1. **Definition of software architecture**
    a. **Provide at least two authoritative definitions (e.g. IEEE, Bass et al.) and compare them.**
    b. **Explain why software architecture is considered a shared understanding among stakeholders.**

1. IEEE (IEEE 1471-2000 / ISO/IEC/IEEE 42010:2011)

“The fundamental organization of a system, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.”

2. Bass, Clements, and Kazman (in Software Architecture in Practice)

“The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”

Comparison

- **Scope and Emphasis**
    - **IEEE** emphasizes fundamental organization and highlights not only internal system components but also their relationship to the environment and the principles that guide evolution. It views architecture as both structural and conceptual.
    - **Bass et al.** emphasize structures of software elements and focus on their externally *visible* properties (interfaces, behaviour visible to other components), making it more pragmatic for software design and analysis.
- **Evolution vs. Properties**
    - **IEEE** explicitly mentions design and evolution, recognizing that architecture changes over time.
    - **Bass et al.** stress externally visible properties to capture what’s important for reasoning about a system at a given point in time.
- **System vs. Software**
    - **IEEE** speaks in terms of systems (broader, can include hardware, environment, and context).
    - **Bass et al.** restrict their scope to software systems.
- **Usefulness**
    - **IEEE’s definition** is more general and abstract, suitable for aligning stakeholders, standards, and interdisciplinary systems.
    - **Bass et al.’s definition** is more practical for software engineers, focusing on how architecture helps reasoning about qualities such as performance, modifiability, and reliability.

2. **Architectural Concerns vs. Design Concerns**
    a. **Discuss the difference between architecture and detailed design.**
    b. **Give an example where architectural decisions strongly influence software evolution.**

Architectural design defines the overall structure of a software system. It identifies the main components or subsystems, their responsibilities, and the way they interact through interfaces, communication protocols, and data flows. The focus is on the high-level organization of the system, deciding what the main parts are and how they fit together. Architectural decisions are primarily driven by non-functional requirements such as scalability, performance, and maintainability, and the result is an architectural model showing the system as a set of cooperating components.

Detailed design, on the other hand, focuses on the internal structure and implementation details of each component identified in the architecture. It specifies algorithms, data structures, class hierarchies, database schemas, and interaction logic. The concern here is how each component fulfills its role as defined by the architecture.

Example Where Architectural Decisions Strongly Influence Software Evolution

Example: Choosing Monolithic vs. Microservices Architecture

At the start, suppose the team chooses a monolithic architecture for faster initial development. As the system grows, adding new features requires redeploying the entire monolith, slowing down releases. Scaling a single service (e.g., user management) independently is impossible — the whole application must be scaled. Technology upgrades are constrained: you can't adopt different programming languages for different parts easily. In contrast, if the team had chosen a microservices architecture, each service could evolve independently (different teams, release cycles). Scaling only the bottleneck service would be possible. However, it introduces complexity in distributed communication, monitoring, and deployment pipelines, making evolution harder if the team lacks DevOps maturity.

3. **Non-Functional requirements (Quality attributes)**
    a. **Select two quality attributes (e.g. performance, scalability, security) and describe how they influence architectural choices.**
    b. **Provide a short example scenario for each.**

a. Two Quality Attributes and Their Influence on Architecture

1. Performance

Influence on architecture:

Performance refers to the ability of the system to respond to user actions quickly and efficiently, even under high load. Architectural decisions that enhance performance include adopting modular and scalable service-oriented architectures, using caching layers, load balancing, and asynchronous communication between components. Reducing communication overhead and optimizing data access paths are key strategies to minimize latency and improve response times.

Example scenario:

In the Smart Food Delivery Platform, during peak meal hours, many customers may place orders simultaneously. To ensure quick order processing and smooth tracking, the architecture can include a dedicated Order & Dispatch Service supported by in-memory caching and asynchronous message queues, so that each component can process tasks independently without blocking others. This ensures that users experience minimal waiting time.

---

2. Security

Influence on architecture:
Security affects how components are designed, connected, and managed. Architectural decisions must ensure data protection, authentication, authorization, and secure communication. This can involve adopting a layered architecture that separates sensitive operations, encrypting data in transit and at rest, and using API gateways or security services for centralized access control and monitoring.

Example scenario:
In the *Smart Food Delivery Platform*, users provide payment information and personal data. To protect this, the architecture includes a Payment & Security Service that handles authentication and payment processing separately from other modules. Data exchanged between clients and services are encrypted, and access to user data is restricted based on roles (customer, restaurant, rider, or admin). This ensures confidentiality and prevents unauthorized access.

Part B- Practical exercise (Smart Food Delivery Platform)

A startup wants to develop a smart food delivery platform that connects customers, restaurants, and delivery riders.

- Customers can place orders and track deliveries
- Restaurants manage menus and confirm orders.
- Riders receive delivery requests and update delivery status.
- The system must be scalable, secure and user-friendly.

1. Architectural Drivers (Key Non-Functional Requirements)

1. Performance

The platform must provide fast and responsive interactions for all users (customers, restaurants, and delivery riders). During peak hours, many concurrent orders and updates occur simultaneously; hence, the system must process requests efficiently and deliver real-time status updates. Poor performance would lead to delays, user frustration, and reduced trust in the platform.

2. Security

The system handles sensitive data such as user credentials, payment information, and delivery addresses. Therefore, strong security mechanisms are essential to protect data confidentiality and integrity. A security breach would directly damage the company's reputation and could result in financial loss or legal issues.

3. Usability

Usability is critical for system adoption and users' retention over the time. Each actor (customer, restaurant, and riders) must be able to perform their tasks easily and quickly. High usability ensures that even non-technical users can interact effectively with the platform, reducing errors and increasing satisfaction.

2. Component Identification (First-level Decomposition)

Customer App / Web Interface – Allows customers to browse menus, place orders, and track deliveries.

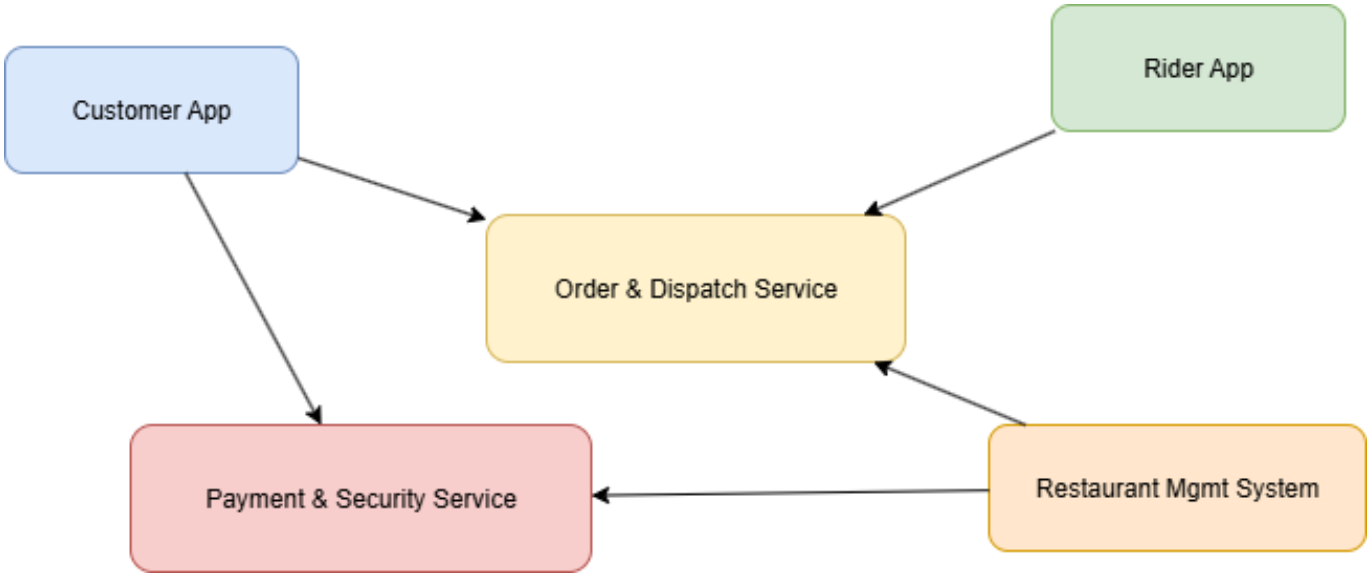
Restaurant Management System – Handles menu updates, order confirmations, and preparation status.

Rider App – Provides delivery requests, navigation, and real-time status updates.

Order & Dispatch Service – Core backend logic that matches orders to restaurants and riders, manages delivery assignments, and tracks order states.

Payment & Security Service – Handles authentication, authorization, and secure payment processing.

3. Representation : Draw a simple box-and-line diagram showing the main components and their interactions.



4. Discussion (How the Architecture Addresses the Non-Functional Requirements)

Performance is achieved through a **modular, service-oriented architecture** where major functions (such as *Order & Dispatch*, *Restaurant Management*, and *Payment & Security*) are implemented as independent services. This allows each service to be scaled horizontally according to workload, ensuring fast response times even during peak hours. The use of **asynchronous communication**, **caching**, and a **Single Page Application (SPA)** frontend further reduces latency and improves user experience by minimizing page reloads and providing smooth interaction.

Security is addressed through the clear separation of responsibilities and controlled access to sensitive operations. The dedicated Payment & Security Service manages authentication, authorization, and payment processing, ensuring that confidential data such as user credentials and payment details remain isolated. All communication between clients and services must be protected using **encryption**, and **role-based access control** ensures that each actor (customer, restaurant, or rider) can only access authorized data and operations.

Usability is a core focus since the platform involves three distinct user groups with different goals. The architecture provides **dedicated dashboards** for customers, restaurants, and riders, each exposing only the relevant features. The **SPA frontend** contributes to a fast and intuitive interface, while consistent design patterns and responsive layouts ensure accessibility across devices. This improves user satisfaction, reduces cognitive load, and encourages system adoption.