System Design :

Applying CS concepts (networks, distributed systems, parallel computing) to build large scale systems.

1) High Availability — requests should not fail
2) Scalability — handle increasing workload
3) Consistency — same information across all servers
4) Fault tolerance — function despite component failures
5) Load distribution — evenly spread traffic / workload
6) Extensibility — easy to add / modify components
7) Performance — speed and efficiency

Primers —

- Vertical Scaling
- CRON jobs
- Resilience and Backups (Master-Slave Architecture)
- Horizontal Scaling
- Microservices
- Distributed Systems
- Load Balancing
- Decoupling
- Logging and Metrics (Analytics, Auditing, Reporting, ML)
- Extensibility
- High-level Design
- Low-level Design

Code on computer = takes input → output

To share : expose code via API

Request → input sent by user

Respons → output returned by system

Problems with self hosting

- Requires database setup and endpoint configuration
- Risk of downtime

Solutions is Cloud

- Cloud = provider's computers rented (eg. AWS)
- Offers computing power, reliability, configuration management
- Remote login allows hosting services on cloud

Scaling for growth

1) Vertical Scaling → bigger machines (more CPU, RAM)
2) Horizontal Scaling → more machines, load distributed

Vertical Scaling	Horizontal Scaling
• Fast (interprocess communication)	• Slow (network calls, remote procedure calls)
• Consistent data (single source of truth)	• Data consistency issues (need loose guarantees)
• Single point of failure	• Resilient (failure handled by other machines)
• Hardware limit	• Scales well (linear with more servers)
• Load balancing not required	• Load balancing required

Real World Solution (Scalability) – Hybrid Approach

- Start with vertical scaling
- Move to horizontal scaling as users grow
- Vertical → speed + consistency
  Horizontal → scalability + resilience

Server - Client :

→ Server = process requests
→ Client = sends requests
→ Respons = processed result

Scaling Problem :

1) One server → fine for small load
2) Many ~~clients~~ → multiple servers needed

- Load Balancing
- Evenly distribute requests across N servers.
- Simple method : hash (request ID) % N
- X requests → X/N load → Load factor : 1/N
- Works well initially

Problem with Naive Hashing

- Adding / removing a server changes N.
- Causes remapping of almost all requests → cache invalidation & instability
- Ex : Going from 4 server to 5 → nearly 100% requests remapped
- Request stickiness : same user ID should map to same server for caching
- Naive re-hashing → breaks cache locality → wasted computations and poor performance.

Consistent Hashing

It ensures that when servers are added / removed, only a small fraction of requests are remapped, preserving cache locality and balancing load efficiently.

Instead of remapping all requests:

1) Only a small portion of requests get redistributed when servers are added / removed.
2) Each server owns a segment of a hash ring (0...M-1)
3) Adding a new server = steals small slices from all existing servers (not full redistribution).

How it works:

- Hash Requests → map requests IDs to positions on ring
- Hash Servers → map server IDs via hash (% M) to ring positions
- Request Mapping Rule → each request is served by the nearest server clockwise on the ring

Practical Problem

- With two servers → skewed distribution possible
  Ex: one server may take ~50% load if unlucky placement
- Theoretical avg = 1/N, but variance is high.

Solution → Virtual Servers

- Assign multiple points per server on the ring.
- Implementation → use K different hash functions per server.
- Each server now appears multiple times → load spreads out
- If K ≈ log N, load distribution ≈ uniform
- Adding / removing a server → only fraction of requests move
- No single server ends up overloaded.

Consistent Hashing + Virtual Servers = Scalable, fault tolerant, and evenly distributed request mapping with minimal remapping.

Messaging Queues / Task Queues

- Producers (clients) send tasks / requests.
- Consumers (servers / workers) process them.
- Requests are placed asynchronously → producers gets an acknowledgment, not the result immediately.
- Tasks go into a queue and are processed in order or based on priority.

Benefits of Asynchronous Processing

1) Clients don't wait → can do other tasks
2) Server can prioritize tasks
3) Better resource utilization + smoother client experience.

Scaling

- Multiple servers can pull from queue
- If one server fails:
  - Non persistent tasks are lost
  - Persistent tasks can be redistributed to healthy servers.

Fault Tolerance

- Simple in-memory queues fail if a server crashes → persistence is required
- Queue systems store metadata in Database such as:
  → Task ID
  → Payload
  → Status (pending, in-progress, completed)

- Mechanism : checks if server alive (eg. ping every 15)
  → If dead → redistribute unfinished orders to other servers

Problem → Duplication
- Risk of multiple servers picking same task.
- Solution → Load Balancing (eg. Consistent Hashing) ensures:
  1) Balanced load
  2) No duplicate requests processed.

Message Queue = Encapsulation of Complexity
Features combined into one system:
1) Task persistence (Database)
2) Assignment to correct servers
3) Heartbeat monitoring (mechanism done by notifier)
4) Load balancing + retry if server fails
5) Duplicate avoidance

Message Queue is a fault tolerant, asynchronous, persistent, load balanced task handler that encapsulates all the complexity of distributed request processing.

Ex:-
- RabbitMQ
- ZeroMQ
- JMS (Java Messaging Service)
- AWS SQS / SNS

Monolith vs Microservices Misconceptions

- Monolith ≠ one big machine → can scale horizontally across multiple servers + Databases
- Microservice ≠ tiny code → it's a business unit (data + functions) bundled together
- Microservices may be few (eg. 3-5), not "hundreds of tiny ones".
- Clients usually talk via an API Gateway, not directly to services.

Monolith :

Advantages -

1) Small / Cohesive Teams → simpler to manage
2) Less moving parts → easier deployments
3) Shared setup code (connections, configs, tests) → no duplication
4) Performance → in-process calls (no network latency)

Disadvantages -

1) Steep learning curve for new devs (need full context)
2) Coupled deployments → any change = redeploy entire system
3) Complex testing → everything touches everything
4) Single point of failure → crash = whole system down

Microservices :

Advantages -

1. Scalability → scale services independently
2. Easier onboarding → devs only need service context
3. Parallel development → teams work independently
4. Targeted scaling → allocate resources to hot services only.

Disadvantages –

1. complex design → needs strong architecture
2. Over-splitting risk → if two services only talk to each other, they should probably be one
3. RPC overhead → network calls slower vs local calls.

Monlitho = simpler, faster, good for small teams (eg StackOverflow)
Microservice = scalable, modular, fault-tolerant, but need smart design (eg. Google FB)

Cache

Caching = storing data temporarily in fast-access memory to avoid repeat computations / queries

Goal → Reduce latency (response time) (fast)
Save repeated computation or DB lookups (reduce load on DB)
Improve performance, user experience and scalability

Feed Fetching Flow

- Without cache : User → Server → DB query → Server → User
- With cache : similar queries (cohorts of users) can be served from cache instead of quering DB each time
- Result : Latency drops from 220 ms to 202 ms ~10% saving

Why Cache ?

→ Reduce latency : Faster retrieval compared to recompoting or fetching from DB
→ Reduce load on databases / servers : serve repeat requests from memory
→ Improve user experience : Faster responses
→ Enable scalability : same system can handle more requests

Cache Placement

1) Client-side cache (eg. app/browser stores data locally)
   - Saves network calls
2) Server in-memory cache (within the application)
   - Fastest, lives alongside app (eg. HashMap in RAM)
   - But limited to a single server
3) Database cache (built into DB)
   - Common queries auto-cached internally
   - Usually opaque to developers
4) Distributed / Global cache (dedicated caching system)
   - Independent service (eg. Redis, Memcached)
   - Shared across multiple services
   - Can scale independently; update without redeploying main app

Large-scale systems usually use all of them together.

Cache Policies

1. Write Policies (when to update cache vs DB)
   → Write through : Update DB + cache simultaneously
   → Write back/behind : Update cache immediately, write to DB later
   → Write around : Write only to DB, load into cache only on reads
2. Eviction Policies (what to remove when cache is full)
   → LRU (Least Recently Used) - remove oldest accessed item
   → LFU (Least Frequently Used) - remove least accessed item
   → FIFO (First In First Out) - remove oldest inserted item
   → Random / ML based - advanced adaptive methods

Limitations & Trade Offs

1) Size Limits
- Cannot fit entire DB in memory
- Must choose hot / frequently accessed data

2) Cache Miss & Poor Hit Rate
- If queries don't match cached data → fallback to DB
- Leads to wasted lookup + DB call → overhead

3) Thrashing
- Frequent evictions and reloads
- Causes higher latency + wasted memory usage

4) Stale Data / Eventual Consistency
- Cache is copy of DB → may not always be updated in real time → depends on cache update strategy
- Acceptable for views / likes, risky / dangerous for critical data

Single Point of Failure :
SPOF = A single component whose failure crashes the entire system
Ex: One database server → if it fails, whole system fails.

- Tests resiliency of architecture
Non resilient system = multiple services relying on one fragile point

Mitigation Strategies :
1) Add Redundancy / More Nodes
- Duplicate servers (profile servers, APIs)
- Not useful as "cold backups" unless data is replicated

2) Database Replication (Master - Slave)
- Writes → Master
- Reads → Slaves
- Failover → promote slave if master fails
- Failure probability reduces from $p \rightarrow p^2$

3) Load Balancers
- Distribute requests
- But load balancer itself = SPOF
- Solution → Multiple load balancers behind ~~SPOF~~ DNS

4) DNS with Multiple IPs
- Clients resolve same host (eg. facebook.com) to multiple IPs (different load balancers)

5) Multi Region Deployment
- Avoid full system failure from regional disasters
- Active - active / Active - passive regions

6) Coordinator / Leader Election
- Avoid SPOF in coordinators using similar replication / redundancy strategies.

Ex: Netflix Chaos Monkey → randomly kills nodes in production to test resiliency

Build pipeline of distributed, fault tolerant system

Content Delivery Network :

CDN = Geo-distributed cache servers that store static content (HTML, videos, images, files) close to users

Purpose → Faster, cheaper, and regulation-compliant content delivery

Why CDN is needed — Central Cache Limitations
→ High latency for users far from server (eg. India vs Japan vs US)
→ No single server location suits all → always slow for some region
→ Local data regulations (eg. movies allowed only in India)

Working

- Content is cached in local edge servers around globe
- Users connect to the nearest CDN node instead of central server
- CDN servers = real servers with file system, managed by origin server.

Benefits

1. Low latency → faster webpages loads → higher trust & professionalism (Amazon/ Google)
2. Compliance → Server region -- specific content per regulations
3. Relevance → local caches can store popular / regional data only

CDNs Providers

- Large cloud companies (hard to build your own)
- EX: AWS CloudFront
  - cheap, reliable, integrated with S3
  - Auto-triggered updates when new files added
- Ex: GCP, Azure CDNs

Event Driven Services

Problem with Request - Response in Microservices

- In microservices, one service often needs to trigger actions in multiple others.

1) Services wait for each other → increased latency
2) If one service fails → cascading timeouts
3) Can lead to stale or duplicate data updates
4) Hard to scale when new services need the same events.

Publisher - Subscriber Model

- Services publish events to a Message Broker (Kafka, RabbitMQ).
- Other services subscribe to the events they care about.
- Broker delivers to subscribers asynchronously
- Fire-and-Forget : Publisher doesn't care who consumes, Broker guarantees delivery

Flow :

Client → Service → Broker → Subscribers

Advantages

1. Decoupling → Services don't know subscribers
2. Reliability → Broker persists and replays if subscriber down
3. Easier Dev → One generic interface for publishers
4. Transaction Guarantee → at least once delivery
5. Scalability → Easy to add new consumers
6. Simplified architecture → Fewer direct dependencies

Drawbacks

1) No Strong Consistency / Atomicity
   - Problematic for financial systems as they require atomicity
   - Cross service transactions can leave inconsistent balances
2) Idempotency Issues
   - Same message replayed → duplicate operations
   - Needs RequestID + app logic to enforce idempotency
3) Not fit for mission critical systems needing strict success/failure guarantees.

Use Cases

→ Good : Analytics, gaming events, social feeds (Twitter)

→ Not good for : Banking, fund transfer, payments

Request Response → Pull based

- A service asks another for something, waits for the response
- Tight coupling, dependencies, timeout issues, stale data risk

Event Driven Architecture → Push based

- A service publishes an event: "something happend"
- Subscribers decide if it matters to them
- Loosely coupled, scalable, more resilient
- Flow: Producer → Event Bus → Consumers

EX:

Git → commits as events

React / Node.js → UI & server-side event driven framework

Gaming → headshots validated by replaying events with timestamps

Advantages

1. Decoupling - services don't depend directly on each other
2. Availability - services persist relevant events locally, system still works even if others are down
3. Event Log (History) - allows debugging, replays, and state restoration
4. Smooth Replacements - new service can replay past events to sync with old state
5. Transactional Guarantees -
   - At most once (optimal, eg. welcome email)
   - At least once (critical, eg. invoice email)
6. Scalable - easy to add new consumers without changing producers
7. Storing Intent - keep why a change happened, useful for future services.

Drawbacks

1. Consistency Issues - replicated local data may get outdated
2. External calls problem - replaying past events may yield different results
3. Lack of fine control - can't easily enforce timing, order, or exact consumers
4. Security Concerns - hard to limit who consumes events
5. Event Replay Complexity :
   - Replay from start (impractical at scale)
   - Diff-based (store changes only)
   - Undo logs (limited, not always possible)
   - Compaction (squash old events into snapshots)
6. Hard to Reason About Flow - tracing event propagation is difficult for developers
7. Vendor Lock In - moving back to request-response isn't smooth

SQL (Relational DBs)

- Data Model → Tables, rows, columns
- Relationships → Foreign keys, joins
- Schema → Fixed (rigid, hard to change)
- Consistency → Strong → ACID (Atomicity, Consistency, Isolation, Durability)
- Best For → Financial systems, transactions, structured data, strong consistency
- Ex: MySQL, PostgreSQL, Oracle

NoSQL (Non-Relational DBs)

- Data Model → Document (JSON), Key-Value, Wid-Column, Graph
- Relationships → Not implicit (joins hard/expensive)
- Schema → Flexible (can add/remove attributes anytime)
- Consistency → Eventual (sacrifies strict ACID)
- Best for → scalability, flexible schema, high write throughput, analytics, caching
- Ex: Cassandra, MongoDB, DynamoDB, Redis

Advantages

1) Single Blob Access → Insert/retrive whole object at once
   → Great for INSERT + SELECT * type use cases
2) Flexible Schema → Easy to add/remove fields (no costly table ALTER).
3) Horizontal Scalability → In built sharding/partitioning
   → Handles huge scale & availability
4) Aggregation Friendly → Optimized for metrics and analytics

Disadvantages

1. Weak Consistency → No guaranteed ACID → bad for transactions (UPDATE = DELETE + INSERT)
2. Slow for Reads → Reads entire blob even if only 1 column needed
3. No Implicit Relations → Hard to enforce foreign keys
4. Joins are Manual → Expensive and error prone

Use :

- Large scale apps needing availability and writes-heavy workloads
- Flexible schemas, fast evolving products
- Analytics, aggregations, redundancy

Cassandra Architecture

1. Cluster of Nodes → Data spread across multiple machines
2. Hashing (Consistent Hashing) →
   Keys → Hashed → Assigned to nodes
   - Good hash → equal load distribution
   - Bad hash → imbalance → solved via multi level sharding
3. Replication → Each piece of data stored in multiple nodes
   - Improves availability + fault tolerance
4. Load Balancing → Queries routed evenly across nodes
5. High Availability > Strong Consistency

Problem : Replicas may not be in sync

- Replication Factor (RF) → no. of copies of data stored
- Quorum → majority must agree on a value (latest timestamp wins)
  (Eg. RF = 3, Quorum = 2 → 2/3 nodes must agree on value)
  - Ensures eventual consistency
  - Accepts small risks of stale reads for high availability
- Distributed Consensus → Nodes agree on latest value
  Methods : Quorum, Gossip protocol, Paxos, etc.

Data storage :

- Writes go to in-memory structure (MemTable)
- Writes stored in log-structured merge (LSM) style
- First in memory (memtable) → flushed to disk as SSTables (~~Stored~~ sorted String Tables)
- Periodic Compaction → Merges SSTables → improves read efficiency
- Benifits
  - Writes are fast (sequential log appends)
  - Reads slightly slower (need to check multiple SSTables)

Application Programmable Interface (API)

→ A contract that defines how to interact with code, not how it works

→ Defines :

- Function name → what it does
- Parameters → what it needs
- Response object → what it returns
- Possible errors

→ Ex: getAdmins (groupID) → returns list of admins (JSON objects)

Best Practices

1. Placement → belongs in right service (eg. group service)
2. Naming → must match the action (getAdmins should return only admins)
3. Parameters → keep minimal, only what's necessary. extra params only if required for optimization

4. Responses → Return only what's needed
   · don't overload with unnecessary info
5. Errors → Define expected errors (eg. grp not found)
   · avoid overly generic or overly detailed error handling

HTTP endpoints :

- Path : https://site.com/groups/admins?groupId=123
- Versioning : /v1/ for backward compatibility
- Routing → clear and versioned (/groups/v1/admins vs action = getAdmins)
- Use correct HTTP verbs :
  GET → fetch
  POST → send data / add
  DELETE → remove
- Don't duplicate action in URL (avoid /getAdmins)

Common Pitfalls

1) Side Effects
   - API should do one thing only
   - Avoid hidden operations (eg. setAdmins also creating grps)
   - Break into smaller APIs if multiple actions
2) Atomicity
   - Handle dependent operations separately (creategroup then setAdmins)

Handling Large Responses

- Pagination → client requests in chunks (offset, limit)
- Fragmentation → break response into multiple packets (service-to-service)

Data Consistency

- Decide how consistent results need to be:
  Strong consistency → accurate but slow
  Eventual consistency → faster, may serve stale data
- Use caching for high-read APIs (comments, profile lookups)
- Grateful degradation: send minimal info (name, ID) under heavy load instead of failing.

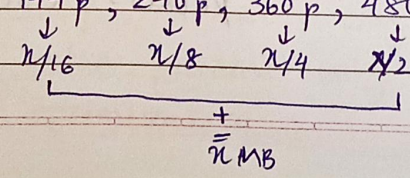
A good API is

→ Intuitive (clear name, right params)
→ Precise (only needed data)
→ Predictable (no side effects)
→ Resilient (handles expected errors & scales with pagination, caching, degradation)

Capacity Estimation:

Storage Estimation (Youtube)

1) Assumptions
   - 1B users → 1 in 1000 uploads → 1M uploads/day
   - Avg video length ≈ 10 mins, 2 hr → 4 GB → 0.4 GB (low quality) (400 MB → 2 hr, 200 MB → 1 hr, 3 MB → 1 min)
   - Video size ≈ 3 MB/min → 30 MB/video
2) Raw Storage
   - Total = 1M × 30 MB/video = 30 TB/day
3) Redundancy
   - 3 copies → 90 TB/day
4) Multi-resolution storage
   - Store in 144p, 240p, 360p, 480p, 720p versions



- Storage ≈ 2X original (x MB + x MB = 2x MB)
- Total ≈ 180 TB ≈ 0.2 PB / day
- Since for 0.4 GB → 0.2 PB/day, 4 GB → 2 PB/day (high quality)

→ How high quality → low quality (Assumption)
2hr → 4GB
optimised code and compression savings = 90%
2 hr → 4GB / 10 GB = 0.4 GB

Metadata Caching Estimate :
- Thumbnails (images, heavier)
- Titles / IDs (lightweight)

Assumption
- Original → 1MB
- Compressed → 100 times → 10 times from both sides
- Display size → 1 MB / 100 ≈ 10 KB
- Cache only popular videos → 90 days last uploads
- 10 KB X 90 days X 1M uploads/day ≈ 10^9 KB → 1 TB RAM

Deployment
- 16 GB servers
- 1 TB → 1000 GB / 16 GB → 64 nodes → cache system
- Redundancy (3 copies) + 50% capacity
  64 X 3 X 2 → 128 X 3 ≈ 500 nodes

Video Processing Power Estimate :
→ Convert uploads / day → MB/s
- Upload volume = 10^7 min / day (10 min X 1M uploads/day)
- Density = 3 MB / min
- Data / day = 10^7 X 3

Video Processing Estimation :

1) Upload volume / day
   - 10^7 min / day → (10 min × 1M uploads /day) → 166,667 hr/day
2) Storage
   - Assume video size density = 1 GB / hr
   - Raw storage / day = 166,667 × 1GB ≈ 166 TB
3) Convert to Throughput
   - Data / day = 166667 × 1024 ≈ 2×10^8 MB / day
   - Seconds / day = 86,400 ≈ 100,000
   - Throughput = $2 \times 10^8 / 10^5$ ≈ 2000 MB/s
4) Processing cost / MB
   - Read = 10 ms
   - Process = 20 ms
   - Write = 20 ms
   - Total = 50 ms / MB = 0.05 s / MB
5) Work / sec
   - 2000 MB/s × 0.05 s/MB ≈ 100 processors

   We need 100 processors in parallel
6) Redundancy + Multiple Resolution (optional)

   100 × 3 (copies) × 2 (resolution) ≈ 600 processors

Log Structured Merge Tree

Motivation –

- Traditional DBs use B+ Trees
  - → Insert = O(log n)
  - → Search = O(log n)
- Issue : Too many small writes → lots of disk I/O + acknowledgments

Goal –

Optimize writes while keeping reads fast enough

Step 1 : Write Optimization

- collect writes in memory → batch them
- Append to a log-like structure (linked list) → O(1) write
- Flush to disk in bulk (reduces I/O + ACK overhead)
- Problem → reads slow → linked list = sequential search → O(n)

Step 2 : Sorted String Table (SSTable)

- Before flushing to disk → sort records → chunks
- Persist sorted arrays (SSTables)
- Reads → Binary search → O(log n) on each SSTable.

Step 3 : Merge SSTable

- Multiple sorted chunks = multiple binary searches
- Solution → Compaction → merge smaller SSTables into bigger sorted SSTables → similar to merge sort → reduce no. of SSTables to search.

Step 4 : Faster Lookups

- Attach Bloom filter per SSTable → chunk on top
- Helps decide if a key might exist in an SSTable.
- Avoids unnecessary reads
- Trade-offs → false positive possible → false negatives impossible.

Bloom Filter

1. Bit array of size M (all 0 initially)
2. Insert string → pass through K hash functions → set bits at K positions = 1
3. Query string → check K positions. (no false negative possible, false positive possible)
   - If any 0 → definitely not present
   - If all 1 → probably present

Accuracy drops if array too full → reset / rebuild needed

Error probability $(1 - (1 - \frac{1}{M})^{Kn})^K$

n = items, M = array size, K = hash functions

Master Slave Architecture

→ Problem

- Typical system : Multiple servers + one database
- Single DB = single point of failure (SPOF)
- If DB crashes → business stops

→ Solution

- Replicate database (copy stored elsewhere, iddy separate hardware).
- Provides backup + fault tolerance.

Types of Replication :

1) Asynchronous
   - Master updates, then later copied to slaves
   - Lower load on master

Problem → Slave can be out of sync → data loss if master fails

2) Synchronous
   - Master update must be copied to slave before commit
   - Strong consistency

Problem → Higher latency → extra network round-trip

Master - Slave Roles :

- Master = accepts writes + reads
- Slave = accepts reads only (no writes)
- Writes always go to Master → replicated to Slaves

Master - Master Roles (Peer - Peer) :

- Both nodes can accept writes + reads
- Load balancing of writes + fault tolerance
- Split Brain Problem
  → If network partition happens, both think they're master
  → Leads to inconsistent data (double deductions, etc.)
- Solution

→ Add third node (arbiter / consensus node)
→ used for distributed consensus → decides which side is valid
→ Ensures only one master state is accepted.

Consensus Protocols
- 2PC (Two Phase Commit) - simple but slow
- 3PC - improvement but still heavy
- MVCC (Multi-Version Concurrency Control) - keeps multiple versions of data, avoids blocking, used in PostgreSQL
- Saga Pattern - long transactions broken into smaller steps with rollback capability

Why use Master-Slave?
1. Backup / Fault tolerance : reduces SPOF risk
2. Read scalability : multiple slaves for read-heavy workloads like analytics
3. Eventual consistency acceptable : for non critical apps
4. Sharding + Master Slave → scale writes by partitioning data

Location Based Databases
- Used in Google Maps, Swiggy, Zomato, Uber, Dating apps
- Requirements
  1. Distance Measurement → find distance between 2 points
  2. Proximity Search → find all points within X km radius
- Early Approach
  - Assign codes (eg. ZIP, PIN) to regions
  - Problem → Not accurate → geography like railways, rivers may cause long detours.

1) Better Representation → Coordinates :

- Use Latitudes & Longitude
- Pros
  - → Scalable granularity (more decimals → more precision)
  - → Easy distance computation (Euclidean / Haversine formula)
- Cons
  - → Naive proximity search : expensive (check all DB points)

2) Datastructure for Location Search :

→ Quad Trees

- Recursive partitioning of 2D space into quadrants
- Can adaptively split based on density (eg: Mumbai → deep, Iceland → shallow)
- Problem → range queries in 2D are expensive

→ Solution : Reduce 2D to 1D

- Range queries are efficient in 1D (segment trees, interval trees).
- Convert 2D spatial data into 1D line while preserving proximity

3) Fractal curves for 2D → 1D Mapping :

- Z-curve : Simple, but poor locality preservation
- Hilber curve :
  - Space filling curve mapping 2D → 1D
  - Preserves locality well (nearby points in 2D → close on 1D line)
  - Recursive → can zoom infinitely for higher precision
- Used for efficient range + proximity ~~range~~ queries.

4) Proximity Queries with Hilbert Curve

- Map 2D points → positions on 1D Hilbert curve
- To find nearby points:
  - Take a ± threshold around a position
  - Eg. pos(29), threshold ±6 → range [23, 34] → nearby region
- Efficient for approximate proximity search
- Edge cases:
  - Sometimes far points appear close (false positives)
  - Sometimes close points appear far (false negatives)
  - Deeper recursion reduces errors

Consistency in Distributed Systems

- Consistency = multiple copies of data should match each other
- If update in one place, all replicas must eventually or immediately reflect the same value

1) Single server

- Easy consistency → only one copy of data
- Problems:
  - Single point of failure (if server dies → data lost)
  - Vertical scaling limit (can't scale beyond supercomputer)
  - High latency (users far away get slow responses)

2) Multiple servers without Sharing

- Servers host only local data → reduced load per server
- Problems:
  - No global data sharing
  - Latency if cross region requests
  - Still single point of failure per region

3) Replication (Multiple Copies of Data)

- Store same data in multiple regions →
  - No single point of failure
  - Reduced latency (read / write locally)
- Problem / Challenge → keeping replicas consistent

→ Update Propagation : updating one copy means syncing all others

→ Message Delivery failures : updating or acknowledgements may get lost

→ Two General Problem
- can't guarantee both sides know for sure the update succeeded
- Infinite ack-of-ack loop problem.

Solutions & Protocols

→ Leader - Follower Model
- One server = Leader (handles all writes)
- Other servers = Followers (replicate data from leader)
- Simplifies consistency management
- Problem :
  - → If leader fails, need re-election
  - → Reads may see stale data until replication finishes

→ Consistency vs Availability Trade off
- If you wait for replication → strong consistency, low availability
- If you respond immediately → High availability, weaker consistency
- This is CAP Theorem in action

→ Two Phase Commit (2PC)
- Used for distributed transactions across multiple systems
- Steps :-
  1. Prepare Phase → Leader asks followers to "prepare" → log updates but don't commit yet
  2. Commit phase → If all acks, leader sends "commit"

Else, rollback.

- Ensures atomic commits
- Problem → Slow + blocking (if a follower hangs, system waits)

Types of Consistency Models

- Strong Consistency → Every read sees the latest write
- Eventual Consistency → Updates propagate asynchronously, replicas eventually become consistent
- Causal Consistency → Related operations preserve order, but not all operations are globally ordered.

Consistency Trade Offs → Latency (High) and Availability (low)

ACID (Properties of transaction in databases)

A - Atomicity → All or nothing
C - Consistency → Preserving database invariants
I - Isolation → concurrent transactions are isolated from each other
D - Durability → data is persisted after transaction is committed even if a system failure

CAP (Theorem for distributed system)

C - Consistency → every read gets the latest write (all nodes show same data)
A - Availability → every request receives a response (no downtime)
P - Partition tolerance → system continues to operate even if network partitions occur.

- In distributed system, you can only guarantee 2 out of 3 ↑

Asynchronous API :

Async is core to scalable systems. Perform tasks without waiting for others to finish.

Frontend need for

- Handling user interactions smoothly
- Running background tasks (API calls, animations, data fetches) without freezing UI

Backend need for

- concurrent requests hitting server
- Parallel execution (quering DB + calling an API + writing logs simultaneously)

Why Asyn important ?

- Higher efficiency & lower cost
- Better customer experience → no waiting
- Real world impact
  - Amazon → +100 ms latency = +1% revenue
  - Google → +500 ms delay = -20% users

Types of Async

1. Concurrency
   - Single worker switching between tasks
   - One task at a time, but rapid switching
2. Parallelism
   - Multiple workers doing tasks simultaneously
   - Faster, but higher cost & idle resources possible
3. Mix of Both
   - Most real systems combine concurrency + parallelism
   - Maximum throughput + low latency.

Drawbacks

- Too much concurrency → context switching → thrashing (wasted CPU cycles)
- Too much parallelism → idle / wasted resources → high cost
- Complexity → harder to write / debug code → race conditions
  → requires language features to manage.

OAuth (Open Authorization) :

OAuth ≠ Authentication (who you are)

OAuth = Authorization (what you can access)

But in practice, used for authentication too

Problem with Passwords

- Too many websites → too many passwords
- Pain users (memory / managers
- Pain for startups (drop off during signup)

OAuth Flow :

1. User signs up → Instead of creating password, site say "Login via Google / Facebook / etc".
2. Redirect to provider (Google)
3. Provider checks if user exists → asks user for permission to share limited data (eg. name, email)
4. Provider sends back verified user details + access token
5. Website uses token for session
6. User can revoke access anytime (token invalidated. from Google dashboard.

War Story (Startup Example)

- Before OAuth : ~50 signups / day
- After OAuth : 2X signups immediately
- Biggest impact : mobile users (hate typing passwords)

Advantages :

1. Outsourced security → no need to handle passwords
2. Standardized protocol → simpler implementation
3. Easy onboarding → click & sign in
4. Mobile friendly → friction less signup
5. Email verification handled by provider
6. Can integrate multiple provider (Google, Github, facebook, etc)

Drawback :

1. Dependency on provider → if Google / Meta is down, your signups fail
2. Less flexibility → limited to what provider allows
3. Risk of data sharing → users may see competitor ads
4. Account deletion issue → if provider account deleted, user may lose access
5. Vendor lock in → provider can ban you.

Security Concepts

- Authentication = verifying identity
- Authorization = granting permission to access resources
- Other mechanisms
  - Token based authentication
  - Single Sign On
  - Access Control Lists & Rule Engines
  - DDOS protection, secure CDN / video access, DRM for content.

Virtualization & Containers

Problem:

- Capacity planning → guess compute, memory, storage needed → buy hardware
- Issues
  - If too small → scale problem
  - If too big → high upfront cost & wasted resources
  - Scaling horizontally → buying more computers
- Resources
  - Multiple apps/users share same machine → resource contention
  - OS manages CPU, Memory, IO, Disk allocation
  - Still → one program can hog resources → affect others.

Virtual Machines (VMs)

- VM = OS running on top of OS (via hypervisor)
- Provides strong isolation → each VM gets dedicated resources
- Benefits
  1. Cloud computing model - rent VMs instead of buying servers
  2. Platform independence - eg. run linux on Windows
  3. Dynamic provisioning - start/stop VMs easily
- Drawbacks
  → Slow boot times (like starting full computer)
  → Heavy resources usage

Containers

- Lightweight virtualization → isolates apps + dependencies, not full OS
- Faster boot/teardown vs VMs (no full OS)
- Mount/unmount process → like attaching filesystem
- Docker → made containerization simple & developer friendly
  - Developers specify: resources + OS
  - Docker handles hardware / VM abstraction

- Advantages
  - Fast provisioning (boot times)
  - Easy portability (code + environment)
  - Efficient resource usage compared to VMs
- Drawbacks
  - Slightly slower than native execution
  - Possible firewall / networking issues
  - Overhead of container management not worth it for simple apps.

Service Discovery & Health Checks :

- Server side focus = reliability & availability, not raw efficiency
- User trust and use services more if they are always available → leads to revenue.

Health Checks

- Health Service monitors all servers ($S_1, S_2, S_3, \ldots$)
- Regularly sends heartbeat messages ("Are you alive")
- If server doesn't respond twice → marked dead / critical
- Dead server → remove from load balancer + restart on ~~monitor~~ another machine.

Problem → sometimes OS responds "yes" but application is dead

Solution → Two way Heartbeat :

1. Health service → server ping
2. Server → proactively sends "I'm alive"

Helps avoids zombies (machines running but unusable)

Service Discovery

- New service runs on multiple servers
- Must register with Load Balancer
  - IP addresses + ports of active instances

- LB maintains a snapshot of all services
- Other services don't need to track IPs → always query LB
- Snapshots are versioned and persisted (DB)
- LB can cache service info for efficiency.

Health checks + Service discovery Integration
- When snapshot changes (new / dead instance)
  - Health Service verifies new servers are alive
  - Keeps connections to service ports.
- Ensures dynamic system awareness (what's alive, what's dead, where requests should go).

Rate Limiting :
Problem : Thundering Herd
→ Sudden spike in requests = crushes servers
→ Example : 4 servers, one crashes → load redistributes → others overload & crash (cascading failure)
→ Cascading Failure : chain reaction of server crashes

Solutions
1) Rate Limiting (Queues + Capacity)
- Each server has queue + max capacity (eg. 300 req/sec)
- Extra requests → dropped (fail response)
- Client retries later → avoids overload
- Return temporary vs permanent errors
  - Temporary : retry later (server busy, DB slow)
  - permanent : bad request (don't retry)

2) Scaling for Events
- Pre scaling → add servers before known spikes
- Auto scaling → cloud auto adds / removes servers
- Viral traffic → fallbacks to rate limiting

3) Job Scheduling
- Avoid running huge jobs at once (eg. 1M new yr mails)
- Break into batches (1k/min) → manageable load
- Batch processing improves reliability

4) Popular Posts Handling
- If a celebrity posts → millions of notifications
- Use batch processing + jitter (staggered sending)
- Metadata (likes / views / comments) → serve approximate stats instead of exact → saves DB load.

5) Good Practices
- Caching → store frequent responses, reduce DB queries
- Gradual Deployments → release updates to small % servers first
- Coupling (with caution)
  - Cache non critical data (profile pic) locally to reduce external calls.
  - Avoid caching sensitive data (auth, forms, passwords)

Anomaly Detection in Server Systems
→ Monitor system health
→ Detect anomalies (unusal patterns)
→ Alert engineers early.

Design Principles
- Prefer false positives > false negatives
  - cheaper for engineer to check / false alarm than to miss a real issue
- Uniform & standardized metric collection from all services
- Use sidecar (Service Mesh)
  - Handles routing, messaging, Logging
  - Collects metrics → sends to anomaly detection engine

Detecting Anomalies

1. Threshold based
   - Define upper / lower limits (static or from historical data)
   - Simple but error prone (ignores long term trends)
2. Dynamic Filters
   - Low pass & High pass filters → follow graph trend
   - Sharp deviations = anomaly
   - Limitation : ignores seasonal / temporal effects (eg. Christmas spikes)
3. Seasonality Awareness
   - Compare with historical seasonal data (eg. last yr's December)
   - Prevents false alarms during predictable peaks.
4. Isolation trees (ML)
   - Based on decision trees
   - Partition data → anomalies = points isolated with fewest cuts
   - Works well for time series anomaly detection.