# System design

## Part I

**System design** is the process of **planning and designing the overall architecture** of a software system—how different components interact, scale, store data, handle traffic, stay secure, and remain reliable.

Overall: How should we build this system so it works efficiently, reliably, at scale.

**What System Design Includes?**

1. **Architecture:**
   How components fit together:
   - Client → Backend → Database
   - Monolithic
   - Microservices
   - Event-driven systems
   - APIs and communication patterns
2. **Scalability**:
   How to handle more users:
   - Horizontal and vertical scaling
   - Load Balancer
   - Caching
   - Sharding
   - Replication
3. **Data Storage:**
   Choosing the right DB:
   - SQL
   - NoSQL
   - Distributed databases
   - Indexing
   - Partitioning
4. **Performance:**
   - Caching (Redis/Memcached)
   - CDN
   - Queues (Kafka/RabbitMQ)
   - Asynchronous processing
5. **Reliability:**
   Ensuring system is always available:
   - Failover
   - Replication
   - Redundancy
   - Backups
   - Avoiding single points of failure

6. **Security:**
    - Authentication
    - Authorization
    - Encryption
    - Rate limiting
    - Secure API design

**Why System Design Is Needed?**

Because real apps must handle:

- Millions of users
- High throughput (Throughput is a measure of the actual amount of data successfully transmitted or processed by a system in a given time frame)
- Low latency (very little delay)
- Fault tolerance (system keeps working even when some parts fail)
- Rapid scaling
- Data consistency

# Describe every Points:

1. Client → Backend → Database:
   The most common flow:
    - **Client (browser/app)** sends requests
    - **Backend (server)** processes logic
    - **Database** stores/returns data
    - Can include middle layers like API gateway or service mesh
2. Microservices:
    - Application is broken into **small independent services**
    - Each service does one job (e.g., user service, payment service)
    - They communicate via APIs or messaging
    - Each service can be deployed independently
    - Each service can use a **different programming language**
    - Databases can be **separate per service**
    - Easier to scale individual services
    - Failure in one service doesn't break the entire app
3. Monolith:
    - The entire application is **one single codebase**
    - All features tightly connected
    - Simple to develop initially, harder to scale as it grows
    - If one-part crashes, whole system may go down
4. Event-driven systems:
   A system where **something happens (an event)** → and that **triggers another action automatically**.
    - Components communicate via **events**
    - Example: "Order Placed" → triggers "Payment Processing" → triggers "Send Email"
    - Very good for asynchronous workflows
    - Useful for real-time actions (notifications, logs, analytics)

- Uses message brokers (Kafka, RabbitMQ)
- Loose coupling → systems depend on **events**, not on direct calls
- High throughput and fault-tolerant

Event: **User places an order**

This single event triggers many other tasks:

- Deduct money
- Reduce inventory
- Send email
- Notify warehouse
- Update analytics
- Generate invoice

5. APIs & communication patterns:
    - REST, GraphQL, gRPC
    - How services talk to each other
    - Defines request/response formats and communication style (sync/async)
    - Synchronous: REST/gRPC (client waits for response)
    - Asynchronous: events/queues (client does not wait)
    - Helps define service boundaries
    - API Gateway routes, validates, and secures calls
6. Horizontal vs Vertical scaling:
    - **Vertical:** Add more power to the same machine (more CPU/RAM)
    - **Horizontal:** Add more machines and distribute load (preferred at scale)
    - Horizontal scaling supports **infinite growth**
    - Vertical scaling has hardware limits
7. Load balancers:
    - Distribute incoming requests across multiple servers
    - Prevents any one server from being overloaded
    - Prevent system overload
    - Enable zero-downtime deployment
    - Can perform health checks
8. Caching:
    - Store frequently used data in memory (Redis, Memcached)
    - educes database load and makes response faster
    - Reduces server load
    - Improves latency
    - Can be used at DB level, API level, or CDN level
9. Sharding:
    - Splitting data across multiple databases
    - Good for huge datasets (millions of rows)
    - Reduces DB bottleneck
    - Requires careful key selection
    - Example: users A–K in DB1, L–Z in DB2
10. Replication:
    - Having multiple copies of the same database
    - Helps with read scaling and redundancy
    - Helps with read-heavy systems

- Provides high availability
- Supports multi-region systems

11. SQL:
    - Structured data
    - ACID transactions
    - Good for finance, orders, critical systems
    - Ensures strong consistency
    - Good for complex queries (joins, transactions)
12. NoSQL:
    - Flexible schema
    - Supports huge scale
    - Good for logs, social networks, real-time apps
    - Handles unstructured data
    - Best for large-scale distributed systems
13. Distributed databases:
    - Data stored across multiple nodes
    - Systems like Cassandra, CockroachDB
    - Designed for global scale
    - Provide automatic failover
    - Ensure data durability across nodes
14. Indexing:
    - Speeds up search queries by creating quick-lookup data structures
    - Like an index at the back of a book
    - Slows down writes but speeds up reads
    - Critical for search-heavy apps
15. Partitioning:
    - Splitting large tables into smaller parts
    - Improves performance and manageability
    - Avoids huge table performance degradation
    - Helps query only relevant partitions (faster reads)
16. Caching (Redis/Memcached):
    - Store hot data in memory
    - Avoid hitting the database frequently
    - In-memory = fastest
    - Reduces database cost
17. CDN (Content Delivery Network):
    - Stores static content (images, CSS, videos) near the user
    - Reduces latency globally
    - Offloads traffic from origin server
    - Essential for global applications
18. Queues (Kafka/RabbitMQ):
    - Used for asynchronous tasks
    - Helps handle spikes in traffic
    - Smoothens traffic spikes
    - Ensures reliability during failures
    - Example: send email, generate report, process notifications

19. Asynchronous processing:
    - Background jobs handle heavy work
    - Keeps the main request fast
    - Keeps UI responsive
    - Ideal for long-running tasks
20. Failover:
    - When a primary server fails, automatically switch to a backup server
    - Can be active-active or active-passive
    - Ensures system stays online
21. Replication:
    - Multiple copies of data to avoid loss and ensure availability
    - Creates durability
    - Helps recovery during disasters
22. Redundancy:
    - Duplicate components (servers, databases, networks)
    - If one breaks, another takes over
    - Used at multiple layers (network, servers, databases)
23. Backups:
    - Periodic snapshots of data
    - Used for disaster recovery
    - Cold backups, warm backups, snapshots
24. Avoiding single points of failure:
    - No single component should bring the whole system down
    - Redundant servers, multi-AZ, multi-region setups
    - Use multiple servers in different availability zones (AZs)

25. Authentication:
    - Verifying who the user is
    - Includes MFA (multi-factor authentication)
    - Example: login with email/password, OAuth, tokens
26. Authorization:
    - Verifying what the user is allowed to do
    - Roles and permissions
    - RBAC (Role-based access control)
    - Example: admins vs normal users
27. Encryption:
    - Secure data so no one can read it without the key
    - Encryption at rest + encryption in transit
    - HTTPS, encrypted database fields
28. Rate limiting:
    - Restrict how many requests a user can make
    - Protects against abuse/DDOS attacks
    - Prevents brute-force login attempts
    - Protects against API spamming

29. Secure API design:
    - Input validation
    - Avoid SQL injection
    - Token-based access
    - Proper error handling
    - Should log suspicious behaviour
    - Should hide sensitive error messages

➢ **In next part, we will discuss about every single point in details**