (Step-by-Step)

1. **Client requests user data → user_id = 25**
2. **App calculates:**

25 % 4 = 1

1. **App directly queries DB1**
2. **No scanning other DBs**

Very fast lookup

Write Flow

1. **New user signs up → user_id = 42**
2. **Calculate:**

42 % 4 = 2

1. **Insert data into DB2**

6.Why Modulus Sharding is Fast?

- **O(1) shard lookup**
- **No metadata lookup**
- **No central routing DB**
- **App decides shard itself**

Real-World Example (Instagram)

- **Shard key → user_id**
- **All user-related data:**
  - **posts**
  - **likes**
  - **comments**
  - **profile**
- **Stored in same shard**

Avoids cross-shard joins

7.Big Problem with Modulus Sharding

Resharding Problem

Suppose:

- **Initially → 4 shards**
- **Later → need 5 shards**

Old formula:

user_id % 4

New formula:

user_id % 5

Almost ALL data moves to new shards

Example:

10 % 4 = 2

10 % 5 = 0

Massive data migration
Downtime risk
Very expensive

8.How Companies Solve This?

1.Consistent Hashing (Phase 7.5)

- **Minimal data movement**
- **Used by DynamoDB, Cassandra**

2.Virtual Shards

- Many logical shards
- Mapped to physical DBs

3.Pre-Sharding

- Start with large shard count (e.g., 1024)
- Add DBs later

9.When to Use Modulus Sharding?

1.High write systems
2.User-based queries
3.Simple access patterns
4.No frequent shard count change

1.Heavy range queries
2.Rapid shard expansion needed

Modulus sharding distributes data evenly using shard_key % N, giving fast reads/writes but making resharding very difficult.