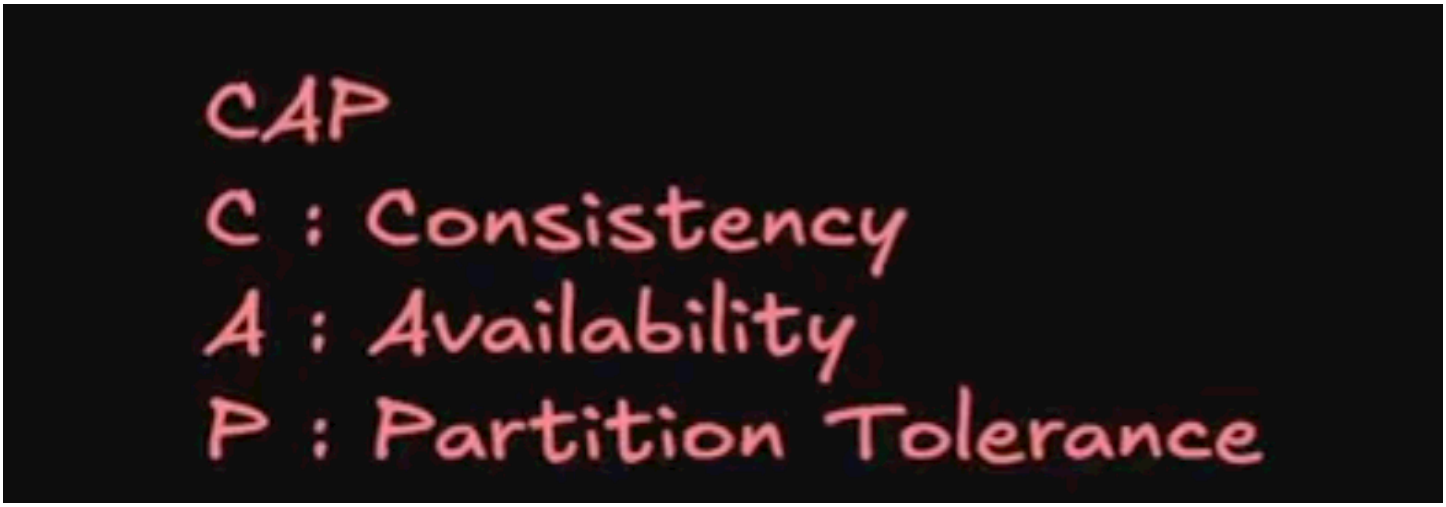
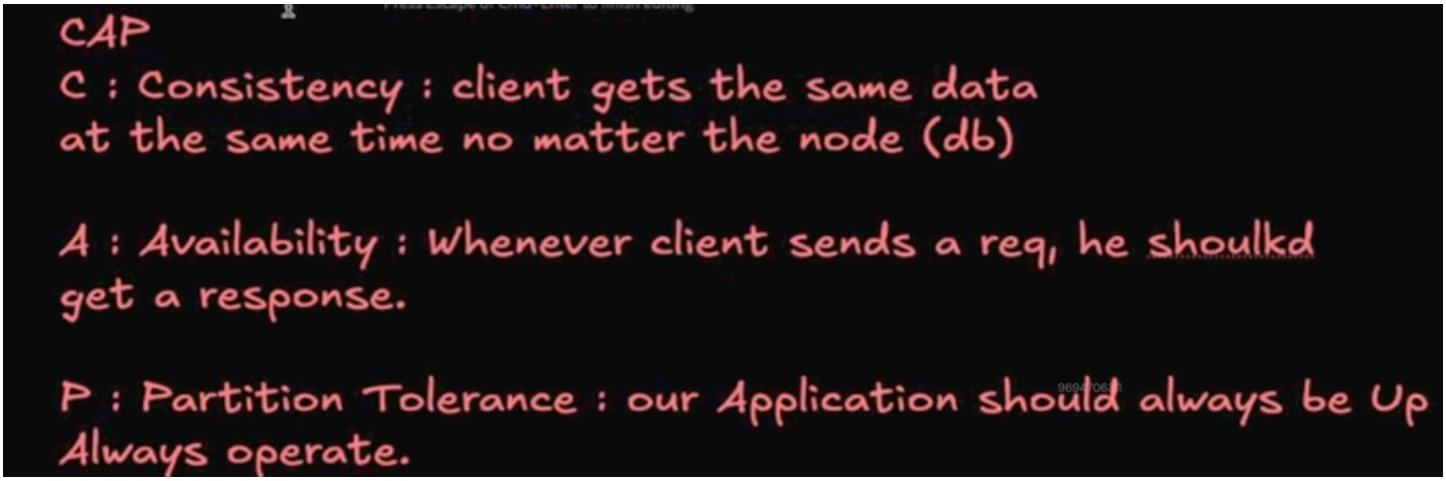
CAP Theornm and Back to the envelop Calculation, Monolith Vs MicroServices





1.CAP Theorem

CAP Theorem says that in a distributed system (system running on multiple servers), you cannot guarantee all three of these at the same time:

1. **Consistency (C)**
   → All users see the same latest data at the same time.
2. **Availability (A)**
   → Every request gets a response, even if some server fails.
3. **Partition Tolerance (P)**
   → System continues to work even if there is a network failure between servers.

Rule:
You can choose only two out of three: C, A, P

1.Why Partition Tolerance is Mandatory

In real-world systems, network failures WILL happen, so Partition Tolerance (P) is always required.

Second reason is our application should not be down.

So the real choice is:

- **CP (Consistency + Partition Tolerance)**
- **AP (Availability + Partition Tolerance)**

CAP with Instagram Example

Instagram chooses AP system

Availability (A)

- **Instagram never stops working**
- **You can open app, scroll feed, like posts even if some servers are down**

Partition Tolerance (P)

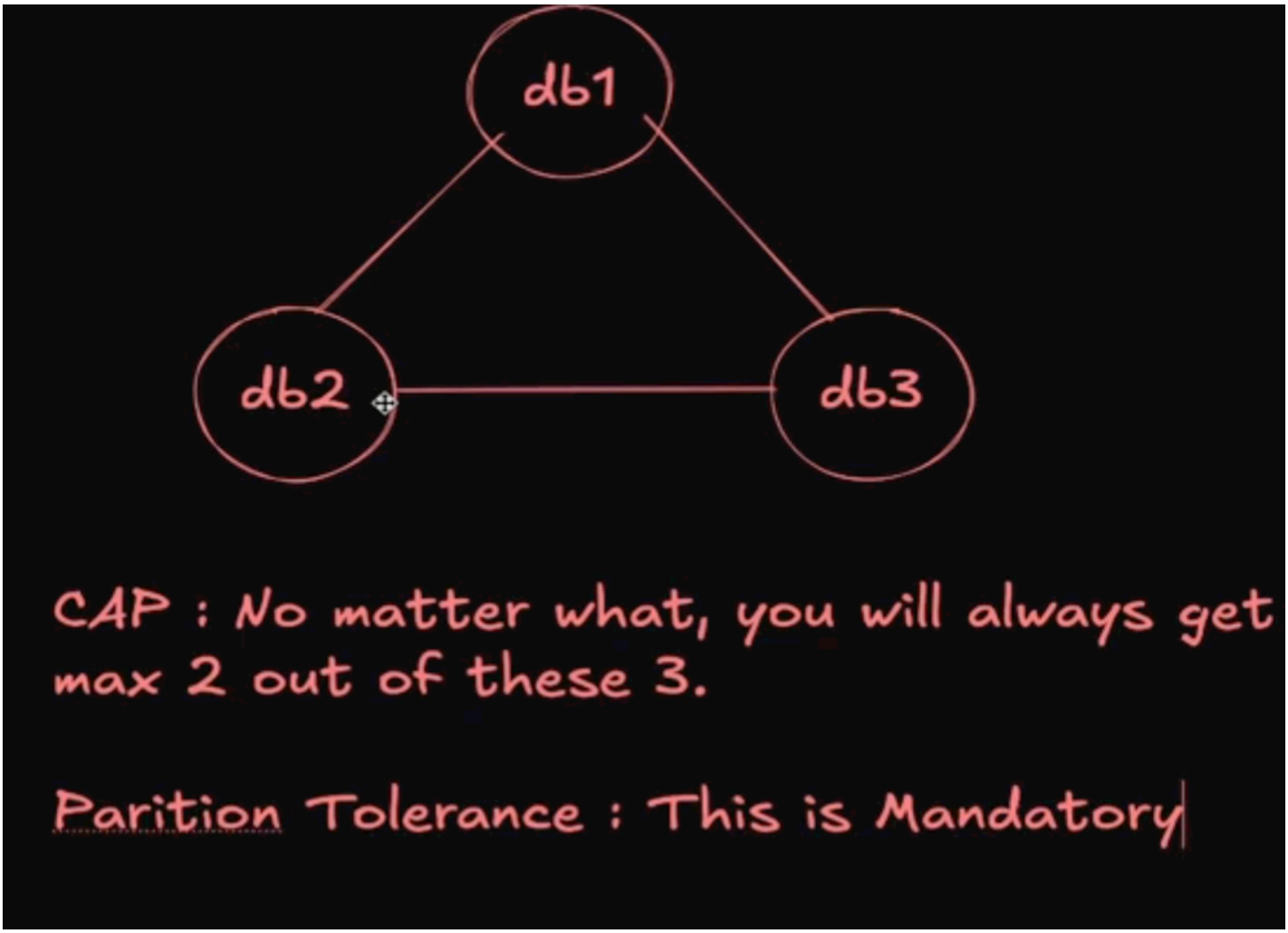
- **Instagram has servers in India, US, Europe**
- **If network issue happens between servers, app still works**

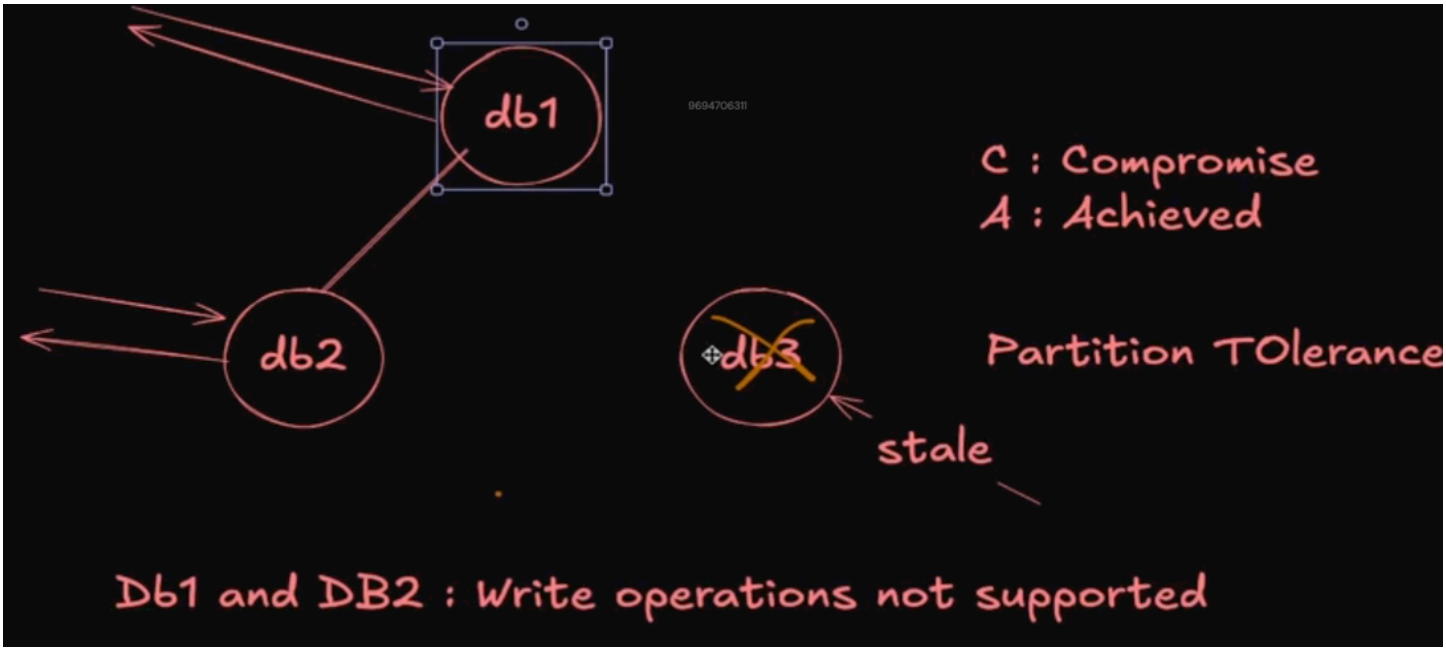
Consistency (C) is relaxed

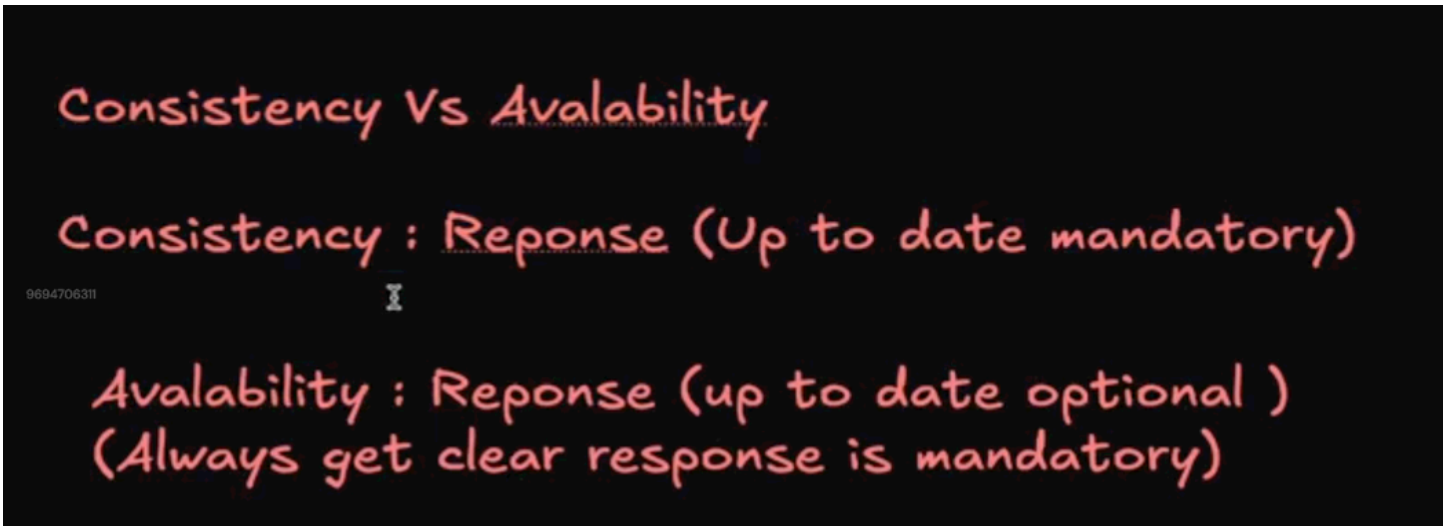
- **You like a post, but your friend doesn't see it immediately**
- **Follower count may be different for few seconds**
- **Comments appear after some delay**

This is called Eventual Consistency

On Instagram, when you like a post and the like count updates after some time, it proves Availability + Partition Tolerance, but not immediate Consistency.

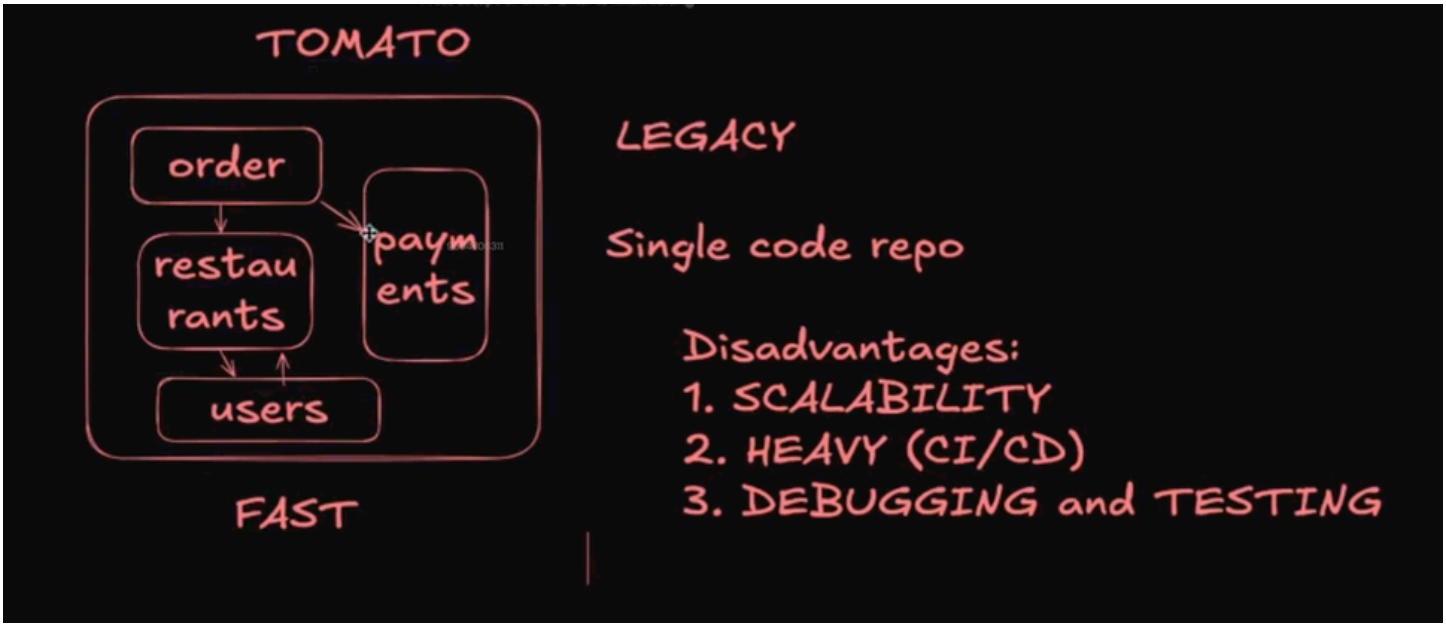

db1
db2
db3
CAP : No matter what, you will always get max 2 out of these 3.
Parition Tolerance : This is Mandatory


db1
9694706311
C : Compromise
A : Achieved
db2
db3
Partition TOlerance
stale
Db1 and DB2 : Write operations not supported



Monolithic Architecture

1.What is Monolith?



Monolith means the whole application is one single system.

Restaurant Example (Monolith)

Imagine a small restaurant:

- **One kitchen**
- **One chef**
- **One billing counter**

All work happens together in one place:

- **Take order**
- **Cook food**
- **Take payment**
- **Serve food**

If one thing fails, the whole restaurant stops.

Problems

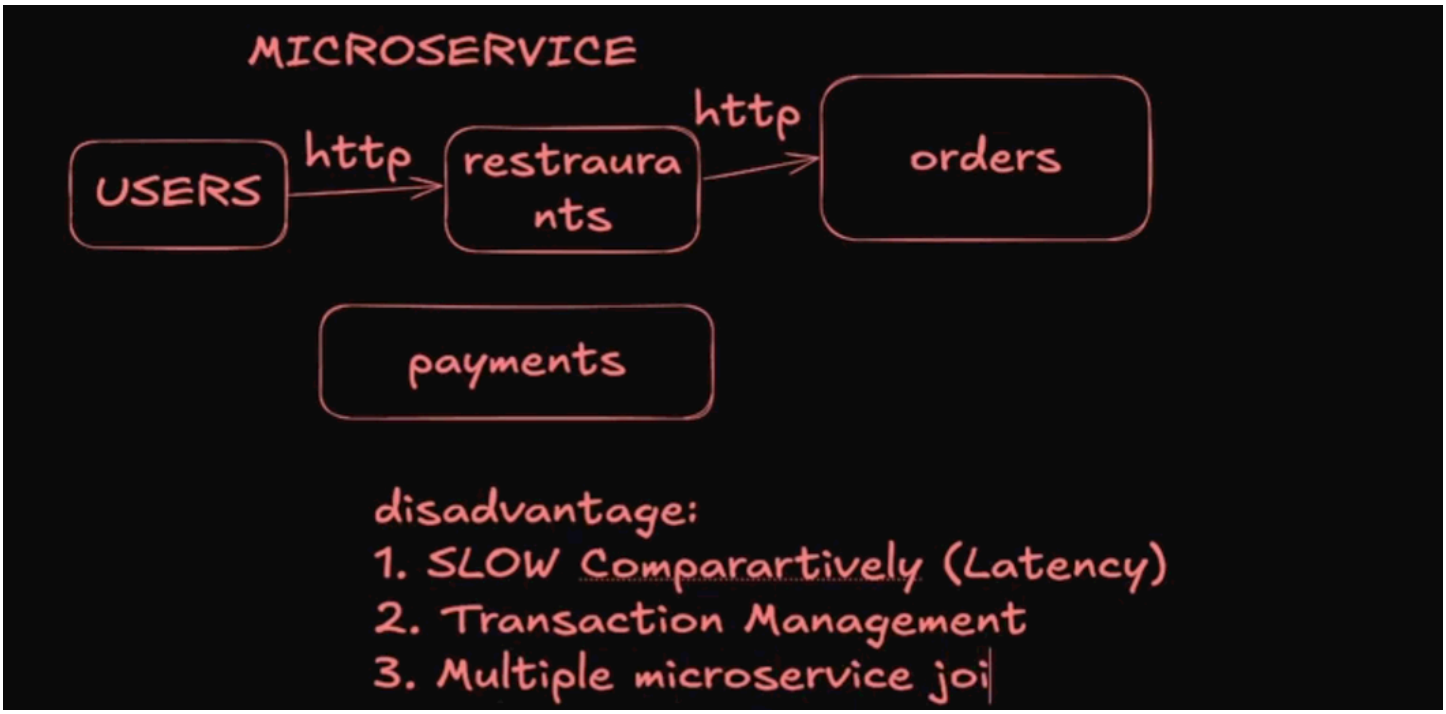
- **If billing is slow → orders also slow**
- **If kitchen crashes → restaurant closed**
- **To change menu → close whole restaurant**

When to use?

Small apps
Less users

Microservices Architecture

1.What are Microservices?



Microservices means the app is divided into **small independent services**.

Restaurant Example (Microservices)

Imagine a **food court**:

- Pizza kitchen
- Burger kitchen
- Drinks counter
- Separate billing

Each part works **independently**.

If pizza kitchen is down, **burger counter still works**.

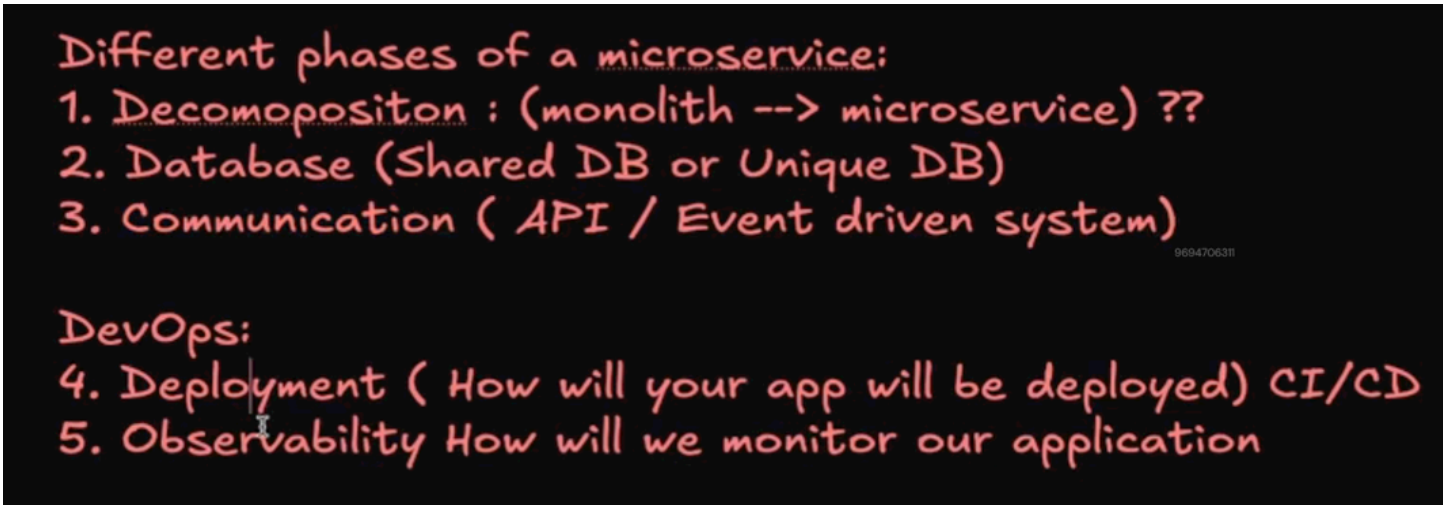
Advantages

- Easy to scale
- One service can be fixed without stopping others
- Better for large apps

Problems

- More complex
- Needs good server and monitoring

Different phases of a Microservices



1.Decomposition (Breaking the Application)

What it means

Decomposition means breaking one big application into small services.

Simple example

Big restaurant → split into:

- **Order Service**
- **Menu Service**
- **Payment Service**
- **Delivery Service**

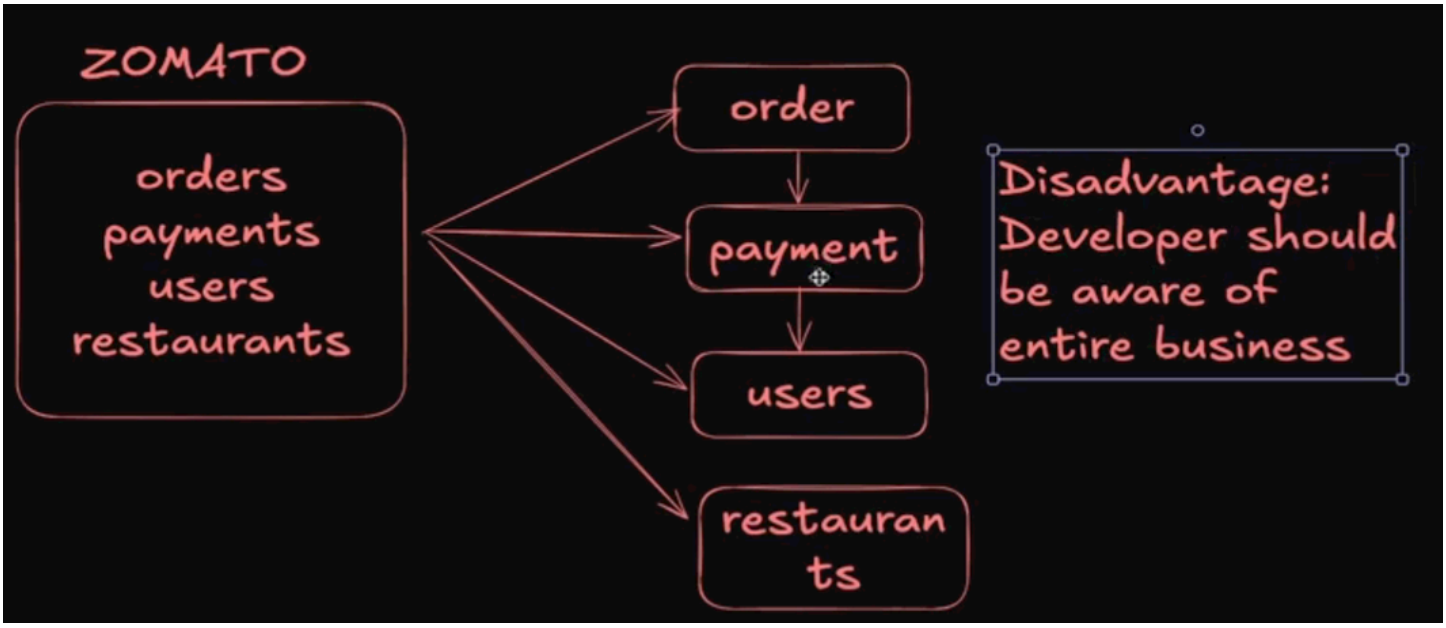
Key point

Each service does one job only.

Rule: One service = one responsibility

2.There are 2 main ways

1.Decomposition by Business Logic



What it means

Split services based on what the system does (features).

Simple rule

One business function = one service

Example (Restaurant)

- **Order Service**
- **Payment Service**
- **Menu Service**
- **Delivery Service**

Each service handles one business work.

Real app example (Instagram)

- **User Service**
- **Post Service**
- **Like Service**
- **Comment Service**
- **Notification Service**

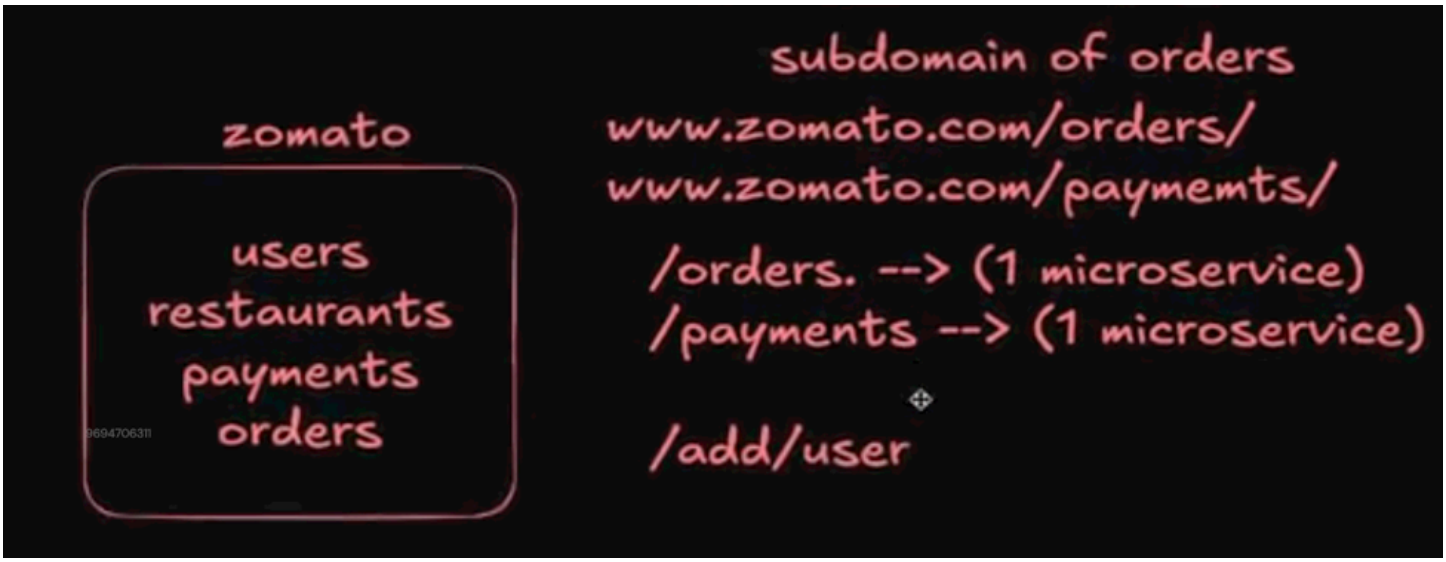
Pros

- **Easy to understand**
- **Matches real business needs**

Cons

- If logic is mixed, boundaries can be wrong

2.Decomposition by Sub-Domains



What it means

Split services based on business domains (DDD concept).

Large business → small sub-domains.

Example (Restaurant Business)

Main Domain: Food Ordering

Sub-domains:

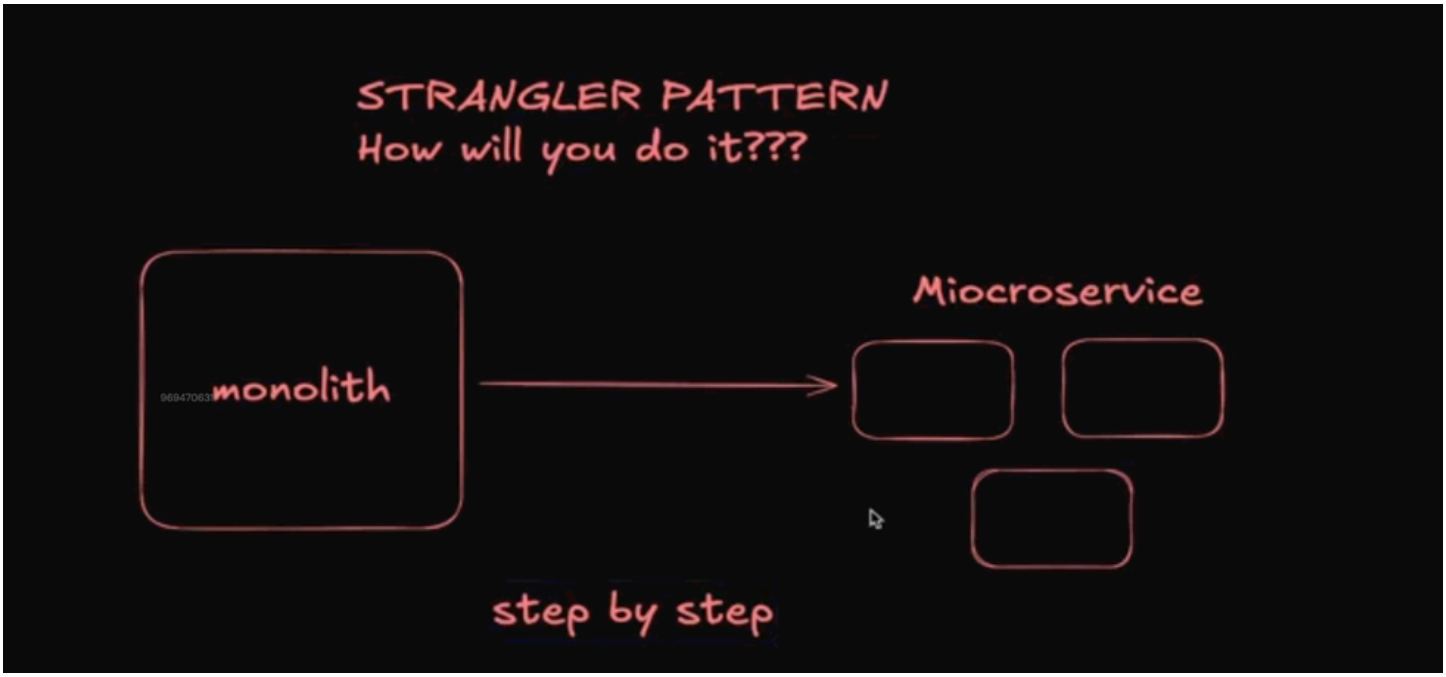
- Customer Domain
- Ordering Domain
- Payment Domain
- Delivery Domain

Each sub-domain can have multiple services inside it.

Real app example (Zomato / Swiggy)

- Customer Domain
  - Login Service
  - Profile Service
- Order Domain
  - Order Service
  - Cart Service
- Payment Domain
  - Payment Service
  - Refund Service

Strangler Pattern



1.What is Strangler Pattern?

Strangler Pattern is a way to slowly convert a Monolithic application into Microservices, without shutting the system down.

We do not rewrite everything at once.
We replace parts step by step.

2.Why the name “Strangler”?

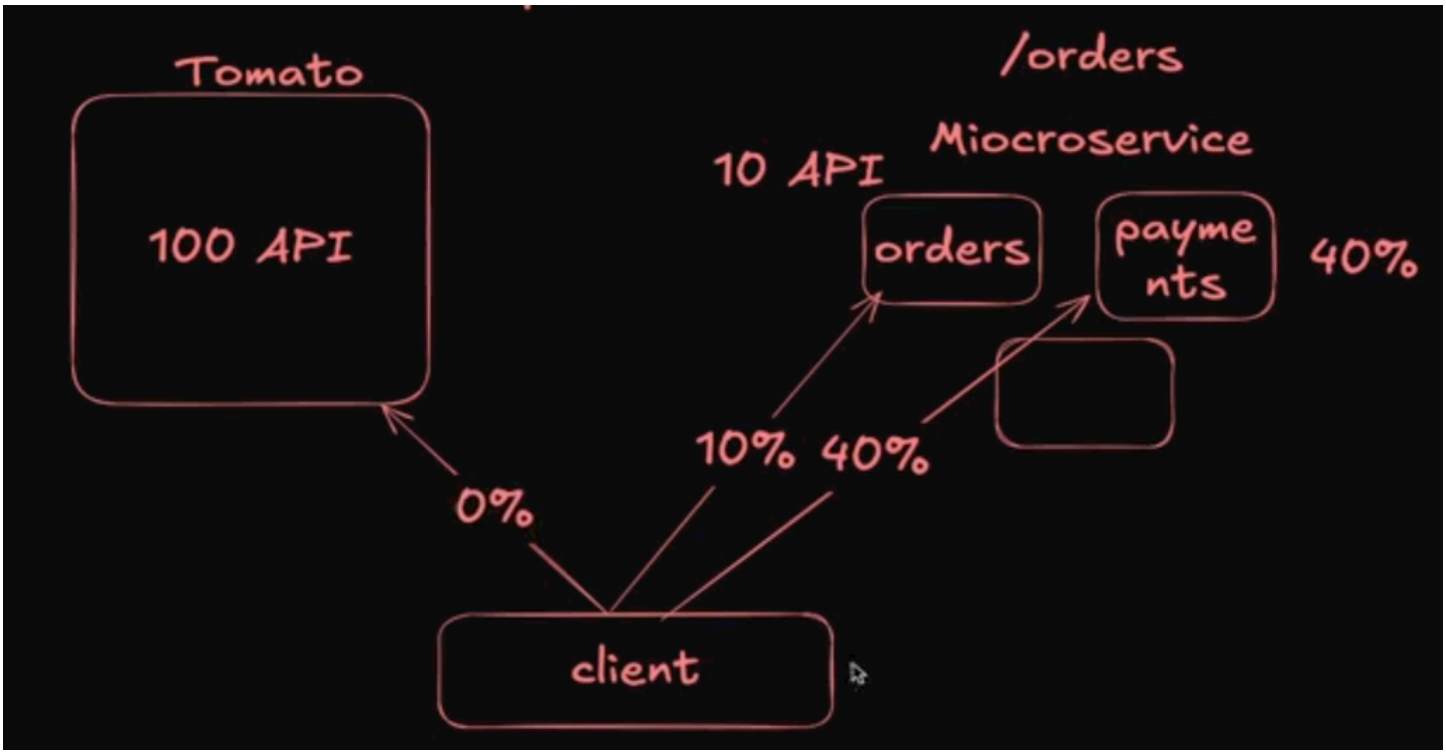
Like a strangler fig tree:

- **It grows around an old tree**
- **Slowly replaces it**

Same idea in software.

3.How it works (Step by Step)

1.You already have a Monolith app
2. Create a new microservice for one feature
3. Route traffic for that feature to the new service
4.Keep old system running
5.Repeat until monolith is gone



Restaurant Example

Old system (Monolith)

One big restaurant:

- **Order**
- **Menu**
- **Payment**
- **Delivery**

All in one system

Apply Strangler Pattern

Step 1:
Move Payment to a new Payment Service

Client

↓

API Gateway

↓

Payment Service (new)

↓

Monolith (for other features)

Step 2:
Move Order Service
Step 3:
Move Delivery Service

Eventually ->No monolith left

Real Example (E-commerce / Instagram type apps)

- **First move Login Service**
- **Then Feed Service**
- **Then Notification Service**

System keeps running all the time

Advantages

- **No big downtime**
- **Less risk**
- **Easy rollback**
- **Business keeps running**

Disadvantage

- **Temporary complexity**
- **Two systems to maintain**
- **Needs good routing (API Gateway)**

Strangler Pattern = Replace old system slowly, not all at once

Strangler Pattern is used to migrate from monolith to microservices by gradually replacing parts of the system while keeping the old application running.

2.Database Phase

In microservices, there are **two approaches**:

1.Shared Database
2.**Unique Database (Database per Service)**

1.Shared Database

What it means

Multiple microservices use the same database.

Example

- Order Service
- Payment Service
  both use one common DB

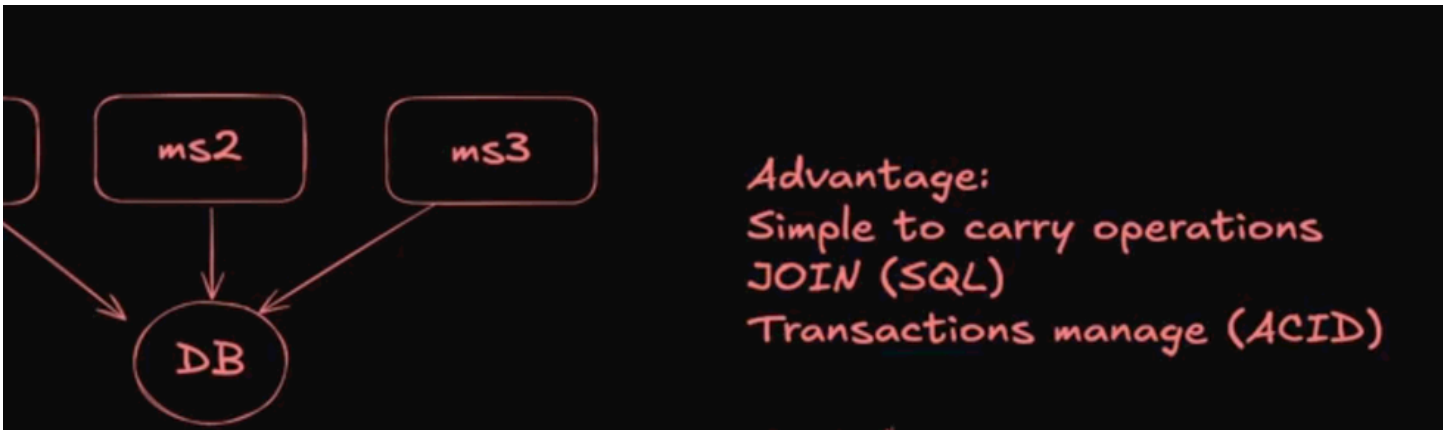
Advantages (Shared DB)

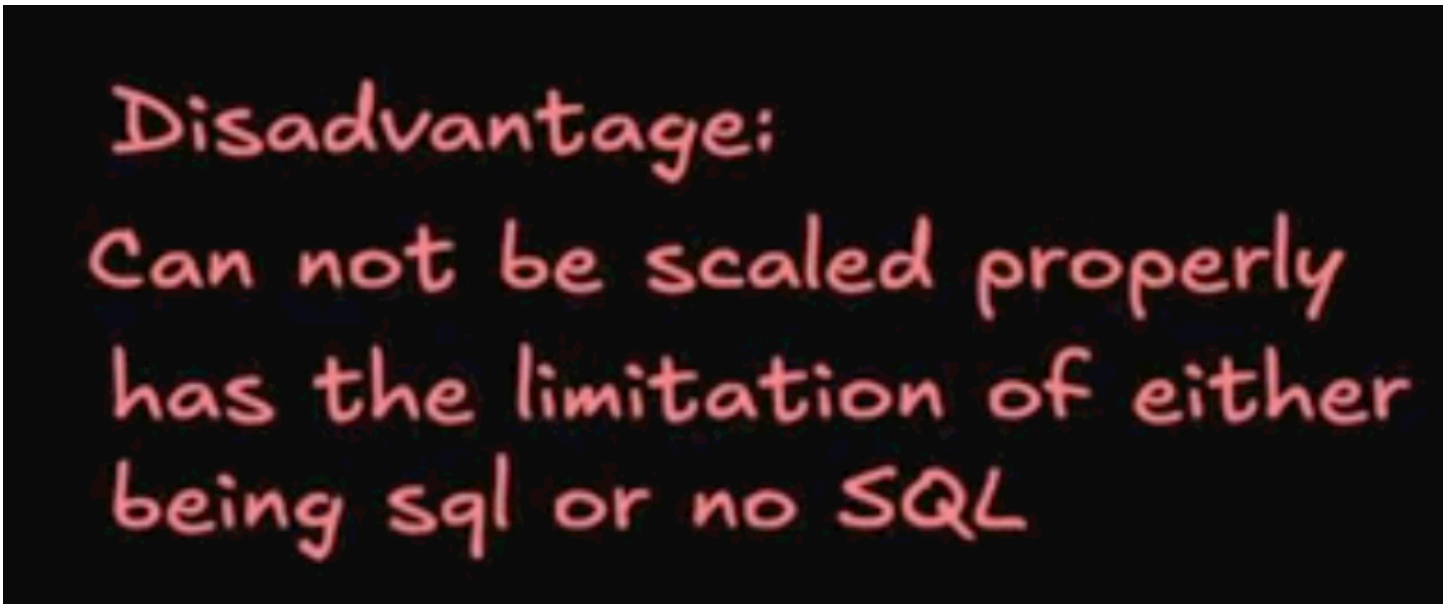
- Easy to implement
- Simple joins & queries
- Less databases to manage

Disadvantages (Shared DB)

- Services are tightly coupled
- One DB change affects all services
- Scaling is difficult
- One DB failure → many services down

  Breaks microservice principle





2.Unique Database (One DB per Service)

What it means

Each microservice has its own database.

Example

- Order Service → Order DB
- Payment Service → Payment DB

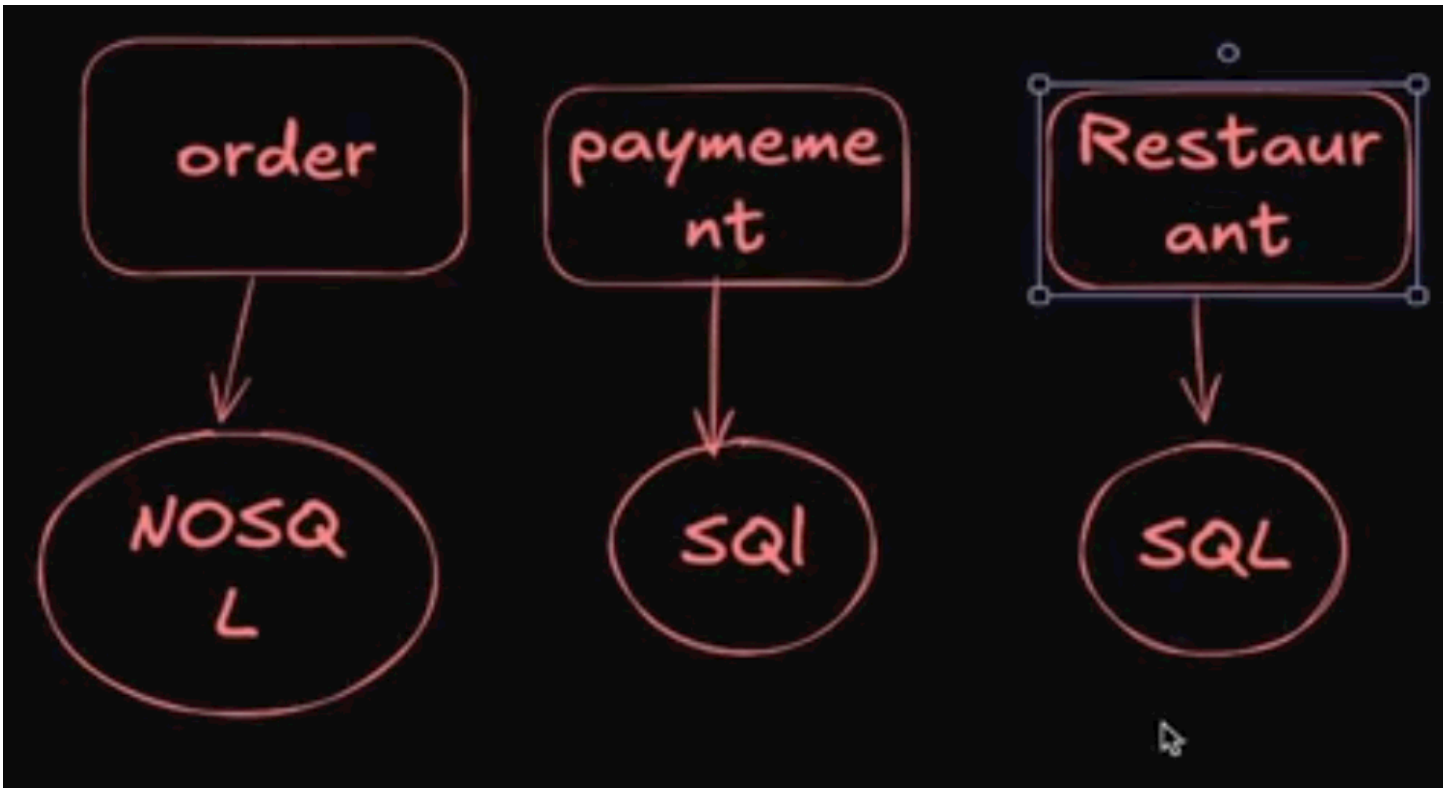
Advantages (Unique DB)

- Services are fully independent
- Easy to scale one service
- DB failure affects only one service
- Freedom to use different DB types

Best practice in microservices

Disadvantages (Unique DB)

- No direct joins(To resolve this we use CQRS)
- Data consistency is harder
- More databases to manage
- Transaction Management.(To solve this we use SAGA Pattern)



SAGA Pattern

1.What is SAGA Pattern?

SAGA Pattern is used to manage data consistency in microservices when each service has its own database.

There is no single transaction like in monolith.
Instead, a sequence of small transactions is used.

If something fails → compensating actions are triggered.

2.Why SAGA is needed?

In microservices:

- Each service has its own DB
- Distributed transaction (2PC) is not good

So we use SAGA to keep data consistent.

Restaurant Example

Order flow (Success case)

1.Order Service → create order
2.Payment Service → payment success
3.Delivery Service → assign delivery

Order completed

Failure case (Payment fails)

1.Order created
2.Payment failed
3.Compensation action:

- Cancel order
- Release items

System comes back to correct state

2.Two Types of SAGA

1.Choreography-based SAGA

No central controller

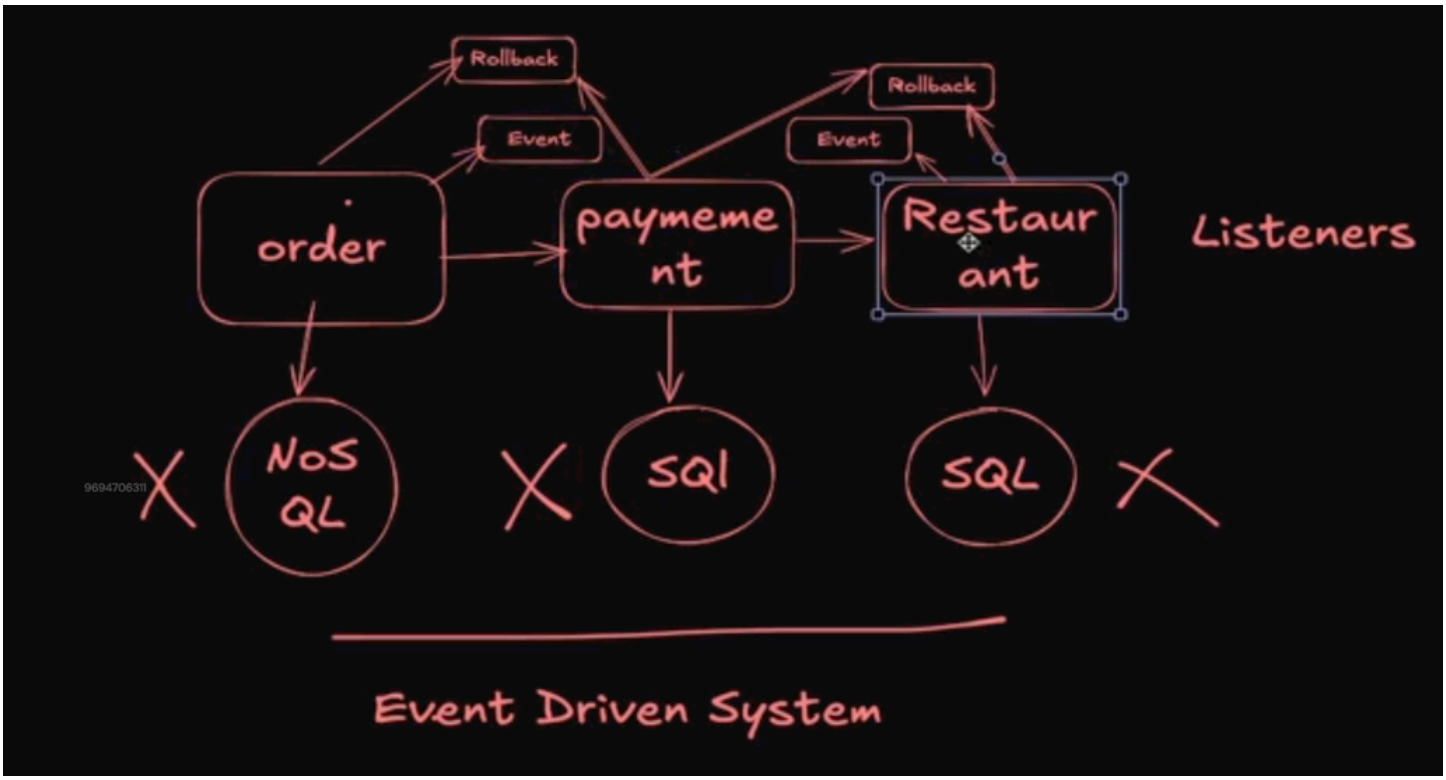
- Services talk using events
- Each service knows what to do next

Example:

- Order Service → emits OrderCreated
- Payment Service → listens → processes payment
- Delivery Service → listens → assigns delivery

Hard to debug

Flow is hidden



2.Orchestration-based SAGA

Central coordinator (Saga Orchestrator)

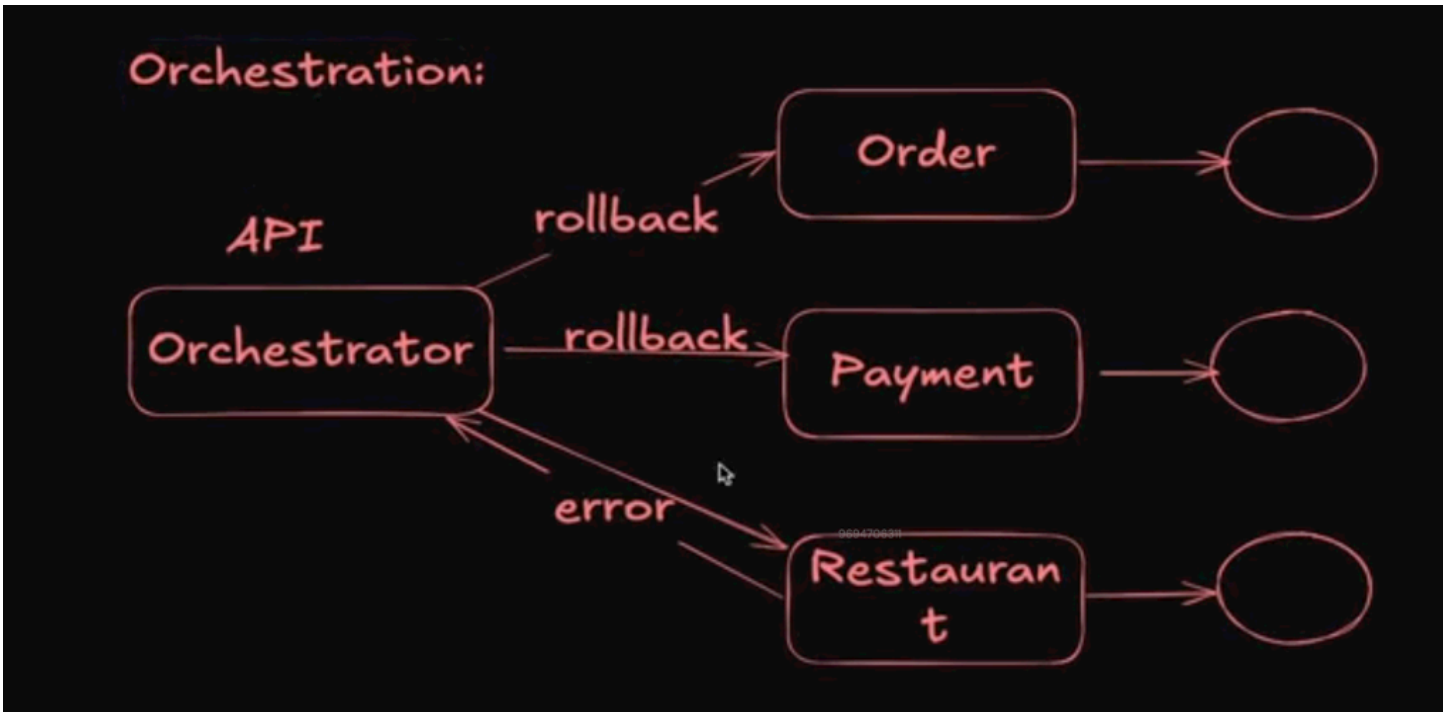
- Orchestrator tells services what to do
- Easier to manage

Example:

- Orchestrator → Order Service
- Orchestrator → Payment Service
- Orchestrator → Delivery Service

Easy monitoring

Clear flow



Advantages

- Maintains data consistency
- Works well with database-per-service
- No global locks

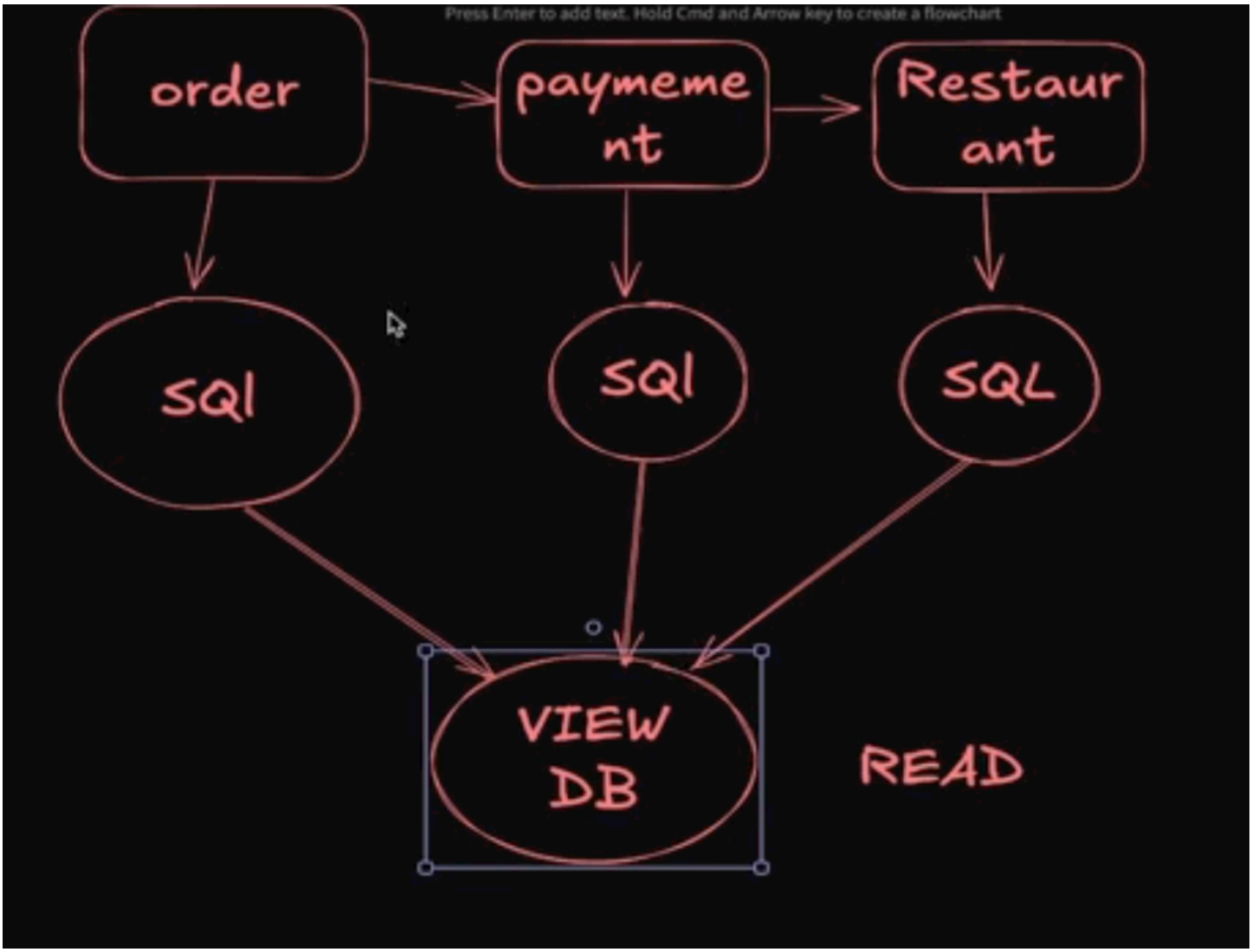
Disadvantages

- Complex logic
- Eventual consistency
- More code & monitoring needed

SAGA = Step-by-step transaction + undo on failure

Saga Pattern manages distributed transactions in microservices by breaking them into local transactions and using compensating actions on failure.

CQRS



1.What is CQRS?

CQRS stands for Command Query Responsibility Segregation.

It means:

- **Commands = write data**
- **Queries = read data**
- **Read and Write are separated**

Do NOT use the same model for reading and writing data.

Restaurant Example

Without CQRS (Normal)

One system does:

- **Place order (write)**
- **View order (read)**

Same database, same model.

With CQRS

Split into two parts:

Command Side (Write)

- **Place Order**
- **Cancel Order**
- **Update Order**

Uses Write DB

👀 Query Side (Read)

- **View Orders**
- **Order History**
- **Order Status**

Uses Read DB (optimized for reads)

Flow

1.Order placed → Command model
2.Data saved → Write DB
3.vent published
4.Read DB updated
5.User reads from Read DB

Real World Example (Instagram)

- **Command: Like a post**
- **Query: View likes count**

Writes and reads are handled separately.

Advantages

- **Faster reads**
- **Better scalability**
- **Different DBs for read & write**
- **Clean separation of logic**

Disadvantages

- **More complexity**
- **Eventual consistency**
- **Extra infrastructure**

Command = Change data

Query = Read data

CQRS = Separate both

CQRS separates read and write operations into different models to improve scalability and performance.

3.Communication (How Services Talk)

What it means

Services talk to each other using APIs or messages.

Common ways

- **REST API (HTTP)**
- **gRPC**
- **Message Queue (Kafka, RabbitMQ)**

Example

Order Service → calls → Payment Service API

👉 **Communication is network-based, not direct code calls.**

4.Deployment (How Services Run)

What it means

Each service is deployed separately.

Example

- **Order Service updated → only Order Service redeployed**
- **Payment Service keeps running**

Tools

- **Docker**
- **Kubernetes**

Benefit: No full system downtime

5.Observability (Monitoring & Debugging)

What it means

Ability to see what is happening inside services.

Includes

- Logs – what happened
- Metrics – performance (CPU, response time)
- Tracing – request flow across services

Example

Track a request from:
Order → Payment → Delivery

Tools

- Prometheus
- Grafana
- ELK
- Jaeger

Without observability, microservices are blind