Palash Bajpai

**sYSTEM dESIGN**

**HLD**

Table of Contents

[**1. What is system design?** 2](#_Toc165803653)

[1. System Design requirements 3](#_Toc165803654)

[**2. Monolithic vs Micro service architecture** 4](#_Toc165803655)

[1. Monolithic architecture 4](#_Toc165803656)

[2. Microservice architecture 5](#_Toc165803657)

[**3. System Design Trade-offs** 6](#_Toc165803658)

[1. Latency 6](#_Toc165803659)

<https://roadmap.sh/system-design>

<https://roadmap.sh/software-design-architecture>

<https://media.geeksforgeeks.org/courses/syllabus/b16669e7619419215993fe653e0b2fee.pdf>

https://www.geeksforgeeks.org/complete-roadmap-to-learn-system-design/

# **1. What is system design?**

A system is a set of interconnected components working together to achieve a common goal, while system design is the process of defining and implementing the architecture, components, and interfaces of a system to fulfill specific requirements.

1. High Level Design (HLD): describe system architecture details, database design, service and processes, relationship between different modules and features.
2. Low Level Design (LLD): describe design of each component in detail, classes, interfaces, relationship between them, and various actual logic of different components.

Let’s see a real world example of planning a birthday party and how designing fits in here:

1. **Inviting Guests:**

In software development, inviting guests to a birthday party is similar to inviting users to use your software application. You might send out invitations (marketing emails or social media posts) to let people know about your app and encourage them to try it out.

1. **Choosing a Theme:**

This is like deciding on the overall design and aesthetic of your software application. You might choose a theme (such as minimalist, playful, or professional) that reflects the personality and purpose of your app.

1. **Buying Supplies:**

Buying supplies for a birthday party is akin to acquiring resources and tools for software development. You might purchase software licenses, development kits, or other resources needed to build and test your application.

1. **Setting Up Decorations:**

Setting up decorations for the party is similar to designing the user interface (UI) of your software application. You want to create an attractive and visually appealing interface that enhances the user experience and makes navigation intuitive.

1. **Preparing Food and Drinks:**

This is like implementing the functionality of your software application. You need to develop features and functionalities that fulfill user needs and provide value, whether it's processing transactions, managing data, or facilitating communication.

1. **Entertaining Guests:**

Entertaining guests at a birthday party is akin to providing support and engagement for users of your software application. You want to ensure that users have a positive experience with your app and that their needs are met through timely support, updates, and feedback channels.

Though basic building blocks of all systems are same, namely servers, databases, cache, application, message queue etc. All systems are combined in different ways to serve different users. Let’s see one example how same components help to create different applications and learn about system design in technical aspects.

E-commerce Platform vs. Social Media Platform:

**E-commerce Platform:**

1. In an e-commerce platform like Amazon, servers host the website and handle user requests, while databases store product information, user profiles, and order history.
2. Caches are used to store frequently accessed data such as product listings or user session information, improving performance and reducing latency.
3. Applications manage the business logic, including product search, shopping cart functionality, and payment processing.
4. Message queues are employed for order processing, inventory management, and notifications, ensuring reliability and scalability of backend processes.

**Social Media Platform:**

1. In a social media platform like Facebook, servers host the website and handle user interactions, while databases store user profiles, posts, comments, and media content.
2. Caches are utilized for storing frequently accessed content such as profile information, news feed updates, and image thumbnails, improving responsiveness and user experience.
3. Applications manage complex features such as news feed algorithms, content recommendation systems, and real-time notifications, leveraging machine learning and data analytics.
4. Message queues facilitate real-time communication between users, handling instant messaging, comments, likes, and notifications, ensuring timely delivery and scalability of messaging features.

So here design is important, to use same components but serve different purposes.

## 1. System Design requirements

Categorizing requirements into functional and non-functional types is a common and effective approach. Here's how you can classify different types of requirements:

1. **Non-Functional Requirements:**

* These specify the quality attributes, constraints, and characteristics that the system must possess, beyond its core functionality.
* Non-functional requirements address aspects such as performance, reliability, security, usability, scalability, maintainability, and compatibility.
* Examples include response time targets, availability goals, security controls, usability guidelines, scalability limits, maintainability standards, and compatibility with external systems or standards.

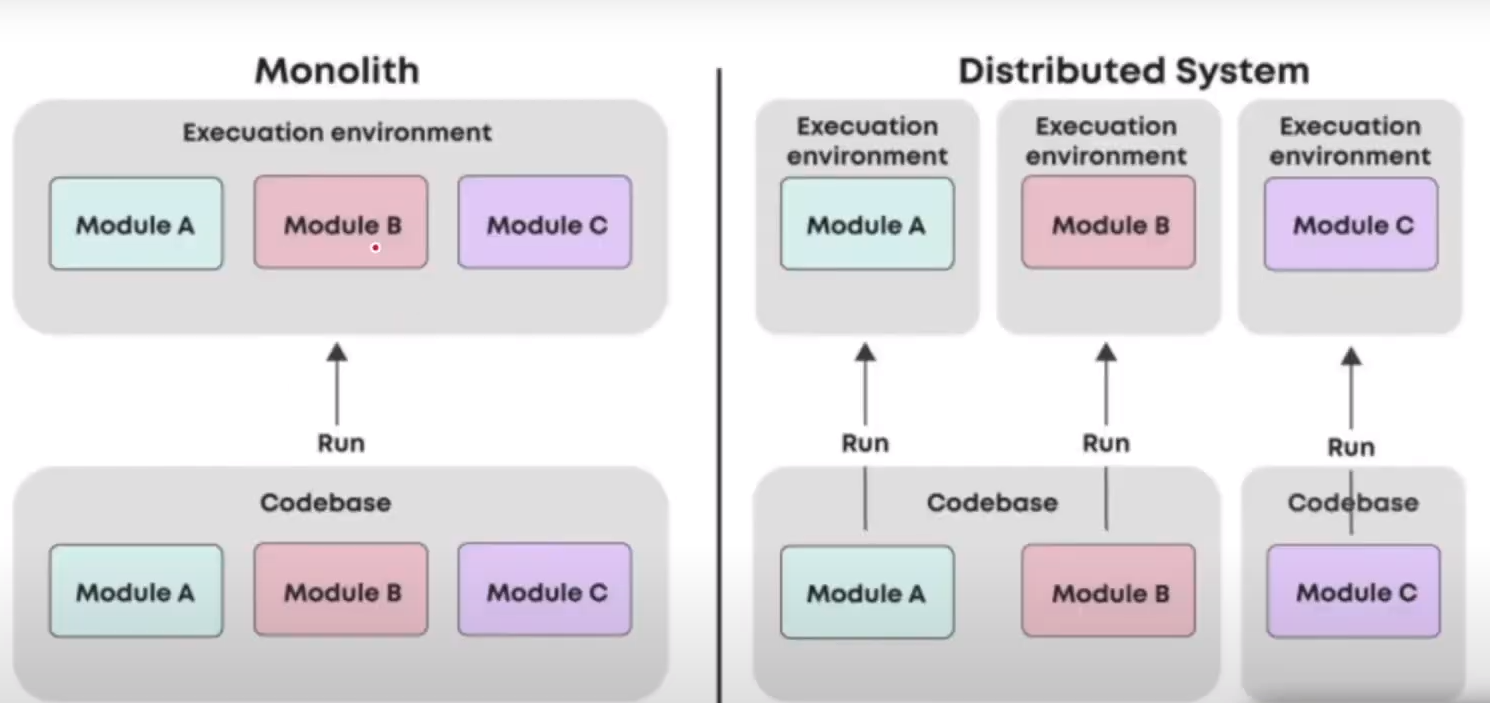
1. **Functional Requirements:**

* These describe what the system should do and define specific functions, features, and behaviors that the system must exhibit.
* Examples include user authentication, data validation, report generation, transaction processing, and user interface interactions.
* Functional requirements are typically expressed as user stories, use cases, or specific system functionalities.

# **2. Monolithic vs Micro service architecture**

Web Application = frontend + backend + database

1. Monolithic Architecture: Entire application is built as single, self-contained unit all of its components (UI, db, business logic) are tightly integrated and deployed together.
2. Microservice Architecture: Application is decomposed into smaller, independently deployable services, each responsible for specific business function or feature. Each microservice runs as specific process and communicates with each other through well-defined API’s.



## 1. Monolithic architecture

Advantages:-

1. Monolithic architecture is suitable for small-scale projects with straightforward requirements and limited scalability needs.
2. It's effective for applications with low complexity and where rapid development and deployment are prioritized.
3. Monolithic architecture simplifies deployment and management by consolidating all components into a single codebase and deployment unit.
4. Monolithic architecture may be preferable for applications with stable and predictable workloads, where the overhead of microservices is unnecessary.

Disadvantages:-

1. Monolithic systems are hard to scale, as horizontal scaling of individual components is not possible here.
2. In monolithic architecture, every module is combined in single system, so if there is an error or bug in single module, it can destroy the complete system.
3. Hard to update as updating a monolithic application often requires modifying the entire codebase, leading to higher risk and longer development cycles.
4. Monolithic architectures can hinder team autonomy and independence. Since all components are tightly coupled within the same codebase, different teams may need to coordinate closely when making changes, leading to slower development cycles and increased dependencies.

## 2. Microservice architecture

Advantages:-

1. Scalability: Microservices enable efficient, independent scaling of services to handle varying workloads.
2. Agility: Independent development and deployment foster faster iteration and time-to-market.
3. Resilience: Fault isolation ensures that failures in one service don't disrupt the entire system.
4. Technology Diversity: Allows for the use of diverse technologies and frameworks within the architecture.
5. Ease of Deployment: Individual service deployment simplifies updates, reduces downtime, and enables continuous delivery.

Disadvantages:-

1. Complexity: Managing a large number of services increases complexity in development, deployment, and monitoring.
2. Distributed System Challenges: Communication overhead, network latency, and data consistency issues can arise in distributed systems.
3. Operational Overhead: Requires robust infrastructure and DevOps practices for managing and coordinating services.
4. Increased Latency: Inter-service communication can introduce latency compared to monolithic architectures.
5. Testing Complexity: Testing becomes more complex due to dependencies between services and the need for comprehensive integration testing.

# **3. Non-functional Requirements**

## 1. Availability

Availability in system design refers to the ability of a system to remain operational and accessible to users, typically measured as a percentage of time that the system is functional and responsive.

Measured in percent, 99.9%(3 nines) means it is available for 99.9% time.

Monolithic system have single point of failure, and hence are less available.

Ways to improve Availability:

1. Utilize load balancers to distribute traffic across multiple servers, ensuring high availability.
2. Deploy redundant hardware components such as servers, network devices, and storage systems to mitigate single points of failure.
3. Implement automated monitoring and alerting systems to quickly detect and respond to system failures or performance issues.
4. Utilize cloud-based infrastructure and services with built-in redundancy and high availability features.
5. Design disaster recovery plans and procedures to minimize downtime and ensure business continuity in the event of catastrophic failures or disasters.
6. Create redundant resources as a backup for components that may fail.

## 2. Consistency

Consistency in system design refers to ensuring that all copies of data across different components or nodes in a distributed system are synchronized and up-to-date

Types of Consistency:

1. Strong Consistency: System does not allow read operation until all nodes with replicated data are updated.
2. Eventual Consistency: Some users might receive old data, but eventually all data gets updated to latest data, read request re not halted.
3. Weak Consistency: Here it is not necessary to have same data in all nodes.

## 3. Partition tolerance

Partition tolerance in system design refers to a system's ability to continue functioning and providing services even if network partitions occur, causing communication failures between different components or nodes of the system.

* In distributed systems, network partitions can occur due to various reasons such as network failures, hardware failures, or geographical distribution of components. When a partition occurs, the network is divided into separate segments, and communication between nodes in different segments becomes unavailable or unreliable.
* Achieving partition tolerance is crucial for ensuring the resilience and fault tolerance of distributed systems, particularly in environments where network connectivity cannot be guaranteed at all times. Partition-tolerant systems are designed to handle network partitions gracefully and continue operating without compromising data consistency or availability.

To achieve partition tolerance in a system:

1. Design the system to operate even if parts of it can't communicate with each other due to network failures or partitions.
2. Implement distributed algorithms and protocols that allow nodes to continue functioning independently during network partitions.
3. Use redundancy and replication techniques to ensure that copies of data are available across multiple partitions, enabling continued access to data even in the face of network disruptions.
4. Design the system to gracefully handle inconsistencies or conflicts that may arise due to partitioned operations, such as using conflict resolution mechanisms or eventual consistency models.
5. Test the system thoroughly under various network failure scenarios to ensure that it remains available and operational during partitioned conditions.

## 6. Security

Security in system design involves implementing protective measures to safeguard against unauthorized access, data breaches, and malicious attacks.

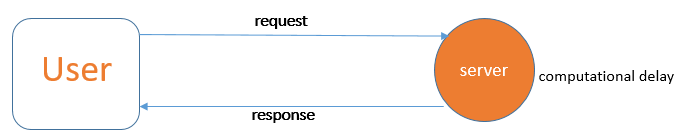
Different ways to achieve security in a system include:

1. **Encryption**: Protecting data by converting it into a coded format that can only be accessed with a decryption key.
2. **Access Controls**: Restricting access to resources based on user roles, permissions, and authentication mechanisms.
3. **Firewalls**: Filtering network traffic to block unauthorized access and prevent malicious activity.
4. **Intrusion Detection Systems (IDS):** Monitoring network or system activities for signs of unauthorized access or malicious behavior.
5. **Vulnerability Management**: Regularly scanning systems for security vulnerabilities and applying patches and updates to address them.
6. **Authentication Mechanisms:** Verifying the identity of users or systems to prevent unauthorized access.
7. **Security Policies and Procedures:** Establishing rules and guidelines for secure system usage, configuration, and maintenance.
8. **Physical Security Measures:** Securing physical access to servers, data centers, and other critical infrastructure.
9. **Secure Coding Practices:** Following best practices to write secure code and minimize vulnerabilities in software applications.
10. **Security Awareness Training:** Educating users and employees about security risks and best practices to prevent security breaches.

Few other non-

## 4. Latency

It refers to time it takes for data to travel from its source to its destination.



Latency = computational delay + network delay (b/w user and system + within components of system)

For monolithic system internal calls are lesser and hence latency is less.

Ways to reduce latency:

1. Optimize network infrastructure and reduce data transfer distances to minimize latency.
2. Use caching to store frequently accessed data closer to users, reducing retrieval time and latency.
3. Implement parallel processing and asynchronous operations to reduce waiting time and improve latency.
4. Employ content delivery networks (CDNs) to serve content from edge servers, reducing round-trip time and latency.
5. Utilize compression techniques to reduce data size and transmission time, improving overall latency.
6. Optimize algorithms and data structures to streamline processing and reduce computational overhead, enhancing latency.

## 5. Throughput

In system design, throughput refers to the rate at which a system can process and handle a certain volume of work within a given period of time.

It is a measure of the system's capacity to effectively handle incoming requests, transactions, or operations.

Throughput is typically expressed in terms of the number of units of work completed per unit of time (e.g., transactions per second, requests per minute). It reflects the system's ability to efficiently utilize its resources, such as CPU, memory, disk I/O, and network bandwidth, to process incoming workloads.

Ways to improve throughput:

1. Optimize resource utilization to reduce processing bottlenecks and improve throughput.
2. Implement concurrency mechanisms to handle multiple tasks simultaneously and enhance throughput.
3. Use caching, indexing, and compression techniques to minimize overhead and improve throughput.
4. Distribute workloads evenly across multiple servers using load balancing to maintain consistent throughput.
5. Design efficient software architectures to minimize conflict and optimize resource usage for improved throughput.
6. Improve network performance by optimizing configurations and reducing latency to enhance overall system throughput.

Architectural design patterns are a higher-level concept compared to low-level design (LLD) patterns. While **LLD design patterns** (like Singleton, Factory, Builder, etc.) focus on object creation, structuring, and behavior within a module, **architectural design patterns** focus on the overall structure of an application.

### **Architectural Design Patterns (HLD Focus)**

Some important **architectural patterns** include:

* **MVC (Model-View-Controller)** – Used in web and desktop applications.
* **MVVM (Model-View-ViewModel)** – Common in UI frameworks like React and Android.
* **Microservices** – For distributed, scalable applications.
* **Layered Architecture** – Common in enterprise apps (Controller → Service → Repository).
* **Event-Driven Architecture** – Used in real-time systems and message-based applications.

### **Where Do They Fit?**

Since you are focusing on **low-level design**, architectural patterns might not be the main priority here. However, if you want to **extend** the scope to cover HLD as well, I can create detailed content on **architectural patterns** too.

Would you like me to include architectural patterns along with LLD, or do you want a separate course/document for them?