**System Design :**

* Systems Design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements
* It involves high-level planning of how different parts of a system (or multiple systems) will interact to achieve the desired functionality and performance

**Key Aspects of System Design:**

1. **High-Level Design (HLD):**
   * Focuses on the overall architecture and structure of the system.
   * Defines components like databases, servers, APIs, and how they interact.
   * Identifies key external systems or third-party integrations.
   * Covers aspects like scalability, performance, and security.
2. **Low-Level Design (LLD):**
   * Focuses on individual modules and components within the system.
   * Involves designing detailed workflows, class diagrams, and function-level implementation details.
   * Specifies data structures, algorithms, and the logic inside the components.

**Types of System Design:**

1. **Monolithic Architecture:**
   * The system is designed as a single, unified unit.
   * Easier to implement initially but harder to scale and maintain as the system grows.
2. **Microservices Architecture:**
   * The system is divided into small, independent services that communicate via APIs.
   * Each service can be scaled, deployed, and maintained independently.
3. **Distributed Systems:**
   * System components run on different machines, potentially across different geographic regions.
   * Requires consideration of network communication, consistency, and fault tolerance.

**Common Design Considerations:**

* **Scalability:** How the system handles increasing loads (horizontal vs. vertical scaling).
* **Availability:** Ensuring the system remains operational even when parts of it fail (redundancy, failover).
* **Consistency:** Maintaining data accuracy across different parts of the system.
* **Latency and Performance:** Minimizing response times for end users.
* **Security:** Protecting sensitive data and ensuring secure communications.
* **Fault Tolerance:** Ability to recover from hardware or software failures.

**Choosing the right database ?**

Choosing the right database in system design is a crucial decision that can significantly impact the performance, scalability, and maintainability of your application

**Data Type and Structure:**

* **Relational Databases (RDBMS):** Best suited for structured data with well-defined relationships, such as customer information, orders, and inventory. Examples: MySQL, PostgreSQL, Oracle.
* **NoSQL Databases:** Ideal for unstructured or semi-structured data, such as JSON, XML, or key-value pairs. Examples: MongoDB, Cassandra, Redis.

**Scalability:**

* **Horizontal Scalability:** The ability to add more nodes to a database cluster to handle increased load. NoSQL databases are generally better at horizontal scaling.
* **Vertical Scalability:** The ability to increase the resources (CPU, memory) of a single database node to handle increased load. RDBMS can sometimes be scaled vertically, but it's often more limited.

**Performance:**

* **Query Performance:** Consider the types of queries your application will perform and how well the database can handle them. RDBMS are often better for complex joins and aggregations, while NoSQL databases can be faster for simple read/write operations.
* **Data Consistency:** RDBMS are generally stronger in terms of data consistency (ACID properties), while NoSQL databases often prioritize availability (BASE properties).

ACID Properties

**1. Atomicity**

* **Definition:** Ensures that all operations within a transaction are treated as a single "atomic" unit. Either all operations in the transaction are successfully executed, or none of them are applied. If any part of the transaction fails, the entire transaction is rolled back.
* **Example:** Imagine you're transferring $100 from Account A to Account B. The transaction involves two operations:
  + Deduct $100 from Account A.
  + Add $100 to Account B.

If the deduction from Account A succeeds but the addition to Account B fails (due to a system crash, for example), Atomicity ensures that the entire transaction is rolled back, leaving both accounts unchanged.

**2. Consistency**

* **Definition:** Ensures that a transaction brings the database from one valid state to another valid state, maintaining all predefined rules, such as integrity constraints (e.g., primary key, foreign key constraints).
* **Example:** Suppose there's a rule that the balance in an account cannot be negative. If you're transferring money between accounts, the consistency property ensures that, after the transaction, both accounts respect this rule (i.e., none of the accounts has a negative balance).

**3. Isolation**

* **Definition:** Ensures that multiple transactions occurring concurrently do not interfere with each other. Transactions are isolated from each other until they are completed.
* **Example:** Assume two users are trying to withdraw money from the same bank account at the same time. One transaction deducts $100, and the other deducts $50. Without isolation, the final balance could be incorrect. Isolation ensures that one transaction is processed fully before the other starts, so the final balance remains accurate.

**4. Durability**

* **Definition:** Ensures that once a transaction has been committed, it will remain so, even in the event of a system failure. The results of the transaction are permanently stored in the database.
* **Example:** After successfully transferring $100 from Account A to Account B, even if the system crashes immediately after the transaction is committed, the transaction will not be lost. When the system is restored, the changes (i.e., the updated balances in both accounts) will still be in place.