Mastering System Design

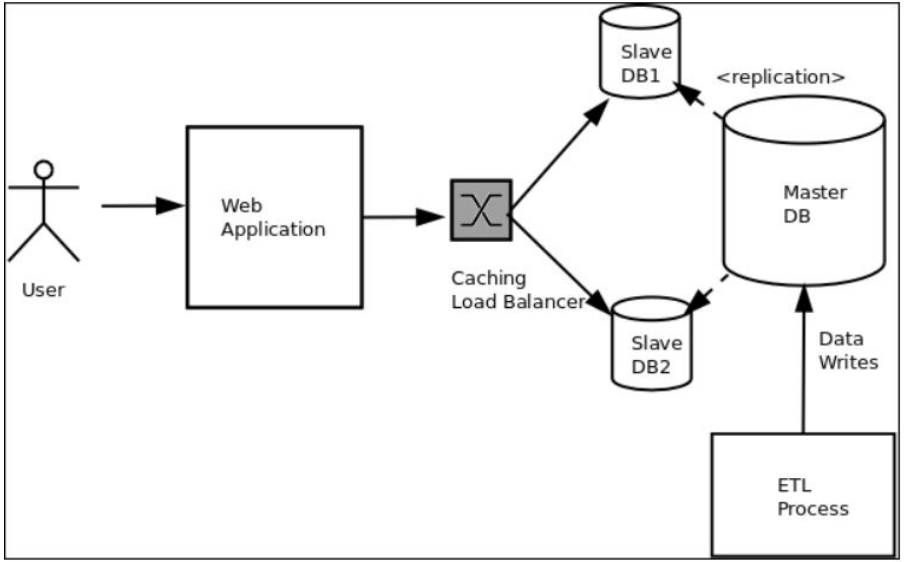
Section 4: Architectural Patterns

System Design Fundamentals

What is Software Architecture?

- **Definition**: The structure of a system, including its components and relationships.
- **Why It Matters**: Architecture impacts scalability, performance, and maintainability.
- Importance in System Design & Key Design Considerations
  - Scalability: Ability to handle increased data or traffic.
  - Maintainability: How easily the system can be updated or modified.
  - Performance: Efficiency and responsiveness under load.

"Keep in mind, the architectural choices you make directly influence system behavior."



Section Agenda

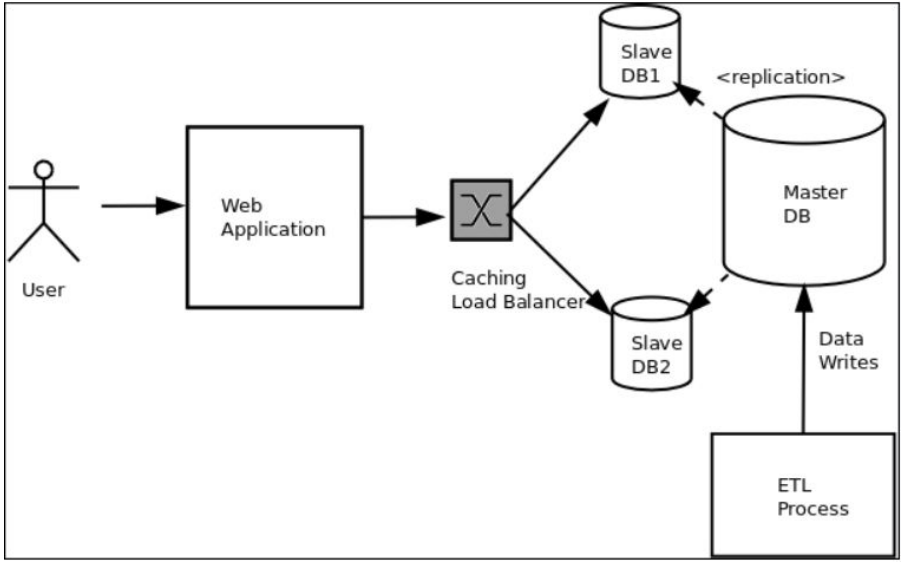
1. Introduction to Architectural Patterns
2. Software Architecture Patterns & Styles
3. Multi-Tier Architecture
4. Microservices Architecture
5. Event-Driven Architecture
6. Summary & Practical Applications

Software Architecture Patterns & Styles

System Design Fundamentals - Architectural Patterns

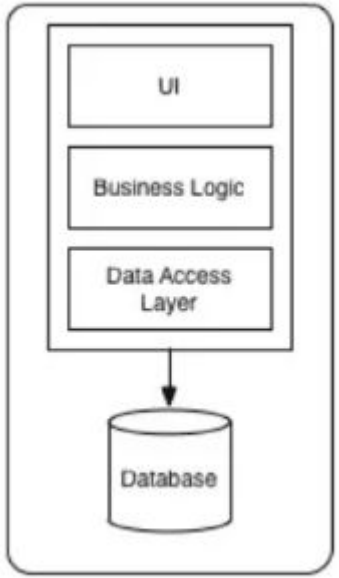
What is Software Architecture?

- **Definition**: The high-level structure of a software system, defining components, their relationships, and the way they interact.
- **Why It Matters**: It impacts scalability, maintainability, performance, and system behavior.
- Common Architectural Styles
  - Monolithic
  - Layered (N-tier)
  - Client-Server
  - Microservices
  - Event-Driven



Monolithic Architecture

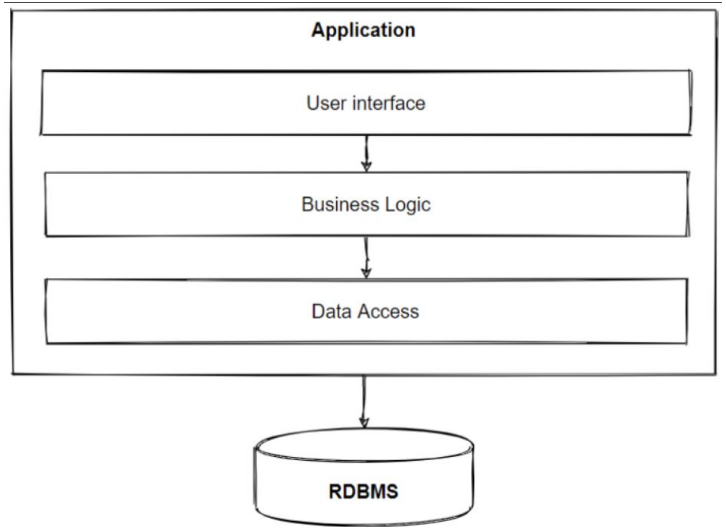
- **Definition**: A single unified system where all components are tightly coupled and work as a single unit.
- Pros:
  - Simple to develop and deploy.
  - Easier to manage in smaller applications.
- Cons:
  - Hard to scale.
  - Difficult to maintain as the codebase grows.
  - High risk of failure: A single bug can take down the entire system.
- Use Cases: Small-scale applications, startups, simple CRUD-based apps.



Monolithic Architecture

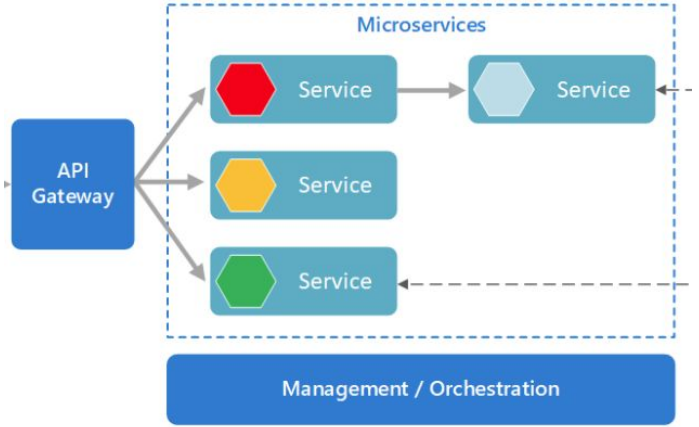
Layered (N-Tier) Architecture

- **Definition**: A system split into multiple layers (e.g., Presentation, Business Logic, Data) to separate concerns.
- Pros:
  - Clear separation of concerns.
  - Easier to scale and maintain compared to monolithic.
- Cons:
  - Performance overhead due to layers.
  - May result in tight coupling between certain layers.
- Use Cases: Enterprise applications, CRM systems, Banking applications.



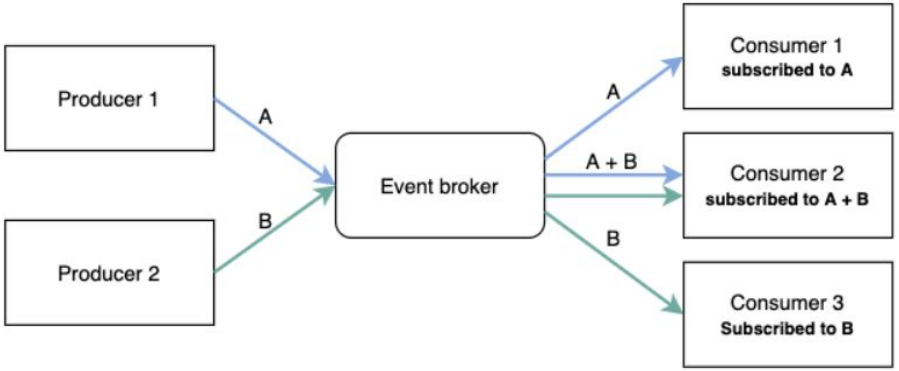
Microservices Architecture

- **Definition**: A system built as a collection of small, independent services, each focused on a specific business capability.
- Pros:
  - Independent services can be scaled, deployed, and developed separately.
  - Flexibility in technology stack for each service.
  - Better fault tolerance.
- Cons:
  - Increased complexity in communication and coordination.
  - Need for robust DevOps and automation pipelines.
- Use Cases: Large-scale applications, e-commerce platforms, modern cloud-based systems.



Event-Driven Architecture

- **Definition**: A system where components communicate through events (messages) instead of direct calls, enabling loose coupling.
- Pros:
  - Highly decoupled architecture.
  - Excellent for handling asynchronous workflows.
  - Better scalability for high-traffic systems.
- Cons:
  - Debugging and tracing become more complex.
  - Difficult to ensure data consistency across services.
- Use Cases: Real-time systems, IoT applications, financial trading platforms.



Factors Influencing Architecture Selection

- Business Needs: What problem are we solving? What is the business goal?
- Scalability: How much traffic and data should the system handle?
- Performance: How fast should the system respond to users?
- Maintainability: How easy is it to make updates, fix bugs, and evolve the system over time?

Interview questions

1. What is the difference between Monolithic and Microservices architecture?
2. When would you choose a Layered architecture over Microservices?
3. What are the pros and cons of Event-Driven Architecture?
4. How does architecture impact system scalability and performance?
5. How do business requirements influence architectural decisions?
6. Can you explain the key differences between Layered and Client-Server architectures?
7. What are the main challenges in maintaining a Monolithic system as it grows?
8. How do you decide between Monolithic and Microservices for a new system?
9. What role does fault tolerance play in Microservices architecture?
10. How do event-driven systems handle real-time data?

Summary & Key Takeaways

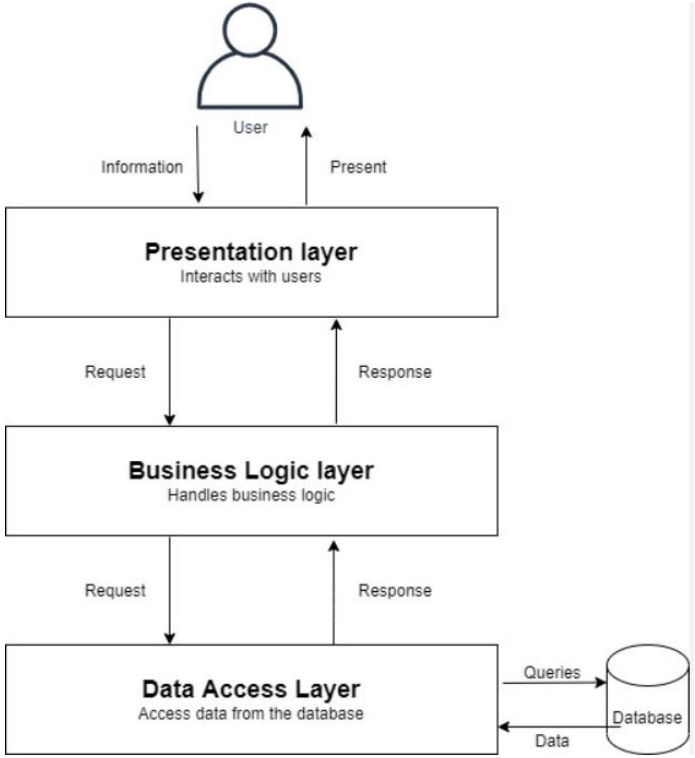
- Understand different architectural styles: Monolithic, Layered, Client-Server, Microservices, Event-Driven.
- Each style has its pros, cons, and best-use cases.
- Architecture decisions impact scalability, performance, maintainability, and system behavior.
- Choose architecture based on business needs, technical requirements, and scalability goals.
- Whats next:
  - Multi-Tier Architecture

Multi-Tier Architecture

System Design Fundamentals - Architectural Patterns

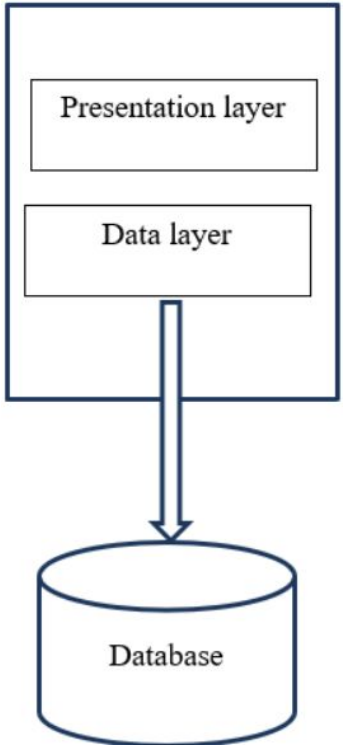
What is Multi-Tier Architecture?

- Multi-Tier Architecture is a software design pattern that structures applications into multiple layers, each responsible for specific functions. This separation enhances scalability, maintainability, and security.
- Key Points:
  - Organizes applications into independent layers
  - Separates concerns: UI, business logic, and data storage
  - Enables better scalability, performance, and security
  - Used in web applications, enterprise systems, and cloud architectures



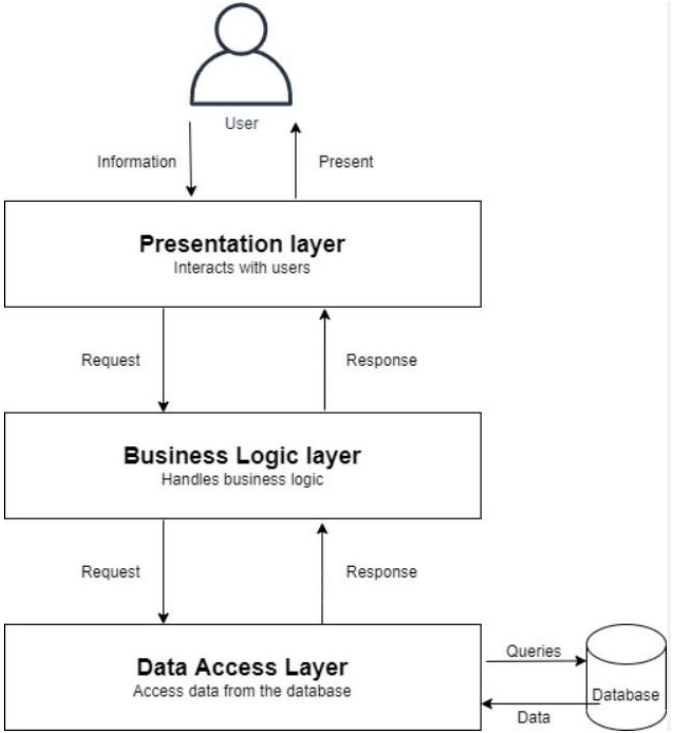
2-Tier Architecture

- **Definition**: The 2-Tier Architecture consists of a Client Layer and a Database Layer. The client interacts directly with the database, without an intermediate business logic layer.
- How It Works:
  - Client Layer: User interface, application logic
  - Database Layer: Stores and retrieves data
  - Data flows directly between the client and the database
- ✅ Pros:
  - ✔️ Simple to implement
  - ✔️ Fast for small-scale applications
- ❌ Cons:
  - ❌ Poor scalability (limited to a few users)
  - ❌ Security risks (direct database access)
- Example Use Case:
- 📌 A desktop application directly querying an SQL database



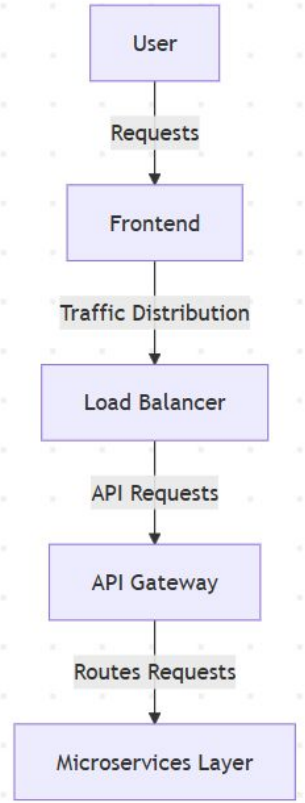
3-Tier Architecture

- **Definition**: 3-Tier Architecture introduces a middle layer (Business Logic Layer) between the UI and the database.
- How it works:
  - The frontend interacts with the business logic layer (API Server).
  - The business logic layer processes requests and communicates with the database.
- Pros:
  - ✅ Improves scalability & security.
  - ✅ Better separation of concerns.
  - ✅ Easier maintenance.
- Cons:
  - Slightly higher latency due to extra processing.
- Example:
  - Traditional web applications



N-Tier Architecture

- **Definition**: N-Tier Architecture extends beyond 3-Tier by adding more specialized layers like caching, API gateway, microservices, etc.
- Why use N-Tier?
  - Handles high-traffic & complex business logic.
  - Allows independent scaling of different services.
- Examples:
  - ✅ Microservices-based applications.
  - ✅ Large-scale enterprise software.



Performance & Scalability Impacts

- How tiers impact system performance & scalability:
  - Latency Considerations:
    - More layers = higher latency if not optimized.
    - Solution: Caching (Redis, Memcached), Load Balancing.
  - Scaling Strategies:
    - Vertical Scaling (adding resources to a single server).
    - Horizontal Scaling (adding more servers, distributing load).

Interview Questions

- Basic Questions:
  - What is Multi-Tier Architecture, and why is it used?
  - How does a 2-Tier architecture differ from a 3-Tier architecture?
  - What are the key components of a 3-Tier architecture?
  - Can you give an example of a real-world application that follows an N-Tier architecture?
- Intermediate Questions:
  - How does scalability improve in a multi-tier system?
  - What are the challenges of adding more tiers in an application?
  - How does load balancing work in an N-Tier architecture?
  - What strategies can be used to reduce latency in a multi-tier application?
- Advanced Questions:
  - In a large-scale system, when would you choose microservices over a traditional N-Tier architecture?
  - How would you secure inter-tier communication in a distributed multi-tier system?
  - How would you design a fault-tolerant multi-tier system for a high-traffic application like Netflix or Amazon?

Summary & Key Takeaways

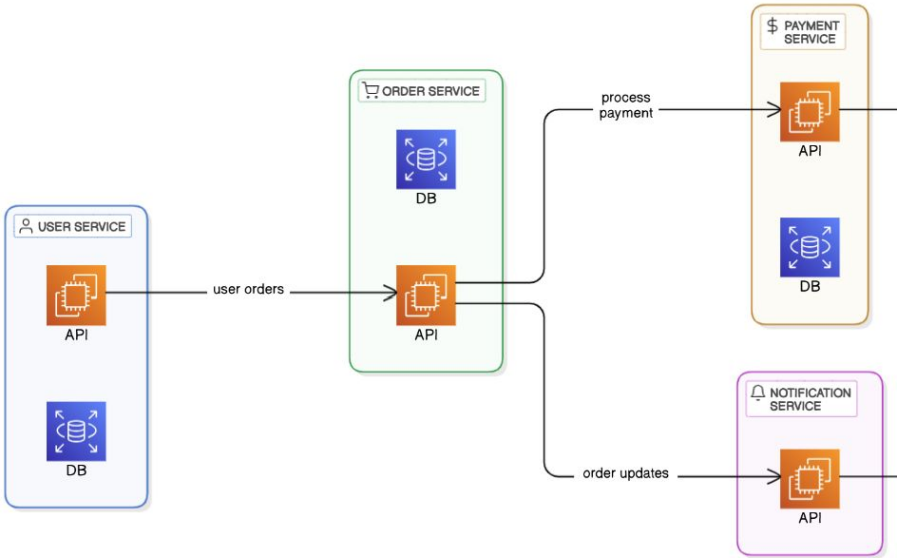
- ✅ Multi-Tier = Better Scalability & Maintainability.
- ✅ 2-Tier is simple but limited.
- ✅ 3-Tier is the standard for web apps.
- ✅ N-Tier is for large-scale, cloud-native systems.
- What's Next:
  - Microservices Architecture

Microservices Architecture

System Design Fundamentals - Architectural Patterns

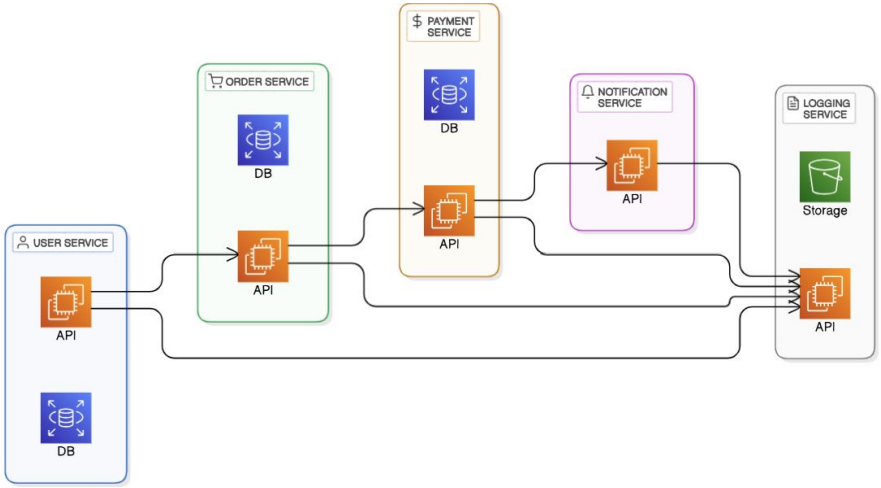
What is Microservices Architecture?

- **Definition**: Microservices architecture is a software design pattern where applications are structured as a collection of small, loosely coupled services, each responsible for a specific function.
- Characteristics:
  - ✅ Independently deployable services
  - ✅ Loosely coupled & modular
  - ✅ Scalable & fault-tolerant



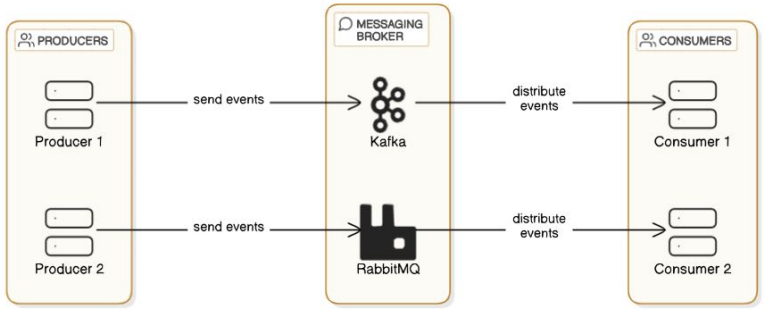
Identifying and Structuring Microservices

- How to Identify Microservices?
  - ✅ Business Capabilities: Each service should align with a specific business function (e.g., Payments, Orders, Users).
  - ✅ Single Responsibility Principle: A microservice should do one thing well.
  - ✅ Data Ownership: Each microservice owns its own database—avoid shared databases.
  - ✅ Independently Deployable: Should be deployable and scalable without affecting others.
- 🏗️ How to Structure Microservices?
  - Decompose by Business Domain: Use Domain-Driven Design (DDD) to group services logically.
  - Define Clear APIs: Services should communicate via well-defined APIs (REST, gRPC, or events).
  - Choose the Right Granularity: Avoid making microservices too large (monolith-in-disguise) or too small (high complexity).
  - Implement Observability: Use logging, monitoring, and tracing to track service interactions.



Communication in Microservices

- Synchronous Communication:
  - REST APIs (Simple, widely used, but high latency)
  - gRPC (Efficient, binary format, better performance)
- Asynchronous Communication:
  - Event-Driven Messaging (Kafka, RabbitMQ, SNS/SQS)

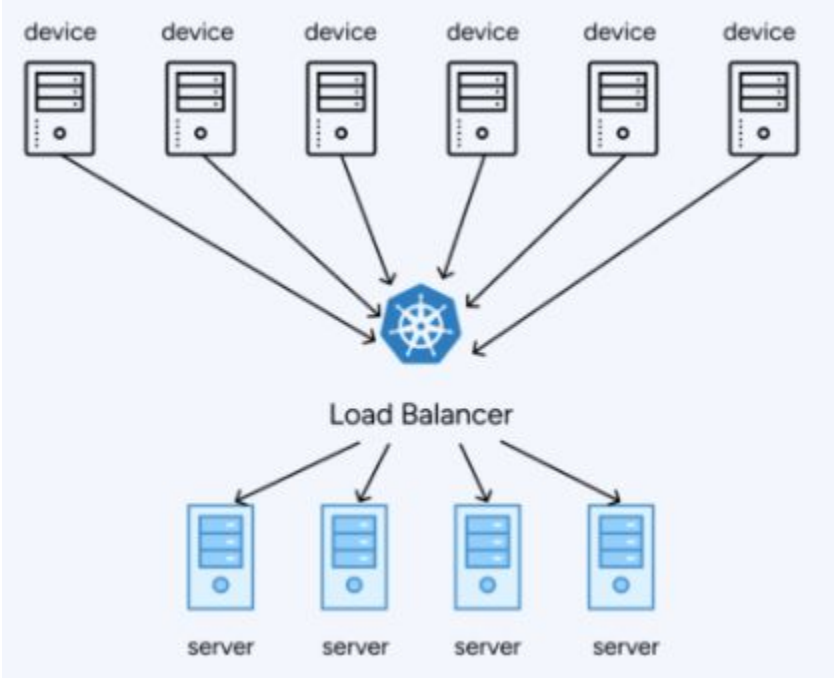


Challenges of Microservices

- ❌ Data Consistency: Distributed databases → Eventual consistency
- ❌ Distributed Tracing: Difficult to debug & track requests
- ❌ Network Overhead: More API calls → Increased latency
- ❌ Security: Authentication, authorization, data protection

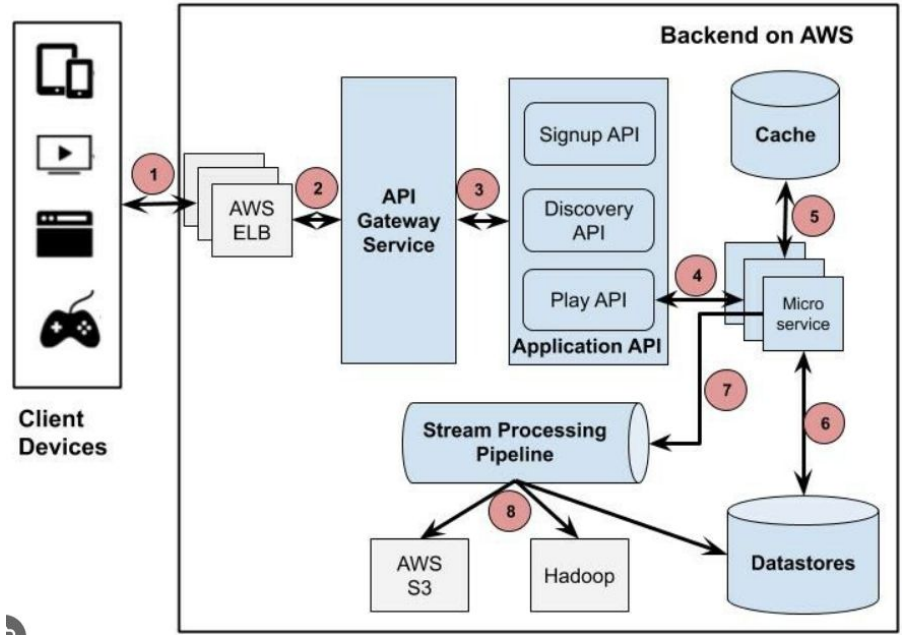
Scaling Strategies in Microservices

- Horizontal Scaling: Add more instances of a service
- Auto-scaling: Scale up/down automatically
- Sharding & Database Scaling: Split databases for high-traffic services



Real-World Examples of Microservices

- Netflix: Uses microservices for video streaming & personalization
- Uber: Scales ride-matching, payments, and navigation independently
- Amazon: Each service (search, payments, recommendations) runs separately



Interview Questions on Microservices

- What are microservices, and how do they differ from monolithic architecture?
- What are the benefits and challenges of microservices?
- Explain the role of an API Gateway in a microservices system.
- How do microservices communicate with each other?
- How can you ensure data consistency in a microservices-based system?

Summary & What's Next

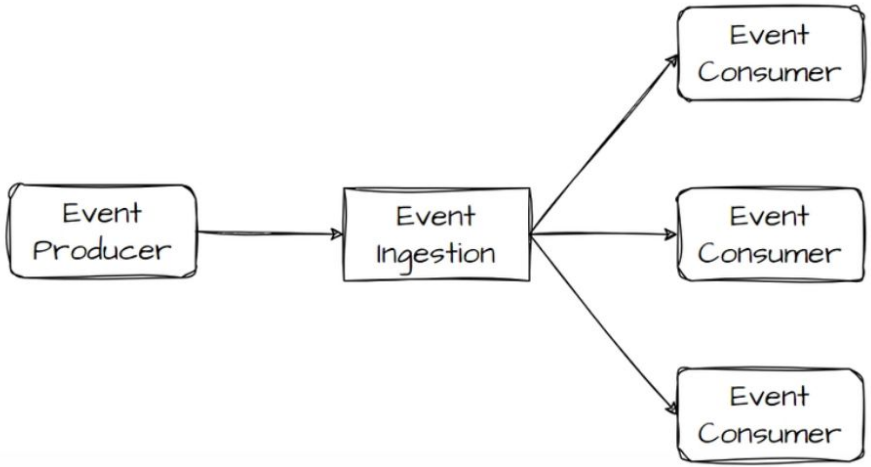
- Microservices = Scalability, Fault Tolerance, Faster Development
- Requires API Gateway, Service Discovery, Load Balancing
- Challenges: Data consistency, debugging, deployment complexity
- Real-world usage: Netflix, Uber, Amazon
- Next Lecture:
  - Event-Driven Architectures!

Event-Driven Architecture

System Design Fundamentals - Architectural Patterns

Introduction to Event-Driven Architecture

- **Definition**: A system design where components communicate through events rather than direct calls.
- Key Characteristics:
  - Asynchronous processing
  - Loose coupling
  - Scalability & flexibility
- Why Use It?
  - Enhances system responsiveness
  - Enables real-time event processing
  - Supports complex workflows

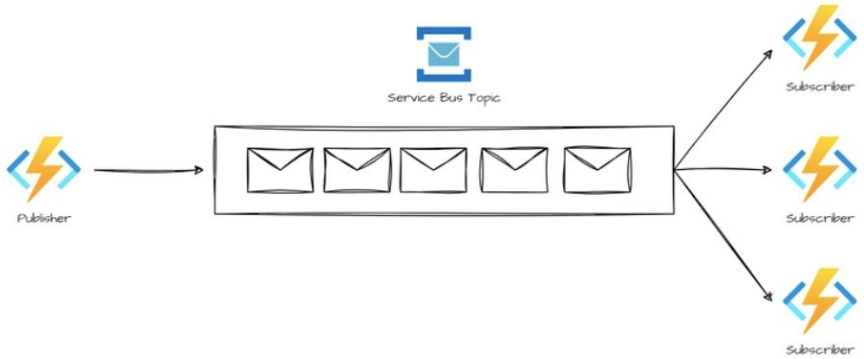


Synchronous vs. Asynchronous Systems

- Synchronous Communication (Request-Response Model)
  - Blocking calls
  - Tight coupling
  - Example: Traditional HTTP APIs
- Asynchronous Communication (Event-Driven Model)
  - Non-blocking
  - Decoupled components
  - Example: Message queues & event brokers

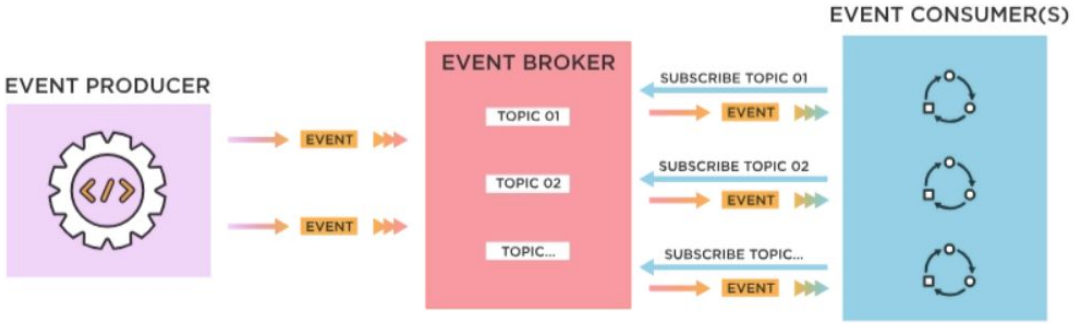
Pub-Sub vs. Event Streaming

- Publish-Subscribe Model (Pub-Sub)
  - Events are broadcasted to multiple subscribers
  - Each subscriber gets the event once
  - Example: RabbitMQ, AWS SNS
- Event Streaming
  - Events are stored & consumed in order
  - Consumers process events at different times
  - Example: Kafka, AWS Kinesis



Key Components of an Event-Driven System

- Event Producers (Generate events)
- Event Brokers (Transmit & store events) - Examples: Kafka, RabbitMQ, AWS EventBridge
- Event Consumers (React to events)
- Event Storage (Log-based persistence for replaying events)



Challenges in Event-Driven Architecture

- Eventual Consistency (No immediate data synchronization)
- Ordering Guarantees (Ensuring events are processed in sequence)
- Fault Tolerance & Retries (Handling failures gracefully)
- Debugging Complexity (Tracing events across microservices)

Best Practices

- Use idempotent event processing to avoid duplicates
- Implement dead-letter queues for failed messages
- Choose the right event broker based on system needs
- Ensure event versioning to handle schema changes

Use Cases of Event-Driven Architecture

- Logging & Auditing (Track changes over time)
- Real-Time Notifications (Chat apps, stock price updates)
- Microservices Decoupling (Independent scalability & fault tolerance)
- IoT Systems (Sensor data processing)
- E-commerce Order Processing (Order placed → Payment processed → Inventory updated)

Important Interview Questions on Event-Driven Architecture

- Fundamentals
  - What is Event-Driven Architecture (EDA), and how does it differ from traditional request-response architectures?
  - Explain the difference between Pub-Sub and Event Streaming models.
  - What are the key components of an event-driven system?
- Scalability & Fault Tolerance
  - What are some challenges of Event-Driven Architecture, and how do you handle eventual consistency?
  - How can you ensure event ordering in distributed event processing?
  - What are dead-letter queues (DLQs), and why are they important?
- Implementation & Technologies
  - What are the differences between Kafka, RabbitMQ, and AWS EventBridge?
  - How do you make event processing idempotent to avoid duplicate execution?
- Use Cases & Real-World Applications
  - Can you give a real-world example where Event-Driven Architecture is a better choice than a traditional architecture?
  - What strategies can you use to handle schema evolution in an event-driven system?

Summary and Key takeaways

- Event-driven architecture is key to building scalable, real-time, and decoupled systems.
- Choosing between Pub-Sub vs. Event Streaming depends on system needs.
- Architectural patterns like CQRS & Event Sourcing enhance flexibility.
- Challenges exist but can be mitigated with best practices.
- What's next:
    - Summary - Architecture patterns

Section Summary - Architectural Patterns

- Architectural Patterns in System Design
  - Explored different architectural patterns and their trade-offs.
  - Multi-Tier Architecture: Layered approach for scalability and separation of concerns.
  - Microservices: Decoupled services enabling independent scaling & deployment.
  - Event-Driven Architecture: Asynchronous, scalable, and loosely coupled systems.
  - Learned real-world use cases, performance considerations, and best practices for choosing the right architecture.
- What's Next:
  - Web Concepts in System Design