System Design 100 topics to learn

System Requirements:

1. **Functional requirements**:

These are requirements that define specific behavior or functions, often represented by use cases.

- **How to Define**: Start with user needs and business goals. Outline the primary tasks users should accomplish.
- **Working Backwards Approach**: Begin with the end goal and trace steps backwards to identify requirements.

2. **High Availability**:

Availability is a measure of system uptime.

- **Time-based**: E.g., 99.9% (or "three nines") uptime.
- **Count-based**: E.g., 1 failure allowed in 1 million requests.
- **Design Principles**: Redundancy, failover, replication, monitoring, and regular testing.
- **Processes**: Regular maintenance, backup procedures, disaster recovery planning.
- **SLO vs SLA**: Service Level Objective (SLO) is the target level of service. Service Level Agreement (SLA) is the contract with the customer promising certain SLOs.

3. **Fault Tolerance, Resilience, Reliability**:

- **Error**: An incorrect internal state.
- **Fault**: A component's incorrect behavior.
- **Failure**: When a system as a whole stops providing the required service.
- **Fault Tolerance**: Ability of a system to behave in a well-defined manner once a fault occurs.
- **Resilience**: Ability to return to a normal state after a failure.
- **Game Day vs Chaos Engineering**: Game Day is a planned simulation. Chaos Engineering is introducing random failures.
- **Reliability**: Probability a system will fail in a given period.

4. **Scalability**:

- **Vertical Scaling**: Increasing the resources of a single node.
- **Horizontal Scaling**: Adding more nodes.
- **Elasticity vs Scalability**: Elasticity is the ability to scale automatically based on demand.

5. **Performance**:

- **Latency**: Time taken to process a single operation.
- **Throughput**: Number of operations processed in a given time period.
- **Percentiles**: E.g., 95th percentile latency = under which 95% of the request latencies fall.
- **How to Increase**: Optimize algorithms, caching, distributed systems.

6. **Durability**:

Ensuring data is safely stored and retrievable.

- **Backup Types**:
    - **Full**: Everything is backed up.
    - **Differential**: Only changes since the last full backup.
    - **Incremental**: Only changes since the last backup of any type.
- **RAID**: Redundant Array of Independent Disks. A method for redundant storage.
- **Replication**: Copying data to ensure durability and availability.
- **Checksum**: A computed value to detect errors in data.

7. **Consistency**:

Ensuring all users see the same data.

- **Eventual Consistency**: Data will become consistent over time.
- **Linearizability**: Real-time consistency.
- **Monotonic Reads**: If a process reads the value of a data item, any successive reads by that process will always return that value or a more recent one.
- **Read-Your-Writes**: Guarantees that a write by a process is visible to a subsequent read by the same process.
- **Consistent Prefix Reads**: Guarantees that every read receives a prefix (some initial segment) of the writes, in order.

8. **Maintainability, Security, Cost**:

- **Maintainability**: Ease with which a system can be maintained. Includes monitoring, testing, and deployment strategies.
- **Security**:
    - **CIA Triad**: Confidentiality, Integrity, Availability.
    - **Identity and Permissions Management**: Who can do what.

    - **Infrastructure Protection**: Protecting physical and virtual resources.
    - **Data Protection**: Ensuring data confidentiality, integrity, and availability.
- **Cost Aspects**: Engineering, maintenance, hardware, and software costs.

9. **Summary of System Requirements**:
    - **Most Popular Non-Functional Requirements**: Availability, Scalability, Performance, Security, Maintainability.

10. **Regions, Availability Zones, etc.**:
    - **Regions**: Geographic areas that consist of multiple, isolated data centers.
    - **Availability Zones**: Isolated locations within a region.
    - **Data Centers**: Physical facilities housing servers.
    - **Racks**: Structures holding multiple servers.
    - **Servers**: Physical machines running software.

11. **Physical Servers to Serverless**:
    - **Physical Servers**: Tangible hardware running software.
    - **Virtual Machines (VMs)**: Emulated computer systems.
    - **Containers**: Lightweight VMs that share the same OS kernel and isolate the application processes from each other.
        - **Pros**: Lightweight, fast, consistent environment, scalable.
        - **Cons**: Less isolation than VMs.
    - **Serverless**: Cloud-computing model which can run individual functions in response to events. Resources are fully managed by the cloud provider.
        - **Pros**: Scalability, no infrastructure management, cost-effective.
        - **Cons**: Cold starts, limited customization, statelessness.

12. **Synchronous vs. Asynchronous Communication**:
    - **Synchronous**: Sender waits for the receiver to respond.
    - **Asynchronous**: Sender sends the message and proceeds without waiting for a response.

13. **Asynchronous Messaging Patterns**:
    - **Message Queuing**: Temporarily storing messages to be processed later.
    - **Publish/Subscribe**: Senders (publishers) send messages to channels without specifying receivers. Receivers (subscribers) subscribe to channels they're interested in.

- **Competing Consumers**: Multiple consumers process messages from a single channel, distributing load.
- **Request/Response Messaging**: A two-way messaging pattern; a request is followed by a response.
- **Priority Queue**: Messages are processed based on priority rather than order of arrival.
- **Claim Check**: Large message is dropped off at a service and a claim check is provided to the requester, which can be used to get the full message later.

14. **Network Protocols**:
    - **TCP (Transmission Control Protocol)**: Reliable, connection-oriented protocol.
    - **UDP (User Datagram Protocol)**: Connectionless, no guarantee of message delivery.
    - **HTTP (Hypertext Transfer Protocol)**: Application protocol for distributed, collaborative, hypermedia systems.
        - **HTTP Request and Response**: Basic units of HTTP communication, where a client sends a request and a server responds.
15. **Blocking vs. Non-blocking I/O**:
    - **Blocking I/O**: The process is blocked until data is available.
    - **Non-blocking I/O**: The process continues even if the data is not available.
    - **Thread per Connection Model**: Each connection gets a dedicated thread.
    - **Thread per Request with Non-blocking I/O Model**: Threads are used only when processing a request.
    - **Event Loop Model**: Single thread handles multiple connections using events.
    - **Concurrency vs. Parallelism**: Concurrency is multiple tasks starting, running, and completing in overlapping time periods, while parallelism is multiple tasks running at exactly the same time.
16. **Data Encoding Formats**:
    - **Textual Formats (e.g., JSON, XML)**: Human-readable, more overhead.
    - **Binary Formats (e.g., Protobuf, Avro)**: Efficient, not human-readable.
    - **Schema Sharing Options**: How schema (structure) of data is shared between sender and receiver.
    - **Backward and Forward Compatibility**: Ensuring newer systems can read older data and older systems can read newer data, respectively.
17. **Message Acknowledgment**:
    - **Safe Acknowledgment Modes**: Receiver confirms safe receipt of a message.

- **Unsafe Acknowledgment Modes**: Sender doesn't wait for a confirmation.

18. **Deduplication Cache**:
    - **Local vs. External Cache**: Storage close to the CPU vs. external storage systems.
    - **Adding Data to Cache**: Can be added explicitly by the application or implicitly by the caching system.
    - **Cache Data Eviction**: Removing data from cache, can be based on size, time, or other criteria.
    - **Expiration vs. Refresh**: Time after which data is removed vs. updating data periodically.
19. **Metadata Cache**:
    - **Cache-aside Pattern**: Application is responsible for loading cache.
    - **Read-through and Write-through Patterns**: Cache controls the data load and update.
    - **Write-behind (Write-back) Pattern**: Data is written to cache and is asynchronously written back to the storage.
20. **Queue**:
    - **Bounded and Unbounded Queues**: Limited vs unlimited size.
    - **Circular Buffer (Ring Buffer)**: Fixed-size data structure that uses a single, continuous region of memory.
21. **Full and Empty Queue Problems**:
    - **Load Shedding**: Discarding excess traffic.
    - **Rate Limiting**: Limiting the rate of requests by users.
    - **Failed Requests Handling**: Strategies like retries or moving to dead-letter queues.
    - **Backpressure**: Signal to the producer to slow down.
    - **Elastic Scaling**: Dynamically adjusting resources based on demand.
22. **Start with Something Simple**:
    - **Single Machine vs. Distributed System Concepts**: Many concepts in distributed systems (like caching, load balancing) can be understood with the analogy of single-machine concepts.
    - **Interview Tip**: For system design interviews, start with a basic solution and then incrementally add complexity based on the requirements and constraints presented.
23. **Blocking Queue and Producer-Consumer Pattern**:
    - **Producer-Consumer Pattern**: Separate components that produce and consume data, often operating at different rates.
    - **Wait and Notify**: Mechanisms to synchronize between producers and consumers.

- **Semaphores**: A synchronization primitive used to control access to a shared resource.
- **Blocking Queue Applications**: Used in thread pooling, task scheduling, etc.

24. **Thread Pool**:

- **Pros**: Efficient use of resources, controlled resource usage, quick task start-up.
- **Cons**: Complexity, potential resource contention.
- **CPU-bound vs. I/O-bound tasks**: CPU-bound tasks spend most of their time using the CPU, while I/O-bound tasks spend time waiting for external operations (like disk or network operations).
- **Graceful Shutdown**: Properly terminating threads and releasing resources when shutting down a pool.

25. **Big Compute Architecture**:

- **Batch Computing Model**: Processing high volumes of data where a group of transactions is collected over a period.
- **Embarrassingly Parallel Problems**: Problems where little or no effort is needed to separate the problem into tasks to run in parallel.

26. **Log**:

- **Memory vs. Disk**: Storing logs in volatile memory (faster, but not persistent) vs. non-volatile disk storage (slower but persistent).
- **Log Segmentation**: Splitting logs into segments to manage them more efficiently.
- **Message Position (Offset)**: The position of a message within a log.

27. **Index**:

- **How to Implement an Efficient Index for a Messaging System**: Use data structures like B-trees or Hash Indexes to allow for quick lookups and writes.

28. **Time Series Data**:

Storing and retrieving data points indexed in time order.

- **How to Store and Retrieve**: Use databases optimized for time series data, like InfluxDB, to ensure efficient storage and fast queries.

29. **Simple Key-Value Database**:

- **How to Build**: Utilize hash tables for quick lookups. Address issues like collision resolution.
- **Log Compaction**: Reducing the size of logs by removing redundant data.

30. **B-tree Index**:

- **Usage in Databases and Messaging Systems**: B-trees allow for efficient data storage and retrieval in databases and are used to index data in messaging systems.

31. **Embedded vs. Remote Database**:

- **Embedded Database**: Runs within the same address space as the application. E.g., SQLite.
- **Remote Database**: Separate from the application and is accessed over a network. E.g., PostgreSQL, MySQL.

32. **RocksDB**:

A high-performance embedded database for key-value data.

- **Memtable**: In-memory data structure where RocksDB writes are inserted.
- **Write-ahead Log (WAL)**: A log that stores changes to data, ensuring durability.
- **Sorted Strings Table (SSTable)**: Persistent, ordered immutable map of key-value pairs.

33. **LSM-tree vs. B-tree**:

- **Log-Structured Merge-Tree (LSM-tree)**: Optimized for systems that write data more often than reading.
- **Write Amplification vs. Read Amplification**: Trade-offs between the cost of writing data vs. reading data.

34. **Page Cache**:

- **Increasing Disk Throughput**: Techniques like batching and zero-copy read can help in efficiently reading/writing to disks.

35. **Push vs. Pull**:

- **Push**: Server initiates sending data to the client.
- **Pull**: Client requests data from the server.

36. **Host Discovery**:

- **DNS**: The Domain Name System translates domain names to IP addresses.
- **Anycast**: Routing strategy where a single destination address has multiple routing paths to two or more endpoint destinations.

37. **Service Discovery**:

- **Server-side and Client-side Discovery Patterns**: Mechanisms where services find each other's network locations.
- **Service Registry**: A database containing the network locations of service instances.

38. **Peer Discovery**:

- **Peer Discovery Options**: How nodes in a network find each other.

- **Gossip Protocol**: Nodes randomly exchange information, which gradually propagates through the system.

39. **Choosing a Network Protocol**:
    - **TCP, UDP, and HTTP**: Decision depends on use case requirements like reliability, speed, and connection state.

40. **Video over HTTP**:
    - **Adaptive Streaming**: Adjusting video quality in real-time based on the viewer's network and playback conditions.

41. **CDN (Content Delivery Network)**:
    - **How to Use & Benefits**: CDNs distribute content across multiple servers to optimize access for users globally. They reduce latency and protectagainst traffic surges.
    - **Point of Presence (POP)**: Physical locations or access points that connect to the internet and host servers, acting as the local source for cached content.

42. **Push and Pull Technologies**:
    - **Short Polling**: Client frequently asks the server for new data.
    - **Long Polling**: Client requests information and waits for the server to respond.
    - **Websocket**: Protocol that allows two-way communication with a server over a single, long-lived connection.
    - **Server-Sent Events**: Server pushes updates to the client over an HTTP connection.

43. **Push and Pull Technologies in Real-Life Systems**:
    - **Quiz**: Real-life examples would be chat applications (like Slack) using Websockets for real-time messaging, or stock trading apps using long polling for updating stock prices.

44. **Large-scale Push Architectures**:
    - **C10K and C10M Problems**: Challenges of handling 10,000 and 10 million concurrent connections respectively.
    - **Large-Scale Push Architectures**: Techniques like event-driven programming to handle large numbers of simultaneous connections.
    - **Problems of Long-Lived Connections**: Resource management, detecting stale or "dead" connections, and handling dropped connections.

45. **Building Reliable, Scalable, and Fast Systems**:
    - **Common Problems in Distributed Systems**: Network failures, machine failures, performance bottlenecks.
    - **System Design Concepts**: Caching, load balancing, sharding, replication, and partitioning.

- **Three-Tier Architecture**: Presentation, logic, and data tiers.

46. **Timeouts**:
    - **Fast Failures vs. Slow Failures**: Quickly failing a request vs. taking a long time before failing.
    - **Connection and Request Timeouts**: Time limits set for establishing a connection or waiting for a response.
47. **Handling Failed Requests**:
    - **Strategies**:
        - **Cancel**: Terminate the request.
        - **Retry**: Attempt the request again.
        - **Failover**: Redirect the request to another system or component.
        - **Fallback**: Use an alternate backup solution.
48. **When to Retry**:
    - **Idempotency**: Ensuring that operations can be repeated without side effects.
    - **AWS API Failures**: Not all API failures should be retried. For instance, validation errors shouldn't be retried.
49. **How to Retry**:
    - **Exponential Backoff**: Increase the wait time between retries exponentially.
    - **Jitter**: Adding randomness to retry intervals to spread out the load.
50. **Message Delivery Guarantees**:
    - **At-most-once**: Messages are delivered once or not at all.
    - **At-least-once**: Messages are delivered one or more times.
    - **Exactly-once**: Messages are delivered precisely once.
51. **Consumer Offsets**:
    - **Log-Based Messaging Systems**: Systems where messages are stored in order and consumers track their position using offsets.
    - **Checkpointing**: Periodically saving the state of a process.
52. **Batching**:
    - **Pros and Cons**: Batching can improve efficiency but may introduce latency.
    - **Handling Batch Requests**: Ensuring that all items in a batch are processed, even if some fail.
53. **Compression**:

- **Pros and Cons**: Reduces data size but adds computational overhead.
- **Compression Algorithms**: Techniques like GZIP, Brotli, and LZ4, each with their trade-offs.

54. **Scaling Message Consumption**:
    - **Single Consumer vs. Multiple Consumers**: One consumer can ensure order but is limited in throughput. Multiple consumers can increase throughput but introduce complexity in message ordering.
    - **Problems with Multiple Consumers**: Ensuring order, handling duplicate processing, etc.
55. **Partitioning in Real-Life Systems**:
    - **Pros and Cons**: Helps in horizontal scaling and distributing loads but can introduce complexities.
    - **Applications**: Databases, messaging systems, etc.
56. **Partitioning Strategies**:
    - **Lookup Strategy**: Direct lookup.
    - **Range Strategy**: Based on a range of data values.
    - **Hash Strategy**: Using a hash function for uniform distribution.
57. **Request Routing**:
    - **Physical and Virtual Shards**: Actual data partitions vs logical partitions.
    - **Routing Options**: Methods for directing a request to the appropriate server or data partition.
58. **Rebalancing Partitions**:

Ensuring data is evenly distributed across systems, especially when adding or removing nodes.

59. **Consistent Hashing**:
    - **Implementation**: Uses a circular keyspace.
    - **Advantages and Disadvantages**: Reduces the number of re-mapped keys when scaling but can lead to non-uniform data distribution.
    - **Virtual Nodes**: Multiple hash values for a single node to ensure a more uniform distribution.
60. **System Overload**:
    - **Importance**: Protecting against system overload ensures continued availability and prevents cascading failures across dependent systems.
    - **Protection Mechanisms**: Load shedding, rate limiting, backpressure, and queuing.
61. **Autoscaling**:
    - **Scaling Policies**:

    - **Metric-based**: Scale based on metrics like CPU utilization or memory usage.
    - **Schedule-based**: Scale based on known usage patterns or times.
    - **Predictive**: Scale based on predicted future traffic patterns.
- **Autoscaling System Design**: Involves metrics collection, decision-making algorithms, and mechanisms to add or remove resources.

62. **Load Shedding**:
    - **Implementation**: Prioritize and process only the most critical tasks, while dropping or delaying non-essential tasks.
    - **Considerations**: Decide which requests to drop, how to notify clients, and when to trigger load shedding.
63. **Rate Limiting**:
    - **Purpose**: To prevent any single user or client from overloading the system.
    - **Implementation**: Token buckets, leaky buckets, and fixed window counters are popular methods.
64. **Synchronous and Asynchronous Clients**:
    - **Admission Control Systems**: Mechanisms to decide whether to process a request immediately or defer it.
    - **Blocking I/O vs. Non-Blocking I/O Clients**: Synchronous vs. Asynchronous client operations.
65. **Circuit Breaker**:
    - **Finite-State Machine**: Mechanism where the circuit breaker maintains states (Closed, Open, Half-Open) to decide whether to process or deny requests.
    - **Considerations**: Setting thresholds, reset intervals, and monitoring the health of downstream services.
66. **Fail-Fast Design Principle**:
    - **Problems with Slow Services**: Slow services can cause chain reactions leading to larger system outages.
    - **Solutions**: Implementing timeouts, circuit breakers, and other mechanisms to quickly fail and return control to the caller.
67. **Bulkhead**:
    - **Implementation**: Isolating parts of a system so that if one fails, it doesn't bring down the whole system.
    - **Applications**: Can be applied at various levels, from processes to servers to data centers.

68. **Shuffle Sharding**:

- **Implementation**: Combining sharding and redundancy to reduce the blast radius of failures.
- **Applications**: Reducing the impact of failures in multi-tenant systems.

69. **Host Discovery**:

- **DNS**: Translates domain names to IP addresses, enabling clients to discover hosts.
- **Anycast**: Uses DNS to route to the nearest or best-performing physical location.

70. **Service Discovery**:

- **Server-side and Client-side Discovery Patterns**: Mechanisms where services find each other's network locations, either with a central server or clients querying a service registry.
- **Service Registry**: Stores locations of service instances, allowing for service discovery.

71. **Peer Discovery**:

- **Peer Discovery Options**: Mechanisms like bootstrap lists or centralized directories.
- **Gossip Protocol**: Nodes share information about their peers, allowing for decentralized discovery.

72. **Choosing a Network Protocol**:

- **TCP vs. UDP vs. HTTP**: Depending on the requirements (reliability, connection setup, data size), one might be chosen over the other.

73. **Network Protocols in Real-Life Systems**:

Analyzing different scenarios and challenges to choose the best protocol. For example, streaming services might prefer UDP, while banking transactions would use TCP.

74. **Video Over HTTP**:

- **Adaptive Streaming**: The video quality is dynamically adjusted based on the viewer's network conditions.

75. **CDN**:

- **Benefits**: Faster content delivery by caching content closer to the end-users and reducing origin server load.
- **How It Works**: Content is distributed and stored across a network of servers. When a user requests content, it's served from the nearest cached server.

76. **Push and Pull Technologies in Real-Life Systems**:

Real-world applications like social media updates or live sports updates may use push technologies like WebSockets, while email clients might use pull technologies to check for new mails.

77. **Large-Scale Push Architectures**:

Design considerations for building systems that can handle millions of simultaneous connections.

78. **Timeouts**:
    - **Importance**: Ensure that a system doesn't hang indefinitely.
    - **Types**:
        - **Connection Timeouts**: Time a task will wait while attempting to establish a connection.
        - **Read/Write Timeouts**: Time a task will wait for a read or write operation to complete.
    - **Handling**: Opt for graceful degradation of service, provide feedback to users, and use retries judiciously.
79. **Handling Failed Requests**:
    - **Logging**: Essential for diagnostics. What failed, where, and why?
    - **Retry Logic**: Implementing algorithms like exponential backoff.
    - **Notifications**: Inform stakeholders or trigger systems about persistent failures.
80. **When to Retry**:
    - **Transient vs Permanent Failures**: Temporary glitches (network blip) vs. consistent failures (invalid credentials).
    - **Safety**: Ensuring retries don't exacerbate the problem.
81. **How to Retry**:
    - **Immediate vs Delayed**: Waiting between retry attempts can be crucial.
    - **Exponential Backoff**: Increasingly long waits between retries.
    - **Jitter**: Introducing randomness to retry intervals.
82. **Message Delivery Guarantees**:
    - **Importance**: Ensuring data isn't lost or seen multiple times.
    - **Strategies**: Acknowledgments, transaction logs, checkpoints.
83. **Consumer Offsets**:
    - **Checkpointing**: Storing position in a stream or log.
    - **Benefits**: Allows consumers to pick up where they left off after failures.
84. **Batching**:
    - **Benefits**: Can significantly reduce overheads and increase throughput.
    - **Challenges**: Introducing latency, complexity in error handling.

85. **Compression**:
    - **Trade-offs**: CPU time vs. bandwidth.
    - **Use Cases**: Important in bandwidth-constrained environments or for storage savings.
86. **Scaling Message Consumption**:
    - **Parallelism**: Distributing load across multiple consumers.
    - **Ordering**: Ensuring that the sequence is maintained, if necessary.
87. **Partitioning in Real-Life Systems**:
    - **Consistency vs Availability**: Deciding trade-offs based on system requirements.
88. **Partitioning Strategies**:
    - **Consistency Hashing**: Minimizing reorganization of data when nodes are added or removed.
89. **Request Routing**:
    - **Load Balancers**: Distributing incoming requests to prevent any single resource from being overwhelmed.
    - **Content-based Routing**: Directing traffic based on the content of the request.
90. **Rebalancing Partitions**:
    - **Importance**: Ensuring data and load are evenly distributed across nodes.
    - **Strategies**: Manual interventions, algorithmic rebalancing.
91. **Consistent Hashing**:
    - **Virtual Nodes**: A mechanism to ensure a more uniform distribution of data.
92. **System Overload**:
    - **Monitoring and Alerts**: Real-time insights into system health.
    - **Throttling**: Limiting user or service requests.
93. **Autoscaling**:
    - **Cloud Services**: Many cloud providers offer autoscaling services that can automatically add or remove resources based on rules or machine learning.
94. **Load Shedding**:
    - **Dynamic Adjustments**: Dropping non-critical tasks when the system is under heavy load.
95. **Rate Limiting**:
    - **API Gateways**: Many come with built-in rate limiting features.

- **Client Feedback**: Informing clients when they're exceeding limits.

96. **Synchronous and Asynchronous Clients**:
    - **Trade-offs**: Responsiveness vs. resource usage.
    - **Use Cases**: Asynchronous operations for long-running tasks.
97. **Circuit Breaker**:
    - **Resilience Patterns**: Preventing system failures from cascading.
98. **Fail-Fast Design Principle**:
    - **Health Checks**: Regular checks to ensure services are operational.
    - **Fallbacks**: Having backup systems or data sources in place.
99. **Bulkhead**:
    - **Isolation**: Making sure failures in one part don't impact others.
100. **Shuffle Sharding**:
    - **Resilience**: Improving system reliability by reducing the impact radius of failures.