**Microservices in System Design**

Microservices is an architectural style where an application is broken into smaller, independent services that communicate via APIs.

**When to Use Monolithic vs. Microservices Architecture**

| **Factor** | **Monolithic Architecture** | **Microservices Architecture** |
| --- | --- | --- |
| **Best for** | Small applications | Large, scalable applications |
| **Scalability** | Hard to scale specific parts | Can scale individual services |
| **Deployment** | Entire application redeployed | Independent deployments |
| **Development Speed** | Faster for small teams | Faster for large teams working in parallel |
| **Technology Choice** | Single technology stack | Different services use different stacks |
| **Fault Isolation** | A bug can crash the whole system | A failure in one service doesn’t affect others |

**Microservices Example: Uber**

**1️⃣ Initial Phase (Monolithic System)**

* Uber started with a **monolithic** system where a single application handled:
  + User Authentication
  + Ride Matching
  + Payment Processing
  + Notifications
* **Problems:**
  + As Uber expanded globally, **a single database became a bottleneck**.
  + Code changes affected the entire system, slowing down updates.
  + A failure in one feature (e.g., payment) could crash the whole system.

**2️⃣ Transition to Microservices**

✅ **Breaking the Monolith:**

* Uber split its system into microservices such as:
  + **User Service** (handles authentication)
  + **Ride Matching Service** (connects riders & drivers)
  + **Payment Service** (processes payments)
  + **Notification Service** (sends alerts & messages)

✅ **Database Sharding:**

* Instead of a single database, Uber used **sharded databases** for different microservices.

✅ **Asynchronous Communication:**

* **Kafka (event-driven architecture)** was used to allow services to communicate asynchronously.

**3️⃣ Final Scalable System**

* **Each service can be deployed independently** without affecting others.
* **Auto-scaling:** The ride-matching service can scale separately from the payment service.
* **Different technologies**: Uber’s **ride-matching** runs on Node.js, while **payments** use Java.
* **Fault isolation:** If the **notification service** fails, ride-matching continues to work.

**Key Takeaways**

* **Use Monolithic Architecture** for small applications where simplicity is key.
* **Use Microservices** when the system needs to scale independently, handle high traffic, or support multiple tech stacks.
* **Use event-driven communication (Kafka, RabbitMQ)** when microservices need to work asynchronously.
* **Use API gateways** to manage communication between microservices efficiently.