System-Design Interview: top 25 topics-

1) ***Scalability - How will your system handle growth? Vertical or Horizontal scaling?***
   - ➢ Ideally, your system should be designed for horizontal scaling, allowing you to add more machines or instances to your infrastructure as the demand grows. Vertical scaling, increasing the resources on a single machine, has its limitations and can become expensive and impractical. Kubernetes for container orchestration allows for easy scaling of containerized applications across multiple nodes in a cluster.
2) ***Latency and Throughput - Essential performance measures.***
   - ➢ Latency refers to the time it takes for a single request to be processed, while throughput measures the number of requests that can be handled within a given time frame. Both are crucial performance metrics that need to be optimized depending on the specific requirements of your system. Prometheus and Grafana are monitoring tools used to measure and optimize latency and throughput metrics in real-time.
3) ***Database design - SQL or NoSQL? Understand their strengths and weaknesses.***
   - ➢ SQL databases are relational and are great for structured data with complex queries and transactions. NoSQL databases offer flexibility and scalability for unstructured or semi-structured data. Choose based on your data structure and requirements. PostgreSQL for SQL databases and MongoDB for NoSQL databases. PostgreSQL offers robust support for complex queries and transactions, while MongoDB provides scalability and flexibility for unstructured data.
4) ***Indexing - Understand database indexes and their performance implications.***
   - ➢ Indexes help improve the speed of data retrieval operations by providing quick access to specific rows in a table. However, excessive indexing can slow down write operations and increase storage requirements. PostgreSQL's B-tree and GIN indexes improve query performance, while in NoSQL databases like MongoDB, compound indexes can be utilized.
5) ***Load Balancing - Understand different algorithms and their use cases.***
   - ➢ Load balancing distributes incoming network traffic across multiple servers to ensure no single server is overwhelmed. Algorithms like Round Robin, Least Connection, and IP Hash are commonly used for different use cases. NGINX and HAProxy are popular load balancers that distribute incoming traffic across multiple backend servers using various algorithms like Round Robin or Least Connections.
6) ***Caching - Caching can dramatically increase the speed of data retrieval. Know different strategies, benefits and trade-offs.***
   - ➢ Caching involves storing frequently accessed data in memory to reduce the need for repeated queries to the database. Strategies include in-memory caching, CDN caching, and database query caching, each with its benefits

and trade-offs. Redis and Memcached are widely used in-memory caching solutions that dramatically increase the speed of data retrieval by storing frequently accessed data in memory.

7) ***Networking Basics - TCP/IP, HTTP vs HTTPS, UDP, DNS lookups.***
    - ➢ TCP/IP is the fundamental protocol suite for internet communication. HTTP is for transmitting web pages, while HTTPS adds a layer of encryption. UDP is a lightweight protocol for fast data transmission, and DNS is used for translating domain names to IP addresses. Tools like Wireshark for packet analysis and dig or nslookup for DNS lookups help understand TCP/IP communication and resolve domain names to IP addresses.

8) ***Microservices Architecture - Learn how to decompose a large system into manageable parts.***
    - ➢ Decompose large applications into smaller, independent services that can be developed, deployed, and scaled independently. This architecture promotes flexibility, scalability, and resilience. Frameworks like Spring Boot (Java), Flask (Python), and Express.js (JavaScript) facilitate the development of microservices-based applications by providing tools for building and deploying individual services.

9) ***Data Sharding - As your data grows, how will you distribute it? Understand different sharding techniques and their trade-offs.***
    - ➢ Distribute data across multiple servers to improve scalability and performance. Techniques include range-based, hash-based, and key-based sharding, each with its own trade-offs. MongoDB supports sharding out of the box, and tools like Vitess can be used for sharding in MySQL databases.

10) ***CAP Theorem - Consistency, Availability, and Partition Tolerance - you can only pick two.***
    - ➢ States that a distributed system cannot simultaneously guarantee consistency, availability, and partition tolerance. In a distributed system, you can only prioritize two out of the three. Distributed databases like Apache Cassandra prioritize partition tolerance and availability over strong consistency, making them suitable for scenarios where high availability is critical.

11) ***System Interfaces - API design and best practices.***
    - ➢ Design APIs that are intuitive, consistent, and well-documented. Follow RESTful principles for web APIs, and consider factors like versioning, authentication, and error handling. Swagger and OpenAPI Specification are tools for designing and documenting RESTful APIs, ensuring consistency and ease of integration for consumers.

12) ***Redundancy and Failover - Learn how to design a highly available system.***
    - ➢ Design systems with redundancy to ensure high availability. Implement failover mechanisms to automatically switch to backup systems in case of failures. Amazon RDS (Relational Database Service) and Amazon Route 53

(DNS service) provide built-in redundancy and failover capabilities for databases and domain name resolution, respectively.

13) ***Message Queues - Asynchronous communication, ensuring reliable data transfer.***
   - ➢ Enable asynchronous communication between components of a system, ensuring reliable and scalable data transfer. Message queues like RabbitMQ, Kafka, and SQS are commonly used for this purpose. Apache Kafka and RabbitMQ are popular message queuing systems that support asynchronous communication between components of a system, ensuring reliable data transfer.

14) ***Data Replication - Storing duplicate data across multiple machines to increase reliability and availability.***
   - ➢ Replicate data across multiple nodes to increase reliability and availability. Techniques include master-slave replication, master-master replication, and sharding. MySQL's built-in replication feature allows you to replicate data across multiple MySQL instances for increased reliability and availability.

15) ***Concurrency - Handling multiple requests simultaneously.***
   - ➢ Handle multiple requests or tasks simultaneously to improve system responsiveness and resource utilization. Use techniques like threading, multiprocessing, and asynchronous programming. Python's asyncio library and Java's CompletableFuture class enable asynchronous programming, allowing you to handle multiple concurrent tasks efficiently.

16) ***Multithreading - The art of handling multiple threads and synchronization.***
   - ➢ Manage multiple threads efficiently, ensuring proper synchronization to avoid race conditions and deadlock situations. Java's java.util.concurrent package provides classes like ThreadPoolExecutor and Semaphore for managing multiple threads and synchronization.

17) ***Rate Limiting - Controlling request rates, why, and how.***
   - ➢ Control the rate of incoming requests to prevent overloading your system. Implement rate-limiting algorithms based on tokens, quotas, or sliding windows. Tools like Redis or NGINX can be used to implement rate-limiting mechanisms based on tokens or request quotas.

18) ***Consensus Algorithms - Get familiar with Raft, Paxos, and more.***
   - ➢ Ensure consistency and fault tolerance in distributed systems. Raft and Paxos are widely used consensus algorithms for achieving distributed consensus. etcd, a distributed key-value store, uses the Raft consensus algorithm to ensure consistency and fault tolerance in distributed systems.

19) ***Idempotency - Repeated requests should yield the same result.***
   - ➢ Design operations in such a way that repeating the same request multiple times yields the same result, ensuring reliability and consistency. APIs can be designed to support idempotent operations using UUIDs or request timestamps, ensuring that repeating the same request multiple times yields the same result.

20) ***REST vs RPC - When and why to use them.***

- ➢ REST is an architectural style for designing networked applications, while RPC (Remote Procedure Call) is a protocol for communication between distributed systems. Choose based on the requirements of your system. REST APIs are widely used for web applications, while gRPC is gaining popularity for inter-service communication in microservices architectures.

21) ***Polling vs Push mechanism - Differences and their appropriate usage.***
- ➢ Polling involves repeatedly checking for updates, while the push mechanism sends updates automatically when they occur. Choose based on factors like real-time requirements and resource consumption. WebSocket protocol enables bidirectional communication between client and server, allowing for real-time updates without the need for polling.

22) ***Heartbeats - Regular signals sent between machines to indicate network connectivity and system health.***
- ➢ Regular signals sent between machines to indicate network connectivity and system health, ensuring timely detection of failures and maintenance of system reliability. Tools like Consul or ZooKeeper can be used to implement heartbeat mechanisms for detecting network connectivity and system health.

23) ***Eventual vs Strong Consistency - Trade-offs and use cases.***
- ➢ Eventual consistency allows for temporary inconsistencies between replicas, while strong consistency ensures that all replicas are immediately consistent. Choose based on the requirements of your application. Apache Cassandra offers eventual consistency by default but also supports tunable consistency levels for scenarios where strong consistency is required.

24) ***Logging and Monitoring - Their significance in maintaining a healthy system.***
- ➢ Logging records system events and errors for troubleshooting and auditing purposes, while monitoring tracks system performance metrics in real-time. Both are crucial for maintaining a healthy system. ELK stack (Elasticsearch, Logstash, Kibana) and Prometheus/Grafana are popular logging and monitoring solutions for collecting and visualizing system logs and performance metrics.

25) ***Circuit Breakers - Learn how they can prevent system failures.***
- ➢ Mechanisms to prevent cascading failures in distributed systems by temporarily halting requests to a failing component, allowing it to recover before resuming normal operation. This helps in maintaining system stability and preventing overload. Netflix's Hystrix library provides circuit breaker patterns for Java applications, allowing you to gracefully handle failures and prevent cascading system failures.