Microservices

Introduction to microservices

What is Microservices Architecture?

Microservices architecture is a design style in which applications are composed of small, loosely coupled, independently deployable services. Each service in a microservices architecture is designed to handle a specific

business function and communicate with other services via APIs, typically using HTTP/REST, gRPC, or message queues.

Characteristics of Microservices:

1. **Independence:** Each microservice runs in its own process and communicates with others through well-defined APIs.
2. **Decentralization:** Data management and storage are decentralized, with each service managing its own database or repository.
3. **Scalability:** Individual services can be scaled independently based on demand.
4. **Flexibility:** Developers can choose different technology stacks for different services.
5. **Resilience:** Failures in one microservice do not necessarily bring down the entire system.

Why Use Microservices Architecture Over Monolithic Architecture?

In a **monolithic architecture**, the entire application is developed, deployed, and scaled as a single unit. Although simple to develop in the early stages, monolithic applications can grow to become too complex, making it hard to scale, maintain, or introduce changes. Here's a detailed comparison and explanation of why microservices architecture is preferred in certain situations.

---

1. Scalability

- **Monolithic Architecture:** In a monolithic system, scaling is typically done by replicating the entire application. This means you must scale even the components that don't need it, leading to inefficient resource usage.
- **Microservices Architecture:** Each microservice can be scaled independently. For example, if only the payment service needs more resources, you can scale just that part of the system without affecting other services.

2. Development Speed and Flexibility

- **Monolithic Architecture:** In a monolithic architecture, different parts of the system often depend on each other, which can make development and

testing slow. A small change in one part of the application may require the entire application to be redeployed.

- **Microservices Architecture:** Since microservices are independent, teams can work on different services simultaneously. Continuous integration and deployment practices allow faster release cycles and easier management of changes.

3. Fault Isolation

- **Monolithic Architecture:** In a monolithic system, a failure in one module (e.g., payment processing) can crash the entire application, causing downtime.
- **Microservices Architecture:** Microservices architecture ensures fault isolation. If a service fails, it does not bring down the entire application. Other services can continue to operate normally, improving system resilience.

4. Technology Diversity

- **Monolithic Architecture:** All parts of the system must use the same technology stack, limiting flexibility. If you want to introduce a new technology, you might need to refactor large parts of the system.
- **Microservices Architecture:** Each microservice can use different programming languages, databases, or frameworks as required. For example, one service can be written in Java while another is in Python, allowing the best tool for each job.

5. Code Maintenance and Complexity

- **Monolithic Architecture:** As the application grows, the codebase can become large and difficult to maintain. Developers have to understand the entire system, and introducing new features or making changes becomes more complicated.
- **Microservices Architecture:** The code for each microservice is smaller and easier to manage. Teams can focus on maintaining and enhancing their specific services without worrying about the entire application.

6. Deployment Flexibility

- **Monolithic Architecture:** In a monolithic architecture, you typically need to deploy the entire application, even if only one small component has changed, leading to slower deployment processes.
- **Microservices Architecture:** Individual services can be deployed independently without affecting the rest of the system. This allows for more frequent updates and quicker time to market.

7. Better Organization for Large Teams

- **Monolithic Architecture:** In large teams, it can be difficult to manage a monolithic architecture as everyone works on the same codebase.
- **Microservices Architecture:** Large organizations can divide their development teams into smaller, independent teams, each responsible for a particular service. This leads to better collaboration and parallel development.

Microservices vs. Monolithic Architecture (Summary Table)

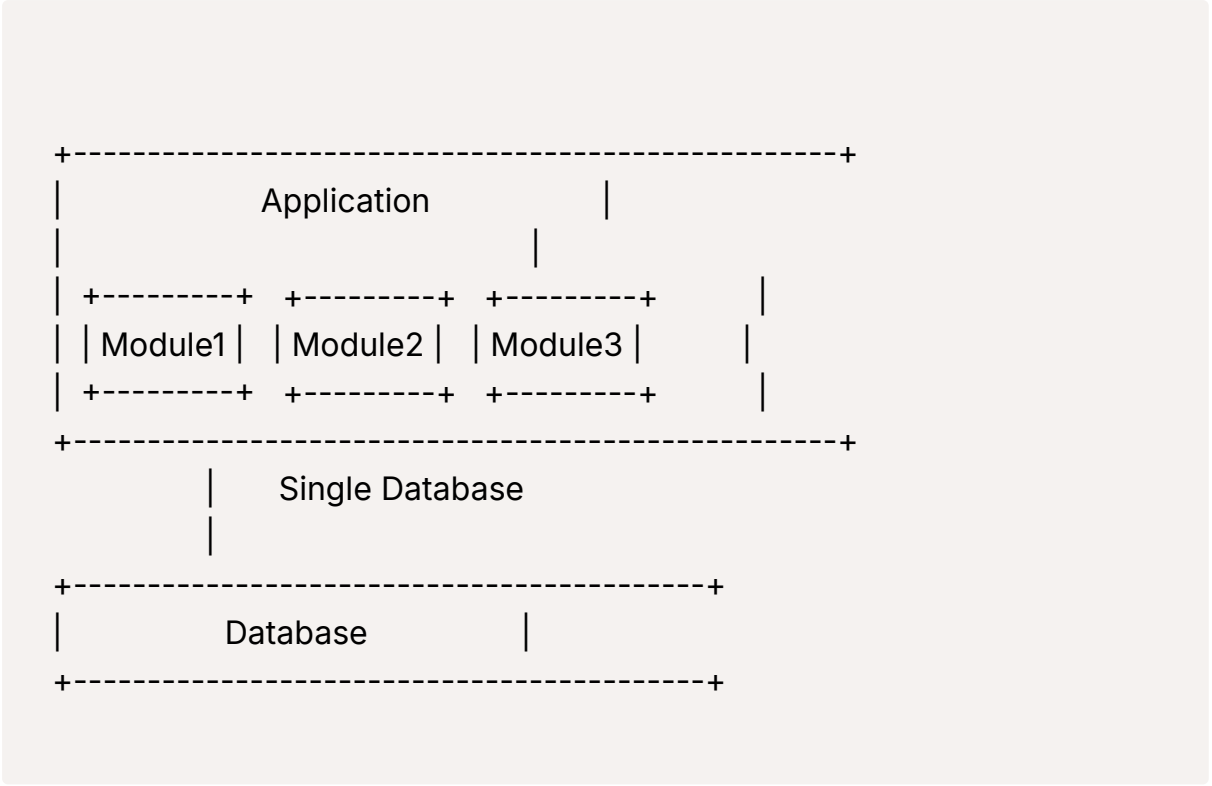
Aspect	Monolithic	Microservices
Scalability	Entire app scaled together	Independent scaling of services
Development Speed	Slower due to interdependencies	Faster with independent teams and services
Fault Tolerance	Single failure affects entire app	Isolated failures with minimal impact
Technology Stack	Single stack for entire app	Multiple stacks per service
Deployment	Deploy entire app on change	Independent service deployment
Complexity	Increases with app size	Easier to manage smaller, independent services
Maintenance	Difficult as app grows	Easier due to independent codebases
Team Management	Large teams work on the same codebase	Small teams focus on specific services

---

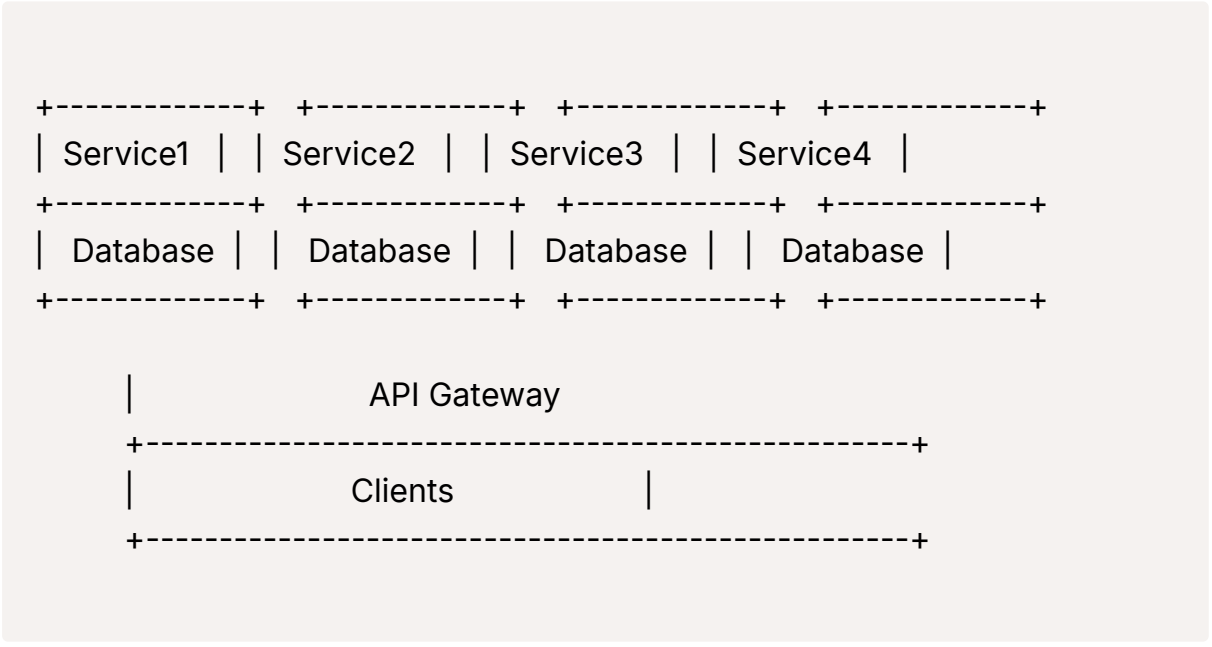
Diagram of Microservices vs Monolithic Architecture

Here's a simple diagram to illustrate the difference between **Monolithic** and **Microservices Architecture**.

Monolithic Architecture Diagram:



Microservices Architecture Diagram:



In the **Monolithic Architecture** diagram, all modules are tightly coupled and share a single database. Changes in one module may affect the entire application.

In the **Microservices Architecture** diagram, each service is independent, has its own database, and communicates with others through APIs. This setup provides flexibility, scalability, and resilience.

Microservices Limitations

1. Increased Complexity

- **Challenge:** Managing multiple services is much more complex than managing a single monolithic application. You need to handle inter-service communication, service discovery, load balancing, and managing dependencies between services.
- **Example:** Instead of deploying a single application, you now have to manage and deploy numerous independent services, which adds overhead.

2. Distributed Systems Challenges

- **Challenge:** Microservices rely on network communication between services, which introduces latency, potential network failures, and security concerns.
- **Example:** A network failure between two microservices could lead to a system outage if not properly handled with retries, fallbacks, or circuit breakers.

3. Data Consistency and Management

- **Challenge:** In a monolithic system, a single database ensures data consistency across the application. In microservices, each service can manage its own database, leading to issues with maintaining **eventual consistency** across the system.
- **Example:** Implementing **distributed transactions** across multiple services is complex and requires mechanisms like **sagas** or **eventual consistency** models.

4. Testing and Debugging

- **Challenge:** Testing microservices is more difficult compared to monolithic applications. Testing the integration of services, especially when they are developed independently, requires thorough coordination.
- **Example:** Unit tests are easier to manage, but **end-to-end testing** becomes challenging as you need to simulate the interactions between all services and handle potential network issues.

5. Deployment Complexity

- **Challenge:** Deploying and managing multiple microservices involves more complex infrastructure and tools. Automated deployment, container orchestration (e.g., Docker, Kubernetes), and monitoring services are essential but difficult to set up and maintain.
- **Example:** While you can deploy each service independently, managing dozens or hundreds of services with version control, rollback, and updates can be overwhelming.

6. Service Dependency Management

- **Challenge:** Microservices often depend on each other to function, and managing these dependencies (especially if they evolve independently) can lead to version conflicts and challenges in coordinating updates.
- **Example:** If Service A relies on an old version of Service B, but Service B gets updated, you may have to ensure backward compatibility or update Service A as well.

7. Latency and Performance Overhead

- **Challenge:** Communication between microservices, which typically involves RESTful APIs or message queues, adds network latency compared to internal method calls within a monolith.
- **Example:** A request that would take milliseconds within a monolithic app may take significantly longer in a microservices setup due to inter-service network delays.

8. Security

- **Challenge:** Each microservice has its own entry point, and securing each one individually is more difficult than securing a single monolithic

application. You need to manage **authentication, authorization, and encryption** across all services.

- **Example:** Managing access control between services, securing APIs, and ensuring consistent security standards across all services can be complex and error-prone.

9. Operational Overhead

- **Challenge:** Operating microservices at scale requires advanced infrastructure, monitoring, and management tools like **Kubernetes**, **service meshes**, and **distributed tracing** tools (e.g., Jaeger, Zipkin).
- **Example:** Monitoring the health and performance of multiple services in a distributed system can be more difficult than monitoring a monolith. Centralized logging, tracing, and alerting tools are needed to track and debug issues.

10. Team Coordination and Ownership

- **Challenge:** While microservices allow different teams to work independently, coordination is still required to ensure that services interact correctly. Miscommunication between teams can lead to misaligned APIs and integration issues.
- **Example:** If teams are not synchronized on API changes, one service might update its API without notifying dependent services, causing failures.

11. Cost of Resources

- **Challenge:** Running many microservices can increase infrastructure costs. Each service might need its own set of resources, such as databases, memory, and compute power, even for small workloads.
- **Example:** The operational cost of managing multiple microservices with their own resources can be higher than running a single monolithic application, especially in environments with limited resources.

12. Skill Set and Expertise Required

- **Challenge:** Implementing microservices requires developers, DevOps engineers, and architects to have expertise in cloud platforms, containerization, networking, and automation tools. This can be a barrier for teams unfamiliar with distributed systems.

- **Example:** A team used to working with monolithic applications may struggle to shift to microservices unless they undergo proper training in distributed systems and tools like Kubernetes.

Lets Create Microsevices

1. Course Services uses Postgress
2. Category Service uses mysql
3. Video Service uses mongo db

Flow in video to create services

Communication

Service Registry/ Service Discovery/ Load Balancing

Edge Server/API Gateway

Config Server

Microservices Resiliency

Event Driven Microservices

Spring Cloud Functions

Apache Kafka

Docker- Containerising Services using Docker