Software Engineer Terms:

1. Scalability, Elasticity & Performance

- Scalability - system's ability to handle load by adding resources
- elasticity - dynamically adding/removing resources based on demand
- horizontal scaling - adding more machines (scale out)
- vertical scaling - increasing resources in the same machine (scale up)
- Fault Tolerance - system continues working despite failures
- High Availability - Ensuring minimal downtime and maxtime uptime
- Load Balancing - Distributing traffic across multiple servers to prevent overload
- Latency - Time taken for a request to travel from source to destination
- Throughput - Amount of data processed per unit time
- Rate Limiting - Restricting the number of requests a user can make in a given time

2. Caching & Data Access

- Cache : A temporary storage layer to speed up data retrieval
- Cache Hit : Requested data is found in the cache
- Cache Miss : Requested data is not in cache & must be reviewed from a slower source.
- Write Through Cache: Writes data to both cache & database simultaneously
- Write Back Cache: Writes data to cache first & updates database later
- Content Delievery Network: A network of distributed servers that deliever content faster.

3. Databases & Storage

- SQL - Language used for querying relational databases
- NoSQL - with flexible data structures like key, value ; document
- Indexing - Technique to speed up database queries
- Sharding - Splitting a db into smaller parts for better performance
- Replication - copying data into smaller chunks across multiple servers.
- Eventual Consistency - Data becomes consistent over time in distributed systems. (availability ↑)
- Cold Storage: storing infrequently accessed data on slower storage
- Hot Storage: storing frequently accessed data on fast, high performance storage.

M T W T F S S

4. System Architecture & Design Patterns

- Monolithic Architecture - A single, tightly integrated application
- Microservices - Breaking down an application into smaller, independent services
- Polling - Repeatedly checking for updates at regular intervals
- WebSockets - Realtime, 2 way communication b/w a client & a server
- Message Queue - A system for asynchronous communication by storing and forwarding messages (eg, Kafka)
- Circuit Breaker - A pattern to prevent repeated requests to a failing service, avoiding system overload
- Idempotency - Ensuring that repeated operations produce the same result
- API Gateway - A single point entry managing API requests to backend services

5. Networking & Communication

- DNS - Translates Domain Names into IP addresses
- Load Balancer - Distributes traffic across multiple servers for efficiency
- Reverse Proxy: A server that sits b/w client & backend servers to handle requests efficiently
- Edge Computing: processing data closer to the source instead of central cloud
- Auto Scaling: Automatically adjusting computing resources based on traffic

6. Cloud Computing & Virtualization

- Virtual Machine - A software based simulation of a physical computer
- Containerization - Running application in light weight, isolated environment
- Kubernetes - An orchestration tool for managing containerized apps
- IaaS - Renting VMs, storage networks (eg, AWS EC2)
- Paas - Providing development platforms without managing infrastructure
- Saas - Fully managed software accessible via the internet
- Serverless Computing - Running applications without managing servers (eg AWS Lambda)

SYSTEM DESIGN IN ACTION

Let's say you are designing a social media app like Instagram, starting with a single server and gradually scaling it to handle millions of users worldwide.

Phase 1: Starting Small (Single Server Architecture)

Initially your app runs on one server that handles everything:

- User Requests (logins, posts, comments)
- Database operations (storing user profiles, images & content)
- Serving static content (profile pictures, videos)

Everything works fine, but as user traffic grows, you hit performance bottlenecks

Phase 2: Scaling Up and Scaling Down

1. Vertical Scaling (Upgrading the Server)

You first increase CPU, RAM and storage to handle more users. This is easy but has limits - eventually, you can't upgrade hardware forever

2. Horizontal Scaling

To handle more traffic, you add multiple servers and distribute requests among them. But now, how do we ensure that requests are fairly spread out?

Phase 3 : Load Balancing (Distributing Traffic Efficiently)

A load balancer sits between users and your servers.
It distributes requests using algorithms like:

- Round Robin : Cycles requests evenly across all servers
- Geolocation-based Routing: Sends users to the nearest server to reduce latency

Now, users are handled smoothly across multiple servers, but there's still one problem – our database is a bottleneck

Phase 4: Optimizing Database Performance

4. Database Sharding (Splitting Data)

Instead of storing everything in a database, we split user data across multiple shards (e.g, users A-M in one DB, N-Z in another.

5. Replication (Ensuring Redundancy & Fast Reads)

- A **LEADER DATABASE** handles writes, and FOLLOWER DATABASE handle reads
- Users requesting profiles read from followers, while new posts & comments go to the leader

Now, database load is distributed, but our app still has slow loading times for frequently accessed data

PHASE 5 : Improving Speed with Caching

6. CDN (Faster Delivery of Images & Videos)

A Content Delivery Network caches static files (profile pics, videos) on global servers, reducing load time

7. Caching (Storing Frequently Accessed Data In Memory)

Instead of quering the database repeatedly, we store popular user profiles and posts in cache (e.g, Redis, Memcached)

- Cache Hit : Data is found in cache → Fast Response
- Cache Miss: Data is fetched from DB and stored in cache for next time

Now, our app is fast & scalable, but we also need to handle real-time updates (e.g, new messages, notifications)

Phase 6 : Enabling Real-time Features

8. WebSockets (Instant Messaging & Notifications)

Instead of polling the server every few seconds, we use WebSockets for realtime updates:

- When a user sends a msg, the server immediately pushes it to the recipient
- Notifications (likes, comments) are also instantly updated.

Now, the app has millions of users, real-time updates and optimized performance - but how do we ensure reliability in case of failures?

Phase 7: Making the System Fault-Tolerant

9. Redundancy & Fault Tolerance

- Multiple servers in different regions prevent a single point of failure
- Failover Mechanisms: If one server crashes, another takes over automatically

10. CAP Theorem & Data Consistency

In distributed system, we must balance:

- CONSISTENCY: Every user sees the latest data!
- AVAILABILITY: System always online
- PARTITION TOLERENCE: This system can handle network failures

Can only provide any two!

Phase 8: Efficient API Communication

- Rest API (Traditional): Each request fetches a fixed set of data
- GraphQL (Efficient Queries): Clients request only the needed fields, reducing over-fetching
- gRPC (Fast Server to Server Communication): Uses Protocol Buffers instead of JSON, making it faster

Phase 9: Asynchronous Processing with Message Queues

12. Messaging Queue (Handling Background Jobs)

To prevent delays in the main app, we offload heavy tasks to a message queue (Kafka, RabbitMQ):

- When a user uploads a video, it doesn't process immediately. Instead, a job is sent to a queue for background processing
- Email Notifications (eg, "You got a new follower!") are also handled asynchronously.

13. Security Measures:

- **Authorization** - checking if a user has permission to access a resource
- **Authentication** - verifying the identity of a user (e.g, login with username & password)
- **OAuth** - A protocol for secure authentication (eg, login with Google)
- **DDOS** - Overloading a system with fake traffic to disrupt service (Rate Limiting: Restricting req per second)
- **Zero Trust Architecture** - A security model that assumes no trust, requiring verification at every step.