Foundations of System Design

Properties of Distributed Systems

<u>Vertical Scaling</u> – Optimize processes & increase throughput using same resources

~~Ex~~ · Increasing capacity of a single server by adding more resources like CPU, RAM, Storage

P~~reprosse~~

<u>Preprocessing</u> – Preprocess or perform certain heavy tasks in advance & store results for quick access when actually required. Can be done through – Batch Jobs (cron), Offline Workers (kafka consumer),

<u>Resilience</u> – Fault tolerant such that system never fails. ~~Like take backups or~~

1. Take & keep backups.
2. No single point of failure.
3. Can keep master slave architecture.

<u>Horizontal scaling</u> – Buying more machines of same type to get more work done.

Microservices - Breaking large applications into smaller, independent & loosely coupled services each having a single responsibility.

1. Faster & independent development & maintainence
2. Flexible.
3. Single point of responsibility.
4. Easier deployment & testing & scalability.
5. Resilient as single service cannot crash entire system

Distributed systems - Collection of nodes that work together as a single system for the user. Characteristics -

1. Distribution of loads -
2. Complete system is fault tolerant to user.
3. Resource sharing & message passing.
4. Act as local server to nearby requests

Load Balancer - A traffic manager. Distributes incoming requests across multiple servers so that no single server is overloaded & system stays fast, reliable & scalable. Characteristics -

1. Traffic spread evenly.

Decoupling - Separating out concerns so that it can be handled independently. Can be achieved by -

1. Abstraction
2. Async communication
3. Dependency injection

Avoiding Single Point of Failures

Avoiding Single Point of failures -

Any single point that can fail should not make the whole system down.

1. Services - Adding more nodes.
2. DB - Backups in MASTER-SLAVE architecture. slaves take backups
3. Whole system - Having same setup in Multiple regions.

~~Less imp (effects & next steps to take due to above).~~

1. Due to multiple services/nodes - We ~~might~~ would need Load Balancer (with a gateway).
2. With Load Balancer, again its single point failure - group of LBs.
3. Now due to group of LBs - need to use DNS which will route to appropriate LB.

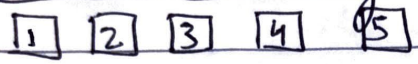
Scalability in distributed systems

Scalability

Ability to handle more requests

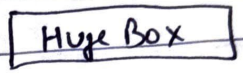
1. Vertical scaling - Buying bigger machine
2. Horizontal scaling - Buying more machines.

Horizontal scaling



1. Load Balancer required.
2. Resilient - if 1 goes down, others serve
3. Network calls between 2 systems (RPC) - SLOW
4. Data inconsistency - as transactions can be spread across servers
5. Scales well as users increase, kind of Linear.

Vertical scaling



1. ~~Single point of failure.~~ No LB req
2. Single point of failure.
3. Interprocess communication between 2 services - FAST
4. Data consistency - Single point of transactions.
5. There is a Hardware limit

OPTIMAL SOLUTION - Hybrid → Horizontal scaling where each box is a huge box so that system is Resilient Consistent & scales well & Faster.

Replication in Distributed Systems

Replication in Distributed Systems

Making copies of data across multiple nodes. Types

1. Full Replication — • All data copied to all nodes.
   • Overhead, Performance hit, but Consistent
2. Partial Replication – • Only specific ~~part~~ portions of data replicated.
   • Faster, can be inconsistent.
3. Leader/Follower Replication — • One leader handles writes
   • Other nodes follow.
   • ~~Reads can go to the leader~~ • One form is- Master-Slave.

4. Quorum Based Replication –
   • Instead of only 1 leader, multiple nodes participate in deciding.
   • Client writes/reads are considered successful only if a majority (Quorum) of replicas agree.
   • Helps balance C vs A.
   • Rule – $R+W > N$, where N = Total replicas

Architectures

Client–Server Architecture

A client–server architecture is a model where multiple clients (users, applications, or devices) request services or resources from a centralized server.

- The server hosts, manages, and provides resources (like data, files, computation).
- The client initiates communication by sending a request, and the server responds.

Key Characteristics

1. Two main entities: Client (consumer) and Server (provider).
2. Request–Response model: Client sends a request → Server processes → Server sends back response.
3. Centralization: The server is typically a central authority/resource holder.
4. Scalability: Can scale vertically (more power to the server) or horizontally (multiple servers with load balancers).
5. Examples
    a. Web applications: Browser (client) ↔ Web server (server).
    b. Email: Outlook/Gmail client ↔ Mail server.
    c. Database access: Application ↔ Database server.
6. Advantages
    a. Centralized management (easier to maintain).
    b. Clients don’t need to handle heavy processing — offloaded to server.
    c. Security and backups can be centralized.

7. Disadvantages
    a. Single point of failure: If server goes down, all clients are affected.
    b. Scalability challenges: Too many clients can overload a single server.
    c. Network dependency (client can't function offline).
8. Evolution
    a. Early days: Single server handling requests.
    b. Modern times: Evolved into multi-tier (3-tier) and distributed architectures (microservices, service mesh).

3-Tier Architecture

A 3-tier architecture is a client–server architecture where the application is split into below layers where each tier runs on a separate system or process, communicating via well-defined protocols (usually HTTP, JDBC, etc.):

1. Presentation Layer (UI)
    a. user interface or the "client-facing" part.
    b. Examples: Web browser, mobile app, React/Angular frontend.
    c. Responsible for displaying data and collecting user input.
2. Application Layer (Business Logic)
    a. The "brain" of the system.
    b. processes requests, rules, workflows, authentication, authorization.
    c. Example: Spring Boot, Node.js, Django services.

3. Data Layer (Database)
    a. stores and retrieves data and transactions.
    b. Example: MySQL, PostgreSQL, MongoDB.

1. Advantages
    a. Separation of concerns → Easier to maintain & scale independently.
    b. Scalability → App layer and DB can scale differently.
    c. Security → DB is never directly exposed to client.
    d. Flexibility → Different tech stacks for each layer (React + Java + MySQL).
2. Disadvantages
    a. More complexity than 2-tier (needs networking, deployment setup).
    b. Performance overhead due to extra hops.
    c. Debugging is harder across multiple layers.
3. Evolution
    a. 2-tier (Client ↔ DB) → tightly coupled, hard to scale.
    b. 3-tier solved coupling and introduced separation.
    c. Now extended into n-tier & microservices, where the application layer itself is decomposed into multiple small services.

Microservices Architecture

Microservices architecture is a style where a large application is broken into a collection of small, loosely coupled services. Each service is:

- Independent (can be developed, deployed, and scaled on its own).
- Owns its own data (separate database or schema).
- Communicates with other services via APIs (HTTP/REST, gRPC, or messaging queues like Kafka/RabbitMQ).

1. Structure
    a. Client (Presentation Layer) – same as before (web, mobile).
    b. API Gateway – entry point that routes requests to the correct microservice.
    c. Microservices – each handling a domain like Auth Service, Cart Service, Order Service
    d. Database per Service – avoids tight coupling.
    e. Messaging/Event Bus (optional) – for asynchronous communication (Kafka, RabbitMQ).
2. Advantages
    a. Scalability – scale hot services (e.g., Payment) without scaling everything.
    b. Agility – teams work independently on different services.
    c. Fault Isolation – failure in one service (e.g., Notifications) doesn’t bring down the entire app.
    d. Technology Flexibility – each service can use different tech (Java, Node.js, Python).

e. Faster Deployment – small services can be deployed independently (CI/CD).

3. Disadvantages

    a. Complexity – distributed system problems (network latency, retries, failures).

    b. Operational Overhead – need service discovery, monitoring, logging, tracing.

    c. Data Management – difficult to keep data consistent across services.

    d. Testing is harder – end-to-end testing across multiple services.

    e. DevOps dependency – requires containerization (Docker, Kubernetes).

Event-Driven Architecture (EDA)

An event-driven architecture is a design pattern where the system is built around events. An event = a significant change in state (e.g., UserRegistered, OrderPlaced, PaymentProcessed). Instead of direct service-to-service calls, services publish events and other services subscribe to those events.

1. Core Components

    a. Event Producer – generates/publishes events (e.g., Order Service publishing OrderPlaced).

    b. Event Broker – middleware that routes events (e.g., Kafka, RabbitMQ, AWS SNS/SQS).

    c. Event Consumer – listens for events and reacts (e.g., Notification Service sends email when OrderPlaced is received).

2. Types of Event-Driven Communication

    a. Point-to-Point (Queue)

        i. One producer → one consumer.

ii. Example: Order placed → Payment service (only one service consumes).

b. Publish/Subscribe (Topic)

i. One producer → multiple consumers.

ii. Example: Order placed → Inventory, Payment, Notification all consume.

3. Advantages

a. Loose Coupling – producers don't care who consumes.

b. Scalability – consumers can scale independently.

c. Asynchronous – services don't block waiting for responses.

d. Extensibility – adding a new service (e.g., Analytics) is easy, just subscribe.

e. Resilience – if one consumer is down, events can be replayed later (depending on broker).

4. Disadvantages

a. Eventual Consistency – data isn't updated instantly everywhere.

b. Debugging/Tracing is hard – hard to track an event's journey across services.

c. Message Duplication – consumers must handle duplicate events.

d. Complexity – requires strong monitoring, dead-letter queues, retries.

5. When to Use

a. High scale, decoupled systems.

b. Event-heavy applications:

c. E-commerce (OrderPlaced → Payment, Inventory, Shipping).

d. IoT (sensors generating events).

e. Logging & monitoring pipelines.

f. Real-time systems (stock trading, chat apps).

Peer-to-Peer (P2P) Architecture

In P2P architecture, there is no central server. Every node (called a peer) acts as both: Client (requesting resources) and Server (providing resources). Responsibility is distributed across peers.

1. Examples

    a. File sharing: BitTorrent, eMule.

    b. Messaging: Early Skype (before moving to more centralized infra).

    c. Blockchain/cryptocurrency: Bitcoin, Ethereum.

2. Advantages

    a. No single point of failure – network continues even if some peers fail.

    b. Scalability – more peers joining = more resources.

    c. Cost-effective – no central server needed.

    d. Self-organizing – peers can dynamically join/leave.

3. Disadvantages

    a. Data consistency issues – keeping all peers updated is hard.

    b. Security risks – every peer is exposed, making it harder to enforce trust.

c. Performance variability – depends on peers' availability & bandwidth.

d. Difficult monitoring/management – no central control point.

4. Types of P2P

    a. Unstructured P2P

        i. Peers randomly connect.

        ii. Example: Gnutella, early P2P networks.

        iii. Pros: Easy to join.

        iv. Cons: Searching for data is inefficient (flood queries).

    b. Structured P2P

        i. Uses a specific topology and algorithms like Distributed Hash Tables (DHTs).

        ii. Example: BitTorrent, Kademlia.

        iii. Pros: Efficient searching & routing.

        iv. Cons: More complex to implement.

5. When to Use P2P

    a. File distribution (large files, many users → torrenting).

    b. Blockchain (needs decentralization & fault tolerance).

    c. Collaborative apps (real-time P2P gaming, messaging, IoT).

Serverless Architecture

Serverless does not mean there are no servers — it means developers don't manage servers directly. Cloud providers (AWS, Azure, GCP) manage infrastructure, scaling, and maintenance. Developers just write functions (called FaaS – Function as a Service) or use managed services, and the cloud runs them on demand.

1. How It Works
    a. The developer writes small independent functions (e.g., processPayment()).
    b. Function is deployed to a cloud provider.
    c. The function is triggered by an event (HTTP request, DB update, file upload, message in queue).
    d. Cloud provider:
        i. Allocates compute on demand
        ii. Executes the function
        iii. Scales automatically
        iv. Shuts down when not needed
    e. You only pay for execution time (not idle servers).
2. Examples - AWS Lambda, Azure Functions, Google Cloud Functions.
3. Advantages
    a. No server management – infra handled by cloud.
    b. Scales automatically with load.
    c. Cost-efficient – pay per execution, not 24/7 uptime.
    d. Faster development – focus only on business logic.

e. Event-driven – integrates easily with APIs, queues, databases.

4. Disadvantages

    a. Cold starts – first request after inactivity is slow.

    b. Limited execution time – functions often capped (e.g., AWS Lambda 15 mins).

    c. Vendor lock-in – tied to provider APIs.

    d. Debugging & monitoring harder – distributed execution across ephemeral instances.

    e. Not suitable for long-running processes (like ML training).

5. Use Cases

    a. Web APIs (REST/GraphQL endpoints)

    b. Data processing (image resize, log processing).

    c. Real-time notifications (IoT, chat, push notifications).

    d. Scheduled jobs (cron-like tasks).

    e. Event-driven workflows (payment → notification → email).

Monolith vs Microservice

Monolith Architecture	Microservice architecture
1. Single unified codebase where all modules like UI, business logic, DB, etc reside as 1 deployable unit.	1. Independent services that has single responsibility & works together as a system by communicating via APIs.
2. Deployment as single unit.	2. Deployment independent as per service
3. Scalability - Need to scale entire app irrespective of module.	3. Each independent service can be scaled as required.
4. Tight coupling	4. Loose coupling
5. Inter module communication is faster due to RPCs.	5. Inter service communication is slow due to Network calls &.
6. Faulty - if monolith service can be goes down, whole system goes down.	6. Better fault isolation, i.e, if 1 service is down, whole system is not down.
7. Better for small teams, small apps, small projects.	7. Better for large, heavy systems that might require scalability in future.

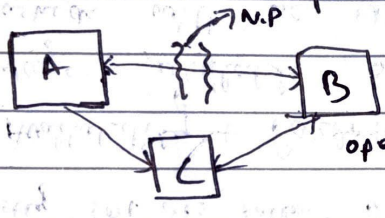
Distributed Coordination

CAP Theorem

CAP

In a distributed system, it is impossible to guarantee all 3 - CONSISTENCY (C), AVAILABILITY (A) and PARTITION TOLERANCE (P). You can have atmost 2 out of 3.

1. PARTITION TOLERANCE (P) - In a distributed system, a network partition (or split brain) occurs when a network failure causes a distributed system to split into 2 or more isolated subnetworks, preventing nodes within different partitions from communicating with each other. Partition Tolerance (P) is the ability of a system to continue operating correctly even when network partitions occur between nodes.



N.P means if network issue happen due to which 2 partitions (A alone & B+C) operates independently.

Now in a distributed system, Partition Tolerance is unavoidable (coz networks can always fail), so in general the choice is between the two - Consistency & Availability

☆ CP (Consistency + Partition Tolerance) -

- Always consistent but may sacrifice Availability during partitions
- Use cases - When value of info needs to be accurate.
  e.g Banking transactions, Payments, Text messages, Reservation systems, Inventory
- DB options - Mongo DB, HBase, Redis.
- Mostly SQL dbs follow ACID

☆ AP (Availability + Partition Tolerance) -

- Always available, but may return stale data (which will be eventually consistent).
- Use cases - When service is more imp than info.
  e.g. Social Media feeds, Product catalogs, E-com.
- DB options - Cassandra, Dynamo DB, Couch DB, Cosmos.
- Mostly NO SQL DB follow BASE

☆ CA (Consistency + Availability)

- Possible on a single-node system (with no partitions)
- In real distributed systems, NOT Possible.

Interview - Df with always (like Banking) → Pick CP
Df always up, even if slightly stale (like FB post likes) → AP
DB will depend on what AP/CP (some DB provide options to flip & prefer - C or A, eg. Cassandra, Cosmos).

Consistent Hashing

Consistent Hashing

This a technique for distributing requests or data across multiple servers in a way that minimizes reassignments when servers are added or removed.

Used in - Distributed systems (e.g caching sharding DB) & L.B.

How it works ->

1. Hashing the server - Each server is assigned a position on a "hash ring" using a hash function.
2. Hashing the request - The request's key (e.g user id, IP) is hashed & placed on same hash ring.
3. The request goes clockwise to first nearest server.

In a typical hashing & request to server assignment, if a server is added (say 4 from 4 to 5 servers), almost all keys change & all requests reshuffle, cache misses happen. Consistent hashing solves this by remapping only a small fraction of keys.

Here Hash Ring is imp. ~~It has~~ It ->

1. All servers & requests are hashed & arranged in this ring format.
2. Each server is actually assigned to more than 1 point (say N) using Sever using n number of hash functions. This is imp as it solves the problem of skewness or overloading on 1 server incase of failure.
3. Each request is that is placed once on the ring is served by clockwise nearest server.

Gossip Protocol

The Gossip Protocol (also called epidemic protocol) is a peer-to-peer communication mechanism used in distributed systems to spread information (like cluster membership, state, or updates) in a way that mimics how gossip spreads in real life.

Instead of one central node telling everyone, each node randomly picks another node and shares what it knows. Over time, information “infects” the entire network.

Key Ideas:

1. Randomized communication:
    - Each node periodically selects a random peer and exchanges state.
    - Information spreads gradually but quickly.
2. Eventually consistent:
    - Not all nodes are updated at the same time, but eventually the entire cluster converges to the same state.
3. Scalability & fault-tolerance:
    - No central coordinator, so even if some nodes fail, the gossip continues.
    - Works well in large-scale distributed systems.
4. Strengths:
    - Scalable, decentralized, fault-tolerant, low-overhead.
5. Weaknesses:
    - Eventual consistency (not instant), redundant messages.

Where is the Gossip Protocol used?

- Cluster membership & failure detection → Used in systems like Cassandra, DynamoDB, Akka, Consul to keep track of which nodes are alive.

- Data replication & consistency → Spreads updates about data across nodes.
- Leader election support → Helps nodes detect failures and elect a new leader.

Consensus Algorithms

In a distributed system, multiple nodes (servers) must agree on a single value/state, even if some nodes fail or give conflicting info.

Consensus = “agreeing on the truth in a cluster of machines.”

Without consensus → different nodes may “think” differently → chaos.

Why is it needed?

1. Leader Election → Who is the master/leader?
2. Consistency of data → All replicas agree on updates.
3. Fault Tolerance → Even if some nodes crash, the system continues.

Main Algorithms -

1. Paxos
    a. The earliest well-known algorithm.
    b. Ensures consensus even if some nodes fail.
    c. Very theoretical & complex (not widely implemented directly).
    d. Key roles: Proposer, Acceptor, Learner.
    e. Works via multiple rounds of proposals until majority agrees.

    f. Known as correct but too hard to implement in real-world systems.

2. Raft

    a. Created later as a more understandable alternative to Paxos.

    b. Most modern distributed systems (etcd, Consul, Kafka's controller election) use Raft.

    c. Key steps:

        i. Leader Election → Cluster picks a leader node.

        ii. Log Replication → Leader appends commands to log, replicates to followers.

        iii. Safety → Only committed entries are applied.

    d. Easier to explain → widely used in practice.

3. Zab (Zookeeper Atomic Broadcast)

    a. Used in Apache Zookeeper (coordination service).

    b. Similar to Raft but tailored for Zookeeper's needs.

Service Discovery

In microservices, services run on multiple hosts/containers and their IPs/ports may change dynamically (due to scaling, failures, or restarts). Service Discovery is the mechanism by which services automatically find and communicate with each other without hardcoding addresses.

Two main types of Service Discovery

1. Client-Side Service Discovery

a. How it works:

    i. The client is responsible for finding the service location.

    ii. The client queries the Service Registry (like Eureka, Consul, Zookeeper) to get the list of available instances.

    iii. The client then applies load balancing logic (e.g round robin, weighted)

    iv. Finally, the client directly calls the chosen service instance.

b. Flow Example: Order Service → queries Eureka → gets IP:Port of Payment Service instance → calls it directly.

c. Pros

    i. Simple, no need for an extra network hop (client talks directly to service).

    ii. Clients can implement smart load balancing.

d. Cons

    i. Every client must have service discovery + load balancing logic built in.

    ii. Harder to update logic in many clients (tight coupling).

e. Examples:

    i. Netflix Eureka + Ribbon (Java clients).

    ii. Consul client-side discovery libraries.

2. Server-Side Service Discovery

    a. How it works:

i. The client does not query the registry directly.

ii. Instead, the client sends the request to a load balancer or proxy.

iii. The load balancer queries the service registry, picks a healthy instance, and forwards the request.

b. Flow Example: Order Service → calls Load Balancer (e.g., AWS ELB) → LB queries Consul → LB forwards to Payment Service instance.

c. Pros

i. Clients are simple (they just call one endpoint).

ii. Centralized load balancing logic.

iii. Easier to manage in large heterogeneous environments.

d. Cons

i. Adds extra network hop (client → load balancer → service).

ii. Load balancer could be a single point of failure (though usually made redundant).

e. Examples:

i. AWS Elastic Load Balancer, NGINX + Consul.

ii. Kubernetes Service (ClusterIP, NodePort, Ingress → kube-proxy acts as load balancer).

Components of Service Discovery

1. Service Registry

- A central database where services register themselves when they start, and deregister when they stop.

- Examples: Eureka, Consul, Zookeeper, etcd.

2. Health Checking

    - Ensures only healthy instances are discoverable.

3. Load Balancing

    - Distributes requests among available instances.

Why is Service Discovery needed?

Without it:

- Services must hardcode IPs/ports of other services → brittle.
- Scaling and dynamic orchestration (Kubernetes, Docker) would break things.

With it:

- Services just call “UserService”, registry resolves to correct IP/port.

Communication and APIs

REST vs GraphQL vs RPC

Aspect	REST	GraphQL	RPC (e.g., gRPC, Thrift)
Definition	API style using resources (/users/1) and HTTP verbs (GET, POST, etc.)	Query language for APIs where clients specify exactly what data they need	Remote Procedure Call – invokes methods on a remote server as if local
Data Model	Resource-oriented (nouns: /users, /orders)	Flexible, strongly-typed schema with queries & mutations	Function/method-oriented (getUser(id))
Transport	Mostly HTTP/HTTPS	HTTP/HTTPS (typically POST)	HTTP/2 (commonly) or custom protocols
Payload Format	JSON (usually)	JSON response (client decides fields)	Protocol Buffers (gRPC), Thrift, Avro → compact binary
Over-fetching / Under-fetching	Common (fixed endpoints return too much or too little data)	Avoided (client specifies exact fields)	Not an issue (methods return what is defined)
Performance	Can require multiple round trips (e.g., fetch user, then orders)	Single query can fetch nested data in one request	Very efficient (binary serialization,

			multiplexed streams in gRPC)
Schema	Not enforced by default (can use OpenAPI/Swagger)	Strongly typed schema with introspection	Strict interface definition (IDL like .proto in gRPC)
Ease of Use	Simple, human-readable, easy to test via browser/curl	Flexible for clients, but needs schema management	Requires code generation, but efficient for machine-to-machine comm.
Use Cases	Public APIs, CRUD services, simple web/mobile apps	Client-driven apps (mobile, frontend-heavy apps)	Internal microservices, high-performance backend systems
Learning Curve	Low	Medium (requires GraphQL schema + resolvers)	Higher (needs IDL, codegen, binary protocol understanding)

WebSocket

A communication protocol providing full-duplex (two-way) communication between client and server over a single TCP connection. Unlike HTTP (request → response), WebSocket allows persistent connection where both client and server can push data anytime.

How it works

1. Starts as an HTTP handshake (client sends request with Upgrade: websocket).
2. Server upgrades the connection → switches protocol from HTTP → WebSocket.
3. Connection stays open until explicitly closed.
4. Data can now flow bi-directionally in real time.

Why WebSocket?

1. HTTP is stateless & client-driven → server can't push updates unless client keeps polling.
2. Polling = inefficient (lots of wasted requests).
3. WebSocket = efficient real-time communication.

Use Cases

1. Chat applications (WhatsApp Web, Slack).
2. Live notifications (Facebook/Instagram likes, Twitter updates).
3. Stock tickers / Crypto trading dashboards (real-time price updates).
4. Online gaming (low latency, constant communication).
5. Collaborative tools (Google Docs live editing).

WebSocket vs Alternatives

1. Polling: client asks server every X seconds → wasteful.

2. Long Polling: client request kept open until server has data → better, but still costly.
3. Server-Sent Events (SSE): server → client one-way push only.
4. WebSocket: full-duplex, real-time, efficient.

DNS (Domain Name System)

DNS is like the phonebook of the internet. It translates human-readable domain names (e.g., google.com) into IP addresses (e.g., 142.250.182.78) that computers use to communicate.

Why DNS is Important?

- Users can't remember IP addresses.
- IPs may change (due to scaling, load balancing, cloud migration).
- DNS makes applications resilient to these changes by keeping the domain name constant.

How DNS Resolution Works

1. User types www.example.com in the browser.
2. The request goes through:
    a. Browser Cache → check if IP is already cached.
    b. OS Cache (local machine).
    c. ISP's Recursive Resolver → asks multiple servers.
    d. Root DNS Server → points to TLD (.com, .org, etc.).

e. TLD Server → points to the authoritative name server.

f. Authoritative DNS Server → final mapping of domain → IP.

3. IP is returned → browser connects to server.

Types of DNS Records

- A Record → Domain → IPv4 address.
- AAAA Record → Domain → IPv6 address.
- CNAME → Alias to another domain.
- MX Record → Mail servers.
- NS Record → Name servers.
- TXT Record → Metadata (SPF, security, verification).

DNS in System Design

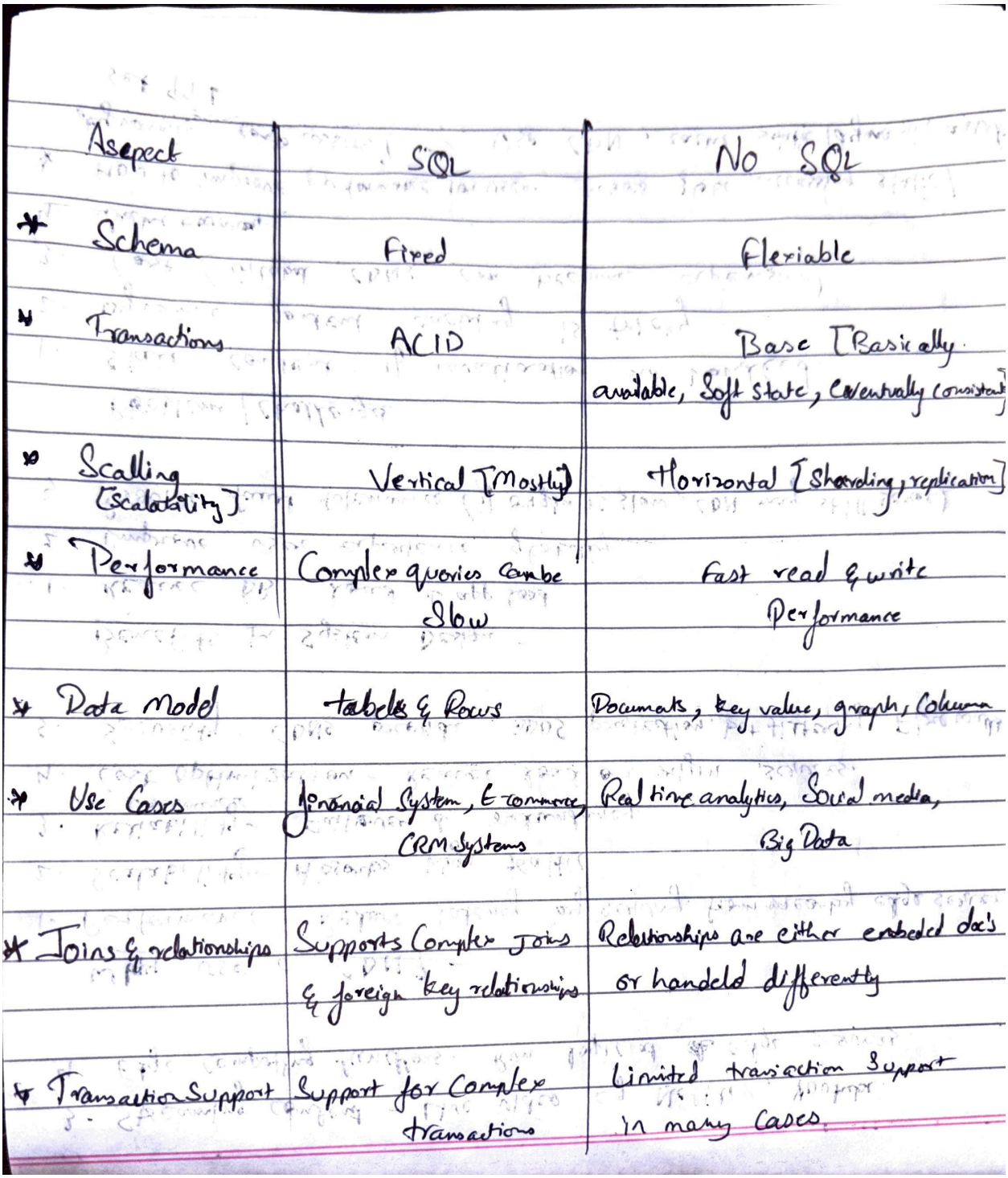
- Used in Load Balancing → domain resolves to different IPs (round robin, geo-based).
- CDN → DNS directs users to nearest edge server.
- Service Discovery → microservices may use DNS to find other services.
- Failover → DNS can redirect traffic to backup servers when one is down.

DNS Challenges

- Propagation Delay → DNS updates can take time due to caching.
- Single Point of Failure → If DNS server is down, whole app is unreachable.
- Latency → multiple lookups add time.

Databases

SQL vs No-SQL



Asepect	SQL	No SQL
Schema	Fixed	Flexiable
Transactions	ACID	Base [Basically available, Soft state, Eventually consistant]
Scalling [Scalability]	Vertical [Mostly]	Horizontal [Sharding, replication]
Performance	Complex queries can be slow	Fast read & write Performance
Data model	tables & Rows	Documents, key value, graph, Column
Use Cases	Financial System, E-commerce, CRM Systems	Real time analytics, Social media, Big Data
Joins & relationships	Supports Complex Joins & foreign key relationships	Relationships are either embeded doc's or handeld differently
Transaction Support	Support for Complex transactions	Limited transaction Support in many cases

ACID Transactions

ACID Transactions

ACID refers set of properties that are designed to ensure reliability & consistency of DB transactions even in case of failures.

1. **Atomicity** - • ALL or NONE - either fully complete or full rollback
   • e.g → A transfer 100 to B., so both credit/debit should happen or NONE
2. **Consistency** - • Transactions must leave DB in valid state.
   • Data follows all rules, constraints, triggers.
   • e.g → After debit/credit total money in system remains same
3. **Isolation** - • No 2 transactions should interfere.
   • Concurrent Txs behave as if executed sequentially.
   • e.g → 2 users book last ticket → only 1 gets it.
4. **Durability** - • Once Tx is comitted, it survives crashes / restarts
   • eg → If money is transferred & DB crashes, after recovery transfer must still exist

Advantages	Disadvantages
1. Data integrity	1. Performance overhead - Performance affected due to extra processing overhead req. by ACID.
2. Consistency	2. Deadlocks - Multiple Txs waiting can cause Deadlock
3. Isolation	3. Scalability - ACID Txs can be difficult to implement in large scale distributed systems.
4. Durability	

Alternatives to ACID

1. BASE (Basically Available, Soft state, Eventually Consistent)
   - Alternate models used by NO SQL
   - Basically Available - guarantees availability.
   - Soft State - Data may change over time, even without new input.
   - Eventually Consistent - Data may be stale but eventually will be C.
2. CAP
3. No SQL DB

Solutions for maintaining ACID in Distributed systems -

1. 2-Phase Commit. (2PC)
2. Multi-Version Concurrency Control (MVCC)
3. DB Replication (Master-Slave, Leader-Follower, Quorum)
4. DB sharding

Interviews

Mostly → SQL DB → ACID → CP (e.g Banking)
No SQL DB → BASE → AP (e.g Social Media)

Partitioning in DB and Sharding

Partitioning Database

DB Partitioning is splitting a large database table into smaller, more manageable pieces (partitions) to improve performance, scalability, & manageability. The data still represents a single logical table, but physically, its stored in multiple servers.

Why to Partition?

1. Performance - Queries can scan only relevant partitions instead of entire DB table.
2. Scalability - Distribute data across multiple servers.
3. Manageability - Easier backups, archiving & deletion.
4. Availability - Failure of 1 partition might not affect other partition

Types of Partitioning →

1. Range partitioning - Rows are stored based on value ranges of a column like by order_date P1 → Jan-Mar, P2-Mar-Jul.
2. Hash Partitioning - Using a hash function to decide partition.
   Eg: Partition_number = hash(user_id) % 4.
3. Vertical Partitioning - split columns into different tables / partitions. Like keeping ~~frequently~~ frequently used accessed columns together & move rarely used columns to another table, e.g USER_MAIN (userid, name, email) and USER_DETAILS (userid, D.O.B., profilepic)
   This is used when we want to reduce I/O & when only a subset of columns is queried often.

4. Horizontal Partitioning (Sharding) - Split rows of a table into different partitions/shard. Each partition/shard has the same schema but diff data ranges. E.g P1 contains user_id 1 - 1M and P2 contains user_id 1M+1 to 2M.
This is often used with Large datasets with independent access patterns.

a. Each chunk/partition is known as a "shard" & each shard has same db scheme as original db.
b. We distribute the data in such a way that each row appears in exactly 1 shard → consistency.

How sharding works?

1. Shard key selection - A shard key is a column that decides where a row goes. e.g user_id, region. Good shard key spreads data evenly & reduce cross-shard queries.
2. Shard Mapping - Based on shard key, the system decides the shard. It could be Range based (divides based on key ranges like ids 1-1M) or hash based (applying hash function) to decide the shard)
3. Routing - An application layer or a middleware router sends queries to right shard. eg tools like ProxySQL in MySQL.

Sharding challenges -

1. Choosing the right shard key - A bad choice of shard key can result in uneven loads ("hot shards").
2. Rebalancing shards - when one shard grows faster
3. Cross-shard joins - Can be expensive & slow.
4. Consistency - Distributed writes can introduce latency in eventual consistency systems.
5. Operational complexity - backups, monitoring & migrations are hard.

Master-slave in sharding (Replication) -

1. Each shard has its own db (with own data & schema)
2. Without replication, each shard is a single source of failure.
3. So, ~~data~~ each shard is replicated -
   a. Master - Handles writes & often some reads
   b. Slaves (Replicas) - Handles read-only queries & act as backup for failures

* Sharding solves scaling problems (too much data for 1 DB).
* Replication solves availability + read performance problems (single node failures & read heavy workloads).
* In a large distributed systems, both are used together.

Sharding Replication

Transaction Commits

Transaction Commits

2-Phase Commit (2PC)

When multiple DB/servers are involved in a Tx, how do we ensure All commit or All Rollback?

Actors –

1. Coordinator (Tx Manager) - Controls commit/rollback.
2. Participants (DB nodes/servers) - executes Txs.

Flow

Phase 1 : Prepare (Voting Phase)

1. Coordinator asks all Participants if they can commit?
2. Each Participants execute Tx - locking resources.
3. Participants reply → Yes - can commit, No - Cant commit

Phase 2: Commit / Abort

1. If all say Yes, Coordinator sends commit.
2. If any say No, " " rollback / abort.

Problems –

1. If Coordinator crashes after Phase 1, Participants stuck holding locks
2. Single Point of Failure - Coordinator.
3. Slow - Requires multiple network round trips.

Alternatives - 3PC, Consensus, Eventual Consistency (NoSQL).

3-Phase Commit (3PC)

Problem with 2PC → blocking. If coordinator crashes after Prepare but before commit, Participants are stuck holding locks.

Phase 1: Can Commit (Voting req)

- Coordinator asks - can you commit.
- Participants just say - Yes or No

Phase 2: Pre Commit

- If all say Yes, Coordinator sends PreCommit
- Participants - executes, locks resources, sends ACK.

Phase 3: Do Commit

- ~~All~~ all say ACK, ~~to~~ Coordinator sends DoCommit.
- Participants commit & release locks.

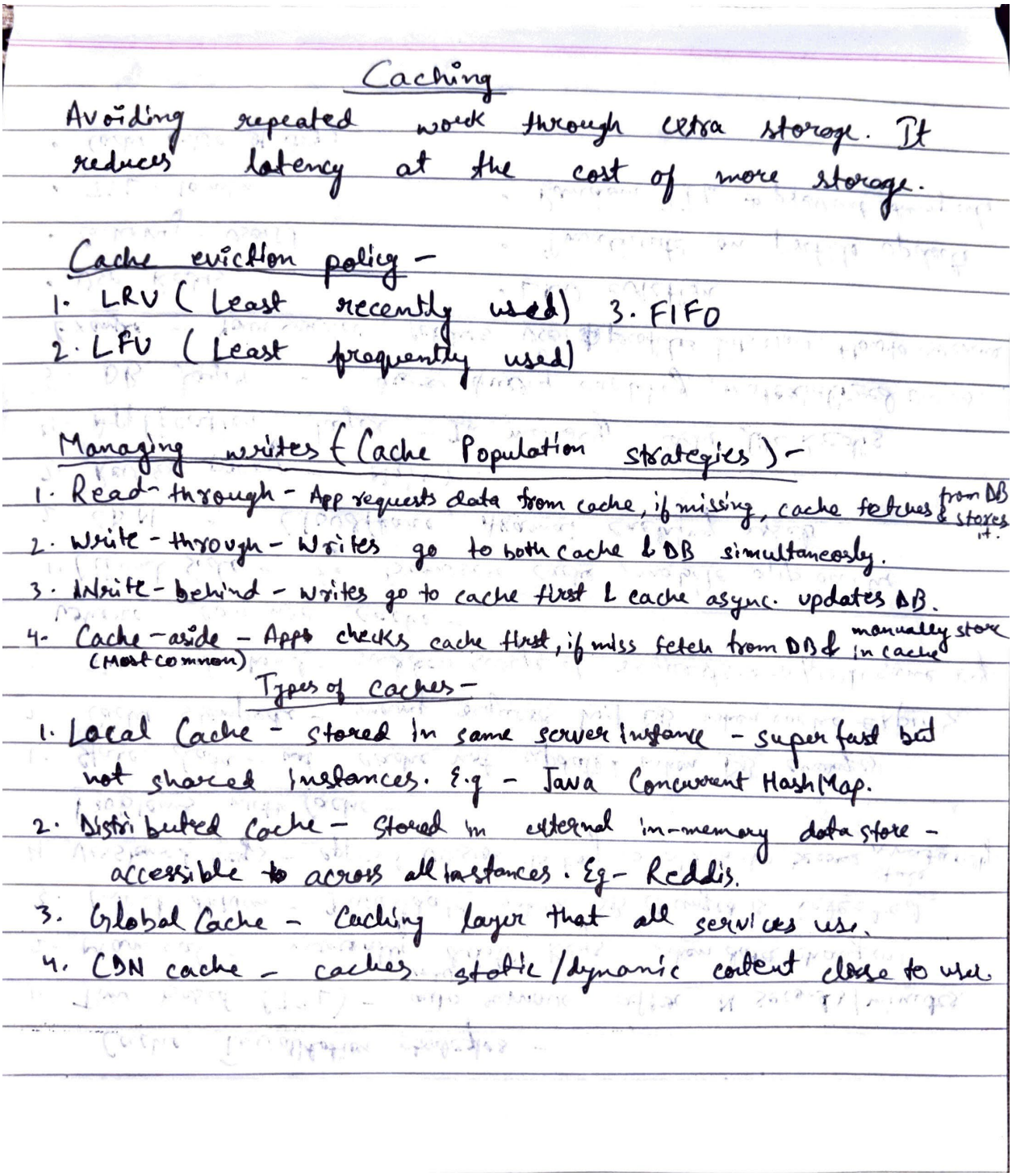
Problems -

1. Still not perfect.
2. If network partitions happen (split-brain scenario) some nodes may commit while others abort

Alternatives - Consensus Algos (Paxos or ~~Raft~~ Raft)

Caching and Performance

Caching



Caching

Avoiding repeated work through extra storage. It reduces latency at the cost of more storage.

Cache eviction policy -

1. LRU (Least recently used)
2. LFU (Least frequently used)
3. FIFO

Managing writes (Cache Population strategies) -

1. Read-through - App requests data from cache, if missing, cache fetches from DB & stores it.
2. Write-through - Writes go to both cache & DB simultaneously.
3. Write-behind - Writes go to cache first & cache async. updates DB.
4. Cache-aside (Most common) - Apps checks cache first, if miss fetch from DB & manually store in cache

Types of caches -

1. Local Cache - stored in same server instance - super fast but not shared instances. E.g - Java Concurrent HashMap.
2. Distributed Cache - Stored in external in-memory data store - accessible to across all instances. Eg - Reddis.
3. Global Cache - Caching layer that all services use.
4. CDN cache - caches static / dynamic content close to user.

Cache Invalidation strategies -

1. Time-based (TTL) - auto remove after N seconds/minutes.
2. Manual - manually (code) delete keys when data changes
3. Event-driven - Invalidate when DB changes is detected.
4. Versioned keys - append version to key so old cache become stale naturally.

Problems with Cache -

1. Stale data - cache not updated when DB changes
2. Cache stampede - many requests hit DB when cache expires.
3. Thundering herd - sudden surge of requests to refresh same key.

Where can we cache -

1. Client side - Eg Browser cache, mobile app cache
2. CDN - Cloudflare, Akamai caching assets
3. Reverse proxy - Nginx.
4. Application layer - In memory cache like Redis
5. DB layer - Query caching, materialized views.

Example - Your service fetches user profiles & it's slow. How to use cache

- Use Redis
- Cache key - userId
- TTL - 10 min
- Cache aside strategy
- LRU eviction
- Invalidate on profile update.
- Random TTL to prevent stampede.

CDN

CDN (Content Delivery Networks)

A CDN is a geographically distributed network of servers that deliver content to users from the nearest location. Goal is to → Reduce latency, increase availability

CDN concepts –

1. POP (Point of Presence) → Physical CDN Data Center location
2. Edge server → server in a POP that serves cached content.
3. Origin server → main application server (fallback when cache misses)
4. Cache invalidation → removing / refreshing outdated content.
5. Geo-replication → Keeping copies of content across globe.

How CDN works (Flow) –

1. User requests e.g example.com/image.png
2. DNS directs user to nearest CDN edge server.
3. If cache hit → edge server serves directly - faster.
4. If cache miss → edge server fetches from origin server, stores it & serves it.

What can a CDN cache

1. Static content – Images, CSS, JS, Videos, PDFs
2. Dynamic content – API responses (with TTL rules)

3. Streaming content - Live video e.g Netflix, Youtube.
4. Edge computing functions - Run logic at edge servers.

Why use a CDN? -

1. Performance - Reduce latency by serving from nearby edge server.
2. Scalability - Absorbs high traffic
3. Reliability - Failover & redundancy
4. Cost optimization - Reduce load on origin servers.
5. Security - CDNs provide DDOS protection, bot filtering, Firewalls

Benefits in System Design -

1. Reduce DB load & app load
2. Improve user experience globally
3. Provide fault tolerance (if origin is slow, CDN may still serve)

Problems / Challenges

1. Stale content if invalidation not handled.
2. Dynamic content caching is tricky
3. Cost (Global CDNs can become expensive)
4. ~~Cache warm~~

* How to improve performance for users across globe accessing static/dynamic ~~code~~ assets? → Use CDN, cache static/dynamic assets, set TTL

Rate limiter

A rate limiter controls how many requests a user/system can make in a given time window.

- Prevents abuse (e.g., spam, brute-force attacks).
- Protects backend services from being overloaded.
- Ensures fair usage across users.

Common Algorithms used for Rate limiting -

1. Fixed Window Counter
    a. Divide time into fixed intervals (e.g., 1 min).
    b. Keep a counter for each user → reset every window.
    c. Pros: Simple, memory efficient.
    d. Cons: Can allow bursts at window boundaries.
    e. Example: Limit = 5 req/min. User makes 5 requests at 12:00:58
2. Sliding Window Counter
    a. Like fixed window, except for the starting point of each window.
    b. With sliding window, time frame only starts when a new request comes in, not a predetermined time.
    c. Pros: handles variable trasffic pattern better.
    d. Cons: complex & required more memory and computation.
3. Token Bucket
    a. Bucket holds tokens (capacity = limit).
    b. Tokens added at a fixed rate (e.g., 5 per second).

c. Each request consumes a token.

d. If request has no tokens → reject/throttle.

e. Pros: Allows burst traffic up to bucket size, then enforces steady rate.

f. Example: Bucket = 10 tokens, refill = 1/sec. Can burst up to 10 requests instantly. Then max 1 request/sec.

4. Leaky Bucket

a. Similar to token bucket, but requests processed at a fixed outflow rate.

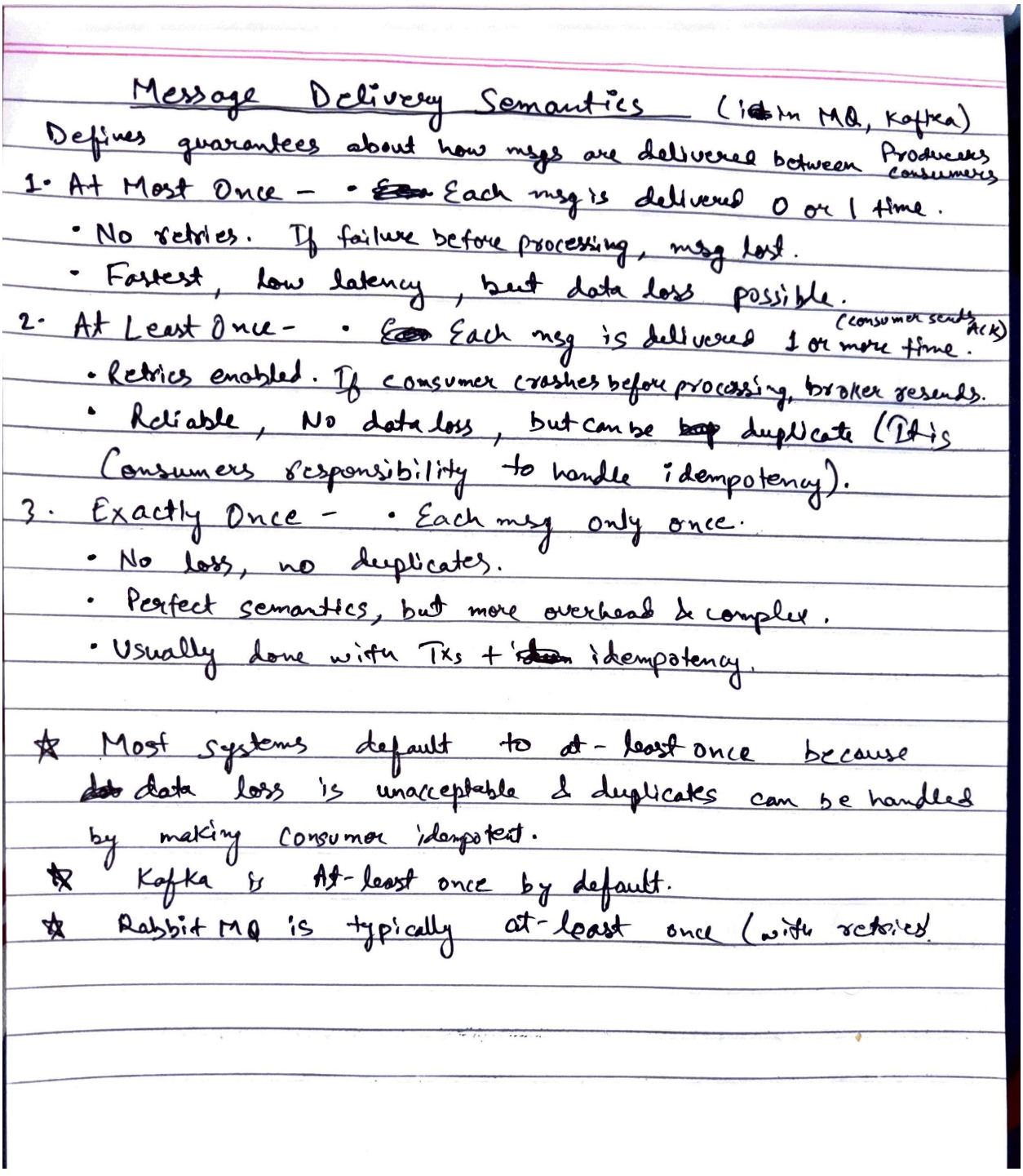
b. Excess requests → dropped or queued.

c. Pros: Ensures constant rate (good for traffic shaping).

Token bucket is most commonly used in industry (e.g., Google, AWS)

Messaging and Streaming

Message Delivery Semantics



Message Delivery Semantics (in MQ, Kafka)

Defines guarantees about how msgs are delivered between Producers consumers

1. At Most Once – • Each msg is delivered 0 or 1 time.
   - No retries. If failure before processing, msg lost.
   - Fastest, low latency, but data loss possible.
2. At Least Once – • Each msg is delivered 1 or more time. (consumer sends Ack)
   - Retries enabled. If consumer crashes before processing, broker resends.
   - Reliable, No data loss, but can be duplicate (It is Consumers responsibility to handle idempotency).
3. Exactly Once – • Each msg only once.
   - No loss, no duplicates.
   - Perfect semantics, but more overhead & complex.
   - Usually done with Txs + idempotency.

★ Most systems default to at-least once because data loss is unacceptable & duplicates can be handled by making consumer idempotent.

★ Kafka is At-least once by default.

★ Rabbit MQ is typically at-least once (with retries)

Messaging Queues

Messaging Queues

It is a communication mechanism where services send or receive messages asynchronously. Impt points -

1. Decoupling - Sender does not need to know who/when will process the request.
2. Async processing.
3. Load Management
4. Reliability | Persistence - Even if service goes down, MQ still has the message.

MQ models -

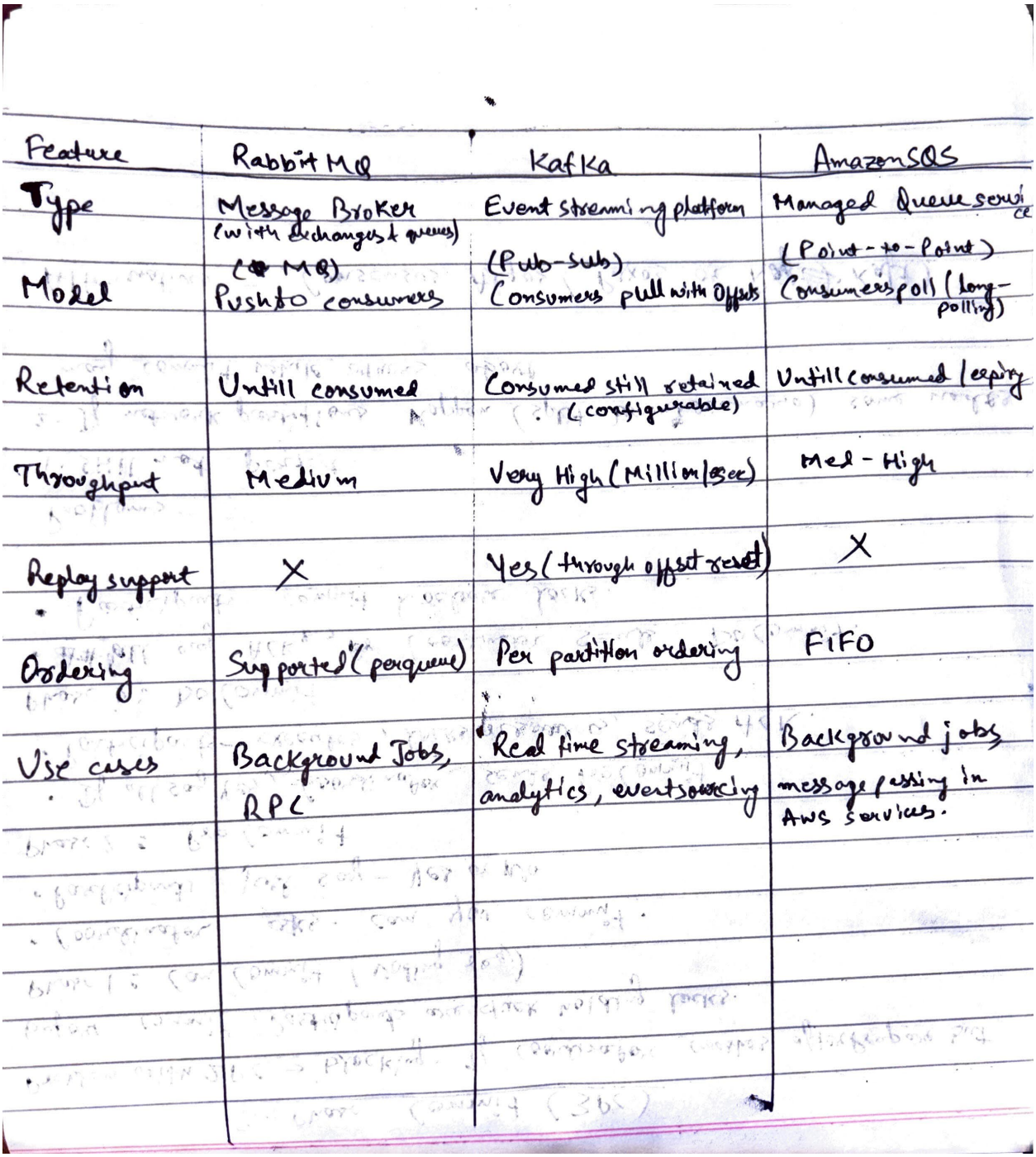
1. Publish - Subscribe (Topic model) →

   One message → delivered to all subscribers.

2. Point to Point (Queue model)

   One message → delivered | consumed by exactly 1 consumer.

Rabbit MQ vs Kafka vs Amazon SQS



Feature	Rabbit MQ	Kafka	Amazon SQS
Type	Message Broker (with exchanges + queues) (e.g. MQ)	Event streaming platform (Pub-Sub)	Managed Queue service (Point-to-Point)
Model	Push to consumers	Consumers pull with offsets	Consumers poll (long-polling)
Retention	Until consumed	Consumed still retained (configurable)	Until consumed / expiry
Throughput	Medium	Very High (Million/sec)	Med - High
Replay support	X	Yes (through offset reset)	X
Ordering	Supported (per queue)	Per partition ordering	FIFO
Use cases	Background Jobs, RPC	Real time streaming, analytics, event sourcing	Background jobs, message passing in AWS services.

Batch Processing vs Stream Processing

Feature	Batch Processing	Stream Processing
Data Processing	Processes a large volume of data at once.	Processes data as it arrives, record by record.
Latency	High latency, as processing happens after data collection.	Low latency, providing near real-time insights.
Throughput	Can handle vast amounts of data at once.	Optimized for real-time but might handle less data volume.
Use Case	Ideal for historical analysis or large-scale data transformations.	Best for real-time analytics, monitoring, and alerts.
Complexity	Relatively simpler to implement with predefined datasets.	More complex, requires handling continuous streams.
Data Scope	Operates on a finite set of data.	Operates on potentially infinite streams of data.
Error Handling	Errors can be identified and corrected before execution.	Requires real-time handling of errors and failures.
Resource Usage	Resource-intensive during processing, idle otherwise.	Continuous use of resources.
Cost	Cost-effective for large volumes of data.	More expensive due to continuous processing.

Reliability and Recovery

Distributed Tracing

Distributed Tracing is a method to track a single request as it flows through multiple microservices in a distributed system. It helps answer:

- Where did the request spend most of the time?
- Which service is slow?
- Where did the failure happen?

Why Do We Need It?

- In monoliths, debugging is easy because everything is in one process.
- In microservices, a single request may pass through 10–20 services. Without tracing, finding bottlenecks is very hard.

How It Works

1. Unique Trace ID
    a. Every request is assigned a trace ID (e.g., UUID).
    b. This trace ID is passed along all services handling the request (usually via HTTP headers).
2. Span
    a. Each service creates a span → representing its work (start time, end time, metadata). Example: PaymentService span = 200ms.

3. Trace
    a. A trace = collection of spans across services for one request.

Disaster Recovery (DR)

Disaster Recovery is the strategy and set of processes a system uses to recover and continue operations after a catastrophic failure. Failures could be due to: Natural disasters (earthquake, flood, fire), Data center outage, Network failure, Cyber attacks or accidental data deletion.

Key Metrics in DR

1. RPO (Recovery Point Objective) - Maximum acceptable data loss (measured in time). Example: RPO = 5 mins → system should be able to restore data up to 5 mins before the failure.

2. RTO (Recovery Time Objective) - Maximum acceptable downtime after a disaster. Example: RTO = 1 hour → system must be back online within 1 hour.

Disaster Recovery Strategies

1. Backup & Restore
    a. Regularly back up data to remote storage.
    b. Restore when disaster occurs.
    c. Pros: Cheap.
    d. Cons: Very high RTO (slow recovery).
2. Pilot Light

    a. Keep a minimal version of your system running in another region (only core services).

    b. In case of disaster, scale it up quickly.

    c. Balance between cost and recovery speed.

3. Warm Standby

    a. Keep a smaller but fully functional copy of the system running in another region.

    b. It handles some traffic but can scale up during disaster.

    c. RTO and RPO are moderate.

4. Hot Standby / Active-Active

    a. Two (or more) full systems run in parallel in different regions.

    b. If one goes down, traffic is instantly redirected.

    c. Pros: Very low RTO and RPO.

    d. Cons: Expensive.

Techniques Used in DR

1. Data Replication (synchronous vs asynchronous).
2. Failover Mechanisms (automatic DNS switching, load balancer failover).
3. Geo-Redundancy (multiple regions).
4. Chaos Engineering (test recovery by intentionally causing failures).

Important Links –

https://www.linkedin.com/posts/ashishps1_how-i-would-learn-system-design-fundamentals-activity-7181881895985528832-OA7Q?utm_source=li_share&utm_content=feedcontent&utm_medium=g_dt_web&utm_campaign=copy