📘 Lecture 3: *CAP Theorem* | Back of the Envelope Calculation | Monolithic vs Microservice Architecture

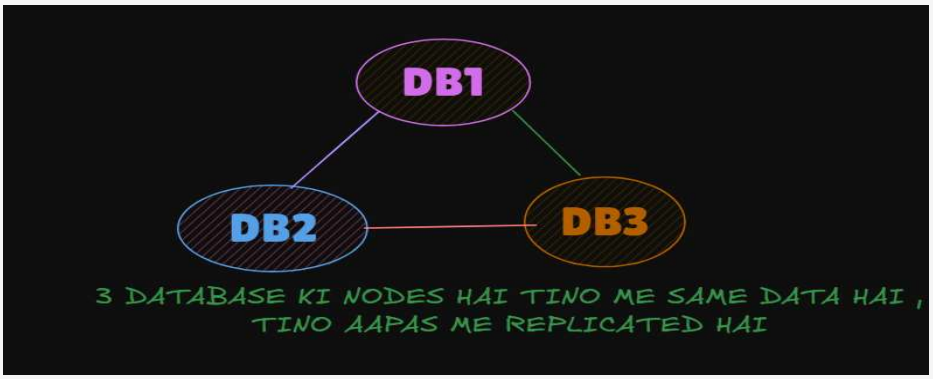
---

⚡ CAP Theorem क्या है?

👉 यह theorem बताती है कि scalable और distributed system बनाने के लिए हमें 3 properties में से केवल **2 ही चुननी पड़ती हैं।**
👉 Centralized system ❌ scalable नहीं होता, इसलिए हमें distributed DB चाहिए।

3 Properties of CAP:

- 🔷 **C = Consistency**
- 🔷 **A = Availability**
- 🔷 **P = Partition Tolerance**



📌 Rule → **एक समय पर केवल 2 properties ही achieve कर सकते हैं।**

🔷 1) Consistency (C)

👉 सभी clients को हर समय **same और updated data** मिलना चाहिए।

📊 **Example with db1, db2, db3:**

- User A ने **db1** में profile pic बदली।
- उसी समय db2 और db3 पर भी updated pic replicate होनी चाहिए।

- अगर सभी followers को same नई pic दिखी → ✅ **Consistent system**।
- अगर किसी को पुरानी pic और किसी को नई दिखी → ❌ **Inconsistent**।

💡 Example:
👉 Instagram profile pic update → सबको तुरंत same pic दिखनी चाहिए।

🔷 2) Availability (A)

👉 Client की हर request का **response आना चाहिए** (चाहे पुराना data हो या नया)।

📊 **Example with db1, db2, db3:**

- User B ने **db2** से request भेजी।
- भले ही db2 busy है या sync नहीं हुआ → फिर भी उसे कोई response देना होगा।
- मतलब app हमेशा चालू रहेगी → ❌ response कभी रुकना नहीं चाहिए।

💡 Example:
👉 Instagram feed scroll → थोड़ा slow load होता है लेकिन app crash नहीं होती।

🔷 3) Partition Tolerance (P)

👉 जब कोई DB node fail हो जाए या nodes आपस में connect न कर पाएं, तब भी system काम करता रहना चाहिए।

📊 **Example with db1, db2, db3:**

- मान लो db1 और db2 के बीच network टूट गया।
- फिर भी db3 और db1 को data serve करना चाहिए।
- यानी system का कुछ हिस्सा fail हो जाए, तो भी पूरा system down नहीं होना चाहिए।

💡 Example:
👉 YouTube, Instagram जैसी apps कभी पूरी तरह से down नहीं होतीं → क्योंकि इनके पास partition tolerance होता है।

🔷 CAP Theorem Combinations :

Distributed system me **Consistency (C), Availability (A), Partition Tolerance (P) –** ye 3 properties hoti hain.
👉 Lekin ek system ek saath tino achieve nahi kar sakta.
👉 **Rule:** Out of 3 → maximum 2 properties choose kar sakte ho.

⚡ 3 Possible Combinations

1. **CA → Consistency + Availability**
2. **CP → Consistency + Partition Tolerance**
3. **AP → Availability + Partition Tolerance**

💡 **Important:** Partition Tolerance (P) ko ignore nahi kar sakte, kyunki distributed systems me **network failures** hone hi hote hain.
➡️ Matlab: **Partition Tolerance toh hamesha hona hi chahiye.**

🟢 Case 1 → CP (Consistency + Partition Tolerance)

```
[User]
  |
DB1 (Update → Profile Pic 🖼️)
  |
DB2 (Sync from DB1 ✅)
  |
DB3 (Wait till sync ✅)
```

👉 Flow:

1. User DB1 पर profile pic update करता है।
2. DB1 पहले खुद update करता है और फिर DB2 और DB3 को sync कराता है।
3. जब तक sync पूरा नहीं होता, system **write operation रोक देता है**।
   मतलब नया profile pic update करने का option temporarily block हो जाता है।

📌 **Result**:

- सभी DBs (DB1, DB2, DB3) पर data same (Consistency ✅)
- लेकिन users को wait करना पड़ता है (Availability ❌ Compromise)

🟡 Case 2 → AP (Availability + Partition Tolerance)

```
[User]
   |
  DB3 (Update → Profile Pic 🌄)
   |
 /   \
DB1 ❌   DB2 (No Sync / Old Data)
```

👉 Flow:

1. User DB3 पर profile pic update करता है।
2. Update DB2 तक पहुँच गया लेकिन DB1 तक नहीं पहुँचा।
3. अब situation:
    - अगर user DB3 से देखेगा → updated pic ✅
    - अगर user DB2 से देखेगा → old pic ❌
    - DB1 अभी भी पुराना data दिखा रहा है।

📌 **Result**:

- System हमेशा response दे रहा है (Availability ✅)
- लेकिन हर जगह data same नहीं है (Consistency ❌ Compromise)

🔴 Why Partition Tolerance is Mandatory?

```
[Users] → Load Balancer → DB1 ✅
                        → DB2 ✅
                        → DB3 ❌ (Down)
```

👉 मान लो DB3 crash हो गया।
👉 Load Balancer users की request को DB1 और DB2 पर redirect कर देगा।

👉 System up रहेगा (Partition Tolerance ✅), लेकिन आपको **Consistency vs Availability** में से एक चुनना ही होगा।

✨ Final Golden Rule

- **Consistency + Partition Tolerance (CP)** → Data हमेशा same रहेगा, लेकिन update late होगा।
- **Availability + Partition Tolerance (AP)** → System हमेशा response देगा, लेकिन हर user को अलग data मिल सकता है।
- **Partition Tolerance (P)** → हर distributed system में ज़रूरी है, वरना पूरा system down हो जाएगा।

Consistency vs Availability Kab Kya Prefer

🟢 Consistency

📌 Jab **up-to-date response** देना mandatory ho, tab **consistency** bhi mandatory hoti hai.

🏦 Example: Banking Application

- Ek account se doosre account me **payment** kiya gaya.
- Agar:
    - Pehle account se paise **cut gaye**
    - Lekin doosre account me **credit nahi hue**
- To transaction **rollback ho jaayega** (fail ho jaayega).

🔒 Behavior:

- Jab tak system **consistent nahi hota**, customer **next transaction** nahi kar sakta.
- Aisi applications:
    - ✅ **"Up" bhi hoti hain** (system chal raha hai)
    - ✅ **"Maintained" bhi hoti hain** (data safe hai)

- 🔁 **Transaction loss nahi hota**

🟡 Availability

📌 Jab **up-to-date response optional** ho, lekin **kisi bhi response ka milna zaroori** ho, tab **availability** important hoti hai.

📱 Example: Social Media Apps

- Instagram
- LinkedIn
- Twitter

❓ Kyun availability zaroori hai?

- Kyunki ye apps ka goal hai ki:
    - User ko **turant response** mile
    - Chahe wo response **old data** ho

✅ App **open ho jaani chahiye**
❌ Chahe profile pic purani ho, ya post thodi late dikhe

📊 Availability Numbers

Availability	Downtime (Per Year)
100%	⛔ **Impossible** (Ideal case)
99%	❌ 3.65 days/year
99.9%	✅ 8.77 minutes/year *(Default metric)*
99.99%	✅ 52.6 minutes/year

99.999%	⭐ ~5.25 minutes/year *(High-end systems like Google, Amazon, Microsoft)*
98% – 97%	❌ Poor performance

🔺 Note:

- **Availability %** batata hai ki ek application saal bhar me **kitna time down** rahegi.
- ✅ **99.9%** is considered **default acceptable standard**
- ⭐ **99.999%** (Five nines) is for **mission-critical systems**

📘 Back of the Envelope Calculations :

🔍 Purpose

Estimate karna:

- 📏 Application kitni badi hai
- 👥 Kitne users handle kar sakti hai
- ⚙️ Platform us load ko support kar payega ya nahi

⚡ Key Concepts

✅ QPS (Queries Per Second)

- Ek second mein application par kitni **read/write** queries aati hain.
- **Read** → `SELECT`
- **Write** → `INSERT`, `UPDATE`, `DELETE`

🔷 QPS se kya pata chalta hai?

- Database par **load** kitna hai
- Kitni **CPU/RAM** chahiye hogi

- **Cache** lagega ya nahi

✅ Storage Units (S.U.)

- Total data **store** karne ke liye DB ko kitni memory chahiye
- Future planning ke liye hardware ki sizing ka estimation

📏 Understanding Data Units (Power of Two)

📐 Computers binary system use karte hain, isliye sizes powers of 2 me calculate kiye jaate hain:

Unit	Size	Power of 2
1 KB	1,024 Bytes	2^{10}
1 MB	1,024 KB	2^{20}
1 GB	1,024 MB	2^{30}
1 TB	1,024 GB	2^{40}
1 PB	1,024 TB	2^{50}
1 EB	1,024 PB	2^{60}

🔢 Data Size Reference (Power Index)

Power	Equivalent	Unit
10	Thousand	Kilobyte
20	Million	Megabyte
30	Billion	Gigabyte
40	Trillion	Terabyte
50	Quadrillion	Petabyte
60	Quintillion	Exabyte

🗓️ Planning for 5 Years

Agar application 5 साल tak continuously chalni hai, toh:

- Data storage
- Hardware requirement
- Scalability — sabka long-term estimate lena zaroori hai

📱 Instagram – Estimation Assumptions

🔷 User Base

- **Monthly Active Users ≈ 2 Billion**
- **Daily Active Users** (60%) **≈ 1.2 Billion**

🔷 User Activity (Daily Average)

- 1 user → **80 feed views**
- 1 user → **1 photo/reel upload**

🔷 System Impact

- Har **feed view** → 1 read query → hits the DB
- Har **post** (photo/video) → 1 write query → hits the DB

🔢 QPS Calculations

📥 Feed View QPS

```
Total daily feed views = 1.2B users * 80 = 96B
QPS = 96B / 86400 ≈ 1,111 queries/sec
```

➡️ **Feed View QPS ≈ 1,111/sec**

📤 Post Upload QPS

```
Total daily posts = 1.2B users * 1 = 1.2B
QPS = 1.2B / 86400 ≈ 13,888/sec
Peak QPS = 2 * 13,888 = ~27,776/sec
```

➡️ **Post Upload QPS ≈ 14K/sec**
📈 **Peak QPS** (high traffic time) **≈ 28K/sec**

📝 Holidays ya weekends me traffic aur badh sakta hai

💾 Storage Calculations

🧮 Compression-Based Assumption

- **20% uploads** = **Videos** (Average size: 50MB)
- **80% uploads** = **Photos** (Average size: 3MB)

🖼️ Photos Storage per Day

```
Photos/day = 1.2B * 80% = ~1B
1B * 3MB = 3 Billion MB = 3 * 2^20 * 2^30 = 3 * 2^50 = 3 PB
```

📦 **Photos Storage/Day ≈ 3 Petabytes**

🎥 Videos Storage per Day

```
Videos/day = 1.2B * 20% = ~0.24B
0.24B * 50MB = 12 Billion MB = 12 * 2^50 = 12 PB
```

📦 **Videos Storage/Day ≈ 12 Petabytes**

📊 Total Daily Storage

```
Total = Photos + Videos = 3 PB + 12 PB = 15 PB/day
```

➡️ **Total Storage Needed per Day** = **15 Petabytes**

🗓️ Storage Requirement for 5 Years

```
15 PB/day * 365 days/year * 5 years
= 15 * 365 * 5 = 27,375 PB
≈ 27,000 PB (approx.) = 27 * 10^15 Bytes
= 27 Exabytes (approx.)
```

📂 **5-Year Total Storage ≈ 27 Exabytes**

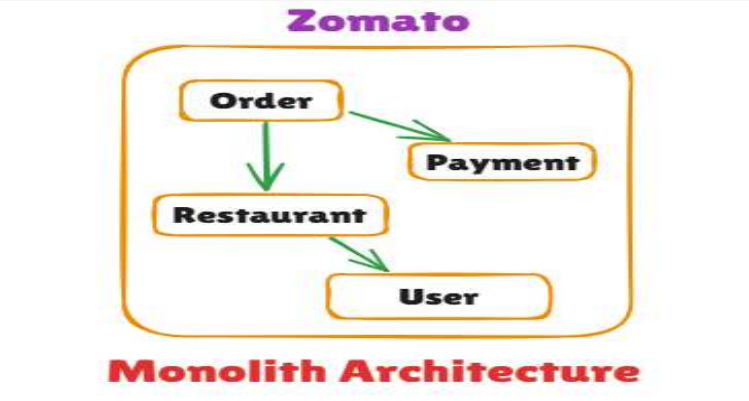
✅ Sirf **photo/video uploads** ke liye
❗ Baaki system logs, metadata, comments, likes, etc. iske alawa hai

⚙️ Monolith vs Microservice – Notes

🏗️ Monolith Architecture

🔷 Definition

Ek **single application** jisme poora business logic ek hi jagah rehta hai.
📦 **Single codebase**
🔁 Sabhi modules tightly coupled hote hain (Order, Payment, User, etc.)



📌 **Example (Zomato):**

```
Order

Payment

Users

→ Sab ek hi application ke andar handle hote hain
```

✅ Advantages of Monolith

- 🚀 **Fast Development**

    - Saare features (Order, Payment) ek file mein likh sakte ho.
- 🧩 Easy to start
    - Setup simple hota hai, ek hi codebase manage karna hota hai.

❌ Disadvantages of Monolith

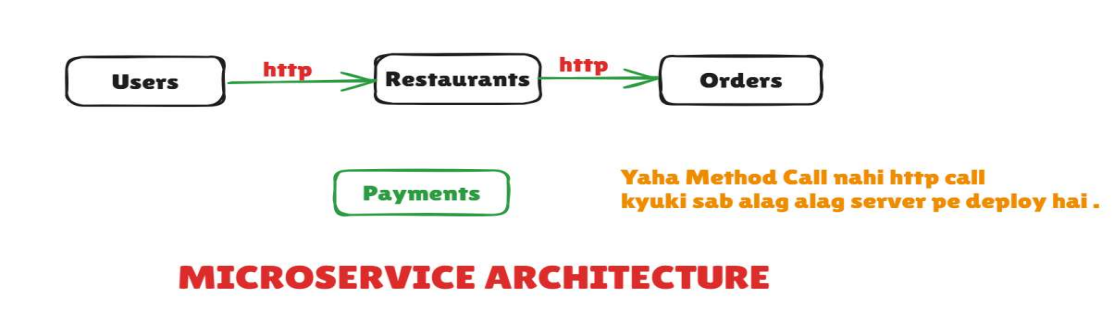
- 📈 **Scalability Issues**
    - Agar "Order" service scale karni hai, to puri application scale karni padti hai.
- 📦 **Heavy Codebase**
    - Naye developers ko code samajhne mein dikkat hoti hai.
- 🐞 **Debugging/Testing Tightly Coupled**
    - Ek module fail ho to poori application down ho sakti hai.
- 🔁 **Full Redeployment Required**
    - Ek chhoti change ke liye bhi puri application ko deploy karna padta hai.

🧩 Microservice Architecture

🔷 Definition

Application ko multiple **independent services** mein divide kiya jata hai – each with its own logic and deployment.
🌐 Services communicate using **HTTP calls**
🚀 Alag-alag servers par deploy hote hain

📌 Diagram (Zomato Style)



```
User -----> HTTP -----> Order
             ↓           ↓
            HTTP        Payment
```

✅ Advantages of Microservice

- 📈 **Better Scalability**
  - Sirf jis service ko scale karna hai, wahi scale karo.
- 🔄 **Independent Deployment (CI/CD)**
  - Agar Order service mein change hai, to sirf Order deploy karo.
- 🧪 **Faster Testing & Debugging**
  - Har service independent hai, toh issue track karna easy hota hai.

❌ Disadvantages of Microservice

- 🕒 **Latency / Slower Performance**
  - Services ke beech **HTTP calls** hone ki wajah se response slow ho sakta hai.
- 🔄 **Transaction Management Issues**
  - Ek saath multiple microservices ko join karna mushkil hota hai (distributed transactions).

🧩 Different Phases of Microservice Architecture

Microservices architecture ko implement karne ke liye 6 key phases hoti hain:

1️⃣ Decomposition

➡️ Monolith ko chhoti-chhoti microservices mein तोड़ना

2️⃣ Database

➡️ Shared DB use karein ya har microservice ke liye unique DB?

3️⃣ Communication

➡️ Microservices आपस में कैसे बात करेंगी? (API, Event-driven, Messaging)

4️⃣ DevOps

➡️ Microservice-based system ka deployment pipeline kaise hoga?

5️⃣ Deployment (CI/CD)

➡️ Har service ka independent deployment setup kaise hoga?

6️⃣ Observability

➡️ Logs, monitoring, alerting — System को monitor kaise karoge?

🔷 1. Decomposition Phase

> ❓ *How will you break a Monolith into Microservices?*

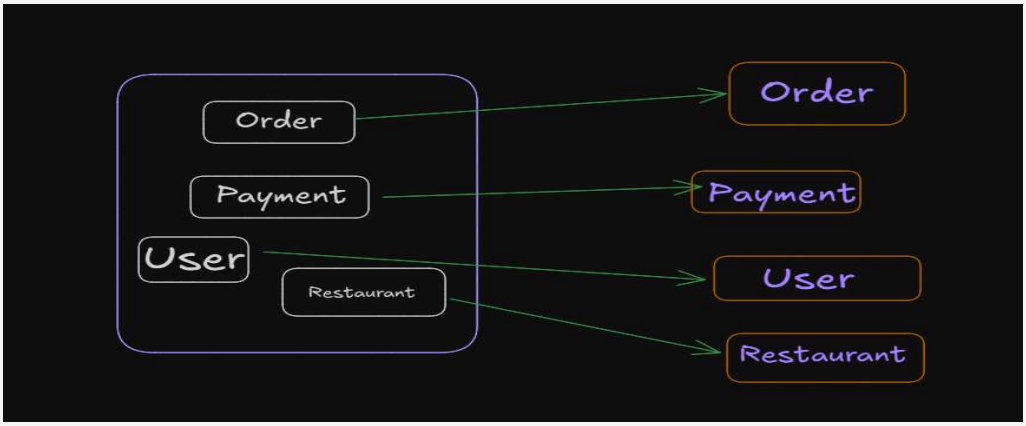
Microservices architecture ka **pehla aur sabse important step** hota hai — breaking the Monolith.

```
Monolith → Microservice 1 + Microservice 2 + Microservice 3 ...
```

🧠 A. Decomposition by Business Logic

👉 Business functionalities ke basis par Monolith को अलग-अलग logical services में divide किया जाता है.

📌 Example:



⚠️ **Disadvantage:**
Is approach mein developer ko **पूरी business logic की समझ** होनी चाहिए. Nahi to services ka split galat ho sakta hai.

🧠 B. Decomposition by Sub-domain

👉 Web-based systems mein **sub-domain structure** ke basis par breakdown kiya ja sakta hai.

📌 Example: Zomato (Monolith)

🔗 Sub-domain URLs:

```
www.zomato.com/orders/pay      →  Order-related microservice

www.zomato.com/payments/       →  Payment-related microservice
```

➡️ Result:

- `/orders` → 1 Microservice
- `/payments` → 1 Microservice

🎯 **Advantage:** Sub-domain based decomposition se **clear microservice boundaries** define ho jaati hain.

🔄 Strangler Pattern

> ❓ *How to gradually migrate Monolith to Microservices without downtime?*

Strangler Pattern ek migration strategy hai jisme Monolith system ko धीरे-धीरे Microservices में convert किया जाता है — bina pura system break kiye.

🧩 Zomato Example:

- Monolith → 100 APIs
- Client → 100 users
- New Microservices Introduced:
  - ✅ **Orders Microservice** (10 APIs)
  - ✅ **Payments Microservice** (10 APIs)

🔁 Traffic Split

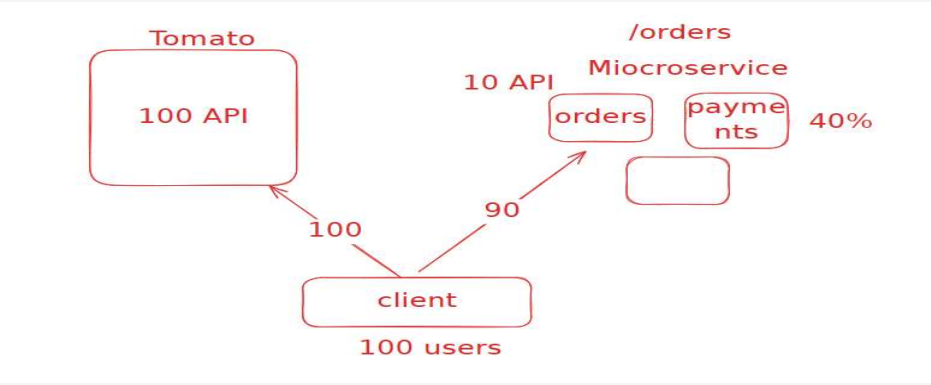
🔷 Initially:

```
Client → 100% traffic goes to Monolith
```

🔶 Gradually:

```
Client → 10% traffic → Orders/Payments Microservices

         90% traffic → Still handled by Monolith
```



⏳ Over Time:

- More APIs are moved out of Monolith
- Traffic gradually shifts to Microservices

- Monolith becomes lighter and eventually deprecated

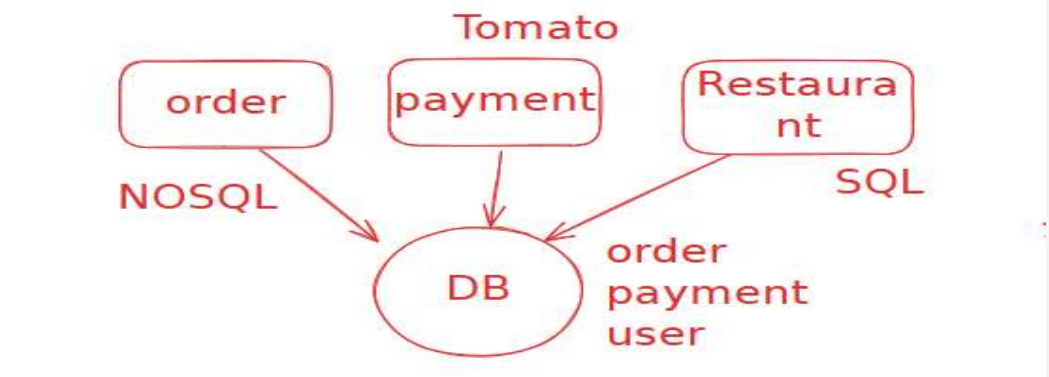
🎯 **Goal:**
👉 Step-by-step, safe migration
👉 **No downtime**, no system crash
👉 Better scalability & maintainability

🔷 2. DATABASE Phase :

Each microservice can have:

- **Shared DB**
- **Unique DB**

1️⃣ Shared DB



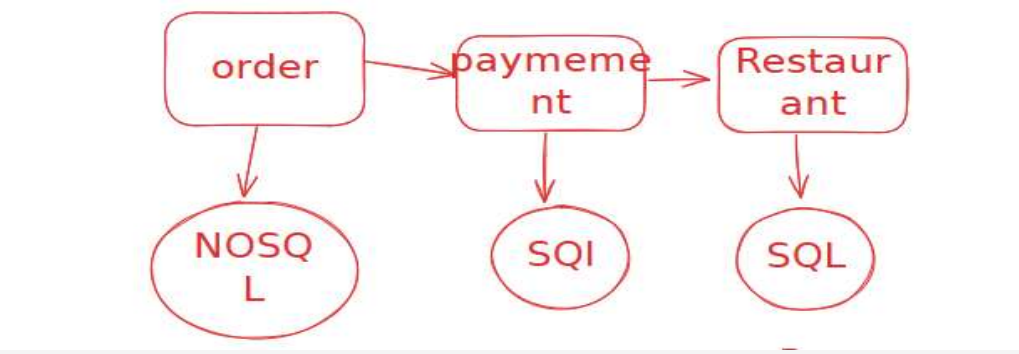
✅ **Advantage:**

- Simple to operate
- Supports **JOINs (SQL)**
- Supports **Transaction Management (ACID)**

❌ **Disadvantage:**

- Cannot be scaled properly
- Limitation of either being only **SQL** or only **NoSQL**
- **NoSQL (MongoDB)** is better suited for **Orders**

2️⃣ Unique DB



➡️ AGAR **ORDER JYADA** AA RAHE HAI TO ONLY **ORDER WALA DB KO CHANGE KREGE**

- ORDER KA SERVER PAYMENT KE SERVER PAR CALL **NAHI KAR** SKTA.

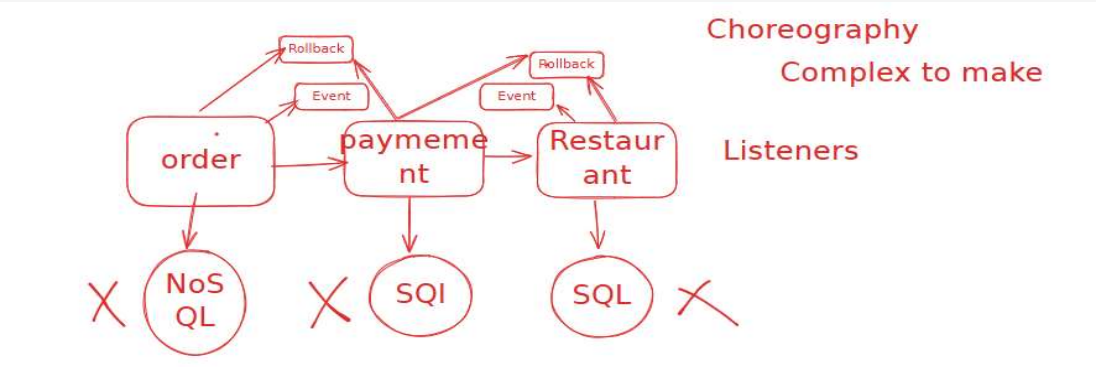
❌ **Disadvantages:**

- JOIN not possible
  → Use **CQRS** (design pattern) to solve

- Transaction management not possible
  → Use **SAGA Pattern** to solve
  → Transaction rollback bhi success hoga

🔁 SAGA Pattern

🔶 Choreography (Complex to manage)



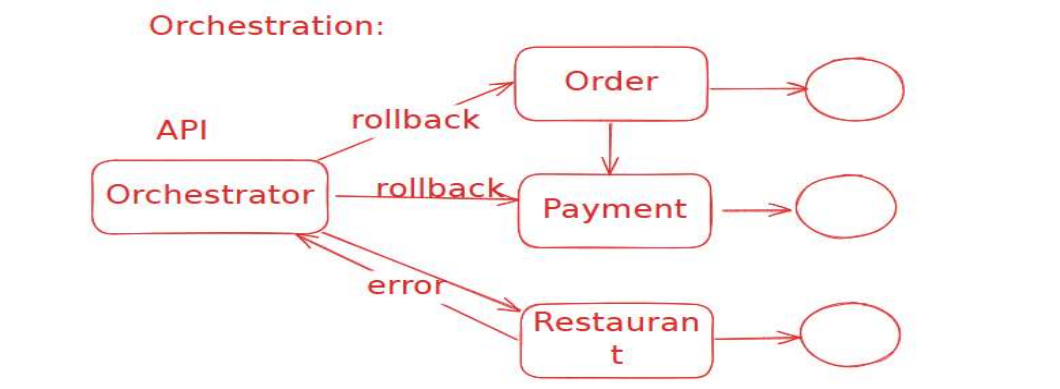
➡️ **Event-driven system**:

- Jab **Order** apne DB me kuch change karega, ek **event publish** karega
- **Payment** and **Restaurant** are **Event Listeners**
- Jab bhi change hoga, ye listen karenge

- Agar **Restaurant** ne kuch disconnect kar diya, to transaction success nahi hoga → system **rollback** karega

---

🔶 Orchestration



- Teeno services ek hi jagah se kaam karenge
- Jo **Orchestrator** karega, wahi final hoga
- Agar **Payment** disconnect hua, to **Restaurant** ko bhi **rollback** kar sakta hai

---

📗 CQRS (Command Query Responsibility Segregation)

➡️ Teeno services ek **View DB** ko point karenge for **READ (JOIN)**

- Supports **Master-Slave model**
- Do tarah ke operations honge:
  - **Read**
  - **Write**
- SQL DB ka **common operation integration** hoga **saare DBs ka**