Software architecture is the high-level structure of a software system. It defines the system's components and their interactions. Below is a differentiation of the key architectural styles:

a. Layered Architecture (also called N-tier Architecture)

- Structure: A system is divided into layers where each layer has a specific responsibility. Typical layers include the Presentation Layer (UI), Business Logic Layer, Data Access Layer, and Database.
- Advantages:
    - Simple to design and understand.
    - Supports separation of concerns, which makes the system more maintainable.
    - Helps in organizing and structuring code.
- Disadvantages:
    - Can lead to performance issues if the layers are tightly coupled.
    - Layers may become too rigid over time, limiting flexibility.

b. Client-Server Architecture

- Structure: The system is divided into two main components: the client (which initiates requests) and the server (which provides services or resources).
- Advantages:
    - Centralized data management (on the server) makes it easier to secure and maintain.
    - Scalable as clients can be added without affecting the server.
- Disadvantages:
    - Single point of failure (if the server goes down, all clients are affected).
    - Performance bottlenecks can occur if the server is overloaded with requests.

c. Event-Driven Architecture

- Structure: This architecture revolves around events. Components communicate through events, which are triggered and then processed by event handlers.
- Advantages:
    - Highly decoupled, making it more flexible and easier to extend.
    - Suitable for real-time applications and systems requiring asynchronous communication.
- Disadvantages:
    - Can be complex to design and debug.
    - Hard to maintain if event-driven logic grows complicated.

d. Microservices Architecture

- Structure: A system is divided into small, loosely coupled services, each responsible for a specific functionality (e.g., user management, inventory management, order processing). These services communicate via APIs.
- Advantages:
    - Independent scalability and fault isolation.
    - Flexibility in technology choices for each service.
    - Faster deployment and continuous delivery.
- Disadvantages:
    - Complex to manage and orchestrate multiple services.
    - Requires a robust system for inter-service communication and data consistency.

e. Repository Style

- Structure: A central repository manages shared data, and components interact with the repository to access or modify data.

- Advantages:
    - Centralized data management makes it easy to maintain consistency.
    - Easy to add new components that interact with the central repository.
- Disadvantages:
    - Single point of failure.
    - Can become inefficient if the repository grows too large or becomes too complex.

System design principles are essential for creating scalable, maintainable, and efficient software. Some common principles and how they apply to real-world applications are:

a. Separation of Concerns (SoC)

- **Real-world Application**: In an e-commerce application, the presentation layer (UI) is separate from the business logic (e.g., order processing) and data access (e.g., database management). This makes it easier to modify one layer without affecting others.

b. Modularity

- **Real-world Application**: In a social media application, separate modules handle different features: user profiles, posts, notifications, etc. Each module can evolve independently.

c. Scalability

- **Real-world Application**: In a cloud storage application (e.g., Dropbox), scalability is crucial. As users upload more files, the system must scale horizontally (adding more servers) to handle increased demand.

d. High Availability

- **Real-world Application**: Online banking systems require high availability, meaning that they should be up and running 24/7. This can be achieved through redundant servers and failover strategies.

e. Fault Tolerance

- **Real-world Application**: E-commerce platforms like Amazon need to be fault-tolerant to ensure that even if one service fails (e.g., payment processing), the rest of the system continues to function properly.

f. Performance Optimization

- **Real-world Application**: In a video streaming service (e.g., Netflix), the system should be optimized for fast video loading times, high-quality video playback, and low buffering, using techniques like caching and content delivery networks (CDNs).