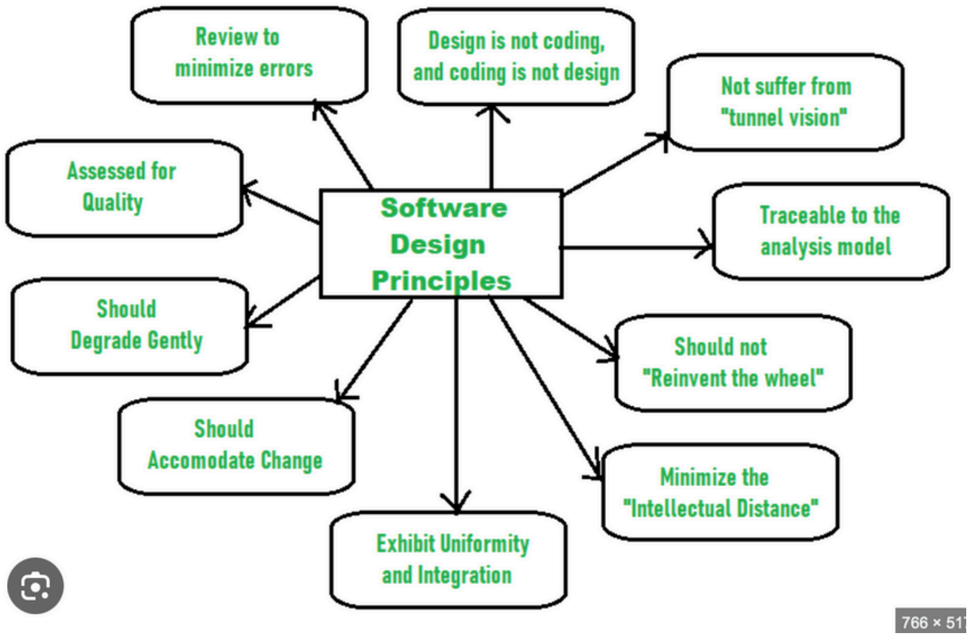
Unit-5

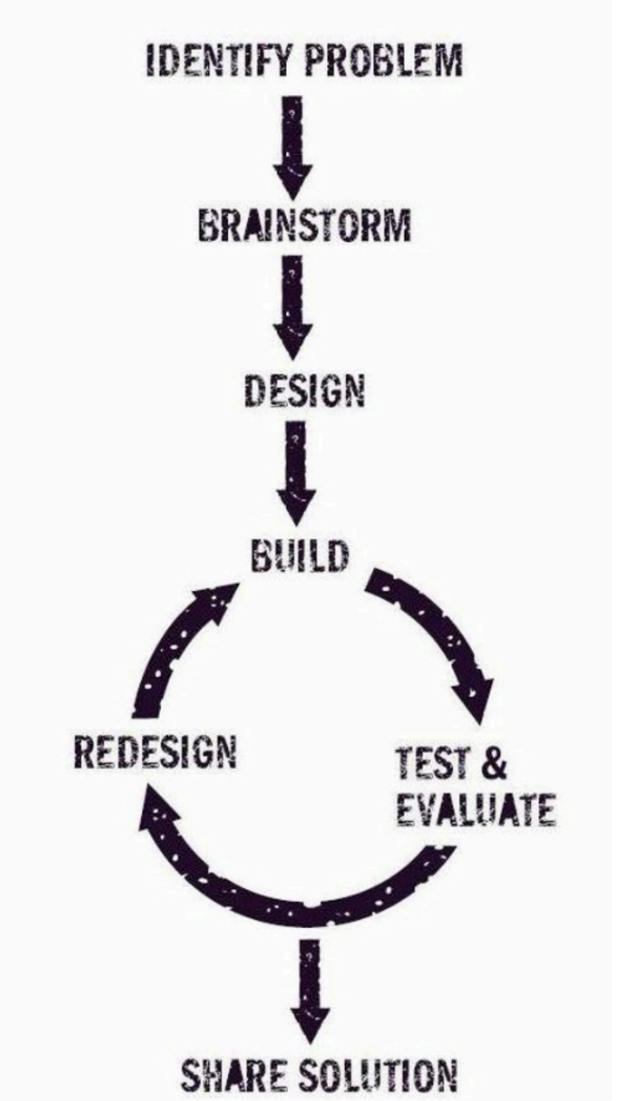
Software Design

Software Design

Software design is a critical phase in the software development lifecycle where the architecture, structure, and behavior of a software system are conceptualized and defined. It involves transforming user requirements and specifications into a blueprint that guides the implementation and construction of the software product.



- Software design aims to translate user requirements into a technical solution that fulfills the desired functionality, performance, and quality criteria.
- It establishes the foundation for the development process by defining the overall structure, components, and interactions of the software system.



Key principles of Software Design

- **Abstraction:** Simplifying complex systems by focusing on essential aspects and hiding unnecessary details.
- **Modularity:** Breaking down the software system into discrete, self-contained modules that can be developed, tested, and maintained independently.
- **Encapsulation:** Bundling data and operations into cohesive units, known as objects, to enforce data hiding and protect the integrity of the system.
- **Hierarchy:** Organizing modules and components into hierarchical structures to facilitate understanding, scalability, and maintainability.
- **Decomposition:** Breaking down the system into smaller, manageable units to analyze, design, and implement effectively.

Phases of Software Design

- **Architectural Design:** Establishing the high-level structure and components of the software system, including the distribution of functionality and communication between subsystems.
- **Detailed Design:** Elaborating on the architectural design by defining the internal structure, interfaces, and algorithms of individual components and modules.
- **User Interface Design:** Designing the graphical user interface (GUI) or user experience (UX) to facilitate interaction between users and the software system.
- **Database Design:** Defining the structure, relationships, and constraints of the underlying data storage and retrieval mechanisms.

Considerations in Software Design

- **Requirements Analysis:** Understanding and refining user requirements to ensure alignment with the design solution.
- **Quality Attributes:** Addressing non-functional requirements such as performance, scalability, reliability, and security in the design process.
- **Trade-offs:** Balancing conflicting objectives and constraints, such as time-to-market versus robustness or flexibility versus simplicity.
- **Iterative Development:** Embracing an iterative and incremental approach to software design to accommodate evolving requirements and feedback.

Role of Software Design in Development Process

- Software design serves as a bridge between requirements specification and implementation, providing a detailed blueprint for developers to follow.
- It enables stakeholders to visualize and validate the proposed solution before investing resources in implementation.
- Effective software design reduces the risk of errors, rework, and cost overruns during the development lifecycle.

Characterstics of Good Software Design

Clarity and Understandability:

- Clear and easy to understand, facilitating comprehension of structure, components, and functionality.

Modularity and Encapsulation:

- Broken down into cohesive, independent modules with well-defined interfaces, promoting reusability, maintainability, and scalability.
- Utilizes encapsulation to hide internal details, reducing dependencies and facilitating changes without impacting other parts of the system.

Abstraction and Simplification:

- Utilizes abstraction to simplify complex systems by focusing on essential aspects and hiding unnecessary details, aiding in comprehension and maintenance.

High Cohesion and Low Coupling:

- Demonstrates high cohesion with modules organized around single, well-defined purposes, promoting code clarity, reusability, and maintainability.
- Exhibits low coupling, minimizing dependencies between modules, allowing changes to be made independently and reducing the risk of cascading failures.

Scalability and Flexibility:

- Designed to be scalable, accommodating changes in scope, requirements, and workload without extensive rework or compromising performance.
- Offers flexibility for customization and adaptation to different environments, user needs, and evolving business requirements.

Performance and Efficiency:

- Prioritizes performance and efficiency, employing appropriate algorithms, data structures, and optimization techniques to minimize resource consumption and maximize throughput.

Robustness and Reliability:

- Demonstrates robustness, handling unexpected inputs, errors, and exceptions gracefully with appropriate error handling and recovery mechanisms to prevent system failures and data loss.
- Reliable, delivering consistent and predictable behavior under normal and adverse conditions, with rigorous testing and validation to verify correctness and stability.

Maintainability and Extensibility:

- Designed for maintainability with clean, well-structured code, clear documentation, and adherence to coding standards, facilitating ease of modification, debugging, and troubleshooting.
- Supports extensibility, allowing for the addition of new features, functionalities, and integrations with minimal impact on existing code, promoting long-term sustainability and evolution of the software.

Usability and User Experience:

- Prioritizes usability and user experience with intuitive interfaces, logical workflows, and responsive feedback, ensuring users can interact with the software effectively and efficiently.

Adherence to Standards and Best Practices:

- Follows established standards, guidelines, and best practices in software engineering, including design patterns, architectural principles, and coding conventions, promoting consistency, interoperability, and maintainability across projects and teams.

Design Principles

DRY (Don't Repeat Yourself):

- Avoid duplication of code or logic by abstracting common functionality into reusable components or functions. This principle promotes code reusability, maintainability, and reduces the risk of inconsistencies.

SOLID Principles:

- **S** - **Single Responsibility Principle (SRP):** Each part of your code should do one thing well. A class or module should have only one reason to change, focusing on a single responsibility or concern.
- **O** - **Open/Closed Principle (OCP):** Your code should be easy to extend without changing its existing parts. Software entities (classes, modules, functions) should be open for extension but closed for modification, allowing new functionality to be added without altering existing code.
- **L** - **Liskov Substitution Principle (LSP):** Any part of your code should be replaceable with a subtype without breaking the code. Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.
- **I** - **Interface Segregation Principle (ISP):** Make small, specific interfaces instead of one big interface. Clients should not be forced to depend on interfaces they do not use, promoting smaller, cohesive interfaces tailored to specific client needs.
- **D** - **Dependency Inversion Principle (DIP):** Depend on abstractions, not on concrete implementations. High-level modules should not depend on low-level modules but rather on abstractions, promoting decoupling and flexibility.

Design Principles

KISS (Keep It Simple, Stupid):

- Keep things simple! Don't make them more complicated than they need to be.
- Favor simplicity over complexity in design and implementation. Strive for the simplest solution that meets requirements without unnecessary embellishment, reducing cognitive load and potential for errors.

YAGNI (You Aren't Gonna Need It):

- Only add features when you need them. Don't overcomplicate things with stuff you might not use.
- Avoid adding functionality until it's actually needed. This principle discourages speculative or premature optimization, promoting simplicity, flexibility, and responsiveness to changing requirements.

Separation of Concerns:

- Split your code into parts, each dealing with one thing. This makes it easier to understand and change later.
- Divide the software system into distinct modules or components, each responsible for a single aspect or concern. This principle promotes modularity, maintainability, and flexibility by isolating changes and minimizing dependencies.

Design Principles

Single Responsibility Principle (SRP):

- Each part of your code should be responsible for doing just one thing.
- A class or module should have only one reason to change, encapsulating a single responsibility or functionality. This principle enhances cohesion, reduces coupling, and facilitates easier maintenance and testing.

Composition over Inheritance:

- Prefer building complex things by putting together simple things, rather than making everything based on what's already there.
- Favor composition (building complex objects by combining simpler ones) over inheritance (creating specialized classes by extending existing ones) to achieve greater flexibility, reusability, and maintainability.

Loose Coupling and High Cohesion:

- Keep things separate, but connected. Don't let one part of your code depend too much on another.
- Aim for loose coupling between modules or components, minimizing dependencies to reduce the impact of changes and promote modifiability and testability.
- Strive for high cohesion within modules, ensuring that elements within a module are closely related and contribute to a single, well-defined purpose or functionality.

Design Principles

Encapsulation:

- Hide the details! Keep the inner workings of your code hidden away, and only show what's necessary.
- Encapsulate data and functionality within modules or classes, exposing only necessary interfaces while hiding implementation details. This principle enhances modularity, information hiding, and maintainability.

Consistency:

- Keep everything looking and working the same way. It helps make your code easier to understand and work with.
- Maintain consistency in naming conventions, coding style, and design patterns throughout the software system. Consistency promotes readability, reduces cognitive overhead, and facilitates collaboration among team members.

Design Concepts

Design concepts in software engineering encompass fundamental ideas and approaches used to create effective and efficient software solutions.

Abstraction

- Abstraction involves simplifying complex systems by focusing on essential aspects while hiding unnecessary details. It helps manage complexity and promotes clarity in design.

Encapsulation

- Encapsulation involves bundling data and methods into a single unit, known as a class in object-oriented programming, and restricting access to certain parts of the object. It enhances data security, promotes code reusability, and facilitates maintenance.

Modularity

- Modularity involves breaking down a software system into smaller, independent modules or components, each responsible for a specific functionality or feature. It promotes reusability, maintainability, and scalability by isolating changes and minimizing dependencies.

Inheritance

- Inheritance is a mechanism in object-oriented programming where a new class (subclass) inherits properties and behaviors from an existing class (superclass). It enables code reuse, promotes extensibility, and facilitates polymorphism.

Design Concepts

Polymorphism

- Polymorphism allows objects to be treated as instances of their parent class, enabling flexibility and extensibility in design. It allows for the same interface to be used for different data types or objects, facilitating code reuse and flexibility.

Composition

- Composition involves building complex objects by combining simpler objects or components. It promotes flexibility, reusability, and maintainability by allowing objects to be composed of other objects, rather than relying solely on inheritance.

Decomposition

- Decomposition involves breaking down a problem or system into smaller, more manageable parts. It aids in understanding complex systems, facilitates collaboration, and promotes modularity and scalability in design.

Design Concepts

Concurrency

- Concurrency involves the execution of multiple tasks or processes simultaneously. It enables efficient resource utilization, responsiveness, and scalability in software systems, particularly in multi-threaded or distributed environments.

Patterns

- Design patterns are reusable solutions to common problems encountered in software design. They provide templates and guidelines for structuring code, solving recurring design challenges, and promoting best practices.

Architectural Styles

- Architectural styles define the overall structure and organization of software systems. Examples include layered architecture, client-server architecture, and microservices architecture, each with its own benefits and trade-offs in terms of scalability, maintainability, and performance.

Design Strategies

Design strategies in software engineering outline the overarching approach or methodology used to conceptualize, plan, and implement software solutions.

Iterative Design

- Iterative design involves incremental development and refinement of software solutions through repeated cycles of prototyping, testing, and feedback gathering. It enables flexibility, adaptability, and continuous improvement throughout the development lifecycle.

User-Centered Design (UCD)

- User-centered design focuses on understanding user needs, preferences, and behaviors to create software solutions that are intuitive, usable, and satisfying. It involves involving users in the design process, conducting user research, and iterating based on feedback.

Agile Design

- Agile design emphasizes collaboration, adaptability, and customer satisfaction by embracing iterative development, cross-functional teams, and frequent delivery of working software. It prioritizes individuals and interactions over processes and tools and responds to change over following a plan.

Design Strategies

Design Thinking

- Design thinking is a human-centered approach to innovation that combines empathy, creativity, and rationality to solve complex problems and generate innovative solutions. It involves empathizing with users, defining problem statements, ideating potential solutions, prototyping, and testing iteratively.

Component-Based Design

- Component-based design involves building software systems by assembling pre-defined, reusable components or modules. It promotes modularity, reusability, and maintainability by abstracting common functionality into self-contained units.

Domain-Driven Design (DDD)

- Domain-driven design focuses on modeling software solutions based on real-world business domains and concepts. It emphasizes collaboration between domain experts and software developers, strategic design patterns, and ubiquitous language to ensure alignment between the software model and the problem domain.

Design Strategies

Service-Oriented Architecture (SOA)

- Service-oriented architecture is an architectural approach that structures software systems as a collection of loosely coupled, interoperable services. It promotes flexibility, scalability, and reusability by encapsulating business functionality into independent, self-contained services.

Model-Driven Design

- Model-driven design involves creating abstract models of software systems using modeling languages such as UML (Unified Modeling Language) or DSLs (Domain-Specific Languages). It facilitates communication, visualization, and analysis of system designs and requirements.

Test-Driven Design (TDD)

- Test-driven design advocates writing automated tests before implementing the actual software functionality. It promotes clarity, correctness, and maintainability by focusing on defining testable requirements and driving development through iterative test-case creation.

Parallel Design

- Parallel design involves dividing the development process into parallel streams, each focusing on a specific aspect of the software solution (e.g., user interface design, backend development, testing). It accelerates development, promotes specialization, and mitigates project risks.

Design Approaches

Top-Down Design Approach

- In top-down design, the system is designed from a high-level perspective, starting with an overview of the entire system and then breaking it down into smaller, more detailed components or modules.
- The process begins with defining the system's overall structure, functionality, and objectives, often represented using techniques like flowcharts, data flow diagrams, or pseudocode.
- Once the high-level architecture is established, each component is further refined and decomposed into more detailed sub-components or functions, continuing until the lowest level of granularity is reached.
- Top-down design emphasizes abstraction, modularity, and abstraction, allowing developers to focus on system-level concerns before delving into implementation details.
- Advantages of top-down design include a clear understanding of the system's architecture, early identification of key components, and alignment with user requirements and business objectives.
- However, it may lead to oversimplification or overlooking of lower-level details, potential difficulty in integrating components, and the need for frequent revisions as implementation progresses.

Design Approaches

Bottom-Up Design Approach

- In bottom-up design, the system is built by first developing individual components or modules and then integrating them into larger subsystems and the overall system.
- The process starts with identifying and implementing smaller, more atomic components that perform specific functions or tasks, often referred to as building blocks.
- These components are then gradually combined and integrated into larger, more complex modules, subsystems, and eventually the complete system.
- Bottom-up design emphasizes reusability, flexibility, and incremental development, allowing developers to focus on building and refining individual components iteratively.
- Advantages of bottom-up design include early validation of component functionality, better utilization of existing resources and libraries, and flexibility in adapting to changing requirements.
- However, it may result in a lack of a cohesive overall architecture, potential difficulties in ensuring consistency and compatibility between components, and challenges in addressing system-level concerns.

Design Process and Design Quality

Steps in the software design process

a. **Requirements analysis:** Understand the problem domain and the functional and non-functional requirements.
b. **Architectural design:** Define the overall system structure, including the major components and their interactions.
c. **Detailed design:** Specify the internal design of individual components, including data structures, algorithms, and interfaces.
d. **Design review and evaluation:** Assess the design for correctness, efficiency, maintainability, and other quality attributes.

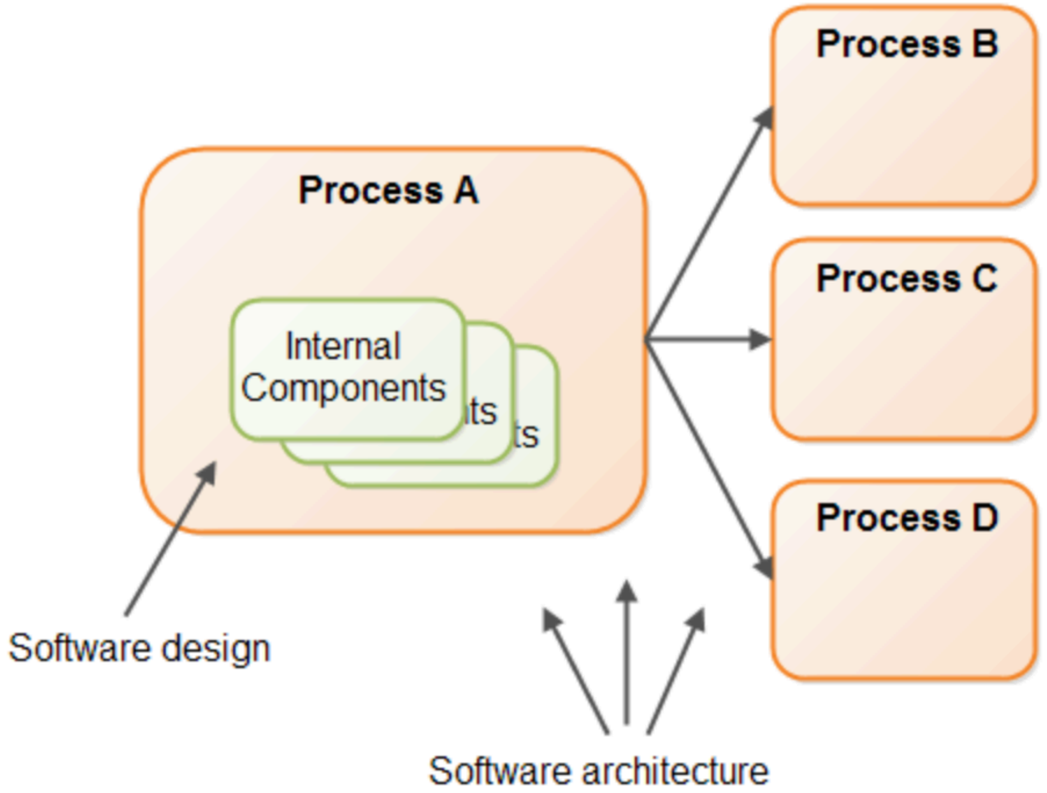
Design Process and Design Quality

Metrics for evaluating design quality

a. **Complexity:** Measures the degree of complicatedness or intricacy in the design, often using metrics like cyclomatic complexity.
b. **Coupling and Cohesion:** Evaluates the interdependence between modules (coupling) and the relatedness of elements within a module (cohesion).
c. **Maintainability:** Assesses the ease with which the design can be understood, modified, and extended in the future.
d. **Reusability:** Measures the extent to which design elements can be reused in other contexts or applications.

Software Architecture

Software architecture refers to the high-level structure of a software system, encompassing its components, relationships, and principles governing its design and evolution. It serves as a blueprint for organizing and integrating software components to meet functional and non-functional requirements effectively.

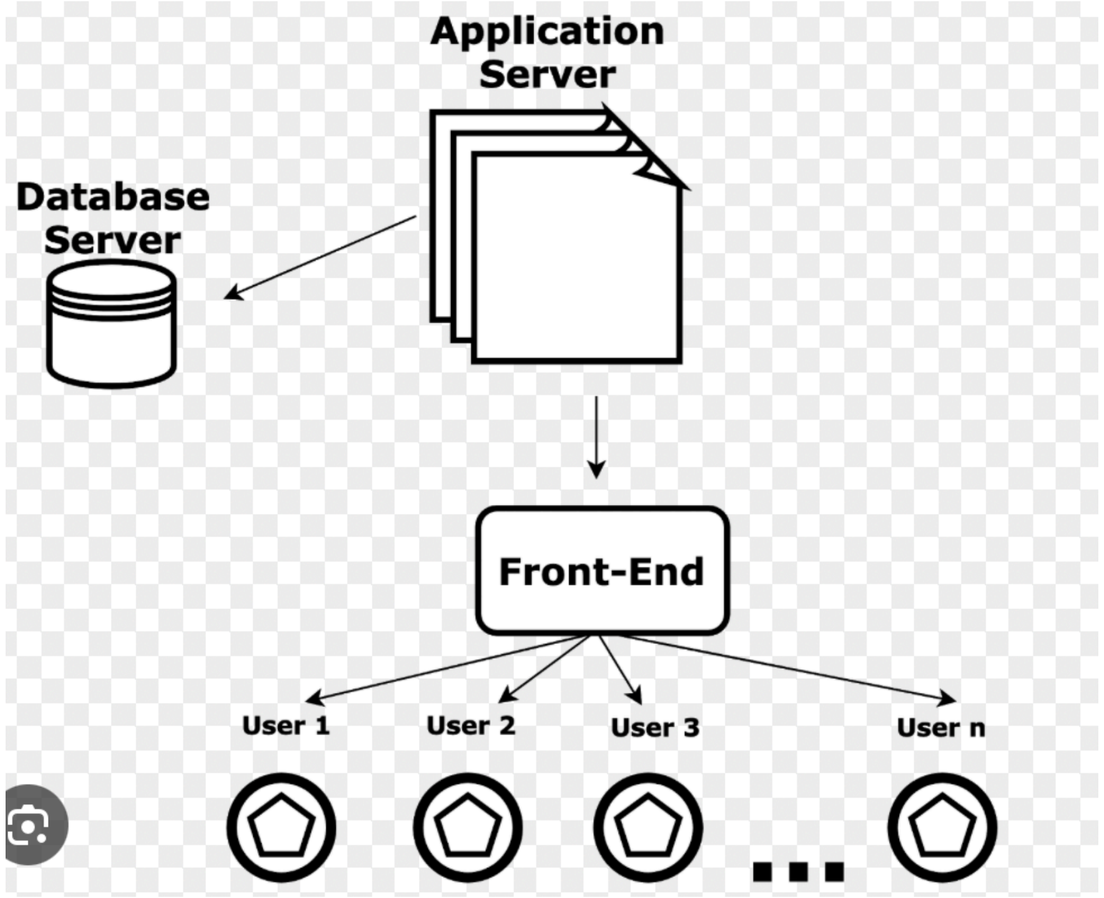


Importance of Software Architecture

1. **Guidance:** Provides a roadmap for the development team, guiding decision-making and ensuring consistency throughout the development lifecycle.
2. **Scalability:** Facilitates the growth and evolution of the software system by accommodating changes in requirements, functionality, and technology.
3. **Maintainability:** Enhances the ease of maintenance and updates by promoting modularity, encapsulation, and separation of concerns.
4. **Performance:** Optimizes system performance and resource utilization by defining efficient component interactions and architectural patterns.
5. **Quality:** Enhances software quality attributes such as reliability, security, and usability by enforcing architectural principles and design patterns.
6. **Risk Management:** Identifies and mitigates architectural risks early in the development process, reducing the likelihood of costly rework and project delays.

Client-Server/ Data centered Architecture

Client-server architecture is a two-tier architecture model that divides an application into two distinct roles: client and server. The client is responsible for presenting the user interface and interacting with users, while the server is responsible for managing and providing access to resources, data, and services.



Client

- The client is typically a user-facing application or interface that communicates with the server to request and receive data or services.
- It can be a desktop application, web browser, mobile app, or any other software component that interacts with the user.
- Clients are responsible for initiating communication with the server, sending requests, and processing responses to present information to users.

Server

- The server is a centralized system or software component that manages and provides resources, data, or services to clients.
- It hosts and executes the business logic, processes client requests, and performs data processing, storage, and retrieval.
- Servers can be physical machines, virtual servers, or cloud-based instances, depending on the scalability and performance requirements of the application.

Communication

- Communication between clients and servers typically follows a request-response model.
- Clients send requests to the server, specifying the desired action or data.
- Servers process the requests, perform necessary operations, and return responses containing the requested information or status updates.
- Communication between clients and servers is facilitated by various network protocols such as HTTP, HTTPS, TCP/IP, and WebSocket.
- These protocols define the rules and formats for transmitting data over the network, ensuring reliable and secure communication between clients and servers.

Advantages

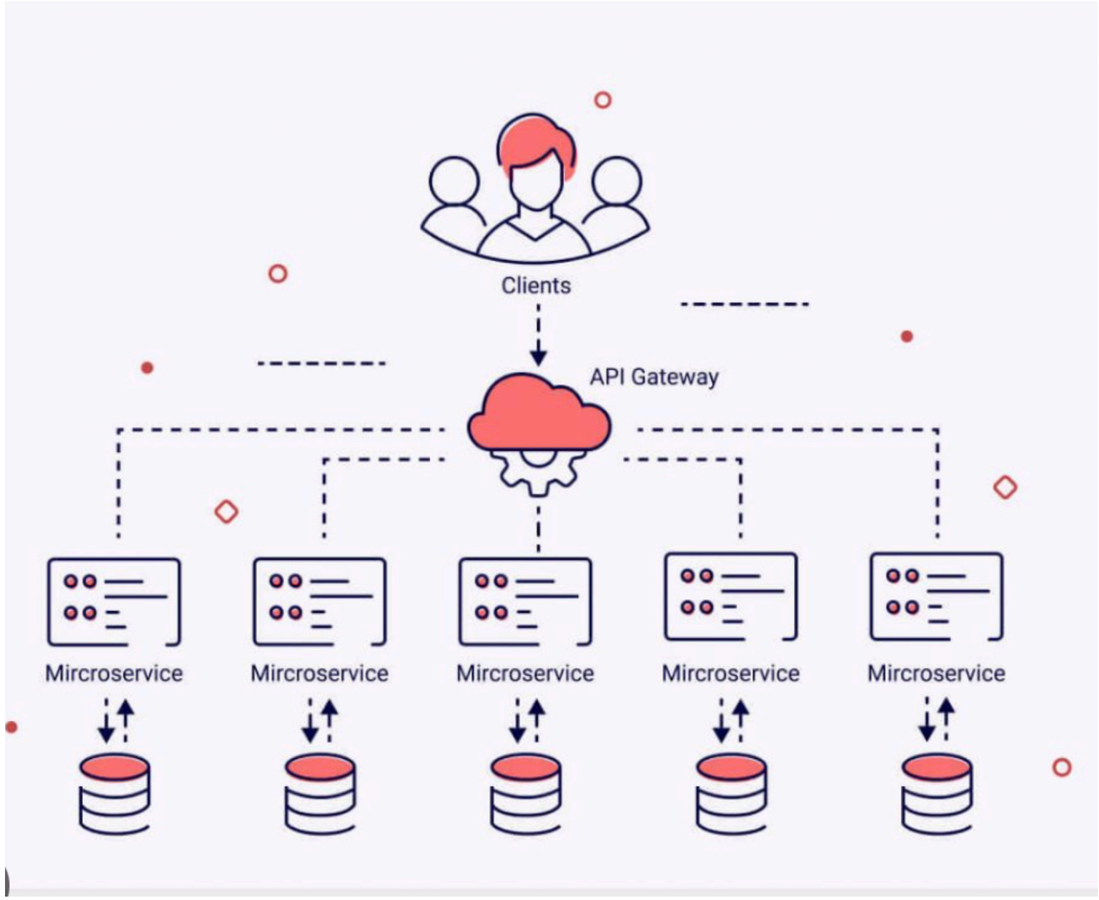
- Client-server architecture enables horizontal scalability by distributing client requests across multiple servers, allowing applications to handle increasing user loads efficiently.
- Centralized servers facilitate centralized management and administration of resources, data, and services, promoting consistency, security, and ease of maintenance.
- Client-server architecture allows clients and servers to be developed and deployed independently, enabling flexibility in technology choices, platform support, and deployment options.
- Centralized servers can implement security measures such as authentication, authorization, and encryption to protect sensitive data and resources from unauthorized access or tampering.

Examples

- **Web Applications:**
  - Web applications follow a client-server architecture, with web browsers acting as clients and web servers hosting and serving web pages and resources.
  - Clients send HTTP requests to web servers, which process the requests, execute server-side logic, and return HTML, CSS, and JavaScript files to render in the browser.
- **Database Systems:**
  - Database systems use client-server architecture, with database clients (such as database management systems or applications) interacting with database servers to perform data operations.
  - Clients send SQL queries or commands to database servers, which execute the queries, retrieve or manipulate data, and return results to clients.
- **Email Servers:**
  - Email servers use client-server architecture, with email clients (such as email clients or webmail interfaces) communicating with email servers to send, receive, and manage email messages.
  - Clients connect to email servers using protocols like SMTP, IMAP, or POP3 to send and retrieve email messages stored on the server.

Microservices Architecture

Microservices architecture is an architectural style that structures an application as a collection of loosely coupled, independently deployable services, each representing a specific business function or capability. Unlike monolithic architectures, where the entire application is developed and deployed as a single unit, microservices architecture decomposes the application into smaller, manageable services that can be developed, deployed, and scaled independently.



Key characteristics

- **Service Decomposition**
  - The application is decomposed into multiple small services, each responsible for a specific business function or feature.
  - Services are designed to be cohesive, autonomous, and independently deployable, with clear boundaries and responsibilities.
- **Loose Coupling**
  - Services are loosely coupled, meaning they interact with each other through well-defined APIs or communication protocols.
  - Loose coupling enables services to evolve, scale, and be replaced or updated independently without impacting other services.

Key characteristics

- **Independent Deployment**
  - Microservices can be developed, deployed, and updated independently of each other, allowing for continuous delivery and deployment practices.
  - Each service can be deployed using its own technology stack, programming language, or infrastructure, depending on its specific requirements.
- **Scalability**
  - Microservices architecture enables horizontal scalability by allowing individual services to be scaled independently based on demand.
  - Services can be deployed across multiple instances or containers, and load balancers can distribute traffic to maintain performance and availability.

Key characteristics

- **Resilience and Fault Isolation**
  - Microservices are designed for resilience, with failure isolation mechanisms that prevent failures in one service from cascading to others.
  - Services can implement circuit breakers, retries, timeouts, and fallback mechanisms to handle and recover from failures gracefully.
- **Polyglot Persistence**
  - Microservices allow for polyglot persistence, where each service can use the most appropriate data storage technology for its specific requirements.
  - Services can use different databases or data storage solutions, such as relational databases, NoSQL databases, or in-memory caches, depending on their data access patterns and performance needs.

Key characteristics

- **Event-Driven Communication**
  - Microservices architecture often relies on asynchronous, event-driven communication patterns to enable loose coupling and scalability.
  - Services can publish events or messages to message brokers or event streams, allowing other services to react to changes or events in real-time.

Examples

- **Netflix**
  - Netflix uses a microservices architecture to build and operate its streaming platform, with services responsible for functions such as user authentication, content recommendation, and video playback.
  - Microservices enable Netflix to scale its platform globally, deliver personalized experiences to users, and innovate rapidly in response to changing market dynamics.
- **Uber**
  - Uber's ride-sharing platform is built using microservices architecture, with services handling functions such as user authentication, ride allocation, payment processing, and driver management.
  - Microservices enable Uber to scale its platform to support millions of users worldwide, deliver real-time experiences, and experiment with new features and services.