[System Design]

* Why need System Design?

→ Before actually coding the system first step is design

→ For creating scalable, efficient, reliable softwares:

1. Scalability

- **Need:** Systems must handle increasing users, data, or transactions as they grow.
- **Design Approach:** Incorporating load balancers, database sharding, caching, and horizontal scaling ensures scalability.

2. Reliability

- **Need:** Users expect systems to function correctly and consistently.
- **Design Approach:** Designing for fault tolerance, redundancy, and failover mechanisms ensures minimal downtime.

3. Performance Optimization

- **Need:** Fast response times are critical for user experience and system efficiency.
- **Design Approach:** Properly designing APIs, optimizing database queries, and using caching mechanisms improve performance.

4. Cost Efficiency

- **Need:** Resources must be used wisely to minimize operational costs.
- **Design Approach:** Efficient resource allocation, cloud-based solutions, and microservices can lower costs.

5. Security

- **Need:** Protecting sensitive data and ensuring system integrity is vital.
- **Design Approach:** Designing authentication, encryption, and secure data flows ensures robust security.

6. Maintainability

- **Need:** Systems need to be easy to update and debug over time.
- **Design Approach:** Modular, loosely coupled components with clear documentation make maintenance simpler.

7. Meeting Business Needs

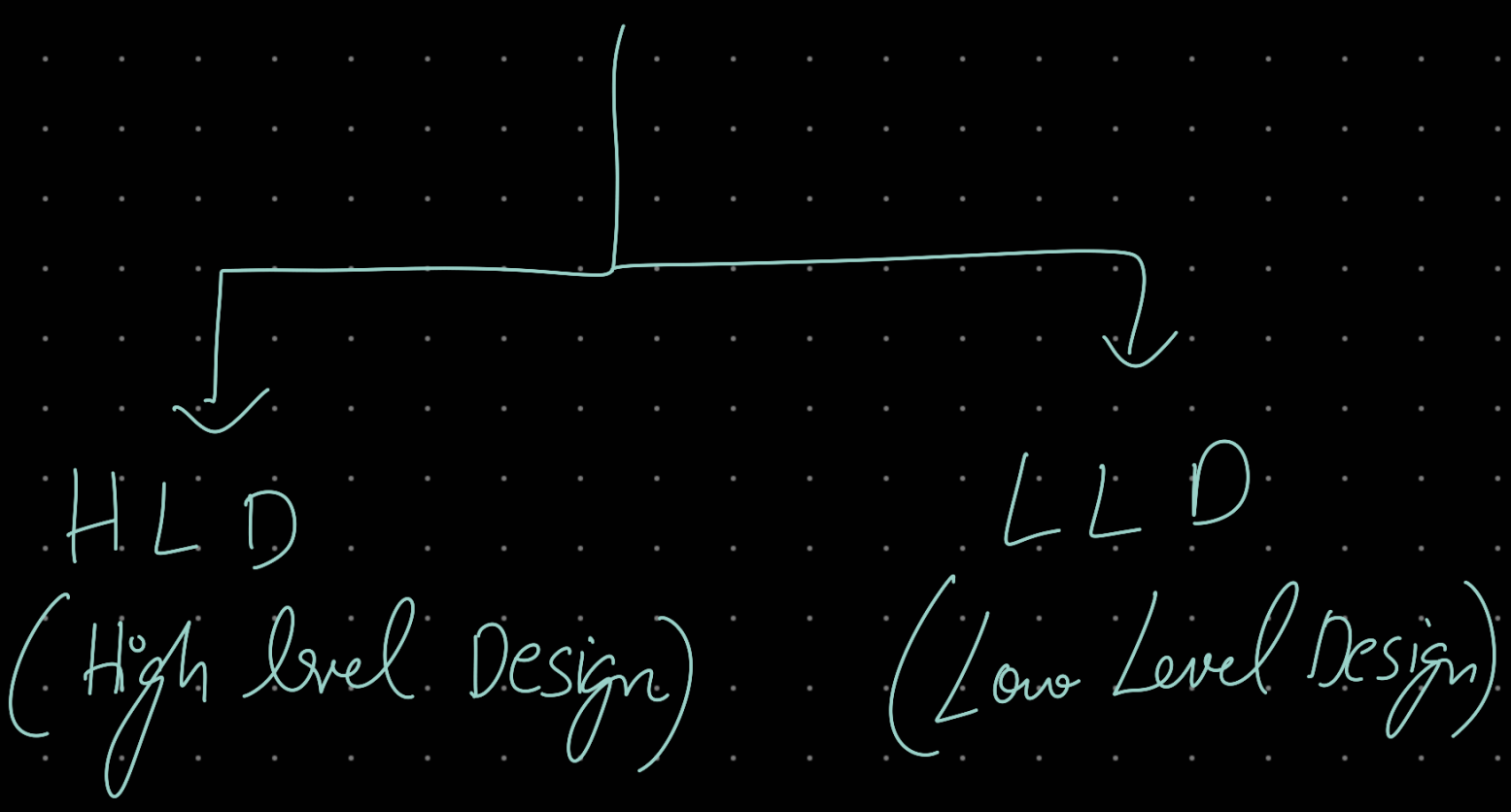
- **Need:** Systems must align with organizational goals and user expectations.
- **Design Approach:** Translating business requirements into technical solutions ensures relevance and success.

8. Future-Proofing

- **Need:** Technology and user needs evolve; systems must adapt without total overhauls.
- **Design Approach:** Flexible architectures, such as microservices or event-driven models, prepare systems for the future.

9. Collaboration

- **Need:** Teams must have a shared understanding of the system.
- **Design Approach:** Clear diagrams and well-documented designs improve cross-functional collaboration.



1. High-Level Design (HLD)

HLD provides a broad view of the system architecture.

Key Features:

- Focuses on the **overall architecture** and interaction between system components.
- Defines the **modules, components, and their relationships**.
- Identifies **data flow** and **high-level algorithms**.
- Describes the **technology stack** and **platforms** used.
- Helps stakeholders understand the **system's big picture**.

Includes:

- **Architecture diagrams**: E.g., client-server model, microservices, or monolithic structure.
- **Component relationships**: How components like databases, APIs, and services interact.
- **Functional modules**: Logical grouping of features (e.g., user management, payment processing).
- **Non-functional requirements**: Scalability, reliability, performance, and security considerations.

Audience:

- Architects, product managers, and technical leads.

2. Low-Level Design (LLD)

LLD dives into the details of how individual components are implemented.

Key Features:

- Provides a **detailed view** of the system's components and their behavior.
- Focuses on the **implementation details** and logic.
- Specifies **class diagrams**, **method definitions**, **database schemas**, and **pseudo-code**.
- Aims to guide developers on how to build individual parts.

Includes:

- **Detailed class diagrams**: Including attributes and methods for each class.
- **Database design**: Tables, relationships, and data constraints.
- **API details**: Endpoint definitions, request/response formats, and error handling.
- **Detailed algorithms**: Pseudo-code or flowcharts for specific functionalities.

Audience:

- Developers and engineers who will implement the system.