High-Level Design (HLD) - Key Concepts Overview

1. What is High-Level Design (HLD)?

High-Level Design (HLD) is a stage in the software design process that outlines the system architecture and major components. It provides a macro-level view of the system, showing how various modules and components interact with each other. HLD focuses on the overall architecture, data flow, integration points, and non-functional aspects like scalability, reliability, and security. It serves as a blueprint for developers and stakeholders, bridging the gap between requirements and Low-Level Design (LLD).

2. Objectives of HLD

- Define the system architecture and design approach.
- Identify key components and their responsibilities.
- Establish communication protocols and data flow.
- Make decisions on technology stack and deployment.
- Address scalability, availability, and performance needs.
- Serve as a foundation for LLD and implementation.

3. Key Components of HLD

- System Architecture Diagram: Visual representation of modules and their interactions.
- Component Responsibilities: Describes the role of each major component.
- Communication: Sync vs Async (REST, Kafka, etc.).
- Database Design: ER diagrams, SQL vs NoSQL.
- Deployment: Cloud, on-premises, containers.
- NFRs: Non-Functional Requirements like scalability, reliability, etc.
- Security: Auth, data protection, encryption.
- Trade-offs: CAP theorem, consistency vs availability, etc.

4. Scalability

Scalability refers to a system's ability to handle increased load. It can be achieved by:

- Vertical Scaling: Upgrading server hardware.
- Horizontal Scaling: Adding more servers or nodes.

- Load Balancing: Distributes incoming traffic across servers.
- Caching: Reduces load on databases and APIs.
- Asynchronous Processing: Offloading non-critical tasks to queues.

5. Reliability and Fault Tolerance

A reliable system consistently performs its intended function. Fault tolerance ensures it continues to operate in the event of failure.

Strategies include:

- Redundancy: Multiple instances of critical services.
- Replication: Data duplication across regions.
- Retries and Circuit Breakers: For transient failures.
- Health Checks and Auto-restart: Ensures recovery from crashes.

6. Synchronous vs Asynchronous Communication

- Synchronous: Blocking calls (e.g., REST APIs) where response is immediate. Suitable for real-time needs like login.
- Asynchronous: Non-blocking (e.g., Kafka, RabbitMQ) where messages are processed later. Suitable for background tasks, decoupling.
- Use both strategically based on system requirements.

7. SQL vs NoSQL Databases

- SQL: Relational (e.g., MySQL, PostgreSQL). Enforces schema and supports ACID transactions. Ideal for structured data and complex queries.
- NoSQL: Flexible schema (e.g., MongoDB, Cassandra). Supports scalability and high availability. Ideal for unstructured data, real-time apps.
- Decision based on use case: transactional vs flexible data, scalability needs, consistency requirements.

8. Non-Functional Requirements (NFRs)

These are quality attributes that determine how a system performs:

- Scalability: Can grow with user load.

High-Level Design (HLD) - Key Concepts Overview

- Availability: Uptime and minimal downtime.
- Performance: Fast response times.
- Security: Auth, encryption, data privacy.
- Maintainability: Ease of updates and bug fixes.
- Observability: Logging, monitoring, alerting.

9. Deployment and Infrastructure

- Use of cloud platforms (AWS, GCP, Azure) for scalability and flexibility.
- Containerization using Docker and orchestration with Kubernetes.
- CI/CD pipelines for automation.
- Multi-region deployment for high availability and latency reduction.

10. Design Patterns and Best Practices

- Singleton, Factory, Strategy for object creation and behavior.
- Circuit Breaker, Retry for resilience.
- API Gateway, Service Mesh for communication and control.
- Event-driven architecture for loose coupling.