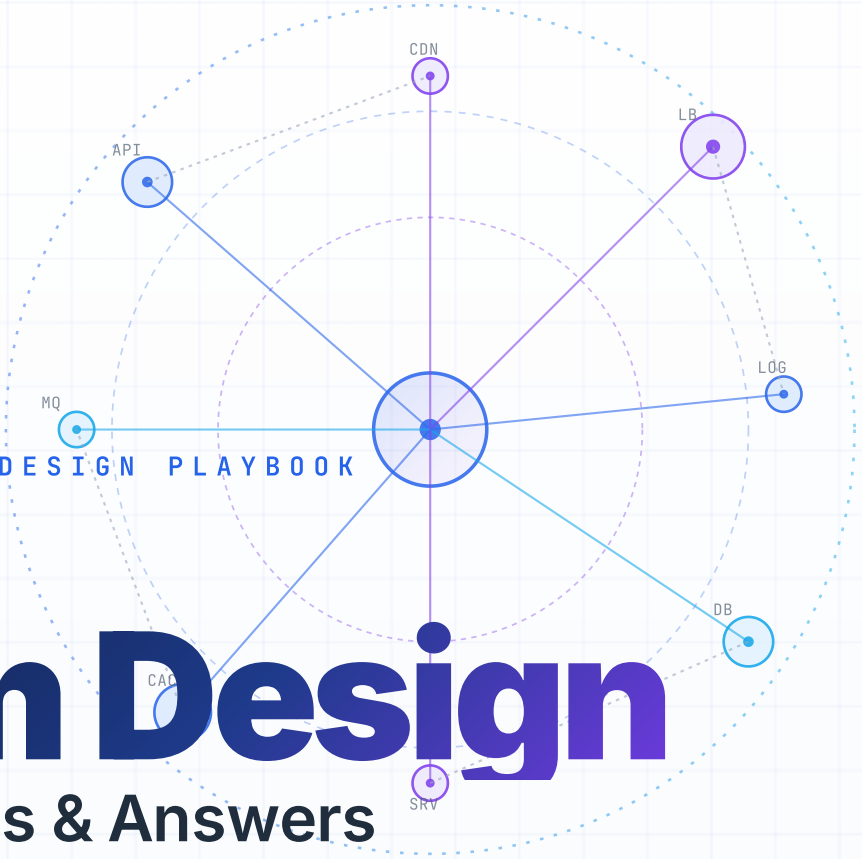
INTERVIEW PREPARATION GUIDE

SDE · SENIOR · STAFF · TECH LEAD

2026 / EDITION



THE DEFINITIVE SYSTEM DESIGN PLAYBOOK

Top 100 System Design Interview Questions & Answers

A **comprehensive, interview-focused guide** covering **100 hand-picked questions** with detailed solutions, capacity estimates, architecture diagrams, trade-off analyses, and mock interviews. Built for engineers targeting **Google, Amazon, Meta, Microsoft, Netflix, Uber** and other top-tier product companies.

100	30	20	4
QUESTIONS	SECTIONS / Q	MOCK INTERVIEWS	DIFFICULTY LEVELS

TARGET COMPANIES

Google · Amazon · Microsoft · Meta · Netflix · Uber · Airbnb · Apple · Oracle · Adobe · Salesforce ·

Z.ai Press
ENGINEERING INTERVIEW SERIES

2026
VOLUME I · FIRST EDITION

TABLE OF CONTENTS

Table of Contents

PART I

Introduction to System Design

Why it matters, what interviewers expect, and the vocabulary every engineer must master.

1.1 What is System Design?

System design is the process of defining the architecture, components, modules, interfaces, and data flows of a system to satisfy specified functional and non-functional requirements. In the context of software engineering interviews, system design refers to the open-ended, end-to-end design of a real-world distributed system at scale — think "Design Twitter" or "Design a URL shortener" — where the candidate is expected to reason about users, traffic, storage, latency, consistency, failure modes, and operational concerns.

Unlike coding interviews that test algorithmic problem-solving on a constrained input, system design interviews test **judgement under ambiguity**. There is no single correct answer. Two equally strong candidates may produce wildly different architectures and both pass, provided they can articulate the trade-offs they made and defend their choices with first-principles reasoning. The skill being evaluated is not recall of facts but the ability to **decompose a vague problem**, identify the hard parts, make reasonable assumptions, and produce a coherent design that an engineering team could realistically build and operate.

A typical 45-minute system design round follows a recognizable arc: scope the problem (5 min), estimate capacity (5 min), propose high-level design (10 min), deep-dive on a subsystem (15 min), identify bottlenecks and wrap up (10 min). Strong candidates explicitly drive this structure rather than waiting for the interviewer to direct them.

> **Interview Insight**
>
> The single biggest mistake candidates make is jumping into a solution before scoping the problem. Spend the first five minutes asking clarifying questions: Who are the users? What is the expected scale? Is this read-heavy or write-heavy? What are the latency SLAs? What consistency guarantees do we need? Five minutes of scoping saves twenty minutes of redesign.

1.2 Why Companies Ask System Design Questions

Top-tier product companies ask system design questions because the problems they face daily — serving billions of requests, storing petabytes of data, surviving regional outages, rolling out features without downtime — cannot be solved by a single algorithm or library call. They require engineers who can reason holistically about distributed systems, understand the cost of every architectural choice, and make decisions that compound over years of operation.

For senior and staff-level roles, system design is often the **highest-weighted interview** in the loop. A candidate who aces coding but stumbles on design will be down-levelled; a candidate who codes adequately but demonstrates exceptional design judgement will be up-levelled. This is because the marginal cost of a poor algorithmic choice is bounded (a function runs 10x slower), whereas the marginal cost of a poor architectural choice is unbounded (a sharding scheme that cannot be rebalanced may require a multi-year rewrite).

Table 1.1 — System Design expectations by level

Level	What is evaluated	Sample question
SDE-1 / New Grad	Basic components, simple APIs, single DB	Design URL Shortener, Pastebin
SDE-2 / Mid	HLD with caching, queues, scaling reasoning	Design Instagram, Twitter

Level	What is evaluated	Sample question
Senior SWE	Deep-dive on subsystems, trade-offs, bottlenecks	Design Uber, Netflix
Staff Engineer	Multi-region, consistency models, organizational impact	Design Kafka, Multi-region DB
Tech Lead	Cross-team architecture, migration strategies, cost	Design a migration from monolith to microservices

1.3 Interview Expectations

Interviewers are trained to evaluate candidates along four orthogonal axes. A strong performance requires demonstrating competence on **all four**, not just one.

1. Problem decomposition. Can you take an ambiguous prompt ("Design YouTube") and break it into a tractable set of subproblems (video upload pipeline, transcoding, storage, CDN distribution, recommendation, playback)? Do you explicitly state assumptions and verify them with the interviewer?

2. Technical breadth and depth. Do you know the standard building blocks (LB, cache, queue, DB, CDN) and when to apply each? Can you go deep on at least one subsystem — explaining why Cassandra and not MongoDB, why Kafka and not RabbitMQ, why consistent hashing for shard assignment?

3. Trade-off analysis. Every design decision has a cost. Strong candidates name the alternative they rejected and why: "We could use strong consistency here, but it would cap our write throughput at the latency of the slowest replica, so I am choosing eventual consistency with read-repair."

4. Operational maturity. How do you monitor the system? What happens when a node dies? How do you roll out a schema change without downtime? How do you debug a latency spike at 3 a.m.?

1.4 High-Level Design (HLD) vs Low-Level Design (LLD)

A high-level design describes the system at the level of major components and their interactions — "users hit a load balancer in front of stateless API servers, which read from a Redis cache and write to a PostgreSQL primary with read replicas." HLD is what you draw on the whiteboard in the first 15 minutes.

A low-level design zooms into a single component and specifies its internal structure: class diagrams, interface signatures, data structures, algorithms, concurrency model, and persistence format. For a rate limiter, LLD would specify the sliding-window algorithm, the Redis key layout, the Lua script for atomic check-and-increment, and the fallback behaviour when Redis is unavailable.

Table 1.2 — HLD vs LLD at a glance

Dimension	High-Level Design	Low-Level Design
Scope	Whole system	Single component or service
Audience	Architects, EMs, cross-team reviewers	IC engineers implementing the service
Artifacts	Block diagram, data flow, API contracts	Class diagram, sequence diagram, schema

Dimension	High-Level Design	Low-Level Design
Time in interview	First 15-20 minutes	Last 15-20 minutes (deep-dive)
Typical question	How do we scale to 10M users?	How does the rate limiter handle clock skew?

1.5 Functional vs Non-functional Requirements

Functional requirements describe what the system *does* — the behaviours and capabilities visible to a user. For a URL shortener: "a user submits a long URL and receives a short URL; visiting the short URL redirects to the original." Functional requirements are typically testable with a single API call.

Non-functional requirements describe *how well* the system does it — the quality attributes that govern production behaviour: latency, throughput, availability, durability, consistency, security, cost. Non-functional requirements are usually expressed as SLOs with numeric targets ("99.9% of reads return within 50ms").

In interviews, candidates often over-index on functional requirements and forget non-functional ones. Interviewers care about the latter because they are what make systems fail in production. A URL shortener that "works" but takes 3 seconds to redirect is a failed URL shortener.

Table 1.3 — The standard NFR checklist

NFR	Definition	Example target
Latency	p50, p95, p99 response time	p99 read < 100ms
Throughput	Requests per second sustained	100K RPS write, 1M RPS read
Availability	Uptime percentage over a window	99.95% monthly
Durability	Probability of data loss	11 nines for object storage
Consistency	Read-after-write guarantees	Strong for billing, eventual for feed
Scalability	Growth headroom without rewrite	10x traffic without code change
Security	AuthN, authZ, encryption, audit	TLS 1.3, mTLS internally
Cost	Dollars per request or per user	< $0.001 per shortening

1.6 CAP Theorem

Eric Brewer's CAP theorem states that a distributed data store can simultaneously provide at most two of the following three guarantees: **Consistency** (every read receives the most recent write or an error), **Availability** (every request receives a non-error response, without guarantee that it reflects the latest write), and **Partition tolerance** (the system continues to operate despite an arbitrary number of messages being dropped or delayed by the network).

Because network partitions are unavoidable in real distributed systems, the practical choice is between **CP** (consistency + partition tolerance — refuse reads/writes during partition) and **AP** (availability + partition tolerance — serve stale data during partition). Spanner, HBase, and etcd are CP. Cassandra, Dynamo, and eventually-consistent caches are AP. CA systems are a theoretical curiosity — they describe single-node databases.

> **Common Misconception**
>
> CAP does not say "pick two forever." It says "during a network partition, you must choose between C and A." Outside of partitions, modern systems can offer both strong consistency and high availability via protocols like Paxos, Raft, and 2PC. CAP is a network-failure behaviour, not a steady-state property.

Table 1.4 — CAP choices in popular systems

System	Choice	Rationale
etcd / Consul	CP	Raft consensus; rejects writes if quorum unavailable
Spanner	CP	Paxos groups; externally consistent via TrueTime
MongoDB (replica set)	CP	Primary fails over; reads block until new primary elected
Cassandra	AP	Tunable consistency; accepts writes during partition, reconciles later
DynamoDB	AP	Eventually consistent by default; strong reads optional at 2x cost
Redis (single-node)	CA	Single-threaded; no partition scenario, but no replication HA

1.7 Scalability, Availability, Reliability, Fault Tolerance

Scalability is the ability of a system to handle increased load by adding resources. A system scales *vertically* when you make each node bigger (more CPU, RAM, disk) and *horizontally* when you add more nodes. Vertical scaling has a hard ceiling (the largest available machine) and a cost curve that accelerates steeply; horizontal scaling is bounded only by your ability to partition work and coordinate nodes.

Availability is the fraction of time a system is operational and accessible. It is typically measured in "nines" — 99% (two nines) allows 3.65 days of downtime per year, 99.999% (five nines) allows only 5.26 minutes. Availability is improved by eliminating single points of failure, deploying across availability zones and regions, and implementing graceful degradation.

Reliability is the probability that a system correctly performs its intended function for a specified period. A reliable system produces correct results; an available system responds. The two are related but distinct — a system can be available but unreliable (always returns a response, sometimes wrong) or reliable but unavailable (always correct when it responds, but crashes frequently).

Fault tolerance is the ability to continue operating correctly after a component fails. It is achieved by redundancy (multiple replicas), health checking, automatic failover, and design choices that localize failure (bulkheads, circuit breakers, timeouts). A fault-tolerant system degrades gracefully rather than failing catastrophically.

Table 1.5 — Availability targets and allowed downtime

Availability	Annual downtime	Monthly downtime	Typical use case
99% (two nines)	3.65 days	8.76 hours	Internal tools, batch jobs
99.9% (three nines)	8.76 hours	43.8 minutes	Most SaaS products
99.95%	4.38 hours	21.9 minutes	Critical business apps
99.99% (four nines)	52.6 minutes	4.38 minutes	Payment, auth, DNS
99.999% (five nines)	5.26 minutes	25.9 seconds	Telecom, emergency systems

1.8 Latency, Throughput, Consistency

Latency is the time taken for a single operation to complete. In distributed systems latency is dominated by network round-trips and disk seeks. Always report percentiles (p50, p95, p99) — averages hide tail latency, and the tail is what users actually experience. A common rule of thumb: 1ms for in-memory cache hit, 5ms for SSD read, 20ms for cross-AZ network, 100ms for cross-region network, 150ms for global round trip.

Throughput is the rate at which operations complete, typically expressed in requests per second (RPS) or events per second. Throughput and latency are related but not equivalent: a system can have high throughput and high latency (batch processed in parallel) or low throughput and low latency (single-threaded in-memory operation). The relationship is captured by Little's Law: *throughput = concurrency / latency*.

Consistency in distributed systems refers to the visibility of writes. The consistency spectrum runs from *strong* (linearizability — all reads see the latest write) through *sequential* (writes appear in the same order to all readers) and *causal* (causally related writes are ordered) down to *eventual* (given no new writes, all replicas eventually converge). Stronger consistency costs latency and availability; weaker consistency buys performance.

1.9 Partitioning, Replication, Sharding

Partitioning (also called **sharding** when applied to databases) is the practice of splitting a dataset across multiple nodes so that each node holds only a subset. The three common partitioning strategies are *range-based* (keyspace divided into ranges, good for range queries but prone to hotspots), *hash-based* (key hashed to a node, even distribution but loses range locality), and *consistent hashing* (hash ring with virtual nodes, minimizes data movement on add/remove).

Replication is the practice of storing multiple copies of the same data on different nodes. The three common topologies are *single-leader* (writes go to one node, replicated to followers; simple, but leader is a bottleneck), *multi-leader* (writes accepted at multiple nodes, replicated peer-to-peer; supports multi-region writes but requires conflict resolution), and *leaderless* (any node accepts writes; quorum reads reconcile; used by Dynamo-style systems).

Sharding and replication compose: each shard has its own replicas. The interaction between the two is non-trivial — for example, in a sharded single-leader database, failover of a single shard's leader must not cascade into a global outage.

1.10 Load Balancing, Caching, Queues, CDN

Load balancing distributes incoming requests across multiple backend instances. L4 load balancers (HAProxy, AWS NLB) operate on TCP/UDP and are fast but unaware of application semantics. L7 load balancers (Nginx, Envoy, AWS ALB) parse HTTP and can route by path, header, or cookie. Common algorithms: round-robin, least-connections, consistent hashing (for session affinity), and weighted (for heterogeneous fleets).

Caching exploits the temporal locality of access patterns: data read once is likely to be read again soon. A cache hit returns in microseconds; a cache miss falls through to the slower backing store. Common patterns include *cache-aside* (application manages cache explicitly), *read-through* (cache fetches on miss), *write-through* (writes go to cache and store synchronously), and *write-behind* (writes go to cache, persisted asynchronously). Cache invalidation remains one of the two hard problems in computer science.

Message queues decouple producers from consumers. They smooth bursty traffic (queues absorb spikes), enable async processing (producer does not wait), and provide backpressure (consumers process at their own rate). The two dominant patterns are *point-to-point* (RabbitMQ, SQS — one consumer per message) and *publish-subscribe* (Kafka, SNS — multiple subscribers per message).

Content Delivery Networks (CDNs) cache static assets (and increasingly dynamic content) at edge locations close to end users. A user in Mumbai fetching an image from an origin in Virginia experiences 250ms of network latency; the same image served from a Mumbai PoP completes in 15ms. Cloudflare, Akamai, and CloudFront operate hundreds of PoPs worldwide.

1.11 Databases, Microservices, Event-Driven, Distributed Systems

Databases come in many flavours. Relational databases (PostgreSQL, MySQL) provide ACID transactions, schema enforcement, and SQL querying — ideal for transactional workloads with complex queries. NoSQL databases trade some of these guarantees for horizontal scalability: document stores (MongoDB, Couchbase) for flexible schemas, wide-column stores (Cassandra, HBase) for time-series and heavy writes, key-value stores (Redis, DynamoDB) for low-latency lookups, and graph databases (Neo4j) for relationship queries. NewSQL systems (Spanner, CockroachDB, TiDB) attempt to combine SQL with horizontal scalability.

Microservices decompose a monolithic application into small, independently deployable services, each owning its own data store. The benefits include independent scaling, technology diversity, and failure isolation. The costs are substantial: distributed transactions become hard (sagas, outbox pattern), network calls replace function calls (latency, retries, circuit breakers), observability requires distributed tracing, and deployment complexity explodes. A common rule of thumb: start with a modular monolith and extract microservices only when scale or organizational boundaries demand it.

Event-driven architecture structures a system around the production, routing, and consumption of events. Producers emit facts ("OrderPlaced", "PaymentCaptured"); consumers react by updating state, triggering workflows, or deriving views. Benefits: loose coupling (producers do not know who consumes), temporal decoupling (consumers process asynchronously), and auditability (events form an immutable log). Costs: harder to reason about end-to-end flow, eventual consistency is the default, and debugging requires event replay.

Distributed systems are characterized by independent failure of components, lack of a global clock, and message-passing communication. The *eight fallacies of distributed computing* (Peter Deutsch, Sun Microsystems)

list the assumptions that newcomers make and that bite them in production: the network is reliable, latency is zero, bandwidth is infinite, the network is secure, topology doesn't change, there is one administrator, transport cost is zero, and the network is homogeneous. None are true.

> **Key Takeaway**
>
> System design interviews reward engineers who can name the building blocks, explain when each applies, and reason about the cost of every choice. This book walks through 100 realistic design problems to build that vocabulary and judgement through repetition.

PART II

System Design Fundamentals

Thirty-three building blocks every engineer should know cold.

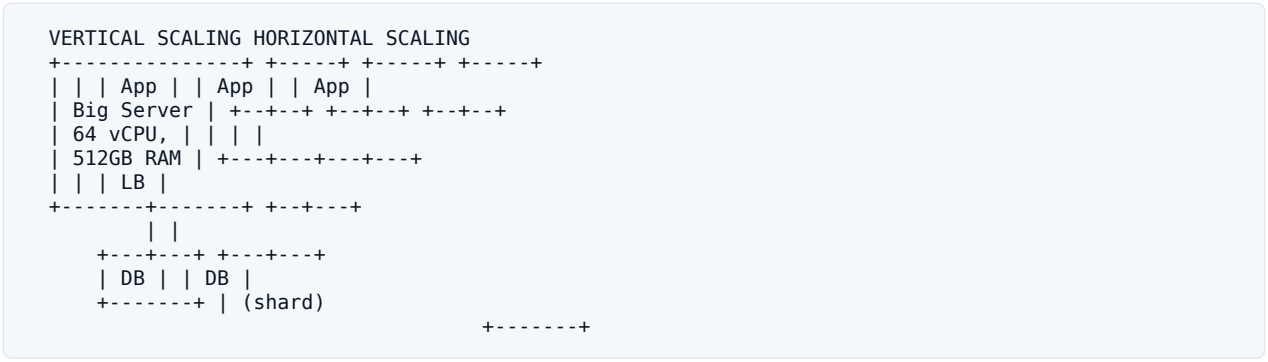
2.1 Horizontal vs Vertical Scaling

Definition. Vertical scaling (scale up) adds more CPU, RAM, or disk to a single machine. Horizontal scaling (scale out) adds more machines to a fleet. Vertical scaling is simpler (no code changes, no distributed state) but bounded by the largest available instance and a hard failure domain. Horizontal scaling has no theoretical ceiling but requires stateless services, partitionable data, and load balancing.

When to use. Vertical for in-memory databases, single-tenant systems, and quick wins. Horizontal for stateless web/API tiers, partitionable data stores, and systems that must exceed the capacity of one machine.

Trade-offs. Vertical: simpler ops, higher unit cost, single point of failure. Horizontal: lower unit cost, fault tolerant, but requires distributed coordination.

Diagram



2.2 Monolithic vs Microservices

Definition. A monolith packages all functionality into a single deployable unit sharing one process and one database. Microservices split functionality into small services that own their own data and communicate over the network. Monoliths are simpler to build, deploy, and test; microservices enable independent scaling, deployment, and technology choice at the cost of distributed-systems complexity.

When to use. Monolith for early-stage products, small teams, and unclear domain boundaries. Microservices for large teams, independent scaling requirements, and stable domain boundaries.

Trade-offs. Monolith: simple, but a change to one module redeploys everything. Microservices: independent deploys, but distributed transactions and observability become hard.

Diagram

2.3 SQL vs NoSQL

Definition. SQL databases store data in tables with rigid schemas and provide ACID transactions and SQL query language. NoSQL databases trade schema rigidity and ACID for horizontal scalability: document (MongoDB), wide-column (Cassandra), key-value (Redis), and graph (Neo4j) families each optimize for a different access pattern.

When to use. SQL for transactional systems with complex joins and strong consistency. NoSQL for high write throughput, flexible schema, or simple access patterns.

Trade-offs. SQL: strong consistency, joins, but vertical scaling. NoSQL: horizontal scaling, but limited joins and eventual consistency.

Diagram

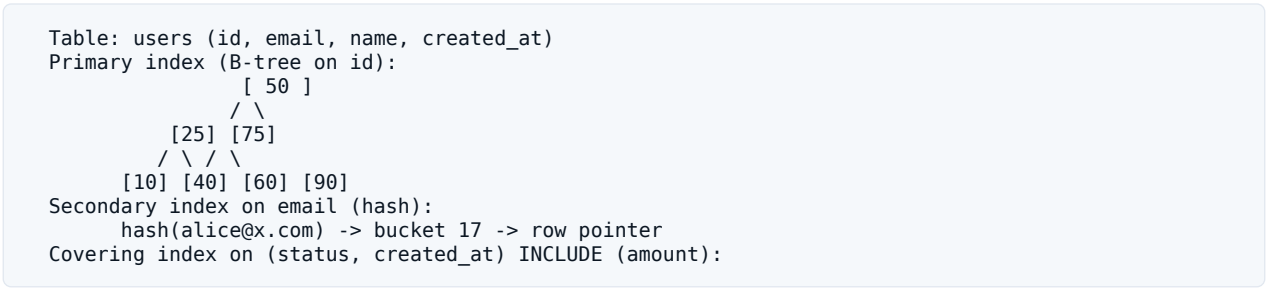
2.4 Database Indexing

Definition. An index is an auxiliary data structure that accelerates reads at the cost of slower writes and additional storage. B-tree indexes support range queries and point lookups in O(log n). Hash indexes support only point lookups in O(1). Composite indexes optimize multi-column predicates; covering indexes include all columns needed by the query to avoid table lookups.

When to use. Index columns used in WHERE, JOIN, ORDER BY, and GROUP BY clauses. Avoid over-indexing write-heavy tables.

Trade-offs. Faster reads vs slower writes. Each index adds ~10% write overhead.

Diagram



2.5 API Gateway

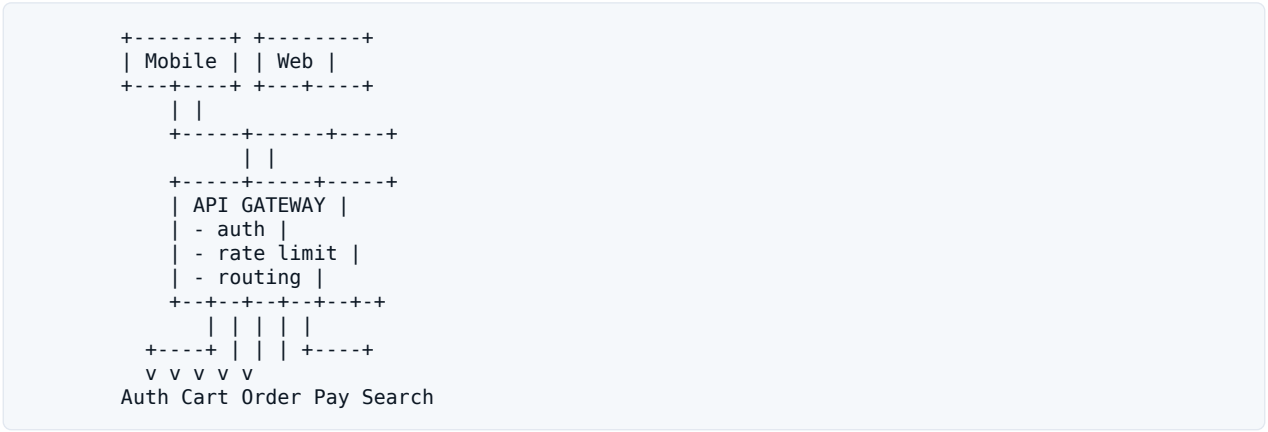
Definition. An API gateway is the single entry point for all external clients. It handles authentication, rate limiting, request routing, protocol translation, response transformation, and observability. Modern gateways (Kong, Envoy, AWS API Gateway) also support request/response rewriting, JWT validation, and circuit

breaking.

When to use. Any system with multiple backend services exposed to external clients. Essential for microservices architectures.

Trade-offs. Centralizes cross-cutting concerns vs adds a network hop and a single point of failure (mitigated by horizontal scaling).

Diagram



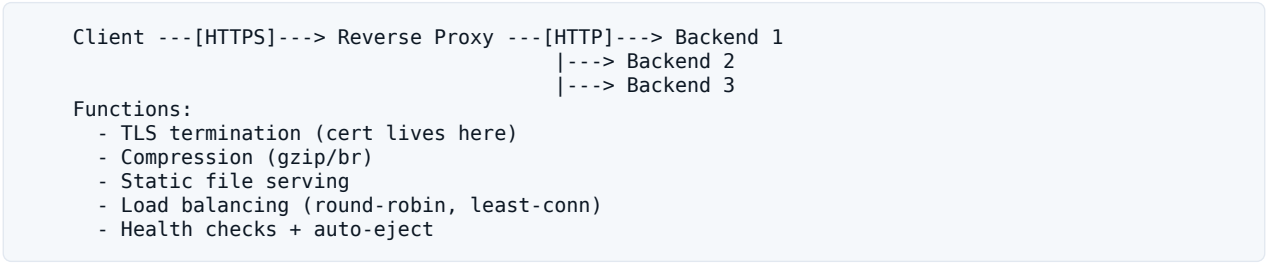
2.6 Reverse Proxy

Definition. A reverse proxy sits in front of backend servers and forwards client requests to the appropriate server. Unlike a forward proxy (which protects the client), a reverse proxy protects the server. Nginx, HAProxy, and Envoy are common reverse proxies; they also provide TLS termination, compression, caching, and load balancing.

When to use. Always — even a single-backend deployment benefits from TLS termination and access logging at the proxy.

Trade-offs. Adds a network hop (1-2ms) but centralizes concerns and protects backends.

Diagram



2.7 Message Queues

Definition. A message queue accepts messages from producers and delivers them to consumers, decoupling the two. Point-to-point queues (RabbitMQ, SQS) deliver each message to exactly one consumer. Pub-sub topics (Kafka, SNS) deliver each message to all subscribers. Queues provide backpressure, async processing, and burst absorption.

When to use. Any time a slow operation should not block a fast request: email sending, image processing, order fulfillment, log ingestion.

Trade-offs. Adds operational complexity and latency for the consumer. Requires idempotent consumers (messages may be delivered at-least-once).

Diagram

```
Producer -> [ Queue ] -> Consumer
                |
                v
            Persistence
                |
                v
            Dead-letter
            (failed msgs)

Patterns:
  - At-least-once (default; consumers must be idempotent)
  - At-most-once (no retry; data may be lost)
  - Exactly-once (hard; requires transactional outbox)
```

2.8 Redis

Definition. Redis is an in-memory key-value store that supports strings, lists, sets, sorted sets, hashes, streams, and pub/sub. It is single-threaded (mostly) and achieves sub-millisecond latency. Persistence is optional (RDB snapshots and AOF append log). Redis Cluster provides sharding and high availability.

When to use. Caching, session store, rate limiting, leaderboards (sorted sets), real-time analytics (HyperLogLog), and distributed locks (Redlock).

Trade-offs. In-memory = fast but volatile. Persistence is asynchronous = potential data loss on crash. Cluster mode adds operational complexity.

Diagram

```
App --+--> [ Redis Primary ] --[replication]--> [ Replica ]
      | ^
      | |
      +---[ Sentinel ] (failover arbiter)

Cluster mode (sharding):
Slot 0-5460: Node A + replica
Slot 5461-10922:Node B + replica
Slot 10923-16383:Node C + replica
(16,384 hash slots; key -> CRC16(key) % 16384)
```

2.9 Memcached

Definition. Memcached is a simpler in-memory key-value store than Redis. It supports only strings, has no persistence, no replication, and no advanced data structures. Its simplicity makes it extremely fast and lightweight — ideal for pure caching workloads where data loss on restart is acceptable.

When to use. Pure cache layer where restarts are tolerable. Use Redis when you need persistence, replication, or advanced structures.

Trade-offs. Simpler and faster than Redis for raw cache throughput, but no persistence, no replication, no atomic operations beyond get/set.

Diagram

```
App -> consistent-hash ring -> [Memcached A] [B] [C] [D]
(no replication; a dead node just causes cache misses)
Eviction: LRU per slab class
Memory: slab allocator (pre-allocated chunks by size class)
```

2.10 Kafka

Definition. Kafka is a distributed, partitioned, replicated commit log. Producers write messages to topics; consumers read them in order. Topics are split into partitions; each partition is an append-only log on disk. Kafka is built for high throughput (millions of events/sec), durability (replicated across brokers), and replayability (consumers track their own offset).

When to use. Event streaming, log aggregation, change data capture, real-time analytics pipelines, microservice event buses.

Trade-offs. High throughput and durability, but complex operations, high memory use, and no per-message priority.

Diagram

```
Producer --+--> [Topic: orders, 3 partitions]--> Consumer Group A
           | Partition 0: [m0][m1][m2][m3]
           | Partition 1: [m0][m1][m2]
           | Partition 2: [m0][m1][m2][m3][m4]
           | ^
           | |
          +--------[ ZooKeeper / KRaft ]
Each partition replicated to N brokers (RF=3 typical)
Consumer group: each partition consumed by exactly one consumer in group
```

2.11 RabbitMQ

Definition. RabbitMQ is a general-purpose message broker implementing AMQP. It supports rich routing (direct, topic, fanout exchanges), message acknowledgements, priority queues, and dead-letter exchanges. Unlike Kafka, RabbitMQ deletes messages once consumed — it is a queue, not a log.

When to use. Work queues, request/reply patterns, complex routing between services, transactional workflows that need acknowledgement.

Trade-offs. Richer routing than Kafka, lower throughput (~50K msg/s vs millions), no built-in replay (consumed messages are gone).

Diagram

```
Producer -> [Exchange]
              |
  +------------+------------+
  | | |
[Queue1] [Queue2] [Queue3]
  | | |
Consumer Consumer Consumer

Exchange types: direct, topic, fanout, headers
```

2.12 Load Balancers

Definition. A load balancer distributes incoming traffic across multiple backend servers. L4 (transport) LBs operate on TCP/UDP — fast, no protocol awareness. L7 (application) LBs inspect HTTP headers, paths, cookies — enable content-based routing. Algorithms: round-robin, least-connections, consistent-hash, weighted.

When to use. Always, when you have more than one backend instance. Even two instances benefit from health-check-driven failover.

Trade-offs. L4: faster, simpler, but no app awareness. L7: richer routing but higher latency and TLS overhead.

Diagram

```
 Client
   |
   v
+--------+ health-check
| LB |<------+-----------+
+--+--+--+ | |
   | | v v
   v v [OK 200] [500 X]
 [S1] [S2] S3 ejected

Algorithms:
  round-robin - cyclic
  least-conn - fewest in-flight
  consistent-h - same key -> same server
  weighted - by capacity
```

2.13 CDN (Content Delivery Network)

Definition. A CDN caches content at edge locations (PoPs) geographically close to end users. On a cache hit, the CDN serves the content directly; on a miss, it fetches from origin and caches for next time. CDNs dramatically reduce latency for static assets and can absorb traffic spikes that would otherwise overwhelm origin servers.

When to use. Static assets (images, CSS, JS), video streaming, large file downloads, and increasingly dynamic content via edge compute (Cloudflare Workers).

Trade-offs. Substantial latency reduction for users far from origin, but adds cache invalidation complexity and a small per-request cost.

Diagram

```
User (Mumbai) Origin (Virginia)
   | ^
   v |
[CDN PoP Mumbai] --miss--> [Origin via fiber]
   | |
   | (hit: 15ms) |
   v |
 User <---- served from edge cache |
   (vs 250ms cross-globe RTT to origin) |
```

2.14 DNS (Domain Name System)

Definition. DNS translates human-readable hostnames (example.com) into IP addresses (93.184.216.34). It is a hierarchical, distributed database organized into zones. DNS lookups are cached at multiple levels: browser, OS, recursive resolver, and authoritative server. TTLs control how long records may be cached.

When to use. Always — every Internet-facing service uses DNS. Use DNS-based traffic routing (GeoDNS, weighted routing) for multi-region failover.

Trade-offs. DNS caching enables fast lookups but means changes take TTL time to propagate. Use low TTLs (60s) for entries that may change.

Diagram

```
Browser -> OS cache -> Resolver cache -> Recursive -> Authoritative
                                                        |
                                                        v
Root (.) -> TLD (.com) -> Authoritative (example.com)
Records: A (IPv4), AAAA (IPv6), CNAME (alias), MX (mail),
         TXT (verification), NS (delegation), SOA (zone meta)
```

2.15 HTTP vs HTTPS

Definition. HTTP sends data in plaintext over TCP. HTTPS wraps HTTP in TLS, providing confidentiality (eavesdroppers cannot read), integrity (tampering is detected), and authentication (the server proves its identity via a certificate). Modern TLS 1.3 with TLS resumption and 0-RTT adds minimal latency over HTTP.

When to use. Always use HTTPS in production. HTTP is acceptable only for internal traffic behind an already-encrypted boundary (and even then, prefer mTLS).

Trade-offs. HTTPS adds ~1-2ms for TLS handshake (negligible with resumption) and requires cert management (ACME / certbot).

Diagram

```
HTTP: Client ---[plaintext]---> Server
HTTPS: Client ---[TLS handshake]---> Server
        1. ClientHello (cipher list, random)
        2. ServerHello + cert + key exchange
        3. Client verifies cert (CA chain)
        4. Both derive session key
        5. Application data (encrypted)
```

2.16 REST APIs

Definition. REST (Representational State Transfer) is an architectural style for HTTP APIs. Resources are identified by URLs, manipulated via HTTP verbs (GET, POST, PUT, PATCH, DELETE), and represented as JSON (or XML, Protobuf). Stateless requests enable horizontal scaling. Hypermedia links (HATEOAS) are the ideal but rarely implemented in practice.

When to use. Public APIs, CRUD-style operations, mobile/web backends, and any context where HTTP semantics align with the resource model.

Trade-offs. Simple, ubiquitous, cacheable. But over-fetching/under-fetching is common, and versioning is messy.

Diagram

```
GET /users/123 -> fetch user
POST /users -> create user
PUT /users/123 -> full update
PATCH /users/123 -> partial update
DELETE /users/123 -> delete user
GET /users/123/orders -> sub-resource
```

2.17 GraphQL

Definition. GraphQL is a query language for APIs. Clients specify exactly which fields they need in a single request; the server returns precisely those fields. This solves REST's over/under-fetching problem. Schemas are strongly typed and introspectable. Mutations modify data; subscriptions push real-time updates.

When to use. Mobile clients with bandwidth constraints, complex UIs needing aggregated data from multiple services, and rapidly-evolving frontends.

Trade-offs. Flexibility for clients, but server complexity rises (resolver logic, N+1 query problem, caching harder than REST).

Diagram

```
Client query:
  query { user(id:1) { name orders { id total } } }
Server response:
  { "data": { "user": {
      "name": "Alice",
      "orders": [{"id":"o1","total":99}]
  }}}
Single endpoint: POST /graphql
```

2.18 WebSockets

Definition. WebSocket is a protocol providing full-duplex communication over a single TCP connection. After an HTTP Upgrade handshake, the connection stays open and either side can push messages. Ideal for real-time applications: chat, live sports, collaborative editing, multiplayer games.

When to use. Real-time bidirectional communication: chat, notifications, dashboards, collaboration.

Trade-offs. Lower latency than polling, but stateful connections are harder to scale (sticky sessions, connection draining).

Diagram

```
Client --HTTP Upgrade--> Server
   <---101 Switching--
=====WebSocket (full-duplex)=====
Client <--> Server (any direction, any time)
Frames: text/binary/ping/pong/close
```

2.19 Long Polling

Definition. Long polling is an emulation of server push over HTTP. The client sends a request; the server holds it open until new data is available (or a timeout expires), then responds. The client immediately re-issues the request. A fallback when WebSockets are not available.

When to use. When WebSockets are blocked (some corporate proxies) or when the server cannot maintain stateful connections.

Trade-offs. Works over plain HTTP, but higher overhead than WebSockets (every message requires a new request/response cycle).

Diagram

```
Client ---request---> Server (holds open 30s)
Client <---response-- Server (when data arrives)
Client ---request---> Server (immediately)
...loop...
```

2.20 Rate Limiting

Definition. Rate limiting caps the number of requests a client can make in a time window. Algorithms: fixed window (simple but bursty at boundaries), sliding window (smoother), token bucket (allows bursts up to a cap), and leaky bucket (smooths output rate). Limits can be per-user, per-IP, per-API, or global.

When to use. Public APIs (prevent abuse), authentication endpoints (brute force), payment systems (retry storms), and any shared resource.

Trade-offs. Protects the service but may reject legitimate traffic. Tune limits carefully and always return 429 with Retry-After.

Diagram

```
Token Bucket (capacity=10, refill=1/s):
  Each request removes 1 token. No tokens = 429.
  Buckets refill continuously.

Sliding Window Log:
  Keep timestamps of requests in last 60s.
  Reject if count > limit.

Implementation:
  Redis key per user:
    INCR ratelimit:user123
    EXPIRE ratelimit:user123 60
```

2.21 Authentication vs Authorization

Definition. Authentication (AuthN) verifies *who* you are (login, password, MFA, biometric). Authorization (AuthZ) verifies *what* you can do (read file X, access resource Y). The two are often confused but distinct: a successfully authenticated user may still be unauthorized for a specific resource.

When to use. Always — any non-trivial system needs both. AuthN at the edge (gateway), AuthZ at the resource (service).

Trade-offs. Coarse-grained AuthZ is fast but rigid; fine-grained AuthZ (per-resource ACLs) is flexible but expensive to evaluate.

Diagram

```
User ---[login]---> Auth Service
       <---[token]---
User ---[token + request]---> API Gateway
  Gateway: AuthN (verify token)
  Service: AuthZ (check role/permission on resource)
```

2.22 OAuth 2.0

Definition. OAuth 2.0 is an authorization framework that lets a user grant a third-party application limited access to their resources on another service, without sharing their password. Flows: authorization code (web apps), implicit (deprecated, single-page apps), client credentials (server-to-server), device code (smart TVs).

When to use. "Login with Google/GitHub/Facebook" buttons, third-party API access, server-to-server access.

Trade-offs. Removes password sharing but adds complexity (refresh tokens, scopes, redirect URIs).

Diagram

```
User -> App: "login with Google"
App -> Google: redirect to consent screen
User -> Google: approve
Google -> App: authorization code (in URL)
App -> Google: code + client_secret
Google -> App: access_token + refresh_token
App -> Google API: Bearer access_token
```

2.23 JWT (JSON Web Token)

Definition. JWT is a compact, self-contained token format consisting of three Base64-URL parts: header, payload, signature. The server signs the token with a secret (HMAC) or private key (RSA/ECDSA); the client presents it on each request. Stateless verification — no session DB lookup needed.

When to use. Stateless authentication for APIs, microservice-to-microservice identity, short-lived access tokens.

Trade-offs. No server-side revocation unless you maintain a blacklist. Always pair short-lived JWT (15 min) with long-lived refresh token (server-side).

Diagram

```
Header . Payload . Signature
{"alg":"HS256"} {"sub":"u1","exp":1700000000} HMAC(base64(header)+'.'+base64(payload), secret)

Verification:
  1. Split on '.'
  2. Recompute signature
  3. Compare to token signature
  4. Check exp, iat, iss, aud
```

2.24 Service Discovery

Definition. Service discovery lets services find each other without hardcoded IPs. Two patterns: *client-side* (client queries a registry like Consul/Eureka and load-balances itself) and *server-side* (a proxy like Envoy queries the registry and routes requests). In Kubernetes, the kube-dns/CoreDNS service provides built-in discovery via DNS names.

When to use. Any microservices architecture with dynamic instances (autoscaling, crashes, deployments).

Trade-offs. Adds a registry as another moving part. Health checking must be robust.

Diagram

```
Service A registers -> [Registry: Consul/etcd/k8s]
Service B queries: "where is A?"
Registry: "10.0.1.5:8080, 10.0.1.6:8080"
B -> 10.0.1.5:8080 (client-side LB)
Or:
B -> Sidecar Proxy -> Registry -> A (server-side LB)
```

2.25 Circuit Breaker

Definition. A circuit breaker protects a service from cascading failures by failing fast. It has three states: *closed* (requests flow normally), *open* (requests fail immediately, no call to downstream), and *half-open* (a probe request is allowed; if it succeeds, circuit closes). Trip thresholds are typically error-rate or latency-based over a sliding window.

When to use. Any synchronous call to a downstream service whose failure should not cascade. Wrap every external dependency.

Trade-offs. Protects the system but may amplify brief outages if thresholds are too sensitive.

Diagram

```
Caller ---request---> [Closed] ---call---> Downstream
                          |
                       errors > threshold (e.g. 50% in 10s)
                          v
                       [Open] ---fail fast---> Caller
                          |
                       wait 30s
                          v
                    [Half-Open] ---probe---> Downstream
                       | |
                    ok | | fail
                       v v
                    Closed Open
```

2.26 Event Sourcing

Definition. Event sourcing persists state as an append-only log of events ("OrderPlaced", "PaymentCaptured", "OrderShipped"). The current state is derived by replaying the event log. Benefits: complete audit trail, temporal queries ("what was the state at time T?"), and easy event-driven integration.

When to use. Financial systems, inventory, anywhere auditability and temporal queries matter. Often combined with CQRS.

Trade-offs. Auditability and replayability, but event schema evolution is hard and replaying large logs is slow.

Diagram

```
Command -> [Event Store] (append-only log)
                |
                v
            Projector -> Read Model (materialized view)
                |
                v
            Query API

Events: OrderPlaced(v1) -> ItemAdded(v2) -> Paid(v3) -> Shipped(v4)
```

2.27 CQRS (Command Query Responsibility Segregation)

Definition. CQRS separates the write model (commands that change state) from the read model (queries that return data). Writes go through a domain model with business rules; reads come from pre-computed projections optimized for the query. The two models are kept in sync via events.

When to use. Systems with read-heavy workloads (10:1 read:write ratio), complex queries spanning multiple aggregates, or where write throughput differs from read.

Trade-offs. Read/write scaling independence, but eventual consistency between write and read side, and operational complexity.

Diagram

```
Client ---write---> Command Handler -> Write Model (DB)
                                          |
                                       events
                                          v
                                    Projector -> Read Model (denormalized)
                                                    ^
Client ---read------------------------------query--+
(read model optimized for queries; can be a different DB)
```

2.28 Distributed Cache

Definition. A distributed cache spreads keys across multiple cache nodes. Two patterns: *consistent hashing* (each key lives on one node; client computes node from key) and *replicated cache* (each node has all keys; fast reads, expensive writes). Memcached and Redis Cluster use consistent hashing; Hazelcast and Infinispan offer replicated mode.

When to use. Cache dataset exceeds single-node memory, or when you need HA (replication plus failover).

Trade-offs. Consistent hash: scales linearly, but a node loss causes partial cache miss. Replicated: full hit on any node, but writes serialize.

Diagram

```
Consistent Hashing Ring:
       Node A (slots 0-5460)
          / \
  Node C Node B
(10923-16383) (5461-10922)

key -> CRC16(key) % 16384 -> Node
Virtual nodes (150 per physical) smooth distribution
```

2.29 Leader Election

Definition. Leader election chooses one node in a cluster to act as the primary/leader for a period. The leader coordinates writes, schedules jobs, or serves as the single source of truth. If the leader fails, a new election is held. Algorithms: Bully, Ring, Raft, Paxos, ZAB (ZooKeeper).

When to use. Any clustered system that needs a single coordinator: replicated databases (MySQL primary, MongoDB primary), scheduler (Kubernetes API server).

Trade-offs. Simpler consistency model than leaderless, but leader is a bottleneck and a single point of failure until re-election completes.

Diagram

```
Cluster of 5 nodes:
  N1 N2 N3 N4 N5
   | | | | |
   +--v--+--+--+
   | Raft Vote |
   | term=5 |
   +-----+-------+
         |
         v
      N3 wins (got 3 votes)
      N3 = LEADER for term 5
      N1 N2 N4 N5 = followers
```

2.30 Consensus Algorithms

Definition. Consensus algorithms allow distributed nodes to agree on a single value despite failures. **Paxos** is the classic (correct but hard to implement). **Raft** is the modern default (understandable, used by etcd, Consul). **ZAB** powers ZooKeeper. **Paxos variants** (Multi-Paxos, Fast Paxos) extend it for practical use.

When to use. Any system requiring strong consistency across replicas: configuration stores, distributed locks, leader election.

Trade-offs. Strong consistency, but requires majority quorum (3 or 5 nodes for HA), higher latency than leaderless (one round trip for quorum).

Diagram

```
Raft Log Replication:
Leader -> Followers (AppendEntries RPC)
  1. Leader appends entry to local log
  2. Leader sends AppendEntries to all followers
  3. Followers append, reply ACK
  4. Leader marks committed once majority ACK
  5. Leader notifies followers to apply to state machine
```

2.31 Monitoring

Definition. Monitoring observes a system's behaviour to detect anomalies and measure SLOs. The four golden signals: **latency**, **traffic**, **errors**, **saturation**. Tools: Prometheus (metrics), Grafana (dashboards), Datadog (commercial APM), StatsD (push metrics). Alert on symptoms (user-visible errors), not causes (CPU high).

When to use. Always. A system without monitoring is a system you cannot operate.

Trade-offs. Detailed monitoring adds overhead (CPU for metrics, storage). Sample judiciously; aggregate before persisting when possible.

Diagram

```
App --metrics--> [StatsD] --> [Prometheus] --query--> [Grafana]
App --logs----> [Fluentd] --> [Elasticsearch] --> [Kibana]
App --traces--> [Jaeger] (OpenTelemetry SDK)
Alerts: Alertmanager -> PagerDuty / Slack
```

2.32 Logging

Definition. Logging records discrete events with timestamps. Three levels: INFO (normal operation), WARN (unexpected but recoverable), ERROR (failure). Structured logs (JSON, not free text) are machine-parseable and searchable. Centralize logs with ELK (Elasticsearch + Logstash + Kibana) or Loki.

When to use. Always — debug failures, audit access, meet compliance requirements.

Trade-offs. Verbose logging helps debugging but costs storage and I/O. Sample or aggregate old logs.

Diagram

```
App -> stdout (JSON lines)
  -> Fluentd/Fluent Bit (ship)
    -> Elasticsearch (index)
      -> Kibana (search & visualize)
Log schema: { ts, level, service, trace_id, msg, fields... }
```

2.33 Metrics & Tracing

Definition. Metrics are numeric measurements aggregated over time (counters, gauges, histograms). Tracing follows a single request across service boundaries via a trace_id propagated through headers. The two are complementary: metrics tell you *that* something is slow; traces tell you *where*.

When to use. Metrics for system-wide health and alerting. Tracing for debugging specific slow requests. Use both.

Trade-offs. Tracing every request is expensive; sample (e.g. 1% or 100% of errors).

Diagram

```
Request enters API Gateway (trace_id=abc)
  -> Auth Service (span: 4ms)
  -> Order Service (span: 12ms)
      -> Postgres (span: 8ms) <- slow!
      -> Kafka (span: 2ms)
  -> Response (total: 18ms)
Distributed trace shows the 8ms DB call is the bottleneck.
```

PART III

Beginner Questions (Q1 – Q20)

Foundational system design problems for SDE-1 and early-career engineers.

QUESTION 01 BEGINNER

Design a URL Shortener (TinyURL)

1. Problem Statement

Design a service that takes a long URL and returns a short alias. Visiting the alias redirects the user to the original URL. The service should be highly available, low-latency, and handle viral spikes when a shortened link is shared broadly.

2. Functional Requirements

- Given a long URL, return a short alias (e.g. **bit.ly/xyz7K**) much shorter than the original.
- Redirect short alias to the long URL with HTTP 301 (or 302).
- Optionally: custom aliases, expiry, analytics (click count, geo, referrer).
- User accounts and link dashboard (optional).

3. Non-functional Requirements

- Availability: 99.99% — redirects must succeed even during partial outages.
- Latency: p99 redirect < 50ms (cached).
- Read-heavy: 100:1 read-to-write ratio.
- Short codes must be unguessable (prevent enumeration) for non-public links.
- URLs persist indefinitely (or until user-deleted).

4. Capacity Estimation

100M new URLs/month, 10B redirects/month. 100:1 read:write. Storage: 500 bytes per (short, long, metadata) × 100M × 60 months = 30GB over 5 years. QPS: 100M writes/mo = ~40 writes/s avg, ~400 peak; 10B reads/mo = ~4000 reads/s avg, ~40K peak.

5. Back-of-the-envelope Math

Short code length: Base62 (a-z, A-Z, 0-9) of 7 chars = 62^7 ≈ 3.5 trillion combinations, enough for decades. Storage growth 30GB/5yr fits on a single SSD today, but we shard for throughput not capacity. Cache hit rate target 90% — 90% of redirects served from Redis.

6. API Design

```
POST /api/v1/shorten
  body: {"long_url": "https://example.com/very/long/path", "custom_alias": "mylink"}
  returns: {"short_url": "https://s.xy/xyz7K", "alias": "xyz7K"}

GET /{alias}
  returns: 301 Location: https://example.com/very/long/path
  (302 if analytics must be recorded before redirect; 301 caches at browser)

GET /api/v1/urls/{alias}/stats # click count, referrers, geo
```

7. Database Schema

```
Table: urls
  alias VARCHAR(7) PRIMARY KEY
  long_url TEXT NOT NULL
  user_id BIGINT NULL
  created_at TIMESTAMP DEFAULT NOW()
  expires_at TIMESTAMP NULL
  click_count BIGINT DEFAULT 0

Index: idx_long_user (long_url, user_id) -- prevent duplicate shortening
```

8. High-Level Design (HLD)

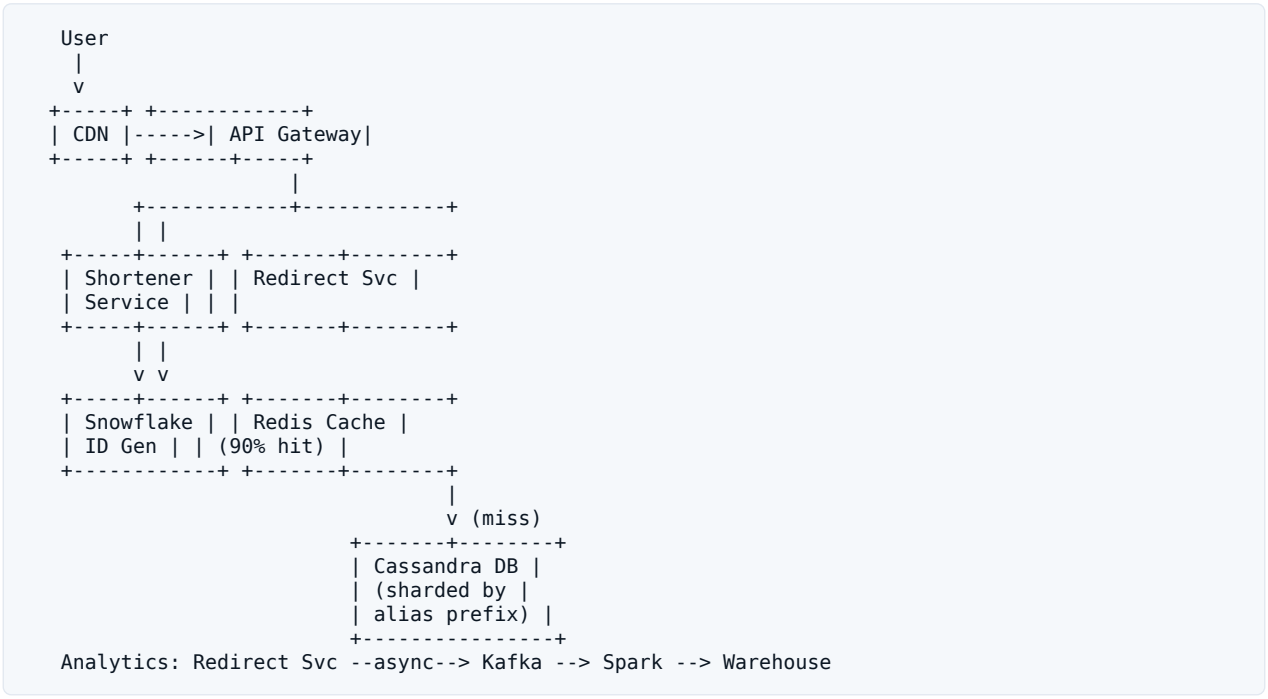
Client → CDN (for static assets) → API Gateway → Shortener Service (writes) or Redirect Service (reads) → Cache (Redis) → DB (Cassandra or sharded Postgres). A async worker writes analytics events to Kafka for batch processing.

9. Low-Level Design (LLD)

Shortener Service: generates alias via Base62 encoding of a monotonically increasing ID from a Snowflake-like ID generator (avoids collisions without coordination). Redirect Service: lookup alias in Redis; on miss, query DB and backfill cache. Use 301 for permanent redirects (browser-cacheable) or 302 for analytics-rich redirects.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Shorten flow:
  1. POST /shorten hits API Gateway.
  2. Gateway authenticates, rate-limits, routes to Shortener Service.
  3. Shortener gets next ID from Snowflake, Base62-encodes to alias.
  4. Insert (alias, long_url) into DB; write-through to Redis.
  5. Return short URL to client.

Redirect flow:
  1. GET /{alias} hits Redirect Service via CDN edge.
  2. Lookup alias in Redis cache.
  3. Cache hit: return 301 with Location header.
  4. Cache miss: query DB, backfill cache (TTL 1h), return 301.
  5. Async: publish click event to Kafka for analytics.
```

12. Component Explanation

- **CDN:** Cloudflare/Akamai at edge; caches 301 redirects near users.
- **API Gateway:** Kong/Envoy; TLS termination, JWT auth, rate limiting.
- **Shortener Service:** Stateless Go/Java service; ~10 replicas for HA.
- **Redirect Service:** Stateless; optimized for reads, separate fleet from writes.
- **Snowflake ID Generator:** Twitter Snowflake-style 64-bit ID from timestamp+worker+seq.
- **Redis Cache:** Cluster mode; alias → long_url, TTL 1h, ~90% hit rate.
- **Cassandra DB:** Sharded by alias hash; RF=3; tunable consistency.
- **Kafka:** Click events; partitioned by alias for ordered analytics.

13. Technology Stack

- **Language:** Go (high concurrency, low memory) for redirect service.
- **Storage:** Cassandra (write-optimized, horizontally scalable).
- **Cache:** Redis Cluster (sub-ms reads).
- **Queue:** Kafka (analytics pipeline).
- **LB:** Envoy + AWS ALB.
- **Deployment:** Kubernetes, multi-AZ.

14. Database Choice

Cassandra over Postgres because: (1) write throughput scales linearly with nodes, (2) tunable consistency per query, (3) no single primary to fail over. Postgres would require sharding logic in the app and would cap at ~10K writes/s per shard.

15. Cache Strategy

Redis cache-aside. Read path checks Redis first; on miss, query DB and backfill with TTL. Hot aliases (top 1% by traffic) get an additional CDN-level cache (301 responses cached at the edge for 60s). Cache stampede protection via singleflight.

16. Load Balancer Strategy

L7 ALB terminates TLS and routes by path (/shorten vs /{alias}). Round-robin across instances within an AZ; cross-AZ only when local AZ is saturated. Health checks every 5s.

17. Message Queue Usage

Kafka topic **click_events** with 64 partitions, keyed by alias. Consumers aggregate into per-alias, per-geo, per-hour rollups written to ClickHouse for analytics dashboards.

18. Scaling Strategy

Horizontal: stateless services scale on CPU. Redis Cluster adds shards as memory grows. Cassandra adds nodes and rebalances. Vertically: nothing — everything scales out. Read replicas for redirect service in regions with high traffic.

19. Fault Tolerance

Multi-AZ deployment (3 AZs minimum). Redis with replicas + Sentinel failover. Cassandra RF=3 tolerates 1 node failure per shard. Snowflake ID generator: each worker pre-allocates IDs to survive brief coordinator loss. Graceful degradation: if Redis is down, redirect service reads directly from DB (slower but functional).

20. Bottlenecks

- Single Redis cluster can saturate on hot aliases — mitigate with CDN edge caching.
- Snowflake clock skew can cause duplicate IDs — use NTP and reject future timestamps.
- Cassandra compaction pauses — use leveled compaction for read-heavy workload.

21. Trade-offs

- **301 vs 302:** 301 is browser-cached (faster, less load) but loses click analytics. 302 always hits service.
- **Random vs hash-based alias:** Random is unguessable but no locality. Hash of long URL dedups but enumerable.
- **Cassandra vs Postgres:** Scalability vs transactional guarantees (we do not need transactions here).

22. CAP Theorem Analysis

AP. We prioritize availability over consistency — a stale redirect is acceptable; a failed redirect is not. Cassandra tuned to consistency level LOCAL_QUORUM (reads see latest write within region, but cross-region may lag).

23. Security Considerations

Rate limit per IP and per user (token bucket, 100/min). Validate long_url against malware/phishing blocklists (Google Safe Browsing API). HTTPS-only. Sign custom aliases to prevent tampering. Optional: require auth for create, public for redirect.

24. Monitoring & Logging

Golden signals: redirect p99 latency, RPS, error rate (5xx), cache hit ratio. Alert on: hit rate < 80%, p99 > 100ms, error rate > 0.1%. Distributed tracing via OpenTelemetry. Per-alias access logs sampled at 0.1% for debug.

25. Deployment Strategy

Blue-green deploys for stateless services. Cassandra rolling restarts one node at a time. Redis Cluster uses slot migration for zero-downtime scaling. Feature flags for new logic (e.g. custom alias limits).

26. Interview Tips

- Start by clarifying 100:1 read:write ratio — it shapes every later decision.
- Calculate QPS early. "10B reads/mo" sounds big but is only 4K RPS avg — one Redis node handles it.
- Mention 301 vs 302 trade-off explicitly — shows operational maturity.
- Do not forget analytics — real URL shorteners monetize via click data.

27. Common Follow-up Questions

- How would you handle a viral link that suddenly gets 1M clicks/min?
- How would you support custom aliases without collisions?
- What if we need to delete a shortened URL?
- How would you implement expiry?
- How would you migrate from a single Postgres to Cassandra with zero downtime?

28. Common Mistakes

- Using MD5 hash of long_url and truncating — collision-prone.
- Forgetting to deduplicate (same long_url should return same alias for same user).
- Storing all data in one DB without sharding plan.
- Using 302 without explaining the analytics trade-off.

29. Optimization Ideas

- Pre-warm cache for top 1000 aliases daily via batch job.
- Use bloom filters in front of DB to reject non-existent aliases without disk hit.
- Compress long URLs before storage (some are 2KB+).
- Use HTTP/3 (QUIC) for redirect service to cut TLS handshake latency.

30. Final Summary

A URL shortener is the canonical "first system design question" because it touches every major concept — capacity estimation, ID generation, caching, sharding, AP vs CP, and analytics pipelines — without overwhelming complexity. The key insights are (1) the read:write ratio shapes the entire architecture, (2) a 7-char Base62 code is sufficient for trillions of URLs, and (3) caching at multiple layers (CDN, Redis, in-memory) is what delivers sub-50ms latency at scale.

QUESTION 02 BEGINNER

Design Pastebin

1. Problem Statement

Design a service where users can upload plain text and receive a unique URL to share. Visitors fetch the paste via the URL. Support syntax highlighting, expiry, privacy, and a "trending pastes" feature.

2. Functional Requirements

- Upload text content, receive a unique shareable URL.
- Fetch paste content via URL (read-only).
- Optional: syntax highlighting, password protection, expiry, view counter.
- User accounts to manage own pastes (optional).

3. Non-functional Requirements

- Availability 99.95%.
- Read-heavy (10:1 read:write).
- Latency: p99 fetch < 100ms.
- Pastes can be up to 1MB; reject larger.
- Trending pastes computed hourly.

4. Capacity Estimation

5M new pastes/day, avg 10KB each = 50GB/day new content. 50M reads/day. Storage over 5 years: 50GB × 365 × 5 = ~91TB. QPS: ~60 writes/s, ~600 reads/s avg; 6K peak reads.

5. Back-of-the-envelope Math

Storage 91TB needs object storage (S3), not DB. Metadata (URL, owner, expiry, view count) is small (~1KB per paste) = 5M × 1KB × 365 × 5 = ~9TB — fits in sharded Postgres.

6. API Design

```
POST /api/v1/pastes
  body: {"content": "...", "language": "python", "expiry": "1h|1d|1w|never",
         "private": false, "password": null}
  returns: {"url": "https://pb.xy/Ab3xK9", "id": "Ab3xK9"}

GET /{id}
  returns: HTML with syntax-highlighted content (or raw text via Accept header)

GET /api/v1/pastes/trending # top 100 pastes in last hour
```

7. Database Schema

```
Table: pastes
  id VARCHAR(8) PRIMARY KEY
  content_key VARCHAR(64) NOT NULL -- S3 object key
  user_id BIGINT NULL
  language VARCHAR(20) NULL
  private BOOLEAN DEFAULT FALSE
  password_hash VARCHAR(64) NULL
  created_at TIMESTAMP DEFAULT NOW()
  expires_at TIMESTAMP NULL
  view_count BIGINT DEFAULT 0

Table: paste_views (for trending, hourly rollup)
  paste_id, hour, views
```

8. High-Level Design (HLD)

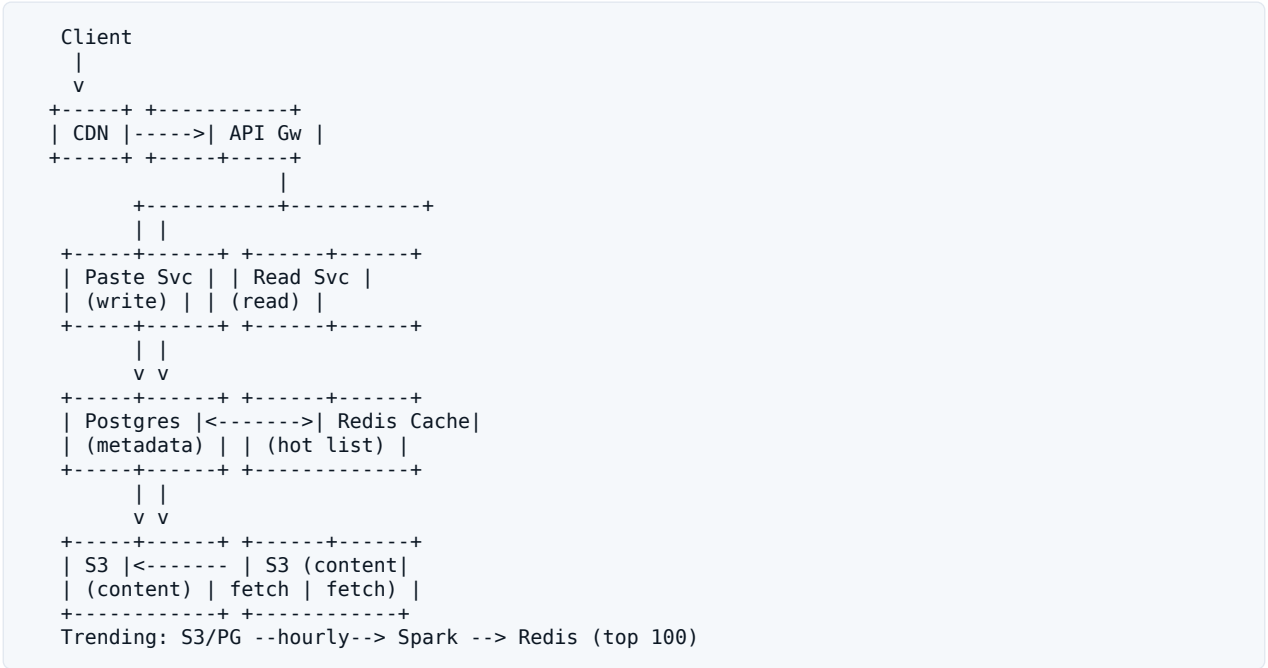
Client → CDN → API Gateway → Paste Service (write) or Read Service → S3 for content, Postgres for metadata. Redis cache for hot pastes and trending list.

9. Low-Level Design (LLD)

Paste Service: 8-char Base62 ID, store content in S3 with key=ID, metadata in Postgres. Read Service: check Redis cache for hot pastes; on miss fetch from S3 and cache. Trending: hourly Spark job aggregates paste_views into top-100 list cached in Redis.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Upload:
  1. POST /pastes → API Gateway → Paste Service.
  2. Generate ID, encrypt content (if password), upload to S3.
  3. Insert metadata in Postgres.
  4. Return URL.

Fetch:
  1. GET /{id} → Read Service via CDN edge.
  2. Lookup metadata in Redis cache (or Postgres).
  3. Fetch content from S3, render with syntax highlighter.
  4. Cache rendered HTML in CDN for 5 min.
  5. Async: increment view_count via Kafka.
```

12. Component Explanation

- **CDN:** Cloudflare; caches public paste HTML at edge.
- **API Gateway:** Kong; auth, rate limit, size validation.
- **Paste Service:** Go/Node; handles uploads, ID generation, S3 writes.
- **Read Service:** Go; optimized for fetches, Redis-backed.
- **Postgres:** Metadata; sharded by ID prefix when >10M pastes.
- **S3:** Object storage for paste content; lifecycle rules for expiry.
- **Redis:** Cache for hot pastes + trending list.
- **Spark:** Hourly batch job for trending computation.

13. Technology Stack

- **Language:** Go for services, JS for syntax highlighter.
- **Storage:** S3 for content, Postgres for metadata.
- **Cache:** Redis Cluster.
- **Queue:** Kafka for view events.
- **Compute:** Kubernetes, multi-AZ.

14. Database Choice

Postgres for metadata (ACID, secondary indexes for user lookups). S3 for content (cheap, durable, lifecycle rules for auto-expiry). Hybrid: do not stuff large text into a relational DB.

15. Cache Strategy

Two-tier: CDN caches public paste HTML for 5 min (handles 80% of reads). Redis caches metadata + raw content for hot pastes (handles 15%). Only 5% reach S3. Trending list cached in Redis, refreshed hourly.

16. Load Balancer Strategy

L7 ALB routes /pastes (writes) and /{id} (reads) to dedicated fleets. Read fleet is 5x larger than write fleet. Health checks: HTTP /health returns 200.

17. Message Queue Usage

Kafka topic **view_events** keyed by paste_id. Consumers increment view_count in DB in batches (every 1000 events or 1 min) to avoid write amplification.

18. Scaling Strategy

Scales horizontally. Paste Service: ~5 replicas. Read Service: ~25 replicas. Postgres: read replicas per shard. Redis Cluster: 3 shards × 3 replicas. S3 auto-scales.

19. Fault Tolerance

Multi-AZ. S3 cross-region replication for DR. Postgres WAL streaming to standby. Redis sentinel failover. Graceful: if S3 is down, return cached content only.

20. Bottlenecks

- S3 PUT latency (~30ms) limits write throughput per instance — mitigate with batching.
- Single Postgres primary on hot users — shard by user_id when needed.
- Trending Spark job can be slow on heavy hours — pre-compute top 1000 then trim.

21. Trade-offs

- **S3 vs DB for content:** S3 is cheaper and unbounded; DB enables full-text search.
- **CDN caching vs privacy:** Public pastes only on CDN; private always hit origin.
- **Sync vs async view count:** Sync gives accurate counter; async is faster but laggy.

22. CAP Theorem Analysis

AP. Availability is critical (users share links expecting them to work). S3 is AP. Postgres sharded with eventual consistency on view counts.

23. Security Considerations

Password-protected pastes use AES-256 with a key derived from password (PBKDF2). Rate limit uploads per IP (10/hour). Scan content for malware (ClamAV). Strip JavaScript from HTML rendering to prevent XSS.

24. Monitoring & Logging

Track upload latency, fetch latency, S3 error rate, cache hit ratio, trending job duration. Alert on: S3 5xx > 1%, fetch p99 > 200ms, trending job > 1h.

25. Deployment Strategy

Kubernetes Deployments with HPA on CPU. Postgres operator for managed failover. S3 lifecycle rules: move to Glacier after 30 days, delete after expiry.

26. Interview Tips

- Distinguish "content storage" from "metadata storage" early — it shapes the design.
- Mention S3 lifecycle rules for expiry — shows operational thinking.
- Discuss CDN caching of HTML but bypass for private pastes.
- Calculate storage in TB not GB — interviewer wants to see big-number comfort.

27. Common Follow-up Questions

- How would you add full-text search across all pastes?

- How would you handle paste edits / version history?
- How would you implement collaborative editing (like Google Docs)?
- How would you detect and remove malware pastes?
- How would you implement paste analytics dashboard for owners?

28. Common Mistakes

- Storing paste content in Postgres — kills DB at 91TB.
- Forgetting to dedupe view counts (F5 spam).
- Not handling paste expiry — storage grows forever.
- Using a single Redis instance without HA plan.

29. Optimization Ideas

- Use S3 Transfer Acceleration for fast uploads from far regions.
- Pre-render syntax-highlighted HTML at upload time, store alongside raw.
- Use bloom filter to reject non-existent IDs without DB lookup.
- Compress content with gzip before S3 PUT (50-70% saving on text).

30. Final Summary

Pastebin is a slight evolution of URL shortener: same overall architecture, but content storage moves from DB to object storage (S3) because paste content is much larger. The key insight is separating metadata (small, relational) from content (large, blob) and choosing the right storage for each. Trending is a classic map-reduce pattern: emit view events to Kafka, aggregate hourly with Spark, cache results in Redis.

QUESTION 03 BEGINNER

Design a Parking Lot System

1. Problem Statement

Design an object-oriented parking lot system. Vehicles (car, motorcycle, truck) enter, are assigned a spot, and pay on exit based on duration and spot type. Support multiple floors, handicap spots, and an admin panel for capacity management.

2. Functional Requirements

- Issue ticket on entry with timestamp and assigned spot.
- Find nearest available spot of correct type for vehicle.
- Calculate fee on exit (hourly rate, daily max, spot-type multiplier).
- Accept cash, card, mobile payment.
- Admin: add/remove spots, view occupancy, set rates.

3. Non-functional Requirements

- Consistency: no two vehicles assigned same spot.
- Availability 99.9% (a downed system blocks gates).
- Real-time spot occupancy (no stale data).
- Payment must be atomic (no double-charge, no free exit).

4. Capacity Estimation

Single parking lot: 1000 spots, 10K vehicles/day, peak 500 entries/hour. Multi-lot operator: 100 lots × 1000 spots = 100K spots total. QPS modest.

5. Back-of-the-envelope Math

Storage trivial. Concurrency is the real challenge: 100 entries/min at peak means 100 simultaneous spot-assignments. Use DB transactions with row-level locking.

6. API Design

```
POST /entry body: {vehicle_type, license_plate} -> {ticket_id, spot_id}
POST /exit/{ticket} body: {payment_method} -> {fee, receipt_id}
GET /spots -> [{floor, type, available_count}]
POST /admin/spots body: {floor, type, count} -> {spot_ids}
PUT /admin/rates/{type} body: {hourly, daily_max}
```

7. Database Schema

```
Table: spots
  spot_id VARCHAR(10) PK -- "F1-A-023"
  lot_id BIGINT
  floor INT
  type ENUM('MOTORCYCLE','COMPACT','LARGE','HANDICAP')
  status ENUM('FREE','OCCUPIED','RESERVED','OUT_OF_SERVICE')

Table: tickets
  ticket_id BIGINT PK (Snowflake)
  spot_id VARCHAR(10) FK
  vehicle_type ENUM
  license_plate VARCHAR(20)
  entry_time TIMESTAMP
  exit_time TIMESTAMP NULL
  fee_cents INT NULL
  payment_id VARCHAR(64) NULL
```

8. High-Level Design (HLD)

A single-tenant system: Entry Kiosk → Parking Service (Go) → Postgres (single instance sufficient). Exit Kiosk → Parking Service → Payment Gateway → Postgres. Admin Web → Parking Service. Real-time occupancy view via WebSocket push from service.

9. Low-Level Design (LLD)

Object model: ParkingLot has List; Floor has List; Spot has type and status. SpotAssignmentStrategy pattern: NearestFirstStrategy, HandicapFirstStrategy. FeeStrategy pattern: HourlyRateStrategy, FlatRateStrategy. Use DB row-level lock (SELECT ... FOR UPDATE) on spot to atomically assign.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Entry:
  1. Kiosk POST /entry with vehicle type.
  2. Service starts DB transaction.
  3. SELECT nearest FREE spot of matching type FOR UPDATE.
  4. UPDATE spots SET status='OCCUPIED' WHERE spot_id=?.
  5. INSERT ticket. COMMIT. Return ticket + spot.
  6. Push occupancy update via WebSocket.

Exit:
  1. Kiosk POST /exit/{ticket}.
  2. Compute fee from entry_time, exit_time, spot type.
  3. Call Payment Gateway (idempotent with receipt_id).
  4. On success: UPDATE ticket SET exit_time, fee, payment_id. UPDATE spot FREE.
  5. Push occupancy update.
```

12. Component Explanation

- **Parking Service:** Core business logic; stateless; multiple instances OK.
- **Payment Gateway:** Stripe wrapper; idempotency keys; receipt storage.
- **Postgres:** ACID source of truth; row locks for atomic assignment.
- **WebSocket Hub:** Pushes occupancy to admin dashboards.
- **Kiosk Client:** Embedded app on entry/exit hardware.

13. Technology Stack

- **Language:** Java (Spring Boot) or Go.
- **DB:** PostgreSQL.
- **Cache:** Not needed at this scale.
- **Realtime:** WebSocket (Socket.io or native).
- **Deployment:** Single VM or small K8s cluster per lot.

14. Database Choice

Postgres for ACID. The atomicity of "assign spot + create ticket" is non-negotiable. NoSQL would require complex compensating transactions.

15. Cache Strategy

Optional: Redis cache for occupancy counts displayed on entry signage. Refresh every 5s. Source of truth remains Postgres.

16. Load Balancer Strategy

Single lot: no LB needed (1 server + standby). Multi-lot: per-lot DNS, regional LB for cross-lot admin.

17. Message Queue Usage

Optional: Kafka for audit events (entries, exits, payments) fed to data warehouse for business analytics. Not on critical path.

18. Scaling Strategy

Single lot fits on one box. Multi-lot: shard by lot_id; each lot has its own DB instance. Centralized reporting via CDC into a warehouse.

19. Fault Tolerance

Active-passive standby with synchronous WAL replication. Failover < 30s. Kiosks have local offline mode (print manual ticket, sync on recovery).

20. Bottlenecks

- Concurrent entry spikes (post-event) saturate single DB — mitigate with read replicas for lookups, primary only for writes.
- Payment Gateway latency blocks exit gate — mitigate with offline payment (cash), pre-pay via app.

21. Trade-offs

- **DB locks vs NoSQL:** Locks are simpler and correct; NoSQL is faster but requires app-level coordination.
- **Synchronous vs async payment:** Sync blocks exit; async risks unpaid exits.
- **Centralized vs per-lot DB:** Centralized simplifies ops but couples lots; per-lot isolates faults.

22. CAP Theorem Analysis

CP. Strong consistency is mandatory — two vehicles cannot occupy the same spot. During network partition between kiosk and DB, kiosk enters offline mode (manual ticket).

23. Security Considerations

TLS for all traffic. Kiosk certs (mTLS) to prevent rogue devices. Payment tokenization (PCI compliance). Admin auth via SSO + 2FA. Audit log for rate changes.

24. Monitoring & Logging

Track entry/exit throughput, payment success rate, DB lock wait time, kiosk heartbeat. Alert on: payment failure > 1%, kiosk offline > 5 min, DB CPU > 80%.

25. Deployment Strategy

Blue-green for service. DB failover via Patroni. Kiosk firmware OTA via signed bundles.

26. Interview Tips

- Clarify single-lot vs multi-lot operator early — different scales.
- Emphasize ACID for spot assignment — interviewer wants to see consistency reasoning.
- Discuss offline mode for kiosks — real-world operational thinking.
- Mention fee strategy pattern — shows OOP design maturity.

27. Common Follow-up Questions

- How would you handle a sold-out lot (queue or reject)?
- How would you implement dynamic pricing (surge)?
- How would you support EV charging spots with timed slots?
- How would you detect fraud (same plate entering twice)?
- How would you add a mobile "reserve spot" feature?

28. Common Mistakes

- Using in-memory only (data lost on crash).
- Forgetting concurrency control (two cars get same spot).
- Hardcoding rates instead of using a strategy/rule engine.
- Coupling payment into the main flow without idempotency.

29. Optimization Ideas

- Pre-allocate tickets at kiosk boot to survive DB blips.
- Use license-plate OCR for frictionless entry/exit (no ticket).
- Predictive occupancy for "parking app" use case (show nearby lots with capacity).
- Dynamic pricing based on real-time occupancy (surge).

30. Final Summary

A parking lot is a classic object-oriented design question that tests the candidate's ability to model a real-world domain and reason about concurrency. The key technical challenge is atomic spot assignment under concurrent entries, solved with DB row-level locks (or a more sophisticated distributed lock for multi-lot). The architecture is deliberately simple — a single ACID database is correct — but the design must acknowledge real-world concerns like kiosk offline mode and payment idempotency.

QUESTION 04 BEGINNER

Design a Library Management System

1. Problem Statement

Design a system for a public library to manage books, members, loans, reservations, and fines. Support multiple branches, search, and a member portal.

2. Functional Requirements

- Catalog books with metadata (title, author, ISBN, copies, branch).
- Member registration and account management.
- Checkout (loan) and return with due date calculation.
- Reserve a book that is currently loaned out.
- Calculate and track fines for overdue returns.
- Search by title, author, ISBN, genre.

3. Non-functional Requirements

- Consistency: no double-loan of the same physical copy.
- Availability 99.5% (library tolerates downtime; not life-critical).
- Search latency < 500ms.
- Audit trail for all loans/returns (compliance).

4. Capacity Estimation

1M books, 100K active members, 50 branches. 10K checkouts/day. Storage trivial; QPS modest. Search index ~1M documents.

5. Back-of-the-envelope Math

DB size ~5GB. Search index ~500MB. Single Postgres + Elasticsearch handles everything.

6. API Design

```
POST /members body: {name, email, address}
GET /books?query=...&page=1 -> [{book_id, title, author, available_count}]
POST /loans body: {book_id, member_id, branch_id}
POST /returns/{loan_id}
POST /reservations body: {book_id, member_id}
GET /members/{id}/fines
```

7. Database Schema

```
Table: books
  book_id BIGINT PK
  isbn VARCHAR(13) UNIQUE
  title TEXT
  author TEXT
  genre VARCHAR(50)
  published_year INT

Table: book_copies
  copy_id BIGINT PK
  book_id BIGINT FK
  branch_id BIGINT FK
  status ENUM('AVAILABLE','LOANED','RESERVED','LOST')

Table: loans
  loan_id BIGINT PK
  copy_id BIGINT FK
  member_id BIGINT FK
  checkout_date DATE
  due_date DATE
  return_date DATE NULL
  fine_cents INT DEFAULT 0
```

8. High-Level Design (HLD)

Web/Mobile → API Gateway → Library Service → Postgres + Elasticsearch. Cron job computes fines nightly. Email service sends due-date reminders.

9. Low-Level Design (LLD)

Use Service Layer pattern: BookService, LoanService, MemberService. Loan checkout is a transactional operation: SELECT copy FOR UPDATE, UPDATE copy status, INSERT loan. Search delegates to Elasticsearch with denormalized book_copy documents.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Checkout:
  1. POST /loans with book_id, member_id, branch_id.
  2. Verify member has < max_loans and no overdue fines.
  3. SELECT an AVAILABLE copy of book at branch FOR UPDATE.
  4. UPDATE copy status = LOANED.
  5. INSERT loan with due_date = today + 14 days.
  6. COMMIT. Return loan_id.

Return:
  1. POST /returns/{loan_id}.
  2. UPDATE loan SET return_date = today.
  3. If today > due_date: calculate fine, store in loan.
  4. UPDATE copy status = AVAILABLE.
  5. If reservation exists for book: notify next member.
```

12. Component Explanation

- **Library Service:** Core logic; transactional; Java/Spring or Python/FastAPI.
- **Postgres:** ACID source of truth for loans/copies.
- **Elasticsearch:** Search index; denormalized book documents.
- **Cron Job:** Nightly: compute fines, send reminders.
- **Email Service:** SendGrid wrapper for notifications.

13. Technology Stack

- **Language:** Java (Spring Boot) — enterprise default.
- **DB:** PostgreSQL.
- **Search:** Elasticsearch.
- **Cache:** Optional Redis for hot book lookups.
- **Deployment:** VMs or small K8s.

14. Database Choice

Postgres for ACID transactions. Elasticsearch for search (denormalized, eventually consistent with PG via CDC).

15. Cache Strategy

Optional Redis for hot books (bestsellers). TTL 5 min. Invalidate on checkout/return.

16. Load Balancer Strategy

L7 LB; small fleet (3 instances) for HA. Session affinity not needed (stateless).

17. Message Queue Usage

Optional: Kafka for events (loan, return, fine) feeding analytics. Not on critical path.

18. Scaling Strategy

Single Postgres with read replicas handles 10K checkouts/day easily. Search scales horizontally in ES clusters.

19. Fault Tolerance

Active-passive Postgres with WAL streaming. ES cluster of 3 nodes. Service multi-AZ.

20. Bottlenecks

- Search latency on common queries — mitigate with ES warmers and good analyzers.
- Fine calculation cron can be slow on large backlogs — batch and parallelize.

21. Trade-offs

- **ES vs PG full-text:** ES is purpose-built but adds an eventually-consistent copy. PG full-text is simpler.
- **Cron vs event-driven fines:** Cron is simpler; event-driven is real-time but more complex.
- **Branch-local vs centralized DB:** Centralized simplifies cross-branch search; local isolates.

22. CAP Theorem Analysis

CP for loans (cannot double-loan). Search (ES) is AP — stale availability info is OK.

23. Security Considerations

Member auth via OAuth/SSO. PII encryption at rest (name, email, address). Audit log for all librarian actions (rate changes, manual overrides).

24. Monitoring & Logging

Track checkout latency, search latency, cron duration, ES lag. Alert on: PG replication lag > 5s, search p99 > 1s, cron failure.

25. Deployment Strategy

Blue-green for service. Postgres operator for failover. ES rolling restarts.

26. Interview Tips

- Distinguish "book" (title) from "copy" (physical instance) — the central modeling decision.
- Use SELECT FOR UPDATE for atomic checkout — interviewer wants to see this.
- Discuss ES vs PG full-text search trade-off.
- Mention fine calculation strategy — shows you think about edge cases (holidays, weekends).

27. Common Follow-up Questions

- How would you handle inter-branch loans (book transfers)?
- How would you support ebook lending with DRM?
- How would you implement a recommendation engine ("readers also liked")?
- How would you handle a member with $1000 in unpaid fines?
- How would you scale to 1000 branches nationally?

28. Common Mistakes

- Confusing book vs copy — the most common modeling error.
- Forgetting transaction on checkout (double-loan bug).
- Using NoSQL for loans — ACID is essential.
- Hardcoding fine logic instead of using a strategy.

29. Optimization Ideas

- Pre-compute "available_count" per book for fast UI rendering.
- Cache search results for common queries (5min TTL).
- Batch fine reminders via email (one digest per member).
- Predict demand for stock purchasing decisions (ML on loan history).

30. Final Summary

A library management system is fundamentally a CRUD application with one tricky concurrency problem: atomic loan checkout. The key modeling insight is separating "book" (the title) from "copy" (the physical instance) — a confusion that breaks naive designs. The architecture is intentionally simple (single ACID database), with Elasticsearch added only for search quality. This question tests object modeling and transactional thinking more than distributed-systems scale.

QUESTION 05 BEGINNER

Design a Movie Ticket Booking System (BookMyShow)

1. Problem Statement

Design a system where users can browse movies, select showtimes, choose seats, and book tickets. Support multiple cities, theatres, screens, and showtimes. Handle the "seat hold" problem during concurrent bookings.

2. Functional Requirements

- Browse movies by city, theatre, language, genre.
- View showtimes and seat layout for a show.
- Select seats and hold them temporarily (5 min) for payment.
- Pay via UPI/card/netbanking; receive ticket with QR code.
- Cancel booking with refund (partial/full based on policy).

3. Non-functional Requirements

- Consistency: no two users can book the same seat.
- Availability 99.95% during booking windows.
- Latency: seat map render < 500ms, payment < 3s.
- Handle spikes (blockbuster releases: 100x normal traffic).

4. Capacity Estimation

10M users, 100K shows/day across 1000 theatres. Avg 100 seats/show, 30% occupancy = 3M tickets/day. Peak on release day: 100K concurrent users, 10K bookings/min.

5. Back-of-the-envelope Math

Storage: shows, seats, bookings ~100GB. QPS: 10K bookings/min = 167 RPS sustained, 1K RPS peak. Read heavy on seat maps (10x more views than bookings).

6. API Design

```
GET /cities/{city}/movies -> list of movies currently running
GET /movies/{movie_id}/shows?city=... -> showtimes
GET /shows/{show_id}/seats -> seat layout + availability
POST /shows/{show_id}/holds body: {seat_ids: [...]} -> {hold_id, expires_at}
POST /holds/{hold_id}/pay body: {payment_method} -> {booking_id, ticket_url}
POST /bookings/{booking_id}/cancel -> {refund_amount}
```

7. Database Schema

```
Table: shows
  show_id BIGINT PK
  movie_id BIGINT FK
  screen_id BIGINT FK
  start_time TIMESTAMP
  base_price INT

Table: seats (per show, materialized)
  seat_id BIGINT PK (show_id + row + col)
  show_id BIGINT FK
  row_label CHAR(2)
  col_num INT
  status ENUM('AVAILABLE','HELD','BOOKED','BLOCKED')
  hold_expires_at TIMESTAMP NULL
  booking_id BIGINT NULL

Table: bookings
  booking_id BIGINT PK
  user_id BIGINT
  show_id BIGINT FK
  total_cents INT
  status ENUM('HELD','CONFIRMED','CANCELLED')
  created_at TIMESTAMP
```

8. High-Level Design (HLD)

Mobile/Web → CDN (for static assets) → API Gateway → Booking Service → Postgres (sharded by city) + Redis (seat holds). Payment via Payment Gateway. Notifications via Kafka.

9. Low-Level Design (LLD)

Seat hold is the critical operation: BEGIN TXN, SELECT seats FOR UPDATE, verify AVAILABLE, UPDATE status=HELD with expiry, INSERT booking row with HELD status, COMMIT. Cron job every 30s expires stale holds.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Hold seats:
  1. User selects seats, clicks "Pay".
  2. POST /holds with seat_ids.
  3. BEGIN; SELECT ... FOR UPDATE; check AVAILABLE;
  4. UPDATE seats SET status=HELD, hold_expires_at=NOW()+5min;
  5. INSERT booking (HELD). COMMIT. Return hold_id.

Confirm payment:
  1. POST /holds/{hold_id}/pay.
  2. Call Payment Gateway (idempotent).
  3. On success: UPDATE seats SET status=BOOKED; UPDATE booking SET CONFIRMED.
  4. Generate ticket PDF with QR code. Email/SMS to user.

Hold expiry (cron every 30s):
  SELECT seats WHERE status=HELD AND hold_expires_at<NOW();
  UPDATE status=AVAILABLE; UPDATE booking SET CANCELLED.
```

12. Component Explanation

- **API Gateway:** Kong; rate limit per user (10 holds/min).
- **Browse Service:** Read-only; serves movies/shows/seat-maps; cached in Redis.
- **Booking Service:** Transactional; uses Postgres row locks.
- **Postgres:** Sharded by city; ACID for seat status.
- **Redis:** Cache for browse; not used for hold state (must be ACID).
- **Payment Gateway:** Stripe/Razorpay; idempotency key = hold_id.
- **Notification Service:** Kafka consumer; sends email/SMS/PDF.

13. Technology Stack

- **Language:** Java/Spring or Go.

- **DB:** PostgreSQL sharded by city.
- **Cache:** Redis Cluster.
- **Queue:** Kafka.
- **Search:** Optional Elasticsearch for movie search.
- **Deployment:** Kubernetes multi-AZ.

14. Database Choice

Postgres for ACID. Shard by city (most queries are city-scoped). Each shard has read replicas. No NoSQL — seat booking is fundamentally transactional.

15. Cache Strategy

Redis cache for browse data (movies, showtimes) with TTL 60s. Seat availability is NOT cached — always fetched from Postgres for accuracy. WebSocket pushes seat map updates to all viewers of a show.

16. Load Balancer Strategy

L7 LB with sticky sessions for WebSocket. Round-robin otherwise. Health checks every 5s.

17. Message Queue Usage

Kafka topic for booking events (confirmed, cancelled) consumed by notification service and analytics pipeline.

18. Scaling Strategy

Shard Postgres by city. Browse fleet scales 10x larger than booking fleet. Read replicas absorb browse traffic. Pre-warm caches before blockbuster releases.

19. Fault Tolerance

Multi-AZ Postgres with synchronous replicas. If payment gateway times out, mark booking as PENDING and reconcile via webhook. Idempotency everywhere.

20. Bottlenecks

- Seat hold conflicts on blockbuster shows — mitigate with optimistic UI (gray out held seats via WebSocket).
- Payment gateway latency — 3s timeout, fall back to retry with idempotency.
- Spike on release day — pre-scale booking fleet, queue users virtually (waiting room).

21. Trade-offs

- **DB lock vs Redis atomic:** DB lock is correct and simple; Redis is faster but complicates ACID.
- **5 min hold vs 10 min:** 5 min reduces lost inventory; 10 min reduces payment failures from rush.
- **Synchronous vs async notification:** Sync confirms; async may lag but doesn't block booking.

22. CAP Theorem Analysis

CP for booking (no double-booking). Browse (cache) is AP — stale movie list is fine.

23. Security Considerations

PCI compliance via payment tokenization (no card data in our DB). User auth via OAuth. Anti-bot: rate limit + CAPTCHA for high-demand shows. Audit log for cancellations.

24. Monitoring & Logging

Track hold success rate, payment latency, booking conversion, seat-map latency. Alert on: hold conflict rate > 1%, payment timeout > 5%, DB lock wait > 100ms.

25. Deployment Strategy

Pre-scale for blockbuster releases (capacity tests + manual HPA bump). Blue-green for service. Postgres operator for failover. Payment gateway circuit breaker.

26. Interview Tips

- Emphasize the seat hold problem — it is the core challenge.
- Use SELECT FOR UPDATE on seats — shows transactional thinking.
- Discuss WebSocket for real-time seat map updates — shows UX awareness.
- Mention pre-scaling for blockbuster releases — operational maturity.

27. Common Follow-up Questions

- How would you handle 1M users trying to book the same show (e.g. Taylor Swift concert)?
- How would you implement dynamic pricing (platinum/gold/silver seats)?
- How would you support group bookings with split payment?
- How would you handle a payment that succeeds after hold expiry?
- How would you detect and prevent bot scalping?

28. Common Mistakes

- Caching seat availability in Redis — leads to double-booking on cache miss/invalidation race.
- Using a single Postgres for all cities — shards needed for scale.
- Forgetting idempotency on payment — double-charge on retry.
- No WebSocket for seat map — users see stale data.

29. Optimization Ideas

- Use Redis for seat-lock check (fast path) backed by Postgres (correct path).
- Pre-render seat map HTML at show creation; cache aggressively.
- Batch seat-status updates via WebSocket (1 message/sec/show, not per click).
- Predict demand and pre-allocate capacity (auto-scale before spike).

30. Final Summary

Movie ticket booking is the canonical "seat hold" problem: how to prevent two users from booking the same seat without holding long database locks. The solution is a two-phase pattern — temporary hold (5 min) with row-level locking, then payment confirmation. The architecture is sharded by city (most queries are local), with WebSocket-pushed seat maps for real-time UX. Operational maturity matters: pre-scaling for blockbuster releases, idempotent payments, and bot detection are real differentiators.

QUESTION 06 BEGINNER

Design a Hotel Booking System (OYO/Booking.com)

1. Problem Statement

Design a system where users can search hotels by city/date, view rooms and prices, and book a stay. Handle the inventory problem: rooms are date-scoped (a room can be booked tonight but available tomorrow).

2. Functional Requirements

- Search hotels by city, check-in/out dates, guests, price.
- View room types, amenities, photos, reviews.
- Book a room for a date range with payment.
- Manage inventory: hotels set availability and pricing per date.
- Cancellation and refund per policy.

3. Non-functional Requirements

- Consistency: no double-booking of a room for a date.
- Availability 99.95%.
- Search latency < 500ms.
- Handle seasonal spikes (festivals, holidays).

4. Capacity Estimation

10K hotels, 1M rooms, 100K bookings/day. Search QPS: 10K. Date range queries are expensive — need pre-materialized availability.

5. Back-of-the-envelope Math

Inventory: 1M rooms × 365 days × 2 years = 730M date-room rows. Each row ~50 bytes = 36GB. Search index ~10GB. Booking DB grows 100K/day × 5 years = 180M rows.

6. API Design

```
GET /search?city=...&checkin=...&checkout=...&guests=2 -> [hotel summaries]
GET /hotels/{id}/rooms?checkin=...&checkout=... -> [available room types]
POST /bookings body: {hotel_id, room_type, checkin, checkout, guests, payment}
POST /bookings/{id}/cancel
PUT /hotels/{id}/inventory body: {date, room_type, available, price} (hotelier API)
```

7. Database Schema

```
Table: hotels
  hotel_id, name, city, lat, lng, star_rating, ...

Table: room_inventory (date-scoped)
  hotel_id, room_type, date, total_rooms, available_rooms, price_cents
  PRIMARY KEY (hotel_id, room_type, date)

Table: bookings
  booking_id, user_id, hotel_id, room_type, checkin_date, checkout_date,
  total_cents, status, payment_id, created_at

Table: booking_nights (decomposed for inventory check)
  booking_id, hotel_id, room_type, date -- one row per night booked
```

8. High-Level Design (HLD)

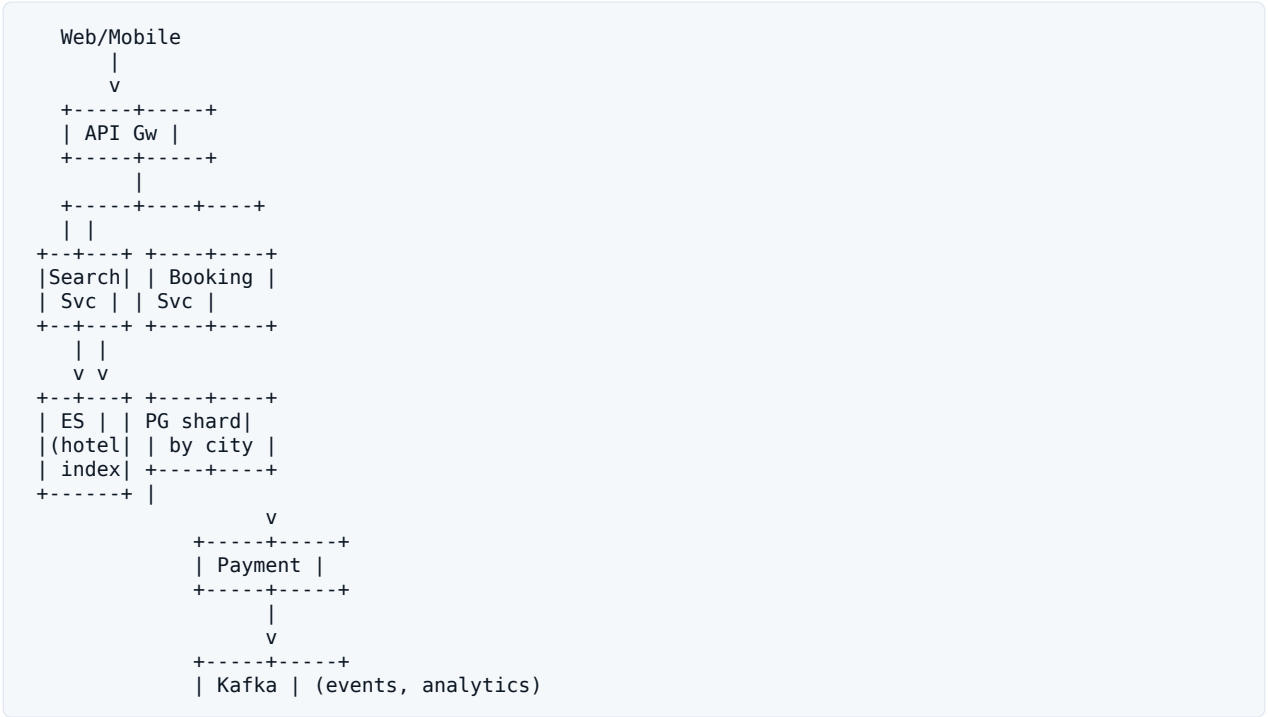
Mobile/Web → CDN → API Gateway → Search Service (Elasticsearch) + Booking Service (Postgres sharded by region) + Inventory Service. Kafka for events. Cron for availability rollups.

9. Low-Level Design (LLD)

Booking transaction: BEGIN; SELECT available_rooms FROM room_inventory FOR UPDATE WHERE hotel_id, room_type, date IN [checkin..checkout); decrement each; INSERT booking and booking_nights; COMMIT. Roll back on payment failure.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Search:
  1. User enters city, dates, guests.
  2. Search Service queries ES for hotels in city.
  3. For each hotel, fetch available room types from room_inventory (date range).
  4. Return ranked list with prices and availability.

Book:
  1. POST /bookings.
  2. BEGIN TXN.
  3. SELECT available_rooms FOR UPDATE for each night in [checkin, checkout).
  4. Verify all > 0. Decrement each. INSERT booking + booking_nights.
  5. COMMIT. Initiate payment.
  6. On payment success: status=CONFIRMED. On failure: rollback inventory, status=CANCELLED.
```

12. Component Explanation

- **Search Service:** Read-heavy; queries ES + inventory cache.
- **Booking Service:** Transactional; row-locks inventory.
- **Inventory Service:** Hotelier-facing; updates room_inventory.
- **Postgres:** Sharded by city; ACID for inventory.
- **Elasticsearch:** Hotel search index (city, amenities, price).
- **Redis:** Cache for inventory summaries (hotel + date range).
- **Kafka:** Booking events for analytics, partner notifications.

13. Technology Stack

- **Language:** Java/Spring or Go.
- **DB:** PostgreSQL sharded by city.
- **Search:** Elasticsearch.
- **Cache:** Redis Cluster.
- **Queue:** Kafka.
- **Deployment:** Kubernetes multi-AZ.

14. Database Choice

Postgres for ACID inventory transactions. Elasticsearch for hotel search (denormalized hotel info, full-text on amenities). The two are kept in sync via CDC.

15. Cache Strategy

Redis caches "hotel availability summary" (rooms_available next 30 days) refreshed every 5 min. The cache is for browse speed; booking always hits Postgres for accuracy.

16. Load Balancer Strategy

L7 LB; sticky session not needed (stateless). Search fleet 5x booking fleet.

17. Message Queue Usage

Kafka topic for booking events (created, cancelled, modified) consumed by analytics and partner (hotel PMS) sync services.

18. Scaling Strategy

Shard by city. Search scales horizontally with ES. Booking scales per-shard. Pre-scale for festival seasons (capacity tests).

19. Fault Tolerance

Multi-AZ Postgres with sync replica. ES 3-node cluster. Payment idempotent via booking_id. Rollback inventory on payment timeout.

20. Bottlenecks

- Date-range inventory queries are expensive — materialize per-date rows.
- High competition for same hotel on festival dates — use SELECT FOR UPDATE and graceful "currently unavailable" messages.
- Search latency with many filters — pre-compute common filter results.

21. Trade-offs

- **Materialized inventory vs on-the-fly:** Materialized is faster but uses more storage.
- **Shard by city vs hotel_id:** City matches query pattern (search is city-scoped).
- **Sync vs async payment:** Sync blocks booking; async risks inventory hold abuse.

22. CAP Theorem Analysis

CP for booking. AP for search (stale availability info is acceptable).

23. Security Considerations

PCI compliance (tokenized payments). Hotelier auth via OAuth + 2FA. Anti-fraud: detect fake bookings, payment fraud, review manipulation. Rate limit booking API.

24. Monitoring & Logging

Track search latency, booking conversion, payment latency, inventory accuracy. Alert on: search p99 > 1s, booking conflict rate > 0.5%, payment timeout > 5%.

25. Deployment Strategy

Blue-green for service. ES rolling restarts. Postgres operator for failover. Pre-scale capacity for known festival dates.

26. Interview Tips

- Emphasize date-scoped inventory — the core modeling insight.
- Use materialized room_inventory per date — shows data modeling maturity.
- Discuss SELECT FOR UPDATE for atomic multi-night booking.
- Mention pre-scaling for festival seasons — operational thinking.

27. Common Follow-up Questions

- How would you handle dynamic pricing (price changes based on demand)?

- How would you support overbooking (airline-style)?
- How would you integrate with hotel PMS systems (real-time sync)?
- How would you implement "rooms are selling out fast" social proof?
- How would you handle multi-room group bookings?

28. Common Mistakes

- Storing inventory as a single row per room (date not modeled).
- Using Redis for booking state (loses ACID).
- Forgetting to lock all nights atomically (partial booking bug).
- No rollback on payment failure (inventory leak).

29. Optimization Ideas

- Use Redis for fast inventory summaries (top 30 days per hotel).
- Pre-compute "popular hotels in city" for fast default search.
- Use CDC to sync inventory to ES for "available hotels" filters.
- Batch booking events for analytics (Kafka producer batching).

30. Final Summary

Hotel booking extends the seat-hold pattern across a date range — the core modeling insight is that room inventory is per-date, not per-room. The booking transaction must lock all dates atomically (a 5-night stay touches 5 inventory rows). Search and booking are separate services with different scaling profiles. The architecture rewards data modeling maturity over distributed-systems heroics.

QUESTION 07

BEGINNER

Design an Elevator System

1. Problem Statement

Design the control system for a bank of elevators in a high-rise building. Optimize for average wait time and energy efficiency. Support dispatch, floor requests, door control, and emergency modes.

2. Functional Requirements

- Passenger presses up/down button on a floor; an elevator is dispatched.
- Passenger presses floor button inside elevator; elevator moves there.
- Multiple elevators coordinate to minimize wait time.
- Door opens/closes safely (sensor-based).
- Emergency: fire mode (return to ground), medical (priority dispatch), power failure.

3. Non-functional Requirements

- Real-time response (< 1s decision).
- Safety: door never closes on obstacle; never opens between floors.
- Availability 99.99% (cannot strand passengers).
- Energy efficiency (avoid moving empty elevators unnecessarily).

4. Capacity Estimation

10-50 elevators, 20-100 floors. ~1000 requests/hour peak. Latency target: dispatch decision < 500ms.

5. Back-of-the-envelope Math

Compute trivial. Real-time constraints dominate. Use embedded controller per elevator + central dispatch coordinator.

6. API Design

```
// Internal elevator API (not REST &mdash; real-time RPC)
requestElevator(floor, direction) -> assigned elevator_id
requestFloor(elevator_id, floor)
getElevatorState(elevator_id) -> {floor, direction, doors, load}
setEmergencyMode(mode) // FIRE, MEDICAL, POWER_OFF
```

7. Database Schema

```
// In-memory state (not DB-backed):
Elevator {
  id, current_floor, direction (UP/DOWN/IDLE),
  target_floors: Set<int>,
  door_state: OPEN/CLOSED,
  load_kg, status: NORMAL/MAINTENANCE/EMERGENCY
}

FloorRequest { floor, direction, timestamp, assigned_elevator_id? }
```

8. High-Level Design (HLD)

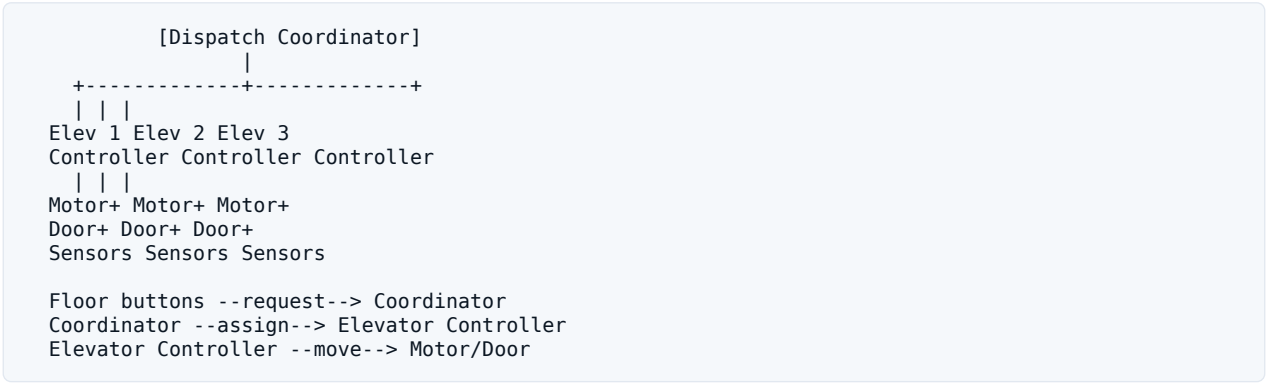
Each elevator has an embedded controller. A central Dispatch Coordinator receives floor requests and assigns elevators via an algorithm (SCAN, LOOK, or zone-based). Communication via in-building network (deterministic latency).

9. Low-Level Design (LLD)

Elevator class with state machine: IDLE → MOVING_UP → DOORS_OPEN → DOORS_CLOSED → MOVING. DispatchStrategy interface: ScanStrategy (elevator continues in current direction until no more requests), LookStrategy (changes direction when no further requests ahead), ZoneStrategy (each elevator owns a floor range).

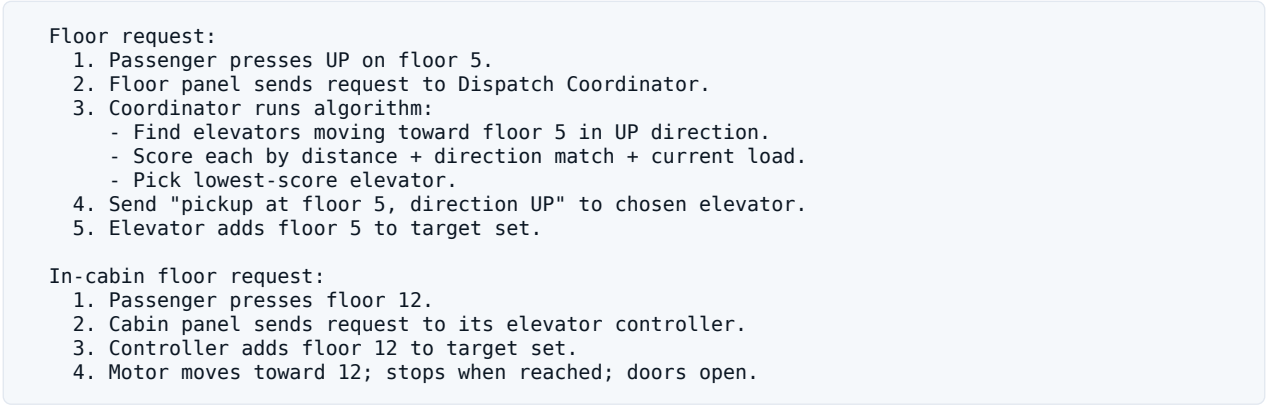
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Dispatch Coordinator:** Central; assigns floor requests to elevators.
- **Elevator Controller:** Per-elevator; manages state machine and hardware.
- **Floor Panel:** Embedded client; sends up/down requests.
- **Cabin Panel:** Embedded client; sends floor selection.
- **Sensors:** Door obstacle, floor arrival, overload, fire alarm.
- **Persistent Store:** Optional: log events for maintenance analytics.

13. Technology Stack

- **Language:** C/C++ for embedded; Java/Python for coordinator.
- **Communication:** Modbus, CAN bus, or MQTT over wired LAN.
- **Storage:** Time-series DB (InfluxDB) for maintenance logs.
- **Deployment:** On-prem hardware; no cloud dependency.

14. Database Choice

No DB on critical path. Time-series DB (InfluxDB) for offline maintenance analytics.

15. Cache Strategy

In-memory state only. No external cache — latency must be deterministic.

16. Load Balancer Strategy

Not applicable — coordinator is active-passive with hot standby.

17. Message Queue Usage

Optional: MQTT for sensor events. Not on critical control path.

18. Scaling Strategy

Single building: one coordinator. Skyscraper complex: per-tower coordinators + a meta-coordinator for cross-tower requests (rare).

19. Fault Tolerance

Coordinator HA: active-passive with heartbeat. If coordinator dies, elevators fall back to autonomous mode (each handles its own cabin requests, ignores hall calls). Hardware fail-safes: brake engages on power loss, doors open at nearest floor.

20. Bottlenecks

- Single coordinator — mitigate with active-passive HA.
- Algorithm quality — LOOK is significantly better than FIFO in wait time.
- Sensor latency — wired sensors, deterministic bus.

21. Trade-offs

- **Central vs distributed dispatch:** Central is simpler; distributed is more resilient.

- **SCAN vs LOOK vs Zone:** LOOK minimizes wait; Zone is fair across elevators.
- **Energy vs wait time:** Idling saves energy but increases wait.

22. CAP Theorem Analysis

CP. Strong consistency required — two elevators cannot both be assigned to the same pickup (causes confusion). Coordinator is single source of truth.

23. Security Considerations

Physical security: locked machine room. Network isolation (no public internet). Emergency override: fire department master key. Audit log for emergency mode triggers.

24. Monitoring & Logging

Track wait time p99, elevator utilization, door open/close cycles, sensor alerts. Alert on: wait time > 60s, elevator offline, sensor anomaly, emergency mode.

25. Deployment Strategy

Hardware controllers flashed via signed firmware. Coordinator deployed via containerized service with rollback capability.

26. Interview Tips

- Recognize this is an OOP + embedded design question, not a distributed-systems one.
- Use Strategy pattern for dispatch algorithm — shows design pattern fluency.
- Discuss state machine for elevator — shows real-time thinking.
- Mention emergency modes — shows you think about safety-critical systems.

27. Common Follow-up Questions

- How would you handle a sudden power failure with passengers in elevators?
- How would you implement "destination dispatch" (passenger enters floor in hall, not cabin)?
- How would you optimize for rush hour (morning UP surge, evening DOWN surge)?
- How would you predict maintenance needs from sensor data?
- How would you handle an elevator stuck between floors?

28. Common Mistakes

- Modeling elevators as REST API — this is a real-time system.
- Ignoring direction of travel in dispatch — massive wait time increase.
- No fallback when coordinator dies — system becomes unavailable.
- Forgetting safety (door obstacle detection).

29. Optimization Ideas

- Destination dispatch (input floor in hall, group by destination) reduces stops 30%.
- Predictive dispatch: learn traffic patterns by time of day.
- Energy-aware: avoid moving empty elevators; use gravity (regenerative braking).

- Load balancing: assign less-loaded elevator when scores are close.

30. Final Summary

Elevator system is a real-time embedded design question that tests OOP maturity and safety-critical thinking. The architecture is intentionally centralized (one coordinator per building) with hardware-level fail-safes. The dispatch algorithm (LOOK, SCAN, zone-based) is the interesting design variable. Interviewers want to see state machines, the Strategy pattern, and explicit reasoning about safety and fallback modes — not web-scale distributed systems thinking.

QUESTION 08

BEGINNER

Design an ATM

1. Problem Statement

Design the software for an ATM. Support cash withdrawal, balance inquiry, deposit, PIN change, and mini-statement. Handle communication with the bank server and the cash dispenser hardware.

2. Functional Requirements

- Authenticate user via card + PIN.
- Cash withdrawal with denomination dispatch.
- Balance inquiry and mini-statement.
- Cash deposit (with denomination validation).
- PIN change.
- Receipt printing.

3. Non-functional Requirements

- Strong consistency: no double-dispense, no double-debit.
- Availability 99.5% (hardware dependent).
- Security: PIN encryption, card data protection, tamper detection.
- Latency: transaction < 10s end-to-end.

4. Capacity Estimation

Single ATM: ~100 transactions/day. Bank network: 10K ATMs × 100 = 1M/day. QPS modest.

5. Back-of-the-envelope Math

Compute trivial. Security and consistency dominate. Each ATM is an embedded client to the bank core.

6. API Design

```
// ATM-to-bank protocol (ISO 8583 or proprietary)
authenticate(card_number, encrypted_pin) -> session_token
getBalance(session_token) -> amount_cents
withdraw(session_token, amount_cents) -> {success, dispensed_denominations}
deposit(session_token, denominations) -> {success, total_cents}
changePin(session_token, new_encrypted_pin)
```

7. Database Schema

```
// ATM local state (in-memory):
ATMState {
  cash_inventory: {100: 500, 200: 500, 500: 200}, // denomination -> count
  current_session: SessionState | null
  hardware_status: {printer_healthy, card_reader_healthy, ...}
}

// Bank-side tables:
accounts(account_id, holder_name, balance_cents, daily_limit_cents)
cards(card_id, account_id, encrypted_pin, status)
transactions(txn_id, account_id, type, amount, timestamp, atm_id, balance_after)
```

8. High-Level Design (HLD)

ATM (embedded controller with card reader, cash dispenser, PIN pad, receipt printer) → Bank Switch (secure gateway) → Core Banking System (transactional DB). Hardware Abstraction Layer (HAL) separates software from devices.

9. Low-Level Design (LLD)

ATMController state machine: IDLE → CARD_INSERTED → PIN_ENTERED → MENU → TRANSACTION → COMPLETED → IDLE. CashWithdrawalHandler: validate amount, check ATM cash inventory, request bank debit, on success dispense cash (atomic with debit), print receipt. If dispense fails after debit, bank must auto-refund.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Withdrawal:
  1. Insert card; ATM reads card number.
  2. Enter PIN; ATM encrypts and sends to bank.
  3. Bank verifies PIN; returns session_token.
  4. User selects "Withdraw", enters amount.
  5. ATM checks local cash inventory.
  6. ATM sends withdraw request to bank.
  7. Bank: BEGIN TXN; SELECT account FOR UPDATE;
     verify balance >= amount + fee; debit; INSERT txn; COMMIT.
  8. Bank returns success.
  9. ATM dispenses cash. Print receipt.
 10. If dispense fails: ATM sends "compensate" to bank; bank refunds.
 11. Eject card. Reset to IDLE.
```

12. Component Explanation

- **ATM Controller:** Embedded software; state machine; orchestrates devices.
- **HAL:** Hardware Abstraction Layer; uniform API for varied device vendors.
- **Bank Switch:** Secure gateway; routes ISO 8583 to core banking.
- **Core Banking:** Transactional system; ACID for accounts.
- **HSM:** Hardware Security Module; encrypts PINs and keys.

13. Technology Stack

- **Language:** C/C++ for ATM embedded; Java/Cobol for core banking.
- **Protocol:** ISO 8583 over TCP.
- **DB:** DB2/Oracle/Postgres for core banking.
- **Security:** HSM, PIN block encryption (DES/AES), TLS.
- **Deployment:** On-prem hardware; no cloud.

14. Database Choice

Core banking uses a relational DB (DB2/Oracle) for ACID transactions. ATM has no persistent store — state is in-memory and reconciled daily with bank.

15. Cache Strategy

None on critical path. ATM may cache last-known-balance for display if bank is unreachable, but never allows withdrawal without bank confirmation.

16. Load Balancer Strategy

Bank switch is a redundant pair (active-active). Core banking uses clustered DB with synchronous replication.

17. Message Queue Usage

Optional: async event stream of completed transactions for fraud detection and analytics.

18. Scaling Strategy

Bank-side scales with transaction volume. ATM-side is single-instance per machine.

19. Fault Tolerance

If ATM loses power mid-transaction: hardware holds last state; on reboot, completes or compensates. If bank unreachable: ATM shows "Service unavailable", ejects card.

20. Bottlenecks

- Bank core DB write throughput — partition by region.
- Network latency ATM-to-bank — use leased lines, not internet.
- Cash inventory limits withdrawals — daily cash replenishment logistics.

21. Trade-offs

- **Dispense-then-debit vs debit-then-dispense:** Debit-then-dispense is safer (no overdraft) but requires compensation if dispense fails.
- **Online vs offline authorization:** Online is safer; offline allows limited withdrawal during bank downtime.
- **PIN block encryption (DES vs AES):** AES is modern; DES is legacy compatibility.

22. CAP Theorem Analysis

CP. Strong consistency mandatory — no double-dispense or double-debit. During network partition, ATM refuses transactions (fail-safe).

23. Security Considerations

PIN encrypted at PIN pad (never sent in clear). HSM in bank decrypts. Card number tokenized in bank DB. Tamper sensors on ATM (seismic, light, temperature) trigger auto-shutdown. Anti-skimming hardware. Audit log for all access attempts.

24. Monitoring & Logging

Track transaction success rate, dispense failures, hardware errors, cash levels. Alert on: dispense failure after debit (auto-refund), cash < 20%, sensor alert.

25. Deployment Strategy

ATM firmware signed and OTA-updatable. Bank core: rolling upgrades with sync replicas. Encryption keys rotated quarterly via HSM ceremony.

26. Interview Tips

- Recognize this is embedded + transactional, not a distributed-systems question.
- Walk through the withdrawal flow step-by-step — interviewer wants to see compensation logic.
- Discuss PIN encryption at the PIN pad — shows security thinking.
- Mention hardware abstraction layer — shows real-world awareness.

27. Common Follow-up Questions

- What if the ATM dispenses cash but the network fails before confirming to the bank?
- How would you detect skimming devices?
- How would you handle cash deposit with cheque imaging?
- How would you scale to 100K ATMs nationally?

- How would you detect fraud (card cloning, unusual withdrawal patterns)?

28. Common Mistakes

- Dispensing before debiting — causes overdraft if ATM offline.
- Forgetting compensation for dispense-after-debit failure.
- Sending PIN in plaintext over the network.
- No audit log for security events.

29. Optimization Ideas

- Pre-stage common operations (read card while user enters PIN).
- Cash recycling: deposited cash is dispensed to next withdrawer (fewer replenishment trips).
- Predict cash demand per ATM per day (avoid runouts).
- Mobile-app withdrawal: pre-stage on phone, ATM dispenses on tap (cardless).

30. Final Summary

An ATM is a classic embedded + transactional design question. The key challenge is atomicity across two systems (ATM hardware and bank core) connected by an unreliable network. The standard solution is debit-then-dispense with a compensation flow when dispense fails. Security (PIN encryption, HSM, tamper detection) is non-negotiable. Interviewers want to see state-machine thinking, compensation logic, and security maturity — not web-scale distributed systems.

QUESTION 09

BEGINNER

Design a Chat Application (WhatsApp)

1. Problem Statement

Design a 1:1 and group chat application. Users send text/media messages to contacts in real-time. Support offline message delivery, read receipts, and last-seen status.

2. Functional Requirements

- 1:1 and group chat (up to 256 members).
- Send text, image, video, voice messages.
- Real-time delivery (sub-second).
- Offline messages delivered when user comes online.
- Read receipts (blue ticks) and last-seen timestamp.

3. Non-functional Requirements

- Latency: message delivery p99 < 1s.
- Availability 99.99%.
- Messages must not be lost (durability).
- Ordering: messages from same sender must be in order.
- End-to-end encryption (optional, but expected in modern apps).

4. Capacity Estimation

1B users, 100M daily active, 50B messages/day. Avg message 200 bytes. Storage: 50B × 200B × 365 × 5 = ~1.8PB over 5 years. QPS: ~600K msgs/s.

5. Back-of-the-envelope Math

1.8PB needs sharded storage. 600K msgs/s needs ~600 message-router instances (1K msgs/s each). Connection count: 100M concurrent WebSocket connections — 50 connection servers handling 2M each.

6. API Design

```
// WebSocket protocol (after HTTP upgrade):
// Client -> Server
{"type":"send", "to":"user_id", "msg_id":"...", "content":"..."}
{"type":"ack", "msg_id":"..."}
{"type":"read", "msg_id":"..."}

// Server -> Client
{"type":"message", "from":"user_id", "msg_id":"...", "content":"..."}
{"type":"delivered", "msg_id":"..."}
{"type":"read", "msg_id":"..."}

// REST APIs
POST /media/upload -> presigned S3 URL
GET /messages?since=... -> catch up after reconnect
```

7. Database Schema

```
// Sharded by conversation_id
Table: messages
  msg_id BIGINT PK (Snowflake)
  conversation_id BIGINT -- shard key
  sender_id BIGINT
  type ENUM('TEXT','IMAGE','VIDEO','AUDIO')
  content TEXT / media_url
  timestamp BIGINT (ms)
  status ENUM('SENT','DELIVERED','READ')

Table: conversations
  conversation_id BIGINT PK
  type ENUM('1:1','GROUP')
  members JSON -- [user_id, ...]
  last_msg_at BIGINT
```

8. High-Level Design (HLD)

Mobile/Web → Connection Servers (WebSocket fanout) → Message Router → Storage (sharded) + Push Notification Service (FCM/APNs) for offline users. Media via CDN + S3.

9. Low-Level Design (LLD)

Message Router: receives SEND from sender's connection server, looks up recipients, fans out to recipient connection servers (if online) or queues for push (if offline). Writes to sharded storage. Conversation sharded by hash(conversation_id).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Send message:
  1. Sender types message, app sends via WebSocket to its Conn Svc.
  2. Conn Svc forwards to Message Router.
  3. Router: persist to sharded storage (key=conversation_id).
  4. Router: lookup recipient Conn Svc (from session registry in Redis).
  5. If recipient online: push to recipient Conn Svc → recipient.
  6. If offline: enqueue push notification (FCM/APNs).
  7. Router: send DELIVERED ack to sender.
  8. Recipient app: display, send READ receipt when user opens.
```

12. Component Explanation

- **Connection Service:** Holds WebSocket connections; routes inbound msgs to Message Router.
- **Message Router:** Stateless; persists + fans out.
- **Session Registry:** Redis; user_id → conn_svc_id mapping.
- **Sharded Storage:** Cassandra or sharded Postgres; conversation-scoped.
- **Push Service:** FCM (Android), APNs (iOS); for offline users.
- **Media Service:** S3 + CDN for images/videos; presigned URLs for upload.

13. Technology Stack

- **Language:** Erlang/Elixir (WhatsApp original) or Go.
- **DB:** Cassandra (write-heavy, horizontal scale).
- **Cache:** Redis (sessions, last-seen).
- **Realtime:** WebSocket / MQTT.
- **Push:** FCM + APNs.
- **Media:** S3 + CloudFront.

14. Database Choice

Cassandra for messages (write-heavy, partitioned by conversation_id, time-ordered within partition). Redis for ephemeral session state.

15. Cache Strategy

Redis caches: (1) online user → connection server mapping, (2) last-seen timestamps, (3) recent messages per conversation (last 50).

16. Load Balancer Strategy

L7 LB with sticky sessions (WebSocket connection stays on one server). Health check on /health endpoint.

17. Message Queue Usage

Kafka for message persistence pipeline (router → Kafka → storage consumer). Decouples write latency from delivery latency.

18. Scaling Strategy

Connection Service scales by connection count (50 servers × 2M conns). Message Router scales by msg/s. Cassandra scales by partition.

19. Fault Tolerance

Multi-AZ. Connection Service: if a server dies, clients reconnect to another (replay missed msgs via REST). Cassandra RF=3. Push service has retry+TTL.

20. Bottlenecks

- Single WebSocket server fanout limit — load balance by hash(user_id).
- Group fanout amplifies writes — 256-member group = 256 fanout per message.
- Cassandra hot partitions — avoid high-velocity single conversations.

21. Trade-offs

- **WebSocket vs MQTT:** WebSocket is web-friendly; MQTT is more efficient on mobile.
- **End-to-end encryption vs server features:** E2E blocks server-side search and analytics.
- **Persist-then-deliver vs deliver-then-persist:** Persist-first ensures durability; deliver-first is faster but risky.

22. CAP Theorem Analysis

AP. Messages can be delivered out of order across senders, but in-order per sender (sequence number per sender). Eventual consistency on read receipts.

23. Security Considerations

TLS for transport. End-to-end encryption via Signal Protocol (Double Ratchet) for modern apps. Media encrypted at rest in S3. Authentication via OAuth/JWT.

24. Monitoring & Logging

Track message delivery latency, delivery success rate, connection count, push notification latency. Alert on: delivery p99 > 5s, connection drop rate > 1%/min, storage write failures.

25. Deployment Strategy

Blue-green for stateless services. Cassandra rolling restarts. Connection Service uses graceful drain (notify clients to reconnect before shutdown).

26. Interview Tips

- Distinguish connection layer (WebSocket) from message routing layer — key architectural decision.
- Discuss offline message delivery via push notifications — real-world thinking.
- Mention end-to-end encryption — security maturity.
- Calculate connection count: 100M concurrent needs many servers — shows scale awareness.

27. Common Follow-up Questions

- How would you implement message search across history?
- How would you handle voice/video calls?
- How would you implement "delete for everyone"?
- How would you scale to 10B messages/day?
- How would you implement message forwarding and quoted replies?

28. Common Mistakes

- Using a single Postgres — won't scale to 1B users.
- Forgetting offline delivery (push notifications).
- No message ordering guarantee per sender.
- Synchronous media upload on send path — blocks message.

29. Optimization Ideas

- Use MQTT instead of WebSocket for mobile (lower battery).
- Batch message acks (1 ack per 100 messages).
- Compress images/videos before upload (saves bandwidth + storage).
- Pre-fetch recent conversations on app launch (warm cache).

30. Final Summary

A chat application is a real-time fanout system. The key architectural decision is separating the connection layer (millions of long-lived WebSocket connections) from the message routing layer (stateless, horizontally scalable). Message persistence must be sharded by conversation_id for ordering and locality. Offline delivery via push notifications is essential. End-to-end encryption is expected in modern apps and constrains server-side features.

QUESTION 10 BEGINNER

Design a Vending Machine

1. Problem Statement

Design the software for a vending machine. Accept coins/bills/cards, dispense products, return change, and report inventory. Handle edge cases like insufficient change, sold-out products, and partial payment refunds.

2. Functional Requirements

- Display product list with prices and availability.
- Accept payment (coins, bills, card, NFC).
- Dispense selected product and return change.
- Refund on cancel or insufficient stock.
- Report inventory and cash levels to backend.

3. Non-functional Requirements

- Strong consistency: no free product, no lost money.
- Availability 99% (hardware dependent).
- Real-time response (< 3s per transaction).
- Offline operation during network outages.

4. Capacity Estimation

Single machine: 100 transactions/day. Backend: 10K machines × 100 = 1M/day.

5. Back-of-the-envelope Math

Compute trivial. State machine + hardware integration dominate.

6. API Design

```
// Local state machine (no remote calls during transaction):
selectProduct(product_id) -> price_cents
insertCoin(denomination) -> current_balance_cents
insertBill(denomination) -> current_balance_cents
swipeCard(payment_token) -> authorization_result
confirmPurchase() -> {product_dispensed, change_dispensed}
cancel() -> refund_cents

// Backend sync (batch, nightly):
POST /machines/{id}/report body: {sales, inventory, cash_levels}
```

7. Database Schema

```
// Local state (in-memory + daily-synced to backend):
VendingMachine {
  products: [{id, name, price_cents, stock_count}],
  cash_inventory: {denomination: count},
  card_reader_status, bill_validator_status,
  current_transaction: TransactionState | null
}

TransactionState {
  selected_product_id,
  inserted_amount_cents,
  payment_method: CASH | CARD,
  state: AWAITING_PAYMENT | READY_TO_DISPENSE | DISPENSING | DONE
}
```

8. High-Level Design (HLD)

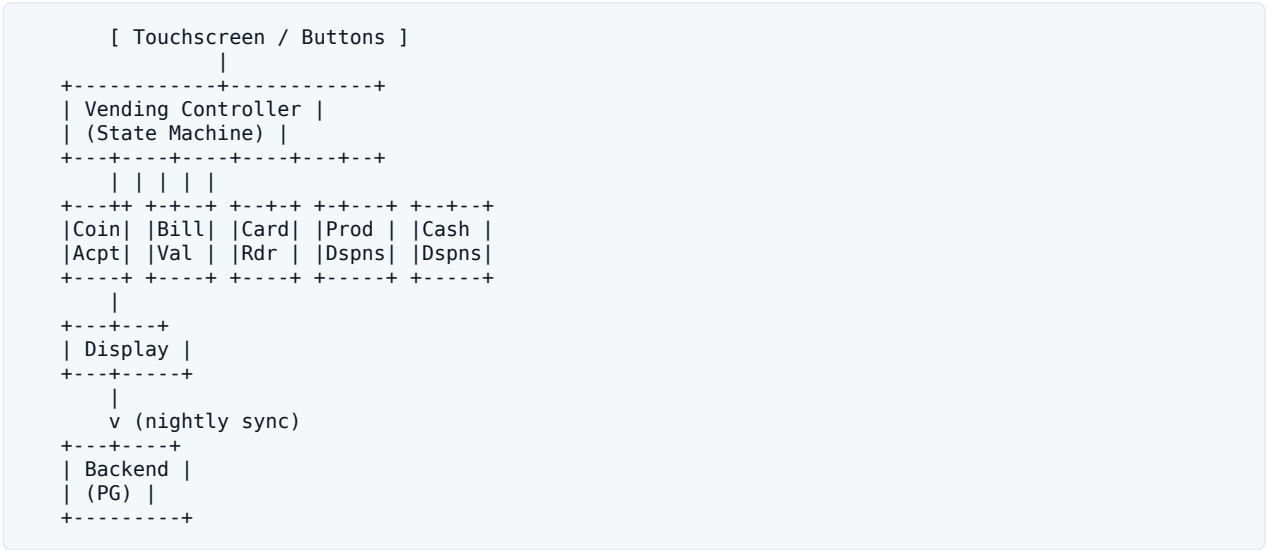
Vending Machine Controller (embedded) with hardware: bill/coin acceptor, card reader, product dispenser, change dispenser, display. Syncs daily with backend over cellular network for inventory and sales reporting.

9. Low-Level Design (LLD)

State machine: IDLE → PRODUCT_SELECTED → PAYMENT_IN_PROGRESS → DISPENSING → RETURN_CHANGE → IDLE. InventoryStrategy: stop accepting selections when stock=0. ChangeStrategy: greedy (largest denominations first). RefundStrategy: return exact inserted amount.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Cash purchase:
  1. User selects product. Controller checks stock > 0 and shows price.
  2. User inserts coins/bills. Controller tracks balance.
  3. When balance >= price, "Confirm" button enables.
  4. User confirms. Controller:
     a. Decrement product stock.
     b. Compute change = balance - price.
     c. Verify change can be made (enough denominations).
     d. Dispense product.
     e. Dispense change.
     f. Record sale in local log.
  5. Return to IDLE.

Cancel:
  1. User presses Cancel anytime after inserting money.
  2. Controller refunds exact inserted amount using available denominations.
  3. If exact refund impossible: display "Use exact change" warning, attempt best-effort.
```

12. Component Explanation

- **Vending Controller:** Embedded software; state machine.
- **Coin Acceptor:** Hardware; validates coin by size/weight/magnetism.
- **Bill Validator:** Hardware; optical + magnetic validation.
- **Card Reader:** NFC + magnetic stripe; talks to payment gateway.
- **Product Dispenser:** Motorized coil per slot; rotates to release product.
- **Change Dispenser:** Tube-based; dispenses coins by denomination.
- **Display:** LCD/touchscreen for product selection and prompts.

13. Technology Stack

- **Language:** C/C++ for embedded controller.
- **Backend:** Java/Python; PostgreSQL.
- **Communication:** Cellular (4G/LTE) for backend sync.
- **Payment:** Stripe/Adyen for card payments.
- **Deployment:** On-prem hardware; backend in cloud.

14. Database Choice

No DB on the machine (in-memory + log file). PostgreSQL backend for sales, inventory, machine health across the fleet.

15. Cache Strategy

None on critical path.

16. Load Balancer Strategy

Backend LB across multiple instances. Machine-to-backend is request/response; no LB needed on machine side.

17. Message Queue Usage

Optional: Kafka for sales events fed to analytics.

18. Scaling Strategy

Each machine is independent. Backend scales horizontally for fleet reporting.

19. Fault Tolerance

If network down: machine operates offline, syncs later. If card reader fails: display "Cash only". If change dispenser jams: display "Exact change only".

20. Bottlenecks

- Change inventory limits refund capability — daily replenishment.
- Card reader latency (~3s) — acceptable but slow vs cash.
- Hardware jams — sensor detects and reports to backend.

21. Trade-offs

- **Cash vs cashless:** Cash adds change complexity; cashless simplifies but needs network.
- **Offline vs online authorization:** Offline is faster; online is safer.
- **Stock-then-pay vs pay-then-stock:** Stock-then-pay avoids refund; pay-then-stock is faster.

22. CAP Theorem Analysis

CP. Strong consistency mandatory — no free products, no lost money. During network partition, machine operates offline with cached product data and reconciles later.

23. Security Considerations

TLS for backend sync. Card payments tokenized (PCI). Physical security: locked cash box, tamper sensors. Audit log for all transactions and refunds.

24. Monitoring & Logging

Track sales per machine, stock levels, cash levels, hardware errors. Alert on: stock < threshold, cash box > 80% full, hardware error, no sales in 24h.

25. Deployment Strategy

Firmware OTA via signed bundles. Backend blue-green. Daily cash/stock replenishment routes computed by backend (TSP optimization).

26. Interview Tips

- Recognize this is OOP + embedded, not distributed systems.
- Walk through the state machine explicitly — interviewer wants to see states and transitions.
- Handle the "cannot make change" edge case — shows thoroughness.
- Discuss offline operation — real-world thinking.

27. Common Follow-up Questions

- How would you handle a paper jam in the bill validator mid-transaction?
- How would you implement dynamic pricing (happy hour discounts)?
- How would you support mobile app payment (scan QR to pay)?

- How would you predict replenishment routes from sales data?
- How would you detect fraud (counterfeit coins, card cloning)?

28. Common Mistakes

- Dispensing product before completing payment — money lost if payment fails.
- No "cannot make change" handling — user gets stuck.
- No audit log — cannot reconcile cash at end of day.
- Single-threaded state machine blocks on card authorization — UI freezes.

29. Optimization Ideas

- Pre-validate coins in parallel with display update.
- Cache common product combos for fast re-selection.
- Predict stock-out and warn backend before it happens.
- Dynamic pricing based on time-of-day demand.

30. Final Summary

A vending machine is an OOP + embedded design question that tests state-machine thinking and edge-case handling. The critical concern is consistency: no free products, no lost money. The architecture is intentionally simple — a controller state machine orchestrating hardware devices, with offline operation and nightly backend sync. Edge cases (insufficient change, hardware jams, network outages) must be handled gracefully. Interviewers want to see explicit state transitions and refund logic.

QUESTION 11

BEGINNER

Design a Blackjack Game

1. Problem Statement

Design a multiplayer blackjack game. Support a dealer, multiple players per table, betting, hit/stand/double/split actions, and a real-time UI showing each player's hand.

2. Functional Requirements

- Create/join a table (up to 7 players + 1 dealer).
- Place bets, deal cards, perform hit/stand/double/split.
- Real-time sync of game state across all players.
- Settle bets at round end (win/lose/push/blackjack payouts).
- Maintain player balance and game history.

3. Non-functional Requirements

- Latency: state update broadcast < 500ms.
- Consistency: all players see same game state.
- Availability 99.5% (it is a game, not life-critical).
- Fairness: RNG must be cryptographically secure.

4. Capacity Estimation

10K concurrent tables × 7 players = 70K concurrent users. ~1M games/day. Storage for game history: ~1GB/day.

5. Back-of-the-envelope Math

Compute trivial. Real-time fanout via WebSocket per table is the main concern.

6. API Design

```
POST /tables/join body: {user_id, buy_in} -> {table_id, seat}
POST /tables/{id}/bet body: {amount}
POST /tables/{id}/action body: {action: HIT|STAND|DOUBLE|SPLIT}

// WebSocket /tables/{id}/stream (real-time state updates)
// server -> client: {type:"state", players:[...], dealer:[...], turn}
// client -> server: {type:"action", action:"HIT"}
```

7. Database Schema

```
Table: users (user_id, balance_cents, created_at)
Table: tables (table_id, status, current_turn, created_at)
Table: players (table_id, user_id, seat, bet_cents, hand_json, status)
Table: games (game_id, table_id, deck_seed, started_at, ended_at, result_json)
```

8. High-Level Design (HLD)

Web/Mobile → API Gateway → Game Service (manages tables) → WebSocket Hub for real-time broadcast → Postgres for persistence. RNG service for fair shuffle.

9. Low-Level Design (LLD)

Game Service holds in-memory table state (faster than DB for active games). State machine per table: WAITING → BETTING → DEALING → PLAYER_TURNS → DEALER_TURN → SETTLE → WAITING. GameEngine class enforces rules (card values, bust, blackjack 3:2 payout).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Round:
  1. Players join table, place bets within 30s window.
  2. Game Service deals 2 cards each, 1 to dealer (1 hidden).
  3. Each player in turn: HIT/STAND/DOUBLE/SPLIT.
  4. After all players, dealer plays (hit until 17).
  5. Settle: each player wins/loses/pushes based on hand vs dealer.
  6. Update balances, broadcast final state, persist history, reset for next round.
```

12. Component Explanation

- **Game Service:** In-memory table state; rules engine; turn management.
- **WebSocket Hub:** Fans out state updates to all players at table.
- **RNG Service:** Cryptographically secure random (e.g. /dev/urandom + seed).
- **Postgres:** User balances, game history.
- **API Gateway:** Auth, rate limit.

13. Technology Stack

- **Language:** Go or Node.js (event-loop fits turn-based game).
- **DB:** PostgreSQL for persistence.
- **Realtime:** WebSocket (Socket.io or native).

- **Deployment:** Kubernetes.

14. Database Choice

Postgres for ACID on user balances. Game state in-memory (Redis backup for crash recovery).

15. Cache Strategy

In-memory game state. Redis optional as backup for crash recovery.

16. Load Balancer Strategy

Sticky session per table (table state is in-memory on specific instance).

17. Message Queue Usage

Optional: Kafka for game-completed events for analytics.

18. Scaling Strategy

Stateful scaling: route by table_id to specific Game Service instance. Add more instances as table count grows.

19. Fault Tolerance

If Game Service instance dies: tables on it are lost. Mitigate with Redis backup of in-memory state, restore on failover.

20. Bottlenecks

- Single-instance-per-table creates hot spots — rebalance tables across instances.
- WebSocket connection count — scale WS Hub horizontally.

21. Trade-offs

- **In-memory vs DB-backed state:** In-memory is fast; DB-backed is durable.
- **P2P vs server-authoritative:** Server-authoritative prevents cheating.

22. CAP Theorem Analysis

CP for balances. AP for game state (a crashed table is annoying, not catastrophic).

23. Security Considerations

Server-authoritative (clients cannot manipulate deck). TLS. RNG audited. Anti-collusion detection for multiplayer.

24. Monitoring & Logging

Track active tables, average round duration, WebSocket connection count, error rate.

25. Deployment Strategy

StatefulSets in Kubernetes; sticky routing via consistent hashing on table_id.

26. Interview Tips

- Recognize this is OOP + real-time, not big-data.
- Use Strategy pattern for actions (HitStrategy, StandStrategy, etc.).
- Discuss server-authoritative vs client-authoritative for fairness.
- Mention in-memory state with Redis backup for crash recovery.

27. Common Follow-up Questions

- How would you handle a player disconnecting mid-round?
- How would you support tournaments with leaderboards?
- How would you prevent collusion between players?
- How would you add side bets (insurance, perfect pairs)?
- How would you scale to 100K concurrent tables?

28. Common Mistakes

- Client-authoritative deck — cheaters can predict cards.
- Storing active game state only in DB — too slow for turn-based.
- No timeout for inactive players — tables stall.

29. Optimization Ideas

- Pre-shuffle decks in background to reduce round-start latency.
- Batch WebSocket messages per table (1 update per state change, not per card).
- Compress JSON state for low-bandwidth clients.

30. Final Summary

A blackjack game tests OOP design (rules engine, strategy pattern) and real-time multiplayer architecture (WebSocket fanout, in-memory state). The key decisions are: (1) server-authoritative to prevent cheating, (2) in-memory table state for speed with Redis backup for durability, (3) sticky routing for stateful scaling. The architecture is simple but the edge cases (disconnects, timeouts, fairness) are what separate a working implementation from a production one.

QUESTION 12 BEGINNER

Design a Chess Game (Multiplayer)

1. Problem Statement

Design a multiplayer chess platform. Support matchmaking, real-time moves, move validation, game state persistence, ELO rating, and a spectator mode.

2. Functional Requirements

- Matchmaking: pair players by ELO rating.
- Real-time move broadcast with sub-second latency.
- Move validation (legal moves, check, checkmate, castling, en passant).
- Game state persistence and replay (PGN format).
- Spectator mode (watch ongoing games).
- Rating update on game end.

3. Non-functional Requirements

- Latency: move broadcast < 500ms.
- Consistency: both players see identical state.
- Availability 99.5%.
- Move validation server-side (anti-cheat).

4. Capacity Estimation

100K concurrent games × 2 players + 10K spectators = 210K concurrent users. ~1M games/day. PGN history: ~2KB/game × 1M × 365 × 5 = 3.6TB.

5. Back-of-the-envelope Math

210K WebSocket connections need ~5 connection servers (40K each). Move validation is CPU-light; one Game Service instance handles 1K concurrent games.

6. API Design

```
POST /matchmaking/join body: {user_id, rating} -> {game_id, opponent, color}
POST /games/{id}/move body: {from, to, promotion?} -> {valid, new_state}
POST /games/{id}/resign
GET /games/{id}/spectate -> WebSocket stream

// WebSocket /games/{id}/stream
// server -> client: {type:"move", from, to, fen, clock}
// server -> client: {type:"end", result, rating_delta}
```

7. Database Schema

```
Table: users (user_id, username, elo, games_played)
Table: games (game_id, white_id, black_id, status, pgn_moves, started_at, ended_at,
result)
Table: moves (game_id, move_num, from_sq, to_sq, fen_after, timestamp) -- for replay
```

8. High-Level Design (HLD)

Client → API Gateway → Matchmaking Service (finds opponent) → Game Service (validates moves, holds state) → WebSocket Hub (broadcasts to players + spectators).

9. Low-Level Design (LLD)

GameEngine class: holds board state (FEN), validates moves (legal move generator), updates clock, detects check/checkmate/draw. MoveValidator pattern. Game state in-memory with periodic snapshot to Postgres (every 5 moves).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Matchmaking:
  1. Player joins queue with rating.
  2. Matchmaking Service finds opponent within ±100 ELO within 30s, else widens range.
  3. Create game, assign colors, notify both players via push + WebSocket.

Move:
  1. Player submits move via WebSocket.
  2. Game Service validates (legal move generator on current FEN).
  3. Update board state, append to PGN, update clock.
  4. Broadcast new state to both players + spectators.
  5. Every 5 moves: snapshot to Postgres.
  6. On game end: update ELO, persist full PGN.
```

12. Component Explanation

- **Matchmaking Service:** Pairs players by ELO; queue in Redis sorted set.
- **Game Service:** In-memory state, move validation, clock management.

- **WebSocket Hub:** Broadcasts moves to players + spectators.
- **Postgres:** PGN history, ELO ratings, user profiles.
- **Move Validator:** Legal move generator (chess.js library server-side).

13. Technology Stack

- **Language:** Node.js (event-loop fits turn-based) or Go.
- **DB:** PostgreSQL.
- **Realtime:** WebSocket.
- **Cache:** Redis (matchmaking queue, active game IDs).
- **Deployment:** Kubernetes.

14. Database Choice

Postgres for PGN history (relational, queryable) and ELO. In-memory for active game state.

15. Cache Strategy

Redis for matchmaking queue (sorted set by ELO + wait time). Active game IDs in Redis for routing to correct Game Service instance.

16. Load Balancer Strategy

Sticky routing by game_id to specific Game Service instance (stateful).

17. Message Queue Usage

Optional: Kafka for game-end events (ELO updates, analytics).

18. Scaling Strategy

Game Service: stateful, scale by game count. WebSocket Hub: stateless, scale by connection count.

19. Fault Tolerance

Game state snapshot every 5 moves to Postgres. On instance failure, restore from snapshot and replay recent moves from client ACKs.

20. Bottlenecks

- Spectator fanout for popular games — use a separate broadcast tier.
- Matchmaking latency spikes during peak — widen ELO range after 30s wait.

21. Trade-offs

- **Server vs client validation:** Server prevents cheating; client is faster but exploitable.
- **In-memory vs DB state:** In-memory is fast; DB is durable. Hybrid: snapshot every N moves.

22. CAP Theorem Analysis

CP for game state (both players must agree on board). AP for spectator view (slightly stale OK).

23. Security Considerations

Server-authoritative move validation. TLS. Anti-cheat: detect impossible move sequences, engine-use detection (move timing patterns).

24. Monitoring & Logging

Track active games, matchmaking latency, move validation latency, WebSocket connections.

25. Deployment Strategy

StatefulSets for Game Service (sticky routing). Stateless WebSocket Hub scales horizontally.

26. Interview Tips

- Recognize server-authoritative is essential for fairness.
- Discuss move validation library (chess.js) — shows practical awareness.
- Mention spectator mode scaling separately.
- Snapshot every N moves balances durability vs write load.

27. Common Follow-up Questions

- How would you handle a player disconnecting for 10s?
- How would you implement chess clocks (time control)?
- How would you detect cheating (engine use)?
- How would you support correspondence chess (moves over days)?
- How would you add live commentary on grandmaster games?

28. Common Mistakes

- Client-side validation — cheaters submit illegal moves.
- No snapshotting — lost state on crash.
- Synchronous DB write per move — kills latency.

29. Optimization Ideas

- Pre-compute legal moves for current position to speed up validation.
- Use Redis pub/sub for spectator fanout (decoupled from game service).
- Compress PGN with delta encoding for storage.

30. Final Summary

A chess game tests real-time multiplayer architecture with strong consistency requirements. The key decisions are: (1) server-authoritative move validation to prevent cheating, (2) in-memory game state with periodic DB snapshots for durability, (3) stateful scaling via sticky routing. Spectator mode adds a fanout dimension that benefits from a separate broadcast tier. The architecture rewards clean separation of matchmaking, game logic, and broadcast.

QUESTION 13 BEGINNER

Design Snake & Ladders (Multiplayer)

1. Problem Statement

Design a multiplayer Snake & Ladders game. 2-4 players take turns rolling a die and moving on a 100-cell board with snakes (descend) and ladders (ascend). First to reach cell 100 wins.

2. Functional Requirements

- Create/join a game (2-4 players).
- Take turns: roll die, move token, apply snake/ladder effect.
- Real-time board sync across players.
- Game ends when a player reaches 100; declare winner.

3. Non-functional Requirements

- Latency: state broadcast < 500ms.
- Consistency: all players see same board.
- Availability 99%.
- Fairness: cryptographically secure die roll.

4. Capacity Estimation

10K concurrent games × 3 players avg = 30K users. Compute and storage trivial.

5. Back-of-the-envelope Math

In-memory state per game. Single Game Service instance handles thousands of games.

6. API Design

```
POST /games/create body: {max_players} -> {game_id}
POST /games/{id}/join body: {user_id} -> {seat}
POST /games/{id}/roll -> {die_value, new_position, snake_or_ladder?}

// WebSocket /games/{id}/stream for state updates
```

7. Database Schema

```
// In-memory:
GameState { game_id, players: [{user_id, seat, position, color}],
            current_turn, snakes: {from: to}, ladders: {from: to}, status }

// DB (history):
Table: games (game_id, started_at, ended_at, winner_id, moves_json)
```

8. High-Level Design (HLD)

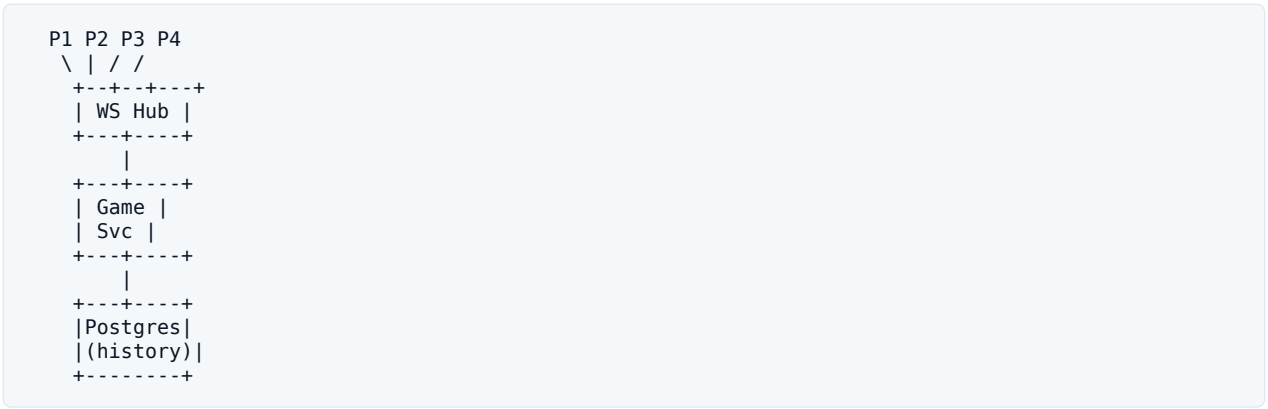
Mobile/Web → API Gateway → Game Service (in-memory state) → WebSocket Hub for broadcast.

9. Low-Level Design (LLD)

GameEngine class holds board config (snakes/ladders positions) and player states. State machine: WAITING → ACTIVE → ENDED. DieRoller service uses crypto-secure RNG.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Turn:
  1. Player whose turn it is calls /roll.
  2. Game Service rolls die (crypto-secure), updates position.
  3. If position lands on snake head: move to tail. If ladder bottom: move to top.
  4. If position == 100: game over, player wins.
  5. Broadcast new state to all players.
  6. Advance turn to next player (skip if won).
```

12. Component Explanation

- **Game Service:** In-memory state, turn management, RNG.
- **WebSocket Hub:** Broadcasts state to all players in a game.
- **Postgres:** Game history (optional).

13. Technology Stack

- **Language:** Node.js or Go.
- **DB:** PostgreSQL (optional, for history).
- **Realtime:** WebSocket.
- **Deployment:** Kubernetes.

14. Database Choice

In-memory only during game. Postgres for history if analytics needed.

15. Cache Strategy

In-memory game state. No external cache needed.

16. Load Balancer Strategy

Sticky session per game_id (stateful).

17. Message Queue Usage

None needed at this scale.

18. Scaling Strategy

Stateful scaling: route by game_id.

19. Fault Tolerance

Game lost on instance crash. Mitigate: snapshot to Redis every turn (cheap).

20. Bottlenecks

- Single-instance-per-game hot spots — rebalance.

21. Trade-offs

- **In-memory vs persisted state:** In-memory is fast; persisted survives crash.
- **Server vs client RNG:** Server prevents cheating.

22. CAP Theorem Analysis

CP for game state. AP for spectator (if added).

23. Security Considerations

Server-authoritative RNG. TLS. Anti-cheat: validate all moves server-side.

24. Monitoring & Logging

Track active games, turn latency, connection count.

25. Deployment Strategy

StatefulSets in Kubernetes.

26. Interview Tips

- Use Strategy pattern for snake/ladder effects.
- Discuss crypto-secure RNG for fairness.
- Mention turn timeout (auto-skip inactive players).

27. Common Follow-up Questions

- How would you add a "boost" power-up (extra turn on roll of 6)?
- How would you implement a tournament bracket?
- How would you add AI players?

28. Common Mistakes

- Client-side die roll — cheaters always roll 6.
- No turn timeout — games stall.

29. Optimization Ideas

- Pre-compute snake/ladder mapping for O(1) lookup.
- Compress state for low-bandwidth clients.

30. Final Summary

Snake & Ladders is a simple turn-based multiplayer game that tests real-time broadcast and turn management. The architecture is a straightforward in-memory state machine with WebSocket fanout. The interesting decisions are server-side RNG for fairness and turn timeouts for liveness. Compute and storage are trivial.

QUESTION 14 BEGINNER

Design Tic-Tac-Toe (Multiplayer)

1. Problem Statement

Design a real-time multiplayer Tic-Tac-Toe game with matchmaking, move validation, win/draw detection, and rematch support.

2. Functional Requirements

- Matchmaking: pair two players.
- Real-time move broadcast.
- Win/draw detection.
- Rematch option after game ends.
- Track wins/losses per player.

3. Non-functional Requirements

- Latency: move broadcast < 300ms.
- Consistency: both players see same board.
- Availability 99%.

4. Capacity Estimation

50K concurrent games = 100K users. Trivial compute/storage.

5. Back-of-the-envelope Math

Board is 9 cells; 3^9 = 19683 states. Move validation trivial.

6. API Design

```
POST /matchmaking/join -> {game_id, opponent, symbol}
POST /games/{id}/move body: {cell: 0-8} -> {valid, winner?}
POST /games/{id}/rematch

// WebSocket /games/{id}/stream
```

7. Database Schema

```
// In-memory:
GameState { game_id, board: [9], current_player, winner, status }

// DB:
Table: users (user_id, wins, losses, draws)
Table: games (game_id, p1_id, p2_id, winner_id, moves, timestamp)
```

8. High-Level Design (HLD)

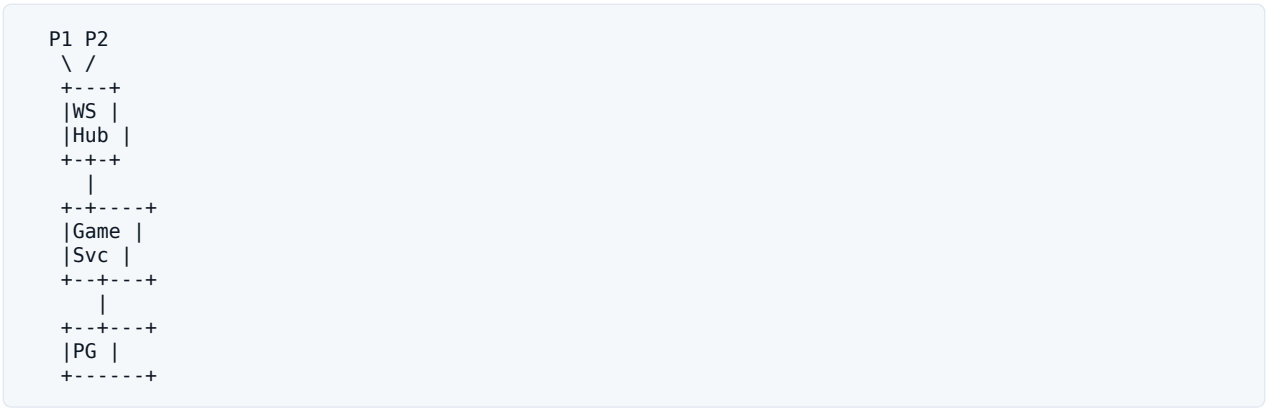
Client → API Gateway → Matchmaking → Game Service (in-memory) → WebSocket Hub.

9. Low-Level Design (LLD)

GameEngine: holds board array, validates move (cell empty + player's turn), checks win (8 winning lines), checks draw (board full).

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Move:
  1. Player submits move via WebSocket.
  2. Game Service validates: cell empty, player's turn.
  3. Update board, check win/draw.
  4. Broadcast new state + winner (if any).
  5. On game end: update stats, offer rematch.
```

12. Component Explanation

- **Matchmaking Service:** Pairs players from queue.
- **Game Service:** In-memory state, move validation, win detection.
- **WebSocket Hub:** Broadcasts to both players.
- **Postgres:** User stats, game history.

13. Technology Stack

- **Language:** Node.js.
- **DB:** PostgreSQL.
- **Realtime:** WebSocket.
- **Deployment:** Kubernetes.

14. Database Choice

Postgres for user stats. In-memory for active games.

15. Cache Strategy

Redis for matchmaking queue.

16. Load Balancer Strategy

Sticky session per game_id.

17. Message Queue Usage

None needed.

18. Scaling Strategy

Stateful scaling by game_id.

19. Fault Tolerance

Snapshot to Redis every move (cheap).

20. Bottlenecks

- Matchmaking latency spikes — widen search criteria after wait.

21. Trade-offs

- **Server vs client validation:** Server prevents cheating.
- **Stats sync vs async:** Async is faster; sync is consistent.

22. CAP Theorem Analysis

CP for game state.

23. Security Considerations

Server-authoritative. TLS.

24. Monitoring & Logging

Track active games, matchmaking latency.

25. Deployment Strategy

StatefulSets in Kubernetes.

26. Interview Tips

- Simple game — focus on clean state machine and win detection.
- Discuss rematch as a state transition, not a new game.

27. Common Follow-up Questions

- How would you add AI opponent with minimax?
- How would you implement a tournament?

28. Common Mistakes

- Client-side validation.

- No rematch support — clunky UX.

29. Optimization Ideas

- Pre-compute winning lines as a constant.
- Compress state to a single byte (9 cells × 2 bits = 18 bits).

30. Final Summary

Tic-Tac-Toe is a simple real-time multiplayer game testing WebSocket architecture and state machine design. The architecture is a textbook in-memory game service with WebSocket fanout. The interesting decisions are server-authoritative validation and rematch flow. Compute and storage are trivial.

QUESTION 15 BEGINNER

Design an Online Code Editor (Like CodeSandbox Lite)

1. Problem Statement

Design a browser-based code editor where users can write, run, and share small code snippets in multiple languages. Support syntax highlighting, execution in a sandbox, and shareable URLs.

2. Functional Requirements

- Edit code in browser with syntax highlighting.
- Run code in multiple languages (Python, JS, Go, etc.).
- Share via URL (read-only or collaborative).
- Persist snippets under a unique URL.
- Show stdout/stderr output and execution time.

3. Non-functional Requirements

- Latency: code execution < 3s for small programs.
- Security: sandbox untrusted code (no host access).
- Availability 99.5%.
- Snippet persistence: indefinite or expiry-based.

4. Capacity Estimation

100K daily users, 10K code runs/day. Storage: ~10KB per snippet × 1M = 10GB.

5. Back-of-the-envelope Math

Execution sandbox: container per run, killed after 10s. Pool of warm containers per language for fast startup.

6. API Design

```
POST /snippets body: {code, language} -> {snippet_id, url}
GET /snippets/{id} -> {code, language, owner}
POST /snippets/{id}/run body: {input?} -> {stdout, stderr, exit_code, duration_ms}
POST /snippets/{id}/fork -> {new_snippet_id}
```

7. Database Schema

```
Table: snippets
  snippet_id VARCHAR(8) PK
  language VARCHAR(20)
  code TEXT
  owner_id BIGINT NULL
  created_at TIMESTAMP
  expires_at TIMESTAMP NULL
  view_count INT DEFAULT 0
```

8. High-Level Design (HLD)

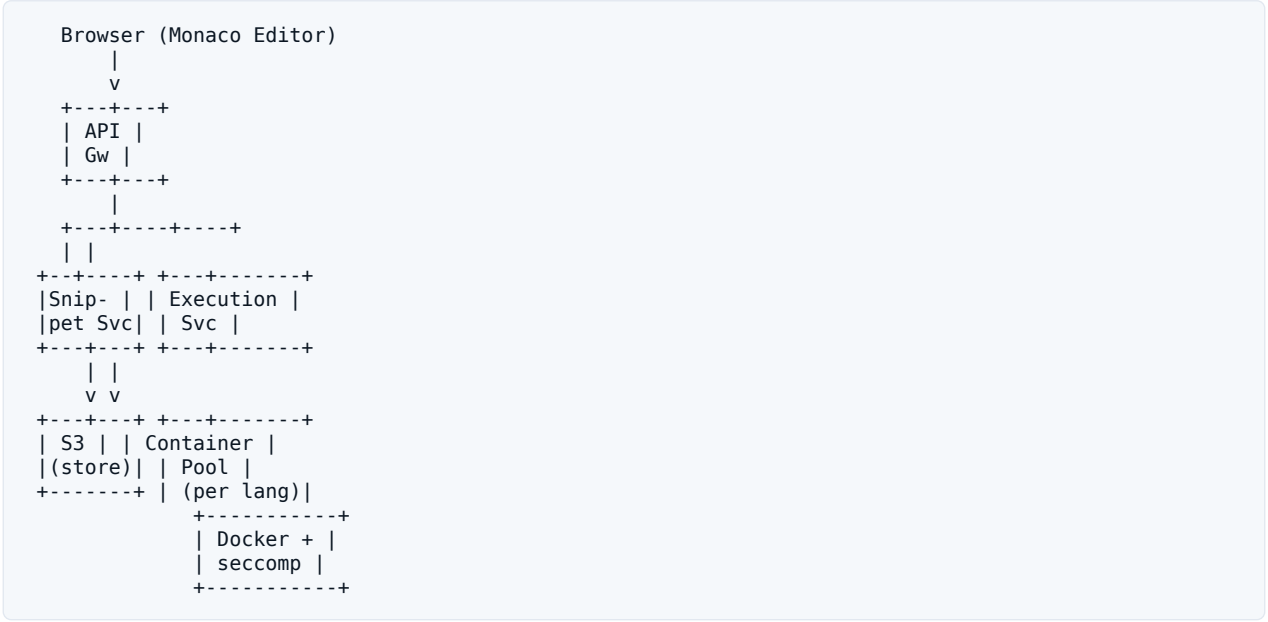
Browser → API Gateway → Snippet Service (CRUD) + Execution Service (runs code in container) → S3 (snippet storage). Container pool per language for fast startup.

9. Low-Level Design (LLD)

Execution Service: receives run request, picks warm container from pool, copies code into container, executes with timeout, captures stdout/stderr, returns to user. Container killed after use (security). Monaco editor in browser for syntax highlighting.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Run code:
  1. User clicks "Run". Browser POST /snippets/{id}/run.
  2. Execution Service picks warm container from language pool.
  3. Copy code into container, execute with 10s timeout + 256MB RAM limit.
  4. Capture stdout/stderr/exit_code.
  5. Kill container. Return result.
  6. Async: log execution metrics.
```

12. Component Explanation

- **Snippet Service:** CRUD for snippets; S3-backed.

- **Execution Service:** Manages container pool, runs code safely.
- **Container Pool:** Pre-warmed Docker containers per language.
- **S3:** Snippet storage.
- **Monaco Editor:** Browser-side editor with syntax highlighting.

13. Technology Stack

- **Language:** Node.js for API; Python/Go for execution service.
- **Sandbox:** Docker + seccomp + cgroups.
- **Storage:** S3 for snippets.
- **Editor:** Monaco (VS Code editor component).
- **Deployment:** Kubernetes.

14. Database Choice

S3 for snippet content (cheap, durable). Postgres optional for metadata if user accounts.

15. Cache Strategy

CDN for snippet fetches (public snippets). In-memory LRU for hot snippets.

16. Load Balancer Strategy

L7 LB. Execution Service needs special routing to node pools with Docker capability.

17. Message Queue Usage

Optional: Kafka for execution events (analytics, abuse detection).

18. Scaling Strategy

Execution Service scales with run QPS. Container pool size auto-scaled by queue depth.

19. Fault Tolerance

Container failures isolated. Execution Service multi-AZ.

20. Bottlenecks

- Container startup latency (~500ms cold) — mitigate with warm pool.
- CPU quota: limit per run to prevent abuse.

21. Trade-offs

- **Warm pool vs on-demand:** Warm is fast; on-demand is cheaper.
- **Docker vs gVisor/Firecracker:** Docker is simpler; Firecracker is more isolated.

22. CAP Theorem Analysis

AP for snippet fetches (CDN-cached). CP for execution (deterministic result required).

23. Security Considerations

Sandbox: seccomp filter, no network access, read-only filesystem, CPU/memory limits. Rate limit per IP. Anti-abuse: scan code for known malicious patterns.

24. Monitoring & Logging

Track execution latency, error rate, container pool size, abuse signals.

25. Deployment Strategy

Kubernetes with dedicated node pool for execution (privileged for container launch).

26. Interview Tips

- Emphasize sandboxing — running untrusted code is the core challenge.
- Discuss warm container pool vs cold start.
- Mention Monaco editor — shows awareness of browser tooling.

27. Common Follow-up Questions

- How would you support collaborative editing (multiple cursors)?
- How would you add language server protocol (LSP) for autocomplete?
- How would you handle long-running programs (web servers)?
- How would you detect and ban crypto-mining code?

28. Common Mistakes

- Running code on host without sandbox — catastrophic security.
- No timeout — infinite loops hang the service.
- Cold container per request — slow UX.

29. Optimization Ideas

- Use gVisor or Firecracker for stronger isolation than Docker.
- Pre-warm containers based on language popularity.
- Cache recent execution results for identical code+input.

30. Final Summary

An online code editor tests sandboxing and execution architecture. The core challenge is safely running untrusted code with low latency. The standard solution is a container pool (warm containers per language) with strict resource limits and seccomp filters. Monaco editor provides the browser UX. The architecture rewards operational thinking: pool sizing, abuse detection, and container lifecycle management.

QUESTION 16 BEGINNER

Design an Online Auction System (eBay Lite)

1. Problem Statement

Design an auction platform where sellers list items with a starting price and end time, buyers place bids, and the highest bidder wins when the auction ends. Support real-time bid updates and anti-sniping (extend auction on last-minute bids).

2. Functional Requirements

- Sellers create auctions (item, starting price, end time).
- Buyers place bids higher than current highest.
- Real-time bid updates to all viewers.
- Anti-sniping: extend auction by 5 min if bid in last 5 min.
- Notify winner and seller on auction end.

3. Non-functional Requirements

- Latency: bid broadcast < 500ms.
- Consistency: no two bidders can place the same amount; bids are atomic.
- Availability 99.95% during auction end windows.

4. Capacity Estimation

1M active auctions, 100K concurrent viewers, 10K bids/min peak. Storage modest.

5. Back-of-the-envelope Math

QPS modest. Real-time fanout is the main concern.

6. API Design

```
POST /auctions body: {item, start_price, end_time} -> {auction_id}
GET /auctions/{id} -> {item, current_bid, leader, time_left}
POST /auctions/{id}/bids body: {amount} -> {accepted, new_current_bid}

// WebSocket /auctions/{id}/stream for live bid updates
```

7. Database Schema

```
Table: auctions
  auction_id, seller_id, item_id, start_price, current_bid, current_leader_id,
  start_time, end_time, status (ACTIVE/ENDED/SOLD)

Table: bids
  bid_id, auction_id, bidder_id, amount, timestamp
  INDEX (auction_id, timestamp DESC)
```

8. High-Level Design (HLD)

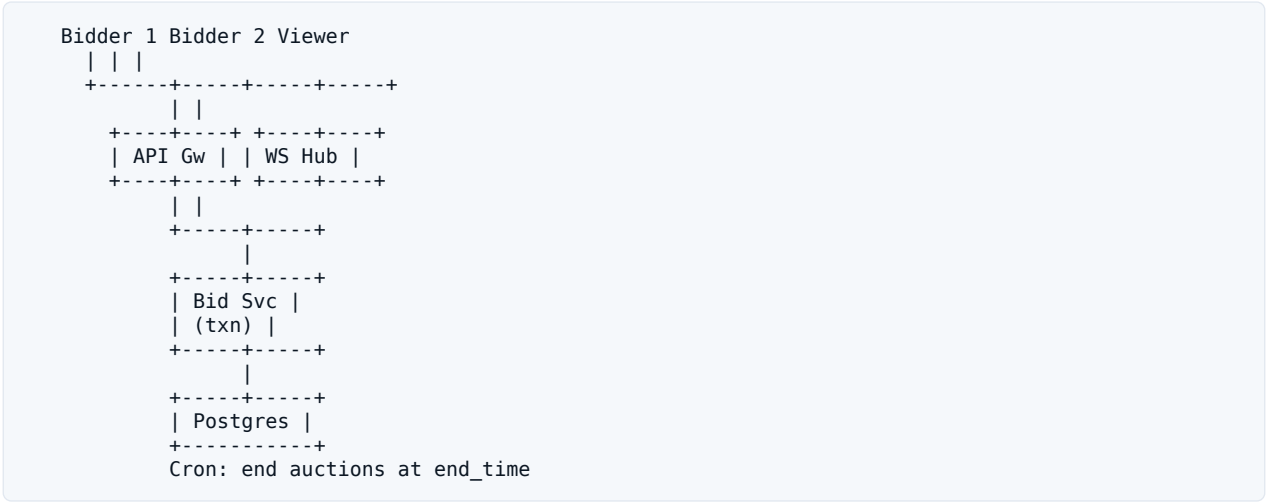
Web/Mobile → API Gateway → Auction Service (CRUD) + Bid Service (transactional) → WebSocket Hub for live updates + Postgres for state.

9. Low-Level Design (LLD)

Bid placement is transactional: BEGIN; SELECT auction FOR UPDATE; verify amount > current_bid; UPDATE auction; INSERT bid; COMMIT. Broadcast new state via WebSocket. Anti-sniping: if bid placed in last 5 min, extend end_time by 5 min.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Place bid:
  1. Bidder submits bid via REST.
  2. Bid Service: BEGIN TXN.
  3. SELECT auction FOR UPDATE; verify amount > current_bid.
  4. UPDATE auction SET current_bid, current_leader.
  5. INSERT bid row. If now within last 5 min: extend end_time by 5 min.
  6. COMMIT. Broadcast new state via WebSocket.
  7. Async: notify previous leader they were outbid.
```

12. Component Explanation

- **Auction Service:** CRUD for auctions.
- **Bid Service:** Transactional bid placement.
- **WebSocket Hub:** Live bid updates to viewers.

- **Postgres:** ACID for bids.
- **Notification Service:** Email/push on outbid, win, auction end.

13. Technology Stack

- **Language:** Java or Go.
- **DB:** PostgreSQL.
- **Realtime:** WebSocket.
- **Cache:** Redis (hot auctions, viewer counts).
- **Queue:** Kafka (notifications, analytics).

14. Database Choice

Postgres for ACID bids. SELECT FOR UPDATE prevents race conditions.

15. Cache Strategy

Redis for current bid display (faster reads). Source of truth is Postgres.

16. Load Balancer Strategy

L7 LB. Sticky session for WebSocket.

17. Message Queue Usage

Kafka for events (bid placed, auction ended, outbid).

18. Scaling Strategy

Shard by auction_id when needed. WebSocket Hub scales by connection count.

19. Fault Tolerance

Multi-AZ. WebSocket reconnect with state catch-up.

20. Bottlenecks

- Hot auctions (viral items) get many bids — mitigate with batch + queue.
- Auction-end spike — pre-scale Bid Service.

21. Trade-offs

- **Synchronous vs async bid:** Sync confirms immediately; async may show stale UI.
- **Anti-sniping vs hard end:** Anti-sniping is fairer; hard end is simpler.

22. CAP Theorem Analysis

CP. Strong consistency for bids (no two bidders at same amount).

23. Security Considerations

TLS. Auth required for bidding. Anti-shill: detect seller bidding on own auctions.

24. Monitoring & Logging

Track bid latency, conflict rate, WebSocket connections.

25. Deployment Strategy

Blue-green. Pre-scale for popular auctions.

26. Interview Tips

- Emphasize transactional bid placement.
- Discuss anti-sniping as a domain rule.
- Mention WebSocket for live updates.

27. Common Follow-up Questions

- How would you handle a viral auction with 1M concurrent viewers?
- How would you implement reserve prices (hidden minimum)?
- How would you detect shill bidding?

28. Common Mistakes

- No transaction on bid — race condition allows same amount twice.
- No anti-sniping — auctions end with last-second snipes.

29. Optimization Ideas

- Batch WebSocket messages per auction (1 update/sec, not per bid).
- Pre-compute "time left" display in browser.

30. Final Summary

An auction system tests transactional bid placement and real-time fanout. The core challenge is atomic bid updates (no two bidders at same amount) under concurrent load, solved with SELECT FOR UPDATE. Anti-sniping (extending end time on last-minute bids) is a domain rule that affects the data model. WebSocket broadcast keeps all viewers in sync. The architecture is sharded by auction_id when scale demands.

QUESTION 17 BEGINNER

Design a Shopping Cart System

1. Problem Statement

Design a shopping cart for an e-commerce site. Support add/remove/update quantity, persist cart across sessions, merge anonymous cart with user cart on login, and handle inventory reservation at checkout.

2. Functional Requirements

- Add item to cart, update quantity, remove item.
- Persist cart across sessions (anonymous and logged-in).
- Merge anonymous cart into user cart on login.
- Reserve inventory at checkout (hold for 10 min during payment).
- Apply promo codes and calculate tax/shipping.

3. Non-functional Requirements

- Availability 99.95%.
- Latency: cart operations < 200ms.
- Consistency: cart total must reflect current prices.
- Inventory reservation must be atomic.

4. Capacity Estimation

10M users, 1M concurrent carts, 100K checkouts/day. Cart size small (10 items avg).

5. Back-of-the-envelope Math

Storage trivial. QPS modest. Real challenge is inventory reservation at checkout.

6. API Design

```
POST /cart/items body: {product_id, quantity} -> {cart}
PATCH /cart/items/{id} body: {quantity}
DELETE /cart/items/{id}
GET /cart -> {items, subtotal, tax, shipping, total}
POST /cart/checkout -> {reservation_id, payment_token}
POST /cart/merge body: {anonymous_cart_id} (called on login)
```

7. Database Schema

```
Table: carts
  cart_id BIGINT PK
  user_id BIGINT NULL -- NULL for anonymous
  session_id VARCHAR(64) -- for anonymous carts
  created_at, updated_at

Table: cart_items
  cart_id, product_id, quantity, added_at
  PRIMARY KEY (cart_id, product_id)

Table: inventory_reservations
  reservation_id, cart_id, product_id, quantity, expires_at, status
```

8. High-Level Design (HLD)

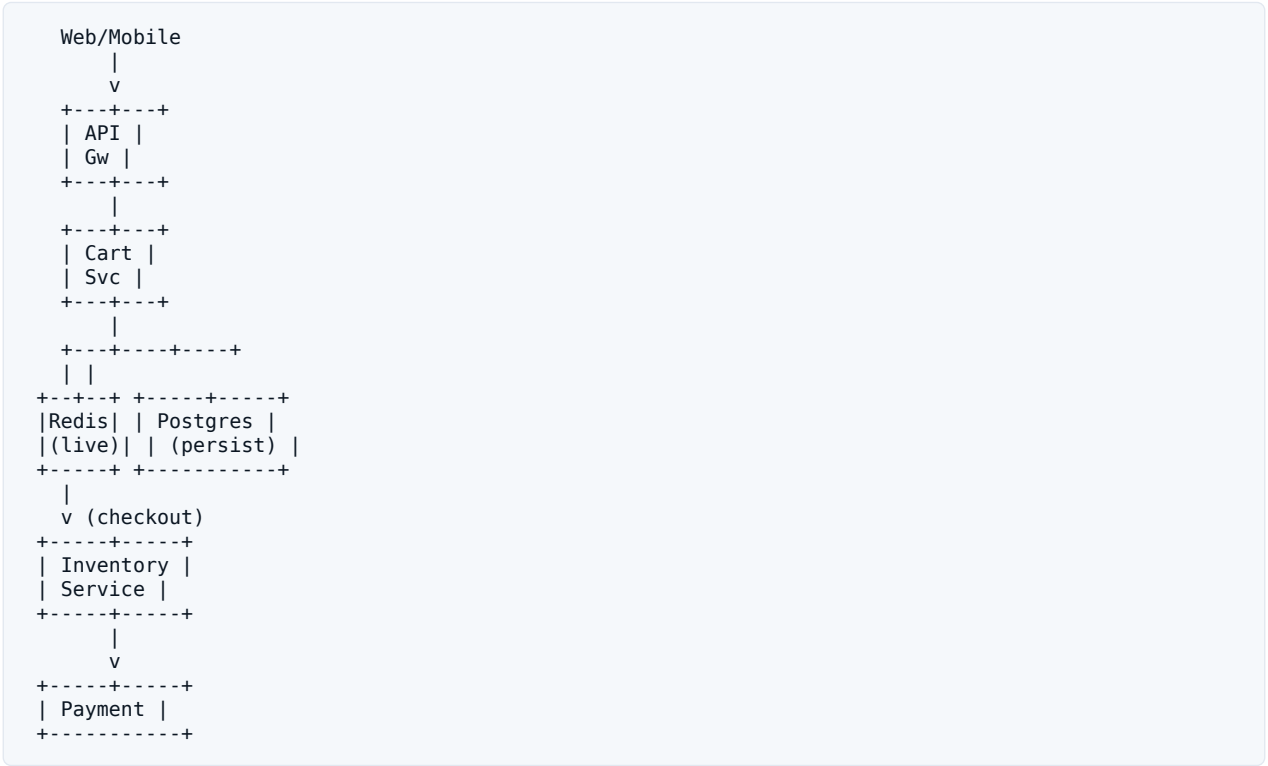
Web/Mobile → API Gateway → Cart Service → Redis (active cart state) + Postgres (persistence) + Inventory Service (for reservation).

9. Low-Level Design (LLD)

Cart stored in Redis for fast access (sub-ms). Periodic snapshot to Postgres. Checkout: call Inventory Service to reserve items (atomic, with 10-min TTL); on payment success, convert reservation to permanent deduction; on failure, release reservation.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Add to cart:
  1. POST /cart/items with product_id, quantity.
  2. Cart Service updates Redis cart (sub-ms).
  3. Async: snapshot to Postgres every 5 min or on logout.

Checkout:
  1. POST /cart/checkout.
  2. Cart Service calls Inventory Service: reserve items for 10 min.
  3. On success: initiate payment with reservation_id as idempotency key.
  4. On payment success: deduct inventory, create order, clear cart.
  5. On payment failure or timeout: release reservation.
```

12. Component Explanation

- **Cart Service:** Stateless; manages cart CRUD.
- **Redis:** Live cart state (fast access).
- **Postgres:** Persistent cart backup; order history.
- **Inventory Service:** Reserves and deducts stock.
- **Payment Service:** Processes payment with idempotency.

13. Technology Stack

- **Language:** Java or Go.
- **Cache:** Redis Cluster.
- **DB:** PostgreSQL.
- **Queue:** Kafka (cart events, order creation).
- **Deployment:** Kubernetes.

14. Database Choice

Redis for live cart (sub-ms access). Postgres for persistence and order history.

15. Cache Strategy

Redis IS the cart store, not a cache. TTL 30 days for anonymous carts.

16. Load Balancer Strategy

L7 LB. Stateless service, no sticky session needed.

17. Message Queue Usage

Kafka for cart events (add, remove, checkout) fed to analytics and recommendation.

18. Scaling Strategy

Cart Service scales horizontally. Redis Cluster for cart storage. Postgres sharded by user_id when needed.

19. Fault Tolerance

Redis with persistence (AOF + RDB) survives restart. Postgres multi-AZ with replica.

20. Bottlenecks

- Hot products cause inventory contention — use SELECT FOR UPDATE or versioned updates.
- Redis memory if carts grow large — cap cart size at 50 items.

21. Trade-offs

- **Redis-only vs Redis+Postgres:** Redis is fast; Postgres adds durability.
- **Reserve at checkout vs add-to-cart:** Checkout reserve avoids abandoned cart inventory loss.

22. CAP Theorem Analysis

AP for cart operations (eventual consistency OK). CP for inventory reservation (no oversell).

23. Security Considerations

TLS. Auth for user carts. Anonymous carts tied to session cookie. Validate product_id and quantity (prevent negative, prevent excessive).

24. Monitoring & Logging

Track cart operation latency, checkout conversion, inventory reservation conflicts.

25. Deployment Strategy

Blue-green. Redis Cluster rolling restarts.

26. Interview Tips

- Distinguish cart (Redis, fast) from order (Postgres, durable).
- Discuss cart merge on login — common real-world feature.
- Emphasize inventory reservation at checkout, not add-to-cart.

27. Common Follow-up Questions

- How would you handle "saved for later" items?
- How would you implement abandoned cart email reminders?
- How would you handle price changes between add-to-cart and checkout?
- How would you support wishlists and gift registries?

28. Common Mistakes

- Reserving inventory at add-to-cart — locks up stock for abandoned carts.
- Storing cart only in Postgres — too slow for frequent updates.
- No cart merge on login — lost items frustrate users.

29. Optimization Ideas

- Pre-compute cart totals in Redis (avoid re-fetching product prices).
- Use Redis hash per cart (efficient field updates).
- Batch cart snapshots to Postgres (reduce write load).

30. Final Summary

A shopping cart tests the separation of fast mutable state (Redis) from durable persistence (Postgres). The core challenge is inventory reservation at checkout — atomic, time-bounded holds that prevent overselling without locking up stock for abandoned carts. Cart merge on login is a common real-world feature. The architecture is simple but the transactional edge cases (price changes, reservation expiry, concurrent checkouts) require careful design.

QUESTION 18

BEGINNER

Design Stack Overflow Lite

1. Problem Statement

Design a Q&A; platform where users ask questions, answer questions, upvote/downvote, and earn reputation. Support search, tags, and accepted answers.

2. Functional Requirements

- Post questions (title, body, tags).
- Post answers to questions.
- Upvote/downvote questions and answers.
- Accept an answer (question owner only).
- Search by text, tag, user.
- Reputation system (+10 for upvote, +15 for accept, etc.).

3. Non-functional Requirements

- Availability 99.95%.
- Search latency < 500ms.
- Read-heavy (100:1 read:write).
- Strong consistency for votes and reputation.

4. Capacity Estimation

10M questions, 50M answers, 100M votes. Storage ~100GB. Read QPS: 10K, write QPS: 100.

5. Back-of-the-envelope Math

Storage modest. Read-heavy workload benefits from caching.

6. API Design

```
POST /questions body: {title, body, tags}
GET /questions/{id} -> {question, answers, votes}
POST /questions/{id}/answers body: {body}
POST /posts/{id}/vote body: {direction: UP|DOWN}
POST /questions/{id}/accept body: {answer_id}
GET /search?q=...&tag=...
```

7. Database Schema

```
Table: posts (polymorphic)
  post_id, type (QUESTION|ANSWER), parent_id (NULL for questions),
  author_id, title, body, score, accepted_answer_id, created_at

Table: votes
  vote_id, post_id, user_id, direction, created_at
  UNIQUE (post_id, user_id)

Table: tags
  tag_id, name, count

Table: post_tags
  post_id, tag_id
```

8. High-Level Design (HLD)

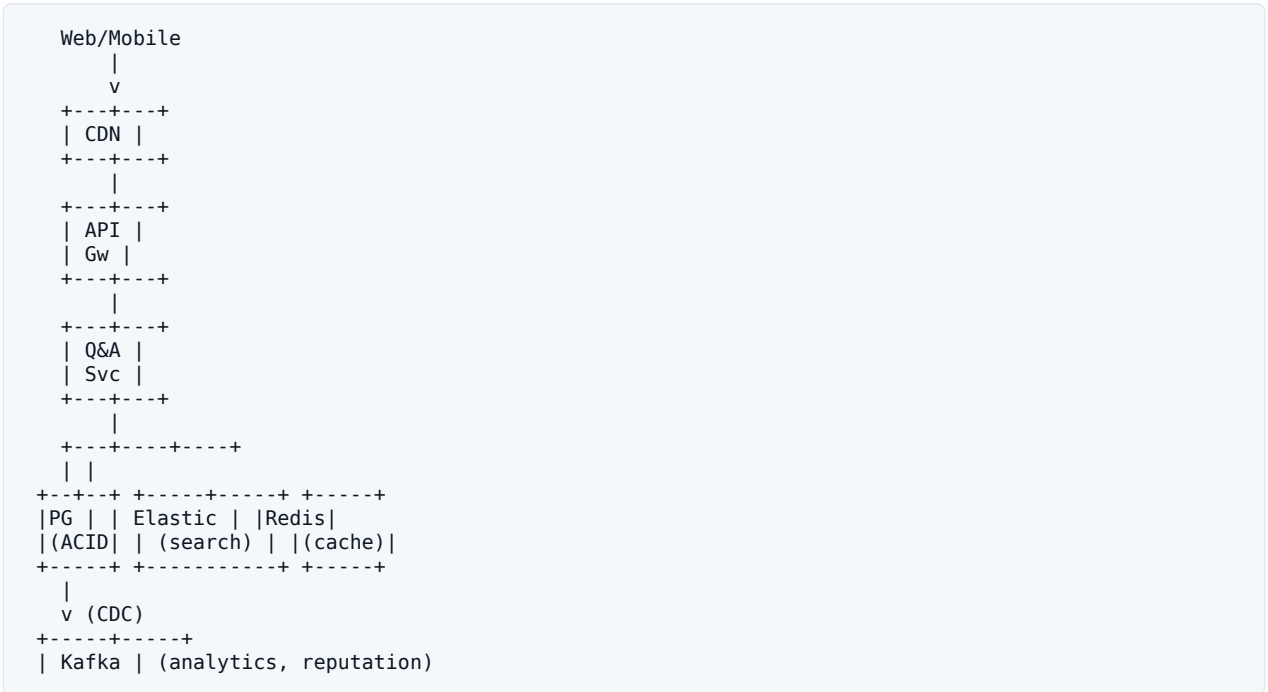
Web → CDN → API Gateway → Q&A; Service → Postgres + Elasticsearch + Redis (cache).

9. Low-Level Design (LLD)

Vote is transactional: BEGIN; verify user hasn't voted; insert vote; update post score; update author reputation; COMMIT. Search uses Elasticsearch indexed via CDC from Postgres.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Vote:
  1. User clicks upvote on post.
  2. Q&A Service: BEGIN TXN.
  3. Check no existing vote by this user on this post.
  4. INSERT vote; UPDATE post SET score = score ± 1;
  5. UPDATE author SET reputation = reputation ± 10.
  6. COMMIT. Invalidate cache for post.
  7. Async: publish vote event for analytics.
```

12. Component Explanation

- **Q&A; Service:** Core CRUD + vote logic.
- **Postgres:** ACID source of truth.
- **Elasticsearch:** Search index (questions, answers, tags).
- **Redis:** Cache for hot questions, reputation counters.
- **Kafka:** CDC stream from PG to ES, analytics events.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** PostgreSQL.
- **Search:** Elasticsearch.
- **Cache:** Redis.
- **Queue:** Kafka.

14. Database Choice

Postgres for ACID (votes, reputation). Elasticsearch for search.

15. Cache Strategy

Redis caches: hot questions, top users, tag counts. TTL 5 min.

16. Load Balancer Strategy

L7 LB. Stateless service.

17. Message Queue Usage

Kafka for CDC (PG → ES) and analytics events.

18. Scaling Strategy

Read replicas for Q&A; Service. ES cluster scales search. Redis Cluster for cache.

19. Fault Tolerance

Multi-AZ. PG with sync replica. ES 3-node cluster.

20. Bottlenecks

- Hot questions get many votes — batch updates.

- Search latency on common queries — pre-warm ES caches.

21. Trade-offs

- **Polymorphic posts vs separate tables:** Polymorphic is simpler; separate is normalized.
- **Sync vs async reputation:** Sync is consistent; async is faster.

22. CAP Theorem Analysis

CP for votes (no double-vote). AP for search (eventually consistent with PG).

23. Security Considerations

TLS. Auth required for posting/voting. Rate limit votes (prevent bot farming). Spam detection on questions/answers.

24. Monitoring & Logging

Track search latency, vote conflict rate, ES lag, cache hit ratio.

25. Deployment Strategy

Blue-green. ES rolling restarts. PG operator for failover.

26. Interview Tips

- Use polymorphic posts table — simplifies voting and reputation.
- Discuss reputation system as a transactional concern.
- Mention CDC for PG → ES sync.

27. Common Follow-up Questions

- How would you implement real-time notifications?
- How would you detect vote manipulation (sock puppets)?
- How would you add a "trending questions" feature?
- How would you support multiple languages (i18n)?

28. Common Mistakes

- No transaction on vote — allows double-voting.
- Searching in Postgres — slow at scale.
- No cache — read-heavy workload suffers.

29. Optimization Ideas

- Pre-compute "hot questions" hourly via Spark.
- Use materialized views for tag counts.
- Compress post bodies for storage.

30. Final Summary

Stack Overflow Lite tests CRUD with voting and reputation. The core challenge is transactional vote updates (no double-voting, atomic score + reputation update). Search is delegated to Elasticsearch synced via CDC. The read-heavy workload rewards aggressive caching. The polymorphic posts table (questions and answers in one table) simplifies the data model.

QUESTION 19 BEGINNER

Design a Notes App (Evernote Lite)

1. Problem Statement

Design a note-taking app. Users create, edit, and organize notes in notebooks. Support rich text, attachments, full-text search, and offline sync across devices.

2. Functional Requirements

- Create/edit notes (rich text + attachments).
- Organize notes into notebooks and tags.
- Full-text search across notes.
- Sync across devices with conflict resolution.
- Offline mode: edit locally, sync when online.

3. Non-functional Requirements

- Availability 99.9%.
- Latency: save < 500ms, search < 1s.
- Eventual consistency across devices (< 5s).
- Conflict resolution for concurrent edits.

4. Capacity Estimation

1M users, 10 notes/user avg = 10M notes. Storage: 50KB/note × 10M = 500GB + attachments.

5. Back-of-the-envelope Math

Storage modest. Sync architecture is the main challenge.

6. API Design

```
POST /notes body: {title, body, notebook_id} -> {note_id, version}
PUT /notes/{id} body: {title, body, version} -> {note_id, new_version}
GET /notes/{id} -> {note, version}
GET /notes?since=... -> [changed notes] (for sync)
POST /notes/{id}/attachments -> presigned S3 URL
GET /search?q=...
```

7. Database Schema

```
Table: notes
  note_id, user_id, notebook_id, title, body, version, updated_at, deleted_at

Table: notebooks
  notebook_id, user_id, name, created_at

Table: attachments
  attachment_id, note_id, s3_key, filename, size, mime_type
```

8. High-Level Design (HLD)

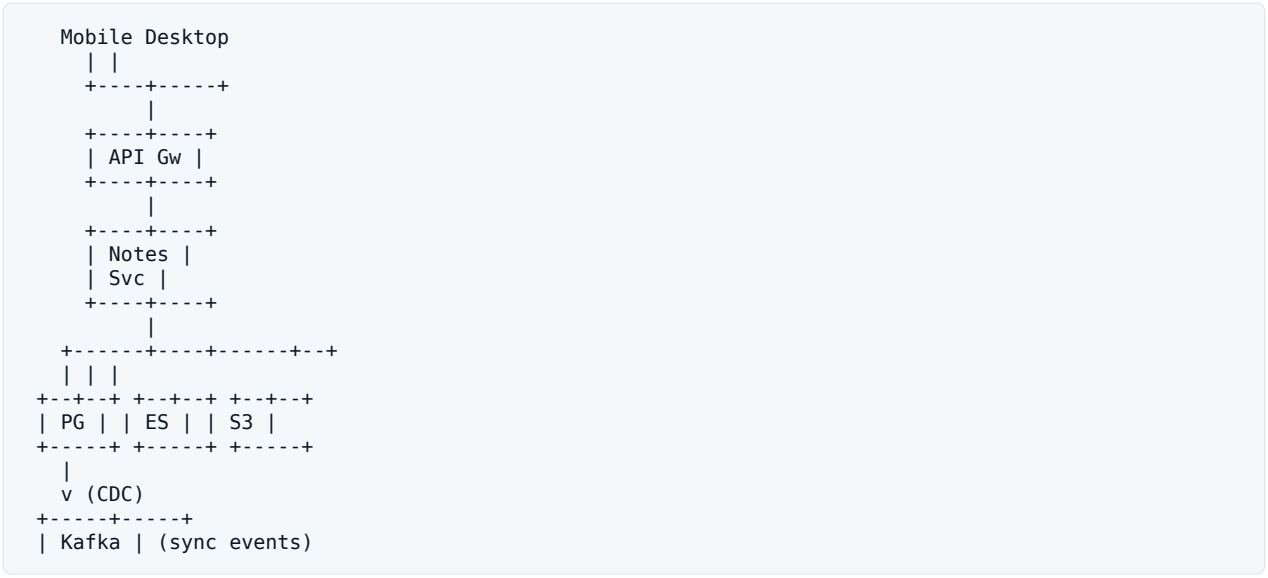
Mobile/Desktop → API Gateway → Notes Service → Postgres + S3 (attachments) + Elasticsearch (search). Sync via versioned updates with last-write-wins or merge.

9. Low-Level Design (LLD)

Each note has a version number. On update, client sends current version; server rejects if version is stale (409 Conflict). Client pulls changes since last sync via /notes?since=TIMESTAMP.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Sync:
  1. App starts, calls /notes?since=last_sync.
  2. Server returns changed notes since timestamp.
  3. App merges changes locally.
  4. App pushes local changes: PUT /notes/{id} with version.
  5. Server: if version matches, accept and bump version. If stale, return 409.
  6. On 409: client fetches latest, merges (last-write-wins per field or full CRDT).
```

12. Component Explanation

- **Notes Service:** CRUD + sync logic.
- **Postgres:** ACID storage.
- **S3:** Attachments.

- **Elasticsearch:** Search.
- **Sync Service:** Optional: dedicated to mobile push.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** PostgreSQL.
- **Search:** Elasticsearch.
- **Storage:** S3.
- **Cache:** Redis (note cache).

14. Database Choice

Postgres for ACID + versioning. S3 for attachments. ES for search.

15. Cache Strategy

Redis for hot notes (recently accessed).

16. Load Balancer Strategy

L7 LB. Stateless service.

17. Message Queue Usage

Kafka for CDC (PG → ES) and sync notifications.

18. Scaling Strategy

Shard by user_id when needed. ES scales search.

19. Fault Tolerance

Multi-AZ. PG with replica. S3 cross-region replication.

20. Bottlenecks

- Concurrent edits on same note — version conflict resolution.
- Search indexing lag — acceptable for notes (not real-time).

21. Trade-offs

- **Last-write-wins vs CRDT:** LWW is simple; CRDT enables real-time collab.
- **Full content vs delta sync:** Full is simpler; delta is bandwidth-efficient.

22. CAP Theorem Analysis

AP. Eventual consistency across devices is acceptable. Strong consistency per device.

23. Security Considerations

TLS. Auth required. Encrypt attachments at rest (S3 SSE). Per-user data isolation.

24. Monitoring & Logging

Track save latency, sync conflict rate, search latency.

25. Deployment Strategy

Blue-green. ES rolling restarts.

26. Interview Tips

- Emphasize versioning for sync — core architectural decision.
- Discuss conflict resolution strategies (LWW, CRDT, manual merge).
- Mention offline mode as a first-class concern.

27. Common Follow-up Questions

- How would you add real-time collaborative editing (multiple cursors)?
- How would you implement handwriting recognition for scanned notes?
- How would you handle very large notes (> 1MB)?

28. Common Mistakes

- No versioning — sync overwrites silently.
- Storing attachments in DB — kills performance.
- Synchronous search indexing — blocks saves.

29. Optimization Ideas

- Use delta sync for large notes.
- Compress note bodies (gzip).
- Pre-index common search terms.

30. Final Summary

A notes app tests sync architecture and versioning. The core challenge is handling concurrent edits across devices — solved with version numbers and explicit conflict resolution (LWW or CRDT). Attachments go to S3, content to Postgres, search to Elasticsearch via CDC. Offline mode is a first-class concern that shapes the client architecture. The system is AP — eventual consistency across devices is acceptable.

QUESTION 20 BEGINNER

Design a Contact Manager (Phonebook App)

1. Problem Statement

Design a contact management app. Users store contacts (name, phone, email, notes), search contacts, sync across devices, and import/export (vCard, CSV).

2. Functional Requirements

- Add/edit/delete contacts.
- Search by name, phone, email.
- Sync across devices.
- Import from vCard/CSV; export to vCard/CSV.
- Merge duplicate contacts.

3. Non-functional Requirements

- Availability 99.9%.
- Latency: search < 200ms.
- Eventual consistency across devices.

4. Capacity Estimation

1M users, 1000 contacts/user avg = 1B contacts. Storage: 2KB/contact × 1B = 2TB.

5. Back-of-the-envelope Math

Storage modest. Search is the main concern at scale.

6. API Design

```
POST /contacts body: {name, phone, email, notes}
GET /contacts/{id}
PUT /contacts/{id} body: {..., version}
DELETE /contacts/{id}
GET /contacts?q=... -> [matching contacts]
POST /contacts/import body: file -> {imported_count}
GET /contacts/export -> vCard/CSV file
```

7. Database Schema

```
Table: contacts
  contact_id, user_id, name, phone, email, notes, version, updated_at
  INDEX (user_id, name) -- for search
  INDEX (user_id, phone)
```

8. High-Level Design (HLD)

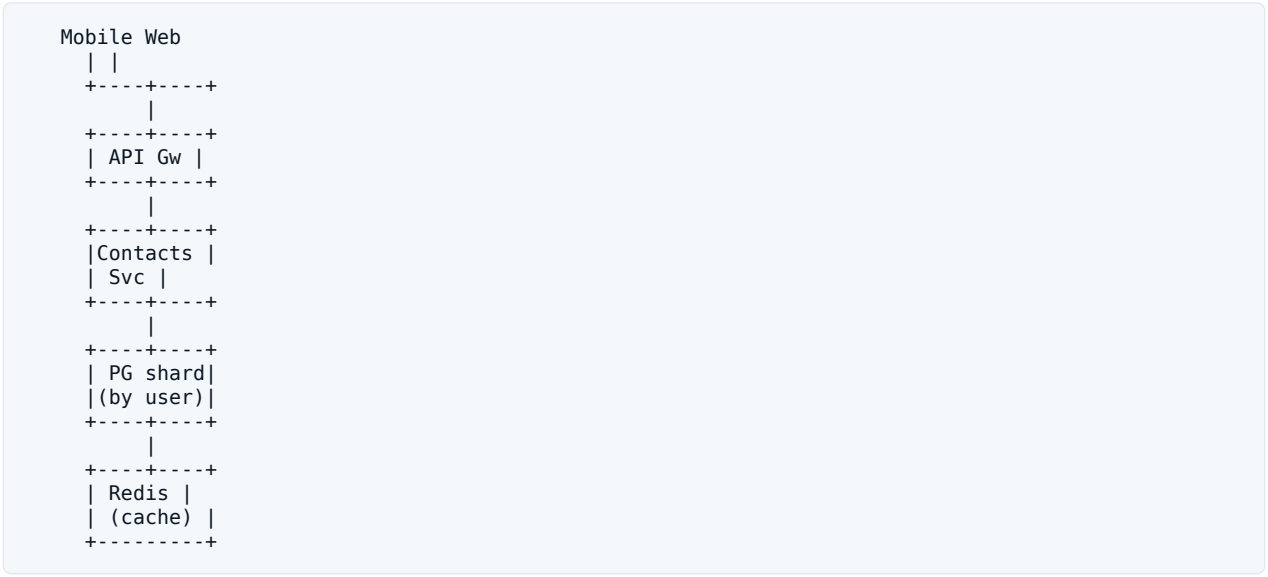
Mobile/Web → API Gateway → Contacts Service → Postgres (sharded by user_id) + Redis (cache).

9. Low-Level Design (LLD)

Contacts sharded by user_id (queries are user-scoped). Search uses PG full-text or a dedicated search index per user. Sync via versioning (same as notes app).

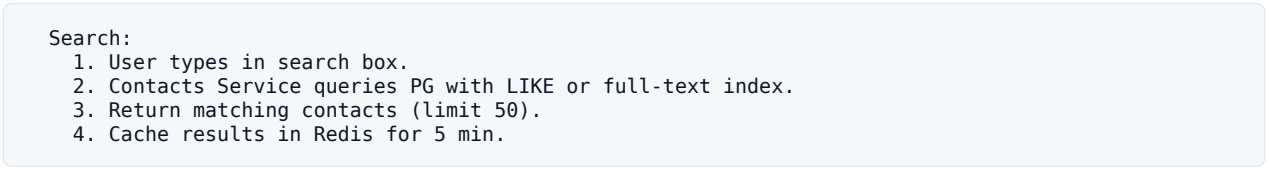
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Contacts Service:** CRUD + search.
- **Postgres (sharded):** Per-user data.
- **Redis:** Cache for search results.
- **Import/Export Service:** Async job for large imports.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** PostgreSQL sharded by user_id.
- **Cache:** Redis.
- **Deployment:** Kubernetes.

14. Database Choice

Postgres sharded by user_id. Each user's contacts fit on one shard.

15. Cache Strategy

Redis for recent search results and frequently accessed contacts.

16. Load Balancer Strategy

L7 LB. Stateless service.

17. Message Queue Usage

Optional: Kafka for sync events.

18. Scaling Strategy

Shard by user_id. Each shard handles ~100K users.

19. Fault Tolerance

Multi-AZ. PG with replica per shard.

20. Bottlenecks

- Large imports block service — process async.

21. Trade-offs

- **PG full-text vs ES:** PG is simpler; ES scales better.
- **Sync vs async import:** Async is faster UX; sync confirms immediately.

22. CAP Theorem Analysis

AP. Eventual consistency across devices acceptable.

23. Security Considerations

TLS. Per-user data isolation. PII encryption at rest.

24. Monitoring & Logging

Track search latency, import job duration, sync conflicts.

25. Deployment Strategy

Blue-green. PG operator for failover.

26. Interview Tips

- Shard by user_id — queries are user-scoped.
- Discuss import as async job — large imports block.
- Mention duplicate detection (fuzzy matching).

27. Common Follow-up Questions

- How would you implement duplicate contact detection?
- How would you sync with Google/Apple contacts?
- How would you handle contact sharing between users?

28. Common Mistakes

- No sharding — single PG won't scale.
- Synchronous import — large files timeout.
- No versioning — sync overwrites.

29. Optimization Ideas

- Use PG trigram indexes for fuzzy search.
- Batch sync updates (debounce).
- Pre-compute "favorites" for fast access.

30. Final Summary

A contact manager tests sharding strategy and search. The key insight is sharding by user_id (queries are user-scoped, so each shard is independent). Search uses Postgres full-text or trigram indexes for fuzzy matching. Import/export is handled as async jobs for large files. Sync via versioning. The architecture is simple but the scale (1B contacts) demands sharding from day one.

PART IV

Intermediate Questions (Q21 – Q50)

Real-world systems for SDE-2 and mid-level engineers.

QUESTION 21 INTERMEDIATE

Design Instagram

1. Problem Statement

Design a photo-sharing app where users post photos/videos with captions, follow others, and see a feed of recent posts from followed accounts. Support likes, comments, stories, and direct messages.

2. Functional Requirements

- Upload photo/video with caption; apply filters.
- Follow/unfollow users.
- View home feed (recent posts from followed users).
- Like and comment on posts.
- Stories (24-hour ephemeral posts).
- Direct messages.

3. Non-functional Requirements

- Availability 99.95%.
- Feed generation < 2s for 95% of users.
- Upload latency < 5s including media processing.
- Eventual consistency acceptable for feed (new post visible within 30s).

4. Capacity Estimation

1B users, 500M DAU. 100M new posts/day. Avg post: image 200KB, video 5MB. Storage: 100M × 1MB × 365 × 5 = ~180TB/5yr. Feed QPS: 50M reads/day = ~600 RPS avg, 6K peak.

5. Back-of-the-envelope Math

Feed generation: pulling posts from N followees per request is too slow. Use push model: on post, fanout to followers' feed lists in Redis. Celeb exception: pull for users with >100K followers.

6. API Design

```
POST /posts body: {media_url, caption, tags}
GET /feed?cursor=... -> [posts]
POST /posts/{id}/like / DELETE /posts/{id}/like
POST /posts/{id}/comments body: {text}
POST /users/{id}/follow / DELETE /users/{id}/follow
GET /stories -> [active stories from followees]
```

7. Database Schema

```
Table: users (user_id, username, bio, follower_count, ...)
Table: posts (post_id, user_id, media_url, caption, like_count, created_at)
Table: follows (follower_id, followee_id, created_at)
Table: likes (post_id, user_id, created_at) PRIMARY KEY (post_id, user_id)
Table: comments (comment_id, post_id, user_id, text, created_at)
Table: stories (story_id, user_id, media_url, expires_at)
```

8. High-Level Design (HLD)

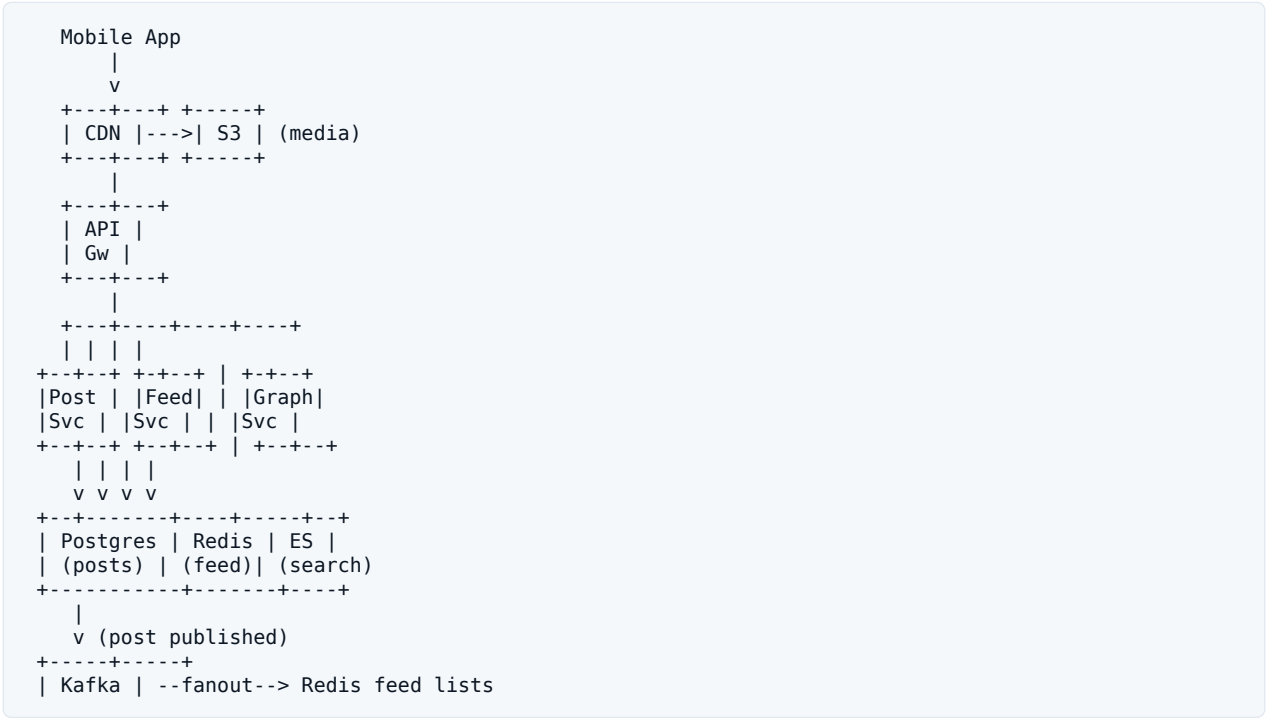
Mobile → CDN → API Gateway → Post Service (upload) + Feed Service (read) + Graph Service (follows). Media via S3 + CloudFront. Push-based feed via Kafka + Redis. Stories via TTL-keyed Redis.

9. Low-Level Design (LLD)

Feed generation: pre-computed. On post publish, fanout worker reads followers list, writes post_id to each follower's feed list in Redis (capped at 1000 items). On feed read, fetch from Redis list. For celebrities, use pull: merge celebrity's recent posts into feed at read time.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Post publish:
  1. User uploads media to S3 via presigned URL.
  2. POST /posts with caption + media_url.
  3. Post Service writes to Postgres, publishes event to Kafka.
  4. Fanout Worker consumes event, reads followers from Graph Service.
  5. Push post_id to each follower's Redis feed list.
  6. For celebrity (>100K followers): skip push; mark for pull.

Feed read:
  1. GET /feed with cursor.
  2. Feed Service reads user's feed list from Redis (last 50).
  3. Merge in recent posts from celebrities user follows (pull).
  4. Rank by recency + engagement signals.
  5. Return hydrated post objects.
```

12. Component Explanation

- **Post Service:** Uploads, captions, metadata.
- **Feed Service:** Reads pre-computed feed; merges celebrity posts.
- **Graph Service:** Follows/unfollows; followers list.
- **Fanout Worker:** Consumes Kafka events; pushes to Redis feed lists.
- **Media Service:** S3 presigned URLs; image/video processing.
- **Redis:** Feed lists (per user), like counters, story TTL keys.
- **Postgres:** Posts, follows, comments — sharded by user_id.

13. Technology Stack

- **Language:** Python/Go for services.
- **DB:** PostgreSQL sharded by user_id; Cassandra for messages.
- **Cache:** Redis Cluster.
- **Queue:** Kafka.
- **Search:** Elasticsearch for hashtags/search.
- **Media:** S3 + CloudFront CDN.
- **Deployment:** Kubernetes multi-region.

14. Database Choice

Postgres sharded by user_id for posts/follows (ACID for likes). Cassandra for DMs (write-heavy, time-series). Redis for feed lists and counters.

15. Cache Strategy

Multi-tier: CDN for media, Redis for feed lists and post metadata, in-memory LRU in Feed Service for hot posts.

16. Load Balancer Strategy

L7 LB; stateless services; geographic routing for lower latency.

17. Message Queue Usage

Kafka topics: posts (fanout), likes (counter updates), comments (notifications).

18. Scaling Strategy

Shard by user_id. Fanout workers scale with post-publish rate. Redis Cluster for feed lists. Pre-scale for known events (celebrity posts).

19. Fault Tolerance

Multi-AZ. If Redis feed list missing, fall back to live pull from Postgres. Kafka persistence survives fanout worker crashes.

20. Bottlenecks

- Celebrity fanout — mitigate with pull model for >100K follower accounts.
- Feed list memory in Redis — cap at 1000 items per user.
- Hot post like-count contention — batch updates.

21. Trade-offs

- **Push vs pull feed:** Push is fast read but expensive write; pull is opposite. Hybrid wins.
- **Pre-compute vs live rank:** Pre-compute is faster; live rank is fresher.
- **Strong vs eventual consistency for likes:** Eventual is fine; counts converge.

22. CAP Theorem Analysis

AP for feed (eventual consistency acceptable). CP for likes (no double-like).

23. Security Considerations

TLS. Auth via OAuth/JWT. Image moderation (NSFW detection). Rate limit uploads. Anti-bot for likes/comments.

24. Monitoring & Logging

Track feed latency, fanout lag, upload success rate, like counter drift.

25. Deployment Strategy

Blue-green for services. Kafka rolling restarts. Multi-region active-active for read paths.

26. Interview Tips

- Discuss push vs pull feed — the core architectural decision.
- Mention celebrity exception — shows real-world thinking.
- Calculate fanout cost: 1 post × 1000 followers avg = 1000 Redis writes per post.
- Discuss media processing pipeline (thumbnails, video transcoding).

27. Common Follow-up Questions

- How would you implement "Explore" feed (recommended posts from non-followed accounts)?
- How would you handle a celebrity with 100M followers posting (thundering herd)?
- How would you implement live video (Instagram Live)?
- How would you detect and remove bot likes?

- How would you add Reels (short-form video)?

28. Common Mistakes

- Pure pull feed — too slow for users with many followees.
- Pure push feed — celebrity posts crush fanout.
- Storing media in DB — kills performance.
- Synchronous fanout on post — upload latency suffers.

29. Optimization Ideas

- Pre-warm feed for active users on app launch prediction.
- Batch fanout writes to Redis (pipeline).
- Use CDN for post media (saves origin bandwidth).
- Compress images aggressively (WebP instead of JPEG).

30. Final Summary

Instagram is the canonical feed-design question. The core decision is push vs pull: push (pre-compute feed on post publish) is fast for reads but expensive for celebrities; pull (compute feed on read) is opposite. The hybrid model — push for normal users, pull for celebrities — is the production answer. Media goes to S3+CDN, metadata to sharded Postgres, feed lists to Redis, fanout via Kafka. The architecture rewards operational maturity: pre-scaling for viral events, graceful degradation when Redis fails, multi-region deployment.

QUESTION 22 INTERMEDIATE

Design WhatsApp

1. Problem Statement

Design a messaging app supporting 1:1 and group chats, media sharing, voice notes, last-seen, read receipts, and end-to-end encryption. Target 2B users globally.

2. Functional Requirements

- 1:1 and group chat (up to 1024 members).
- Send text, images, videos, documents, voice notes.
- Real-time delivery and read receipts.
- Last-seen and online status.
- End-to-end encryption (Signal Protocol).
- Voice and video calls.

3. Non-functional Requirements

- Latency: message delivery p99 < 1s.
- Availability 99.99%.
- Messages must not be lost.
- Ordering: per-sender sequence numbers.
- Battery-efficient on mobile.

4. Capacity Estimation

2B users, 1B DAU, 100B messages/day. Avg msg 200B. Storage: 100B × 200B × 365 × 5 = ~36TB text + ~5PB media. Concurrent connections: 100M.

5. Back-of-the-envelope Math

100M concurrent connections need ~50 connection servers (2M conns each). Message routing: ~1.2M msgs/s. Media: S3 + CDN.

6. API Design

```
// WebSocket (after upgrade):
// Client->Server: {"type":"send","to":"user_id","msg_id":"...","content":"..."}
// Server->Client: {"type":"message","from":"...","msg_id":"...","content":"..."}
// Server->Client: {"type":"ack","msg_id":"...","status":"delivered|read"}

POST /media/upload -> presigned S3 URL
GET /messages?since=... -> catch up after reconnect
POST /calls/initiate -> WebRTC SDP offer/answer
```

7. Database Schema

```
// Sharded by conversation_id
Table: messages
  msg_id BIGINT, conversation_id, sender_id, type, content_ref,
  timestamp, sender_seq, status PRIMARY KEY (conversation_id, msg_id)

Table: conversations
  conversation_id, type (1:1|GROUP), members_json, last_msg_at

Table: user_presence
  user_id, last_seen, online (in Redis, not PG)
```

8. High-Level Design (HLD)

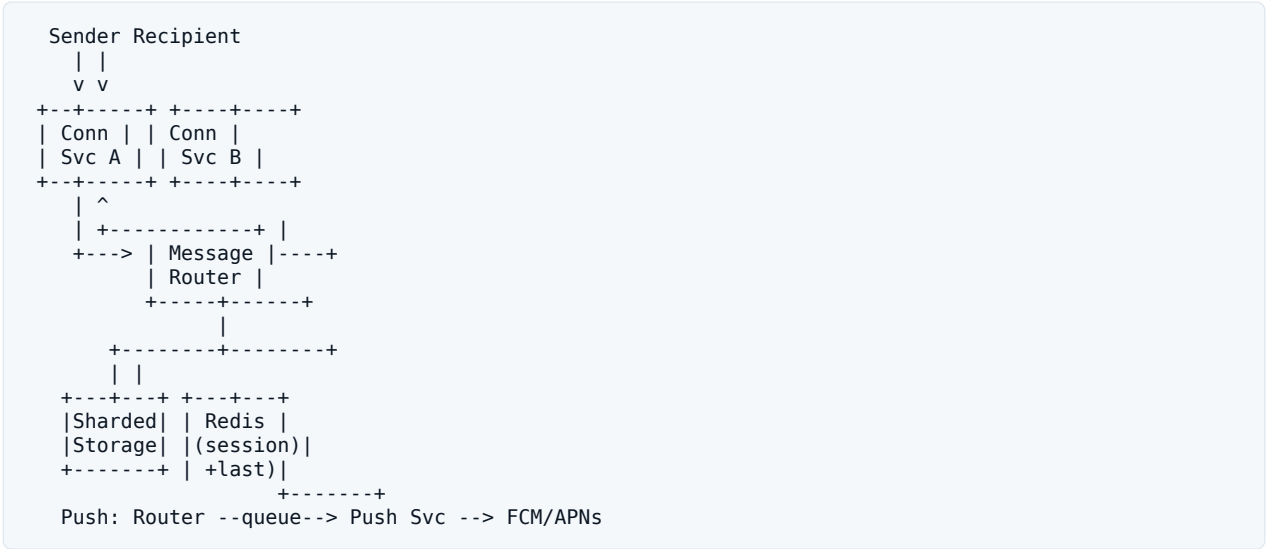
Mobile → Connection Servers (WebSocket) → Message Router → Storage (sharded) + Push Service (FCM/APNs) for offline. Voice/video via WebRTC + TURN/STUN.

9. Low-Level Design (LLD)

On send: Message Router persists to sharded storage, looks up recipients via Session Registry (Redis), pushes to online recipients' Connection Servers, enqueues push for offline. End-to-end encryption: clients exchange keys via Signal Protocol; server cannot decrypt message content.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Send message:
  1. Sender encrypts message with recipient's public key (Signal Protocol).
  2. Sends via WebSocket to Conn Svc.
  3. Conn Svc forwards to Message Router.
  4. Router: persist to sharded storage (key=conversation_id).
  5. Router: lookup recipient Conn Svc from Redis session registry.
  6. If online: push to recipient Conn Svc → recipient.
  7. If offline: enqueue push notification (FCM/APNs) with encrypted payload.
  8. Router: send DELIVERED ack to sender.
  9. Recipient app: decrypt, display, send READ receipt when user opens.
```

12. Component Explanation

- **Connection Service:** Holds WebSocket; routes inbound msgs.
- **Message Router:** Stateless; persists + fans out.
- **Session Registry:** Redis: user_id → conn_svc_id.
- **Sharded Storage:** Cassandra: messages by conversation_id.
- **Push Service:** FCM/APNs for offline users.
- **Media Service:** S3 + CDN; presigned URLs.
- **Call Service:** WebRTC signaling; TURN/STUN servers.

13. Technology Stack

- **Language:** Erlang/Elixir (WhatsApp original) or Go.
- **DB:** Cassandra (write-heavy, partitioned by conversation_id).
- **Cache:** Redis (sessions, last-seen).
- **Realtime:** WebSocket / MQTT.
- **Push:** FCM + APNs.
- **Calls:** WebRTC + coturn.
- **Deployment:** Kubernetes multi-region.

14. Database Choice

Cassandra for messages (write-heavy, partitioned by conversation_id, time-ordered).

15. Cache Strategy

Redis for sessions (user_id → connection server), last-seen timestamps.

16. Load Balancer Strategy

Sticky WebSocket (connection stays on one server).

17. Message Queue Usage

Kafka for message persistence pipeline (router → Kafka → storage consumer).

18. Scaling Strategy

Connection Service: 50 servers × 2M conns. Router: scales by msg/s.

19. Fault Tolerance

Multi-AZ. Cassandra RF=3. Connection drain: notify clients to reconnect.

20. Bottlenecks

- Group fanout: 1024-member group = 1024 fanout per message.
- Connection server memory: 2M conns × 50KB state = 100GB RAM.
- Push notification rate limits: FCM/APNs throttle.

21. Trade-offs

- **WebSocket vs MQTT:** WS is web-friendly; MQTT is more battery-efficient on mobile.
- **E2E encryption vs server features:** E2E blocks server-side search and moderation.
- **Persist-then-deliver vs deliver-then-persist:** Persist-first is safer; deliver-first is faster.

22. CAP Theorem Analysis

AP. Per-sender ordering via sequence numbers. Eventual consistency on read receipts.

23. Security Considerations

TLS for transport. Signal Protocol (Double Ratchet) for E2E. Media encrypted at rest in S3. Auth via OAuth/JWT.

24. Monitoring & Logging

Track delivery latency, connection count, push latency, storage write latency.

25. Deployment Strategy

Blue-green for stateless services. Cassandra rolling restarts. Connection drain on shutdown.

26. Interview Tips

- Emphasize connection layer vs routing layer separation.
- Discuss E2E encryption implications (no server-side search).
- Calculate connection count: 100M concurrent is substantial.
- Mention offline delivery via push notifications.

27. Common Follow-up Questions

- How would you implement message search across history?
- How would you handle voice/video calls at scale (WebRTC)?
- How would you implement "delete for everyone"?
- How would you scale to 10B messages/day?
- How would you detect and prevent spam?

28. Common Mistakes

- Single Postgres — won't scale to 2B users.
- No offline delivery (push notifications).
- No message ordering guarantee per sender.
- Synchronous media upload on send path.

29. Optimization Ideas

- MQTT instead of WebSocket for mobile (lower battery).
- Batch acks (1 per 100 messages).
- Compress media before upload.

- Pre-fetch recent conversations on app launch.

30. Final Summary

WhatsApp extends the chat pattern to global scale (2B users, 100B msgs/day). The architecture separates connection management (millions of WebSocket sessions) from message routing (stateless, horizontally scalable). E2E encryption (Signal Protocol) is non-negotiable in modern messaging. Group fanout (1024 members) is the main scaling challenge. Offline delivery via FCM/APNs is essential. The system is AP with per-sender ordering via sequence numbers.

QUESTION 23 INTERMEDIATE

Design YouTube

1. Problem Statement

Design a video-sharing platform. Users upload videos, watch them via streaming, like/comment, subscribe to channels, and see recommendations. Handle video transcoding, CDN distribution, and global scale.

2. Functional Requirements

- Upload videos (up to 256GB, 12 hours).
- Stream videos adaptively (DASH/HLS).
- Like, comment, subscribe.
- Recommendation feed based on watch history.
- Search by title, channel, tags.
- Live streaming.

3. Non-functional Requirements

- Availability 99.99%.
- Stream startup < 2s.
- Upload latency: < 30s processing for short videos.
- Recommendations refresh daily.

4. Capacity Estimation

2B users, 500 hours uploaded/minute. 1B hours watched/day. Storage: 500hr × 60min × 60s × 5MB/s × 5 res × 365 × 5 = ~700PB/5yr. Bandwidth: 1B hr × 1GB/hr / 86400 = ~12TB/s.

5. Back-of-the-envelope Math

Storage and bandwidth dominate. Use multi-tier storage: hot videos on CDN edge, warm on origin, cold in Glacier. Transcode pipeline: 5 resolutions × 2 codecs per upload.

6. API Design

```
POST /videos/init-upload body: {filename, size} -> {upload_url, video_id}
PUT {upload_url} (chunked upload to S3)
POST /videos/{id}/publish (after upload completes)
GET /videos/{id}/manifest -> DASH/HLS manifest
GET /feed -> recommended videos
POST /videos/{id}/like / POST /videos/{id}/comments
```

7. Database Schema

```
Table: videos
  video_id, channel_id, title, description, duration_sec,
  view_count, like_count, status (UPLOADING|PROCESSING|READY),
  created_at

Table: video_files (multiple resolutions/codecs per video)
  video_id, resolution, codec, bitrate, s3_key, size_bytes

Table: channels (channel_id, owner_id, name, subscriber_count)
Table: subscriptions (subscriber_id, channel_id, created_at)
Table: watch_history (user_id, video_id, watched_at, completion_pct)
```

8. High-Level Design (HLD)

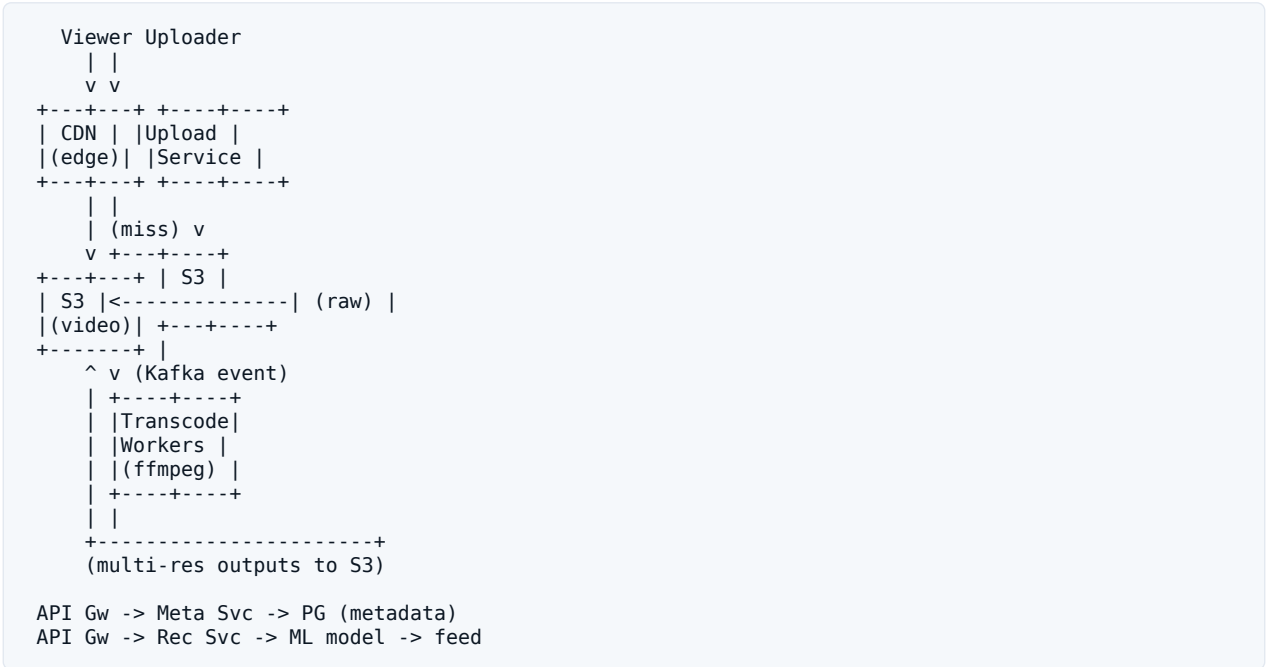
Client → CDN (video chunks) + API Gateway → Upload Service (resumable to S3) + Transcode Pipeline (Kafka + workers + ffmpeg) + Meta Service (CRUD) + Recommendation Service (ML).

9. Low-Level Design (LLD)

Upload: client requests presigned URL, uploads directly to S3 in chunks. On completion, publish event to Kafka. Transcode workers consume event, use ffmpeg to produce 144p, 240p, 480p, 720p, 1080p variants, store to S3, update video status. CDN pulls from S3 on first request, caches at edge.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Upload + transcode:
  1. Client requests upload URL from Upload Service.
  2. Client uploads directly to S3 (resumable, multipart).
  3. Client calls /publish; Upload Service publishes Kafka event.
  4. Transcode Worker consumes event.
  5. Worker runs ffmpeg: extract audio, transcode to 5 resolutions, generate thumbnails.
  6. Stores outputs to S3; updates video status to READY.
  7. Indexes metadata in Elasticsearch.

Playback:
  1. Client requests manifest from CDN.
  2. CDN fetches from S3 on miss; caches at edge.
  3. Client requests video chunks adaptively (DASH).
  4. CDN serves chunks from edge (15ms latency).
  5. Async: log watch event for recommendations.
```

12. Component Explanation

- **Upload Service:** Presigned URLs; resumable upload coordination.
- **Transcode Pipeline:** Kafka + ffmpeg workers; produces multi-res outputs.
- **CDN:** CloudFront/Akamai; caches video chunks at edge PoPs.
- **S3:** Origin storage; lifecycle rules to Glacier for cold videos.
- **Meta Service:** CRUD for video/channel metadata.
- **Recommendation Service:** ML model on watch history; daily batch + real-time signals.
- **Search:** Elasticsearch indexed via CDC.

13. Technology Stack

- **Language:** Python/Go for services; C++ for transcode workers.
- **DB:** PostgreSQL sharded by channel_id; Cassandra for watch history.
- **Cache:** Redis (metadata, view counts).
- **Queue:** Kafka (transcode events, watch events).
- **Storage:** S3 + CloudFront.
- **Search:** Elasticsearch.
- **ML:** TensorFlow/PyTorch for recommendations.
- **Deployment:** Kubernetes multi-region.

14. Database Choice

Postgres sharded by channel_id for metadata (ACID for likes). Cassandra for watch history (write-heavy, time-series). S3 for video files.

15. Cache Strategy

Redis for video metadata, view counters (batch-flushed to PG). CDN for video chunks.

16. Load Balancer Strategy

L7 LB; geographic routing for lower latency. CDN handles most video traffic.

17. Message Queue Usage

Kafka topics: transcode_jobs, watch_events, like_events.

18. Scaling Strategy

Transcode workers scale by queue depth. CDN scales with viewer count. Storage scales with S3 (effectively unlimited).

19. Fault Tolerance

Multi-region. S3 cross-region replication for DR. CDN failover to alternate PoP.

20. Bottlenecks

- Egress bandwidth on viral videos — CDN absorbs.
- Transcode queue depth during upload spikes — auto-scale workers.
- Recommendation ML training cost — daily batch + hourly incremental.

21. Trade-offs

- **Multi-res vs single-res:** Multi-res enables adaptive streaming; costs 5x storage.
- **Push to CDN vs pull:** Push popular videos; pull long-tail.
- **Batch vs real-time recommendations:** Batch is cheaper; real-time is fresher.

22. CAP Theorem Analysis

AP for video delivery (CDN eventually consistent). CP for likes/subscriptions.

23. Security Considerations

TLS. Content moderation (NSFW, copyright detection). DRM (Widevine) for premium. Rate limit uploads. Anti-piracy: watermarking.

24. Monitoring & Logging

Track upload success, transcode latency, CDN hit ratio, stream startup time, recommendation CTR.

25. Deployment Strategy

Blue-green for services. Transcode workers in dedicated node pools (GPU optional).

26. Interview Tips

- Emphasize transcode pipeline — core architectural piece.
- Calculate bandwidth: 1B hours/day is enormous — CDN is non-negotiable.
- Discuss adaptive streaming (DASH/HLS).
- Mention multi-resolution storage cost (5x raw video).

27. Common Follow-up Questions

- How would you implement live streaming (low-latency, < 5s)?
- How would you handle copyright detection (Content ID)?
- How would you implement "Skip Intro" or chapter markers?
- How would you detect and remove deepfakes?

- How would you implement Shorts/TikTok-style vertical videos?

28. Common Mistakes

- Single-resolution video — no adaptive streaming.
- No transcode pipeline — raw uploads eat bandwidth.
- Serving video from origin — kills egress costs and latency.
- Synchronous transcode on upload — UX suffers.

29. Optimization Ideas

- Pre-transcode popular resolutions only; lazy transcode rare ones.
- Use AV1 codec for new uploads (30% smaller than H.264).
- Pre-warm CDN for predicted viral videos.
- Use edge compute for A/B thumbnail selection.

30. Final Summary

YouTube tests video pipeline architecture at extreme scale. The core components are: resumable upload to S3, async transcode pipeline (Kafka + ffmpeg workers producing multi-res outputs), CDN-based delivery (DASH/HLS adaptive streaming), and ML-driven recommendations. Bandwidth dominates (12TB/s) so CDN is non-negotiable. Storage (700PB/5yr) requires multi-tier with Glacier for cold content. The architecture rewards investment in transcoding, CDN strategy, and ML.

QUESTION 24 INTERMEDIATE

Design Facebook News Feed

1. Problem Statement

Design the news feed system that ranks and serves posts to 2B users. Posts include text, photos, videos, links, and life events. Ranking uses engagement signals, recency, and user affinity.

2. Functional Requirements

- Generate ranked feed of posts from friends, groups, pages.
- Support multiple post types (text, photo, video, link, event).
- Real-time updates (new posts appear within 30s).
- Per-post engagement (like, comment, share, react).
- Personalized ranking based on user behavior.

3. Non-functional Requirements

- Feed generation < 1s for 95% of users.
- Availability 99.99%.
- 10B feed views/day.
- Ranking model updated daily; signals fresh.

4. Capacity Estimation

2B users, 1.5B DAU. 5B new posts/day. Feed reads: 10B/day = ~120K RPS avg, 1M peak. Storage: 5B × 1KB × 365 × 5 = ~9TB metadata + ~5PB media.

5. Back-of-the-envelope Math

Feed ranking is ML-driven. Pre-computation: per-user candidate set (top 1000 posts from followees) updated every 5 min. Real-time: rank top 100 by ML score on read.

6. API Design

```
GET /feed?cursor=... -> [ranked posts]
POST /posts body: {type, content, media?}
POST /posts/{id}/react body: {type: LIKE|LOVE|HAHA|...}
POST /posts/{id}/comment body: {text}
POST /posts/{id}/share
GET /posts/{id} -> single post view
```

7. Database Schema

```
Table: posts (post_id, author_id, type, content, media_url, created_at, ...)
Table: edges (source_id, target_id, type: FRIEND|FOLLOW|GROUP_MEMBER, weight)
Table: reactions (post_id, user_id, type, created_at) PK (post_id, user_id)
Table: comments (comment_id, post_id, user_id, text, created_at)
Table: feed_candidates (user_id, post_id, score, cached_at) -- pre-computed
```

8. High-Level Design (HLD)

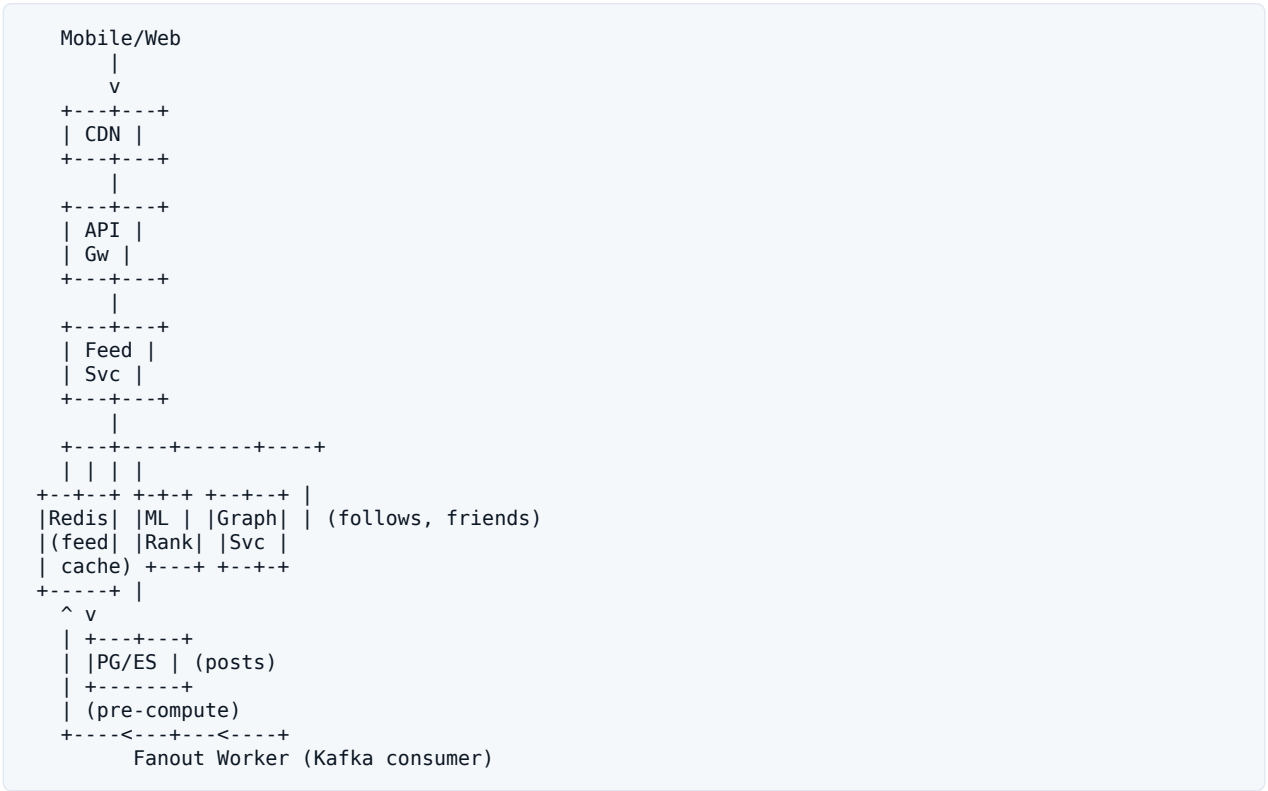
Mobile/Web → CDN → API Gateway → Feed Service → (1) Pre-computed candidates from Redis + (2) Real-time rank via ML model → ranked posts. Posts via Kafka to fanout workers.

9. Low-Level Design (LLD)

Feed generation: candidate generation (pull recent 1000 posts from followees, filter seen, score by ML model). Re-ranking: real-time ML model on top 100. Caching: per-user feed cached in Redis for 5 min. Celebrity pull-merge at read time.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Post publish + fanout:
  1. Author publishes post; Post Service writes to PG.
  2. Publish event to Kafka.
  3. Fanout Worker consumes event, reads author's followers from Graph Service.
  4. For each follower: add post_id to candidate set in Redis (sorted by timestamp).
  5. Pre-score using lightweight ML model (affinity, recency).

Feed read:
  1. GET /feed from Feed Service.
  2. Fetch user's candidate set from Redis (top 1000 posts).
  3. Filter already-seen posts (via seen_set in Redis).
  4. Re-rank top 100 using real-time ML model.
  5. Hydrate post objects from cache/PG.
  6. Return ranked feed; mark posts as seen.
```

12. Component Explanation

- **Feed Service:** Reads candidates, ranks, returns feed.
- **Post Service:** CRUD for posts.
- **Graph Service:** Friends/follows; adjacency lookups.
- **Fanout Worker:** Kafka consumer; pushes posts to candidate sets.
- **Ranking Service:** ML model (real-time) for re-ranking.
- **Redis:** Candidate sets, seen sets, post metadata cache.
- **Postgres:** Posts, reactions, comments (sharded).
- **Elasticsearch:** Search.

13. Technology Stack

- **Language:** Python/Go/C++ (ML in C++ for perf).
- **DB:** PostgreSQL + Cassandra for events.
- **Cache:** Redis Cluster (massive: petabytes).
- **Queue:** Kafka.
- **ML:** PyTorch/TensorFlow + custom C++ for inference.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Postgres sharded by user_id for posts (ACID). Cassandra for events. Redis for feed candidates (in-memory, fast access).

15. Cache Strategy

Multi-tier: CDN for media, Redis for candidate sets and post metadata, in-process LRU for hot posts.

16. Load Balancer Strategy

L7 LB with geographic routing. Sticky session not needed (stateless).

17. Message Queue Usage

Kafka topics: posts (fanout), reactions (counter updates), seen events (de-dup).

18. Scaling Strategy

Feed Service scales by read QPS. Fanout workers scale by post-publish rate. Redis scales by sharding candidate sets across cluster.

19. Fault Tolerance

Multi-region. Redis with replicas + failover. PG multi-AZ. Graceful degradation: if Redis down, fall back to live pull from PG.

20. Bottlenecks

- Celebrity fanout — pull model for >100K followers.
- Redis memory: petabytes — shard aggressively.
- ML inference latency — cache scores, batch inference.

21. Trade-offs

- **Push vs pull:** Hybrid (push normal, pull celebrities) — standard.
- **Pre-compute vs real-time rank:** Pre-compute candidates; real-time re-rank top 100.
- **Strong vs eventual consistency:** Eventual for feed (30s lag OK).

22. CAP Theorem Analysis

AP for feed (eventual consistency). CP for reactions (no double-react).

23. Security Considerations

TLS. Auth via OAuth. Content moderation (NSFW, hate speech). Rate limit posts. Anti-bot for reactions.

24. Monitoring & Logging

Track feed latency, fanout lag, ML inference latency, cache hit ratio.

25. Deployment Strategy

Blue-green. Multi-region active-active for reads. ML models deployed via feature flags.

26. Interview Tips

- Discuss push vs pull vs hybrid — core architectural decision.
- Mention ML ranking — not just recency sort.
- Calculate fanout: 5B posts × 200 followers avg = 1T fanout operations/day.
- Discuss seen-set de-duplication — common pitfall.

27. Common Follow-up Questions

- How would you handle a viral post (1B impressions in 1 hour)?
- How would you implement "Why am I seeing this?" explainability?
- How would you handle downranking of low-quality content?
- How would you A/B test ranking changes?

- How would you implement real-time feed updates (push new posts)?

28. Common Mistakes

- Pure pull feed — too slow at scale.
- No seen-set — users see same posts repeatedly.
- No celebrity exception — fanout explodes.
- Recency-only ranking — poor engagement.

29. Optimization Ideas

- Pre-warm feed for active users on app launch.
- Batch ML inference across users.
- Compress candidate sets (delta encoding).
- Edge-cache ranked feed for 30s.

30. Final Summary

Facebook News Feed extends Instagram's feed design with ML-driven ranking at extreme scale (2B users, 10B reads/day). The architecture is hybrid push/pull: push candidates to per-user Redis sets on post publish, pull and re-rank top 100 on read using a real-time ML model. The candidate set (top 1000) is pre-computed every 5 min; seen-set prevents repeats. Fanout for celebrities uses pull-merge. The system is AP — eventual consistency (30s lag) is acceptable. ML ranking is the differentiator: a strong candidate explains the feature pipeline (recency, affinity, engagement signals).

QUESTION 25 INTERMEDIATE

Design Uber

1. Problem Statement

Design a ride-sharing platform. Riders request rides; drivers accept; system matches based on location, dispatches nearest driver, tracks ride in real-time, and processes payment. Support surge pricing.

2. Functional Requirements

- Rider requests ride; specify pickup + destination.
- Match rider with nearest available driver.
- Real-time driver location tracking.
- Trip tracking (en route, picked up, dropped off).
- Fare calculation with surge pricing.
- Rating and payment after trip.

3. Non-functional Requirements

- Match latency < 5s.
- Driver location update < 1s.
- Availability 99.99%.
- Strong consistency for trip state.

4. Capacity Estimation

10M drivers, 100M riders, 1B trips/year. Peak: 1K requests/sec/city. Driver location updates: 10M × 1/sec = 10M updates/sec globally.

5. Back-of-the-envelope Math

Driver location is high-frequency, time-series data — use Redis geospatial (GEOADD). Trip state is transactional — use sharded Postgres.

6. API Design

```
POST /rides body: {pickup, dropoff} -> {ride_id, status: SEARCHING}
POST /rides/{id}/cancel
POST /drivers/{id}/location body: {lat, lng} (every 5s)
POST /drivers/{id}/status body: {AVAILABLE|BUSY|OFFLINE}
GET /rides/{id} -> {driver, status, location, eta}
// WebSocket /rides/{id}/stream for real-time updates
```

7. Database Schema

```
Table: rides
  ride_id, rider_id, driver_id, pickup_lat, pickup_lng, dropoff_lat, dropoff_lng,
  status (SEARCHING|MATCHED|EN_ROUTE|IN_TRIP|COMPLETED|CANCELLED),
  fare_cents, surge_multiplier, created_at, completed_at

Table: drivers (driver_id, name, car_model, rating, current_lat, current_lng, status)
Table: riders (rider_id, name, payment_method_id, rating)
```

8. High-Level Design (HLD)

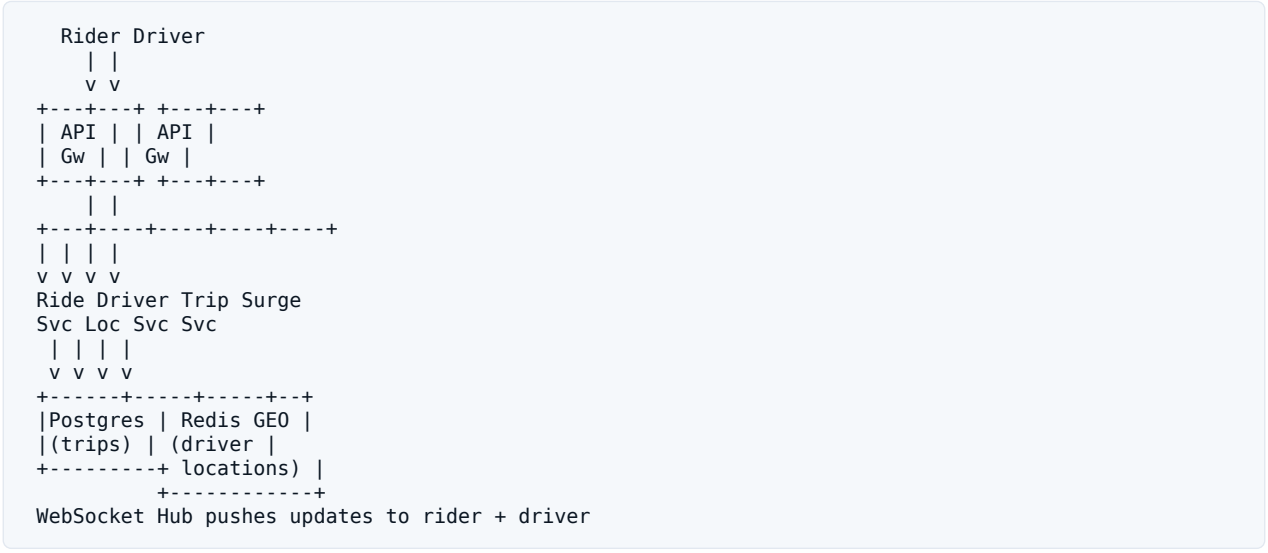
Rider App → API Gateway → Ride Service (matches) + Driver Location Service (geospatial index) + Trip Service (state machine) → Postgres + Redis (geospatial). WebSocket Hub for real-time updates to rider and driver.

9. Low-Level Design (LLD)

Match algorithm: Rider POST /rides with pickup. Ride Service queries Redis GEOSEARCH for nearest AVAILABLE drivers within 5km. Dispatches via WebSocket push; first driver to accept wins (atomic UPDATE with conditional WHERE status=AVAILABLE). Surge pricing: compute demand/supply ratio per geocell every minute.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Request ride:
  1. Rider POST /rides with pickup + dropoff.
  2. Ride Service creates ride (status=SEARCHING).
  3. Query Redis GEOSEARCH for nearest 5 AVAILABLE drivers within 5km.
  4. Push ride request to each driver via WebSocket (parallel).
  5. First driver to accept: atomic UPDATE rides SET driver_id, status=MATCHED
     WHERE ride_id=? AND status=SEARCHING. If 0 rows updated, another driver won.
  6. Notify rider + driver via WebSocket. Driver navigates to pickup.

Driver location updates:
  1. Driver app sends lat/lng every 5s to Driver Location Service.
  2. Update Redis GEOSET (lat/lng indexed for fast radius queries).
  3. If driver in active trip, also push to rider via WebSocket.
```

12. Component Explanation

- **Ride Service:** Matchmaking, ride state machine.
- **Driver Location Service:** Geospatial index in Redis; high-frequency updates.
- **Trip Service:** Trip lifecycle, fare calculation.
- **Surge Service:** Computes demand/supply ratio per geocell every minute.
- **Payment Service:** Charges rider, pays driver.
- **WebSocket Hub:** Real-time updates to rider + driver apps.
- **Redis GEO:** Geospatial index of driver locations.
- **Postgres:** Trip records, user profiles.

13. Technology Stack

- **Language:** Python/Go/Java.
- **DB:** PostgreSQL sharded by city.
- **Cache/Geo:** Redis Cluster with GEO commands.
- **Realtime:** WebSocket.
- **Queue:** Kafka (trip events, analytics).
- **Maps:** Google Maps API or OSRM (self-hosted).
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Postgres for trip records (ACID). Redis GEO for driver locations (sub-ms radius queries). Cassandra for trip event history (time-series).

15. Cache Strategy

Redis GEO is the geospatial index. Additional Redis cache for hot rider/driver profiles.

16. Load Balancer Strategy

L7 LB with geographic routing (rider routed to city-local region).

17. Message Queue Usage

Kafka for trip events (created, matched, completed) consumed by analytics + billing.

18. Scaling Strategy

Per-city sharding. Driver location Redis scales by geocell partition. Ride Service scales by request rate.

19. Fault Tolerance

Multi-AZ. Redis Cluster with replicas. Graceful: if Redis down, fall back to last known driver location.

20. Bottlenecks

- Driver location write throughput (10M/s) — shard Redis by geocell.

- Match contention in dense areas — limit dispatch radius.
- WebSocket connection count — scale Hub horizontally.

21. Trade-offs

- **Push to multiple drivers vs sequential:** Push parallel is faster but causes "ghost" rides.
- **Optimistic vs pessimistic lock on ride:** Optimistic (CAS) is faster under low contention.
- **Geocell size:** Smaller = precise but more cells; larger = fewer cells but less precise.

22. CAP Theorem Analysis

CP for trip state (no double-assignment). AP for driver location (slight staleness OK).

23. Security Considerations

TLS. Auth via OAuth. Background checks for drivers. Payment tokenization (PCI). PII encryption for trip history.

24. Monitoring & Logging

Track match latency, driver location update lag, ride completion rate, surge computation time.

25. Deployment Strategy

Blue-green. Redis rolling restarts with slot migration.

26. Interview Tips

- Use Redis GEO for driver locations — built-in geospatial commands.
- Discuss surge pricing as a separate service with its own compute cycle.
- Calculate driver location write rate: 10M drivers × 1/s = 10M writes/s globally.
- Mention match atomicity via conditional UPDATE.

27. Common Follow-up Questions

- How would you implement carpooling (UberPool)?
- How would you handle a sudden demand spike (concert let-out)?
- How would you implement driver supply prediction (ML)?
- How would you handle disputes (rider says driver never showed)?
- How would you implementScheduled rides (book for tomorrow)?

28. Common Mistakes

- Using Postgres for driver locations — too slow for radius queries.
- Sequential driver dispatch — slow match.
- No surge pricing — driver shortages during peak.
- Synchronous payment on trip end — blocks completion.

29. Optimization Ideas

- Pre-compute surge per geocell every minute (not per request).

- Use QUIC instead of TCP for driver location updates (lower overhead).
- Batch driver location updates (every 5s, not 1s).
- Geo-fenced push: only dispatch drivers within 5km.

30. Final Summary

Uber tests real-time geospatial architecture. The core components are: Redis GEO for driver location index (sub-ms radius queries, 10M writes/s), sharded Postgres for trip state (ACID for match atomicity), WebSocket Hub for real-time updates. Surge pricing is a separate service computing demand/supply per geocell every minute. Match uses parallel dispatch with conditional UPDATE (first-accept-wins). The system is CP for trip state, AP for driver location. Per-city sharding matches the geographic locality of rides.

QUESTION 26 INTERMEDIATE

Design Twitter/X

1. Problem Statement

Design a microblogging platform. Users post tweets (280 chars + media), follow others, see a timeline of recent tweets, search by hashtag, and trending topics.

2. Functional Requirements

- Post tweets (280 chars + image/video).
- Follow/unfollow users.
- View home timeline (recent tweets from followees).
- View user timeline (tweets by a specific user).
- Search tweets by text/hashtag.
- Trending topics by geography.

3. Non-functional Requirements

- Timeline generation < 500ms for 95%.
- Availability 99.99%.
- Tweet publish: < 5s to appear in followers' timelines.
- Read-heavy (100:1 read:write).

4. Capacity Estimation

300M DAU, 500M tweets/day, 5B timeline reads/day. Avg tweet 300B. Storage: 500M × 300B × 365 × 5 = ~275GB text + 5PB media. Read QPS: 60K avg, 600K peak.

5. Back-of-the-envelope Math

Push vs pull: hybrid. Push tweets to fanout Redis lists for normal users; pull for celebrity accounts (over 100K followers). Trending: count hashtag occurrences per geocell every 5 min.

6. API Design

```
POST /tweets body: {text, media?} -> {tweet_id}
GET /home_timeline?cursor=... -> [tweets]
GET /user_timeline/{user_id} -> [tweets]
POST /users/{id}/follow / DELETE
GET /search?q=...&f=live
GET /trends?lat=...&lng=...
```

7. Database Schema

```
Table: tweets (tweet_id, author_id, text, media_urls, created_at, like_count,
retweet_count)
Table: follows (follower_id, followee_id, created_at)
Table: likes (tweet_id, user_id) PK (tweet_id, user_id)
Table: trends (geocell, hashtag, count_5min, window_start) -- rolling counts
```

8. High-Level Design (HLD)

Mobile/Web → CDN → API Gateway → Tweet Service (publish) + Timeline Service (read) + Trends Service (counting) + Search Service (ES). Push fanout via Kafka + workers.

9. Low-Level Design (LLD)

On publish: Tweet Service writes to PG + Cassandra, publishes Kafka event. Fanout Worker pushes tweet_id to each follower's timeline Redis list. Celebrity exception: skip push; pull at read time. Trends: streaming aggregation per geocell every 5 min.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Tweet + fanout:
  1. POST /tweets; Tweet Service writes to PG + Cassandra.
  2. Publish tweet event to Kafka.
  3. Fanout Worker: read followers list.
  4. Push tweet_id to each follower's Redis timeline list (cap 1000 items).
  5. For celebrity author: skip push; mark for pull.
  6. Index tweet in Elasticsearch for search.
  7. Update trends counters (streaming aggregation).

Timeline read:
  1. GET /home_timeline from Timeline Service.
  2. Read user's Redis timeline list (last 50 tweet_ids).
  3. Merge in recent tweets from celebrities (pull).
  4. Hydrate tweet objects from cache/PG.
  5. Return ranked by recency.
```

12. Component Explanation

- **Tweet Service:** Publish tweets; CRUD.
- **Timeline Service:** Read home/user timelines.
- **Fanout Worker:** Kafka consumer; pushes to Redis timeline lists.
- **Trends Service:** Streaming aggregation per geocell.
- **Search Service:** Elasticsearch index.
- **Redis:** Timeline lists (per user).
- **Cassandra:** Tweets (write-heavy, time-series).

13. Technology Stack

- **Language:** Java/Scala (Twitter original) or Go.
- **DB:** Cassandra for tweets; Postgres for user/follows.
- **Cache:** Redis Cluster (timelines, counters).
- **Queue:** Kafka.
- **Search:** Elasticsearch.
- **Streaming:** Apache Storm or Flink for trends.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Cassandra for tweets (write-heavy, partitioned by author_id + time). Postgres for user/follows (ACID). Redis for timelines.

15. Cache Strategy

Redis for timeline lists (per user, capped at 1000 items). Counter cache for likes/retweets.

16. Load Balancer Strategy

L7 LB. Stateless service; no sticky session.

17. Message Queue Usage

Kafka topics: tweets (fanout), likes, retweets, trends.

18. Scaling Strategy

Fanout workers scale by tweet-publish rate. Cassandra scales by tweet volume. Redis Cluster for timelines.

19. Fault Tolerance

Multi-AZ. Cassandra RF=3. Redis with replicas + failover.

20. Bottlenecks

- Celebrity fanout — pull model for >100K followers.
- Trends computation latency — use streaming (Flink).
- Search indexing lag — acceptable for "live" search.

21. Trade-offs

- **Push vs pull timeline:** Hybrid — push normal, pull celebrities.
- **Cassandra vs PG for tweets:** Cassandra is write-optimized; PG has transactions.
- **Streaming vs batch trends:** Streaming is fresher; batch is cheaper.

22. CAP Theorem Analysis

AP for timelines (eventual consistency OK). CP for likes (no double-like).

23. Security Considerations

TLS. Auth via OAuth. Content moderation (hate speech, misinformation). Rate limit tweets. Anti-bot for trending manipulation.

24. Monitoring & Logging

Track fanout lag, timeline read latency, search latency, trends computation time.

25. Deployment Strategy

Blue-green. Cassandra rolling restarts. Multi-region for read latency.

26. Interview Tips

- Discuss push vs pull — core architectural decision.
- Mention celebrity exception explicitly.
- Calculate fanout: 500M tweets × 200 followers avg = 100B fanout ops/day.
- Discuss trends as streaming aggregation, not batch.

27. Common Follow-up Questions

- How would you handle a tweet going viral (1B impressions)?
- How would you implement threaded replies?
- How would you detect trending manipulation (coordinated hashtag campaigns)?
- How would you implement "For You" recommendations?
- How would you handle live audio (Twitter Spaces)?

28. Common Mistakes

- Pure pull timeline — too slow.
- Push for celebrities — fanout explodes.
- Batch trends — latency too high.
- PG for tweets — write throughput too low.

29. Optimization Ideas

- Pre-warm timelines for active users.
- Compress tweet text (delta encoding for replies).
- Use AVRO for Kafka messages (smaller than JSON).
- Edge-cache user timelines (they are read-heavy and stable).

30. Final Summary

Twitter/X is the canonical timeline architecture question. The core decision is push vs pull: push tweets to per-follower Redis lists on publish (fast read), pull for celebrities (>100K followers, fanout too expensive). Trends use streaming aggregation (Flink) per geocell. Search via Elasticsearch. The system is AP for timelines, CP for likes. Cassandra handles tweet writes; Redis handles timeline reads. The architecture rewards operational maturity: fanout lag monitoring, graceful degradation, multi-region deployment.

QUESTION 27

INTERMEDIATE

Design Amazon.com

1. Problem Statement

Design an e-commerce marketplace. Customers browse/search products, add to cart, checkout, pay, track orders. Sellers manage inventory, fulfill orders. Support recommendations and reviews.

2. Functional Requirements

- Browse products by category, search by keyword.
- Product detail page with images, specs, reviews.
- Shopping cart and checkout with payment.
- Order tracking and history.
- Seller portal: inventory, pricing, order fulfillment.
- Recommendations based on browse/buy history.

3. Non-functional Requirements

- Availability 99.99% (downtime = lost revenue).
- Search latency < 500ms.
- Checkout latency < 2s.
- Strong consistency for inventory (no oversell).

4. Capacity Estimation

300M users, 100M products, 10M orders/day. Storage: 100M × 5KB product metadata = 500GB; 10M orders × 5yr × 2KB = 100TB. Search index 1TB. Peak QPS: 1M (Black Friday).

5. Back-of-the-envelope Math

Inventory is the hard consistency problem. Shard by product_id. Cart in Redis. Order history in Postgres (ACID).

6. API Design

```
GET /search?q=...&category=...&page=1
GET /products/{id}
POST /cart/items body: {product_id, quantity}
POST /checkout body: {cart_id, payment, address}
GET /orders/{id}
POST /seller/products body: {sku, price, stock}
POST /seller/orders/{id}/ship body: {tracking_id}
```

7. Database Schema

```
Table: products (product_id, title, description, price, category_id, seller_id, images,
...)
Table: inventory (product_id, seller_id, stock_count, reserved_count)
Table: orders (order_id, user_id, total, status, payment_id, address_id, created_at)
Table: order_items (order_id, product_id, quantity, price_at_purchase)
Table: reviews (review_id, product_id, user_id, rating, text, created_at)
```

8. High-Level Design (HLD)

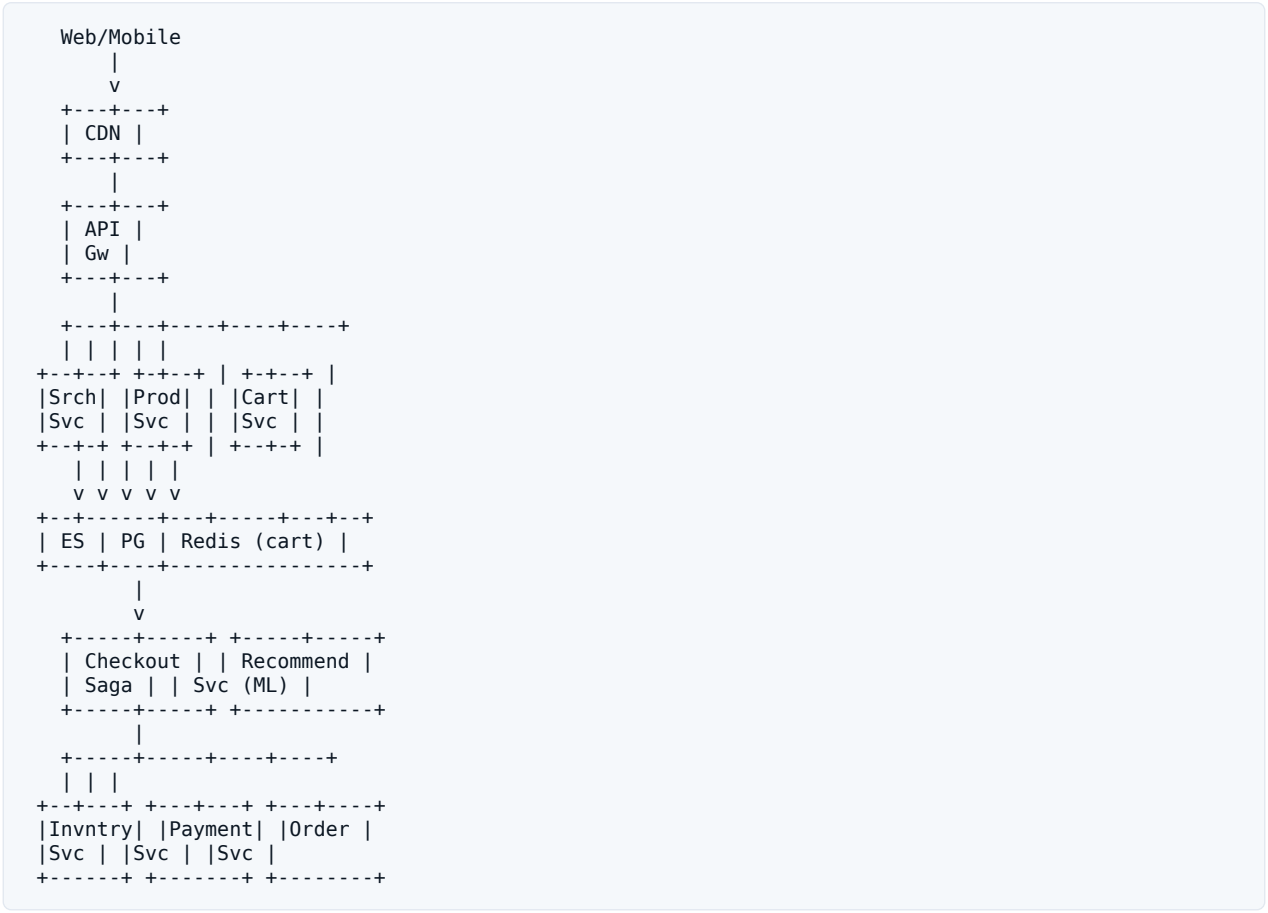
Web/Mobile → CDN → API Gateway → Search Service (ES) + Product Service + Cart Service + Checkout Service (orchestrates inventory, payment, order) + Recommendation Service.

9. Low-Level Design (LLD)

Checkout is a saga: reserve inventory → charge payment → create order → decrement inventory → notify seller. Each step compensates on failure (e.g. release inventory if payment fails).

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Checkout (saga):
  1. POST /checkout with cart_id, payment, address.
  2. Checkout Service: BEGIN saga.
  3. Call Inventory Service: reserve items for 10 min.
  4. Call Payment Service: charge (idempotent via order_id).
  5. On success: create order, decrement inventory, clear cart.
  6. Publish order.created event to Kafka.
  7. Notify seller via email + seller portal.
  8. On failure (e.g. payment declined): release reservation, return error.
  9. Compensating transactions ensure consistency.
```

12. Component Explanation

- **Search Service:** Elasticsearch; product search with facets.
- **Product Service:** CRUD for products; PG-backed.
- **Cart Service:** Redis-backed; fast CRUD.
- **Checkout Service:** Saga orchestrator; transactional.
- **Inventory Service:** Reserves and deducts stock; ACID.
- **Payment Service:** Stripe wrapper; idempotent.
- **Order Service:** Order records; PG-backed.
- **Recommendation Service:** ML model on browse/buy history.

13. Technology Stack

- **Language:** Java/Python/Go.
- **DB:** PostgreSQL sharded by product_id for inventory; by user_id for orders.
- **Cache:** Redis Cluster (cart, hot products).
- **Search:** Elasticsearch.
- **Queue:** Kafka (order events).
- **ML:** TensorFlow for recommendations.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Postgres for inventory (ACID for no-oversell) and orders. Redis for cart. ES for search.

15. Cache Strategy

Redis for cart, hot product pages, recommendation results. CDN for product images.

16. Load Balancer Strategy

L7 LB. Stateless services. Geographic routing for lower latency.

17. Message Queue Usage

Kafka for order events, inventory updates, recommendation signals.

18. Scaling Strategy

Shard PG by product_id (inventory) and user_id (orders). ES scales search. Cart Redis Cluster scales with active carts.

19. Fault Tolerance

Multi-AZ. Saga with compensating transactions ensures consistency. Payment idempotent via order_id.

20. Bottlenecks

- Hot product inventory contention (Black Friday) — use SELECT FOR UPDATE; queue requests.
- Search latency under load — pre-compute facets; cache common queries.
- Recommendation ML inference — batch + cache.

21. Trade-offs

- **Saga vs 2PC:** Saga is eventually consistent; 2PC is strong but slow.
- **Reserve at cart vs checkout:** Checkout avoids abandoned cart inventory loss.
- **Sync vs async payment:** Sync blocks; async risks unpaid orders.

22. CAP Theorem Analysis

CP for inventory (no oversell). AP for search (stale inventory count OK in display).

23. Security Considerations

TLS. PCI compliance (tokenized payments). Auth via OAuth. Anti-fraud: detect unusual order patterns. PII encryption for addresses.

24. Monitoring & Logging

Track search latency, checkout conversion, inventory conflict rate, payment success rate.

25. Deployment Strategy

Blue-green. Pre-scale for Black Friday (10x normal). Multi-region active-active.

26. Interview Tips

- Emphasize saga pattern for checkout — core architectural piece.
- Discuss inventory reservation (no oversell).
- Mention Black Friday pre-scaling — operational maturity.
- Calculate peak QPS: 1M peak is substantial.

27. Common Follow-up Questions

- How would you handle a flash sale (1M users buying same product)?
- How would you implement dynamic pricing?
- How would you handle returns and refunds?
- How would you implement multi-seller marketplace (e.g. eBay)?
- How would you detect fake reviews?

28. Common Mistakes

- No inventory reservation — oversell on concurrent checkout.
- Synchronous saga steps — slow checkout.
- No payment idempotency — double-charge on retry.
- Single PG instance — won't scale.

29. Optimization Ideas

- Pre-compute product availability counts for fast UI.
- Cache recommendation results per user for 1h.
- Batch inventory reservations (single SELECT FOR UPDATE for multi-item cart).
- Edge-cache product images aggressively.

30. Final Summary

Amazon tests e-commerce architecture at scale. The core challenge is transactional checkout across inventory, payment, and order services — solved with the saga pattern (compensating transactions). Inventory uses ACID Postgres with SELECT FOR UPDATE to prevent oversell. Cart is Redis-backed for speed. Search delegates to Elasticsearch. Recommendations via ML. The architecture rewards operational maturity: Black Friday pre-scaling, payment idempotency, multi-region deployment.

QUESTION 28

INTERMEDIATE

Design Netflix

1. Problem Statement

Design a video streaming service. Users browse a catalog, watch videos via adaptive streaming, receive personalized recommendations, and resume playback across devices. Support multiple profiles per account.

2. Functional Requirements

- Browse catalog by category, search by title.
- Watch videos via adaptive streaming (DASH).
- Personalized recommendation rows.
- Resume playback across devices.
- Multiple profiles per account.
- Downloads for offline viewing.

3. Non-functional Requirements

- Availability 99.99%.
- Stream startup < 2s.
- Recommendations refresh hourly.
- Handle 200M concurrent streams globally.

4. Capacity Estimation

250M subscribers, 200M concurrent streams peak. Catalog: 50K titles × 5GB avg = 250TB. Bandwidth: 200M × 5Mbps = 1Tbps egress. Watch history: 1B events/day.

5. Back-of-the-envelope Math

CDN is non-negotiable (1Tbps egress). Netflix uses Open Connect (their own CDN). Recommendations via ML batch + real-time. Playback position in DynamoDB.

6. API Design

```
GET /catalog?category=...&row=...
GET /search?q=...
GET /videos/{id}/manifest -> DASH manifest
POST /videos/{id}/playback body: {position} (every 30s)
GET /recommendations?profile_id=...
POST /downloads/{video_id} (offline viewing)
```

7. Database Schema

```
Table: videos (video_id, title, description, duration, rating, ...)
Table: video_files (video_id, resolution, codec, bitrate, s3_key)
Table: accounts (account_id, plan, billing_id)
Table: profiles (profile_id, account_id, name, preferences)
Table: watch_history (profile_id, video_id, position_sec, completion, watched_at)
Table: recommendations (profile_id, video_id, score, row_type, refreshed_at)
```

8. High-Level Design (HLD)

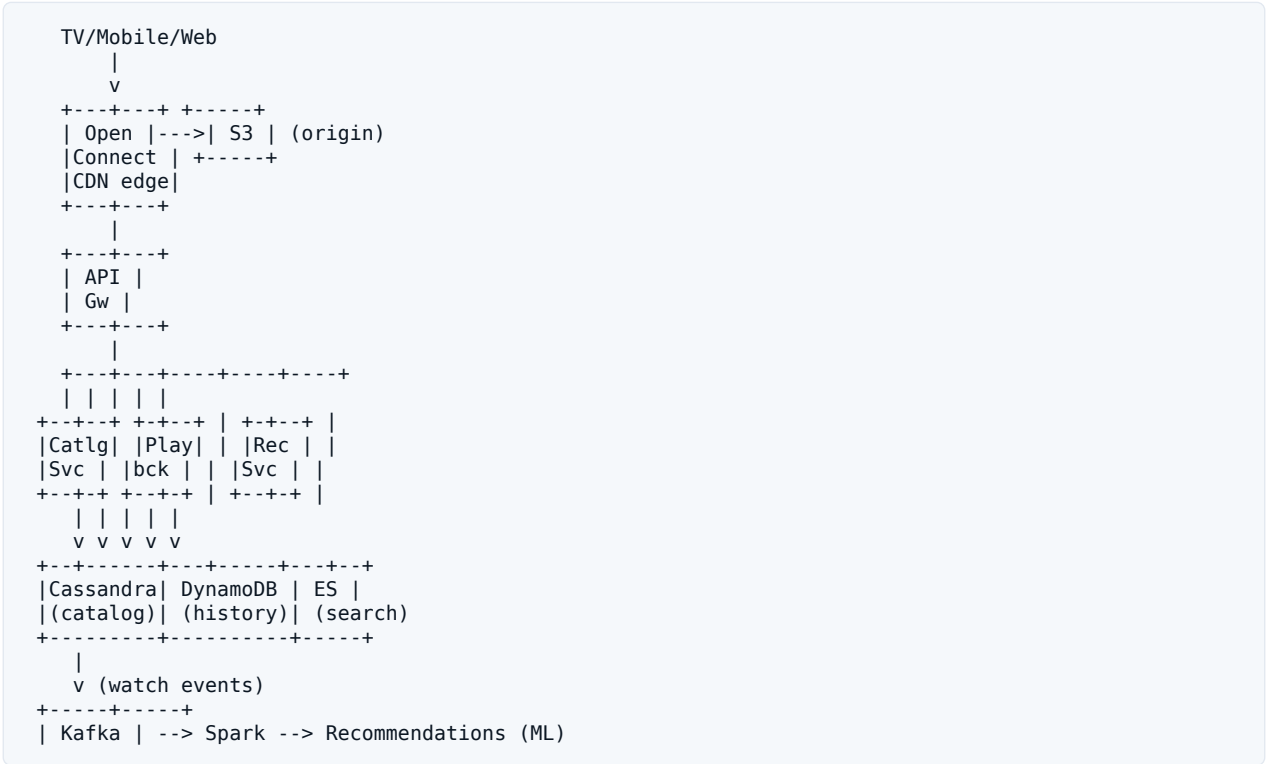
TV/Mobile/Web → CDN (video chunks) + API Gateway → Catalog Service + Playback Service + Recommendation Service + Profile Service. CDN is Open Connect appliances at ISPs.

9. Low-Level Design (LLD)

On play: client requests DASH manifest from CDN. CDN fetches from origin (S3) on miss, caches at edge. Client requests chunks adaptively based on bandwidth. Playback position reported every 30s to Playback Service (DynamoDB) for cross-device resume.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Playback:
  1. User clicks "Play". App requests DASH manifest from CDN.
  2. CDN serves manifest (or fetches from origin on miss).
  3. App requests first chunk; CDN serves from edge (15ms latency).
  4. App adapts bitrate based on bandwidth (DASH).
  5. Every 30s: app reports playback position to Playback Service.
  6. On stop: final position recorded. Resume on any device.

Recommendations:
  1. Watch events stream to Kafka.
  2. Spark job hourly: compute per-profile recommendations.
  3. Store in DynamoDB; serve via Recommendation Service.
  4. Real-time: bias top rows by recently watched.
```

12. Component Explanation

- **Catalog Service:** CRUD for video metadata.
- **Playback Service:** Track position; cross-device resume.
- **Recommendation Service:** ML-driven rows.
- **Profile Service:** Per-account profiles.
- **Open Connect CDN:** Netflix-owned CDN appliances at ISPs.
- **S3:** Origin storage for video files.
- **DynamoDB:** Watch history, playback positions.
- **Cassandra:** Catalog (denormalized for fast reads).
- **Kafka + Spark:** Recommendation pipeline.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** Cassandra (catalog), DynamoDB (history).
- **Cache:** Redis (hot metadata), EVCache (Netflix custom).
- **CDN:** Open Connect (Netflix-owned).
- **Storage:** S3.
- **Streaming:** DASH/HLS.
- **ML:** PyTorch + custom inference.
- **Deployment:** AWS multi-region.

14. Database Choice

Cassandra for catalog (denormalized, fast reads). DynamoDB for watch history (write-heavy, time-series). S3 for video files.

15. Cache Strategy

Redis/EVCache for hot video metadata, recommendation rows. CDN for video chunks.

16. Load Balancer Strategy

L7 LB with geographic routing. CDN handles most video traffic.

17. Message Queue Usage

Kafka for watch events. Spark for batch ML.

18. Scaling Strategy

CDN scales with viewer count. Cassandra scales with catalog size. DynamoDB auto-scales.

19. Fault Tolerance

Multi-region. CDN failover to alternate PoP. S3 cross-region replication.

20. Bottlenecks

- Egress bandwidth (1Tbps) — Open Connect handles.
- Recommendation ML training — hourly batch + real-time bias.
- Catalog read latency — cache aggressively.

21. Trade-offs

- **Own CDN vs cloud CDN:** Own (Open Connect) is cheaper at scale; cloud is simpler.
- **Batch vs real-time recommendations:** Batch is cheaper; real-time is fresher.
- **Multi-res vs single-res:** Multi-res enables adaptive streaming; costs 5x storage.

22. CAP Theorem Analysis

AP for video delivery (CDN eventually consistent). CP for playback position (no lost progress).

23. Security Considerations

TLS. DRM (Widevine, FairPlay) for premium content. Auth via OAuth. Profile PINs for kids. Anti-piracy: watermarking.

24. Monitoring & Logging

Track stream startup time, rebuffer rate, CDN hit ratio, recommendation CTR.

25. Deployment Strategy

Blue-green. Multi-region active-active. CDN appliances managed separately.

26. Interview Tips

- Emphasize CDN — 1Tbps egress is impossible without edge caching.
- Discuss Open Connect — shows real-world awareness.
- Mention adaptive streaming (DASH).
- Discuss ML pipeline for recommendations.

27. Common Follow-up Questions

- How would you implement "Skip Intro"?
- How would you handle live streaming (e.g. NFL games)?
- How would you implement interactive content (Black Mirror: Bandersnatch)?

- How would you detect account sharing?
- How would you optimize for low-bandwidth markets (mobile-only)?

28. Common Mistakes

- No CDN — 1Tbps egress is impossible from origin.
- Single-resolution video — no adaptive streaming.
- Synchronous recommendation computation — kills API latency.
- No cross-device resume — poor UX.

29. Optimization Ideas

- Pre-fetch next episode's manifest on autoplay countdown.
- Use AV1 codec for new content (30% smaller than H.264).
- Cache recommendation rows at edge for 1h.
- Predictive pre-fetch based on user patterns.

30. Final Summary

Netflix tests video streaming architecture at extreme scale (1Tbps egress). The core components are: Open Connect CDN (Netflix-owned appliances at ISPs for cheap egress), Cassandra catalog (denormalized for fast reads), DynamoDB watch history (write-heavy), DASH adaptive streaming (multi-resolution), ML recommendation pipeline (hourly batch + real-time bias). The architecture rewards investment in CDN strategy and ML. The system is AP for video delivery, CP for playback position.

QUESTION 29 INTERMEDIATE

Design Spotify

1. Problem Statement

Design a music streaming service. Users search and play songs, create playlists, follow artists, receive recommendations, and cache songs for offline playback.

2. Functional Requirements

- Search songs, artists, albums.
- Play songs via streaming (with offline cache).
- Create and share playlists.
- Follow artists; get notified of new releases.
- Personalized recommendations (Discover Weekly, Daily Mixes).
- Podcasts and live audio.

3. Non-functional Requirements

- Availability 99.95%.
- Stream startup < 1s.
- Recommendations weekly refresh + real-time bias.
- Offline mode (downloaded songs play without network).

4. Capacity Estimation

500M users, 100M songs. Avg song 4MB. Storage: 100M × 4MB × 3 bitrates = ~1.2PB. Concurrent streams: 30M. Bandwidth: 30M × 320kbps = ~10Tbps peak.

5. Back-of-the-envelope Math

CDN for song delivery. ML for recommendations (matrix factorization + deep learning). Offline cache on device (encrypted).

6. API Design

```
GET /search?q=...&type=track|artist|album
GET /tracks/{id}/stream?quality=... -> audio stream URL
POST /playlists body: {name, tracks: [...]}
GET /recommendations?seed=...
POST /downloads/{track_id} (offline cache)
POST /playback body: {track_id, position, event: PLAY|PAUSE|SKIP}
```

7. Database Schema

```
Table: tracks (track_id, title, artist_id, album_id, duration, popularity, audio_key)
Table: artists (artist_id, name, monthly_listeners)
Table: playlists (playlist_id, owner_id, name, track_ids, created_at)
Table: play_history (user_id, track_id, played_at, completion)
Table: recommendations (user_id, track_id, score, source)
```

8. High-Level Design (HLD)

Mobile/Desktop → CDN (audio chunks) + API Gateway → Search Service + Playback Service + Recommendation Service + Playlist Service. ML pipeline for recommendations.

9. Low-Level Design (LLD)

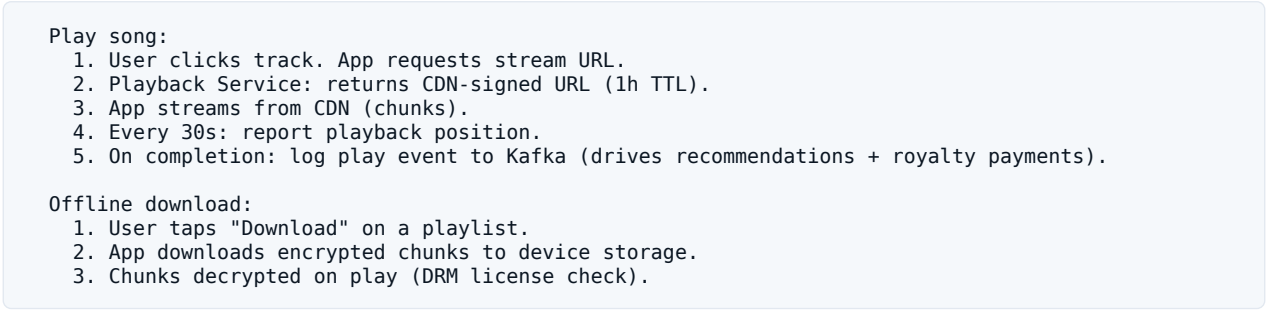
On play: client requests audio URL from Playback Service. Returns CDN URL (signed, TTL 1h). Client streams from CDN. Offline: client downloads encrypted chunks to device.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Search Service:** Elasticsearch index.
- **Playback Service:** Generate signed CDN URLs.
- **Playlist Service:** CRUD for playlists.
- **Recommendation Service:** ML-driven (Discover Weekly, Daily Mixes).
- **CDN:** Audio chunk delivery.
- **S3:** Origin audio storage.
- **Cassandra:** Track metadata, play history.
- **Kafka + Spark:** Recommendation pipeline.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** Cassandra (catalog, history).
- **Cache:** Redis (hot metadata, recent plays).
- **CDN:** CloudFront/Akamai.
- **Storage:** S3.
- **ML:** PyTorch (matrix factorization + DL).
- **DRM:** Widevine.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Cassandra for catalog and play history (write-heavy, time-series). Redis for hot metadata.

15. Cache Strategy

Redis for hot tracks (top charts), user play queue. CDN for audio.

16. Load Balancer Strategy

L7 LB. Geographic routing.

17. Message Queue Usage

Kafka for play events. Spark for batch ML (Discover Weekly).

18. Scaling Strategy

CDN scales with streams. Cassandra scales by writes. ML pipeline scales by event volume.

19. Fault Tolerance

Multi-region. CDN failover. Cassandra RF=3.

20. Bottlenecks

- Egress bandwidth (10Tbps) — CDN.
- Recommendation ML training — weekly batch + real-time bias.
- Royalty computation — batch daily.

21. Trade-offs

- **Streaming vs download-only:** Streaming is modern; download saves bandwidth.
- **DRM vs no DRM:** DRM prevents piracy; adds complexity.
- **Batch vs real-time recommendations:** Batch is cheaper; real-time is fresher.

22. CAP Theorem Analysis

AP for streaming (CDN). CP for play history (royalty accounting).

23. Security Considerations

TLS. DRM (Widevine). Auth via OAuth. Anti-piracy: watermarking. PII encryption.

24. Monitoring & Logging

Track stream startup, rebuffer rate, recommendation CTR, royalty accuracy.

25. Deployment Strategy

Blue-green. Multi-region. CDN managed separately.

26. Interview Tips

- Emphasize ML recommendation pipeline — core differentiator.
- Discuss offline mode with DRM — UX + security.
- Mention royalty accounting — shows business awareness.
- Calculate bandwidth: 10Tbps peak needs CDN.

27. Common Follow-up Questions

- How would you implement collaborative playlists?
- How would you detect and recommend similar songs (audio fingerprinting)?
- How would you implement lyrics sync?
- How would you handle live audio (Spotify Live)?
- How would you compute royalties accurately across countries?

28. Common Mistakes

- No DRM — users rip songs for free.
- Synchronous recommendation computation — kills latency.
- No offline mode — poor mobile UX.
- Single bitrate — no adaptive streaming.

29. Optimization Ideas

- Pre-fetch next track in queue (predictive).
- Use Opus codec (better than MP3 at low bitrates).
- Cache recommendation rows per user for 1h.
- Edge-cache popular tracks (top 1000 globally).

30. Final Summary

Spotify tests streaming architecture with ML-driven personalization. The core components are: CDN for audio delivery, Cassandra for catalog and play history, ML pipeline (matrix factorization + deep learning) for recommendations, DRM for offline mode. The architecture rewards investment in ML (Discover Weekly is the key product differentiator). Royalty accounting is a non-obvious concern: play events must be accurately counted and attributed. The system is AP for streaming, CP for play history.

QUESTION 30 INTERMEDIATE

Design Dropbox / Google Drive

1. Problem Statement

Design a cloud file storage service. Users sync files across devices, share folders, view version history, and edit documents collaboratively. Handle large files, conflicts, and offline edits.

2. Functional Requirements

- Upload/download files; sync across devices.
- Share files/folders with permissions.
- Version history (recover previous versions).
- Collaborative editing (Google Docs).
- Search across file contents.
- Offline edits with conflict resolution.

3. Non-functional Requirements

- Availability 99.99%.
- Sync latency < 5s for small files.
- Strong consistency for file metadata.
- Handle files up to 1TB.

4. Capacity Estimation

500M users, 2GB avg storage = 1EB. 10B files. Block storage: deduplicated = ~500PB. Metadata DB: 10B × 1KB = 10TB.

5. Back-of-the-envelope Math

Files chunked into 4MB blocks; blocks deduplicated by hash. S3 for block storage. Metadata in sharded Postgres. Sync via WebSocket notifications.

6. API Design

```
POST /files body: {path, size} -> {upload_id, chunk_size}
PUT /files/{id}/chunks/{n} body: block_data -> {block_hash}
POST /files/{id}/complete
GET /files/{id}/download -> chunk URLs
POST /files/{id}/share body: {user_id, permission}
GET /changes?since=... (sync)
```

7. Database Schema

```
Table: files (file_id, owner_id, path, size, created_at, modified_at, current_version)
Table: file_versions (version_id, file_id, block_hashes, created_at)
Table: blocks (block_hash, size, s3_key, ref_count) -- dedup by hash
Table: permissions (file_id, user_id, role: OWNER|EDITOR|VIEWER)
Table: devices (device_id, user_id, last_sync_cursor)
```

8. High-Level Design (HLD)

Desktop/Mobile → API Gateway → File Service (metadata) + Block Service (S3) + Sync Service (WebSocket notifications) + Search Service (ES).

9. Low-Level Design (LLD)

Upload: client chunks file into 4MB blocks, hashes each, asks server which blocks are new (dedup), uploads only new blocks to S3. Server creates new file_version with block list. Notifies other devices via WebSocket.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Upload (with dedup):
  1. Client chunks file into 4MB blocks; SHA-256 each.
  2. POST /files with block hashes; server says which are missing.
  3. Client uploads missing blocks to S3 via presigned URLs.
  4. Server creates file_version with block list.
  5. Updates file metadata in PG.
  6. Publishes sync event to Kafka.
  7. Sync Service pushes notification to other devices via WebSocket.
  8. Other devices fetch new file_version on next sync.
```

12. Component Explanation

- **File Service:** Metadata CRUD; version history.
- **Block Service:** Manages S3 block storage; dedup by hash.

- **Sync Service:** WebSocket notifications to devices.
- **Search Service:** Elasticsearch for full-text file search.
- **S3:** Block storage (deduplicated).
- **Postgres:** File metadata (sharded by owner_id).
- **Redis:** Active sync cursors; notification queue.

13. Technology Stack

- **Language:** Python/Go.
- **DB:** PostgreSQL sharded by owner_id.
- **Storage:** S3 (blocks, deduped).
- **Cache:** Redis (notifications, hot metadata).
- **Realtime:** WebSocket.
- **Search:** Elasticsearch.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Postgres for metadata (ACID). S3 for blocks (cheap, durable, deduplicated by hash).

15. Cache Strategy

Redis for hot file metadata and sync notification queue.

16. Load Balancer Strategy

L7 LB. Sticky session for WebSocket.

17. Message Queue Usage

Kafka for sync events and CDC to ES.

18. Scaling Strategy

Shard PG by owner_id. S3 scales with storage. Block dedup saves 30-50% space.

19. Fault Tolerance

Multi-AZ. S3 cross-region replication. PG with replica.

20. Bottlenecks

- Large file uploads — chunk + parallel upload.
- Sync notification fanout for shared folders — batch.
- Search indexing lag — acceptable.

21. Trade-offs

- **Block dedup vs no dedup:** Dedup saves space; adds hash computation.
- **WebSocket vs polling sync:** WebSocket is real-time; polling is simpler.

- **Last-write-wins vs CRDT:** LWW is simple; CRDT enables collab editing.

22. CAP Theorem Analysis

CP for file metadata. AP for sync notifications (slight delay OK).

23. Security Considerations

TLS. Per-file permissions. Encryption at rest (S3 SSE). Auth via OAuth. Audit log for shared file access.

24. Monitoring & Logging

Track upload/download latency, sync notification latency, dedup ratio, storage cost.

25. Deployment Strategy

Blue-green. S3 lifecycle rules for cold blocks (Glacier).

26. Interview Tips

- Emphasize block-level dedup — core architectural piece.
- Discuss chunked upload for large files.
- Mention version history via immutable block references.
- Discuss sync notifications via WebSocket.

27. Common Follow-up Questions

- How would you implement real-time collaborative editing (Google Docs)?
- How would you handle file conflicts from concurrent edits?
- How would you implement "shared with me" view?
- How would you detect and remove malware in uploads?
- How would you handle a 1TB file upload?

28. Common Mistakes

- No chunking — large files fail.
- No dedup — storage explodes.
- No version history — lost data on overwrite.
- Polling for sync — high latency.

29. Optimization Ideas

- Parallel chunk uploads for large files.
- Pre-compute file hashes on client (dedup before upload).
- Edge-cache frequently accessed blocks.
- Compress blocks before S3 PUT.

30. Final Summary

Dropbox/Google Drive tests file storage with sync and deduplication. The core insight is chunking files into blocks (4MB) and deduplicating by hash (SHA-256) — saves 30-50% storage. Block storage goes to S3; metadata

to sharded Postgres; sync notifications via WebSocket + Kafka. Version history uses immutable block references (new version = new block list). The architecture rewards operational thinking: chunked uploads, dedup, and graceful conflict resolution.

QUESTION 31 INTERMEDIATE

Design Online Food Delivery (DoorDash/Swiggy)

1. Problem Statement

Design a food delivery platform. Users browse restaurants, place orders, restaurants receive and prepare, delivery partners pick up and deliver. Real-time tracking.

2. Functional Requirements

- Browse restaurants by location, cuisine, rating.
- Place order with items, quantity, address.
- Restaurant accepts and prepares order.
- Delivery partner assigned; picks up; delivers.
- Real-time order tracking for user.
- Payment and rating.

3. Non-functional Requirements

- Order placement < 2s.
- Driver assignment < 30s.
- Real-time location updates < 5s.
- Availability 99.95%.

4. Capacity Estimation

50M users, 100K restaurants, 500K drivers, 5M orders/day. Driver location updates: 500K × 1/5s = 100K updates/s.

5. Back-of-the-envelope Math

Driver location: Redis GEO. Order state machine: sharded Postgres. Restaurant catalog: ES.

6. API Design

```
GET /restaurants?lat=...&lng=...&cuisine=...
POST /orders body: {restaurant_id, items, address}
POST /orders/{id}/accept (restaurant)
POST /orders/{id}/assign (system to driver)
POST /drivers/{id}/location body: {lat, lng}
GET /orders/{id}/track -> WebSocket stream
```

7. Database Schema

```
Table: orders (order_id, user_id, restaurant_id, items_json, status, total, address,
driver_id)
Table: restaurants (restaurant_id, name, lat, lng, cuisine, rating, is_open)
Table: drivers (driver_id, name, current_lat, current_lng, status: AVAILABLE|BUSY)
Table: deliveries (delivery_id, order_id, driver_id, pickup_time, dropoff_time)
```

8. High-Level Design (HLD)

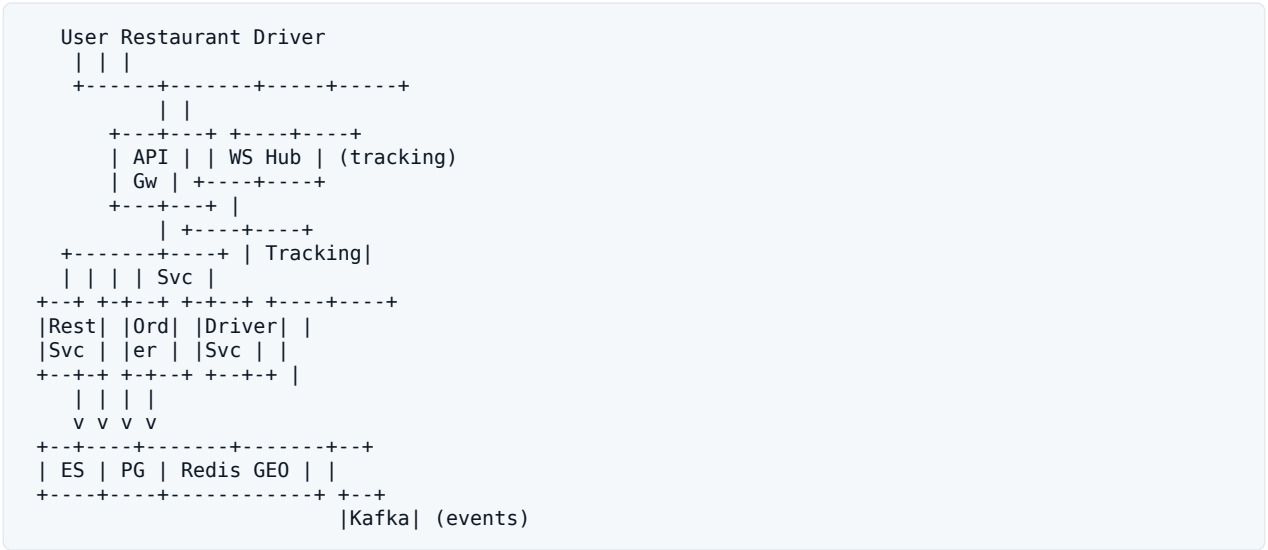
Mobile → API Gateway → Restaurant Service + Order Service + Driver Service (Redis GEO) + Tracking Service (WebSocket). Kafka for order events.

9. Low-Level Design (LLD)

Order lifecycle: PLACED → ACCEPTED (restaurant) → PREPARING → READY → PICKED_UP → DELIVERED. Driver assignment: nearest AVAILABLE driver within 5km of restaurant.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Place order:
  1. User POST /orders; Order Service writes to PG (PLACED).
  2. Publish event to Kafka; Restaurant Service consumes, notifies restaurant.
  3. Restaurant POST /orders/{id}/accept; status=ACCEPTED.
  4. When READY: trigger driver assignment.
  5. Driver Service: Redis GEOSEARCH nearest AVAILABLE driver.
  6. Push to driver; first accept wins (atomic UPDATE).
  7. Driver picks up; status=PICKED_UP; WebSocket tracking starts.
  8. Driver delivers; status=DELIVERED; payment released.
```

12. Component Explanation

- **Restaurant Service:** Catalog, menu, hours.
- **Order Service:** Order state machine.
- **Driver Service:** Geospatial index, assignment.
- **Tracking Service:** WebSocket for real-time location.

- **Payment Service:** Charges user, pays restaurant + driver.
- **Redis GEO:** Driver locations.
- **Kafka:** Order events for analytics.

13. Technology Stack

- **Language:** Java/Python/Go.
- **DB:** PostgreSQL sharded by city.
- **Cache/Geo:** Redis GEO.
- **Realtime:** WebSocket.
- **Queue:** Kafka.
- **Deployment:** Multi-region Kubernetes.

14. Database Choice

Postgres for orders (ACID state machine). Redis GEO for driver locations.

15. Cache Strategy

Redis for restaurant catalog (hot restaurants). Redis GEO for driver locations.

16. Load Balancer Strategy

L7 LB with geographic routing.

17. Message Queue Usage

Kafka for order events (placed, accepted, delivered).

18. Scaling Strategy

Per-city sharding. Driver Redis scales by geocell.

19. Fault Tolerance

Multi-AZ. Redis with replicas.

20. Bottlenecks

- Driver location write throughput — shard Redis.
- Order contention in dense areas — queue requests.

21. Trade-offs

- **Push to multiple drivers vs sequential:** Push parallel is faster.
- **Sync vs async payment:** Async releases driver faster.

22. CAP Theorem Analysis

CP for order state. AP for driver location (slight staleness OK).

23. Security Considerations

TLS. PCI compliance. PII encryption for addresses.

24. Monitoring & Logging

Track order latency, driver assignment time, location update lag.

25. Deployment Strategy

Blue-green. Pre-scale for meal times (lunch/dinner).

26. Interview Tips

- Use Redis GEO for driver locations (same as Uber).
- Discuss order state machine explicitly.
- Mention WebSocket for real-time tracking.

27. Common Follow-up Questions

- How would you implement multi-restaurant orders?
- How would you predict ETA accurately?
- How would you handle driver cancellations mid-delivery?

28. Common Mistakes

- PG for driver locations — too slow for radius queries.
- No real-time tracking — poor UX.
- Synchronous payment — blocks order completion.

29. Optimization Ideas

- Pre-assign drivers based on predicted demand.
- Batch driver location updates.
- Predict ETA using ML on historical data.

30. Final Summary

Food delivery extends Uber's geospatial pattern with a multi-party state machine (user, restaurant, driver). The core architecture: Redis GEO for driver locations, sharded Postgres for order state, WebSocket for real-time tracking. Driver assignment uses parallel dispatch with first-accept-wins atomicity. Pre-scaling for meal times (lunch/dinner peaks) is operationally important.

QUESTION 32 INTERMEDIATE

Design a Ride-Sharing Carpool Service (UberPool)

1. Problem Statement

Design a carpool service where multiple riders share a single ride. The system matches riders going in similar directions, optimizes the route, and splits fare.

2. Functional Requirements

- Rider requests carpool with pickup + dropoff.
- System finds nearby carpool-eligible rides.
- Driver accepts multiple pickups/dropoffs en route.
- Route optimization (insert new rider with minimal detour).
- Fare split among riders based on distance.
- Real-time ETA updates.

3. Non-functional Requirements

- Match latency < 10s.
- Route recalculation < 2s on new pickup.
- Availability 99.95%.

4. Capacity Estimation

5M carpool rides/day. Driver location updates: 100K active drivers × 1/5s = 20K/s.

5. Back-of-the-envelope Math

Matching: find rides whose route passes near rider's pickup + dropoff with < 10 min detour.

6. API Design

```
POST /carpools body: {pickup, dropoff} -> {carpool_id, status: SEARCHING}
POST /carpools/{id}/match body: {driver_id} (driver accepts)
POST /carpools/{id}/riders body: {rider_id, pickup, dropoff} (add rider mid-trip)
GET /carpools/{id} -> {riders, route, ETA, fare_split}
```

7. Database Schema

```
Table: carpools (carpool_id, driver_id, status, route_geojson, started_at, completed_at)
Table: pool_riders (carpool_id, rider_id, pickup_lat, pickup_lng, dropoff_lat,
dropoff_lng,
                    pickup_time, dropoff_time, fare_cents, status)
Table: drivers (driver_id, current_lat, current_lng, carpool_capacity, status)
```

8. High-Level Design (HLD)

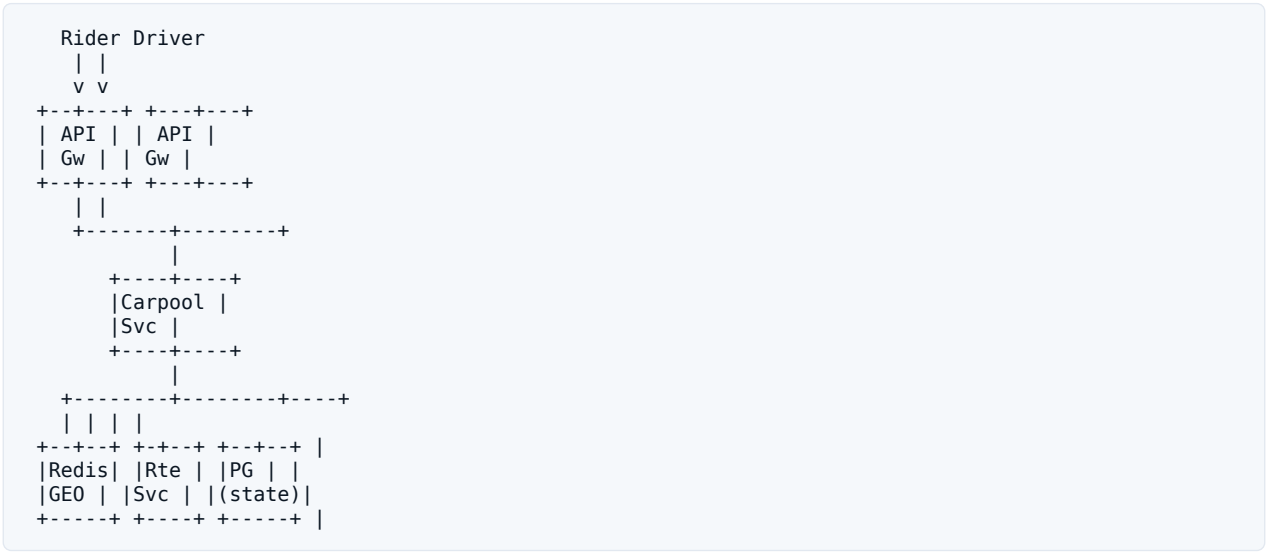
Mobile → API Gateway → Carpool Service (matching) + Driver Location Service (Redis GEO) + Route Service (OSRM/GraphHopper) + WebSocket Hub.

9. Low-Level Design (LLD)

Match algorithm: query Redis GEO for active carpools within 2km of rider pickup. For each candidate, check if route passes near rider dropoff with < 10min detour (call Route Service). Insert rider, recalculate route and fares.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Match:
  1. Rider POST /carpools with pickup + dropoff.
  2. Carpool Service: query Redis GEO for active carpools within 2km.
  3. For each candidate: call Route Service to compute detour if rider added.
  4. Filter candidates with detour < 10 min and capacity available.
  5. Pick best (lowest detour or highest fare efficiency).
  6. Notify driver and rider; on driver accept: insert rider, recalc route + fares.
  7. WebSocket push new ETA to all riders.
```

12. Component Explanation

- **Carpool Service:** Matching + state machine.
- **Route Service:** OSRM/GraphHopper for ETA and detour calc.
- **Driver Location Service:** Redis GEO.
- **WebSocket Hub:** Real-time updates to all parties.

13. Technology Stack

- **Language:** Python/Go.
- **DB:** PostgreSQL sharded by city.
- **Geo:** Redis GEO.
- **Routing:** OSRM (self-hosted) or Google Maps API.

- **Realtime:** WebSocket.

14. Database Choice

Postgres for carpool state. Redis GEO for driver locations.

15. Cache Strategy

Redis GEO for driver locations. Redis cache for hot routes.

16. Load Balancer Strategy

L7 LB with geographic routing.

17. Message Queue Usage

Kafka for carpool events.

18. Scaling Strategy

Per-city sharding. Route Service scales with match rate.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Route computation latency — cache common routes.

21. Trade-offs

- **Detour threshold:** Lower = better UX but fewer matches; higher = more matches but worse UX.
- **Pre-compute vs live route:** Pre-compute is faster; live is accurate.

22. CAP Theorem Analysis

CP for carpool state. AP for driver location.

23. Security Considerations

TLS. PII encryption.

24. Monitoring & Logging

Track match latency, route recalc time, fare accuracy.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Discuss detour calculation explicitly.

- Mention route caching for performance.
- Discuss dynamic fare split.

27. Common Follow-up Questions

- How would you handle rider no-shows?
- How would you optimize for max fare efficiency vs rider convenience?
- How would you handle multi-stop carpools (5+ riders)?

28. Common Mistakes

- No detour calculation — riders added with huge detours.
- No fare split logic.
- Synchronous route computation — slow match.

29. Optimization Ideas

- Pre-compute popular routes.
- Batch route queries to OSRM.
- Predict demand for pre-positioning drivers.

30. Final Summary

Carpool extends ride-sharing with multi-rider matching and route optimization. The core challenge is finding carpools whose route passes near the new rider's pickup and dropoff with acceptable detour (< 10 min). Route Service (OSRM) is critical for ETA and detour calculation. The architecture is per-city sharded, with Redis GEO for driver locations and WebSocket for real-time updates.

QUESTION 33 INTERMEDIATE

Design an Online Banking System

1. Problem Statement

Design a digital banking platform. Users view account balances, transfer money, pay bills, deposit checks via photo, and view transaction history. Strong consistency and audit trails are mandatory.

2. Functional Requirements

- View account balances and transaction history.
- Transfer money between accounts (internal + external).
- Pay bills to registered billers.
- Deposit checks via mobile photo.
- ATM locator and card management.
- Statements and tax documents.

3. Non-functional Requirements

- Availability 99.99%.
- Strong consistency: no double-spend, no lost transfer.
- Audit trail for all transactions (compliance).
- Security: 2FA, fraud detection.

4. Capacity Estimation

10M users, 100M transactions/day. Storage modest. Consistency and security dominate.

5. Back-of-the-envelope Math

Compute trivial. Transaction atomicity is the core challenge. Sharded by user_id.

6. API Design

```
GET /accounts -> [accounts with balances]
GET /accounts/{id}/transactions?from=...&to=...
POST /transfers body: {from_account, to_account, amount}
POST /bills/pay body: {biller_id, amount, account_id}
POST /deposits/check body: {image_url, amount, account_id}
POST /cards/{id}/freeze
```

7. Database Schema

```
Table: accounts (account_id, user_id, type, balance_cents, available_cents, currency)
Table: transactions (txn_id, from_account, to_account, amount, type, status,
                     reference, timestamp, balance_after)
Table: transfers (transfer_id, txn_id, status, initiated_at, completed_at)
Table: audit_log (log_id, user_id, action, ip, timestamp, metadata)
```

8. High-Level Design (HLD)

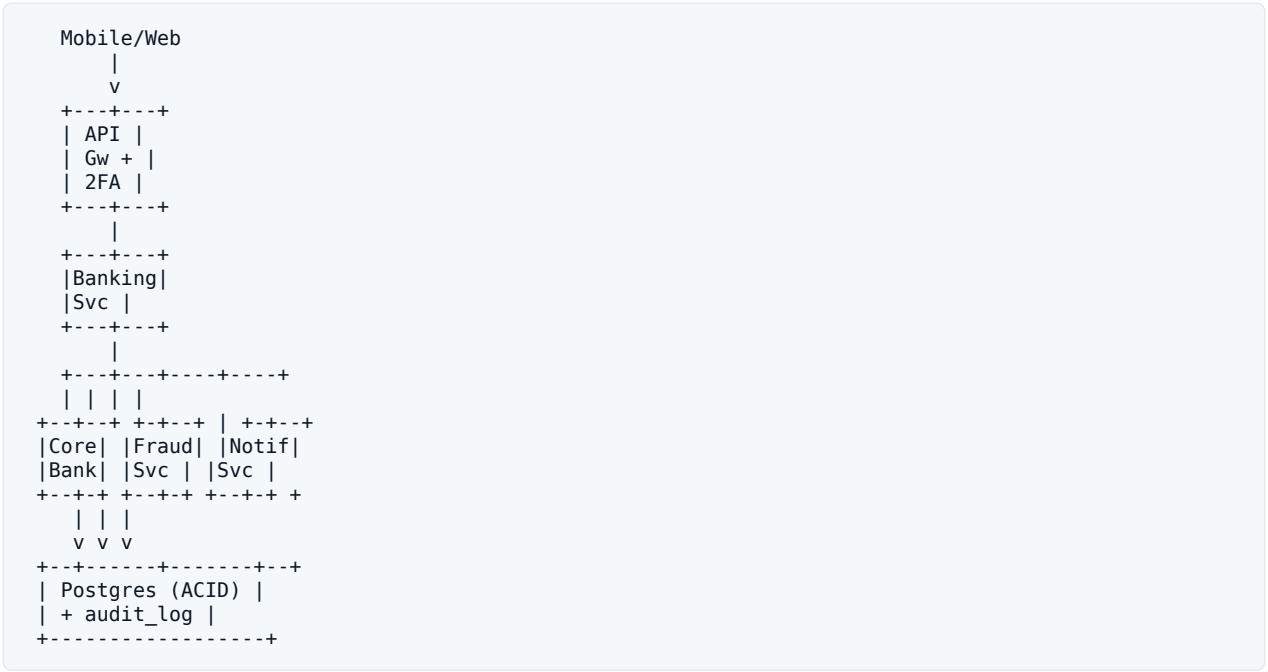
Mobile/Web → API Gateway (with strong auth) → Banking Service → Core Banking (sharded Postgres, ACID) + Fraud Detection Service + Notification Service.

9. Low-Level Design (LLD)

Transfer is a 2-phase transaction: BEGIN; SELECT from_account FOR UPDATE; verify balance; SELECT to_account FOR UPDATE; debit + credit; INSERT transactions; COMMIT. Audit log entry written in same transaction.

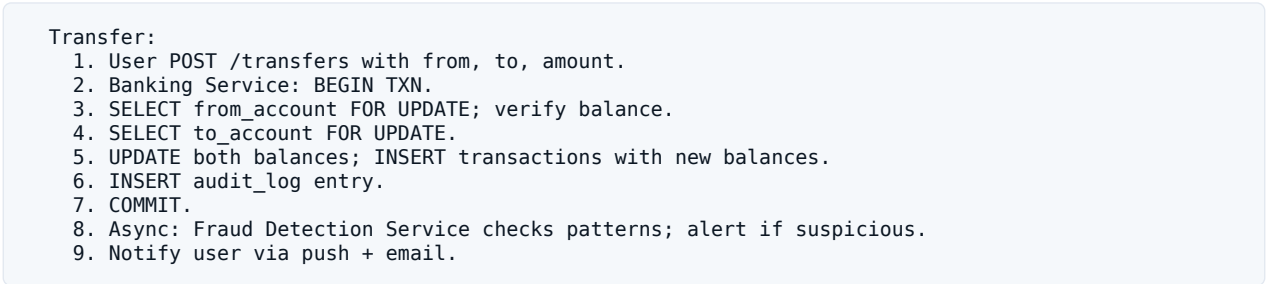
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Banking Service:** Transactional logic; account management.
- **Core Banking DB:** ACID Postgres with audit log.
- **Fraud Detection:** ML model on transaction patterns.
- **Notification Service:** Push, email, SMS for alerts.
- **Audit Service:** Immutable log of all actions.

13. Technology Stack

- **Language:** Java (enterprise-grade).
- **DB:** PostgreSQL with strong ACID, sharded by user_id.
- **Cache:** Redis (read-aside for balances, invalidated on txn).
- **Queue:** Kafka (audit, fraud analysis).
- **Deployment:** Multi-AZ, on-prem or regulated cloud.

14. Database Choice

Postgres for ACID. No NoSQL — banking is fundamentally transactional.

15. Cache Strategy

Redis for read-aside balance display. Invalidate on every transaction. Cache is for UX speed, not source of truth.

16. Load Balancer Strategy

L7 LB. No sticky session needed.

17. Message Queue Usage

Kafka for audit events, fraud analysis, notifications.

18. Scaling Strategy

Shard by user_id. Read replicas for transaction history queries.

19. Fault Tolerance

Multi-AZ with synchronous replication. Active-passive DR site.

20. Bottlenecks

- Hot accounts (high transaction rate) — rate limit + queue.
- Audit log write throughput — batch in same txn.

21. Trade-offs

- **Sync vs async replication:** Sync is safe; async risks data loss.
- **Cache vs no cache:** Cache speeds reads; invalidation adds complexity.

22. CAP Theorem Analysis

CP. Strong consistency mandatory. Reject transactions during partition.

23. Security Considerations

TLS. 2FA mandatory. Encryption at rest (AES-256). PCI compliance. Audit log tamper-proof (append-only, signed). Fraud detection on every txn.

24. Monitoring & Logging

Track transaction latency, conflict rate, fraud alert rate, replication lag.

25. Deployment Strategy

Blue-green. Strict change management. Audit log immutable.

26. Interview Tips

- Emphasize ACID transactions — non-negotiable.
- Discuss audit log as compliance requirement.
- Mention 2FA and fraud detection.
- Discuss SELECT FOR UPDATE for atomic transfers.

27. Common Follow-up Questions

- How would you handle cross-bank transfers (ACH, SWIFT)?
- How would you implement check deposit via photo (OCR)?
- How would you detect sophisticated fraud (account takeover)?
- How would you handle a bank run (mass withdrawals)?

28. Common Mistakes

- No transaction on transfer — double-spend or lost money.
- No audit log — compliance failure.
- Cache as source of truth — incorrect balances.
- No 2FA — security vulnerability.

29. Optimization Ideas

- Batch audit log writes (single INSERT per txn).
- Pre-compute daily balances for fast history queries.
- Async fraud detection (don't block txn).

30. Final Summary

Online banking tests transactional integrity and security at the highest level. The core challenge is atomic money transfer: SELECT FOR UPDATE on both accounts, debit + credit in same transaction, with audit log entry written atomically. Strong consistency (CP) is non-negotiable — reject transactions during network partition rather than risk inconsistency. Audit log is compliance-mandated. The architecture rewards conservative choices: ACID Postgres, no cache as source of truth, 2FA, fraud detection.

QUESTION 34

INTERMEDIATE

Design LinkedIn

1. Problem Statement

Design a professional networking platform. Users create profiles, connect with others, post updates, search jobs, and apply. Recruiter portal for sourcing.

2. Functional Requirements

- User profiles (experience, education, skills).
- Connect with other users (1st, 2nd, 3rd degree).
- Feed of posts from connections.
- Job search and application.
- Messaging (InMail).
- Recruiter portal: search candidates, send messages.

3. Non-functional Requirements

- Availability 99.95%.
- Search latency < 500ms.
- Feed generation < 1s.
- Read-heavy (50:1 read:write).

4. Capacity Estimation

800M users, 50M jobs, 5M recruiters. Storage: profiles ~5TB; graph (connections) ~10TB; feed cache large.

5. Back-of-the-envelope Math

Graph of 800M nodes with billions of edges — use Neo4j or specialized graph service. Feed: push-pull hybrid.

6. API Design

```
GET /profiles/{id}
POST /connections body: {to_user_id}
GET /feed?cursor=...
GET /jobs/search?q=...&location=...&company=...
POST /jobs/{id}/apply body: {resume_url, cover_letter}
GET /recruiter/search?skills=...&title=...
```

7. Database Schema

```
Table: users (user_id, name, headline, location, ...)
Table: experiences (exp_id, user_id, title, company, start, end)
Table: connections (user_id_a, user_id_b, status: PENDING|ACCEPTED, created_at)
Table: posts (post_id, author_id, content, created_at)
Table: jobs (job_id, company_id, title, location, ...)
Table: applications (app_id, job_id, user_id, applied_at, status)
```

8. High-Level Design (HLD)

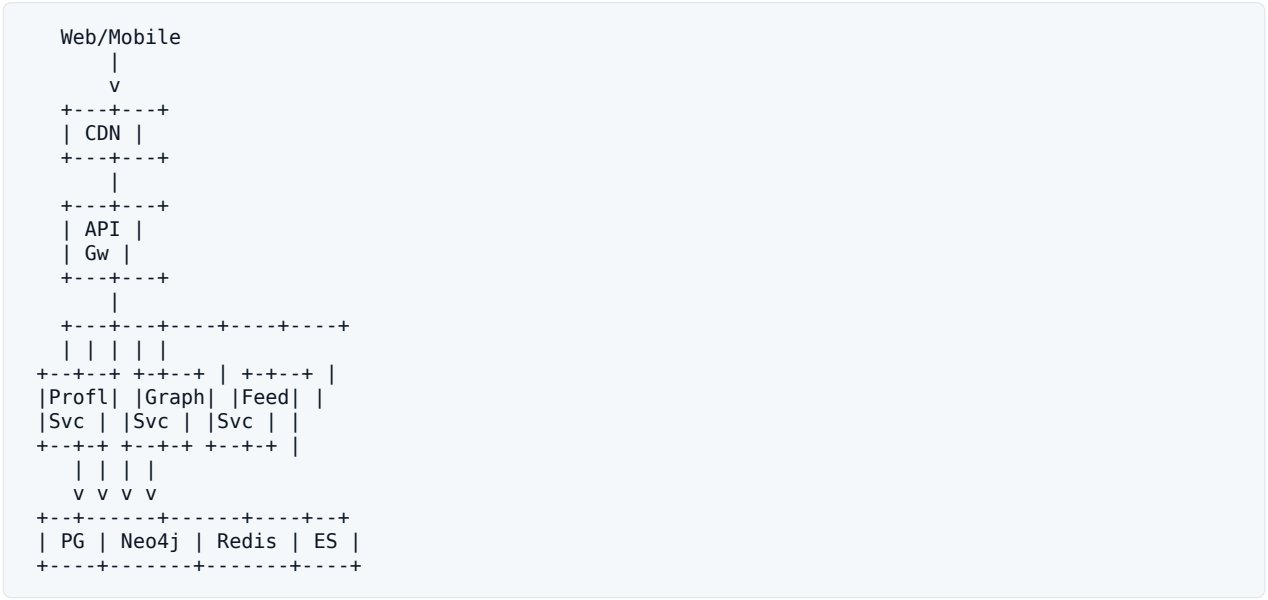
Web/Mobile → CDN → API Gateway → Profile Service + Graph Service (connections) + Feed Service + Job Service + Recruiter Service. ES for search.

9. Low-Level Design (LLD)

Connection graph: stored in Neo4j or Postgres with recursive CTEs for degree-of-separation queries. Feed: push-pull hybrid (push for normal users, pull for "influencers").

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Connection request:
  1. User A sends request to B.
  2. Graph Service: INSERT connections (status=PENDING).
  3. Notify B via push/email.
  4. B accepts: UPDATE status=ACCEPTED.
  5. Update degree-of-separation cache for A and B's networks.

Job search:
  1. User searches by title + location.
  2. Job Service queries ES for matching jobs.
  3. Rank by relevance + recency.
  4. Return paginated results.
```

12. Component Explanation

- **Profile Service:** User profile CRUD.

- **Graph Service:** Connections; degree-of-separation.
- **Feed Service:** Hybrid push-pull feed.
- **Job Service:** Job search and applications.
- **Recruiter Service:** Candidate search; InMail.
- **Neo4j:** Connection graph for fast traversals.
- **Elasticsearch:** Job and profile search.

13. Technology Stack

- **Language:** Java/Scala.
- **DB:** PostgreSQL + Neo4j for graph.
- **Cache:** Redis.
- **Search:** Elasticsearch.
- **Queue:** Kafka.

14. Database Choice

Postgres for profile data (ACID). Neo4j for connection graph (fast traversals). Redis for feed. ES for search.

15. Cache Strategy

Redis for hot profiles, feed candidates, recruiter search results.

16. Load Balancer Strategy

L7 LB. Stateless services.

17. Message Queue Usage

Kafka for connection events, job applications.

18. Scaling Strategy

Shard PG by user_id. Neo4j cluster. ES scales search.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Graph traversal for 2nd/3rd degree — cache aggressively.
- Feed fanout for influencers — pull model.

21. Trade-offs

- **Neo4j vs PG recursive CTE:** Neo4j is faster for deep traversals; PG is simpler.
- **Push vs pull feed:** Hybrid — push normal, pull influencers.

22. CAP Theorem Analysis

AP for feed. CP for connections (no duplicate accept).

23. Security Considerations

TLS. Auth via OAuth. PII encryption. Recruiter access audited.

24. Monitoring & Logging

Track search latency, feed latency, graph traversal time.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Discuss graph storage (Neo4j) explicitly.
- Mention degree-of-separation caching.
- Discuss recruiter search as a separate use case.

27. Common Follow-up Questions

- How would you implement "People you may know" recommendations?
- How would you handle job recommendations?
- How would you implement LinkedIn Learning (video courses)?

28. Common Mistakes

- PG-only for graph — slow at scale.
- No degree-of-separation cache.
- No recruiter portal as separate use case.

29. Optimization Ideas

- Cache 2nd-degree connections per user (refresh daily).
- Pre-compute PYMK recommendations.
- Edge-cache job search results.

30. Final Summary

LinkedIn tests graph data (connections) at scale alongside standard feed/search architecture. The connection graph (800M nodes, billions of edges) benefits from a specialized graph database (Neo4j) for fast degree-of-separation queries. The feed uses the standard push-pull hybrid. Job and recruiter search use Elasticsearch. The architecture rewards graph-aware design: caching 2nd-degree connections, pre-computing recommendations.

QUESTION 35 INTERMEDIATE

Design Pinterest

1. Problem Statement

Design a visual bookmarking platform. Users "pin" images to boards, follow boards, see a feed of related pins, and discover content via recommendations.

2. Functional Requirements

- Create boards and pin images (with link + description).
- Follow boards and users.
- Home feed of pins from followed + recommended.
- Search by keyword, color, visual similarity.
- Save (repin) others' pins.

3. Non-functional Requirements

- Availability 99.95%.
- Feed latency < 1s.
- Image upload + processing < 5s.
- Recommendations refresh hourly.

4. Capacity Estimation

450M users, 200B pins. Storage: 200B × 100KB image = 20PB. Feed reads: 1B/day.

5. Back-of-the-envelope Math

Visual search requires image embeddings (CNN). Pin graph: pin → board → user. Recommendations via graph-based + ML.

6. API Design

```
POST /pins body: {image_url, link, description, board_id}
GET /pins/{id}
POST /boards body: {name, category}
GET /feed?cursor=...
GET /search?q=...&type=visual|text
POST /pins/{id}/repin body: {target_board_id}
```

7. Database Schema

```
Table: pins (pin_id, creator_id, image_url, link, description, board_id, created_at)
Table: boards (board_id, owner_id, name, category, follower_count)
Table: follows (follower_id, followee_id, type: USER|BOARD, created_at)
Table: repins (repin_id, original_pin_id, target_board_id, repinner_id, created_at)
Table: pin_embeddings (pin_id, embedding_vector) -- for visual search
```

8. High-Level Design (HLD)

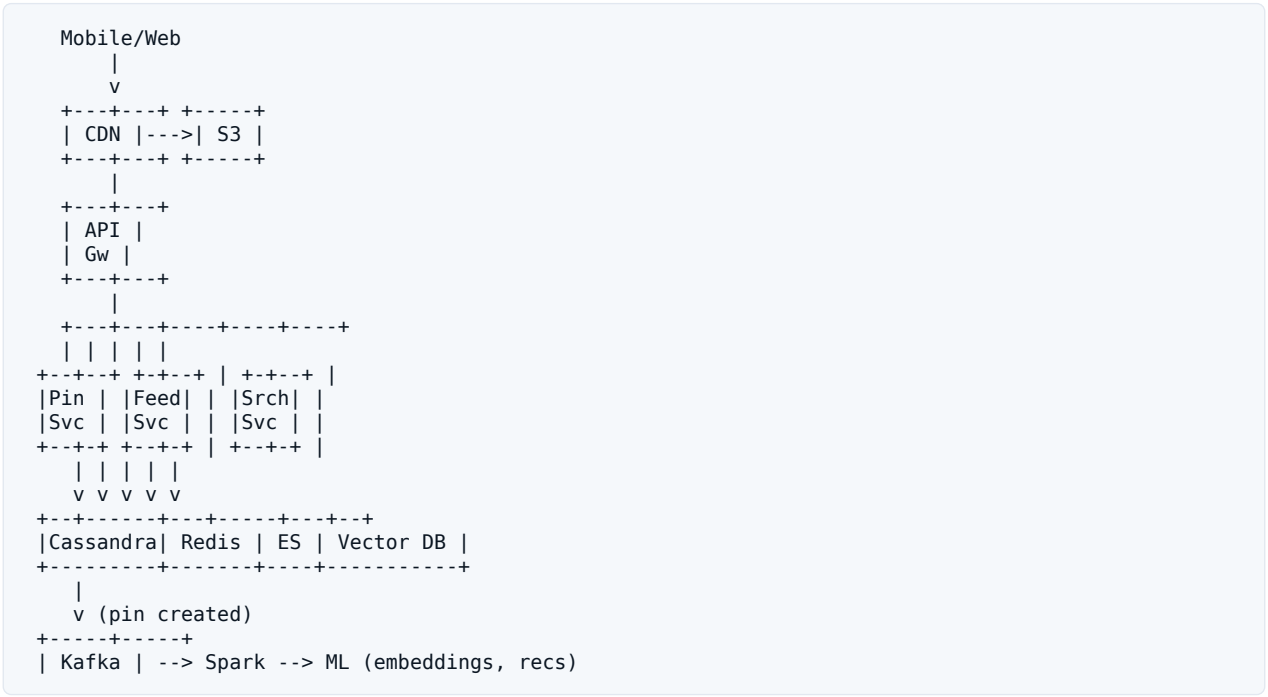
Mobile/Web → CDN → API Gateway → Pin Service + Feed Service + Search Service + Recommendation Service. S3 for images. ES for text search. Vector DB for visual search.

9. Low-Level Design (LLD)

On pin upload: process image (resize, generate thumbnails, compute CNN embedding), store to S3, write metadata + embedding. Feed: candidate set from followed + ML-recommended.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Pin upload:
  1. User uploads image to S3 via presigned URL.
  2. POST /pins with image_url + metadata.
  3. Pin Service writes to Cassandra; publishes event to Kafka.
  4. Image processing worker: resize, thumbnails, CNN embedding.
  5. Store embedding in vector DB (for visual search).
  6. Index text in ES.
  7. Fanout to followers' feed lists.
```

12. Component Explanation

- **Pin Service:** CRUD for pins.
- **Feed Service:** Read feed; merge followed + recommended.
- **Search Service:** Text search (ES) + visual search (vector DB).
- **Recommendation Service:** ML-based pin recommendations.
- **Image Processing:** Resize, thumbnails, CNN embeddings.
- **Vector DB:** Faiss/Milvus for visual similarity search.
- **S3 + CDN:** Image storage and delivery.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** Cassandra (pins), Redis (feed).
- **Search:** Elasticsearch + Faiss/Milvus (vector).
- **ML:** PyTorch (CNN embeddings).
- **Storage:** S3 + CloudFront.
- **Queue:** Kafka.

14. Database Choice

Cassandra for pins (write-heavy). Redis for feed. ES + vector DB for search.

15. Cache Strategy

Redis for feed candidates, hot pins. CDN for images.

16. Load Balancer Strategy

L7 LB. Stateless.

17. Message Queue Usage

Kafka for pin events, image processing, ML pipeline.

18. Scaling Strategy

Cassandra scales by pin volume. Vector DB scales with embedding count. CDN for images.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Visual search latency — approximate nearest neighbor (ANN).
- Image processing pipeline — auto-scale workers.

21. Trade-offs

- **Vector DB vs ES kNN:** Vector DB is faster; ES is simpler.
- **Push vs pull feed:** Hybrid — push followed, pull recommended.

22. CAP Theorem Analysis

AP for feed. CP for pins.

23. Security Considerations

TLS. Image moderation (NSFW). Copyright detection.

24. Monitoring & Logging

Track feed latency, search latency, image processing time, ML training.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Discuss visual search with CNN embeddings — differentiator.
- Mention vector DB (Faiss/Milvus) for ANN search.
- Discuss ML recommendation pipeline.

27. Common Follow-up Questions

- How would you implement "More like this" on a pin?
- How would you detect copyright images?
- How would you implement shopping pins (buyable pins)?

28. Common Mistakes

- No visual search — misses Pinterest's core feature.
- No image processing pipeline — slow uploads.
- Storing images in DB — kills performance.

29. Optimization Ideas

- Pre-compute embeddings on upload (not on search).
- Use ANN (HNSW) for fast vector search.
- Edge-cache popular pins.

30. Final Summary

Pinterest tests visual discovery at scale. The differentiator is visual search: CNN embeddings stored in a vector DB (Faiss/Milvus) enable "more like this" and image-based search. The standard feed (push-pull hybrid) and text search (ES) are familiar from Instagram/Twitter. Image processing pipeline (resize, thumbnails, embeddings) runs async via Kafka. The architecture rewards ML investment.

QUESTION 36 INTERMEDIATE

Design Reddit

1. Problem Statement

Design a social news platform. Users submit links/text to subreddits, upvote/downvote, comment threaded, and see a ranked front page.

2. Functional Requirements

- Submit posts (link/text) to subreddits.
- Upvote/downvote posts and comments.
- Threaded comments.
- Front page (subscribed subreddits) + /r/all.
- Sort by hot/new/top/controversial.

3. Non-functional Requirements

- Availability 99.95%.
- Front page latency < 1s.
- Vote latency < 500ms.
- Eventual consistency acceptable (10s lag).

4. Capacity Estimation

500M users, 100K subreddits, 1B posts, 5B comments. Vote events: 100M/day. Storage modest.

5. Back-of-the-envelope Math

Vote count is the hot path — aggregate in Redis, flush to PG hourly. Front page ranking uses "hotness" algorithm (votes + recency).

6. API Design

```
POST /posts body: {subreddit_id, type, title, content}
POST /posts/{id}/vote body: {direction: UP|DOWN}
POST /posts/{id}/comments body: {text, parent_comment_id?}
GET /r/{subreddit}?sort=hot|new|top|controversial
GET /front -> subscribed subreddits' hot posts
```

7. Database Schema

```
Table: posts (post_id, author_id, subreddit_id, type, title, content, score, created_at)
Table: comments (comment_id, post_id, parent_comment_id, author_id, text, score,
created_at)
Table: votes (post_or_comment_id, user_id, direction) PK (id, user_id)
Table: subreddits (subreddit_id, name, subscriber_count)
Table: subscriptions (user_id, subreddit_id)
```

8. High-Level Design (HLD)

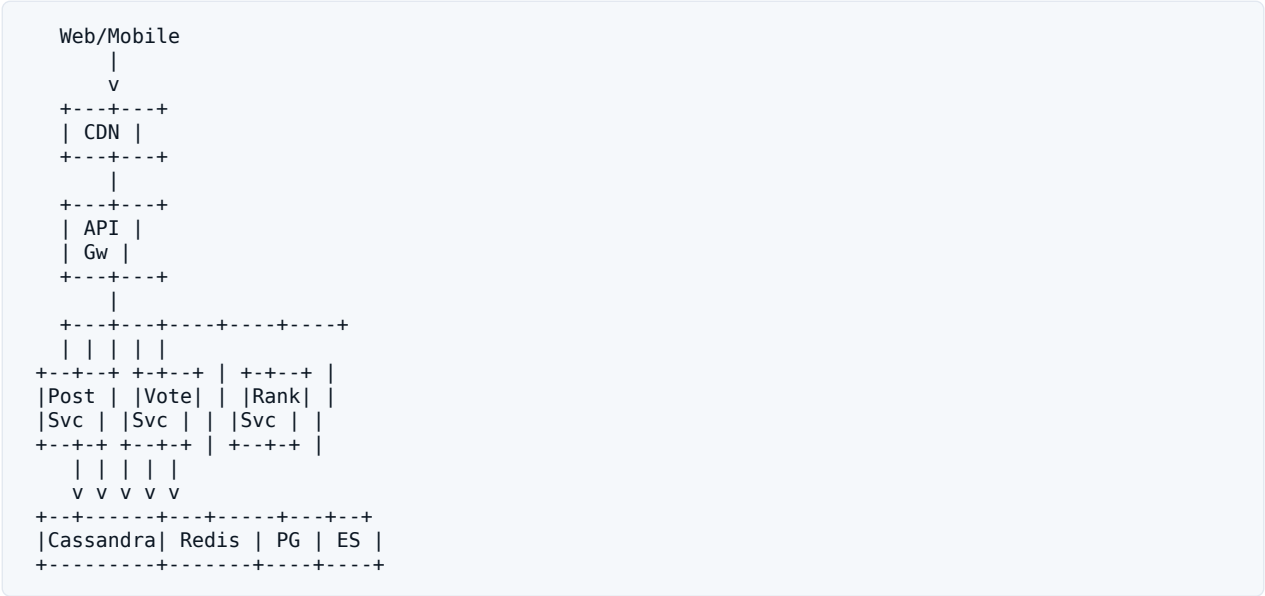
Web/Mobile → CDN → API Gateway → Post Service + Vote Service + Comment Service + Ranking Service. Redis for vote counters. PG for posts/comments.

9. Low-Level Design (LLD)

Vote: increment/decrement counter in Redis (atomic INCR), persist vote record in PG. Hourly batch flushes aggregated scores to PG. Ranking: hotness algorithm combining score + recency; recompute every minute for active subreddits.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Vote:
  1. User clicks upvote. POST /posts/{id}/vote.
  2. Vote Service: BEGIN TXN; check no existing vote; INSERT vote.
  3. Atomic INCR post:score in Redis (immediate visibility).
  4. COMMIT. Invalidate rank cache for post's subreddit.
  5. Async: hourly batch flushes Redis scores to PG.
```

12. Component Explanation

- **Post Service:** CRUD for posts.
- **Vote Service:** Transactional vote logic.
- **Comment Service:** Threaded comments.
- **Ranking Service:** Hotness algorithm; per-subreddit rankings.
- **Redis:** Vote counters (live), rank caches.
- **Cassandra:** Posts and comments (write-heavy).

13. Technology Stack

- **Language:** Python/Go.

- **DB:** Cassandra (posts, comments), PG (users, votes).
- **Cache:** Redis Cluster.
- **Search:** Elasticsearch.
- **Queue:** Kafka.

14. Database Choice

Cassandra for posts/comments (write-heavy). PG for votes (ACID). Redis for counters (fast INCR).

15. Cache Strategy

Redis for vote counters (live), subreddit rankings (refreshed every minute).

16. Load Balancer Strategy

L7 LB. Stateless.

17. Message Queue Usage

Kafka for vote events (batch flush), post events (fanout).

18. Scaling Strategy

Cassandra scales by post volume. Redis Cluster for counters.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Vote counter contention on hot posts — Redis INCR is atomic, fast.
- Rank computation for /r/all — pre-compute every minute.

21. Trade-offs

- **Redis vs PG for counters:** Redis is fast; PG is durable. Hybrid: Redis live + PG hourly.
- **Batch vs real-time rank:** Batch is cheaper; real-time is fresher.

22. CAP Theorem Analysis

AP for rankings (eventual consistency). CP for votes (no double-vote).

23. Security Considerations

TLS. Auth via OAuth. Anti-vote-manipulation (subreddit quarantine). Content moderation.

24. Monitoring & Logging

Track vote latency, rank computation time, hot post contention.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Use Redis for vote counters — atomic INCR is perfect.
- Discuss hotness ranking algorithm (log(score) + time_decay).
- Mention batch flush from Redis to PG for durability.

27. Common Follow-up Questions

- How would you implement nested comments efficiently?
- How would you detect vote manipulation?
- How would you implement "trending" subreddits?

28. Common Mistakes

- PG-only for vote counters — too slow.
- No rank caching — every page load recomputes.
- Synchronous rank update on every vote — kills throughput.

29. Optimization Ideas

- Pre-compute top 100 per subreddit every minute.
- Batch Redis INCRs (pipeline).
- Use CDN for subreddit pages (cache 30s).

30. Final Summary

Reddit tests ranking and vote aggregation. The core pattern is Redis atomic INCR for vote counters (fast, immediately visible) with hourly batch flush to PG (durable). Ranking uses a hotness algorithm (log(score) + time decay) computed every minute per subreddit. Cassandra handles post/comment writes. The architecture rewards careful counter design and aggressive ranking cache.

QUESTION 37

INTERMEDIATE

Design Quora

1. Problem Statement

Design a Q&A; platform. Users ask questions, answer, follow topics and questions, and see a feed of related content. Similar to Stack Overflow but with social features.

2. Functional Requirements

- Ask questions (with topics).
- Answer questions; upvote/downvote answers.
- Follow topics, questions, users.
- Personalized feed of questions/answers.
- Search by topic, keyword.

3. Non-functional Requirements

- Availability 99.95%.
- Feed latency < 1s.
- Search latency < 500ms.
- Read-heavy (50:1).

4. Capacity Estimation

300M users, 100M questions, 400M answers. Storage modest. Feed reads: 200M/day.

5. Back-of-the-envelope Math

Standard feed architecture (push-pull hybrid). ES for search. PG for Q&A.;

6. API Design

```
POST /questions body: {title, body, topics: [...]}
POST /questions/{id}/answers body: {text}
POST /answers/{id}/vote body: {direction}
GET /feed?cursor=...
GET /search?q=...
```

7. Database Schema

```
Table: questions (question_id, author_id, title, body, topics, created_at)
Table: answers (answer_id, question_id, author_id, text, score, created_at)
Table: topics (topic_id, name, follower_count)
Table: follows (user_id, target_id, target_type: TOPIC|QUESTION|USER)
```

8. High-Level Design (HLD)

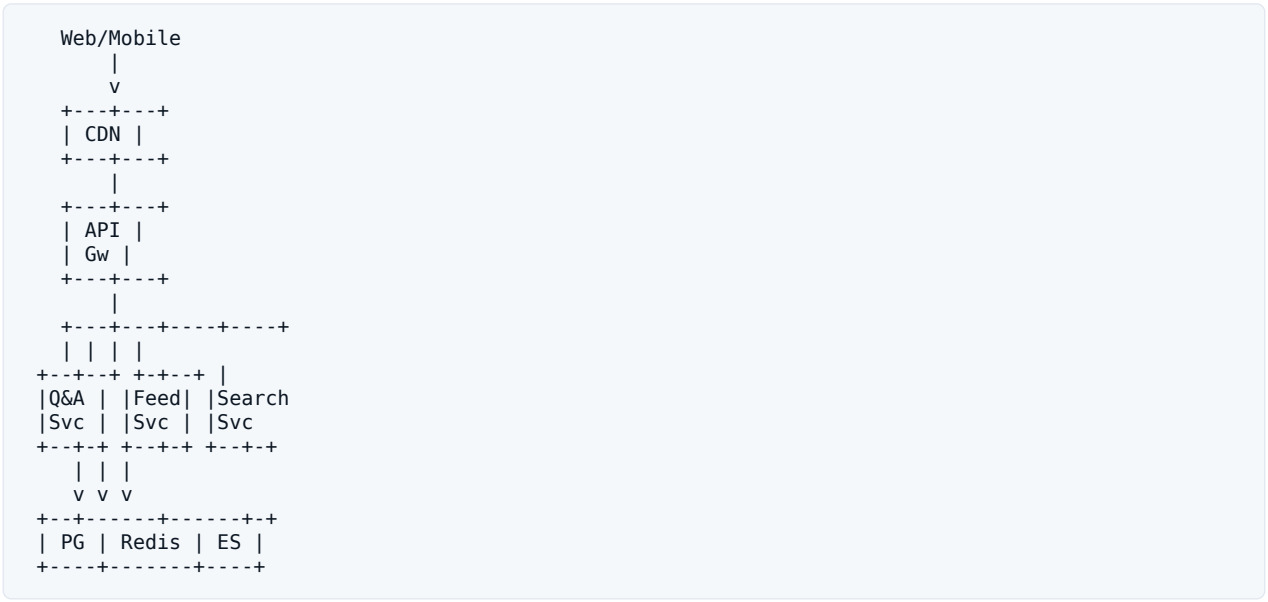
Web/Mobile → CDN → API Gateway → Q&A; Service + Feed Service + Search Service. ES for search. Redis for feed candidates.

9. Low-Level Design (LLD)

Standard feed architecture. Feed merges: questions from followed topics, answers from followed users, recommended questions.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Ask question:
  1. POST /questions; Q&A Service writes to PG.
  2. Publish event to Kafka.
  3. Fanout to followers of question's topics.
  4. Index in ES for search.

Feed read:
  1. GET /feed from Feed Service.
  2. Pull candidates from Redis (followed topics, users).
  3. Merge with recommended questions (ML).
  4. Rank and return.
```

12. Component Explanation

- **Q&A; Service:** CRUD for questions/answers.
- **Feed Service:** Personalized feed.
- **Search Service:** Elasticsearch.
- **Redis:** Feed candidates, vote counters.
- **Postgres:** Questions, answers, follows.

13. Technology Stack

- **Language:** Python/Java.

- **DB:** PostgreSQL.
- **Cache:** Redis.
- **Search:** Elasticsearch.
- **Queue:** Kafka.

14. Database Choice

Postgres for ACID (votes, questions). Redis for feed. ES for search.

15. Cache Strategy

Redis for hot questions, feed candidates, vote counters.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for question/answer events.

18. Scaling Strategy

Shard PG by question_id (for answers) or user_id.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Feed fanout for popular topics — pull model.

21. Trade-offs

- **Push vs pull:** Hybrid.

22. CAP Theorem Analysis

AP for feed. CP for votes.

23. Security Considerations

TLS. Auth. Spam detection. Content moderation.

24. Monitoring & Logging

Track feed latency, search latency, vote conflict rate.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Standard feed + search architecture.

27. Common Follow-up Questions

- How would you implement "ask for translation"?

28. Common Mistakes

- No search index — PG too slow.

29. Optimization Ideas

- Pre-compute top answers per question.

30. Final Summary

Quora is a standard feed + search + CRUD architecture. The interesting features are topic-based following (multi-dimensional graph) and the combination of question feed + answer feed + recommendation feed.

QUESTION 38 INTERMEDIATE

Design Tinder

1. Problem Statement

Design a dating app. Users swipe through candidate profiles (geographically nearby), mutual right-swipes enable messaging. Real-time chat after match.

2. Functional Requirements

- Profile creation (photos, bio, preferences).
- Swipe through nearby candidates (right=yes, left=no).
- Mutual right-swipe = match; enables chat.
- Real-time chat with matches.
- Geolocation-based recommendations.

3. Non-functional Requirements

- Candidate recommendation < 1s.
- Match notification < 5s.
- Chat latency < 1s.
- Availability 99.5%.

4. Capacity Estimation

100M users, 10M DAU, 1B swipes/day. Storage: 100M × 1MB photos = 100TB. Match rate ~5%.

5. Back-of-the-envelope Math

Geospatial query for candidates within X km, filtered by preferences, sorted by match score. Redis GEO + ML scoring.

6. API Design

```
GET /recommendations -> [nearby candidates]
POST /swipe body: {target_user_id, direction}
POST /matches/{id}/messages body: {text}
GET /matches -> [user's matches]
// WebSocket /matches/{id}/stream for real-time chat
```

7. Database Schema

```
Table: users (user_id, name, bio, gender, birthdate, lat, lng, preferences_json)
Table: swipes (swiper_id, target_id, direction, created_at) PK (swiper_id, target_id)
Table: matches (match_id, user_a, user_b, created_at)
Table: messages (msg_id, match_id, sender_id, text, created_at)
```

8. High-Level Design (HLD)

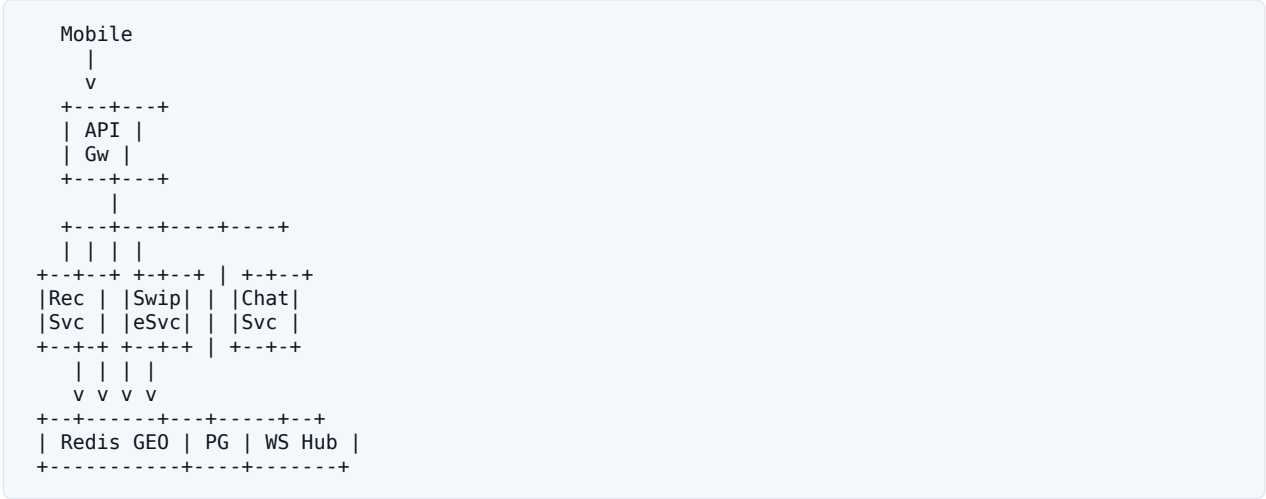
Mobile → API Gateway → Recommendation Service (geospatial + ML) + Swipe Service + Match Service + Chat Service (WebSocket).

9. Low-Level Design (LLD)

On swipe: record in PG. If both swiped right, create match (atomic check). Match triggers WebSocket push to both users + notification.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Swipe + match:
  1. User A swipes right on B. POST /swipe.
  2. Swipe Service: INSERT swipe record.
  3. Check if B also swiped right on A.
  4. If yes: INSERT match; trigger WebSocket push to both A and B.
  5. Notify both via push notification.
  6. Chat enabled.
```

12. Component Explanation

- **Recommendation Service:** Geospatial + ML candidate selection.
- **Swipe Service:** Records swipes; detects matches.
- **Match Service:** Manages matches.
- **Chat Service:** WebSocket-based messaging.
- **Redis GEO:** User locations for nearby queries.

13. Technology Stack

- **Language:** Node.js/Go.
- **DB:** PostgreSQL.
- **Geo:** Redis GEO.
- **Realtime:** WebSocket.
- **ML:** PyTorch (match scoring).

14. Database Choice

Postgres for users/swipes/matches (ACID). Redis GEO for locations.

15. Cache Strategy

Redis for user locations, hot profiles.

16. Load Balancer Strategy

L7 LB. Sticky session for WebSocket.

17. Message Queue Usage

Kafka for swipe events (analytics, ML training).

18. Scaling Strategy

Shard by user_id. Redis GEO scales by geocell.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Hot users (many right swipes) — rate limit.

21. Trade-offs

- **Sync vs async match check:** Sync is immediate; async is faster.

22. CAP Theorem Analysis

CP for swipes/matches. AP for recommendations.

23. Security Considerations

TLS. Photo moderation. Anti-catfish (photo verification).

24. Monitoring & Logging

Track recommendation latency, match detection latency, chat latency.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Use Redis GEO for nearby queries.
- Discuss match detection as atomic conditional INSERT.
- Mention ML scoring for recommendations.

27. Common Follow-up Questions

- How would you implement "Super Like" (one per day)?
- How would you detect fake profiles?
- How would you implement "Boost" (be top profile for 30 min)?

28. Common Mistakes

- PG for nearby queries — too slow.
- No match notification — poor UX.
- Synchronous photo upload on swipe path.

29. Optimization Ideas

- Pre-compute recommendation lists per user.
- Batch swipe writes.
- Cache hot user profiles.

30. Final Summary

Tinder tests geospatial recommendation + real-time matchmaking. The core flow: Redis GEO for nearby candidates, swipe records in PG, atomic match detection (both swiped right), WebSocket for chat. The architecture is straightforward; the differentiator is ML-based recommendation quality.

QUESTION 39 INTERMEDIATE

Design Airbnb (Lite)

1. Problem Statement

Design a marketplace for short-term home rentals. Hosts list properties; guests search by location/date, book, and pay. Reviews and messaging.

2. Functional Requirements

- Hosts list properties (location, photos, price, amenities).
- Guests search by location, dates, guests count.
- Book with payment; host accepts/declines.
- Reviews after stay.
- Messaging between host and guest.

3. Non-functional Requirements

- Availability 99.95%.
- Search latency < 500ms.
- Booking atomicity (no double-book).

4. Capacity Estimation

150M users, 5M listings, 1M bookings/day. Storage modest. Search QPS: 100K peak.

5. Back-of-the-envelope Math

Same as hotel booking: date-scoped inventory, atomic multi-night booking.

6. API Design

```
POST /listings body: {lat, lng, photos, price, amenities}
GET /search?location=...&checkin=...&checkout=...&guests=...
POST /bookings body: {listing_id, checkin, checkout, payment}
POST /bookings/{id}/cancel
POST /listings/{id}/reviews body: {rating, text}
```

7. Database Schema

```
Table: listings (listing_id, host_id, lat, lng, price, amenities, photos, ...)
Table: listing_inventory (listing_id, date, available, price) -- per-date
Table: bookings (booking_id, listing_id, guest_id, checkin, checkout, total, status)
Table: reviews (review_id, listing_id, author_id, rating, text, created_at)
```

8. High-Level Design (HLD)

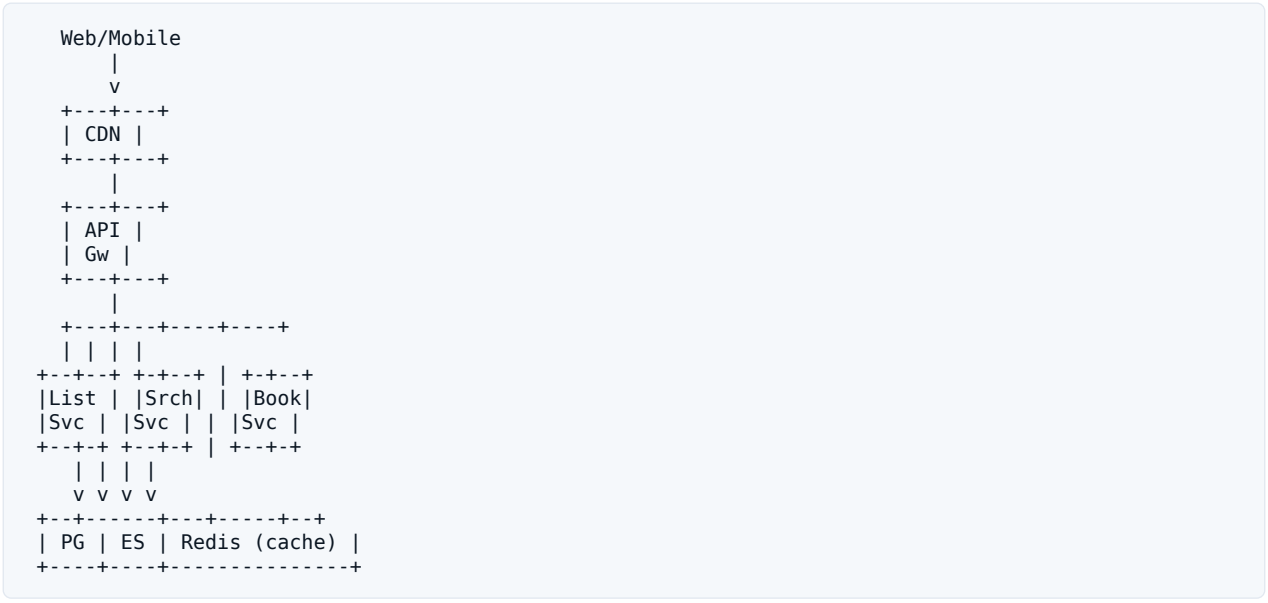
Web/Mobile → CDN → API Gateway → Listing Service + Search Service (ES) + Booking Service + Payment + Messaging. PG for bookings (ACID).

9. Low-Level Design (LLD)

Same as hotel booking: SELECT FOR UPDATE on listing_inventory for each night in [checkin, checkout). Atomic multi-night booking.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Booking:
  1. Guest POST /bookings with listing_id, dates.
  2. Booking Service: BEGIN TXN.
  3. SELECT listing_inventory FOR UPDATE for each night.
  4. Verify all available; decrement; INSERT booking.
  5. COMMIT. Initiate payment.
  6. On payment success: confirm; on failure: rollback inventory.
```

12. Component Explanation

- **Listing Service:** CRUD for listings.
- **Search Service:** Elasticsearch (location, amenities).
- **Booking Service:** Transactional booking.
- **Payment Service:** Stripe wrapper.
- **Messaging Service:** In-app messaging.

13. Technology Stack

- **Language:** Ruby (Airbnb original) or Java.
- **DB:** PostgreSQL sharded by region.
- **Search:** Elasticsearch.
- **Cache:** Redis.
- **Queue:** Kafka.

14. Database Choice

Postgres for ACID bookings. ES for search.

15. Cache Strategy

Redis for hot listings, search results.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for booking events.

18. Scaling Strategy

Shard by region. ES scales search.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Date-range inventory queries — materialize per-date.

21. Trade-offs

- **Materialized vs on-the-fly inventory:** Materialized is faster.

22. CAP Theorem Analysis

CP for bookings. AP for search.

23. Security Considerations

TLS. PII encryption. Payment tokenization (PCI). Anti-fraud.

24. Monitoring & Logging

Track search latency, booking conflict rate, payment latency.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Same date-scoped inventory as hotel booking.
- Mention messaging as a separate service.
- Discuss reviews.

27. Common Follow-up Questions

- How would you implement dynamic pricing (smart pricing)?
- How would you handle host cancellations?
- How would you implement "experiences" (Airbnb activities)?

28. Common Mistakes

- No multi-night atomic booking.
- PG for search — slow.
- No payment idempotency.

29. Optimization Ideas

- Pre-compute availability summaries.
- Cache search results for 5 min.
- Use CDN for listing photos.

30. Final Summary

Airbnb Lite is essentially hotel booking applied to home rentals. The core architecture is identical: date-scoped inventory, atomic multi-night booking with SELECT FOR UPDATE, ES for search, payment with idempotency. The differences from hotels: variable listing quality (reviews matter), messaging between host and guest, and dynamic pricing.

QUESTION 40 INTERMEDIATE

Design Yelp

1. Problem Statement

Design a local business review platform. Users search for businesses (restaurants, salons) by location, view details and reviews, and write their own reviews.

2. Functional Requirements

- Search businesses by location, category, keyword.
- View business details (hours, menu, photos, reviews).
- Write reviews with rating + text + photos.
- Bookmark businesses; follow reviewers.
- Business owner portal (respond to reviews, update info).

3. Non-functional Requirements

- Availability 99.95%.
- Search latency < 500ms.
- Read-heavy (100:1).

4. Capacity Estimation

200M users, 50M businesses, 250M reviews. Storage modest. Search QPS: 50K peak.

5. Back-of-the-envelope Math

Geospatial search (Redis GEO + ES). Reviews sharded by business_id.

6. API Design

```
GET /search?lat=...&lng=...&category=...&q=...
GET /businesses/{id}
POST /businesses/{id}/reviews body: {rating, text, photos}
POST /bookmarks body: {business_id}
POST /businesses/{id}/photos (owner)
```

7. Database Schema

```
Table: businesses (business_id, name, lat, lng, category, rating, review_count, ...)
Table: reviews (review_id, business_id, user_id, rating, text, created_at)
Table: photos (photo_id, business_id, uploader_id, url, created_at)
Table: bookmarks (user_id, business_id)
```

8. High-Level Design (HLD)

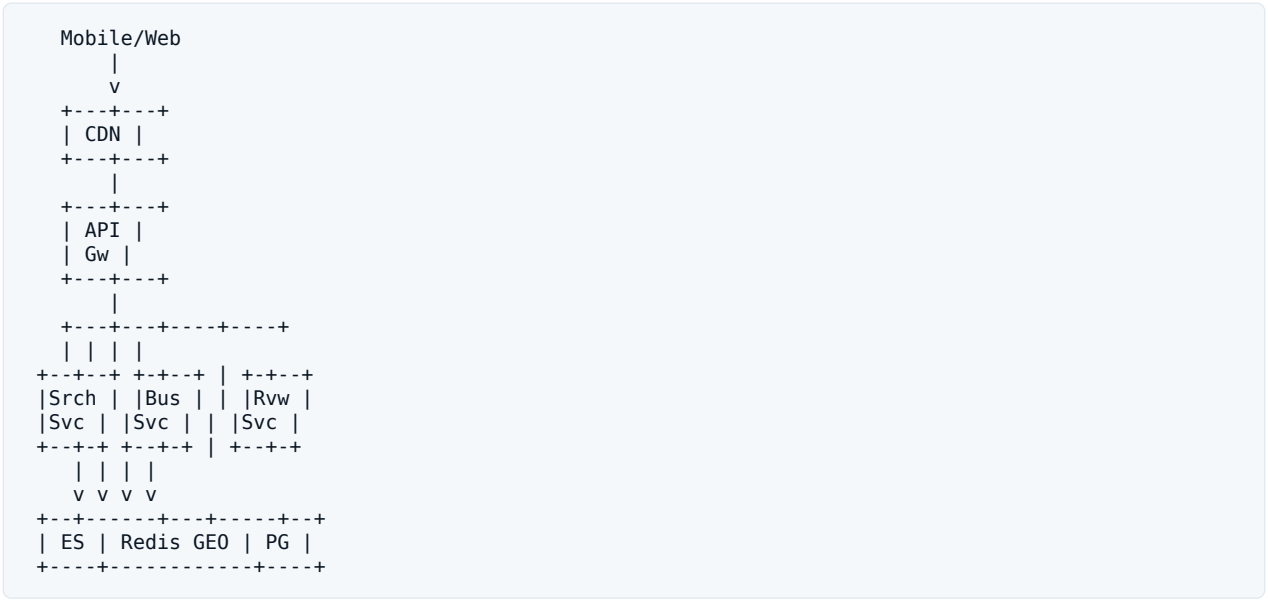
Mobile/Web → CDN → API Gateway → Search Service (ES + Redis GEO) + Business Service + Review Service. PG for reviews.

9. Low-Level Design (LLD)

Search: ES for keyword/category + Redis GEO for location radius. Merge results ranked by distance + rating + relevance.

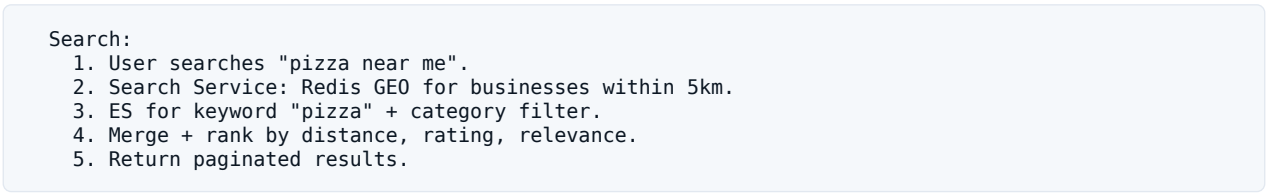
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Search Service:** ES + Redis GEO hybrid.
- **Business Service:** Business CRUD.
- **Review Service:** Review CRUD.
- **Photo Service:** S3 + CDN.

13. Technology Stack

- **Language:** Python/Java.
- **DB:** PostgreSQL.
- **Search:** Elasticsearch + Redis GEO.
- **Storage:** S3 + CDN.

14. Database Choice

Postgres for reviews/businesses (ACID). Redis GEO for location. ES for keyword.

15. Cache Strategy

Redis for hot business pages, search results.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for review events (rating recomputation).

18. Scaling Strategy

Shard PG by business_id. ES scales search.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Geospatial + keyword hybrid search — pre-filter by geo, then keyword.

21. Trade-offs

- **ES geo vs Redis GEO:** ES is simpler; Redis is faster for radius queries.

22. CAP Theorem Analysis

AP for search. CP for reviews.

23. Security Considerations

TLS. Auth. Anti-fake-review detection.

24. Monitoring & Logging

Track search latency, review submission latency.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Discuss hybrid geospatial + keyword search.
- Mention rating recomputation on new review.
- Discuss fake review detection.

27. Common Follow-up Questions

- How would you detect fake reviews (reviewer behavior)?
- How would you implement personalized recommendations?

- How would you handle business owner disputes?

28. Common Mistakes

- PG for search — too slow.
- No rating recomputation — stale averages.

29. Optimization Ideas

- Pre-compute business ratings (denormalized).
- Cache search results for 5 min.
- Use CDN for business photos.

30. Final Summary

Yelp tests geospatial + keyword search. The hybrid pattern: Redis GEO for radius queries (fast) + ES for keyword/category filtering (flexible). Reviews in sharded Postgres. The architecture is straightforward; the differentiator is search quality and fake-review detection.

QUESTION 41 INTERMEDIATE

Design Google Docs (Collaborative Editing)

1. Problem Statement

Design a real-time collaborative document editor. Multiple users edit simultaneously; changes sync in real-time with conflict-free merging. Support comments and version history.

2. Functional Requirements

- Create/edit text documents.
- Real-time collaboration (multiple cursors).
- Comments and suggestions.
- Version history and restore.
- Share with permissions.

3. Non-functional Requirements

- Latency: keystroke-to-render < 200ms.
- Availability 99.95%.
- No lost edits on disconnect.
- Eventual consistency (convergence).

4. Capacity Estimation

100M users, 1M concurrent documents. Avg doc 100KB. Storage: 1B docs × 100KB = 100TB.

5. Back-of-the-envelope Math

Conflict resolution via Operational Transform (OT) or CRDT (Conflict-free Replicated Data Type). WebSocket for real-time sync.

6. API Design

```
POST /docs body: {title, content} -> {doc_id}
GET /docs/{id} -> {content, version, collaborators}
// WebSocket /docs/{id}/stream for real-time ops
POST /docs/{id}/comments body: {text, anchor}
GET /docs/{id}/versions -> [versions]
POST /docs/{id}/versions/{v}/restore
```

7. Database Schema

```
Table: docs (doc_id, owner_id, title, current_version, created_at)
Table: doc_versions (version_id, doc_id, content_snapshot, created_at)
Table: doc_ops (op_id, doc_id, version, op_type, position, content, author_id, timestamp)
Table: collaborators (doc_id, user_id, role, cursor_position)
```

8. High-Level Design (HLD)

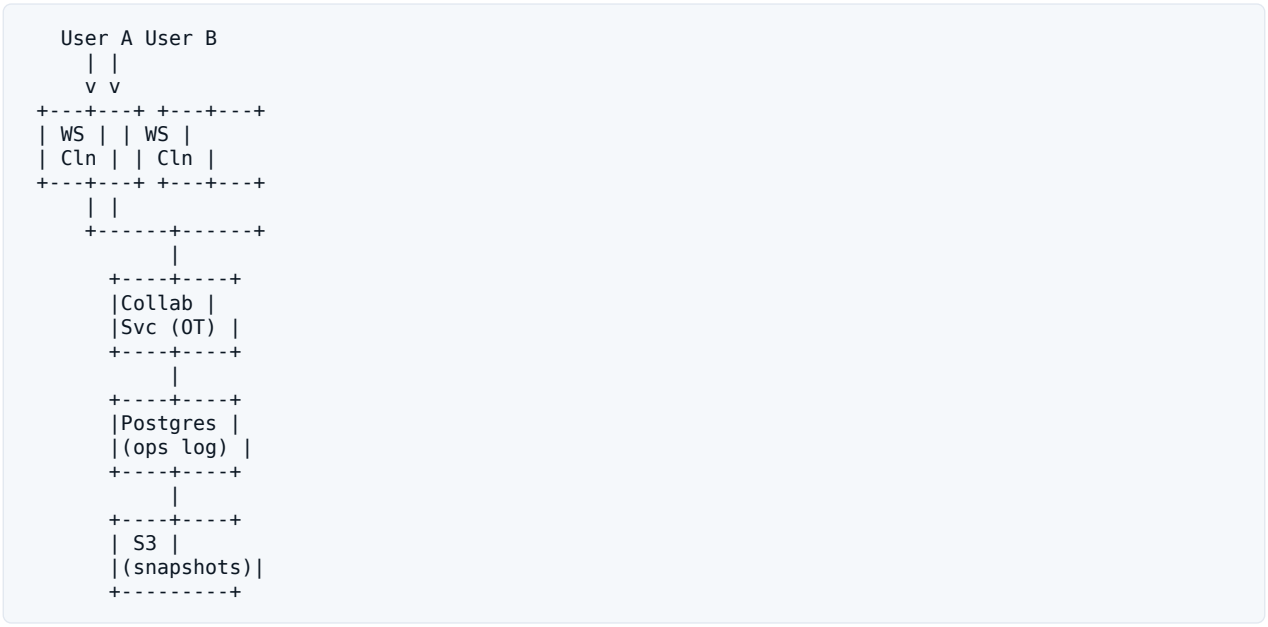
Browser → API Gateway → Doc Service (CRUD) + Collaboration Service (WebSocket, OT/CRDT) + Storage (PG for metadata, S3 for snapshots).

9. Low-Level Design (LLD)

Each keystroke is an operation (insert/delete at position). Operations sent via WebSocket to Collaboration Service, which applies Operational Transform (or CRDT merge) and broadcasts to other collaborators. Snapshots every N ops for fast restore.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Keystroke:
  1. User A types "h". Client sends op {type:INSERT, pos:5, content:"h"} via WebSocket.
  2. Collaboration Service applies op to in-memory doc state.
  3. Transforms op against concurrent ops from B (Operational Transform).
  4. Broadcasts transformed op to B via WebSocket.
  5. Appends op to PG ops log.
  6. Every 100 ops: snapshot to S3 for fast version restore.
```

12. Component Explanation

- **Doc Service:** CRUD for docs.
- **Collaboration Service:** WebSocket + OT/CRDT engine.
- **Operation Log:** PG: append-only op log per doc.

- **Snapshot Store:** S3: periodic doc snapshots for fast restore.
- **WebSocket Hub:** Real-time op broadcast.

13. Technology Stack

- **Language:** JavaScript/TypeScript (browser) + Go/Java (server).
- **DB:** PostgreSQL (op log).
- **Storage:** S3 (snapshots).
- **Realtime:** WebSocket.
- **Algorithm:** Operational Transform (Google Docs original) or CRDT (modern).

14. Database Choice

PG for op log (append-only, ordered). S3 for snapshots (large, infrequent reads).

15. Cache Strategy

In-memory doc state in Collaboration Service (per doc).

16. Load Balancer Strategy

Sticky WebSocket per doc_id.

17. Message Queue Usage

Optional: Kafka for doc events (analytics).

18. Scaling Strategy

Collaboration Service: stateful, scale by doc count (sticky routing).

19. Fault Tolerance

Snapshot every N ops enables restore. Op log is durable (PG).

20. Bottlenecks

- OT computation under high concurrency — limit collaborators per doc.
- Op log growth — snapshot + truncate.

21. Trade-offs

- **OT vs CRDT:** OT is complex but proven; CRDT is simpler but larger payloads.
- **Snapshots vs op replay:** Snapshots are faster restore; op replay is simpler.

22. CAP Theorem Analysis

AP. Eventual consistency via OT/CRDT convergence.

23. Security Considerations

TLS. Per-doc permissions. Audit log. PII handling for content.

24. Monitoring & Logging

Track keystroke latency, op broadcast latency, OT computation time.

25. Deployment Strategy

StatefulSets (sticky routing).

26. Interview Tips

- Discuss OT vs CRDT — core algorithmic choice.
- Mention op log + snapshots for version history.
- Sticky WebSocket routing by doc_id.

27. Common Follow-up Questions

- How would you handle offline edits (reconnect sync)?
- How would you implement comments anchored to text ranges?
- How would you scale to 100 concurrent editors on one doc?
- How would you implement suggest mode (track changes)?

28. Common Mistakes

- No OT/CRDT — concurrent edits conflict.
- No version history — lost data.
- Synchronous PG writes per keystroke — kills latency.

29. Optimization Ideas

- Batch small ops into single broadcast.
- Compress op payloads (delta encoding).
- Pre-allocate doc state in memory on collaboration service start.

30. Final Summary

Google Docs tests real-time collaborative editing. The core algorithmic choice is OT (Operational Transform) vs CRDT — both achieve convergence but with different trade-offs. The architecture: WebSocket per doc, in-memory doc state, append-only op log in PG, periodic snapshots in S3. Stateful scaling via sticky routing by doc_id. The system is AP — eventual consistency via convergence.

QUESTION 42 INTERMEDIATE

Design Google Calendar

1. Problem Statement

Design a calendar application. Users create events, invite attendees, set reminders, and view schedules by day/week/month. Sync across devices.

2. Functional Requirements

- Create/edit/delete events (title, time, location, attendees).
- Invite attendees; track RSVPs.
- Reminders (push, email).
- View by day/week/month.
- Sync across devices.
- Recurring events.

3. Non-functional Requirements

- Availability 99.95%.
- Event creation latency < 500ms.
- Reminder delivery < 1s of scheduled time.
- Cross-device sync < 5s.

4. Capacity Estimation

500M users, 10 events/user avg = 5B events. Reminders: 1B/day. Storage modest.

5. Back-of-the-envelope Math

Compute trivial. Reminder scheduling (cron-like) is the operational challenge.

6. API Design

```
POST /events body: {title, start, end, attendees, recurrence?}
GET /events?from=...&to=... -> [events in range]
POST /events/{id}/rsvp body: {status}
POST /events/{id}/reminders body: {method, offset_min}
GET /calendars -> [user's calendars]
```

7. Database Schema

```
Table: events (event_id, organizer_id, title, start, end, location, recurrence_rule, ...)
Table: attendees (event_id, user_id, status: PENDING|ACCEPTED|DECLINED, ...)
Table: reminders (reminder_id, event_id, user_id, trigger_at, method)
Table: calendars (calendar_id, owner_id, name, color, timezone)
```

8. High-Level Design (HLD)

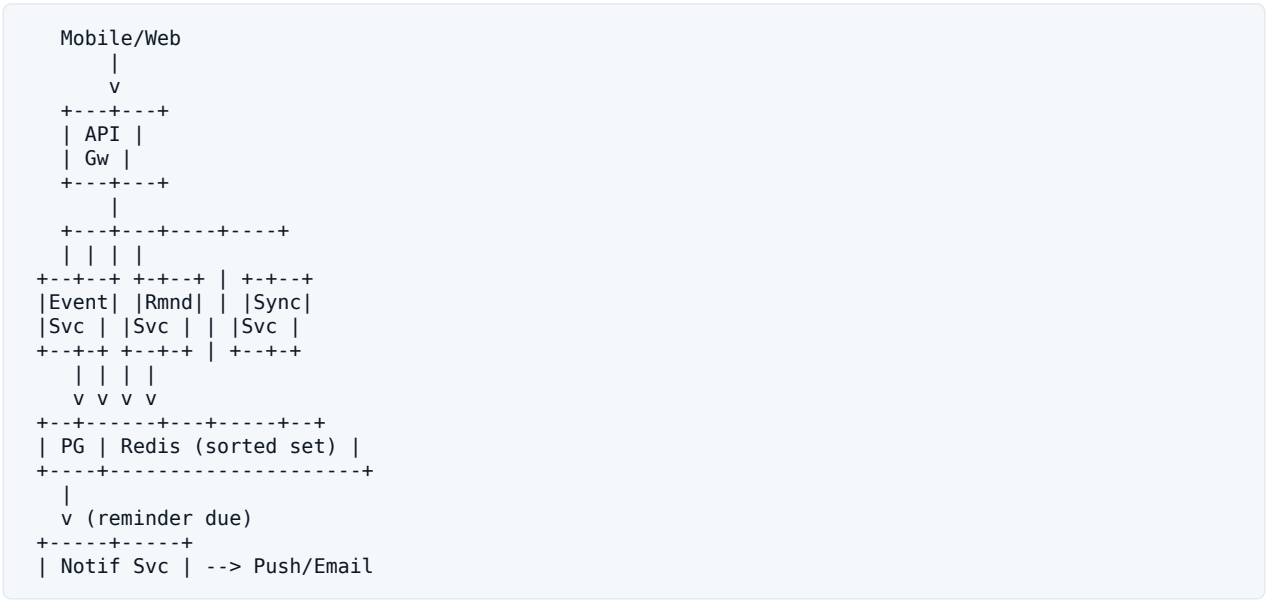
Mobile/Web → API Gateway → Event Service (CRUD) + Reminder Service (scheduling) + Notification Service + Sync Service. PG for events; Redis sorted set for reminders.

9. Low-Level Design (LLD)

Reminder Service: Redis sorted set keyed by trigger_at; worker pulls due reminders every second, dispatches to Notification Service.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Create event + reminders:
  1. POST /events; Event Service writes to PG.
  2. For each reminder: INSERT into Redis sorted set with score=trigger_at.
  3. Invite attendees via email; track RSVP.
  4. Sync to other devices via WebSocket/SSE.

Reminder delivery:
  1. Reminder worker queries Redis ZRANGEBYSCORE for due reminders.
  2. For each: dispatch to Notification Service (push/email).
  3. Remove from Redis.
```

12. Component Explanation

- **Event Service:** CRUD for events.
- **Reminder Service:** Schedules and dispatches reminders.
- **Notification Service:** Push, email, SMS.
- **Sync Service:** Cross-device sync.
- **Redis:** Sorted set for reminder scheduling.

13. Technology Stack

- **Language:** Java/Python.
- **DB:** PostgreSQL.

- **Cache:** Redis (reminders sorted set).
- **Realtime:** WebSocket/SSE for sync.

14. Database Choice

Postgres for events (ACID). Redis sorted set for reminder scheduling.

15. Cache Strategy

Redis for hot events (today's schedule per user).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for event changes (sync, notifications).

18. Scaling Strategy

Shard PG by user_id. Reminder worker scales by reminder volume.

19. Fault Tolerance

Multi-AZ. Redis with replicas.

20. Bottlenecks

- Reminder worker throughput on the hour (many reminders fire at :00).

21. Trade-offs

- **Cron vs Redis sorted set:** Sorted set is faster; cron is simpler.

22. CAP Theorem Analysis

CP for events. AP for sync.

23. Security Considerations

TLS. Auth. PII encryption (event details).

24. Monitoring & Logging

Track reminder delivery latency, event creation latency.

25. Deployment Strategy

Blue-green. Pre-scale reminder workers for peak hours.

26. Interview Tips

- Discuss Redis sorted set for reminder scheduling.

- Mention recurring events expansion (materialize occurrences).
- Discuss cross-device sync.

27. Common Follow-up Questions

- How would you handle timezones correctly?
- How would you implement "find a meeting time" (scheduling assistant)?
- How would you sync with Apple/Outlook calendars?

28. Common Mistakes

- Cron for reminders — doesn't scale.
- No recurring event expansion.
- Synchronous notification on event create.

29. Optimization Ideas

- Materialize recurring events for next 90 days.
- Batch reminder dispatch (per minute, not per second).
- Cache today's schedule per user.

30. Final Summary

Calendar tests scheduling and reminder delivery. The core architecture: Postgres for events, Redis sorted set for reminder scheduling (worker pulls due reminders every second), Notification Service for delivery. Recurring events must be materialized (expanded into individual occurrences) for fast queries. The system is CP for events, AP for sync.

QUESTION 43 INTERMEDIATE

Design a Stock Trading Platform (Robinhood)

1. Problem Statement

Design a stock trading app. Users view real-time prices, place market/limit orders, track portfolio, and receive trade confirmations. Strong consistency for orders.

2. Functional Requirements

- View real-time stock prices (quotes).
- Place market/limit orders.
- Track portfolio (holdings, P&L;).
- View order history and trade confirmations.
- Deposit/withdraw funds.

3. Non-functional Requirements

- Availability 99.99% (lost trades = lost money).
- Quote latency < 100ms.
- Order placement < 500ms.
- Strong consistency: no overspend, no double-trade.

4. Capacity Estimation

10M users, 1M trades/day, 10K stocks. Quote updates: 10K × 10/sec = 100K/s. Storage: 1M trades × 5yr × 1KB = 5GB.

5. Back-of-the-envelope Math

Quote feed from exchange; fan out via WebSocket. Order execution via exchange API. Portfolio in sharded Postgres (ACID).

6. API Design

```
GET /quotes/{symbol} -> {price, bid, ask, volume}
POST /orders body: {symbol, type: MARKET|LIMIT, side: BUY|SELL, qty, price?}
GET /portfolio -> {holdings, cash, total_value}
GET /orders?from=...&to=...
POST /deposits body: {amount, source}

// WebSocket /quotes/stream?symbols=... for real-time quotes
```

7. Database Schema

```
Table: users (user_id, cash_cents, ...)
Table: holdings (user_id, symbol, qty, avg_price_cents)
Table: orders (order_id, user_id, symbol, type, side, qty, price, status, created_at)
Table: trades (trade_id, order_id, exec_price, exec_qty, exec_at)
```

8. High-Level Design (HLD)

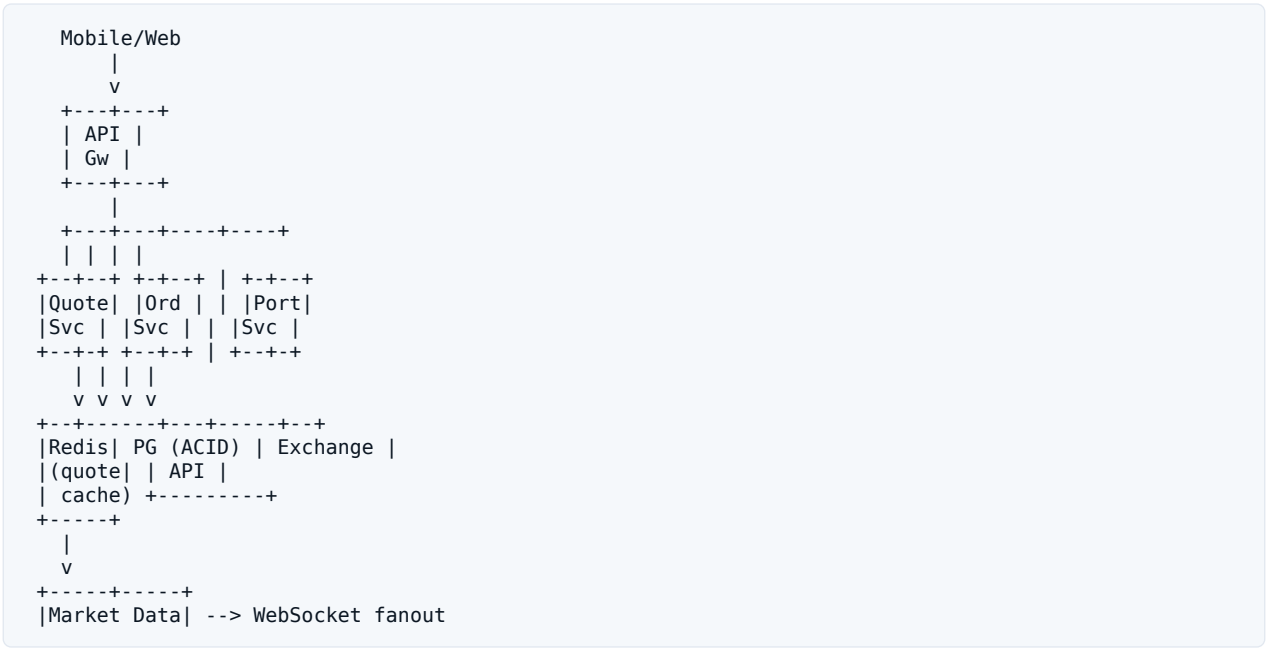
Mobile/Web → API Gateway → Quote Service (WebSocket fanout) + Order Service (transactional) + Portfolio Service + Market Data Feed (from exchange). PG for orders/holdings (ACID).

9. Low-Level Design (LLD)

Order placement: BEGIN TXN; SELECT user FOR UPDATE; verify cash sufficient (for BUY) or holdings sufficient (for SELL); INSERT order; UPDATE holdings/cash; COMMIT. Send order to exchange; on fill, UPDATE order status and INSERT trade.

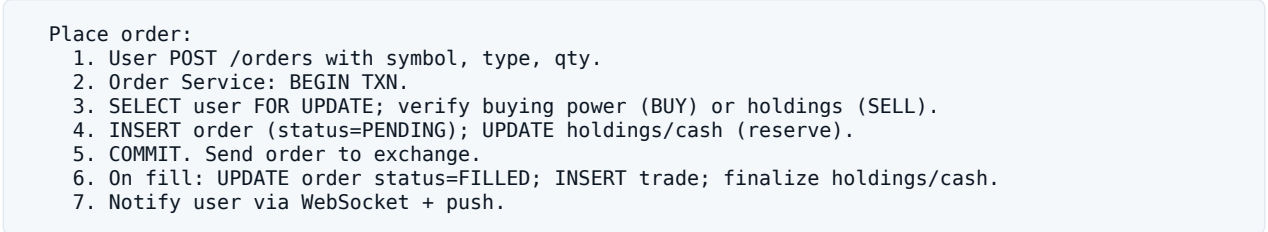
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Quote Service:** Receives market data feed; fans out via WebSocket.
- **Order Service:** Transactional order placement + execution.

- **Portfolio Service:** Holdings, P&L; calculations.
- **Market Data Feed:** From exchange (e.g. NASDAQ ITCH).
- **Exchange API Client:** Sends orders; receives fills.

13. Technology Stack

- **Language:** Java/C++ (low-latency).
- **DB:** PostgreSQL (ACID).
- **Cache:** Redis (quotes, hot portfolios).
- **Realtime:** WebSocket.
- **Queue:** Kafka (order events, audit).

14. Database Choice

Postgres for ACID (no overspend, no double-trade). Redis for quote cache.

15. Cache Strategy

Redis for hot quotes (sub-ms reads), user portfolio snapshots.

16. Load Balancer Strategy

L7 LB. Sticky session for WebSocket.

17. Message Queue Usage

Kafka for order events, audit trail.

18. Scaling Strategy

Shard PG by user_id. Quote fanout scales with subscriber count.

19. Fault Tolerance

Multi-AZ. Synchronous replication for orders.

20. Bottlenecks

- Quote fanout for hot stocks (AAPL) — shard by symbol.
- Order contention on hot IPOs — queue requests.

21. Trade-offs

- **Push vs pull quotes:** Push is real-time; pull is simpler.
- **Sync vs async exchange API:** Sync confirms immediately; async is faster.

22. CAP Theorem Analysis

CP. Strong consistency for orders and money.

23. Security Considerations

TLS. 2FA. PCI compliance. Encryption at rest. SEC-compliant audit log.

24. Monitoring & Logging

Track quote latency, order latency, exchange API latency, portfolio accuracy.

25. Deployment Strategy

Blue-green. Co-located with exchange for low latency.

26. Interview Tips

- Emphasize ACID for orders and money.
- Discuss WebSocket for real-time quotes.
- Mention exchange API integration.

27. Common Follow-up Questions

- How would you implement options trading?
- How would you handle market open spikes?
- How would you detect market manipulation?

28. Common Mistakes

- No transaction on order — overspend.
- Polling for quotes — slow.
- No audit log — compliance failure.

29. Optimization Ideas

- Pre-compute portfolio totals (cached).
- Batch quote fanout per symbol.
- Co-locate with exchange for sub-ms execution.

30. Final Summary

A stock trading platform tests ACID transactions and real-time data fanout. The core challenge is atomic order placement (no overspend, no double-trade) under high concurrency — solved with SELECT FOR UPDATE on user balance/holdings. Quotes are pushed via WebSocket from market data feed. The system is CP — strong consistency is non-negotiable for money. Co-location with exchange minimizes execution latency.

QUESTION 44 INTERMEDIATE

Design Ticketmaster

1. Problem Statement

Design a ticket marketplace. Users browse events, select seats, hold them temporarily, and purchase. Handle high-demand onsales (millions competing for the same event).

2. Functional Requirements

- Browse events by artist, venue, city, date.
- View seat map with availability.
- Hold seats for 5 min during checkout.
- Purchase with payment; receive ticket (QR code).
- Resale marketplace.

3. Non-functional Requirements

- Availability 99.99% during onsales.
- Seat hold atomicity (no double-booking).
- Handle 1M+ concurrent users for popular onsales.
- Latency: seat map < 500ms.

4. Capacity Estimation

100K events/year, 10M tickets sold. Onsale spikes: 1M concurrent users. Storage modest.

5. Back-of-the-envelope Math

Same as movie ticket booking but at higher scale. Use virtual queue for popular onsales.

6. API Design

```
GET /events?artist=...&city=...
GET /events/{id}/seats -> seat map
POST /events/{id}/holds body: {seat_ids: [...]} -> {hold_id, expires_at}
POST /holds/{id}/pay body: {payment}
POST /tickets/{id}/resale body: {price}
```

7. Database Schema

```
Table: events (event_id, artist, venue_id, date, status)
Table: seats (seat_id, event_id, row, col, section, status, hold_expires_at, booking_id)
Table: bookings (booking_id, user_id, event_id, seat_ids, total, status)
Table: tickets (ticket_id, booking_id, seat_id, qr_code, owner_id)
```

8. High-Level Design (HLD)

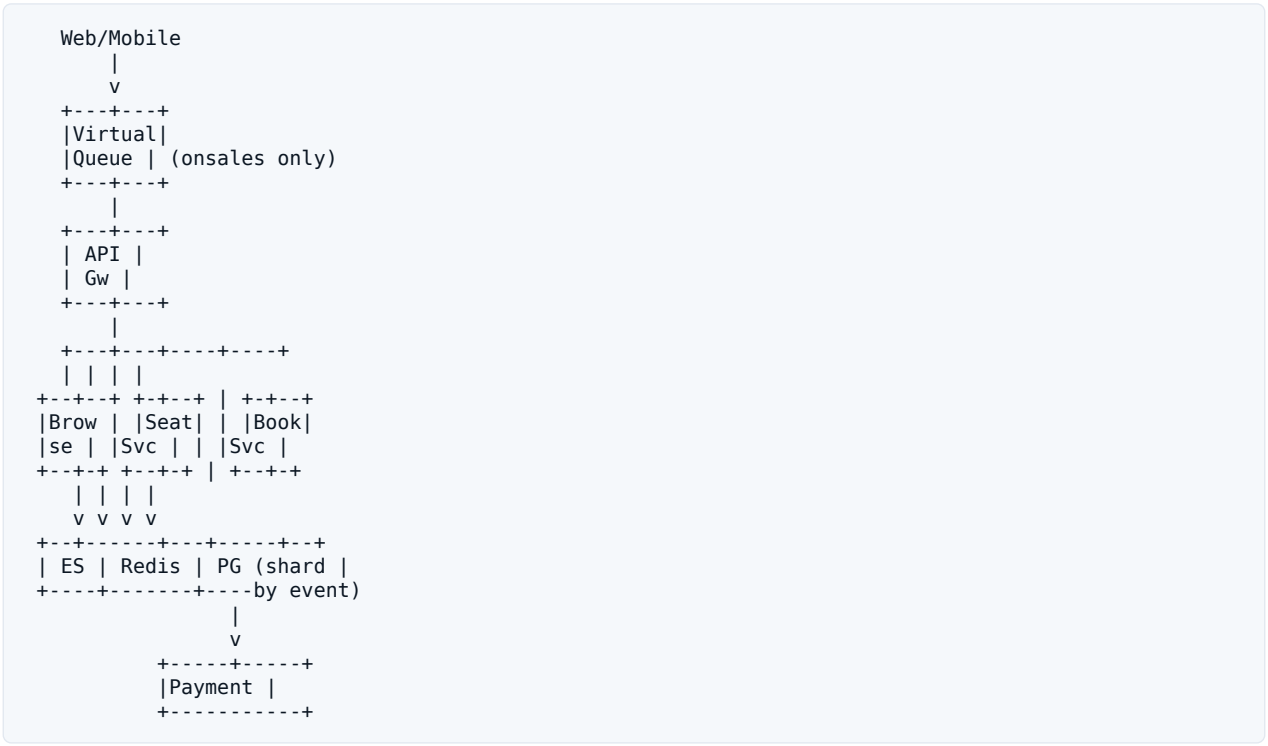
Web/Mobile → CDN → Virtual Queue (for onsales) → API Gateway → Browse Service + Seat Service + Booking Service + Payment. PG sharded by event_id (ACID for seats).

9. Low-Level Design (LLD)

Same as movie ticket booking: SELECT FOR UPDATE on seats, atomic hold with TTL, payment with idempotency. Virtual queue: when concurrent users exceed capacity, show waiting room; admit in batches.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Onsale flow:
  1. 1M users arrive at onsale start.
  2. Virtual Queue: only 10K admitted at a time; others see "you are #N in line".
  3. Admitted users browse seat map (cached in Redis).
  4. Select seats, POST /holds &mdash; SELECT FOR UPDATE; 5 min TTL.
  5. Pay via Payment Service (idempotent).
  6. On success: seats BOOKED, tickets generated. On failure: release hold.
```

12. Component Explanation

- **Virtual Queue:** Throttles users during onsales (waiting room).
- **Browse Service:** Event search (ES).
- **Seat Service:** Seat map rendering; availability.
- **Booking Service:** Transactional seat hold + booking.
- **Payment Service:** Stripe wrapper; idempotent.

13. Technology Stack

- **Language:** Java/Go.
- **DB:** PostgreSQL sharded by event_id.
- **Cache:** Redis (seat availability, queue position).
- **Search:** Elasticsearch.
- **Queue:** Kafka.

14. Database Choice

Postgres for ACID seat booking. Shard by event_id.

15. Cache Strategy

Redis for seat availability (read-heavy), queue position tracking.

16. Load Balancer Strategy

L7 LB. Geographic routing.

17. Message Queue Usage

Kafka for booking events.

18. Scaling Strategy

Pre-scale for known onsales. Virtual queue protects backend.

19. Fault Tolerance

Multi-AZ. Payment idempotent via hold_id.

20. Bottlenecks

- Onsale spike — virtual queue + pre-scale.
- Seat map rendering — pre-render and cache.

21. Trade-offs

- **Virtual queue vs first-come:** Queue is fairer; first-come is faster (but kills backend).
- **5 min vs 10 min hold:** 5 min reduces inventory lock; 10 min reduces payment failures.

22. CAP Theorem Analysis

CP for seat booking (no double-booking).

23. Security Considerations

TLS. Anti-bot (CAPTCHA, rate limit). Anti-scalper (purchase limits per user).

24. Monitoring & Logging

Track queue wait time, hold conflict rate, payment latency.

25. Deployment Strategy

Pre-scale for onsales (10x normal). Blue-green.

26. Interview Tips

- Mention virtual queue — critical for onsale spikes.
- Pre-render seat maps.
- Discuss anti-bot and anti-scalper measures.

27. Common Follow-up Questions

- How would you detect bot scalping?
- How would you implement dynamic pricing (platinum seats)?
- How would you handle resale marketplace?

28. Common Mistakes

- No virtual queue — backend dies on onsale.
- No anti-bot — bots buy all tickets.
- Sync seat availability from DB — too slow.

29. Optimization Ideas

- Pre-render seat maps as static images.
- Batch WebSocket updates per event (1/sec).
- Pre-warm Redis with seat availability before onsale.

30. Final Summary

Ticketmaster extends movie ticket booking with extreme onsale spikes. The core innovation is the virtual queue (waiting room) that throttles users during onsales. Seat booking uses the same SELECT FOR UPDATE pattern with 5-min TTL holds. Pre-rendering seat maps and pre-warming Redis caches are operationally critical. Anti-bot and anti-scalper measures are essential for fairness.

QUESTION 45 INTERMEDIATE

Design Zoom (Video Conferencing)

1. Problem Statement

Design a video conferencing platform. Users host/join meetings, share video/audio/screen, and record sessions. Handle real-time media streaming at global scale.

2. Functional Requirements

- Host/join meetings (up to 1000 participants).
- Real-time video/audio/screen sharing.
- Recording (cloud + local).
- Chat within meeting.
- Mute/unmute, raise hand, breakout rooms.

3. Non-functional Requirements

- Latency: < 200ms end-to-end (interactive).
- Availability 99.95%.
- Handle 1M+ concurrent meetings.
- Audio quality preserved on packet loss.

4. Capacity Estimation

1M concurrent meetings × 10 participants avg = 10M concurrent users. Bandwidth: 10M × 2Mbps (video) = 20Tbps globally.

5. Back-of-the-envelope Math

Media flows peer-to-peer where possible; SFU (Selective Forwarding Unit) for larger meetings. WebRTC for transport. Recording via media server capture.

6. API Design

```
POST /meetings body: {host_id, settings} -> {meeting_id, join_url}
POST /meetings/{id}/join body: {user_id} -> {webrtc_params, participant_id}
// WebRTC media flows via SFU
POST /meetings/{id}/record/start
POST /meetings/{id}/chat body: {text}
POST /meetings/{id}/raise_hand
```

7. Database Schema

```
Table: meetings (meeting_id, host_id, start_time, end_time, settings)
Table: participants (meeting_id, user_id, joined_at, left_at)
Table: recordings (recording_id, meeting_id, url, duration, created_at)
Table: chat_messages (msg_id, meeting_id, user_id, text, timestamp)
```

8. High-Level Design (HLD)

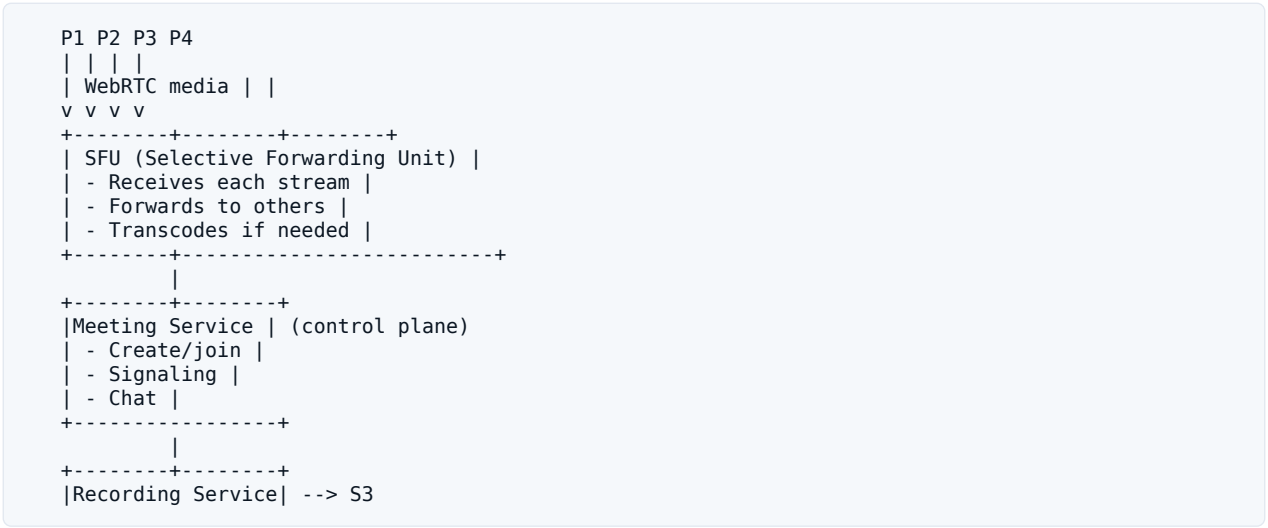
Browser/Mobile → API Gateway → Meeting Service (control plane) + Media Servers (SFU, data plane) + Recording Service. WebRTC for media.

9. Low-Level Design (LLD)

Control plane: REST APIs for create/join meetings, signaling. Data plane: SFU (Selective Forwarding Unit) receives each participant's stream, forwards to others (no P2P mesh beyond ~4 participants). Recording: media server captures mixed stream to S3.

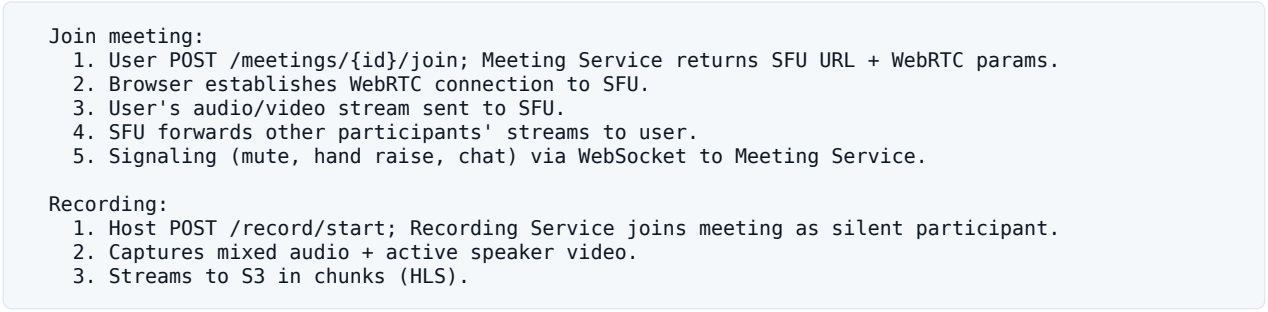
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Meeting Service:** Control plane: create/join, signaling, chat.
- **SFU (Media Server):** Data plane: receives + forwards WebRTC streams.
- **Recording Service:** Captures meeting to S3.
- **TURN/STUN:** NAT traversal for WebRTC.
- **WebSocket Hub:** Signaling + chat.

13. Technology Stack

- **Language:** C++ for media servers; Java/Go for control.

- **Media:** WebRTC + SFU (e.g. mediasoup, Jitsi Videobridge).
- **DB:** PostgreSQL.
- **Storage:** S3 (recordings).
- **Realtime:** WebSocket.

14. Database Choice

Postgres for meeting metadata. Recordings to S3. In-memory state for active meetings.

15. Cache Strategy

Redis for active meeting state (participant list).

16. Load Balancer Strategy

L7 LB. Sticky WebSocket. Geographic routing for SFU (closest to user).

17. Message Queue Usage

Kafka for meeting events (start, end, recording).

18. Scaling Strategy

SFU scales by meeting count + participant count. Add more SFU nodes as needed.

19. Fault Tolerance

Multi-AZ. SFU failover: participants reconnect to alternate SFU.

20. Bottlenecks

- SFU bandwidth — 1000-participant meetings need dedicated SFU.
- TURN server load — scale horizontally.

21. Trade-offs

- **P2P mesh vs SFU:** P2P is low-latency for small meetings; SFU scales to large.
- **Live recording vs post-process:** Live is real-time; post is higher quality.

22. CAP Theorem Analysis

AP. Eventual consistency for chat; real-time media is best-effort.

23. Security Considerations

TLS + DTLS-SRTP for media. End-to-end encryption (optional). Meeting passwords. Waiting room. Anti-Zoom-bombing (host controls).

24. Monitoring & Logging

Track media latency, packet loss, SFU load, meeting join latency.

25. Deployment Strategy

SFU in dedicated node pools (high bandwidth). Multi-region.

26. Interview Tips

- Discuss SFU vs P2P mesh — core architectural decision.
- Mention WebRTC.
- Discuss TURN/STUN for NAT traversal.
- Recording as a separate silent participant.

27. Common Follow-up Questions

- How would you implement end-to-end encryption?
- How would you handle 1000-participant webinars?
- How would you implement virtual backgrounds?
- How would you add live transcription?

28. Common Mistakes

- P2P mesh for large meetings — kills bandwidth.
- No TURN server — NAT traversal fails.
- Synchronous recording on media path.

29. Optimization Ideas

- Simulcast (multiple resolutions per stream) for adaptive quality.
- Server-side noise cancellation.
- Pre-warm SFU capacity during business hours.
- Edge-cache meeting metadata.

30. Final Summary

Zoom tests real-time media architecture. The core decision is SFU (Selective Forwarding Unit) over P2P mesh — SFU scales to large meetings by receiving each participant's stream once and forwarding. WebRTC is the transport. TURN/STUN servers handle NAT traversal. Recording is a separate silent participant that captures the mixed stream to S3. The architecture rewards investment in SFU infrastructure and edge deployment for low latency.

QUESTION 46

INTERMEDIATE

Design Slack

1. Problem Statement

Design a team communication platform. Users join workspaces, channels, send messages, share files, and search history. Real-time updates across devices.

2. Functional Requirements

- Workspaces, channels, DMs.
- Send messages (text, files, code blocks, threads).
- Real-time updates via WebSocket.
- Search across message history.
- Notifications (mention, keyword).
- Integrations (bots, webhooks).

3. Non-functional Requirements

- Latency: message delivery < 500ms.
- Availability 99.95%.
- Search latency < 1s.
- Ordering: messages in channel must be in order.

4. Capacity Estimation

10M paid users, 10M messages/day, 1M concurrent WebSocket connections. Storage: 10M × 5yr × 1KB = 50TB messages.

5. Back-of-the-envelope Math

Similar to chat apps: sharded by workspace_id. WebSocket per user. Search via ES.

6. API Design

```
POST /channels/{id}/messages body: {text, files?}
GET /channels/{id}/messages?cursor=...
GET /search?q=...
POST /files (upload)

// WebSocket /stream for real-time updates
```

7. Database Schema

```
Table: workspaces (workspace_id, name, ...)
Table: channels (channel_id, workspace_id, name, ...)
Table: messages (msg_id, channel_id, author_id, text, created_at, thread_parent_id?)
Table: users (user_id, workspace_id, name, ...)
Table: files (file_id, msg_id, s3_key, filename)
```

8. High-Level Design (HLD)

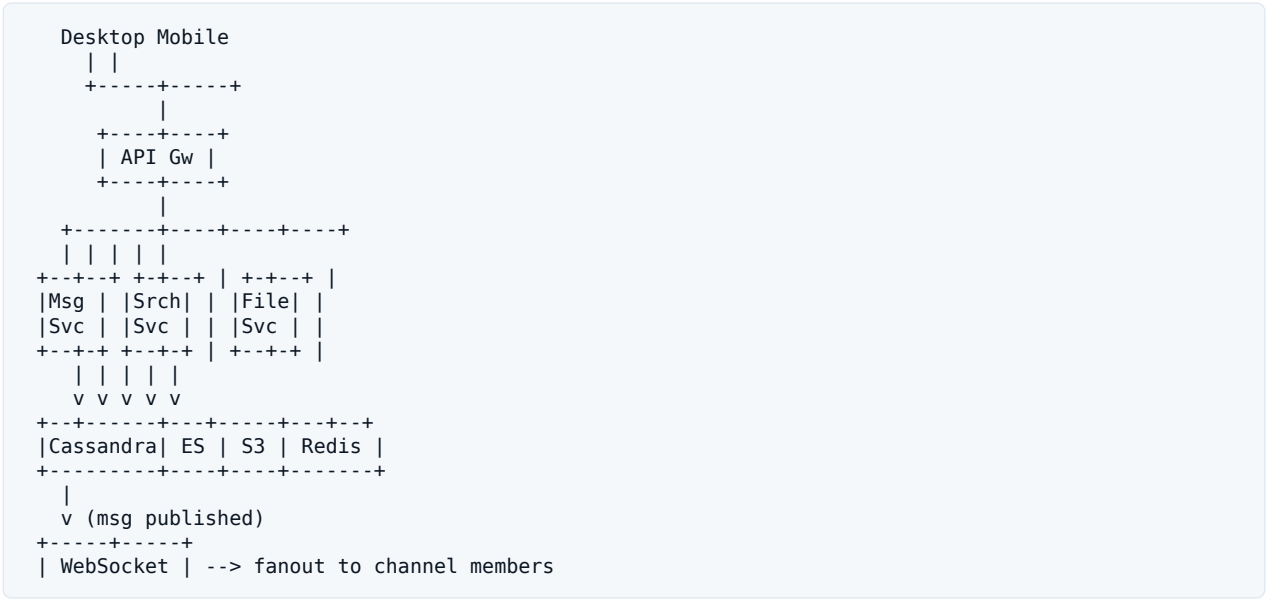
Desktop/Mobile → API Gateway → Message Service + Search Service (ES) + File Service (S3) + WebSocket Hub. Sharded by workspace_id.

9. Low-Level Design (LLD)

Standard chat architecture: WebSocket per user, message router fans out to channel members. Search via ES indexed via CDC.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Send message:
  1. User POST /channels/{id}/messages.
  2. Message Service writes to Cassandra (sharded by channel_id).
  3. Publish event to Kafka.
  4. WebSocket Hub fans out to all channel members online.
  5. Push notification to offline members (if mentioned).
  6. Async: index in ES for search.
```

12. Component Explanation

- **Message Service:** CRUD for messages.
- **WebSocket Hub:** Real-time fanout to channel members.
- **Search Service:** Elasticsearch.
- **File Service:** S3 + CDN.

- **Notification Service:** Push for mentions/keywords.

13. Technology Stack

- **Language:** PHP/HHVM (Slack original) or Go.
- **DB:** Cassandra (messages).
- **Cache:** Redis.
- **Search:** Elasticsearch.
- **Storage:** S3.
- **Realtime:** WebSocket.

14. Database Choice

Cassandra for messages (write-heavy, partitioned by channel_id).

15. Cache Strategy

Redis for hot channel state, user presence.

16. Load Balancer Strategy

Sticky WebSocket per user.

17. Message Queue Usage

Kafka for message events.

18. Scaling Strategy

Shard by workspace_id. WebSocket Hub scales by connection count.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Large channel fanout (10K members) — batch.

21. Trade-offs

- **Push vs pull messages:** Push is real-time; pull is simpler.

22. CAP Theorem Analysis

AP. Eventual consistency for search.

23. Security Considerations

TLS. SSO for enterprise. Audit log. PII encryption.

24. Monitoring & Logging

Track message delivery latency, search latency, WebSocket connections.

25. Deployment Strategy

Blue-green. Multi-region.

26. Interview Tips

- Shard by workspace_id (queries are workspace-scoped).
- Mention notifications as a separate flow.
- Discuss integrations (bots, webhooks).

27. Common Follow-up Questions

- How would you implement threads (replies to messages)?
- How would you add video calls (Slack Huddles)?
- How would you implement Slack Connect (cross-workspace channels)?

28. Common Mistakes

- No search index — PG too slow.
- Synchronous ES indexing — blocks message send.
- No mention notification — poor UX.

29. Optimization Ideas

- Batch WebSocket fanout (per channel, not per user).
- Pre-compute unread counts.
- Compress messages (delta encoding for replies).

30. Final Summary

Slack is a team chat at enterprise scale. The architecture is the standard chat pattern: sharded by workspace_id, WebSocket per user, message router fans out. Search via ES. Notifications for mentions. The interesting features are threads (replies to messages) and integrations (bots, webhooks). The system is AP — eventual consistency for search is acceptable.

QUESTION 47 INTERMEDIATE

Design Twitch

1. Problem Statement

Design a live streaming platform. Streamers broadcast video; viewers watch in real-time, chat, subscribe, and donate. Handle millions of concurrent viewers per stream.

2. Functional Requirements

- Streamers broadcast live video via encoder.
- Viewers watch live streams.
- Real-time chat alongside stream.
- Subscribe/donate to streamers.
- VOD (video on demand) after stream ends.

3. Non-functional Requirements

- Stream latency < 5s (interactive).
- Availability 99.95%.
- Handle 1M+ concurrent viewers per popular stream.
- Chat latency < 500ms.

4. Capacity Estimation

2M streamers, 50K concurrent live streams, 20M concurrent viewers. Popular stream: 1M viewers. Bandwidth: 20M × 5Mbps = 100Tbps.

5. Back-of-the-envelope Math

Video via WebRTC or HLS (low-latency). Chat via WebSocket fanout. CDN for video delivery.

6. API Design

```
POST /streams/start body: {encoder_url, title, category} -> {stream_key}
GET /streams/{id} -> {stream_url, chat_url}
POST /streams/{id}/chat body: {text}
POST /streams/{id}/subscribe body: {user_id, tier}
POST /streams/{id}/donations body: {amount, message}
```

7. Database Schema

```
Table: streams (stream_id, streamer_id, title, category, started_at, viewer_count,
status)
Table: chat_messages (msg_id, stream_id, user_id, text, timestamp)
Table: subscriptions (user_id, streamer_id, tier, started_at)
Table: donations (donation_id, user_id, streamer_id, amount, message, timestamp)
```

8. High-Level Design (HLD)

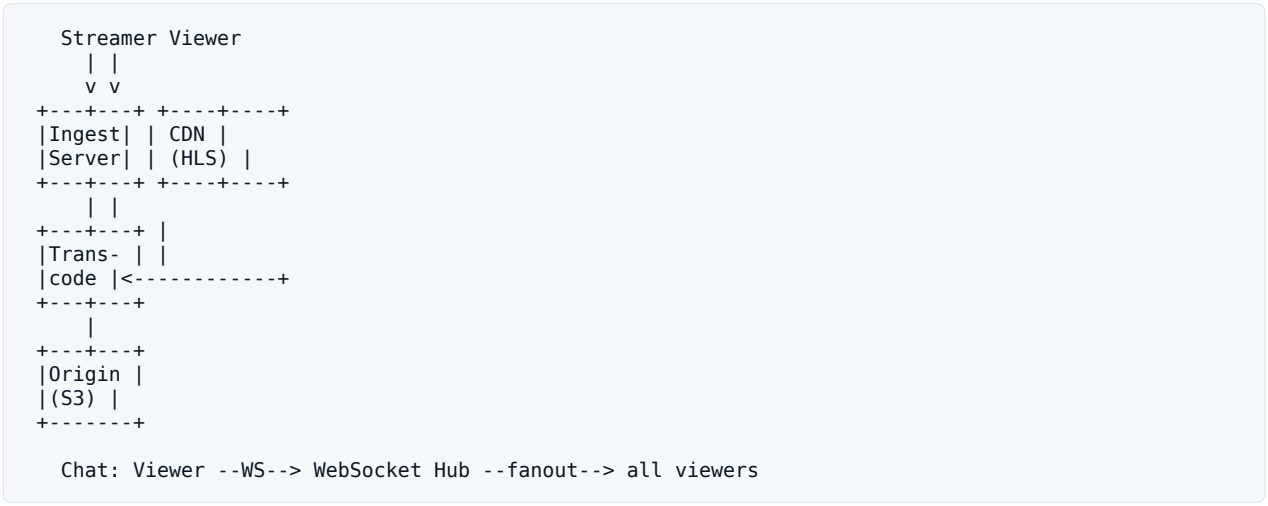
Encoder → Ingest Server → Transcode → Origin → CDN → Viewers. Chat via WebSocket Hub. VOD processing after stream ends.

9. Low-Level Design (LLD)

Video: streamer pushes RTMP to ingest server. Transcode to multiple bitrates. CDN distributes via HLS. Chat: WebSocket fanout — popular streams may have 1M concurrent chatters, requiring sharded fanout.

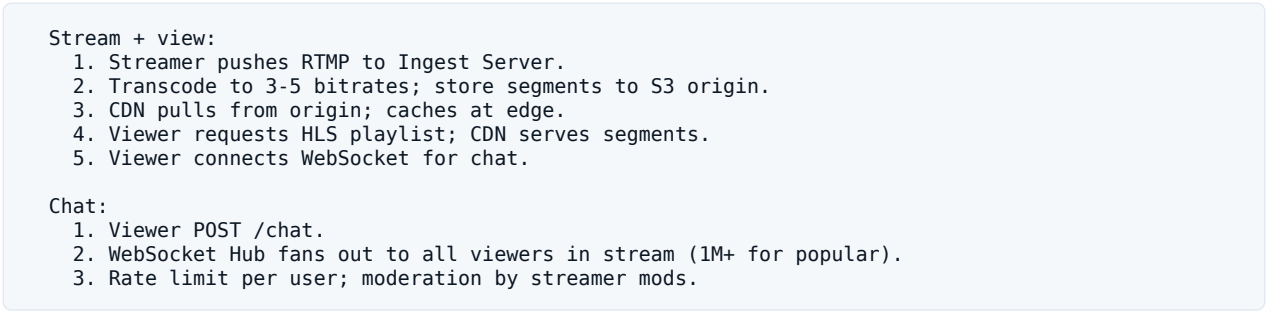
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Ingest Server:** Receives RTMP from streamers.
- **Transcode Pipeline:** Multi-bitrate encoding.
- **CDN:** Video delivery at edge.
- **WebSocket Hub:** Chat fanout.
- **VOD Service:** Process recording after stream ends.

13. Technology Stack

- **Language:** C++ for transcoding; Go/Java for services.
- **Media:** RTMP ingest, HLS delivery.

- **CDN:** CloudFront/Akamai.
- **Storage:** S3.
- **Realtime:** WebSocket.

14. Database Choice

Cassandra for chat (write-heavy, partitioned by stream_id). PG for subscriptions.

15. Cache Strategy

Redis for stream metadata, viewer counts.

16. Load Balancer Strategy

Geographic routing. Ingest servers near streamers.

17. Message Queue Usage

Kafka for chat events, donation events.

18. Scaling Strategy

CDN scales with viewer count. WebSocket Hub sharded for popular streams.

19. Fault Tolerance

Multi-region. Ingest failover.

20. Bottlenecks

- 1M-viewer chat fanout — shard WebSocket Hub by stream_id.
- Egress bandwidth — CDN.
- Transcode compute — GPU instances.

21. Trade-offs

- **HLS vs WebRTC:** HLS is 5-10s latency; WebRTC is < 1s but harder to scale.
- **Live vs VOD transcoding:** Live is real-time; VOD is cheaper.

22. CAP Theorem Analysis

AP. Eventual consistency for chat.

23. Security Considerations

TLS. Stream key auth. Chat moderation (auto-mod for profanity). Anti-harassment.

24. Monitoring & Logging

Track stream latency, transcode lag, chat fanout latency, viewer count.

25. Deployment Strategy

Multi-region. Ingest close to streamers.

26. Interview Tips

- Discuss HLS vs WebRTC for latency.
- Mention chat fanout challenge for popular streams.
- Discuss transcode pipeline.

27. Common Follow-up Questions

- How would you implement low-latency streaming (< 1s)?
- How would you handle clip creation (user-generated highlights)?
- How would you implement raids (redirect viewers to another stream)?

28. Common Mistakes

- No CDN — egress kills origin.
- Single WebSocket Hub — can't handle popular streams.
- Synchronous transcode — latency.

29. Optimization Ideas

- Simulcast (multiple resolutions from streamer).
- Edge-cache stream metadata.
- Batch chat fanout (per 100ms, not per message).

30. Final Summary

Twitch tests live video streaming at massive concurrency. The core architecture: RTMP ingest, transcode to multi-bitrate, HLS delivery via CDN, WebSocket chat fanout. The challenge is popular streams (1M+ viewers, 1M+ chatters) — WebSocket Hub must shard by stream_id. Low-latency streaming uses WebRTC instead of HLS. The architecture rewards CDN investment and chat fanout optimization.

QUESTION 48 INTERMEDIATE

Design Discord

1. Problem Statement

Design a voice + text chat platform for communities (servers). Users join servers, channels (text + voice), and chat in real-time. Voice is the differentiator.

2. Functional Requirements

- Servers, text channels, voice channels.
- Real-time text messaging.
- Real-time voice chat (always-on, push-to-talk).
- Direct messages.
- Roles and permissions.
- Bots and integrations.

3. Non-functional Requirements

- Latency: text < 500ms, voice < 100ms.
- Availability 99.95%.
- Handle 10M concurrent voice users.
- Ordering: text messages per channel.

4. Capacity Estimation

150M MAU, 10M concurrent voice users. Avg voice channel: 5 users. Storage modest.

5. Back-of-the-envelope Math

Voice via WebRTC + SFU. Text via WebSocket. Servers sharded by server_id.

6. API Design

```
POST /channels/{id}/messages body: {text}
GET /channels/{id}/messages?cursor=...

// WebSocket /stream for real-time text + voice signaling
// WebRTC for voice media
```

7. Database Schema

```
Table: servers (server_id, owner_id, name, ...)
Table: channels (channel_id, server_id, type: TEXT|VOICE, name)
Table: messages (msg_id, channel_id, author_id, text, created_at)
Table: members (server_id, user_id, role, joined_at)
```

8. High-Level Design (HLD)

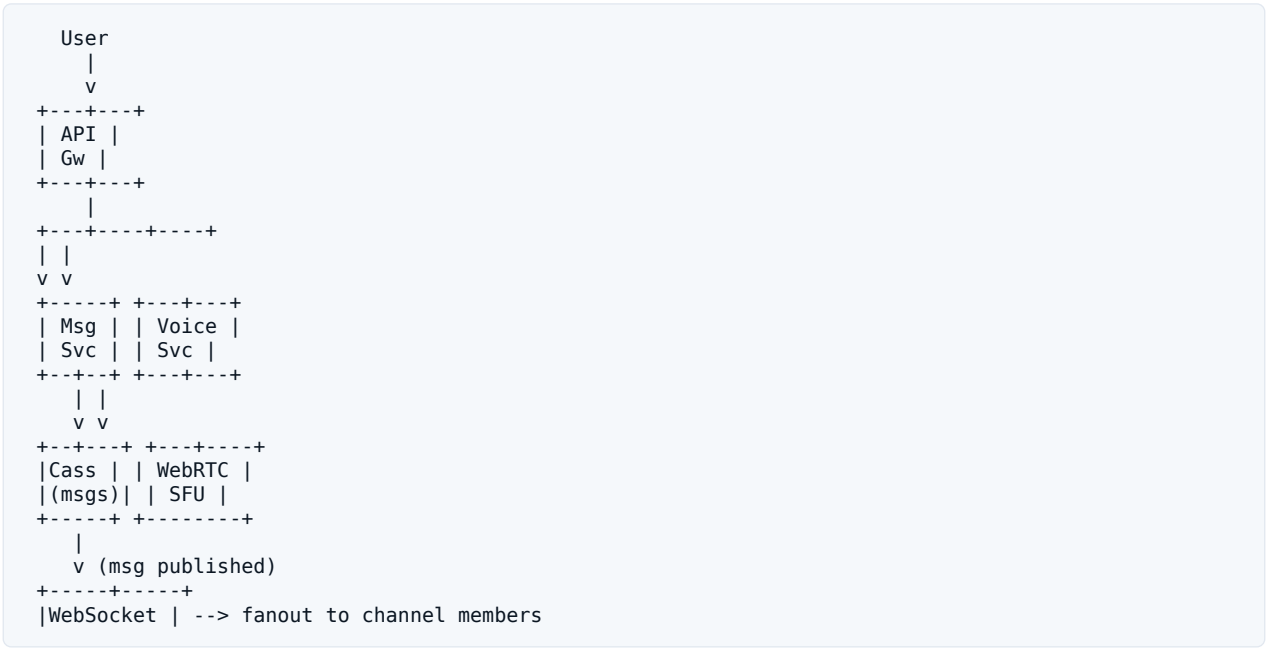
Desktop/Mobile → API Gateway → Message Service (text) + Voice Service (WebRTC SFU) + WebSocket Hub. Sharded by server_id.

9. Low-Level Design (LLD)

Voice: user joins voice channel; connects to SFU via WebRTC. SFU forwards audio to other participants. Text: standard WebSocket fanout.

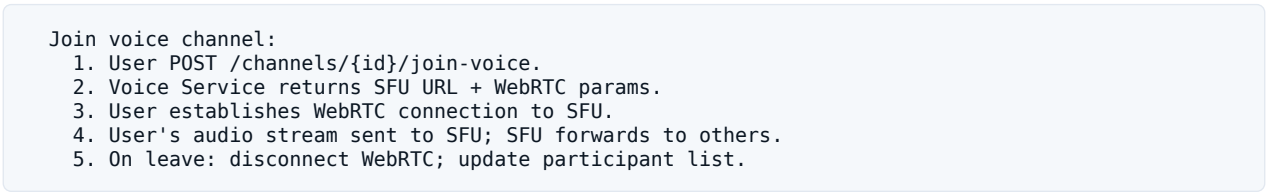
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Message Service:** Text message CRUD.
- **Voice Service:** WebRTC SFU management.
- **WebSocket Hub:** Real-time text fanout.
- **SFU:** Voice media forwarding.

13. Technology Stack

- **Language:** Elixir (Discord original) or Go.
- **DB:** Cassandra (messages).
- **Cache:** Redis.
- **Realtime:** WebSocket + WebRTC.

14. Database Choice

Cassandra for messages. In-memory for voice state.

15. Cache Strategy

Redis for server/channel metadata.

16. Load Balancer Strategy

Sticky WebSocket per user. Geographic routing for SFU.

17. Message Queue Usage

Kafka for message events.

18. Scaling Strategy

Shard by server_id. SFU scales by voice channel count.

19. Fault Tolerance

Multi-AZ. SFU failover: users reconnect.

20. Bottlenecks

- Large voice channels (1000 users) — broadcast mode.

21. Trade-offs

- **SFU vs P2P mesh:** SFU scales; P2P is low-latency for small.

22. CAP Theorem Analysis

AP. Eventual consistency for text.

23. Security Considerations

TLS + DTLS-SRTP. Roles/permissions. Anti-abuse (rate limit, moderation).

24. Monitoring & Logging

Track voice latency, text delivery latency, SFU load.

25. Deployment Strategy

Multi-region. SFU close to users.

26. Interview Tips

- Discuss voice (WebRTC + SFU) as differentiator.
- Mention sharding by server_id.
- Discuss roles/permissions.

27. Common Follow-up Questions

- How would you implement screen sharing?
- How would you handle 1000-person voice channels (stage mode)?
- How would you implement Discord activities (games in voice)?

28. Common Mistakes

- P2P mesh for voice — doesn't scale.
- No voice state in memory — too slow.

29. Optimization Ideas

- Pre-allocate SFU capacity during peak hours.
- Batch text message fanout.
- Use Opus codec for voice (low bandwidth).

30. Final Summary

Discord tests combined text + voice chat. The architecture is two parallel systems: text (WebSocket + Cassandra) and voice (WebRTC + SFU). Sharded by server_id. Voice is the differentiator — SFU scales to large channels. The system is AP for text, real-time for voice.

QUESTION 49 INTERMEDIATE

Design GitHub

1. Problem Statement

Design a code hosting platform. Users create repositories, push code, open pull requests, review code, and track issues. Handle git protocol and large repos.

2. Functional Requirements

- Create repositories (public/private).
- Push/pull code via git protocol.
- Pull requests with code review (comments, approvals).
- Issues tracking.
- Actions (CI/CD).
- Fork and merge.

3. Non-functional Requirements

- Availability 99.95%.
- Git push/pull latency < 5s.
- Strong consistency for repo state.
- Web UI latency < 1s.

4. Capacity Estimation

100M users, 200M repos. Storage: avg repo 100MB = 20PB. Push events: 10M/day.

5. Back-of-the-envelope Math

Git objects stored in object storage (S3) with dedup. PG for metadata. Webhooks for events.

6. API Design

```
POST /repos body: {name, private} -> {repo_id}
// Git protocol: git push/pull over SSH or HTTPS
POST /repos/{id}/pulls body: {title, head, base}
POST /repos/{id}/issues body: {title, body}
POST /repos/{id}/actions (CI/CD triggers)
```

7. Database Schema

```
Table: repos (repo_id, owner_id, name, private, default_branch, created_at)
Table: branches (repo_id, name, head_commit_sha)
Table: commits (commit_sha, repo_id, parent_sha, author_id, message, created_at)
Table: pull_requests (pr_id, repo_id, head_branch, base_branch, status, ...)
Table: issues (issue_id, repo_id, title, body, status, assignee_id)
```

8. High-Level Design (HLD)

Git client → Git Server (SSH/HTTPS) → Object Storage (S3 for git objects) + PG (metadata). Web UI → API Gateway → Repo Service + PR Service + Issue Service.

9. Low-Level Design (LLD)

Git push: client pushes objects to Git Server; server stores new objects in S3 (deduped by SHA), updates branch head in PG. Webhook fires on push to trigger CI/CD.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Git push:
  1. Client `git push` over SSH/HTTPS.
  2. Git Server receives objects; dedup by SHA.
  3. Store new objects to S3; update branch head in PG.
  4. Fire webhook event (push) to subscribers.
  5. Trigger CI/CD if Actions configured.

Pull request:
  1. User opens PR via web UI.
  2. PR Service creates PR record; computes diff.
  3. Reviewers comment/approve.
  4. On merge: combine branches, update head, fire webhook.
```

12. Component Explanation

- **Git Server:** Handles git protocol; manages object storage.
- **Repo Service:** Repo metadata CRUD.
- **PR Service:** Pull request lifecycle.
- **Issue Service:** Issue tracking.
- **Actions Runner:** CI/CD execution (containers).
- **S3:** Git object storage (deduped by SHA).

13. Technology Stack

- **Language:** Ruby (GitHub original) + Go.
- **DB:** PostgreSQL + MySQL (sharded).
- **Storage:** S3 / custom object store.
- **Queue:** Kafka + Sidekiq (background jobs).
- **CI/CD:** Container runners (Kubernetes).

14. Database Choice

PG for metadata (ACID). S3 for git objects (deduped by SHA, cheap).

15. Cache Strategy

Redis for hot repo metadata, PR state.

16. Load Balancer Strategy

L7 LB. Sticky session for SSH.

17. Message Queue Usage

Kafka for push events, webhook delivery.

18. Scaling Strategy

Shard PG by repo_id. Git servers scale horizontally. CI runners auto-scale.

19. Fault Tolerance

Multi-AZ. Object storage cross-region replication.

20. Bottlenecks

- Large repo pushes — chunk + parallel.
- CI runner capacity on popular repos — auto-scale.
- Webhook fanout — queue + retry.

21. Trade-offs

- **S3 vs filesystem for objects:** S3 is scalable; filesystem is faster.
- **Sync vs async webhook:** Async is faster; sync confirms delivery.

22. CAP Theorem Analysis

CP for repo state (no lost commits).

23. Security Considerations

TLS + SSH. Per-repo permissions. Audit log. Secrets scanning. Code scanning (SAST).

24. Monitoring & Logging

Track push/pull latency, webhook delivery rate, CI runner utilization.

25. Deployment Strategy

Blue-green. CI runners in dedicated node pools.

26. Interview Tips

- Discuss git protocol integration.
- Mention object dedup by SHA.
- Discuss CI/CD runners as separate concern.

27. Common Follow-up Questions

- How would you handle monorepos with millions of files?
- How would you implement code search across all repos?
- How would you implement GitHub Copilot (AI code completion)?

28. Common Mistakes

- No object dedup — storage explodes.
- Synchronous webhook delivery — slow push.
- Single PG instance — won't scale.

29. Optimization Ideas

- Pre-compute diffs for common PR views.
- Cache popular repo metadata.
- Batch webhook delivery.

30. Final Summary

GitHub tests git protocol integration and metadata management. The core insight is separating git objects (large, dedupable, stored in S3) from metadata (small, relational, in PG). Push events trigger webhooks for CI/CD and integrations. The architecture rewards investment in CI/CD runner infrastructure and webhook delivery.

QUESTION 50 INTERMEDIATE

Design Wikipedia

1. Problem Statement

Design an online encyclopedia. Users read articles, search, edit (with version history), and discuss on talk pages. Read-heavy (1000:1 read:write).

2. Functional Requirements

- Read articles (rendered HTML).
- Search by keyword.
- Edit articles (with version history).
- Talk pages for discussion.
- Multi-language support.

3. Non-functional Requirements

- Availability 99.95%.
- Read latency < 500ms.
- Search latency < 1s.
- Read-heavy (1000:1).

4. Capacity Estimation

50M articles, 2B monthly readers. Reads: 10B/month. Edits: 10M/month. Storage: 50M × 50KB = 2.5TB.

5. Back-of-the-envelope Math

Extreme read-heavy — aggressive caching (CDN + Redis + edge). Edits are rare.

6. API Design

```
GET /articles/{title} -> rendered HTML
GET /search?q=...
POST /articles/{title}/edits body: {content, summary}
GET /articles/{title}/history -> [versions]
POST /articles/{title}/talk body: {text}
```

7. Database Schema

```
Table: articles (article_id, title, current_version, language, ...)
Table: article_versions (version_id, article_id, content, editor_id, summary, created_at)
Table: talk_pages (article_id, user_id, text, created_at)
Table: search_index (article_id, title, content_text, language)
```

8. High-Level Design (HLD)

Browser → CDN (aggressive caching) → API Gateway → Article Service + Search Service (ES). PG for articles and version history.

9. Low-Level Design (LLD)

On edit: write new version to PG (immutable). Update current_version pointer. Invalidate CDN cache for that article. Render HTML on first read after edit; cache.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Read article:
  1. Browser GET /articles/{title}.
  2. CDN serves cached HTML if available (95% hit rate).
  3. On miss: Article Service fetches from PG, renders HTML, caches.

Edit article:
  1. User submits edit; Article Service writes new version to PG.
  2. Invalidate CDN + Redis cache for article.
  3. Async: re-index in ES.
  4. Notify watchers via email.
```

12. Component Explanation

- **Article Service:** CRUD + rendering.
- **Search Service:** Elasticsearch.
- **CDN:** Aggressive HTML caching.
- **Redis:** In-process + cache for hot articles.

13. Technology Stack

- **Language:** PHP (Wikipedia original) or Go.
- **DB:** PostgreSQL + MariaDB.

- **Cache:** Redis + Varnish (HTTP cache).
- **Search:** Elasticsearch.
- **CDN:** Cloudflare/Akamai.

14. Database Choice

PG/MariaDB for article versions. ES for search. Redis + CDN for reads.

15. Cache Strategy

CDN (95% hit rate) + Redis (4%) + origin (1%).

16. Load Balancer Strategy

L7 LB. Geographic routing.

17. Message Queue Usage

Kafka for edit events (search re-index, notifications).

18. Scaling Strategy

CDN absorbs most reads. Origin scales modestly. ES scales search.

19. Fault Tolerance

Multi-AZ. CDN failover.

20. Bottlenecks

- Edit contention on popular articles — optimistic locking.

21. Trade-offs

- **Cache invalidation vs TTL:** Invalidation is precise; TTL is simpler.

22. CAP Theorem Analysis

AP for reads (cached). CP for edits (version history).

23. Security Considerations

TLS. Anti-vandalism (rate limit, auto-revert). Auth for edits.

24. Monitoring & Logging

Track cache hit ratio, read latency, edit latency.

25. Deployment Strategy

Multi-region. CDN distributed globally.

26. Interview Tips

- Emphasize extreme caching — 1000:1 read:write.
- Discuss CDN as primary read path.
- Mention version history (immutable versions).

27. Common Follow-up Questions

- How would you handle edit conflicts (concurrent edits)?
- How would you implement "random article" efficiently?
- How would you handle multi-language linking?

28. Common Mistakes

- No CDN — origin crushed.
- Synchronous rendering on every read.
- No version history — lost edits.

29. Optimization Ideas

- Pre-render HTML on edit.
- Edge-cache at CDN for 1h.
- Compress article content (gzip).

30. Final Summary

Wikipedia tests extreme read-heavy caching. The core architecture: CDN absorbs 95% of reads, Redis handles 4%, origin handles 1%. Edits are rare but write a new immutable version (history). Search via ES. The system is AP for reads, CP for edits. The architecture rewards aggressive caching investment.

PART V

Advanced Questions (Q51 – Q80)

Distributed systems and infrastructure for Senior SWE.

QUESTION 51

ADVANCED

Design Google Search

1. Problem Statement

Design a web search engine. Crawl the web, index pages, and answer user queries in < 500ms with relevant results ranked by PageRank + signals.

2. Functional Requirements

- Crawl web pages continuously; discover new + updated content.
- Index pages (text, links, metadata) for fast retrieval.
- Answer user queries with ranked results in < 500ms.
- Personalize results (location, history).
- Handle 8B+ queries/day globally.

3. Non-functional Requirements

- Query latency < 500ms.
- Availability 99.99%.
- Index freshness: pages updated within 24h.
- Crawl scalability: 100B+ pages.

4. Capacity Estimation

100B pages indexed, 8B queries/day. Index size: 100B × 5KB text = 500TB. QPS: 100K avg, 1M peak. Crawl: 10B pages/day = 100K pages/s.

5. Back-of-the-envelope Math

Inverted index: word → list of (doc_id, position). Index sharded by word hash. PageRank: pre-computed via iterative algorithm. Query: parse → fetch inverted lists → intersect → rank.

6. API Design

```
GET /search?q=...&page=1&location=... -> [results]

// Internal crawl pipeline:
// URL Frontier (priority queue) -> Crawler -> Parser -> Indexer -> Sharded Index
```

7. Database Schema

```
// Index storage (custom, not SQL):
InvertedIndex {
  term_hash: 64-bit
  postings: [{doc_id, positions, tf, pagerank}] // compressed
}

// Metadata DB (Postgres/Cassandra):
Table: pages (url, doc_id, last_crawled, pagerank, language, ...)
Table: links (src_doc_id, dest_doc_id, anchor_text) -- for PageRank
```

8. High-Level Design (HLD)

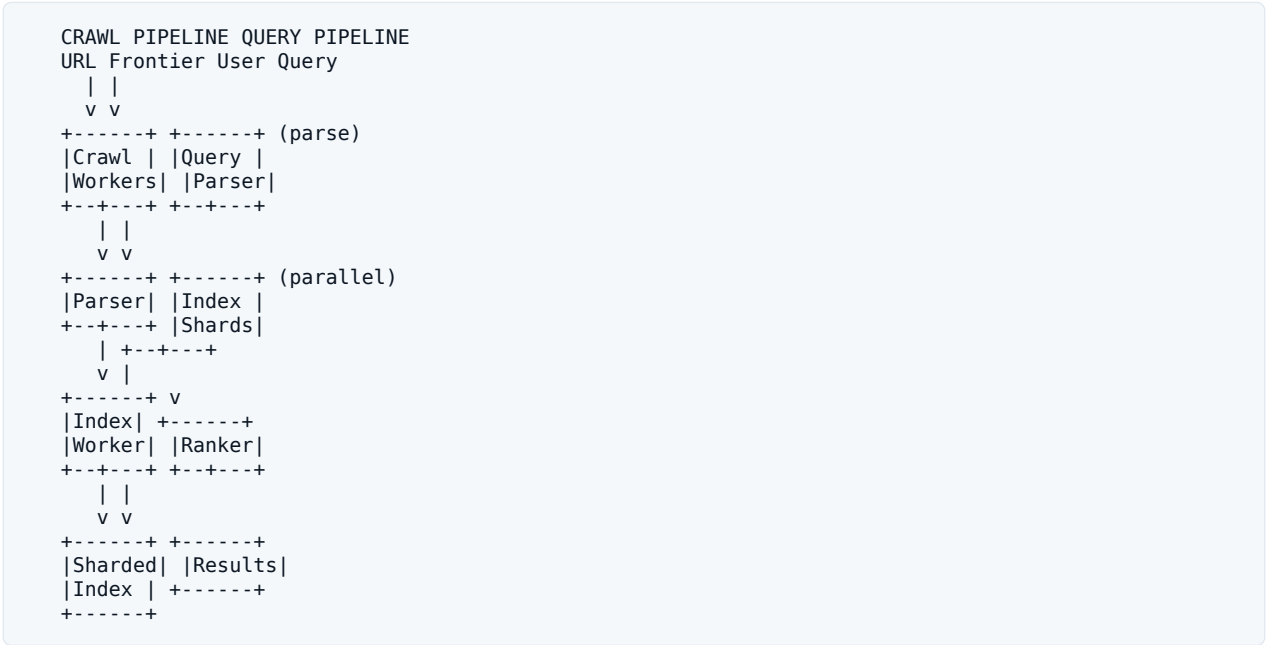
Crawler (distributed) → URL Frontier → Parser → Indexer → Sharded Inverted Index (Bigtable/SSTable). Query: Frontend → Query Parser → Index Shards (parallel) → Ranker → Results.

9. Low-Level Design (LLD)

Crawler: distributed workers with politeness (per-domain rate limit). Indexer: builds inverted index segments, merges periodically. Query: parse to terms, fetch postings from each shard in parallel, intersect, rank by PageRank + 200+ signals.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Crawl + index:
  1. URL Frontier prioritizes URLs (PageRank, freshness).
  2. Crawler fetches HTML; parser extracts text + links.
  3. New links added to Frontier.
  4. Indexer builds inverted index segment; periodically merges.
  5. PageRank recomputed nightly (iterative map-reduce).

Query:
  1. User submits query; frontend routes to nearest region.
  2. Query parser: tokenize, stem, expand synonyms.
  3. Fetch postings for each term from index shards (parallel).
  4. Intersect postings; rank by PageRank + relevance signals.
  5. Return top 10 results with snippets.
```

12. Component Explanation

- **URL Frontier:** Priority queue of URLs to crawl.
- **Crawler:** Distributed workers; fetch + parse pages.
- **Indexer:** Builds inverted index segments; merges.
- **Index Shards:** Distributed inverted index (term → postings).
- **PageRank Computer:** Nightly map-reduce to compute PageRank.
- **Query Frontend:** Receives queries; routes; returns results.
- **Ranker:** Combines PageRank + 200+ signals.

13. Technology Stack

- **Language:** C++ for performance-critical; Java/Python for orchestration.
- **Storage:** Bigtable / SSTable for index; S3 for raw HTML.
- **Compute:** MapReduce (Hadoop/Spark) for batch.
- **Cache:** Redis for query result cache.
- **Deployment:** Multi-region data centers.

14. Database Choice

Custom inverted index storage (Bigtable/SSTable) — relational DBs cannot handle the scale or access pattern.

15. Cache Strategy

Redis for query result cache (top 1M queries). CDN for static results.

16. Load Balancer Strategy

Geographic routing to nearest data center.

17. Message Queue Usage

Kafka for crawl events, index updates.

18. Scaling Strategy

Index shards scale horizontally. Crawler scales by worker count.

19. Fault Tolerance

Multi-datacenter. Index shards replicated.

20. Bottlenecks

- Index size (500TB) — shard by term hash.
- Crawl politeness — per-domain rate limit.
- Rank computation — nightly batch + cached results.

21. Trade-offs

- **Freshness vs scale:** Frequent crawling = fresher but more bandwidth.
- **Personalization vs privacy:** Personalized results use history.
- **PageRank vs ML:** PageRank is interpretable; ML is more accurate.

22. CAP Theorem Analysis

AP. Index is eventually consistent (24h lag).

23. Security Considerations

TLS. Anti-SEO-spam (detect link farms). Rate limit queries.

24. Monitoring & Logging

Track query latency, index freshness, crawl rate, ranking quality.

25. Deployment Strategy

Multi-region data centers with local index copies.

26. Interview Tips

- Discuss inverted index architecture explicitly.
- Mention PageRank + 200+ ranking signals.
- Calculate index size: 100B pages × 5KB = 500TB — sharding required.
- Discuss crawl politeness.

27. Common Follow-up Questions

- How would you handle image search?
- How would you implement "did you mean" spell correction?
- How would you detect and penalize SEO spam?
- How would you implement real-time search (Twitter results)?
- How would you add LLM-based answers (Google AI Overviews)?

28. Common Mistakes

- PG for index — cannot scale to 500TB.
- No crawl politeness — DDOS websites.
- Synchronous ranking — too slow.

- No query cache — repeat queries re-ranked.

29. Optimization Ideas

- Cache top 1M query results in Redis.
- Pre-compute PageRank nightly.
- Compress postings lists (delta encoding).
- Shard index by term hash for parallel query.

30. Final Summary

Google Search is the canonical large-scale system design. The core architecture has two pipelines: crawl (URL frontier → crawler → parser → indexer) and query (parse → fetch from sharded inverted index → rank). The inverted index is the central data structure — term → list of (doc_id, position). Index is sharded by term hash for parallel query. Ranking combines PageRank (pre-computed nightly) with 200+ real-time signals. The system is AP with 24h freshness lag. The architecture rewards deep investment in index compression, sharding, and ranking ML.

QUESTION 52 ADVANCED

Design a Distributed Cache (Redis Cluster)

1. Problem Statement

Design a distributed in-memory cache. Clients set/get key-value pairs. The cache shards across nodes, survives node failures, and provides sub-millisecond latency.

2. Functional Requirements

- GET/SET/DELETE key-value pairs.
- Automatic sharding across nodes.
- Replication for HA (1 primary + N replicas per shard).
- Automatic failover on node failure.
- TTL support.
- Atomic operations (INCR, GETSET).

3. Non-functional Requirements

- Latency: p99 < 1ms.
- Availability 99.99%.
- Survives node failures with < 5s unavailability.
- Linear scalability (add nodes = more capacity).

4. Capacity Estimation

10TB cache data. 1M ops/sec. Sharded across 100 nodes (100GB each).

5. Back-of-the-envelope Math

Consistent hashing for shard assignment. 16,384 hash slots (Redis Cluster). Each node owns a range of slots. Replication: 1 primary + 2 replicas per shard.

6. API Design

```
// Client API (using consistent hashing client):
SET key value [EX seconds]
GET key
DEL key
INCR key
// Cluster management:
CLUSTER NODES, CLUSTER INFO, CLUSTER FAILOVER
```

7. Database Schema

```
// In-memory data structure per node:
HashTable {
  key: string
  value: bytes
  expires_at: timestamp (optional)
}

// Slot ownership (gossip protocol):
SlotMap { slot_id: node_id }
```

8. High-Level Design (HLD)

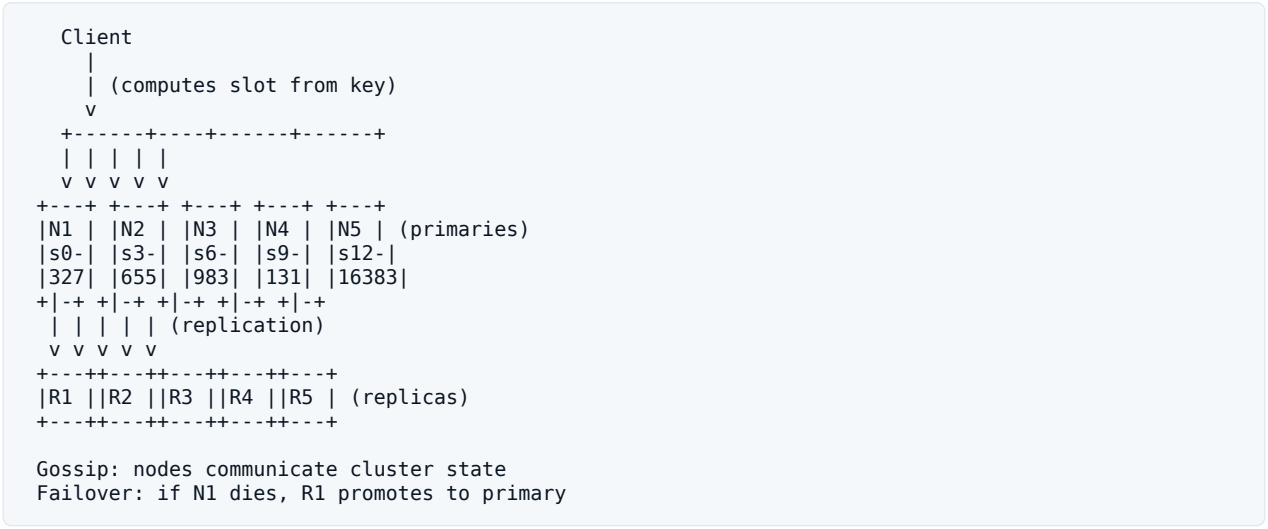
Client (with smart routing) → Cluster of Nodes (each owns hash slots). Gossip protocol for cluster membership. Sentinel/Cluster for failover.

9. Low-Level Design (LLD)

Client computes CRC16(key) % 16384 = slot_id. Looks up slot → node mapping (cached locally). Sends request directly to owning node. If node moved (cluster rebalancing), server returns MOVED/ASK redirect.

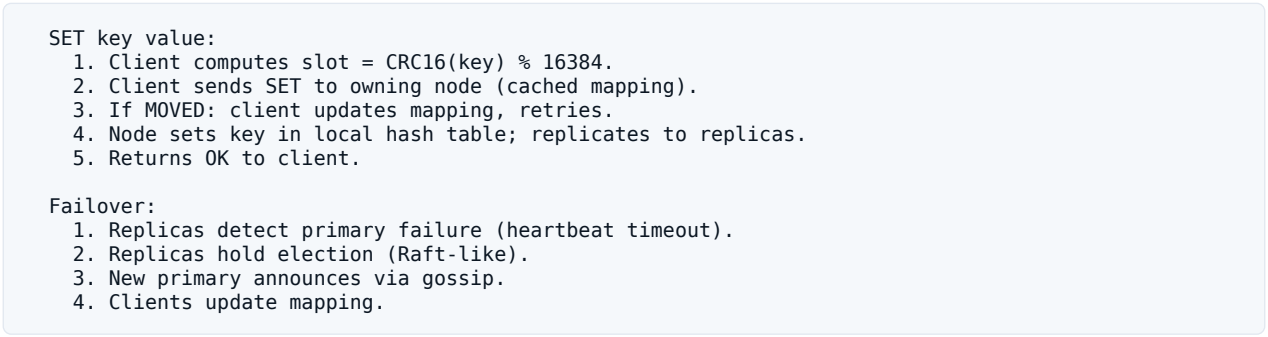
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Client Library:** Smart routing; computes slot; caches mapping.
- **Cache Node:** In-memory hash table; replication; gossip.
- **Gossip Protocol:** Cluster membership + slot ownership propagation.
- **Failover Coordinator:** Election of new primary on failure.

13. Technology Stack

- **Language:** C/C++ (Redis original) for low-latency.
- **Network:** Custom TCP protocol (RESP).
- **Algorithm:** Consistent hashing with 16384 virtual slots.
- **Consensus:** Raft-like for failover election.

14. Database Choice

In-memory only (no DB). Optional: AOF + RDB for persistence (recovery after restart).

15. Cache Strategy

This IS the cache. LRU eviction per shard when memory full.

16. Load Balancer Strategy

Client-side LB (smart routing via consistent hashing).

17. Message Queue Usage

Optional: pub/sub for cache invalidation messages.

18. Scaling Strategy

Add nodes → rebalance slots (migrate slot range to new node). Online operation.

19. Fault Tolerance

Replication (1 primary + 2 replicas per shard). Automatic failover via Raft election.

20. Bottlenecks

- Hot keys (single shard saturated) — client-side caching or read replicas.
- Rebalancing cost — migrate slots in small batches.
- Memory limit per node — add more nodes.

21. Trade-offs

- **Strong vs eventual consistency:** Reads from primary (strong) or replicas (eventual).
- **Persistence vs performance:** AOF adds latency on writes; no persistence risks data loss.
- **Sync vs async replication:** Sync is consistent; async is fast.

22. CAP Theorem Analysis

AP. Survives network partition with eventual consistency. Replica promotion may cause brief inconsistency.

23. Security Considerations

TLS (optional). AUTH password. Network isolation (private VPC).

24. Monitoring & Logging

Track ops/sec, latency, memory usage, replication lag, failover events.

25. Deployment Strategy

Multi-AZ. Nodes distributed across AZs for HA.

26. Interview Tips

- Discuss consistent hashing with virtual slots (16384).
- Mention MOVED/ASK redirect for cluster rebalancing.
- Discuss replication + failover (Raft-like).
- Calculate shard count based on data size + QPS.

27. Common Follow-up Questions

- How would you handle hot keys (single shard saturated)?
- How would you implement distributed locking (Redlock)?
- How would you handle a network split (split-brain)?
- How would you migrate from single Redis to cluster with zero downtime?

28. Common Mistakes

- Modulo hashing — adding node requires rehashing everything.
- No replication — node failure loses data.
- Synchronous replication — kills write throughput.
- No failover plan — manual recovery on failure.

29. Optimization Ideas

- Pipelining (batch multiple commands).
- Connection pooling.
- Client-side caching (cache aside pattern).
- Compress values for large data.

30. Final Summary

A distributed cache tests consistent hashing, replication, and failover. The core architecture: 16,384 hash slots distributed across nodes (Redis Cluster pattern), each shard has 1 primary + N replicas, gossip protocol for cluster membership, Raft-like election for failover. Client-side smart routing computes slot from key. The system is AP — brief inconsistency during failover is acceptable. The architecture rewards deep understanding of consistent hashing and consensus.

QUESTION 53

ADVANCED

Design Kafka

1. Problem Statement

Design a distributed publish-subscribe messaging system. Producers write messages to topics; consumers read in order. Topics are partitioned for parallelism; messages are replicated for durability.

2. Functional Requirements

- Producers write messages to topics.
- Consumers read messages in order per partition.
- Topics partitioned for parallelism.
- Replication for durability (RF=3).
- Consumer groups for parallel consumption.
- Message retention (days/weeks).

3. Non-functional Requirements

- Throughput: millions of messages/sec.
- Latency: < 10ms per message.
- Availability 99.99%.
- Durability: no message loss on node failure.

4. Capacity Estimation

10M msgs/sec sustained. Topics: 100K. Partitions per topic: 100. Retention: 7 days. Storage: 10M × 1KB × 86400 × 7 = 6TB/day = 42TB/week.

5. Back-of-the-envelope Math

Partition = append-only log on disk. Replica = copy on another broker. Consumer offset tracked per partition. ISR (In-Sync Replicas) for commit.

6. API Design

```
// Producer API:
producer.send(topic, key, value)

// Consumer API:
consumer.subscribe(topic)
for msg in consumer.poll():
    process(msg)
    consumer.commit(offset)

// Admin API:
admin.createTopic(topic, partitions=100, replication_factor=3)
```

7. Database Schema

```
// Per-partition on-disk log:
LogSegment {
  base_offset: int64
  messages: [{offset, key, value, timestamp, ...}] // appended sequentially
}

// Broker state (in-memory + ZooKeeper/KRaft):
PartitionState {
  topic, partition_id, leader_broker, isr: [broker_ids],
  high_watermark: int64 // last committed offset
}
```

8. High-Level Design (HLD)

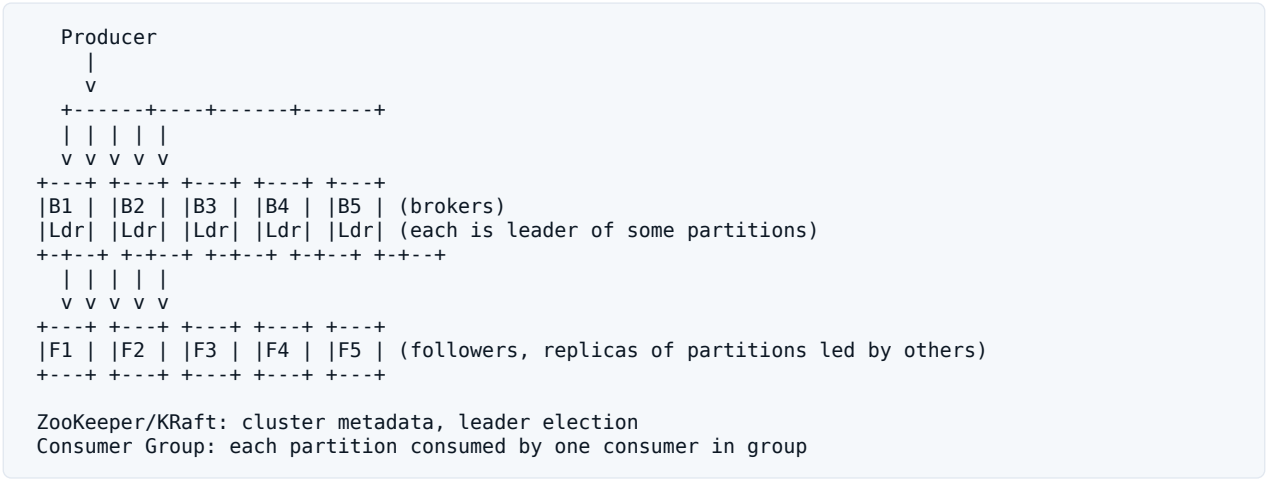
Producer → Broker (partition leader) → Replicas (followers). Consumer reads from leader. ZooKeeper/KRaft for cluster metadata and leader election.

9. Low-Level Design (LLD)

Producer: pick partition (round-robin or key hash), send to partition leader. Broker: append to log, replicate to ISR, ack based on acks setting (0/1/all). Consumer: read from leader, commit offset to internal topic.

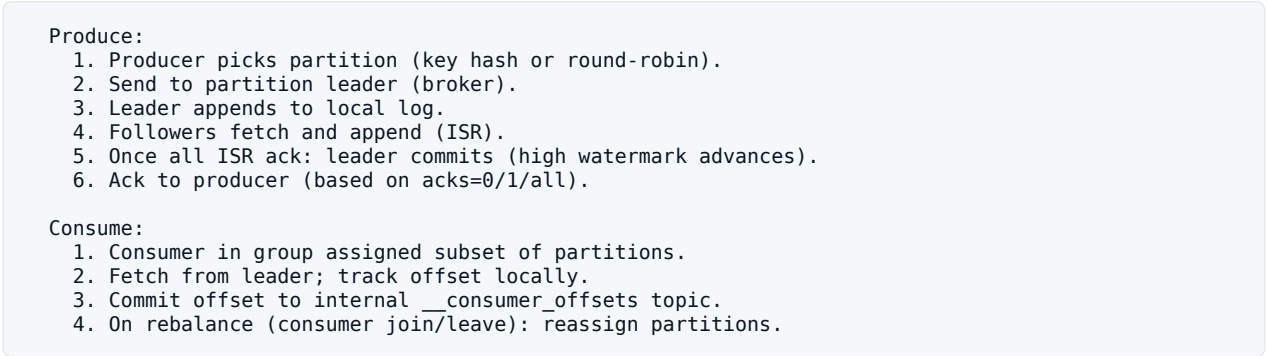
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Broker:** Stores partition logs; serves produce/consume.
- **ZooKeeper / KRaft:** Cluster metadata; leader election.
- **Producer Client:** Picks partition; sends to leader; retries.
- **Consumer Client:** Group membership; fetches; commits offsets.
- **Controller:** Special broker that manages partition leadership.

13. Technology Stack

- **Language:** Scala/Java (Kafka original).
- **Storage:** Custom append-only log files on disk.
- **Coordination:** ZooKeeper or KRaft (built-in Raft).
- **Network:** Custom binary protocol.

14. Database Choice

Custom log files on disk (sequential writes = fast). No DB.

15. Cache Strategy

OS page cache (Kafka relies on it for fast reads). No application-level cache.

16. Load Balancer Strategy

Producer routes by partition (key hash). Consumer group assignment via coordinator.

17. Message Queue Usage

This IS the message queue.

18. Scaling Strategy

Add brokers; reassign partitions. Partition count is fixed at creation (cannot easily change).

19. Fault Tolerance

Replication (RF=3). ISR-based commit ensures durability. Leader failover via controller on broker death.

20. Bottlenecks

- Hot partitions (key skew) — use better partitioning key.
- Consumer lag — add consumers to group (up to partition count).
- Disk I/O — SSDs; sequential writes only.

21. Trade-offs

- **acks=0/1/all:** 0 = fastest, no durability; all = slowest, full durability.
- **Partition count:** More = parallelism, but more metadata overhead.
- **Retention:** Longer = more storage; shorter = less replay.

22. CAP Theorem Analysis

AP during partition. ISR-based commit ensures committed messages are durable. Uncommitted may be lost on leader failure.

23. Security Considerations

TLS. SASL authentication. ACLs per topic. Audit log.

24. Monitoring & Logging

Track produce/consume throughput, consumer lag, ISR size, broker disk usage.

25. Deployment Strategy

Multi-AZ brokers. RF=3 across AZs for HA.

26. Interview Tips

- Discuss partition log architecture explicitly.
- Mention ISR (In-Sync Replicas) for commit.
- Discuss consumer groups and offset tracking.
- Mention ZooKeeper/KRaft for metadata.

27. Common Follow-up Questions

- How would you handle exactly-once semantics?
- How would you implement schema registry?
- How would you handle a hot partition?
- How would you migrate ZooKeeper-based cluster to KRaft?
- How would you implement Kafka Streams (stream processing)?

28. Common Mistakes

- Random partitioning — loses key ordering.
- acks=1 in production — message loss on leader failover.
- Too few partitions — limits consumer parallelism.
- No retention limit — disk fills up.

29. Optimization Ideas

- Batch produces (linger.ms).
- Compress messages (snappy/lz4).
- Use SSDs for high-throughput topics.
- Tune OS page cache settings.

30. Final Summary

Kafka tests distributed log architecture. The core insight: each partition is an append-only log on disk (sequential writes = fast). Replication via ISR (In-Sync Replicas) ensures durability. Consumer groups enable parallel consumption (one consumer per partition per group). Offsets tracked in internal topic. The system achieves millions of msgs/sec via sequential disk I/O, OS page cache, and batched network calls. The architecture rewards

deep understanding of distributed logs and replication protocols.

QUESTION 54 ADVANCED

Design Redis (In-Memory Data Store)

1. Problem Statement

Design an in-memory key-value data store supporting multiple data structures (strings, lists, sets, sorted sets, hashes), persistence options, and replication.

2. Functional Requirements

- GET/SET strings with TTL.
- List operations (LPUSH, RPUSH, LRANGE).
- Set operations (SADD, SMEMBERS, SINTER).
- Sorted set (ZADD, ZRANGE).
- Hash operations (HSET, HGET).
- Pub/sub.
- Persistence (RDB snapshots, AOF).

3. Non-functional Requirements

- Latency: < 1ms per command.
- Throughput: 100K+ commands/sec per instance.
- Single-threaded command execution (no locks).
- Persistence without blocking commands.

4. Capacity Estimation

Single instance: 256GB RAM. 1M ops/sec. Cluster: 1TB+ across nodes.

5. Back-of-the-envelope Math

Single-threaded event loop (no locks). In-memory hash table. Persistence via fork (COW) for snapshot; append log for AOF.

6. API Design

```
SET key value [EX seconds] [NX|XX]
GET key
LPUSH list value
ZADD set score member
HSET hash field value
EXPIRE key seconds
PERSIST key
SUBSCRIBE channel
```

7. Database Schema

```
// In-memory data structures:
redisDb {
  dict: hash table {key -> redisObject}
  expires: hash table {key -> timestamp}
}

redisObject {
  type: STRING|LIST|SET|ZSET|HASH
  encoding: raw|int|embstr|ziplist|skiplist|...
  value: ...
  lru: clock (for eviction)
}
```

8. High-Level Design (HLD)

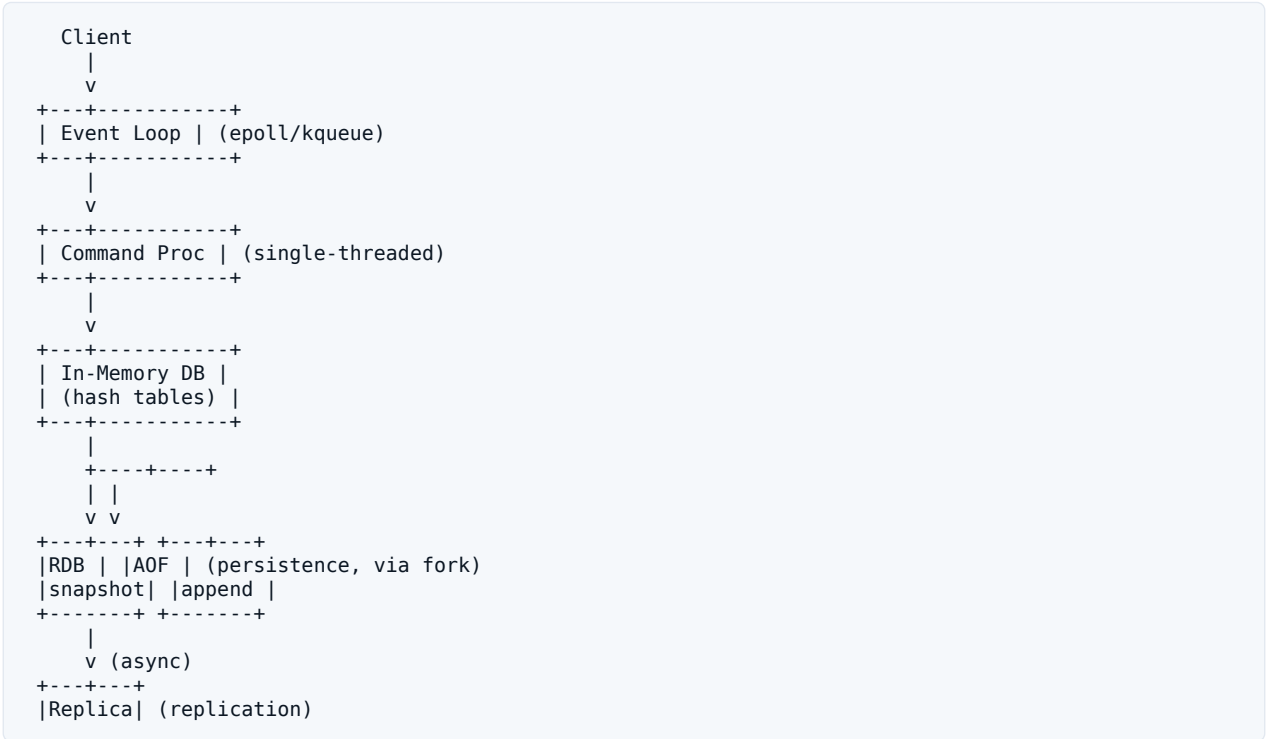
Single-threaded event loop: accept connection → parse command → execute → reply. Persistence: fork process for RDB; append-only file for AOF. Replication: async to replicas.

9. Low-Level Design (LLD)

Event loop uses epoll/kqueue for non-blocking I/O. Single command thread (no locks needed). Data structures: hash table (strings), ziplist (small lists/hashes), skiplist (sorted sets). Eviction: LRU or LFU when maxmemory exceeded.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Command execution:
  1. Client sends command via TCP (RESP protocol).
  2. Event loop accepts connection; reads command.
  3. Command processor parses; looks up in command table.
  4. Executes against in-memory db (single-threaded, no locks).
  5. Returns reply to client.
  6. If write command: append to AOF buffer; replicate to replicas.

RDB snapshot:
  1. Fork process (COW).
  2. Child serializes db to disk.
  3. Parent continues serving.
```

12. Component Explanation

- **Event Loop:** epoll/kqueue-based; non-blocking I/O.
- **Command Processor:** Single-threaded; looks up + executes.
- **In-Memory DB:** Hash tables per data type.
- **Persistence:** RDB (fork snapshot) + AOF (append log).
- **Replication:** Async to replicas; replica can serve reads.

13. Technology Stack

- **Language:** C (Redis original).
- **Network:** Custom RESP protocol over TCP.
- **I/O:** epoll (Linux), kqueue (BSD).
- **Algorithm:** Hash table with incremental rehashing.

14. Database Choice

In-memory only. Optional persistence (RDB + AOF) for recovery.

15. Cache Strategy

This IS the cache. LRU/LFU eviction when maxmemory exceeded.

16. Load Balancer Strategy

Client-side routing in cluster mode (consistent hashing).

17. Message Queue Usage

Pub/sub built-in (no persistence). Kafka for persistent messaging.

18. Scaling Strategy

Vertical (bigger instance) up to single-node limit. Horizontal via Redis Cluster (sharded).

19. Fault Tolerance

RDB + AOF for restart recovery. Replication for HA. Sentinel for failover.

20. Bottlenecks

- Single-threaded CPU — cluster for parallelism.
- Memory limit — sharding for larger datasets.
- Fork latency on large db — use AOF primarily.

21. Trade-offs

- **RDB vs AOF:** RDB is compact; AOF is durable. Both is best.
- **Single vs multi-threaded:** Single is simpler; multi is faster but complex.
- **Sync vs async replication:** Sync is durable; async is fast.

22. CAP Theorem Analysis

AP. Single-node is CA (no partition scenario). Cluster is AP (replica promotion).

23. Security Considerations

TLS (Redis 6+). AUTH password. ACLs (Redis 6+). Network isolation.

24. Monitoring & Logging

Track commands/sec, latency, memory usage, eviction rate, replication lag.

25. Deployment Strategy

Multi-AZ with replicas. Sentinel for HA.

26. Interview Tips

- Discuss single-threaded event loop — key design decision.
- Mention RDB (fork COW) vs AOF.
- Discuss data structure encodings (ziplist for small, skiplist for sorted sets).
- Discuss LRU vs LFU eviction.

27. Common Follow-up Questions

- How would you implement Redis Cluster (sharding)?
- How would you handle a 1TB dataset (bigger than RAM)?
- How would you implement transactions (MULTI/EXEC)?
- How would you add Lua scripting?
- How would you implement modules (RedisJSON, RedisGraph)?

28. Common Mistakes

- Multi-threaded with locks — slower than single-threaded event loop.
- No persistence — data lost on crash.
- Synchronous replication — kills write throughput.
- No eviction policy — OOM crash.

29. Optimization Ideas

- Pipeline commands (batch).
- Use appropriate encoding (ziplist for small structures).
- Tune maxmemory + eviction policy.
- Use SSD for AOF (faster fsync).

30. Final Summary

Redis tests in-memory data structure design. The core insight: single-threaded event loop with non-blocking I/O is faster than multi-threaded with locks for in-memory workloads (no contention). Multiple data structures (strings, lists, sets, sorted sets, hashes) with smart encodings (ziplist for small, skiplist for sorted). Persistence via fork (COW) for RDB; append log for AOF. Replication async to replicas. The architecture rewards minimalism and correctness.

QUESTION 55 ADVANCED

Design a Distributed File System (HDFS/GFS)

1. Problem Statement

Design a distributed file system for large files (TB+). Files split into chunks, replicated across nodes. Support high-throughput reads/writes for analytics workloads.

2. Functional Requirements

- Create/delete files and directories.
- Read/write large files (chunked).
- Append-only writes (HDFS) or random writes (GFS).
- Replication for durability (default RF=3).
- List directories, get file metadata.

3. Non-functional Requirements

- Throughput: GB/s per cluster.
- Availability 99.9% (failure is common at scale).
- Durability: no data loss on node failure.
- Optimized for large sequential reads/writes.

4. Capacity Estimation

10PB cluster. 100K chunks (64MB each) per node. 1000 data nodes. File count: 100M.

5. Back-of-the-envelope Math

Metadata in NameNode (RAM). Data in chunks on DataNodes (disk). Client reads/writes directly to DataNodes (offload from NameNode).

6. API Design

```
// Client API:
create(path) -> file handle
open(path) -> file handle
read(handle, offset, length) -> bytes
write(handle, bytes)
append(handle, bytes)
delete(path)
```

7. Database Schema

```
// NameNode metadata (in-memory + journal):
FileSystem {
  files: {path -> FileMeta}
  blocks: {block_id -> [DataNode locations]}
}

FileMeta { size, blocks: [block_id], replication, ... }

// DataNode on-disk:
block_id -> bytes (raw chunk data)
```

8. High-Level Design (HLD)

Client → NameNode (metadata ops) + DataNodes (data transfer). NameNode tracks file tree + block locations. DataNodes store blocks + send heartbeats.

9. Low-Level Design (LLD)

Write: client asks NameNode for new block ID + target DataNodes; client writes to first DataNode, which pipes to next (replication chain). On completion, NameNode commits. Read: client asks NameNode for block locations; reads directly from nearest DataNode.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Write file:
  1. Client calls create(path); NameNode allocates new file.
  2. Client flushes 64MB chunks. For each: NameNode assigns block_id + 3 DataNodes.
  3. Client writes to DataNode 1; DataNode 1 pipes to DataNode 2; etc.
  4. All 3 acknowledge; NameNode commits block.
  5. On file close: NameNode marks file complete.

Read file:
  1. Client opens file; NameNode returns block list with locations.
  2. Client reads each block from nearest DataNode (network topology-aware).
  3. Data flows directly DataNode → Client.
```

12. Component Explanation

- **NameNode:** Metadata server; in-memory file tree + block map.
- **DataNode:** Stores blocks; sends heartbeats + block reports.
- **Secondary NameNode:** Periodic checkpoint of NameNode state (not HA).
- **Journal Nodes:** HA NameNode shared edit log (3 nodes).
- **Client Library:** Talks to NameNode for metadata; DataNodes for data.

13. Technology Stack

- **Language:** Java (HDFS) or C++ (GFS).
- **Storage:** Local filesystem on DataNodes.
- **Coordination:** ZooKeeper for NameNode failover.
- **Network:** TCP with custom protocol.

14. Database Choice

Custom (NameNode for metadata, DataNode for data). No relational DB.

15. Cache Strategy

OS page cache on DataNodes. Client-side block cache.

16. Load Balancer Strategy

Client picks nearest DataNode for reads (rack awareness).

17. Message Queue Usage

Optional: Kafka for audit events.

18. Scaling Strategy

Add DataNodes for capacity; rebalance blocks. NameNode RAM limits file count.

19. Fault Tolerance

Replication (RF=3). NameNode HA via standby + journal nodes. Auto-replication if DataNode dies.

20. Bottlenecks

- NameNode RAM (file count limit) — Federation (multiple NameNodes).
- Small files (each takes metadata entry) — HAR files, sequence files.
- Replication bandwidth — async replication, throttling.

21. Trade-offs

- **NameNode HA vs simplicity:** HA is complex; single NameNode is simpler but SPOF.
- **Block size:** Larger = better throughput but more wasted space for small files.
- **Replication vs erasure coding:** RF=3 = 3x space; EC = 1.5x space but slower.

22. CAP Theorem Analysis

CP. NameNode is source of truth. During partition, writes blocked.

23. Security Considerations

TLS (HDFS). Kerberos authentication. ACLs per file. Audit log.

24. Monitoring & Logging

Track DataNode heartbeats, block under-replication, NameNode heap usage.

25. Deployment Strategy

Multi-rack for rack-aware replication. NameNode HA via journal nodes.

26. Interview Tips

- Discuss separation of metadata (NameNode) from data (DataNode).
- Mention block replication chain (client writes to first, pipes to rest).
- Discuss rack-aware placement (block on 2 racks).
- Mention NameNode HA.

27. Common Follow-up Questions

- How would you handle small files (each < 64MB)?
- How would you implement erasure coding for storage savings?
- How would you handle NameNode failover?
- How would you implement HDFS snapshots for backup?

28. Common Mistakes

- Sending data through NameNode — bottleneck.
- No replication — data loss on node failure.
- Single NameNode without HA — SPOF.
- Small file problem — metadata explosion.

29. Optimization Ideas

- Use erasure coding instead of RF=3 (saves 50% space).
- Cache hot blocks on SSDs.
- Coalesce small files (HAR files).
- Federation for very large clusters (multiple NameNodes).

30. Final Summary

A distributed file system tests metadata/data separation at scale. The core architecture: NameNode holds metadata in RAM (file tree, block locations); DataNodes store blocks on disk. Client reads/writes directly to DataNodes, bypassing NameNode (no bottleneck). Replication via pipeline (client → DN1 → DN2 → DN3). Rack-aware placement (block on 2 racks for rack-failure tolerance). NameNode HA via journal nodes. The system is CP — metadata consistency is critical. The architecture rewards understanding of failure modes at scale.

QUESTION 56 ADVANCED

Design the Kubernetes Scheduler

1. Problem Statement

Design the component that assigns Pods to Nodes in a Kubernetes cluster. Each Pod has resource requests, node selectors, affinity rules. Scheduler picks the best node and binds the Pod to it.

2. Functional Requirements

- Watch for unscheduled Pods (via API server).
- Filter nodes that meet Pod requirements.
- Score filtered nodes (priority functions).
- Bind Pod to highest-scoring node.
- Respect affinity/anti-affinity, taints/tolerations.

3. Non-functional Requirements

- Schedule latency < 1s for most Pods.
- Cluster size: 5000 nodes, 150K Pods.
- Availability 99.9% (HA scheduler).
- No double-scheduling.

4. Capacity Estimation

5000 nodes, 150K Pods, 30 Pods/sec scheduling rate.

5. Back-of-the-envelope Math

Two-phase: Filter (eliminate infeasible nodes) → Score (rank feasible nodes). Atomic bind via API server optimistic concurrency.

6. API Design

```
// Watch API (informer):
GET /api/v1/pods?watch=true&fieldSelector=spec.nodeName=

// Bind API:
POST /api/v1/pods/{name}/binding body: {target_node: "node1"}
// Optimistic concurrency: fails if Pod modified since watch
```

7. Database Schema

```
// Pod spec (relevant fields):
Pod {
  spec: {
    containers: [{resources: {requests: {cpu, memory}}}],
    nodeName: "", // empty = unscheduled
    nodeSelector: {disktype: "ssd"},
    affinity: {...},
    tolerations: [...],
    priority: int
  }
}
```

8. High-Level Design (HLD)

Scheduler watches API server for unscheduled Pods. For each: Filter phase (predicate functions) → Score phase (priority functions) → Bind via API server. Cache of node info for fast decisions.

9. Low-Level Design (LLD)

Filter predicates: PodFitsResources, PodFitsHost, PodSelectorMatches, PodToleratesTaints. Scoring priorities: LeastRequestedPriority, BalancedResourceAllocation, AntiAffinityPriority. Bind: POST /binding with optimistic concurrency (resourceVersion).

10. Architecture Diagram

Architecture Diagram

```
   API Server
     ^
     | watch (unscheduled pods)
     v
+---+-----------+
| Scheduler |
+---+-----------+
     |
     +-- 1. Filter (predicates): remove infeasible nodes
     +-- 2. Score (priorities): rank feasible nodes
     +-- 3. Bind: POST /pods/{name}/binding
     |
     v
+---+-----------+
| Node Cache | (cached from API server)
+---------------+
  Nodes: [node1: {cpu, memory, pods, ...}, ...]
```

11. Data Flow Diagram

Data Flow Diagram

```
 Schedule Pod:
   1. Scheduler receives Pod via watch (spec.nodeName empty).
   2. Filter: run predicates against all nodes. Eliminate infeasible.
   3. Score: run priorities on remaining nodes. Rank.
   4. Pick top node.
   5. Bind: POST /pods/{name}/binding with target node.
   6. API server validates (optimistic concurrency); updates Pod.spec.nodeName.
   7. If bind fails (e.g. another scheduler raced): retry.
```

12. Component Explanation

- **Informer:** Watches API server; caches Pod + Node state.
- **Filter Phase:** Predicates to eliminate infeasible nodes.

- **Score Phase:** Priorities to rank feasible nodes.
- **Binder:** Posts binding to API server.
- **Node Cache:** In-memory snapshot of cluster state.

13. Technology Stack

- **Language:** Go (Kubernetes original).
- **Coordination:** API server + etcd.
- **Algorithm:** Filter + score (pluggable).
- **Deployment:** HA: multiple scheduler instances with leader election.

14. Database Choice

No DB. etcd (via API server) is source of truth.

15. Cache Strategy

In-memory node cache (refreshed via watch).

16. Load Balancer Strategy

Leader election (one active scheduler at a time).

17. Message Queue Usage

None. Watch API for events.

18. Scaling Strategy

Single scheduler handles 5000 nodes. For larger: multiple schedulers with partitioning.

19. Fault Tolerance

HA via leader election. If active dies, standby takes over.

20. Bottlenecks

- Scheduling throughput on large clusters — parallel scheduling.
- Cache staleness — watch latency.
- Bind contention (many Pods racing for same node) — retry.

21. Trade-offs

- **Spread vs pack:** Spread balances load; pack leaves nodes empty (for scaling down).
- **Single vs multiple schedulers:** Single is simpler; multiple enables custom policies.
- **Predict vs reactive:** Predict (pre-filter) is faster; reactive handles edge cases.

22. CAP Theorem Analysis

CP. etcd is source of truth. Optimistic concurrency prevents double-scheduling.

23. Security Considerations

TLS. RBAC (scheduler service account). Audit log.

24. Monitoring & Logging

Track scheduling latency, Pod pending time, scheduler queue length.

25. Deployment Strategy

HA deployment (3 replicas with leader election).

26. Interview Tips

- Discuss two-phase: filter (predicates) + score (priorities).
- Mention optimistic concurrency for bind (prevents double-scheduling).
- Discuss HA via leader election.
- Mention pluggable scheduling profiles.

27. Common Follow-up Questions

- How would you implement custom schedulers (multi-scheduler)?
- How would you handle gang scheduling (all-or-nothing for groups)?
- How would you implement Pod preemption (evict low-priority Pods)?
- How would you scale to 50K nodes?

28. Common Mistakes

- No optimistic concurrency — double-scheduling.
- Synchronous API calls per node — too slow.
- No caching — repeated API calls.
- Single scheduler instance — SPOF.

29. Optimization Ideas

- Cache node info via informer (avoid API calls).
- Parallel scheduling (score nodes in parallel).
- Pre-filter (skip obviously infeasible nodes).
- Batch scheduling for similar Pods.

30. Final Summary

The Kubernetes scheduler tests two-phase decision algorithms and optimistic concurrency. The core architecture: filter (eliminate infeasible nodes via predicates) → score (rank via priorities) → bind (atomic POST with optimistic concurrency). HA via leader election. The scheduler is stateless — etcd is source of truth. The architecture rewards understanding of distributed consensus and caching patterns.

QUESTION 57 ADVANCED

Design an API Gateway (Kong/Envoy)

1. Problem Statement

Design an API gateway that sits in front of microservices. Handles authentication, rate limiting, routing, protocol translation, observability, and circuit breaking.

2. Functional Requirements

- Route requests to backend services by path/method/header.
- Authenticate (JWT, OAuth, API key).
- Rate limit per user/IP/API.
- TLS termination.
- Request/response transformation.
- Circuit breaking (fail-fast on unhealthy backends).
- Logging + metrics + tracing.

3. Non-functional Requirements

- Latency overhead < 5ms.
- Availability 99.99% (gateway is SPOF if down).
- Throughput: 100K req/sec per instance.
- Horizontal scalability.

4. Capacity Estimation

100K req/sec per instance. 10 instances = 1M req/sec. Config: 10K routes.

5. Back-of-the-envelope Math

Config-driven (YAML/JSON). Hot reload. Plugin architecture for extensibility.

6. API Design

```
// Admin API (for config):
POST /routes body: {path: "/v1/users", service: "user-svc", methods: ["GET"]}
POST /plugins body: {name: "rate-limiting", config: {minute: 100}}

// Proxy API (for client traffic):
GET /v1/users -> routed to user-svc:8080/users
```

7. Database Schema

```
// Config (in DB or YAML):
Route {
  path_prefix, methods: [...],
  service: {name, url},
  plugins: [plugin_ref],
  strip_path: bool
}

Plugin {
  name: "rate-limiting" | "jwt" | "cors" | ...,
  config: {...}
}
```

8. High-Level Design (HLD)

Client → LB → API Gateway instances → Backend services. Each gateway: TLS terminate, authenticate, rate limit, route, transform, log. Config from DB, cached locally.

9. Low-Level Design (LLD)

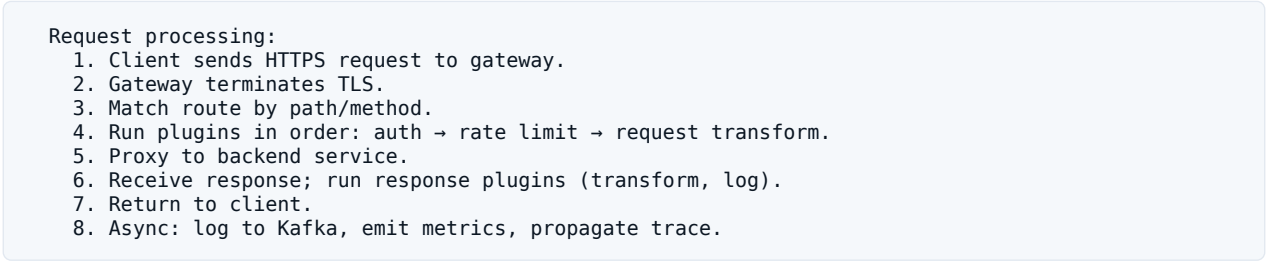
Per-request pipeline: parse → authenticate → rate limit → route → transform request → proxy → transform response → log. Plugins execute in order. Async log to Kafka.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Router:** Match request to route + service.
- **Auth Plugin:** Validate JWT, OAuth, API key.
- **Rate Limiter:** Token bucket per user/API (Redis-backed).
- **Circuit Breaker:** Track backend health; fail-fast when unhealthy.
- **Logger:** Async to Kafka; structured JSON.
- **Config Cache:** In-memory route + plugin config.

13. Technology Stack

- **Language:** Lua (Kong) or C++ (Envoy) or Go.
- **Config Store:** Postgres (Kong) or etcd (Envoy).
- **Cache:** Redis (rate limiting state).
- **Queue:** Kafka (logs, metrics).

14. Database Choice

Postgres for config (small, relational). Redis for rate limiting state.

15. Cache Strategy

In-memory config cache. Hot reload on config change.

16. Load Balancer Strategy

L4 LB in front of gateway instances. Gateway instances stateless.

17. Message Queue Usage

Kafka for logs, metrics, audit.

18. Scaling Strategy

Stateless gateway scales horizontally. Redis Cluster for rate limiting.

19. Fault Tolerance

Multi-AZ. If a gateway instance dies, LB routes to others.

20. Bottlenecks

- Rate limit Redis lookups — local token bucket + periodic sync.
- Config reload — hot reload (no restart).
- Per-request logging — async batch.

21. Trade-offs

- **Centralized vs sidecar gateway:** Centralized is simpler; sidecar (service mesh) is more granular.
- **Sync vs async logging:** Sync confirms; async is faster.
- **Plugin in-process vs out-of-process:** In-process is faster; out-of-process is safer.

22. CAP Theorem Analysis

AP. Stateless gateway; eventual consistency on config.

23. Security Considerations

TLS. mTLS to backends (optional). Auth plugins. Audit log. Rate limit per user.

24. Monitoring & Logging

Track request latency, error rate, circuit breaker state, config reload time.

25. Deployment Strategy

Blue-green. Hot config reload.

26. Interview Tips

- Discuss plugin architecture — core design.
- Mention hot config reload (no restart).
- Discuss rate limiting via Redis.
- Mention circuit breaker for backend health.

27. Common Follow-up Questions

- How would you implement service mesh (sidecar gateway)?
- How would you handle WebSocket through gateway?
- How would you implement GraphQL aggregation at gateway?
- How would you do canary deployments via gateway?

28. Common Mistakes

- Synchronous logging — kills latency.
- No circuit breaker — cascading failures.
- Config in DB only (no cache) — slow.
- Single instance — SPOF.

29. Optimization Ideas

- Local token bucket with periodic Redis sync.
- Batch logs to Kafka.
- Cache route matches.
- Connection pool to backends.

30. Final Summary

An API gateway tests the request pipeline pattern. The core architecture: config-driven routes + plugins, stateless instances behind LB, per-request pipeline (auth → rate limit → route → transform → proxy → log). Hot config reload (no restart). Plugin architecture for extensibility. Redis for rate limiting state. Circuit breaker for backend health. The architecture rewards investment in observability and graceful degradation.

QUESTION 58 ADVANCED

Design a Notification Service

1. Problem Statement

Design a multi-channel notification service. Other services send notifications (email, SMS, push, in-app). The service routes, deduplicates, and delivers with retries and tracking.

2. Functional Requirements

- Accept notification requests (channel, recipient, content).
- Route to appropriate provider (SendGrid, Twilio, FCM, APNs).
- Retry on failure (exponential backoff).
- Deduplicate (idempotency key).
- Track delivery status (sent, delivered, failed).
- User preferences (opt-out per channel/type).

3. Non-functional Requirements

- Availability 99.9%.
- Throughput: 100K notifications/sec.
- Latency: < 30s end-to-end (best effort).
- No duplicate notifications.

4. Capacity Estimation

10M notifications/day per channel = 40M total. Storage: 1 year = 14B records.

5. Back-of-the-envelope Math

Producer → Kafka topic → Consumer → Provider API. Idempotency via (user_id, notification_key) unique constraint.

6. API Design

```
POST /notifications body: {
  user_id, channel: EMAIL|SMS|PUSH|IN_APP,
  template_id, params: {...},
  idempotency_key: "..."
} -> {notification_id, status}

GET /notifications/{id}/status -> {state, attempts, delivered_at}
POST /users/{id}/preferences body: {channel: OPTED_IN|OPTED_OUT}
```

7. Database Schema

```
Table: notifications (
  notification_id, user_id, channel, template_id, params_json,
  status: QUEUED|SENDING|SENT|FAILED,
  attempts, last_error, idempotency_key,
  created_at, delivered_at
)
UNIQUE (user_id, idempotency_key)

Table: user_preferences (user_id, channel, notification_type, status)
```

8. High-Level Design (HLD)

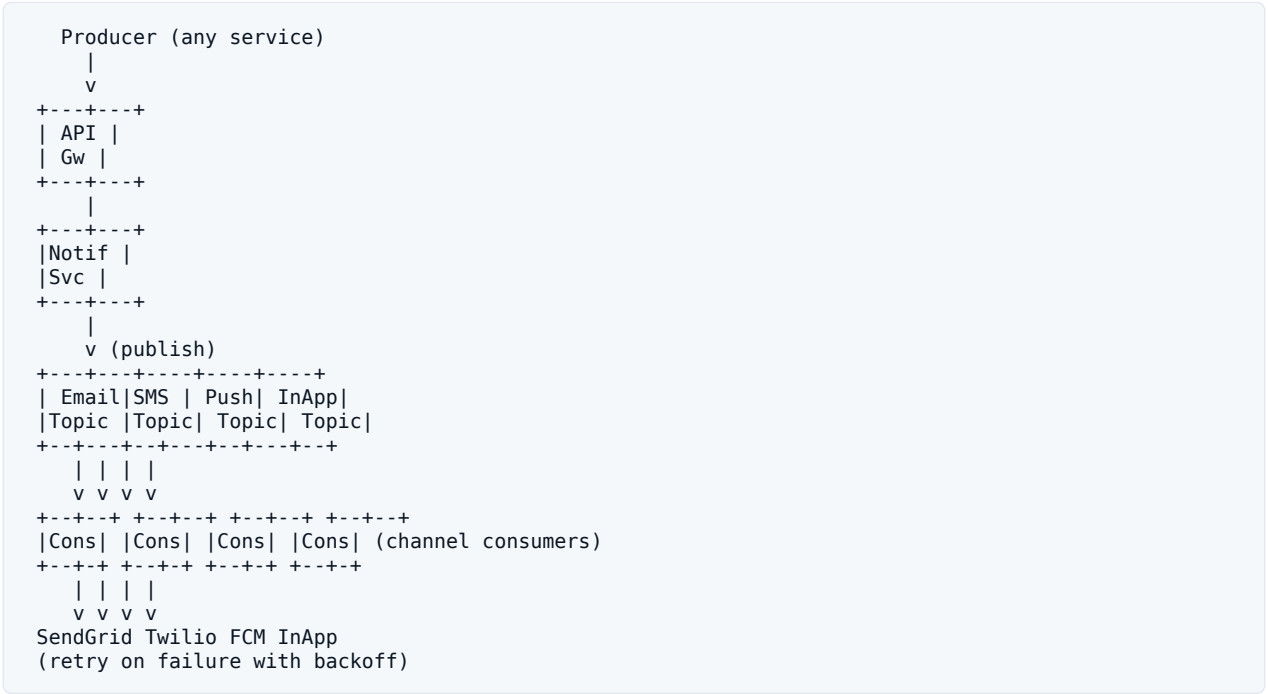
Producer → API Gateway → Notification Service (validate, dedup) → Kafka topic → Channel Consumers (Email/SMS/Push/InApp) → Provider APIs. Status tracked in DB.

9. Low-Level Design (LLD)

Notification Service: validate, check user preferences, check idempotency (unique constraint), insert as QUEUED, publish to Kafka. Channel consumers: pick up, call provider API, update status, retry on failure with exponential backoff.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Send notification:
  1. Producer POST /notifications with idempotency_key.
  2. Notification Service: check user preferences (opt-out?).
  3. Insert notification (QUEUED) with unique (user_id, idempotency_key).
     If duplicate: return existing notification_id.
  4. Publish to channel-specific Kafka topic.
  5. Channel consumer: fetch, render template, call provider API.
  6. On success: update status=SENT. On failure: retry (max 3, backoff).
  7. After max retries: status=FAILED, alert ops.
```

12. Component Explanation

- **Notification Service:** Validates, dedups, queues.
- **Channel Consumers:** Per-channel: Email, SMS, Push, InApp.
- **Provider Clients:** SendGrid, Twilio, FCM, APNs.
- **Template Engine:** Renders templates with params.
- **Preferences Service:** User opt-ins/outs per channel/type.
- **Kafka:** Decouples producer from consumer.
- **Postgres:** Notification records, status tracking.

13. Technology Stack

- **Language:** Java/Go/Python.
- **DB:** PostgreSQL sharded by user_id.
- **Cache:** Redis (preferences cache, rate limits).
- **Queue:** Kafka (per-channel topics).
- **Providers:** SendGrid (email), Twilio (SMS), FCM/APNs (push).

14. Database Choice

Postgres for ACID (idempotency unique constraint).

15. Cache Strategy

Redis for user preferences (avoid DB lookup per send).

16. Load Balancer Strategy

L7 LB. Stateless service.

17. Message Queue Usage

Kafka per-channel topics (Email, SMS, Push, InApp).

18. Scaling Strategy

Consumers scale per channel. Kafka partitions for parallelism.

19. Fault Tolerance

Multi-AZ. Retry with backoff. Dead letter queue for permanently failed.

20. Bottlenecks

- Provider rate limits (FCM/APNs) — batch + throttle.
- DB write throughput — shard by user_id.
- Template rendering — cache compiled templates.

21. Trade-offs

- **Sync vs async send:** Async is faster; sync confirms delivery.
- **Per-channel vs unified queue:** Per-channel isolates failures.
- **Retry vs dead letter:** Retry handles transient; DLQ handles permanent.

22. CAP Theorem Analysis

AP. Eventual delivery; idempotency prevents duplicates.

23. Security Considerations

TLS. PII encryption. Rate limit per producer. Audit log.

24. Monitoring & Logging

Track queue depth, delivery latency, failure rate, retry count.

25. Deployment Strategy

Blue-green. Auto-scale consumers by queue depth.

26. Interview Tips

- Discuss idempotency via unique constraint — prevents duplicates.
- Mention per-channel topics — isolates failures.
- Discuss user preferences (opt-out).
- Mention retry with exponential backoff.

27. Common Follow-up Questions

- How would you implement digest emails (daily summary)?
- How would you handle templating (multi-language)?
- How would you A/B test notification content?
- How would you implement notification analytics (open rate)?

28. Common Mistakes

- No idempotency — duplicate notifications on retry.
- Synchronous provider call — slow + blocks on provider outage.
- No retry — transient failures become permanent.
- No user preferences — users can't opt out.

29. Optimization Ideas

- Batch provider calls (e.g. FCM supports 1000 tokens per request).
- Cache user preferences in Redis.
- Pre-compile templates.
- Use connection pool to providers.

30. Final Summary

A notification service tests async pipeline + idempotency. The core architecture: producer → API → service (validate + dedup via unique constraint) → Kafka per channel → consumer → provider API. Retries with exponential backoff. User preferences (opt-out) respected. The system is AP — eventual delivery, no duplicates via idempotency keys. Per-channel topics isolate failures (email outage doesn't block push).

QUESTION 59 ADVANCED

Design a Payment Gateway (Stripe)

1. Problem Statement

Design a payment gateway. Merchants integrate via API; gateway processes payments, routes to card networks, handles refunds, and reconciles with banks.

2. Functional Requirements

- Accept payment requests (card, bank, wallet).
- Authorize + capture (or just authorize).
- Refunds (full + partial).
- Webhook notifications to merchants.
- Reconciliation with card networks + banks.
- Fraud detection.

3. Non-functional Requirements

- Availability 99.99% (lost payment = lost money).
- Latency: < 3s per transaction.
- Strong consistency: no double-charge, no lost payment.
- PCI compliance.

4. Capacity Estimation

10M transactions/day. Storage: 5yr × 10M × 1KB = 18TB.

5. Back-of-the-envelope Math

ACID DB for transactions. Idempotency via merchant_id + idempotency_key. Async reconciliation with banks.

6. API Design

```
POST /charges body: {amount, currency, source, idempotency_key} -> {charge_id, status}
POST /charges/{id}/capture body: {amount?}
POST /charges/{id}/refund body: {amount?}
GET /charges/{id}
POST /webhooks/subscribe body: {url, events: [...]}

// Gateway calls card network (Visa/MC) async, updates charge status
```

7. Database Schema

```
Table: charges (
  charge_id, merchant_id, amount_cents, currency,
  source_token, status: PENDING|AUTHORIZED|CAPTURED|FAILED|REFUNDED,
  idempotency_key, network_txn_id, created_at, captured_at
)
UNIQUE (merchant_id, idempotency_key)

Table: refunds (refund_id, charge_id, amount_cents, status, created_at)
Table: webhook_deliveries (delivery_id, merchant_id, event_id, status, attempts)
```

8. High-Level Design (HLD)

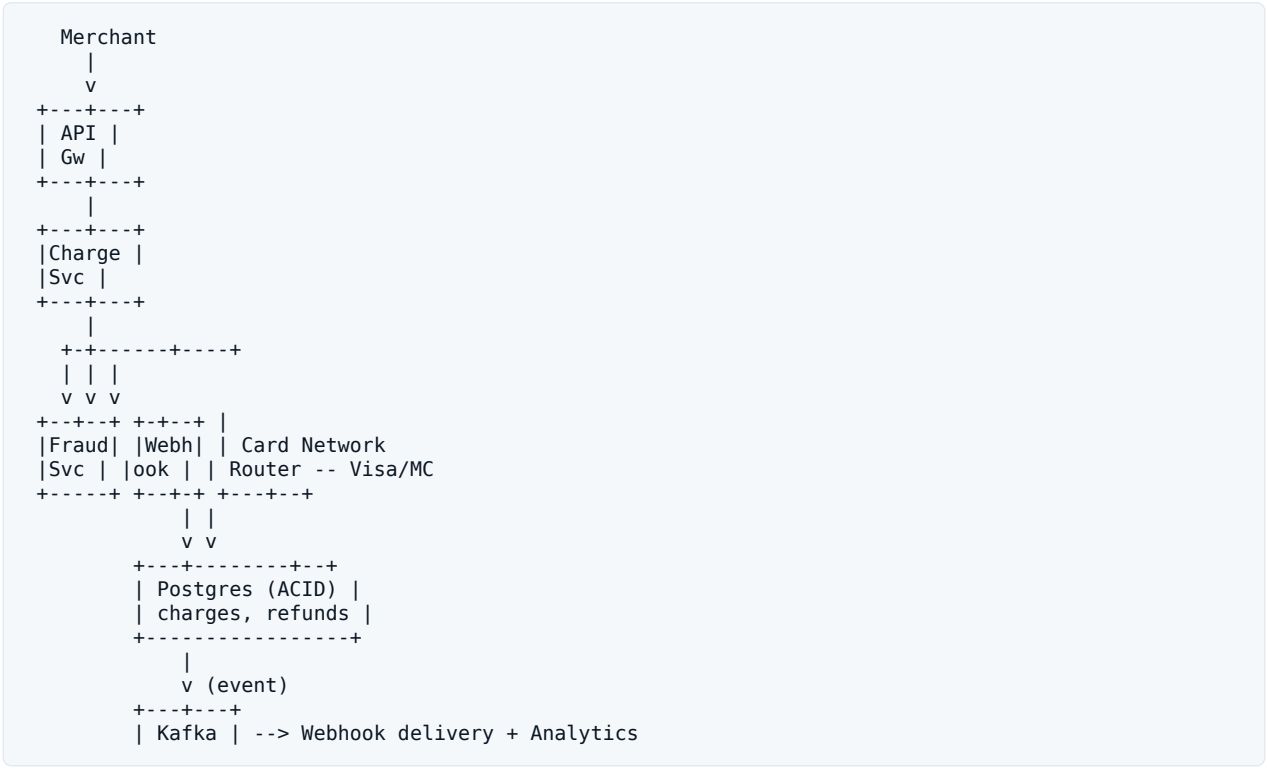
Merchant → API Gateway → Charge Service (transactional) → Card Network Router → Visa/MC API. Webhook Service for notifications. Fraud Service for scoring.

9. Low-Level Design (LLD)

Charge Service: BEGIN TXN; check idempotency; INSERT charge (PENDING); call card network; update status; COMMIT. Publish event to Kafka for webhooks + analytics. Webhook Service retries delivery on failure.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Charge:
  1. Merchant POST /charges with idempotency_key.
  2. Charge Service: BEGIN TXN.
  3. Check (merchant_id, idempotency_key) unique; if exists, return existing.
  4. INSERT charge (PENDING).
  5. Call Fraud Service (async score, block if high risk).
  6. Call Card Network Router (Visa/MC).
  7. On success: UPDATE status=AUTHORIZED. On failure: status=FAILED.
  8. COMMIT. Publish event to Kafka.
  9. Webhook Service delivers to merchant (with retries).
```

12. Component Explanation

- **Charge Service:** Transactional charge + refund logic.
- **Card Network Router:** Routes to Visa/MC/Amex; format conversion.
- **Fraud Service:** ML model scoring; block high-risk.
- **Webhook Service:** Deliver events to merchants with retries.
- **Reconciliation Service:** Match with bank settlement files (daily).

13. Technology Stack

- **Language:** Java/Go (low latency).
- **DB:** PostgreSQL sharded by merchant_id (ACID).
- **Cache:** Redis (rate limits, hot merchants).
- **Queue:** Kafka (events, webhook delivery).
- **ML:** PyTorch (fraud).

14. Database Choice

Postgres for ACID (no double-charge). Sharded by merchant_id.

15. Cache Strategy

Redis for rate limits, hot merchant configs.

16. Load Balancer Strategy

L7 LB. Geographic routing (low latency to merchant).

17. Message Queue Usage

Kafka for events (charge.created, charge.refunded).

18. Scaling Strategy

Shard by merchant_id. Card network router scales by txn volume.

19. Fault Tolerance

Multi-AZ. Idempotent retries. Reconciliation catches inconsistencies.

20. Bottlenecks

- Card network latency (2-5s) — optimize network calls.
- PCI compliance overhead — tokenize cards, minimize PII.
- Reconciliation latency — daily batch.

21. Trade-offs

- **Sync vs async capture:** Sync confirms immediately; async is faster but riskier.
- **Auth + capture vs direct:** Two-phase is safer; one-phase is faster.
- **Block vs review fraud:** Block is safer; review has fewer false positives.

22. CAP Theorem Analysis

CP. Strong consistency for charges. Reject on partition (no double-charge risk).

23. Security Considerations

TLS. PCI DSS Level 1. Tokenization (no raw card storage). Encryption at rest. Audit log. 3D Secure.

24. Monitoring & Logging

Track charge latency, failure rate, fraud rate, webhook delivery rate.

25. Deployment Strategy

Blue-green. Multi-region active-active (with care for consistency).

26. Interview Tips

- Emphasize idempotency (merchant_id + key) — prevents double-charge.
- Discuss ACID transaction + async card network call.
- Mention PCI compliance (tokenization).
- Discuss reconciliation with banks.

27. Common Follow-up Questions

- How would you handle a chargeback dispute?
- How would you implement recurring payments (subscriptions)?
- How would you detect sophisticated fraud?
- How would you handle multi-currency settlements?

28. Common Mistakes

- No idempotency — double-charge on retry.
- Storing raw card data — PCI violation.
- No webhook retry — merchants miss events.
- No reconciliation — silent inconsistencies.

29. Optimization Ideas

- Tokenize cards (no raw data after first charge).

- Batch webhook delivery per merchant.
- Cache fraud scores per user.
- Co-locate with card networks (lower latency).

30. Final Summary

A payment gateway tests ACID transactions + idempotency at the highest level. The core challenge: no double-charge, no lost payment, even under network failure and retry. Solved with idempotency unique constraint (merchant_id + key) + transactional state machine + idempotent card network calls. PCI compliance mandates tokenization (no raw card data). Webhook delivery with retries ensures merchants receive events. Reconciliation with bank settlement files catches silent inconsistencies. The system is CP — strong consistency non-negotiable.

QUESTION 60 ADVANCED

Design a Global CDN (Cloudflare)

1. Problem Statement

Design a Content Delivery Network with edge locations worldwide. Cache static content at edge; serve dynamic content via origin shield. Handle cache invalidation and DDoS protection.

2. Functional Requirements

- Cache static content (images, CSS, JS, video) at edge PoPs.
- Route users to nearest PoP (anycast DNS).
- Origin shield (single fetch per object on cache miss).
- Cache invalidation (purge by URL or tag).
- DDoS protection (rate limit, scrubbing).
- TLS termination at edge.

3. Non-functional Requirements

- Latency: < 50ms to user (edge).
- Availability 99.99%.
- Cache hit ratio > 95%.
- Handle 1Tbps+ global traffic.

4. Capacity Estimation

500 PoPs globally. 10M objects cached per PoP (avg 100KB) = 1TB/PoP = 500TB total. Egress: 1Tbps.

5. Back-of-the-envelope Math

Anycast DNS routes user to nearest PoP. PoP caches objects on first fetch. Origin shield deduplicates origin fetches.

6. API Design

```
// Customer config:
POST /zones/{zone_id}/cache_purge body: {files: [...], tags: [...]}
POST /zones/{zone_id}/firewall/rules

// Internal: edge PoP receives HTTP request, checks cache, fetches from origin on miss
```

7. Database Schema

```
// Per-PoP cache (in-memory + disk):
CacheEntry {
  url: string // cache key
  content: bytes
  headers: {content_type, etag, cache_control, ...}
  expires_at: timestamp
  tags: [...] // for tag-based purging
}

// Central config DB:
Table: zones (zone_id, customer_id, origin_url, ...)
Table: purge_queue (id, zone_id, target, created_at)
```

8. High-Level Design (HLD)

User → Anycast DNS → Nearest PoP → Cache (hit) or Origin Shield → Origin. PoPs share cache via parent-tier (origin shield) to deduplicate fetches. Central config + purge coordination.

9. Low-Level Design (LLD)

PoP receives request: check local cache. On hit: serve. On miss: fetch from origin shield (regional parent PoP). Origin shield fetches from origin if also miss. Purge: central API enqueues purge; propagates to all PoPs via pub/sub.

10. Architecture Diagram

Architecture Diagram

```
   User (Asia)
     |
     v (Anycast DNS)
 +---+---+
 | PoP | (Singapore)
 | cache |
 +---+---+
     | (miss)
     v
 +---+---+
 |Origin | (regional parent, deduplicates fetches)
 |Shield |
 +---+---+
     | (miss)
     v
 +---+---+
 |Origin | (customer server)
 +-------+

 Anycast: same IP announced from all PoPs; BGP routes user to nearest
```

11. Data Flow Diagram

Data Flow Diagram

```
Request + cache:
  1. User requests image; DNS resolves to nearest PoP via Anycast.
  2. PoP checks local cache.
  3. Hit: serve (15ms latency).
  4. Miss: fetch from Origin Shield (regional parent).
  5. Shield miss: fetch from origin; cache at shield + PoP.
  6. Return to user.

Purge:
  1. Customer POST /cache_purge with URL or tag.
  2. Central API enqueues purge to all PoPs via pub/sub.
  3. PoPs invalidate matching entries (typically < 30s globally).
```

12. Component Explanation

- **PoP (Edge Cache):** Caches objects; serves users.
- **Origin Shield:** Regional parent; dedupes origin fetches.
- **Anycast DNS:** Routes user to nearest PoP.
- **Purge Coordinator:** Distributes cache invalidations.
- **DDoS Scrubber:** Rate limit + filter malicious traffic.
- **Config DB:** Customer zones, origin URLs, firewall rules.

13. Technology Stack

- **Language:** C/C++/Rust (high perf edge).
- **Cache:** In-memory (LRU) + SSD tier.
- **DNS:** Anycast BGP.
- **Config:** Postgres + etcd for distribution.
- **Queue:** Kafka / NATS for purge propagation.

14. Database Choice

Per-PoP in-memory + SSD cache. Postgres for central config.

15. Cache Strategy

This IS the cache. Multi-tier: PoP → Origin Shield → Origin.

16. Load Balancer Strategy

Anycast DNS (BGP-based geographic routing).

17. Message Queue Usage

NATS/Kafka for purge propagation to all PoPs.

18. Scaling Strategy

Add more PoPs for geographic coverage. Scale per-PoP vertically.

19. Fault Tolerance

PoP failure: Anycast routes to next nearest. Origin shield HA.

20. Bottlenecks

- Origin fetch on miss — origin shield dedupes.
- Purge propagation latency — pub/sub to all PoPs.
- Cache eviction (LRU) — SSD tier for cold objects.

21. Trade-offs

- **Cache size vs cost:** Larger cache = higher hit ratio but more storage.
- **Purge latency vs consistency:** Faster purge = better consistency but more overhead.
- **PoP count vs cost:** More PoPs = lower latency but more infrastructure.

22. CAP Theorem Analysis

AP. Cache is eventually consistent. Purge takes < 30s globally.

23. Security Considerations

TLS at edge. DDoS scrubbing. WAF rules. Origin IP hidden.

24. Monitoring & Logging

Track cache hit ratio, origin fetch rate, purge latency, PoP health.

25. Deployment Strategy

PoPs in data centers worldwide. Anycast BGP for routing.

26. Interview Tips

- Discuss Anycast DNS — core routing mechanism.
- Mention origin shield — dedupes origin fetches.
- Discuss cache purge (URL + tag-based).
- Mention DDoS protection as a CDN feature.

27. Common Follow-up Questions

- How would you implement edge compute (Cloudflare Workers)?
- How would you handle video streaming (HLS chunks)?
- How would you implement image transformation at edge?
- How would you handle a 100Gbps DDoS attack?

28. Common Mistakes

- No origin shield — every PoP fetches from origin on miss.
- No purge mechanism — stale content.
- Single PoP location — high latency for distant users.
- No DDoS protection — origin exposed.

29. Optimization Ideas

- Pre-fetch popular objects to edge.
- Use SSD tier for cold objects (larger cache).
- Compress content at edge (gzip/br).
- Batch purge propagation.

30. Final Summary

A global CDN tests edge caching + Anycast routing. The core architecture: Anycast DNS routes users to nearest PoP; PoP caches objects on first fetch; origin shield dedupes origin fetches; purge propagation via pub/sub. Multi-tier cache (PoP → shield → origin) optimizes hit ratio. The system is AP — cache eventually consistent, purges take < 30s. The architecture rewards geographic distribution and DDoS protection investment.

QUESTION 61 ADVANCED

Design Zoom (Scale + Recording)

1. Problem Statement

Design Zoom at massive scale: 1M concurrent meetings, recording, breakout rooms, live streaming to YouTube, and webinar mode (1 host to 10K viewers).

2. Functional Requirements

- Host/join meetings up to 1000 participants.
- Webinar mode: 1 host to 10K+ viewers.
- Cloud recording + live streaming to YouTube.
- Breakout rooms.
- Screen sharing + whiteboard.

3. Non-functional Requirements

- Latency < 200ms end-to-end.
- Availability 99.95%.
- 1M concurrent meetings.
- Recording without affecting call quality.

4. Capacity Estimation

1M concurrent meetings × 10 avg = 10M users. Bandwidth: 10M × 2Mbps = 20Tbps. Recording: 1M × 100MB/hr = 100TB/hr.

5. Back-of-the-envelope Math

Webinar mode uses HLS broadcasting (one-to-many) instead of SFU (many-to-many). Recording: media server captures to S3 in chunks.

6. API Design

```
POST /meetings body: {host_id, type: MEETING|WEBINAR}
POST /meetings/{id}/record/start
POST /meetings/{id}/livestream body: {target: YOUTUBE, rtmp_url}
POST /meetings/{id}/breakouts body: [{name, participants}]
```

7. Database Schema

```
Table: meetings (meeting_id, host_id, type, started_at, recording_url)
Table: participants (meeting_id, user_id, joined_at, left_at)
Table: recordings (recording_id, meeting_id, s3_url, duration, started_at)
```

8. High-Level Design (HLD)

Browser/Mobile → API Gateway → Meeting Service (control) + Media Servers (SFU for meetings, HLS for webinars) + Recording Service + Live Stream Service.

9. Low-Level Design (LLD)

Webinar: presenter pushes to ingest server; transcoded to HLS; viewers watch via CDN. Recording: media server captures mixed stream to S3 in HLS chunks.

10. Architecture Diagram

Architecture Diagram

```
Meeting: P1 P2 P3 P4 <-> SFU (mesh)

Webinar:
  Presenter
    |
    v
  +---+----+ (transcode)
  |Ingest |
  +---+----+
      |
      v
  +---+----+ (HLS segments)
  |Origin |
  +---+----+
      |
      v
  +---+----+
  | CDN | --> 10K viewers
```

11. Data Flow Diagram

Data Flow Diagram

```
Webinar stream:
  1. Presenter pushes RTMP/SRT to ingest server.
  2. Transcode to HLS segments; store to S3 origin.
  3. CDN pulls; caches at edge.
  4. Viewers watch via HLS (10s latency acceptable).
  5. Recording: same HLS segments saved to S3.
```

12. Component Explanation

- **SFU:** For meetings (many-to-many).
- **HLS Ingest:** For webinars (one-to-many).
- **Recording Service:** Captures to S3.
- **Live Stream Service:** RTMP output to YouTube/Facebook.
- **CDN:** HLS distribution.

13. Technology Stack

- **Language:** C++ (media), Go (control).
- **Media:** WebRTC + HLS.
- **Storage:** S3.
- **CDN:** CloudFront/Akamai.

14. Database Choice

Postgres for meeting metadata. S3 for recordings + HLS segments.

15. Cache Strategy

Redis for active meeting state.

16. Load Balancer Strategy

Sticky WebSocket. Geographic routing for SFU.

17. Message Queue Usage

Kafka for meeting events.

18. Scaling Strategy

SFU scales by meeting count. CDN scales for webinar viewers.

19. Fault Tolerance

Multi-AZ. SFU failover. Recording replicated.

20. Bottlenecks

- Webinar egress bandwidth — CDN.
- Recording storage — S3 + lifecycle rules.

21. Trade-offs

- **SFU vs HLS for webinar:** SFU is interactive; HLS is one-way but scales.
- **Live vs post-processed recording:** Live is real-time; post is higher quality.

22. CAP Theorem Analysis

AP. Eventual consistency for recording + chat.

23. Security Considerations

TLS + DTLS-SRTP. End-to-end encryption (optional). Waiting room. Anti-Zoom-bombing.

24. Monitoring & Logging

Track media latency, packet loss, recording success rate, CDN egress.

25. Deployment Strategy

SFU in dedicated node pools. Multi-region.

26. Interview Tips

- Distinguish meeting (SFU) vs webinar (HLS) architectures.
- Discuss CDN for webinar distribution.
- Recording as silent participant.

27. Common Follow-up Questions

- How would you implement end-to-end encryption?
- How would you handle 10K-person interactive meetings?
- How would you implement live transcription?

28. Common Mistakes

- SFU for 10K viewers — bandwidth explosion.
- Synchronous recording — affects call quality.
- No CDN for webinar — origin crushed.

29. Optimization Ideas

- Simulcast for adaptive quality.
- Edge-cache webinar HLS.
- Compress recordings (H.265).

30. Final Summary

Zoom at scale tests meeting (SFU) vs webinar (HLS) architectures. Meetings use WebRTC SFU for interactivity; webinars use HLS via CDN for one-to-many scale. Recording captures mixed stream to S3. The architecture rewards understanding of media protocols and CDN distribution.

QUESTION 62 ADVANCED

Design Slack at Enterprise Scale

1. Problem Statement

Design Slack for 10M+ enterprise users. Handle workspace sharding, search across millions of messages, and 100K concurrent WebSocket connections per workspace.

2. Functional Requirements

- Workspaces, channels, DMs, threads.
- Real-time messaging via WebSocket.
- Search across all message history.
- File sharing + previews.
- Integrations (bots, webhooks, apps).

3. Non-functional Requirements

- Latency: message delivery < 500ms.
- Availability 99.95%.
- Search latency < 1s.
- Per-workspace isolation.

4. Capacity Estimation

10M users, 100K workspaces, 10M msgs/day. Storage: 50TB messages + 100TB files.

5. Back-of-the-envelope Math

Shard by workspace_id. Per-workspace: PG for messages, ES for search, Redis for state. WebSocket Hub per workspace (stateful).

6. API Design

```
POST /workspaces/{id}/channels/{id}/messages body: {text}
GET /workspaces/{id}/search?q=...
// WebSocket /workspaces/{id}/stream per user
```

7. Database Schema

```
Table: messages (msg_id, workspace_id, channel_id, author_id, text, created_at)
  SHARD BY workspace_id
Table: channels (channel_id, workspace_id, name, ...)
Table: files (file_id, workspace_id, msg_id, s3_key)
```

8. High-Level Design (HLD)

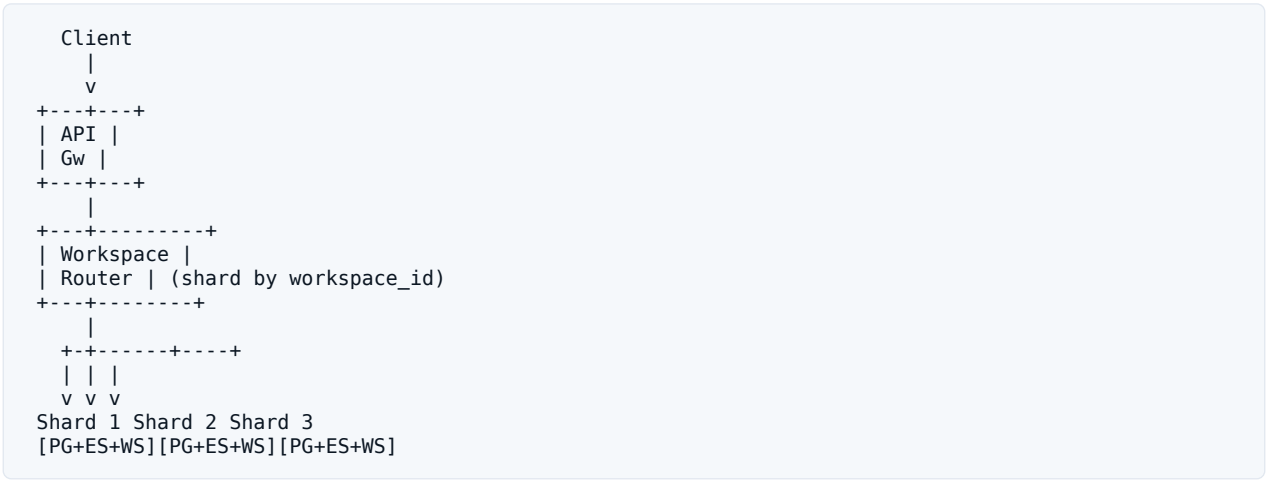
Client → API Gateway → Workspace Router (shards by workspace_id) → Message Service + Search Service + WebSocket Hub. Per-workspace isolation.

9. Low-Level Design (LLD)

Workspace Router: hash(workspace_id) → shard. All requests for a workspace go to the same shard (affinity). WebSocket Hub per shard (stateful).

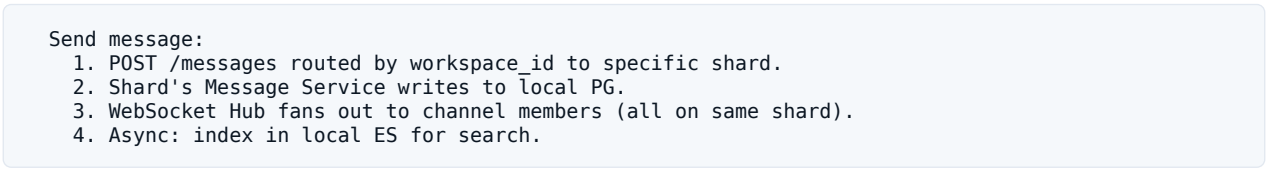
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Workspace Router:** Shards by workspace_id.
- **Per-Shard Services:** Message, Search, WebSocket per shard.
- **File Service:** S3 + CDN.

13. Technology Stack

- **Language:** Go/Java.
- **DB:** PostgreSQL sharded by workspace_id.
- **Search:** Elasticsearch per shard.
- **Cache:** Redis per shard.

14. Database Choice

Postgres per shard (workspace-scoped).

15. Cache Strategy

Redis per shard.

16. Load Balancer Strategy

Workspace affinity routing.

17. Message Queue Usage

Kafka for cross-shard events (rare).

18. Scaling Strategy

Add shards as workspace count grows. Rebalance workspaces across shards.

19. Fault Tolerance

Per-shard HA. Shard failure affects only its workspaces.

20. Bottlenecks

- Large workspace (100K users) — dedicated shard.

21. Trade-offs

- **Per-shard vs shared:** Per-shard isolates; shared is denser.

22. CAP Theorem Analysis

AP. Eventual consistency for search.

23. Security Considerations

TLS. SSO. Per-workspace auth. PII encryption.

24. Monitoring & Logging

Track per-shard latency, WebSocket connections, search latency.

25. Deployment Strategy

Multi-AZ. Shard-aware routing.

26. Interview Tips

- Shard by workspace_id for affinity.
- Per-shard services isolate failures.
- Large workspaces get dedicated shards.

27. Common Follow-up Questions

- How would you handle cross-workspace channels (Slack Connect)?
- How would you migrate a workspace between shards?
- How would you handle a 1M-user workspace?

28. Common Mistakes

- No workspace affinity — messages spread across shards.
- Single ES cluster — can't scale.

29. Optimization Ideas

- Pre-compute unread counts per user.
- Cache hot channels.
- Batch WebSocket fanout.

30. Final Summary

Slack at enterprise scale tests workspace-based sharding. The core insight: shard by workspace_id so all messages, search, and WebSocket connections for a workspace live on one shard. This enables per-shard isolation (a shard failure affects only its workspaces) and affinity (no cross-shard fanout for messages). Large workspaces get dedicated shards. The architecture rewards understanding of stateful sharding.

QUESTION 63

ADVANCED

Design Microsoft Teams

1. Problem Statement

Design an enterprise collaboration platform: chat, video calls, file sharing, and Office 365 integration. 100M+ enterprise users.

2. Functional Requirements

- Chat (1:1, group, channels).
- Video/audio calls.
- File sharing (SharePoint).
- Office 365 integration (Word, Excel co-editing).
- Calendar + email integration.

3. Non-functional Requirements

- Latency: chat < 500ms, voice < 100ms.
- Availability 99.95%.
- Tenant isolation.

4. Capacity Estimation

100M users, 10M tenants, 100M msgs/day.

5. Back-of-the-envelope Math

Shard by tenant_id. Video via SFU. Files via SharePoint.

6. API Design

```
POST /tenants/{id}/channels/{id}/messages
GET /tenants/{id}/search?q=...
POST /calls/initiate
```

7. Database Schema

```
messages(msg_id, tenant_id, channel_id, author_id, text, created_at) SHARD BY tenant_id
```

8. High-Level Design (HLD)

Client → API Gateway → Tenant Router → Chat Service + Call Service (SFU) + File Service (SharePoint) + O365 Integration.

9. Low-Level Design (LLD)

Tenant-based sharding. Chat: WebSocket + PG per shard. Calls: WebRTC SFU. Files: SharePoint integration via Graph API.

10. Architecture Diagram

Architecture Diagram

```
Client → API Gw → Tenant Router → Shard (Chat + Call + Files + O365)
```

11. Data Flow Diagram

Data Flow Diagram

```
Send message → tenant shard → PG → WebSocket fanout. Call → SFU + WebRTC.
```

12. Component Explanation

- **Tenant Router:** Shards by tenant_id.
- **Chat Service:** Per-shard messaging.
- **Call Service:** WebRTC SFU.
- **File Service:** SharePoint integration.

13. Technology Stack

- **Language:** C#/.NET (Microsoft stack).
- **DB:** SQL Server sharded by tenant_id.
- **Files:** SharePoint + OneDrive.
- **Realtime:** WebSocket + WebRTC.

14. Database Choice

SQL Server per shard (tenant-scoped).

15. Cache Strategy

Redis per shard.

16. Load Balancer Strategy

Tenant affinity routing.

17. Message Queue Usage

Azure Service Bus.

18. Scaling Strategy

Add shards as tenant count grows.

19. Fault Tolerance

Per-shard HA. Azure region failover.

20. Bottlenecks

- Large tenants — dedicated shards.

21. Trade-offs

- **Tenant shard vs shared:** Shard isolates; shared is denser.

22. CAP Theorem Analysis

AP. Eventual consistency for search.

23. Security Considerations

TLS. AAD auth. Per-tenant data isolation. Compliance (GDPR, HIPAA).

24. Monitoring & Logging

Track per-shard latency, call quality, file sync.

25. Deployment Strategy

Multi-region Azure.

26. Interview Tips

- Discuss tenant-based sharding.
- Mention O365 integration via Graph API.

27. Common Follow-up Questions

- How would you handle a 1M-user tenant?
- How would you implement live events (webinars)?

28. Common Mistakes

- No tenant isolation.
- Single global cluster.

29. Optimization Ideas

- Pre-compute unread counts.
- Cache hot channels.

30. Final Summary

Microsoft Teams tests tenant-based sharding at enterprise scale. Architecture mirrors Slack (workspace sharding) with tenant affinity. Differentiator: deep O365 integration (Graph API for files, calendar, email).

QUESTION 64 ADVANCED

Design Gmail

1. Problem Statement

Design a webmail service. Users send/receive emails, organize with labels/folders, search across all mail, and access via IMAP/SMTP + web.

2. Functional Requirements

- Send/receive email (SMTP/IMAP).
- Organize with labels, folders, filters.
- Full-text search.
- Spam filtering.
- Attachments up to 25MB.

3. Non-functional Requirements

- Availability 99.99%.
- Search latency < 500ms.
- Spam precision > 99%.
- Storage: 15GB/user.

4. Capacity Estimation

1.5B users, 5B emails/day. Storage: 1.5B × 15GB = 22EB.

5. Back-of-the-envelope Math

Shard by user_id. Per-user: PG for metadata, object storage for body + attachments. ES for search.

6. API Design

```
POST /messages body: {to, subject, body, attachments}
GET /messages?label=...
GET /search?q=...
```

7. Database Schema

```
messages(msg_id, user_id, from, to, subject, snippet, labels, received_at) SHARD BY
user_id
attachments(att_id, msg_id, s3_key, filename, size)
```

8. High-Level Design (HLD)

SMTP Server (receive) → Mail Delivery Service → User Shard (PG + S3 + ES). Web client → API Gateway → User Shard.

9. Low-Level Design (LLD)

Incoming email: SMTP server receives, identifies recipient, routes to user's shard, writes metadata + body, indexes for search. Spam filter pre-delivery.

10. Architecture Diagram

Architecture Diagram

```
Sender -> SMTP Server -> Mail Delivery -> User Shard (PG + S3 + ES) <- Web Client
```

11. Data Flow Diagram

Data Flow Diagram

```
Receive email -> SMTP -> identify recipient -> user shard -> PG metadata + S3 body + ES index. Read ->
web client -> user shard -> return metadata + body.
```

12. Component Explanation

- **SMTP Server:** Receives incoming mail.
- **Mail Delivery Service:** Routes to user shard.
- **Spam Filter:** ML-based classification.
- **Per-User Shard:** PG + S3 + ES.

13. Technology Stack

- **Language:** Java/C++.
- **DB:** PostgreSQL per shard.
- **Storage:** S3 for bodies + attachments.
- **Search:** Elasticsearch per shard.
- **Spam:** TensorFlow ML.

14. Database Choice

PG per shard (user-scoped). S3 for bodies.

15. Cache Strategy

Redis for hot mailboxes.

16. Load Balancer Strategy

User affinity routing.

17. Message Queue Usage

Kafka for incoming mail events.

18. Scaling Strategy

Shard by user_id. Add shards as users grow.

19. Fault Tolerance

Per-shard HA.

20. Bottlenecks

- Spam filter latency — async.
- Search index size — per-shard ES.

21. Trade-offs

- **Push (IMAP IDLE) vs polling:** Push is real-time; polling is simpler.

22. CAP Theorem Analysis

AP. Eventual consistency for search.

23. Security Considerations

TLS. Spam filtering. Phishing detection. Encryption at rest.

24. Monitoring & Logging

Track delivery latency, search latency, spam rate.

25. Deployment Strategy

Multi-region.

26. Interview Tips

- Discuss SMTP/IMAP integration.
- Mention spam filter ML pipeline.

27. Common Follow-up Questions

- How would you handle mailing lists?
- How would you implement Smart Compose (AI suggestions)?

28. Common Mistakes

- Single global cluster — no user isolation.
- Synchronous spam filter.

29. Optimization Ideas

- Pre-compute search index.
- Compress email bodies.
- CDN for attachments.

30. Final Summary

Gmail tests email protocol integration + user-based sharding. Per-user shard holds PG metadata + S3 bodies + ES search index. SMTP receives; Mail Delivery routes to user shard. Spam filter is ML-driven.

QUESTION 65

ADVANCED

Design Google Docs (Real-Time Collaboration at Scale)

1. Problem Statement

Design Google Docs supporting 100M concurrent editors. Operational Transform or CRDT for conflict-free merging. Offline support + cross-device sync.

2. Functional Requirements

- Real-time collaborative editing.
- Comments + suggestions.
- Version history.
- Offline mode + sync on reconnect.
- Cross-device sync.

3. Non-functional Requirements

- Keystroke-to-render < 200ms.
- Availability 99.95%.
- No lost edits.
- Eventual consistency.

4. Capacity Estimation

100M concurrent docs edited. 1B total docs. Storage: 1B × 100KB = 100TB.

5. Back-of-the-envelope Math

CRDT (Conflict-free Replicated Data Type) for conflict-free merging. WebSocket per doc. Op log + snapshots.

6. API Design

```
POST /docs body: {title}
GET /docs/{id}
// WebSocket /docs/{id}/stream for ops
```

7. Database Schema

```
docs(doc_id, owner_id, title, current_version, created_at)
doc_ops(op_id, doc_id, version, op_type, position, content, author_id, timestamp)
```

8. High-Level Design (HLD)

Browser → API Gateway → Doc Service (CRUD) + Collaboration Service (CRDT, WebSocket) + Storage (PG op log + S3 snapshots).

9. Low-Level Design (LLD)

CRDT-based: each keystroke is an op; ops are commutative (any order converges). Collaboration Service maintains in-memory doc state, applies ops, broadcasts. Snapshots every N ops.

10. Architecture Diagram

Architecture Diagram

```
User A -> WebSocket -> Collab Svc (CRDT) <- WebSocket <- User B; Collab Svc -> PG op log + S3 snapshots
```

11. Data Flow Diagram

Data Flow Diagram

```
Keystroke -> op via WebSocket -> CRDT apply -> broadcast to others -> append op log -> periodic snapshot.
```

12. Component Explanation

- **Collaboration Service:** CRDT engine + WebSocket.
- **Doc Service:** CRUD.
- **Op Log:** PG append-only.
- **Snapshot Store:** S3.

13. Technology Stack

- **Language:** JS (browser) + Go (server).
- **Algorithm:** CRDT (Yjs, Automerge).
- **DB:** PostgreSQL.
- **Storage:** S3.

14. Database Choice

PG for op log. S3 for snapshots.

15. Cache Strategy

In-memory doc state per Collaboration Service instance.

16. Load Balancer Strategy

Sticky WebSocket per doc_id.

17. Message Queue Usage

Optional Kafka for doc events.

18. Scaling Strategy

Collaboration Service scales by doc count (sticky routing).

19. Fault Tolerance

Op log + snapshots enable restore.

20. Bottlenecks

- High-concurrency docs — limit collaborators.

21. Trade-offs

- **OT vs CRDT:** CRDT is simpler; OT is proven but complex.

22. CAP Theorem Analysis

AP. Eventual consistency via CRDT convergence.

23. Security Considerations

TLS. Per-doc permissions. PII handling.

24. Monitoring & Logging

Track keystroke latency, op broadcast latency.

25. Deployment Strategy

StatefulSets (sticky routing).

26. Interview Tips

- Discuss CRDT vs OT.
- Mention op log + snapshots.

27. Common Follow-up Questions

- How would you handle 100 concurrent editors?
- How would you implement offline mode?

28. Common Mistakes

- No CRDT/OT — conflicts.
- Synchronous PG writes per keystroke.

29. Optimization Ideas

- Batch small ops.
- Compress op payloads.

30. Final Summary

Google Docs at scale tests CRDT/OT algorithms. The core: each keystroke is an op; CRDT ensures convergence regardless of order. WebSocket per doc, in-memory state, op log in PG, snapshots in S3. Sticky routing by doc_id.

QUESTION 66

ADVANCED

Design a Distributed Counter

1. Problem Statement

Design a counter that increments across multiple servers. Used for view counts, like counts, and metrics. Must handle high write rate and eventually converge.

2. Functional Requirements

- Increment counter (atomic).
- Read current value.
- Per-key counters (e.g. per post, per user).
- Eventually consistent.

3. Non-functional Requirements

- Write throughput: 1M increments/sec per counter.
- Read latency < 100ms.
- Eventual convergence < 1s.

4. Capacity Estimation

1B counters, 10M writes/sec globally.

5. Back-of-the-envelope Math

Shard by counter_id. Per-shard: in-memory counter with periodic flush to DB. CRDT for cross-shard merge.

6. API Design

```
POST /counters/{id}/increment body: {amount: 1}
GET /counters/{id} -> {value}
```

7. Database Schema

```
counters(counter_id, value, updated_at) // in Redis + DB
```

8. High-Level Design (HLD)

Client → API Gateway → Counter Service → Redis (per-shard) + DB (persistent).

9. Low-Level Design (LLD)

Increment: atomic INCR in Redis (per shard). Periodic flush to DB. Read: return Redis value (may be slightly stale).

10. Architecture Diagram

Architecture Diagram

```
Client -> API Gw -> Counter Svc -> Redis (sharded) -> periodic flush -> PG
```

11. Data Flow Diagram

Data Flow Diagram

```
Increment -> INCR Redis -> ack. Periodic flush -> PG. Read -> return Redis.
```

12. Component Explanation

- **Counter Service:** Stateless, routes by counter_id.
- **Redis Cluster:** Sharded counters.
- **Flush Worker:** Periodic Redis -> PG.

13. Technology Stack

- **Language:** Go/Java.
- **Cache:** Redis Cluster.
- **DB:** PostgreSQL.

14. Database Choice

Redis for live counters. PG for durability.

15. Cache Strategy

Redis IS the counter store.

16. Load Balancer Strategy

L7 LB. Stateless service.

17. Message Queue Usage

Optional Kafka for counter events (analytics).

18. Scaling Strategy

Shard Redis by counter_id hash.

19. Fault Tolerance

Redis replication + failover. PG multi-AZ.

20. Bottlenecks

- Hot counter (single shard saturated) — client-side batching.

21. Trade-offs

- **Redis vs DB:** Redis is fast; DB is durable. Hybrid: Redis live + PG periodic.

22. CAP Theorem Analysis

AP. Eventual consistency.

23. Security Considerations

TLS. Auth. Rate limit increments.

24. Monitoring & Logging

Track increment rate, flush latency, Redis memory.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Use Redis atomic INCR.
- Mention periodic flush for durability.

27. Common Follow-up Questions

- How would you handle a hot counter (1M increments/sec on one key)?
- How would you implement per-user unique counters (cardinality)?

28. Common Mistakes

- DB-only — too slow.
- No periodic flush — data lost on Redis crash.

29. Optimization Ideas

- Batch increments client-side.
- Use HyperLogLog for unique counters.

30. Final Summary

A distributed counter tests atomic operations at scale. Redis atomic INCR handles writes; periodic flush to PG for durability. Shard by counter_id. The system is AP — eventual consistency acceptable for view/like counts.

QUESTION 67

ADVANCED

Design a Distributed Rate Limiter

1. Problem Statement

Design a rate limiter that works across multiple server instances. Limits per user, per API, per IP. Must be accurate under distributed conditions.

2. Functional Requirements

- Limit requests per (user, API, time_window).
- Multiple algorithms: token bucket, sliding window.
- Return 429 with Retry-After.
- Configurable limits.

3. Non-functional Requirements

- Latency overhead < 5ms.
- Accuracy > 99%.
- Availability 99.99%.

4. Capacity Estimation

10M req/sec globally. Limits per (user, API).

5. Back-of-the-envelope Math

Redis for shared state (atomic operations). Sliding window log or token bucket.

6. API Design

```
// Internal API (called by gateway):
boolean allowRequest(userId, api) -> {allowed, remaining, retry_after}
```

7. Database Schema

```
// Redis keys:
ratelimit:{user}:{api}:count // counter
ratelimit:{user}:{api}:tokens // token bucket
ratelimit:{user}:{api}:window_start
```

8. High-Level Design (HLD)

Gateway → Rate Limiter Service → Redis (shared state). Lua script for atomic check-and-increment.

9. Low-Level Design (LLD)

Token bucket: Lua script in Redis (atomic: check tokens, decrement or reject). Sliding window: sorted set of timestamps, remove old, count, add new.

10. Architecture Diagram

Architecture Diagram

```
Gateway -> Rate Limiter Svc -> Redis (Lua script, atomic)
```

11. Data Flow Diagram

Data Flow Diagram

```
Request -> gateway -> rate limiter -> Redis Lua script -> allow/deny -> gateway proceeds or 429.
```

12. Component Explanation

- **Rate Limiter Service:** Stateless, calls Redis.
- **Redis:** Shared state with Lua scripts.
- **Config Service:** Per-API limits.

13. Technology Stack

- **Language:** Go/Java.
- **Cache:** Redis Cluster.
- **Algorithm:** Token bucket / sliding window via Lua.

14. Database Choice

Redis only (ephemeral state).

15. Cache Strategy

Redis IS the rate limit store.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

None.

18. Scaling Strategy

Shard Redis by user_id hash.

19. Fault Tolerance

Redis replication + failover.

20. Bottlenecks

- Redis lookup per request — local cache + periodic sync.

21. Trade-offs

- **Token bucket vs sliding window:** Token allows bursts; sliding is precise.
- **Sync vs local:** Sync is accurate; local is fast.

22. CAP Theorem Analysis

AP. Eventual consistency acceptable (slight over-limit OK).

23. Security Considerations

TLS. Internal-only API.

24. Monitoring & Logging

Track reject rate, Redis latency.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Use Redis Lua script for atomicity.
- Discuss token bucket vs sliding window.

27. Common Follow-up Questions

- How would you handle Redis failure?
- How would you implement distributed rate limiting without Redis?

28. Common Mistakes

- Non-atomic check-and-increment — race condition.
- DB for state — too slow.

29. Optimization Ideas

- Local token bucket + periodic Redis sync.
- Batch checks (pipeline).

30. Final Summary

A distributed rate limiter tests atomic operations in a shared store. Redis Lua script provides atomicity (check + increment in one operation). Token bucket allows bursts; sliding window is precise. The system is AP — slight over-limit acceptable during failures.

QUESTION 68

ADVANCED

Design a Schema Registry

1. Problem Statement

Design a schema registry for Kafka. Producers/consumers register schemas (Avro, Protobuf); the registry enforces compatibility rules on schema evolution.

2. Functional Requirements

- Register schemas (subject, version, schema).
- Check compatibility (backward, forward, full).
- Fetch schema by subject + version.
- Soft delete (deprecate, not remove).

3. Non-functional Requirements

- Latency < 100ms.
- Availability 99.9%.
- Strong consistency (no incompatible schema registered).

4. Capacity Estimation

1M schemas, 10M lookups/day.

5. Back-of-the-envelope Math

PG for storage (small dataset). Kafka topic for events. Compatibility check on register.

6. API Design

```
POST /subjects/{subject}/versions body: {schema, type}
GET /subjects/{subject}/versions/{version}
POST /compatibility/subjects/{subject}/versions/{version} body: {schema}
```

7. Database Schema

```
schemas(schema_id, type, schema_text, created_at)
subject_versions(subject, version, schema_id, created_at)
```

8. High-Level Design (HLD)

Producer/Consumer → Schema Registry → PG + Kafka (events). Compatibility checker validates against previous version.

9. Low-Level Design (LLD)

Register: parse schema, check compatibility with latest version (backward/forward/full), reject if incompatible, insert, publish event.

10. Architecture Diagram

Architecture Diagram

```
Producer -> Schema Registry -> PG (schemas) + Kafka (events)
Consumer -> Schema Registry -> fetch schema
```

11. Data Flow Diagram

Data Flow Diagram

```
Register -> parse -> compatibility check vs latest -> insert if compatible -> publish event. Fetch ->
return schema.
```

12. Component Explanation

- **Schema Registry Service:** CRUD + compatibility check.
- **PG:** Schema storage.
- **Kafka:** Events for cache invalidation.

13. Technology Stack

- **Language:** Java.
- **DB:** PostgreSQL.
- **Cache:** Redis (hot schemas).

14. Database Choice

PG for ACID (compatibility check needs transaction).

15. Cache Strategy

Redis for hot schema lookups (consumers fetch often).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for schema events (cache invalidation).

18. Scaling Strategy

Read replicas for high read volume.

19. Fault Tolerance

Multi-AZ.

20. Bottlenecks

- Compatibility check latency — cache results.

21. Trade-offs

- **Strict vs lenient compatibility:** Strict is safer; lenient allows more evolution.

22. CAP Theorem Analysis

CP. Strong consistency (no incompatible schema registered).

23. Security Considerations

TLS. Auth. Audit log.

24. Monitoring & Logging

Track register latency, compatibility rejection rate.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss compatibility types (backward, forward, full).
- Mention caching for fetch.

27. Common Follow-up Questions

- How would you handle schema soft delete?
- How would you implement cross-registry replication?

28. Common Mistakes

- No compatibility check — breaks consumers.
- PG-only without cache — slow.

29. Optimization Ideas

- Cache hot schemas in Redis.
- Pre-compute compatibility results.

30. Final Summary

A schema registry tests compatibility enforcement. The core: register validates schema against previous version (backward/forward/full compatibility) before insert. PG for storage (ACID). Redis cache for hot lookups. Kafka events for cache invalidation. The system is CP — no incompatible schema can be registered.

QUESTION 69 ADVANCED

Design a Service Mesh (Istio/Linkerd)

1. Problem Statement

Design a service mesh: sidecar proxies handle service-to-service communication, with mTLS, traffic control, observability, and policy enforcement.

2. Functional Requirements

- Sidecar proxy per service instance.
- mTLS between services.
- Traffic control (routing, retries, timeouts, circuit breaking).
- Observability (metrics, tracing, logs).
- Policy enforcement (authZ, rate limit).

3. Non-functional Requirements

- Latency overhead < 1ms per hop.
- Availability 99.99%.
- No app code changes.

4. Capacity Estimation

1000 microservices, 10K sidecars, 1M req/sec.

5. Back-of-the-envelope Math

Sidecar (Envoy) per pod. Control plane (Istiod) configures sidecars. xDS protocol for config distribution.

6. API Design

```
// Control plane API (admin):
POST /virtualservices body: {routing rules}
POST /destinationrules body: {subsets, load balancing}
// Data plane: sidecar intercepts pod traffic
```

7. Database Schema

```
// Config (Kubernetes CRDs):
VirtualService {hosts, http: [{match, route, retries, timeout}]}
DestinationRule {host, subsets, trafficPolicy}
```

8. High-Level Design (HLD)

Control plane (Istiod) → Sidecars (Envoy). Sidecars intercept all pod traffic via iptables. xDS for config push.

9. Low-Level Design (LLD)

Sidecar (Envoy) runs in pod alongside app. iptables redirects all traffic through sidecar. Sidecar handles mTLS, routing, retries, metrics. Control plane watches K8s API + CRDs; pushes config via xDS.

10. Architecture Diagram

Architecture Diagram

```
Pod: [App] <-> [Envoy Sidecar] <-> Network
Control Plane (Istiod) --xDS--> all sidecars
```

11. Data Flow Diagram

Data Flow Diagram

```
Request -> app -> sidecar (mTLS, route, retry) -> remote sidecar -> remote app. Metrics + traces
emitted.
```

12. Component Explanation

- **Control Plane (Istiod):** Configures sidecars via xDS.
- **Sidecar (Envoy):** Intercepts traffic; applies policies.
- **CA:** Issues mTLS certs.
- **Telemetry:** Collects metrics + traces.

13. Technology Stack

- **Language:** Go (control), C++ (Envoy).
- **Config:** Kubernetes CRDs.
- **Protocol:** xDS (Envoy discovery).

14. Database Choice

No DB. Kubernetes API + etcd for config.

15. Cache Strategy

Sidecar in-memory config.

16. Load Balancer Strategy

Sidecar does L7 LB.

17. Message Queue Usage

None.

18. Scaling Strategy

Sidecar scales with pods. Control plane scales horizontally.

19. Fault Tolerance

Sidecar failure = pod failure. Control plane HA.

20. Bottlenecks

- Sidecar overhead (CPU, memory) — tune resource limits.

21. Trade-offs

- **Sidecar vs node-level proxy:** Sidecar is granular; node is cheaper.

22. CAP Theorem Analysis

AP. Sidecars continue with cached config if control plane down.

23. Security Considerations

mTLS automatically. Cert rotation. AuthZ policies.

24. Monitoring & Logging

Track sidecar latency overhead, request success rate.

25. Deployment Strategy

Kubernetes. Sidecar injected via admission webhook.

26. Interview Tips

- Discuss sidecar pattern.
- Mention xDS for config distribution.

27. Common Follow-up Questions

- How would you implement canary deployments?
- How would you handle mTLS cert rotation?

28. Common Mistakes

- App handles mTLS — couples to mesh.
- Synchronous xDS — blocks sidecar.

29. Optimization Ideas

- Tune sidecar resources.
- Batch telemetry emission.

30. Final Summary

A service mesh tests the sidecar pattern. Sidecars (Envoy) handle service-to-service concerns (mTLS, routing, observability) without app code changes. Control plane (Istiod) pushes config via xDS. The architecture rewards understanding of L7 proxying and config distribution.

QUESTION 70

ADVANCED

Design a Multi-Region Database

1. Problem Statement

Design a database that spans multiple regions for low latency globally + DR. Handle cross-region writes, conflict resolution, and failover.

2. Functional Requirements

- Reads/writes in each region.
- Cross-region replication.
- Automatic failover on region failure.
- Conflict resolution for concurrent writes.

3. Non-functional Requirements

- Read latency < 50ms locally.
- Write latency < 100ms locally (async replication).
- Availability 99.99%.
- Survives region failure.

4. Capacity Estimation

10TB data per region. 1M writes/sec globally.

5. Back-of-the-envelope Math

Multi-leader (each region accepts writes) with async replication. Conflict resolution via CRDT or last-write-wins (LWW).

6. API Design

```
// Standard DB API, but routed to nearest region
PUT /data/{key} body: {value}
GET /data/{key}
```

7. Database Schema

```
// Per-region DB with replication log:
Table: data (key, value, version, origin_region, timestamp)
Table: replication_log (log_id, region, op_type, key, value, timestamp)
```

8. High-Level Design (HLD)

App → Local Region DB → async replication to other regions. Conflict resolver merges concurrent writes.

9. Low-Level Design (LLD)

Write: local DB + append to replication log. Replicator ships log to other regions. Conflict resolver merges based on CRDT or LWW (clock skew tolerant).

10. Architecture Diagram

Architecture Diagram

```
Region A: App -> DB A
Region B: App -> DB B
DB A <--async replication--> DB B
Conflict Resolver merges concurrent writes
```

11. Data Flow Diagram

Data Flow Diagram

```
Write A -> DB A + log. Replicator ships log to B. B applies; conflict resolver merges. Read B -> local DB (may be slightly stale).
```

12. Component Explanation

- **Per-Region DB:** Local reads/writes.
- **Replicator:** Ships log across regions.
- **Conflict Resolver:** Merges concurrent writes.
- **Failover Coordinator:** Detects region failure; redirects traffic.

13. Technology Stack

- **Language:** C/C++ (DB).
- **Algorithm:** CRDT or LWW.
- **Coordination:** Paxos/Raft for failover.

14. Database Choice

Custom multi-region DB (CockroachDB, Spanner, Cassandra).

15. Cache Strategy

Local cache per region.

16. Load Balancer Strategy

Geographic routing to nearest region.

17. Message Queue Usage

Replication log (custom or Kafka).

18. Scaling Strategy

Add regions for global coverage. Shard within region.

19. Fault Tolerance

Region failure: failover coordinator redirects traffic. Other regions continue.

20. Bottlenecks

- Cross-region replication bandwidth — compress + batch.
- Conflict resolution cost — CRDT is O(1); LWW is simple.

21. Trade-offs

- **Multi-leader vs single-leader:** Multi has low write latency; single has strong consistency.
- **CRDT vs LWW:** CRDT preserves all writes; LWW loses concurrent writes.

22. CAP Theorem Analysis

AP. During region partition, accepts writes; resolves on reconnect.

23. Security Considerations

TLS + mTLS between regions. Encryption at rest.

24. Monitoring & Logging

Track replication lag, conflict rate, failover events.

25. Deployment Strategy

Multi-region. Per-region HA.

26. Interview Tips

- Discuss multi-leader vs single-leader.
- Mention conflict resolution (CRDT, LWW).

27. Common Follow-up Questions

- How would you handle a network split between regions?
- How would you implement strong consistency globally (Spanner)?

28. Common Mistakes

- Single-leader globally — high write latency.
- No conflict resolution — lost writes.

29. Optimization Ideas

- Compress replication log.
- Batch cross-region shipping.
- Use CRDT for conflict-free merging.

30. Final Summary

A multi-region DB tests cross-region replication + conflict resolution. Multi-leader (each region accepts writes) gives low write latency but requires conflict resolution (CRDT or LWW). Single-leader (Spanner) gives strong consistency but higher write latency. The system is AP — accepts writes during partition, resolves on reconnect.

QUESTION 71

ADVANCED

Design Distributed Tracing (Jaeger/Zipkin)

1. Problem Statement

Design a distributed tracing system. Trace requests across microservices; visualize call graph; identify slow spans.

2. Functional Requirements

- Instrument services with trace IDs.
- Collect spans from all services.
- Store + query traces.
- Visualize call graph + latency breakdown.

3. Non-functional Requirements

- Trace sampling (1-100% configurable).
- Query latency < 1s.
- Storage: 7-30 day retention.

4. Capacity Estimation

1B traces/day. 100 spans/trace avg = 100B spans/day. Storage: 100B × 100B = 10TB/day.

5. Back-of-the-envelope Math

OpenTelemetry SDK in each service. Spans reported to collector via gRPC. Collector batches + stores in Cassandra/ES.

6. API Design

```
POST /spans body: [{trace_id, span_id, parent_id, service, operation, start, duration,
tags}]
GET /traces/{trace_id}
GET /traces?service=...&operation=...&minDuration=...
```

7. Database Schema

```
spans(trace_id, span_id, parent_span_id, service_name, operation_name, start_time,
duration, tags_json)
  PARTITION BY trace_id
```

8. High-Level Design (HLD)

Service (SDK) → Collector → Storage (Cassandra/ES). UI queries storage for visualization.

9. Low-Level Design (LLD)

SDK in service: create span per operation, propagate trace_id via headers (W3C Trace Context). Async report to collector. Collector batches + writes to storage.

10. Architecture Diagram

Architecture Diagram

```
Service A (SDK) --spans--> Collector --batch--> Cassandra
Service B (SDK) --spans--> Collector
UI <--queries-- Cassandra
```

11. Data Flow Diagram

Data Flow Diagram

```
Request enters service A; SDK creates span; trace_id propagated via headers to B. B creates child span. Both async report to collector. Collector batches + stores.
```

12. Component Explanation

- **SDK:** Per-service; creates spans; reports.
- **Collector:** Receives spans; batches; stores.
- **Storage:** Cassandra or ES.
- **UI:** Query + visualize.

13. Technology Stack

- **Language:** Go (collector), any (SDK).
- **DB:** Cassandra (write-heavy).
- **SDK:** OpenTelemetry.

14. Database Choice

Cassandra (write-heavy, partitioned by trace_id).

15. Cache Strategy

None.

16. Load Balancer Strategy

L7 LB to collectors.

17. Message Queue Usage

Optional Kafka between SDK and collector (buffer).

18. Scaling Strategy

Collectors scale horizontally. Cassandra scales by trace volume.

19. Fault Tolerance

Multi-AZ. Spans lost on collector failure (acceptable due to sampling).

20. Bottlenecks

- Span write throughput — batch + Cassandra.
- Query latency for large traces — limit span count.

21. Trade-offs

- **Sample rate:** High = better visibility; low = less overhead.
- **Cassandra vs ES:** Cassandra is write-optimized; ES is query-optimized.

22. CAP Theorem Analysis

AP. Spans may be lost; traces still useful.

23. Security Considerations

TLS. PII scrubbing in spans.

24. Monitoring & Logging

Track span ingest rate, query latency, drop rate.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss OpenTelemetry SDK.
- Mention sampling (1% or 100% of errors).

27. Common Follow-up Questions

- How would you implement 100% sampling without overhead?
- How would you integrate with service mesh?

28. Common Mistakes

- No sampling — storage explodes.
- Synchronous span reporting — blocks app.

29. Optimization Ideas

- Batch span reports.
- Compress spans.
- Use adaptive sampling (more for errors).

30. Final Summary

Distributed tracing tests observability at scale. SDK in each service creates spans; trace_id propagates via headers. Collector batches + stores in Cassandra. Sampling (1-100%) controls overhead. The system is AP — span loss acceptable.

QUESTION 72 ADVANCED

Design a Logging Pipeline (ELK)

1. Problem Statement

Design a centralized logging pipeline. Services emit logs; pipeline collects, parses, indexes, and exposes for search + visualization.

2. Functional Requirements

- Collect logs from all services.
- Parse + enrich (add metadata).
- Index for search.
- Visualize (dashboards).
- Alert on patterns.

3. Non-functional Requirements

- Log ingestion: 1M events/sec.
- Search latency < 1s.
- Retention: 7-30 days hot, 1 year cold.

4. Capacity Estimation

1M logs/sec. 100B logs/day. Storage: 100B × 500B = 50TB/day.

5. Back-of-the-envelope Math

Fluentd/Fluent Bit on each node → Kafka buffer → Logstash parser → Elasticsearch index → Kibana UI.

6. API Design

```
POST /logs body: {service, level, message, ...}
GET /logs/search?q=...&from=...&to=...
```

7. Database Schema

```
// ES index per day:
logs-{date} {
  timestamp, service, level, message, trace_id, fields...
}
```

8. High-Level Design (HLD)

Service → Fluent Bit (node agent) → Kafka → Logstash → Elasticsearch → Kibana.

9. Low-Level Design (LLD)

Service writes logs to stdout. Fluent Bit collects, sends to Kafka. Logstash consumes, parses, enriches, writes to ES. Kibana queries ES.

10. Architecture Diagram

Architecture Diagram

```
Service -> Fluent Bit -> Kafka -> Logstash -> Elasticsearch <- Kibana
```

11. Data Flow Diagram

Data Flow Diagram

```
Log emit -> stdout -> Fluent Bit collects -> Kafka buffer -> Logstash parse + enrich -> ES index ->
Kibana query.
```

12. Component Explanation

- **Fluent Bit:** Node agent; collects logs.
- **Kafka:** Buffer for burst tolerance.
- **Logstash:** Parse + enrich.
- **Elasticsearch:** Index + search.
- **Kibana:** UI.

13. Technology Stack

- **Language:** Java (Logstash), Ruby (Fluentd).
- **Queue:** Kafka.
- **Search:** Elasticsearch.
- **UI:** Kibana.

14. Database Choice

Elasticsearch (search-optimized).

15. Cache Strategy

None.

16. Load Balancer Strategy

L7 LB to collectors.

17. Message Queue Usage

Kafka for ingestion buffer.

18. Scaling Strategy

Kafka scales by partition. ES scales by shard. Logstash scales horizontally.

19. Fault Tolerance

Multi-AZ. Kafka persists during ES outage.

20. Bottlenecks

- ES indexing throughput — shard + tune refresh interval.
- Storage cost — lifecycle to cold storage.

21. Trade-offs

- **Hot vs cold storage:** Hot is fast; cold is cheap. Tiered storage.

22. CAP Theorem Analysis

AP. Logs may be lost on pipeline failure (acceptable).

23. Security Considerations

TLS. PII scrubbing. Auth for Kibana.

24. Monitoring & Logging

Track ingestion rate, ES indexing lag, query latency.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss Kafka buffer for burst tolerance.
- Mention ES lifecycle (hot/warm/cold).

27. Common Follow-up Questions

- How would you handle PII in logs?
- How would you implement log-based alerting?

28. Common Mistakes

- No Kafka buffer — ES crushed on burst.
- No lifecycle — storage explodes.

29. Optimization Ideas

- Compress logs.
- Batch ES writes.
- Use hot-warm-cold architecture.

30. Final Summary

A logging pipeline tests ingest + search at scale. Fluent Bit collects → Kafka buffers → Logstash parses → ES indexes → Kibana visualizes. Kafka buffer tolerates bursts. ES lifecycle (hot/warm/cold) controls storage cost.

QUESTION 73

ADVANCED

Design a Metrics Platform (Prometheus)

1. Problem Statement

Design a metrics collection + alerting system. Services expose metrics; platform scrapes, stores, queries, and alerts.

2. Functional Requirements

- Scrape metrics from services (pull model).
- Store time-series data.
- Query via PromQL.
- Alert on thresholds + anomalies.

3. Non-functional Requirements

- Scrape interval: 15-60s.
- Query latency < 1s.
- Retention: 15 days hot, 1 year downsampled.

4. Capacity Estimation

10M series, 1M samples/sec. Storage: 1M × 60/min × 60 × 24 × 15 = 1.3B samples.

5. Back-of-the-envelope Math

Pull model (Prometheus scrapes /metrics endpoint). Time-series DB (TSDB) with compression. Alertmanager for alerts.

6. API Design

```
GET /metrics (service exposes)
// PromQL query:
rate(http_requests_total[5m]) > 100
```

7. Database Schema

```
// TSDB storage:
series(metric_name, labels, samples: [(timestamp, value)])
  Indexed by metric_name + labels
```

8. High-Level Design (HLD)

Service exposes /metrics → Prometheus scrapes → TSDB → Alertmanager + Grafana UI.

9. Low-Level Design (LLD)

Prometheus scrapes /metrics at configured interval. Stores as time-series with labels. Alertmanager evaluates rules; sends alerts to PagerDuty/Slack.

10. Architecture Diagram

Architecture Diagram

```
Service --/metrics--> Prometheus --scrape--> TSDB <--queries-- Grafana
Prometheus --alerts--> Alertmanager --> PagerDuty
```

11. Data Flow Diagram

Data Flow Diagram

```
Service exposes metrics. Prometheus scrapes every 15s. Stores in TSDB. Grafana queries for
dashboards. Alertmanager evaluates rules; fires alerts.
```

12. Component Explanation

- **Prometheus:** Scrapes + stores.
- **Alertmanager:** Evaluates rules; sends alerts.
- **Grafana:** UI.
- **TSDB:** Time-series storage.

13. Technology Stack

- **Language:** Go.
- **DB:** Custom TSDB (Prometheus) or VictoriaMetrics.

14. Database Choice

Custom TSDB (optimized for time-series).

15. Cache Strategy

None.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

None (pull model).

18. Scaling Strategy

Federation (multiple Prometheus instances). VictoriaMetrics for horizontal scaling.

19. Fault Tolerance

Multi-AZ. Federation provides redundancy.

20. Bottlenecks

- High-cardinality labels — explosion of series.
- Long retention — downsample.

21. Trade-offs

- **Pull vs push:** Pull is simpler (Prometheus discovers services); push is more flexible.

22. CAP Theorem Analysis

AP. Metrics may be lost on Prometheus failure (acceptable).

23. Security Considerations

TLS. Auth for UI + alerting.

24. Monitoring & Logging

Track scrape latency, TSDB size, alert latency.

25. Deployment Strategy

Multi-AZ. Federation for scale.

26. Interview Tips

- Discuss pull model (Prometheus scrapes).
- Mention high-cardinality pitfall.

27. Common Follow-up Questions

- How would you handle high-cardinality metrics?
- How would you implement long-term storage?

28. Common Mistakes

- High-cardinality labels — series explosion.
- No retention limit — storage explodes.

29. Optimization Ideas

- Downsample old data.
- Use VictoriaMetrics for horizontal scaling.
- Avoid high-cardinality labels.

30. Final Summary

A metrics platform tests time-series data at scale. Pull model (Prometheus scrapes /metrics). Custom TSDB optimized for time-series. Alertmanager fires alerts. Federation scales. Avoid high-cardinality labels (explosion of series).

QUESTION 74

ADVANCED

Design an Auto-Scaler (HPA + VPA + Cluster Autoscaler)

1. Problem Statement

Design the Kubernetes auto-scaling system: HPA (Horizontal Pod Autoscaler) scales Pod count by load; VPA scales Pod resource requests; Cluster Autoscaler adds nodes.

2. Functional Requirements

- HPA: scale Pod count by CPU/memory/custom metrics.
- VPA: adjust Pod resource requests.
- Cluster Autoscaler: add/remove nodes based on pending Pods.
- Cooldown to avoid thrashing.

3. Non-functional Requirements

- Scale-up latency < 60s.
- Scale-down latency < 5min.
- Avoid thrashing.

4. Capacity Estimation

1000 nodes, 10K Pods. Scale events: 100/min.

5. Back-of-the-envelope Math

HPA: query metrics API every 30s; compute desired replicas; PATCH deployment. Cluster Autoscaler: watch pending Pods; provision nodes.

6. API Design

```
// Internal: HPA controller watches metrics API
GET /apis/metrics.k8s.io/v1beta1/pods
// Then PATCH deployments/scale
```

7. Database Schema

```
HPA {deployment, minReplicas, maxReplicas, targetCPU, currentReplicas, desiredReplicas}
```

8. High-Level Design (HLD)

Metrics Server → HPA Controller → API Server (PATCH scale). Cluster Autoscaler watches pending Pods → cloud provider API (provision node).

9. Low-Level Design (LLD)

HPA: every 30s, query metrics (CPU/memory per Pod). Compute desired = current × (current_metric / target_metric). Clamp to [min, max]. PATCH deployment scale. Cooldown to prevent thrashing.

10. Architecture Diagram

Architecture Diagram

```
Metrics Server <-scrape-- Pods
HPA Controller --query--> Metrics Server --PATCH--> API Server --scale--> Deployment
Cluster Autoscaler --watch pending Pods--> Cloud API --> new node
```

11. Data Flow Diagram

Data Flow Diagram

```
Pods expose metrics -> Metrics Server scrapes -> HPA queries every 30s -> computes desired -> PATCH
deployment -> new Pods created -> if pending, Cluster Autoscaler provisions node.
```

12. Component Explanation

- **Metrics Server:** Scrapes Pod metrics.
- **HPA Controller:** Computes + applies scale.
- **VPA Controller:** Adjusts resource requests.
- **Cluster Autoscaler:** Provisions nodes.

13. Technology Stack

- **Language:** Go.
- **Coordination:** API server + etcd.

14. Database Choice

No DB. etcd via API server.

15. Cache Strategy

In-memory metrics cache.

16. Load Balancer Strategy

Leader election (single active controller).

17. Message Queue Usage

None.

18. Scaling Strategy

Single controller handles 1000 nodes. For larger: shard.

19. Fault Tolerance

HA via leader election.

20. Bottlenecks

- Metrics API latency — cache.
- Node provisioning latency — pre-warm.

21. Trade-offs

- **Aggressive vs conservative scaling:** Aggressive is responsive; conservative saves cost.

22. CAP Theorem Analysis

CP. etcd is source of truth.

23. Security Considerations

TLS. RBAC.

24. Monitoring & Logging

Track scale event latency, thrashing rate.

25. Deployment Strategy

HA (leader election).

26. Interview Tips

- Discuss HPA, VPA, Cluster Autoscaler separately.
- Mention cooldown to avoid thrashing.

27. Common Follow-up Questions

- How would you implement predictive auto-scaling (ML)?
- How would you handle scale-down gracefully (drain)?

28. Common Mistakes

- No cooldown — thrashing.
- Scale-down too aggressive — kills healthy Pods.

29. Optimization Ideas

- Pre-warm node pools.
- Predictive scaling (ML).
- Batch scale decisions.

30. Final Summary

Auto-scaling tests feedback loops. HPA scales Pod count by metrics. VPA adjusts requests. Cluster Autoscaler provisions nodes. Cooldown prevents thrashing. The architecture rewards understanding of control theory + K8s APIs.

QUESTION 75

ADVANCED

Design a Distributed Job Scheduler

1. Problem Statement

Design a distributed job scheduler. Schedule one-off and recurring jobs; distribute across workers; handle retries, dependencies, and failures.

2. Functional Requirements

- Schedule one-off + recurring jobs (cron).
- Distribute jobs across workers.
- Job dependencies (DAG).
- Retries with backoff.
- Failure handling.

3. Non-functional Requirements

- Schedule latency < 1s.
- Throughput: 100K jobs/sec.
- Availability 99.9%.

4. Capacity Estimation

1B jobs/day. 10K workers.

5. Back-of-the-envelope Math

Job queue (Kafka) + scheduler + workers. Cron-like trigger service. DAG executor for dependencies.

6. API Design

```
POST /jobs body: {type, payload, schedule?} -> {job_id}
POST /jobs/dag body: {tasks, dependencies}
GET /jobs/{id}/status
```

7. Database Schema

```
jobs(job_id, type, payload, status, scheduled_at, started_at, completed_at, retries)
job_dag(dag_id, tasks: [{task_id, depends_on: [...]}])
```

8. High-Level Design (HLD)

Scheduler → Job Queue (Kafka) → Workers. DAG Executor for dependencies. Job Store (PG) for state.

9. Low-Level Design (LLD)

Scheduler: trigger due jobs, publish to Kafka. Workers: consume, execute, update status. DAG: topological sort; execute ready tasks; mark complete; trigger dependents.

10. Architecture Diagram

Architecture Diagram

```
Scheduler -> Kafka -> Workers -> update PG
DAG Executor: topological sort -> execute ready -> trigger dependents
```

11. Data Flow Diagram

Data Flow Diagram

```
Job scheduled -> at trigger time, publish to Kafka -> worker consumes -> executes -> updates status.
DAG: task completes -> trigger dependents.
```

12. Component Explanation

- **Scheduler:** Triggers due jobs.
- **Job Queue:** Kafka.
- **Workers:** Execute jobs.
- **DAG Executor:** Manages dependencies.
- **Job Store:** PG for state.

13. Technology Stack

- **Language:** Go/Java.
- **DB:** PostgreSQL.
- **Queue:** Kafka.

14. Database Choice

PG for ACID job state. Kafka for queue.

15. Cache Strategy

Redis for hot job configs.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for job queue.

18. Scaling Strategy

Workers scale by queue depth. Scheduler HA via leader election.

19. Fault Tolerance

Multi-AZ. Retries with backoff. Dead letter queue for permanent failures.

20. Bottlenecks

- DAG with many tasks — topological sort cost.
- Worker startup latency — pre-warm pool.

21. Trade-offs

- **At-least-once vs exactly-once:** At-least-once is simpler; exactly-once requires idempotent workers.

22. CAP Theorem Analysis

AP. Eventual consistency on job state.

23. Security Considerations

TLS. Auth. Job payload encryption.

24. Monitoring & Logging

Track job latency, failure rate, queue depth.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss DAG executor for dependencies.
- Mention retries + dead letter queue.

27. Common Follow-up Questions

- How would you handle long-running jobs (hours)?
- How would you implement job priorities?

28. Common Mistakes

- No retry — transient failures become permanent.
- Synchronous DAG execution.

29. Optimization Ideas

- Pre-warm worker pool.
- Batch job dispatch.
- Use Airflow for complex DAGs.

30. Final Summary

A distributed job scheduler tests queue + worker + DAG patterns. Scheduler triggers due jobs → Kafka → workers execute. DAG executor handles dependencies (topological sort). Retries with backoff. The architecture rewards understanding of distributed task execution.

QUESTION 76

ADVANCED

Design a Configuration Center (Apollo/etcd)

1. Problem Statement

Design a centralized configuration service. Services fetch config at startup + receive live updates without restart.

2. Functional Requirements

- Store config per service/environment.
- Fetch config at startup.
- Push live updates (watch).
- Version history + rollback.
- Audit log.

3. Non-functional Requirements

- Fetch latency < 100ms.
- Update propagation < 5s.
- Availability 99.99%.

4. Capacity Estimation

10K services, 100K config keys. 1M watches.

5. Back-of-the-envelope Math

etcd (Raft consensus) for storage. Watch API for live updates (long poll or gRPC stream).

6. API Design

```
GET /config/{service}/{env}
PUT /config/{service}/{env} body: {key: value}
// Watch (long poll or gRPC stream)
GET /watch?prefix=/service/
```

7. Database Schema

```
// etcd-style KV store:
/service/{name}/{env}/{key} = value
```

8. High-Level Design (HLD)

Service → Config Service (etcd) → Watch (gRPC stream). Config changes push to all watchers.

9. Low-Level Design (LLD)

Service fetches full config at startup. Establishes watch (gRPC stream) for prefix. Config change in etcd → watch fires → service receives update.

10. Architecture Diagram

Architecture Diagram

```
Service --fetch--> Config Service (etcd)
Service <--watch (gRPC stream)-- Config Service
```

11. Data Flow Diagram

Data Flow Diagram

```
Startup: fetch config. Watch: gRPC stream for prefix. Change: etcd update -> watch fires -> service
receives delta.
```

12. Component Explanation

- **Config Service:** etcd-backed KV store.
- **Watch API:** gRPC stream for live updates.
- **Admin UI:** Config CRUD + version history.

13. Technology Stack

- **Language:** Go.
- **DB:** etcd (Raft consensus).

14. Database Choice

etcd (CP, Raft consensus).

15. Cache Strategy

In-memory cache in service (with watch invalidation).

16. Load Balancer Strategy

L7 LB to etcd cluster.

17. Message Queue Usage

None (watch is push).

18. Scaling Strategy

etcd cluster of 3-7 nodes. Read replicas for scale.

19. Fault Tolerance

Raft consensus survives node failure.

20. Bottlenecks

- Watch connection count — shard by prefix.

21. Trade-offs

- **Push vs poll:** Push is real-time; poll is simpler.

22. CAP Theorem Analysis

CP. Strong consistency via Raft.

23. Security Considerations

TLS. Auth. Audit log.

24. Monitoring & Logging

Track fetch latency, watch latency, update propagation.

25. Deployment Strategy

Multi-AZ etcd cluster.

26. Interview Tips

- Discuss etcd + Raft.
- Mention watch (gRPC stream) for live updates.

27. Common Follow-up Questions

- How would you handle config rollback?
- How would you implement config encryption?

28. Common Mistakes

- Polling — high latency.
- No version history — can't rollback.

29. Optimization Ideas

- Batch watch subscriptions.
- Compress config payloads.

30. Final Summary

A config center tests KV store + watch pattern. etcd (Raft consensus) provides CP storage. Watch API (gRPC stream) pushes live updates. The system is CP — strong consistency via Raft.

QUESTION 77

ADVANCED

Design a Service Registry (Consul/etcd)

1. Problem Statement

Design a service registry. Services register at startup; clients discover healthy instances; health checks remove unhealthy ones.

2. Functional Requirements

- Register service instances (name, IP, port).
- Health checks (HTTP, TCP).
- Discover healthy instances.
- DNS + HTTP API.

3. Non-functional Requirements

- Discovery latency < 50ms.
- Health check propagation < 10s.
- Availability 99.99%.

4. Capacity Estimation

10K services, 100K instances, 1M lookups/sec.

5. Back-of-the-envelope Math

etcd/Consul (Raft) for storage. Health checks run on agents. DNS interface for discovery.

6. API Design

```
PUT /service/{name} body: {ip, port, health_check}
GET /service/{name}/healthy -> [{ip, port}]
// DNS: {name}.service.consul -> A records
```

7. Database Schema

```
// KV store:
/service/{name}/{instance_id} = {ip, port, status, last_heartbeat}
```

8. High-Level Design (HLD)

Service → Agent (per-node) → Registry (etcd/Consul). Client → DNS or HTTP API → Registry.

9. Low-Level Design (LLD)

Service registers with local agent. Agent runs health checks; updates registry. Client queries DNS (cached) or HTTP API.

10. Architecture Diagram

Architecture Diagram

```
Service -> Agent (per node) -> Registry (Consul/etcd)
Client -> DNS or HTTP -> Registry -> healthy instances
```

11. Data Flow Diagram

Data Flow Diagram

```
Register: service -> agent -> registry. Health: agent checks -> updates registry. Discover: client ->
DNS/HTTP -> registry -> healthy instances.
```

12. Component Explanation

- **Agent:** Per-node; health checks.
- **Registry:** Consul/etcd cluster.
- **DNS Interface:** For legacy clients.

13. Technology Stack

- **Language:** Go.
- **DB:** Consul or etcd (Raft).

14. Database Choice

Consul or etcd (CP via Raft).

15. Cache Strategy

DNS cache (TTL 30s).

16. Load Balancer Strategy

L7 LB. DNS for clients.

17. Message Queue Usage

None.

18. Scaling Strategy

Consul cluster of 3-5 nodes. Agents scale with nodes.

19. Fault Tolerance

Raft consensus. Agent failure = service deregistered.

20. Bottlenecks

- Health check latency — parallelize.

21. Trade-offs

- **Push (long poll) vs DNS:** Push is real-time; DNS is cached.

22. CAP Theorem Analysis

CP. Strong consistency via Raft.

23. Security Considerations

TLS. mTLS to agents. ACLs.

24. Monitoring & Logging

Track registration latency, health check propagation.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss agent pattern (per-node).
- Mention DNS interface for legacy clients.

27. Common Follow-up Questions

- How would you handle network partition?
- How would you implement service mesh integration?

28. Common Mistakes

- No health checks — stale instances.
- Single registry — SPOF.

29. Optimization Ideas

- Parallel health checks.
- DNS caching.

30. Final Summary

A service registry tests KV store + health checks. Consul/etcd (Raft) for storage. Per-node agents run health checks. DNS + HTTP API for discovery. The system is CP — strong consistency via Raft.

QUESTION 78

ADVANCED

Design a Key-Value Store (DynamoDB-style)

1. Problem Statement

Design a distributed KV store. PUT/GET/DELETE keys; horizontally scalable; tunable consistency; survives node failures.

2. Functional Requirements

- PUT/GET/DELETE keys.
- Tunable consistency (eventual/strong).
- Automatic sharding.
- Replication + failover.

3. Non-functional Requirements

- Latency: p99 < 10ms.
- Availability 99.99%.
- Throughput: 1M ops/sec.

4. Capacity Estimation

100TB data, 1B keys.

5. Back-of-the-envelope Math

Consistent hashing for sharding. Replication (RF=3). Quorum reads/writes (R+W > N for strong consistency).

6. API Design

```
PUT /data/{key} body: {value}
GET /data/{key}
DELETE /data/{key}
```

7. Database Schema

```
// Per-node storage:
{key, value, version, timestamp}
```

8. High-Level Design (HLD)

Client → Coordinator Node → Replicas (via consistent hashing). Quorum protocol for reads/writes.

9. Low-Level Design (LLD)

PUT: coordinator hashes key → finds N replicas. Sends to all N. When W ack, return success. GET: coordinator asks R replicas, returns latest version.

10. Architecture Diagram

Architecture Diagram

```
Client -> Coordinator -> hash(key) -> [N1, N2, N3]
PUT: send to all 3, ack when W=2 ack
GET: ask all 3, return when R=2 respond
```

11. Data Flow Diagram

Data Flow Diagram

```
PUT: coordinator -> replicas -> W acks -> success. GET: coordinator -> replicas -> R responses ->
latest version.
```

12. Component Explanation

- **Coordinator:** Routes request to replicas.
- **Replicas:** Store data.
- **Gossip:** Cluster membership.

13. Technology Stack

- **Language:** Java (Cassandra, DynamoDB).
- **Algorithm:** Consistent hashing + quorum.

14. Database Choice

Custom (LSM-tree storage).

15. Cache Strategy

In-memory cache + bloom filters.

16. Load Balancer Strategy

Client-side routing (consistent hashing).

17. Message Queue Usage

None.

18. Scaling Strategy

Add nodes; rebalance partitions via consistent hashing.

19. Fault Tolerance

Replication (RF=3). Quorum ensures availability if R/W nodes available.

20. Bottlenecks

- Hot keys — rate limit.
- Compaction (LSM) — background.

21. Trade-offs

- **R + W > N:** Strong consistency. **R + W <= N:** Eventual.

22. CAP Theorem Analysis

AP. Tunable via R/W quorum.

23. Security Considerations

TLS. Auth. Encryption at rest.

24. Monitoring & Logging

Track latency, replication lag, compaction.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss consistent hashing + quorum.
- Mention tunable consistency (R/W).

27. Common Follow-up Questions

- How would you handle hot partitions?
- How would you implement transactions?

28. Common Mistakes

- No replication — data lost.
- Synchronous quorum — slow.

29. Optimization Ideas

- Bloom filters for fast key lookup.
- LSM-tree for write throughput.
- Read repair for consistency.

30. Final Summary

A distributed KV store tests consistent hashing + quorum. Coordinator routes to N replicas via consistent hashing. PUT: W acks. GET: R responses. R+W > N for strong consistency; R+W <= N for eventual. The system is AP with tunable consistency.

QUESTION 79 ADVANCED

Design Object Storage (S3)

1. Problem Statement

Design an object storage service. Store arbitrary blobs (images, backups, logs) with high durability (11 nines), versioning, and lifecycle policies.

2. Functional Requirements

- PUT/GET/DELETE objects.
- Bucket organization.
- Versioning.
- Lifecycle policies (transition to cold storage, expire).
- Access control (ACLs, policies).

3. Non-functional Requirements

- Durability 99.999999999% (11 nines).
- Availability 99.99%.
- Latency: < 100ms for small objects.

4. Capacity Estimation

100PB+ per customer. Trillions of objects globally.

5. Back-of-the-envelope Math

Erasure coding (Reed-Solomon) across many disks + facilities. Metadata in distributed KV store.

6. API Design

```
PUT /bucket/{key} body: bytes
GET /bucket/{key}
DELETE /bucket/{key}
PUT /bucket/{key}?versioning
```

7. Database Schema

```
// Metadata store:
bucket/{key}/{version} = {data_nodes, erasure_fragments, size, content_type, created_at}
```

8. High-Level Design (HLD)

Client → API Gateway → Metadata Service (KV store) + Data Nodes (erasure-coded).

9. Low-Level Design (LLD)

PUT: chunk object, erasure-code into fragments, distribute across data nodes, store metadata. GET: fetch fragments, reconstruct, return.

10. Architecture Diagram

Architecture Diagram

```
Client -> API Gw -> Metadata Svc (KV) + Data Nodes (erasure-coded)
PUT: chunk + erasure code -> distribute
GET: fetch fragments -> reconstruct
```

11. Data Flow Diagram

Data Flow Diagram

```
PUT: chunk -> erasure code (e.g. 10+4) -> distribute to data nodes -> metadata. GET: fetch fragments
-> reconstruct -> return.
```

12. Component Explanation

- **Metadata Service:** KV store for object metadata.
- **Data Nodes:** Store erasure-coded fragments.
- **Erasure Coder:** Reed-Solomon.
- **Lifecycle Manager:** Transition + expire.

13. Technology Stack

- **Language:** C/C++/Rust.
- **Algorithm:** Erasure coding (Reed-Solomon).

14. Database Choice

Custom metadata KV store (Cassandra-style).

15. Cache Strategy

CDN for hot objects.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for lifecycle events.

18. Scaling Strategy

Add data nodes; rebalance fragments.

19. Fault Tolerance

Erasure coding (10+4 survives 4 failures). Multi-AZ replication.

20. Bottlenecks

- Erasure coding CPU — background.
- Metadata lookup — cache.

21. Trade-offs

- **Erasure coding vs replication:** EC saves space (1.5x vs 3x); replication is faster.

22. CAP Theorem Analysis

AP. Read-your-writes via metadata service.

23. Security Considerations

TLS. Bucket policies. Encryption at rest (SSE). Audit log.

24. Monitoring & Logging

Track PUT/GET latency, durability, availability.

25. Deployment Strategy

Multi-AZ. Multi-region for cross-region replication.

26. Interview Tips

- Discuss erasure coding vs replication.
- Mention 11 nines durability target.

27. Common Follow-up Questions

- How would you implement cross-region replication?
- How would you handle large objects (multipart upload)?

28. Common Mistakes

- No erasure coding — expensive replication.
- Single AZ — data lost on AZ failure.

29. Optimization Ideas

- Multipart upload for large objects.
- CDN for hot objects.
- Lifecycle to cold storage (Glacier).

30. Final Summary

Object storage tests durability at scale. Erasure coding (Reed-Solomon) across many disks + facilities achieves 11 nines durability. Metadata in distributed KV store. The system is AP — high availability via multi-AZ.

QUESTION 80 ADVANCED

Design a Time-Series Database (InfluxDB)

1. Problem Statement

Design a time-series database optimized for metrics/IoT data. High write throughput, efficient range queries by time, downsampling.

2. Functional Requirements

- Write data points (timestamp, tags, value).
- Query by time range + tags.
- Aggregation (sum, avg, max, min).
- Downsampling (rollups).
- Retention policies.

3. Non-functional Requirements

- Write throughput: 1M points/sec.
- Query latency < 1s.
- Compression: > 10x.

4. Capacity Estimation

1B points/sec globally. 1 trillion points/year. Storage: 1T × 8B = 8TB/year (compressed).

5. Back-of-the-envelope Math

LSM-tree storage. Columnar compression (Gorilla for floats, delta-of-delta for timestamps).

6. API Design

```
POST /write body: "cpu,host=s1 value=85 1434055562000000000"
SELECT mean(value) FROM cpu WHERE time > now() - 1h GROUP BY time(10m)
```

7. Database Schema

```
// Per-series (tag set = series):
series_id, [(timestamp, value)] // columnar compressed
```

8. High-Level Design (HLD)

Write API → Write Ahead Log → MemTable → SSTable (LSM). Query API → Index → SSTable merge.

9. Low-Level Design (LLD)

Write: append to WAL + MemTable. Periodically flush MemTable to SSTable (sorted, compressed). Query: find relevant SSTables, merge by time, aggregate.

10. Architecture Diagram

Architecture Diagram

```
Write -> WAL -> MemTable -> flush -> SSTable (LSM)
Query -> index -> merge SSTables -> aggregate
```

11. Data Flow Diagram

Data Flow Diagram

```
Write: WAL (durable) -> MemTable (in-memory) -> flush to SSTable (disk, compressed). Query: find
series, merge SSTables by time, aggregate.
```

12. Component Explanation

- **Write API:** High-throughput ingest.
- **LSM Storage:** WAL + MemTable + SSTables.
- **Query Engine:** Range scan + aggregate.
- **Downsampler:** Periodic rollups.

13. Technology Stack

- **Language:** Go (InfluxDB) or Rust.
- **Storage:** Custom LSM-tree.
- **Compression:** Gorilla, delta-of-delta.

14. Database Choice

Custom LSM-tree (optimized for time-series).

15. Cache Strategy

MemTable (recent writes).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

None (direct write).

18. Scaling Strategy

Shard by series_id. Read replicas for queries.

19. Fault Tolerance

WAL for durability. Replication across nodes.

20. Bottlenecks

- Compaction (LSM) — background.
- High-cardinality tags — series explosion.

21. Trade-offs

- **LSM vs B-tree:** LSM is write-optimized; B-tree is read-optimized.

22. CAP Theorem Analysis

AP. Eventual consistency on reads during write bursts.

23. Security Considerations

TLS. Auth. Per-database ACLs.

24. Monitoring & Logging

Track write latency, query latency, compaction.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss LSM-tree for write throughput.
- Mention columnar compression (Gorilla, delta-of-delta).

27. Common Follow-up Questions

- How would you handle high-cardinality tags?
- How would you implement continuous queries (auto-downsample)?

28. Common Mistakes

- B-tree storage — write throughput too low.
- No compression — storage explodes.

29. Optimization Ideas

- Batch writes.
- Pre-aggregate (continuous queries).
- Compress with Gorilla.

30. Final Summary

A time-series DB tests write-optimized storage. LSM-tree (WAL + MemTable + SSTable) handles high write throughput. Columnar compression (Gorilla for floats, delta-of-delta for timestamps) achieves 10x+ compression. The architecture rewards understanding of LSM + compression algorithms.

PART VI

Expert Questions (Q81 – Q100)

Multi-region, ML-driven, and large-scale systems for Staff + Tech Lead.

QUESTION 81

EXPERT

Design Google Maps

1. Problem Statement

Design a global mapping platform. Display map tiles, search places, compute routes, real-time traffic, and handle 1B+ users daily.

2. Functional Requirements

- Render map tiles at multiple zoom levels.
- Search places by name/category.
- Route computation (driving, walking, transit).
- Real-time traffic updates.
- Turn-by-turn navigation.
- Street View (panoramic images).

3. Non-functional Requirements

- Tile load latency < 100ms.
- Route computation < 2s.
- Availability 99.99%.
- Handle 1B+ daily users.

4. Capacity Estimation

1B users, 1B tile requests/sec peak. Map data: 1PB. Routes: 1B/day. Traffic events: 100M/day.

5. Back-of-the-envelope Math

Tile pyramid (pre-rendered at all zoom levels). CDN for tile delivery. Dijkstra/A* for routing on graph. Real-time traffic via probe data (mobile phones).

6. API Design

```
GET /tiles/{z}/{x}/{y}.png -> map tile
GET /search?q=...&lat=...&lng=... -> [places]
GET /routes?from=...&to=...&mode=driving -> {steps, eta, distance}
POST /traffic/events body: {location, speed, direction}
GET /streetview?pano_id=... -> panoramic image
```

7. Database Schema

```
// Tile storage (S3 + CDN):
tiles/{z}/{x}/{y}.png // pre-rendered pyramid

// Place search (ES):
places(place_id, name, lat, lng, category, ...)

// Routing graph (custom):
nodes(node_id, lat, lng, ...)
edges(edge_id, from_node, to_node, distance, speed, road_type)

// Traffic (Cassandra):
traffic(edge_id, timestamp, speed) // time-series
```

8. High-Level Design (HLD)

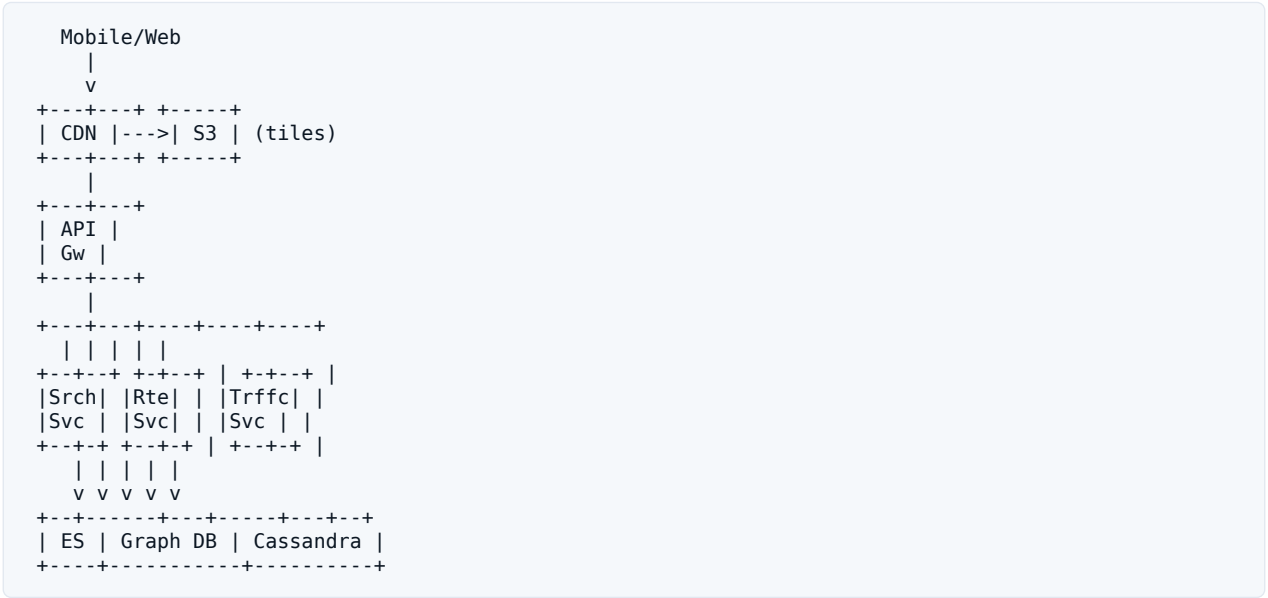
Mobile/Web → CDN (tiles) + API Gateway → Search Service (ES) + Routing Service (graph) + Traffic Service (real-time aggregation) + Navigation Service (turn-by-turn).

9. Low-Level Design (LLD)

Routing: Dijkstra/A* on road graph; heuristic = aerial distance. Real-time traffic: probe data from phones updates edge speeds; routing re-routes if faster path emerges. Tiles: pre-rendered at all zoom levels; CDN delivers.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Compute route:
  1. User enters from + to.
  2. Routing Service: geocode from/to to graph nodes.
  3. A* search on graph with current edge speeds (from Traffic Service).
  4. Return route with steps, ETA, distance.
  5. Navigation: real-time updates; re-route if traffic changes significantly.

Real-time traffic:
  1. Mobile phones report location + speed periodically.
  2. Traffic Service aggregates per edge; updates speed.
  3. Routing Service reads current speeds for accurate ETA.
```

12. Component Explanation

- **Tile Service:** Pre-rendered tiles; CDN delivery.
- **Search Service:** Elasticsearch for places.
- **Routing Service:** A* on road graph with live traffic.
- **Traffic Service:** Aggregates probe data; updates edge speeds.
- **Navigation Service:** Turn-by-turn; re-routes on traffic change.
- **Geocoding Service:** Address to lat/lng.

13. Technology Stack

- **Language:** C++ (routing perf), Java/Python (services).
- **Storage:** S3 + CDN for tiles; custom graph DB for roads; Cassandra for traffic.
- **Search:** Elasticsearch.
- **Algorithm:** A* for routing.

14. Database Choice

Custom graph DB for roads (relational doesn't handle graph traversal well). Cassandra for traffic time-series. S3 for tiles.

15. Cache Strategy

CDN for tiles (90%+ hit rate). Redis for hot routes + geocodes.

16. Load Balancer Strategy

L7 LB. Geographic routing.

17. Message Queue Usage

Kafka for traffic events + probe data.

18. Scaling Strategy

Routing: shard graph by region. Traffic: shard by edge. CDN for tiles.

19. Fault Tolerance

Multi-region. CDN failover. Graph replicas.

20. Bottlenecks

- Routing computation cost — A* with good heuristic; cache popular routes.
- Tile storage (1PB) — pre-rendered; CDN.
- Traffic event ingestion — batch + aggregate.

21. Trade-offs

- **Pre-rendered vs on-the-fly tiles:** Pre-rendered is fast but huge storage; on-the-fly is flexible but slow.
- **A* vs Dijkstra:** A* is faster (heuristic); Dijkstra is exact.
- **Real-time vs batch traffic:** Real-time is fresher; batch is cheaper.

22. CAP Theorem Analysis

AP for tiles (CDN). CP for routing (must be consistent).

23. Security Considerations

TLS. Rate limit. PII handling for probe data (anonymize).

24. Monitoring & Logging

Track tile latency, route computation time, traffic ingestion rate.

25. Deployment Strategy

Multi-region. CDN distributed globally.

26. Interview Tips

- Discuss tile pyramid + CDN.
- Mention A* with heuristic (aerial distance).
- Discuss real-time traffic via probe data.
- Mention re-routing on traffic change.

27. Common Follow-up Questions

- How would you handle offline maps (mobile)?
- How would you implement lane-level navigation?
- How would you detect road closures in real-time?
- How would you implement AR navigation?

28. Common Mistakes

- Dijkstra without heuristic — slow.
- No CDN for tiles — origin crushed.
- No real-time traffic — stale routes.
- Relational DB for graph — poor traversal.

29. Optimization Ideas

- Cache popular routes (top 1000).
- Pre-compute routes between major cities.
- Compress tiles (WebP).
- Use contraction hierarchies for fast routing.

30. Final Summary

Google Maps tests geospatial + graph algorithms at scale. Tile pyramid + CDN for map rendering. A* on road graph for routing, with real-time edge speeds from probe data. Re-routing on traffic change. The architecture rewards investment in CDN, graph databases, and real-time data aggregation.

QUESTION 82

EXPERT

Design Airbnb (Full)

1. Problem Statement

Design Airbnb at full scale: listings, search, bookings, payments, messaging, reviews, dynamic pricing, and ML recommendations.

2. Functional Requirements

- Listings with photos, amenities, calendar.
- Search by location, dates, guests, filters.
- Booking with payment; host accept/decline.
- Messaging between host + guest.
- Reviews + ratings.
- Dynamic pricing (smart pricing).
- ML recommendations ("Similar listings").

3. Non-functional Requirements

- Availability 99.95%.
- Search latency < 500ms.
- Booking atomicity (no double-book).
- Payment idempotency.

4. Capacity Estimation

150M users, 5M listings, 1M bookings/day. Storage: listings 5TB, photos 100TB, reviews 1TB.

5. Back-of-the-envelope Math

Date-scoped inventory (per-night). Hybrid search (ES + Redis GEO). ML for pricing + recs. Saga for booking + payment.

6. API Design

```
POST /listings body: {photos, amenities, price}
GET /search?location=...&checkin=...&checkout=...&filters=...
POST /bookings body: {listing_id, dates, payment}
POST /listings/{id}/reviews body: {rating, text}
GET /listings/{id}/similar -> [recommended listings]
```

7. Database Schema

```
Table: listings (listing_id, host_id, lat, lng, price, amenities, ...)
Table: listing_inventory (listing_id, date, available, price) -- per-date
Table: bookings (booking_id, listing_id, guest_id, checkin, checkout, status, total)
Table: reviews (review_id, listing_id, author_id, rating, text, created_at)
Table: messages (msg_id, thread_id, sender_id, text, created_at)
```

8. High-Level Design (HLD)

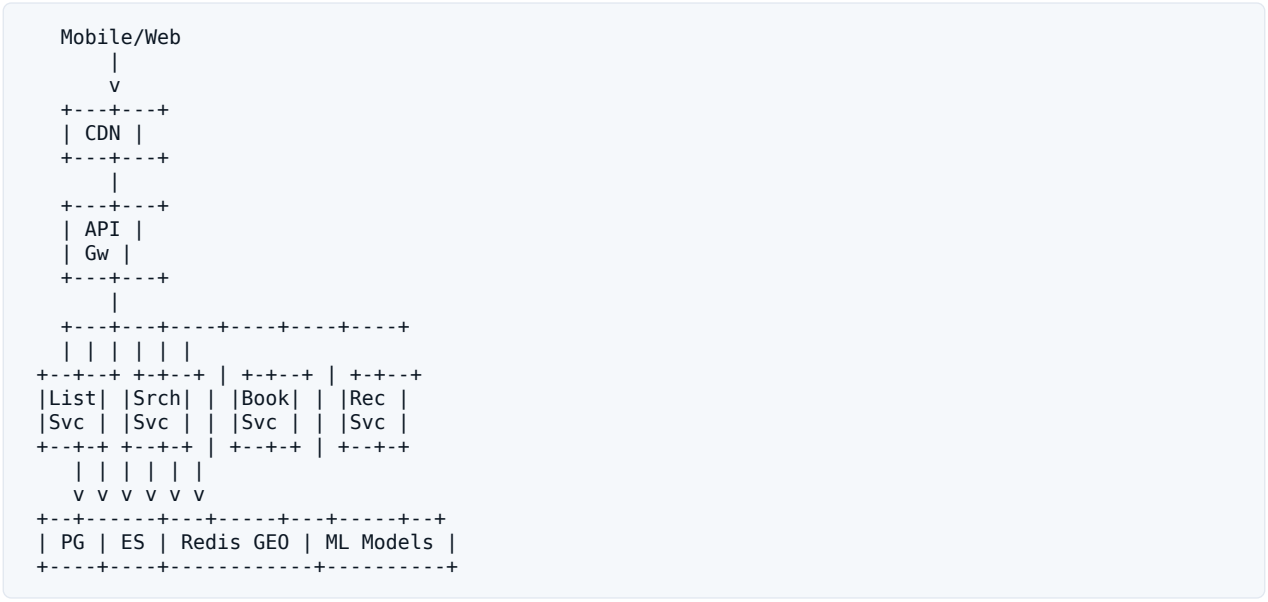
Mobile/Web → CDN → API Gateway → Listing Service + Search Service (ES + Redis GEO) + Booking Service (saga) + Payment + Messaging + Recommendation Service (ML) + Pricing Service (ML).

9. Low-Level Design (LLD)

Same as Airbnb Lite but with ML pricing (predicts optimal price per night based on demand, season, comparable listings) and ML recommendations (similar listings via embedding similarity).

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Search + book:
  1. User searches; ES for keyword, Redis GEO for location.
  2. Filter by availability (per-date inventory).
  3. Rank by relevance + ML recommendations ("similar to viewed").
  4. User books; saga: reserve inventory + charge payment + create booking.
  5. Host accepts/declines.
  6. On completion: enable messaging + review prompts.
```

12. Component Explanation

- **Listing Service:** CRUD + photos.
- **Search Service:** ES + Redis GEO hybrid.
- **Booking Service:** Saga with inventory + payment.
- **Pricing Service:** ML for dynamic pricing.

- **Recommendation Service:** ML for similar listings.
- **Messaging Service:** In-app chat.

13. Technology Stack

- **Language:** Ruby (Airbnb original) or Java.
- **DB:** PostgreSQL sharded by region.
- **Search:** Elasticsearch + Redis GEO.
- **ML:** PyTorch for pricing + recommendations.
- **Storage:** S3 + CDN.

14. Database Choice

Postgres for ACID bookings. ES + Redis GEO for search. S3 for photos.

15. Cache Strategy

Redis for hot listings, search results, ML predictions.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for booking events, ML training.

18. Scaling Strategy

Shard by region. ML models trained offline, served via inference service.

19. Fault Tolerance

Multi-AZ. Saga compensations.

20. Bottlenecks

- ML inference latency — cache predictions.
- Search with many filters — pre-compute common filters.

21. Trade-offs

- **ML vs rule-based pricing:** ML is adaptive; rules are interpretable.
- **Sync vs async recommendations:** Sync is real-time; async is faster.

22. CAP Theorem Analysis

CP for bookings. AP for search.

23. Security Considerations

TLS. PII encryption. PCI compliance. Anti-fraud.

24. Monitoring & Logging

Track search latency, booking conversion, ML model performance.

25. Deployment Strategy

Blue-green.

26. Interview Tips

- Discuss ML for pricing + recommendations.
- Mention saga for booking + payment.
- Hybrid search (ES + Redis GEO).

27. Common Follow-up Questions

- How would you implement "experiences" (Airbnb activities)?
- How would you detect fake listings?
- How would you implement instant book (no host approval)?

28. Common Mistakes

- Rule-based pricing only — not adaptive.
- No ML recommendations — missed engagement.
- No saga — inconsistent booking state.

29. Optimization Ideas

- Pre-compute ML predictions per listing nightly.
- Cache recommendation results.
- Batch ML inference.

30. Final Summary

Airbnb full extends Airbnb Lite with ML for pricing + recommendations. The core architecture: hybrid search (ES + Redis GEO), saga booking (inventory + payment), ML pricing (predicts optimal nightly price), ML recommendations (similar listings via embeddings). The architecture rewards investment in ML pipelines.

QUESTION 83

EXPERT

Design TikTok

1. Problem Statement

Design a short-form video platform. Infinite scroll feed of personalized videos, real-time engagement, video creation with effects, and global scale.

2. Functional Requirements

- Infinite scroll feed of short videos.
- Like, comment, share, duet.
- Video upload with effects + music.
- Personalized feed (ML-driven).
- Live streaming.
- Discover (trending, hashtags).

3. Non-functional Requirements

- Feed latency < 200ms.
- Availability 99.99%.
- Video startup < 1s.
- 1B+ users globally.

4. Capacity Estimation

1B users, 100M videos uploaded/day. Avg video 30MB. Storage: 100M × 30MB × 365 × 5 = 5PB. Feed reads: 10B/day.

5. Back-of-the-envelope Math

ML ranking (deep learning on user + video features). CDN for video delivery. Push-pull hybrid feed (pre-compute candidates, real-time rank).

6. API Design

```
GET /feed?cursor=... -> [ranked videos]
POST /videos body: {video_url, music_id, effects}
POST /videos/{id}/like
POST /videos/{id}/comments body: {text}
GET /trending
```

7. Database Schema

```
Table: videos (video_id, creator_id, music_id, effects, created_at, ...)
Table: likes (video_id, user_id, created_at)
Table: comments (comment_id, video_id, user_id, text, created_at)
Table: feed_candidates (user_id, video_id, score, cached_at) -- pre-computedTable:
user_embeddings (user_id, embedding) -- for ML ranking
```

8. High-Level Design (HLD)

Mobile → CDN (video) + API Gateway → Feed Service (ML rank) + Upload Service + Engagement Service. ML pipeline for ranking.

9. Low-Level Design (LLD)

Feed: candidate generation (broad set of 1000 videos from trending + followed + ML recommended). Real-time rank via deep learning model. CDN for video chunks.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Feed read:
  1. User opens app; GET /feed.
  2. Feed Service: fetch candidate set (1000 videos) from Redis.
  3. Real-time ML rank (deep learning on user + video features).
  4. Return top 10 videos with CDN URLs.
  5. App pre-fetches next 10 in background.
  6. Engagement events (watch time, like, skip) → Kafka → ML training.
```

12. Component Explanation

- **Feed Service:** Candidate generation + ML rank.
- **Upload Service:** Resumable upload to S3 + transcode pipeline.

- **Engagement Service:** Likes, comments, shares.
- **ML Ranking Service:** Deep learning inference.
- **CDN:** Video delivery.
- **Kafka + Spark:** ML training pipeline.

13. Technology Stack

- **Language:** Python (ML), Go (services).
- **DB:** Cassandra (videos, engagement).
- **Cache:** Redis (candidates, embeddings).
- **ML:** PyTorch + custom inference (C++).
- **Storage:** S3 + CDN.

14. Database Choice

Cassandra for video metadata + engagement (write-heavy). Redis for candidate sets + embeddings.

15. Cache Strategy

Redis for candidate sets, user embeddings, video metadata.

16. Load Balancer Strategy

L7 LB. Geographic routing.

17. Message Queue Usage

Kafka for engagement events → Spark → ML training.

18. Scaling Strategy

Feed Service scales by read QPS. ML inference scales by GPU. CDN scales video.

19. Fault Tolerance

Multi-region. CDN failover. ML model fallback (rule-based).

20. Bottlenecks

- ML inference latency — batch + GPU.
- Egress bandwidth (5PB) — CDN.
- ML training cost — daily batch + hourly incremental.

21. Trade-offs

- **Pre-compute vs real-time rank:** Pre-compute is fast; real-time is fresher.
- **Push vs pull feed:** Pull (compute on read) for personalization.
- **Batch vs streaming ML:** Batch is cheaper; streaming is fresher.

22. CAP Theorem Analysis

AP for feed. CP for engagement (no double-like).

23. Security Considerations

TLS. Content moderation (NSFW, hate speech). Copyright detection for music. Anti-bot for engagement.

24. Monitoring & Logging

Track feed latency, video startup, ML inference latency, engagement rate.

25. Deployment Strategy

Multi-region. GPU clusters for ML.

26. Interview Tips

- Emphasize ML ranking — core differentiator.
- Discuss candidate generation + real-time rank pattern.
- Mention CDN for video delivery.
- Engagement events → ML training pipeline.

27. Common Follow-up Questions

- How would you implement video effects (AR filters)?
- How would you handle live streaming?
- How would you detect trending sounds?
- How would you implement duets (split-screen)?

28. Common Mistakes

- No ML ranking — poor engagement.
- No CDN — egress kills origin.
- Synchronous engagement writes — slow feed.

29. Optimization Ideas

- Pre-fetch next 10 videos.
- Batch ML inference across users.
- Compress videos aggressively (AV1).
- Edge-cache popular videos.

30. Final Summary

TikTok tests ML-driven personalization at extreme scale. The core architecture: candidate generation (1000 videos in Redis) → real-time ML rank (deep learning) → CDN delivery. Engagement events flow to Kafka → Spark → ML training. The feed is pull-based (compute on read) for maximum personalization. The architecture rewards deep investment in ML pipeline + CDN.

QUESTION 84 EXPERT

Design a Snowflake ID Generator

1. Problem Statement

Design a distributed unique ID generator producing 64-bit IDs that are sortable by time, geographically distributed, and survive node failures.

2. Functional Requirements

- Generate 64-bit unique IDs.
- Roughly time-ordered (sortable).
- Distributed (no single coordinator).
- Survive node failures (no ID duplication).
- High throughput (1M IDs/sec per node).

3. Non-functional Requirements

- Latency < 1ms per ID.
- Availability 99.99%.
- No duplicates ever.
- Roughly time-ordered.

4. Capacity Estimation

1M IDs/sec per node. 10K nodes globally = 10B IDs/sec.

5. Back-of-the-envelope Math

64-bit ID = 41 bits timestamp (ms, ~69 years) + 10 bits worker ID (1024 workers) + 12 bits sequence (4096 IDs/ms per worker).

6. API Design

```
// Internal API (called by services):
long generateId() -> 64-bit ID

// Or via HTTP (less common):
GET /id -> {"id": 1234567890123456}
```

7. Database Schema

```
// Snowflake bit layout:
| 1 bit (unused) | 41 bits timestamp | 10 bits worker_id | 12 bits sequence |

// Worker registration:
Table: workers (worker_id, hostname, last_heartbeat, datacenter_id)
```

8. High-Level Design (HLD)

Each worker node generates IDs independently. Worker ID assigned via coordination service (ZooKeeper/etcd). Clock synced via NTP.

9. Low-Level Design (LLD)

On ID request: read current timestamp (ms). If same as last, increment sequence. If different, reset sequence to 0. Combine: timestamp | worker_id | sequence. If sequence overflows (4096 in same ms), wait until next ms.

10. Architecture Diagram

Architecture Diagram

```
Service A ---------> [Worker 1] (independent ID gen)
Service B ---------> [Worker 2]
Service C ---------> [Worker 3]
                  ^
                  |
         +--------+--------+
         | Coordinator | (assigns worker IDs)
         | (ZooKeeper/etcd)|
         +-----------------+
Each worker:
  timestamp (41 bits) + worker_id (10 bits) + sequence (12 bits)
```

11. Data Flow Diagram

Data Flow Diagram

```
Generate ID:
  1. Service calls local Snowflake library (no network call).
  2. Library reads current timestamp (ms).
  3. If same ms as last: increment sequence.
  4. If different ms: reset sequence to 0.
  5. Combine: (timestamp << 22) | (worker_id << 12) | sequence.
  6. Return 64-bit ID.

Worker registration:
  1. On startup, worker requests unique worker_id from ZooKeeper.
  2. ZooKeeper assigns sequential ID (0-1023).
  3. Worker uses this ID for all generated Snowflake IDs.
  4. Heartbeat every 30s; if dies, ID can be reassigned.
```

12. Component Explanation

- **Snowflake Library:** Per-service; generates IDs locally.
- **Coordinator:** ZooKeeper/etcd for worker ID assignment.
- **NTP:** Clock sync across nodes.

13. Technology Stack

- **Language:** Any (library available in all languages).
- **Coordination:** ZooKeeper or etcd.
- **Clock Sync:** NTP.

14. Database Choice

No DB. Worker IDs in ZooKeeper/etcd.

15. Cache Strategy

In-memory (library is local).

16. Load Balancer Strategy

No LB (client-side library).

17. Message Queue Usage

None.

18. Scaling Strategy

Add workers (up to 1024 per Snowflake deployment).

19. Fault Tolerance

Each worker independent. Coordinator failure: workers continue with cached IDs.

20. Bottlenecks

- Clock skew — NTP + reject future timestamps.
- Sequence overflow (4096 IDs/ms per worker) — wait for next ms.
- Worker ID exhaustion (1024) — use datacenter bits.

21. Trade-offs

- **Snowflake vs UUID:** Snowflake is sortable + compact (64-bit); UUID is random + 128-bit.
- **Coordinator vs coordinator-less:** Coordinator assigns worker IDs; coordinator-less uses random + retry.
- **Strict ordering vs availability:** Strict needs sync clock; loose is available.

22. CAP Theorem Analysis

AP. Worker generates IDs even if coordinator down. Risk: clock skew may cause slight disorder.

23. Security Considerations

TLS to coordinator. Worker ID is not sensitive.

24. Monitoring & Logging

Track ID generation rate, clock skew, sequence overflows.

25. Deployment Strategy

Workers co-located with services. Coordinator multi-AZ.

26. Interview Tips

- Discuss bit layout (timestamp + worker + sequence).
- Mention worker ID assignment via ZooKeeper.
- Discuss clock skew handling (NTP, reject future).
- No network call per ID (library is local).

27. Common Follow-up Questions

- How would you handle clock skew?
- How would you scale beyond 1024 workers?
- How would you make IDs strictly sortable?
- How would you implement coordinator-less ID gen?

28. Common Mistakes

- Network call per ID — slow.
- No clock skew handling — duplicate IDs.
- UUID — not sortable, 128-bit.

29. Optimization Ideas

- Pre-generate batch of IDs (reduce lock contention).
- Use higher-resolution timestamp (microseconds).
- Embed datacenter ID in worker bits.

30. Final Summary

Snowflake tests distributed unique ID generation. The 64-bit layout (timestamp + worker + sequence) enables sortability + uniqueness without coordination per ID. Worker IDs assigned via ZooKeeper at startup. Each worker generates IDs locally (no network call). Clock skew handled via NTP + rejecting future timestamps. The architecture rewards understanding of bit manipulation + distributed coordination.

QUESTION 85 EXPERT

Design a Distributed Rate Limiter (Production-Grade)

1. Problem Statement

Design a production-grade distributed rate limiter with multiple algorithms (token bucket, sliding window, leaky bucket), per-user and per-API limits, and sub-millisecond latency.

2. Functional Requirements

- Multiple algorithms: token bucket, sliding window, leaky bucket.
- Per-user, per-API, per-IP limits.
- Hierarchical limits (user + API + global).
- Configurable via admin API.
- Return 429 with Retry-After + remaining quota.

3. Non-functional Requirements

- Latency overhead < 1ms.
- Availability 99.99%.
- Accuracy > 99.9%.
- Throughput: 10M req/sec.

4. Capacity Estimation

10M req/sec globally. 1B distinct rate limit keys.

5. Back-of-the-envelope Math

Redis Lua scripts for atomic operations. Local cache + periodic sync for hot keys. Sliding window log for accuracy; token bucket for bursts.

6. API Design

```
// Internal API (called by gateway):
boolean allowRequest(key, algorithm, limit, window) -> {allowed, remaining, retry_after}

// Admin API:
POST /limits body: {key_pattern, algorithm, limit, window}
```

7. Database Schema

```
// Redis keys (Lua-managed):
ratelimit:{key}:tokens // token bucket count
ratelimit:{key}:last_refill // last refill timestamp
ratelimit:{key}:window // sliding window sorted set of timestamps
ratelimit:{key}:leak_level // leaky bucket level
```

8. High-Level Design (HLD)

Gateway → Rate Limiter Service → Redis Cluster (Lua scripts). Local cache for hot keys.

9. Low-Level Design (LLD)

Token bucket: Lua script (atomic: read tokens, refill by elapsed time, decrement or reject). Sliding window: Lua script (remove old timestamps, count, add new). Leaky bucket: Lua script (compute leak, add or reject).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Check + limit:
  1. Gateway calls allowRequest(key, algorithm, limit, window).
  2. Rate Limiter: check local cache for hot key.
  3. If miss: call Redis Lua script (atomic check-and-increment).
  4. Return {allowed, remaining, retry_after}.
  5. Gateway: proceed or return 429.
```

12. Component Explanation

- **Rate Limiter Service:** Stateless; calls Redis.
- **Redis Cluster:** Shared state; Lua scripts.
- **Local Cache:** Hot keys; reduces Redis load.
- **Config Service:** Per-key limit rules.

13. Technology Stack

- **Language:** Go/Rust (low latency).
- **Cache:** Redis Cluster.
- **Algorithm:** Token bucket, sliding window, leaky bucket (Lua scripts).

14. Database Choice

Redis only (ephemeral state).

15. Cache Strategy

Local cache (in-process) for hot keys + Redis for shared state.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Optional Kafka for limit events (analytics).

18. Scaling Strategy

Shard Redis by key hash. Local cache absorbs hot keys.

19. Fault Tolerance

Redis replication + failover. Local cache degrades gracefully.

20. Bottlenecks

- Redis lookup per request — local cache for hot keys.
- Lua script CPU — optimize scripts.

21. Trade-offs

- **Token bucket vs sliding window:** Token allows bursts; sliding is precise.
- **Local vs distributed:** Local is fast but not accurate; distributed is accurate but slow.
- **Sync vs async config:** Sync is consistent; async is fast.

22. CAP Theorem Analysis

AP. Slight over-limit acceptable during failures.

23. Security Considerations

TLS. Internal-only API.

24. Monitoring & Logging

Track reject rate, Redis latency, cache hit ratio.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss Lua scripts for atomicity.
- Mention local cache for hot keys.
- Discuss algorithm trade-offs (token vs sliding vs leaky).

27. Common Follow-up Questions

- How would you handle Redis failure gracefully?
- How would you implement global rate limits (across all gateways)?
- How would you do per-IP rate limiting with NAT (many users behind one IP)?

28. Common Mistakes

- Non-atomic check + increment — race condition.
- DB for state — too slow.
- No local cache — Redis overwhelmed.

29. Optimization Ideas

- Local token bucket + periodic Redis sync.
- Batch checks (pipeline).
- Compress keys.

30. Final Summary

A production-grade distributed rate limiter tests atomic operations + caching. Redis Lua scripts provide atomicity (check + increment in one operation). Local cache for hot keys reduces Redis load. Multiple algorithms (token bucket, sliding window, leaky bucket) for different use cases. The system is AP — slight over-limit acceptable during failures.

QUESTION 86 EXPERT

Design a Distributed Lock Service (ZooKeeper/etcd)

1. Problem Statement

Design a distributed lock service. Multiple clients acquire exclusive locks on named resources; locks have TTL; survive node failures.

2. Functional Requirements

- Acquire lock on named resource.
- Release lock.
- TTL (auto-release if holder dies).
- FIFO ordering (fair locks).
- Re-entrant locks (same holder can re-acquire).

3. Non-functional Requirements

- Latency < 10ms per acquire/release.
- Availability 99.99%.
- No two clients hold same lock simultaneously.
- Survives node failure.

4. Capacity Estimation

1M locks. 100K acquires/sec.

5. Back-of-the-envelope Math

ZooKeeper-style: ephemeral sequential nodes. Lock = create node with lowest sequence number. Watch previous node for release.

6. API Design

```
POST /locks/{resource} body: {holder_id, ttl} -> {lock_id, status: ACQUIRED|QUEUED}
DELETE /locks/{lock_id}
GET /locks/{resource} -> {current_holder}
```

7. Database Schema

```
// etcd/ZK-style storage:
/locks/{resource}/
  {seq_001} = holder_id_1 // ephemeral
  {seq_002} = holder_id_2 // ephemeral
  ...

// Lock holder = lowest sequence number
```

8. High-Level Design (HLD)

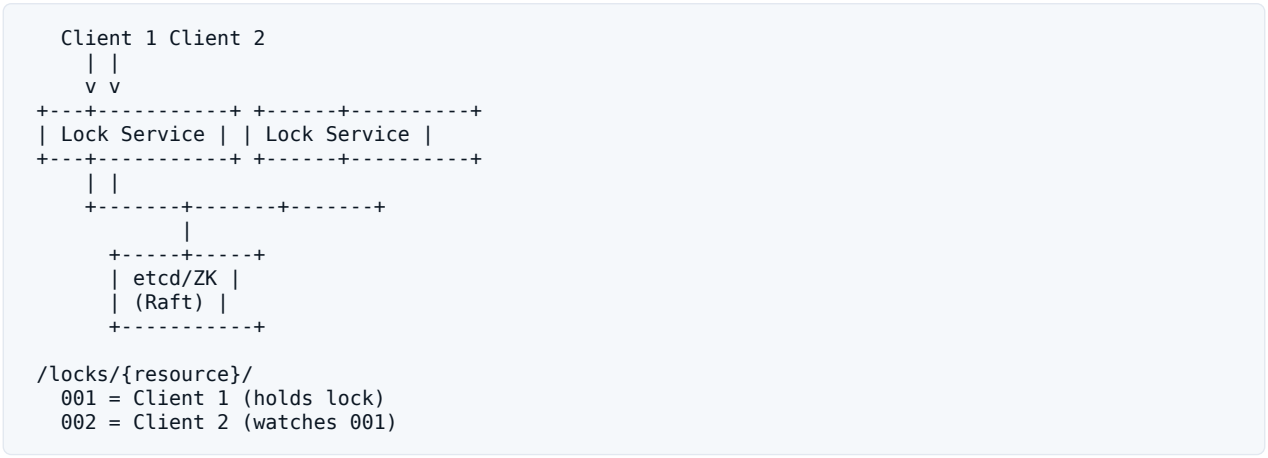
Client → Lock Service → etcd/ZK cluster (Raft consensus). Locks = ephemeral sequential nodes.

9. Low-Level Design (LLD)

Acquire: create ephemeral sequential node under /locks/{resource}/. Get all children; if your sequence is lowest, you hold lock. Else, watch previous node. On watch event (previous released or died), check again.

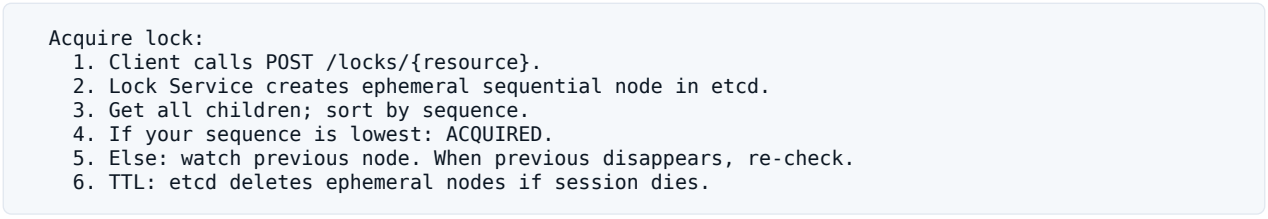
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Lock Service:** Stateless; wraps etcd/ZK.
- **etcd/ZK Cluster:** Raft consensus; ephemeral nodes.
- **Watch Manager:** Notifies clients of node changes.

13. Technology Stack

- **Language:** Go (etcd) or Java (ZooKeeper).
- **Algorithm:** Raft (etcd) or ZAB (ZooKeeper).

14. Database Choice

etcd or ZooKeeper (CP via Raft/ZAB).

15. Cache Strategy

None (consistency required).

16. Load Balancer Strategy

L7 LB to etcd cluster.

17. Message Queue Usage

None.

18. Scaling Strategy

etcd cluster of 3-7 nodes. Read replicas for scale.

19. Fault Tolerance

Raft consensus. Ephemeral nodes auto-deleted on session death.

20. Bottlenecks

- etcd write throughput — shard by resource prefix.

21. Trade-offs

- **Fair vs unfair:** Fair (FIFO) prevents starvation; unfair is faster.
- **etcd vs Redis Redlock:** etcd is CP; Redlock is AP (debated).

22. CAP Theorem Analysis

CP. Strong consistency via Raft.

23. Security Considerations

TLS. Auth. ACLs.

24. Monitoring & Logging

Track acquire latency, lock hold time, queue length.

25. Deployment Strategy

Multi-AZ etcd cluster.

26. Interview Tips

- Discuss ephemeral sequential nodes.
- Mention watch mechanism for fair locks.

- Discuss Raft/ZAB for consensus.

27. Common Follow-up Questions

- How would you handle client crashes mid-operation (lock held)?
- How would you implement re-entrant locks?
- How would you handle network partition (split-brain)?
- How would you implement read-write locks?

28. Common Mistakes

- No TTL — locks held forever on crash.
- Polling for release — high latency.
- Single coordinator — SPOF.

29. Optimization Ideas

- Batch lock acquisitions.
- Use watch (push) not poll.
- Pre-acquire locks for known hot resources.

30. Final Summary

A distributed lock service tests consensus + ephemeral state. etcd/ZooKeeper-style: ephemeral sequential nodes; lock = lowest sequence; watch previous for release. TTL via session death (ephemeral nodes auto-deleted). The system is CP — strong consistency via Raft/ZAB. The architecture rewards understanding of consensus protocols and watch patterns.

QUESTION 87 EXPERT

Design a Recommendation Engine

1. Problem Statement

Design a recommendation engine for an e-commerce platform. Generate personalized product recommendations per user based on browse/buy history + collaborative + content-based signals.

2. Functional Requirements

- Generate "Recommended for you" feed per user.
- Similar items ("Customers also bought").
- Trending + seasonal recommendations.
- Real-time updates based on recent activity.
- A/B test support (multiple model variants).

3. Non-functional Requirements

- Recommendation latency < 100ms.
- Refresh hourly + real-time bias.
- Availability 99.9%.
- Handle 100M users, 10M products.

4. Capacity Estimation

100M users, 10M products. User-item interaction matrix: 100M × 10M = 1T entries (sparse, ~1B non-zero). Embeddings: 100M × 100 dims = 40GB.

5. Back-of-the-envelope Math

Hybrid: collaborative filtering (matrix factorization) + content-based (product embeddings) + real-time signals. Pre-compute top-100 per user hourly; real-time bias.

6. API Design

```
GET /users/{id}/recommendations?context=... -> [product_ids with scores]
GET /products/{id}/similar -> [similar product_ids]
POST /events body: {user_id, product_id, event: VIEW|CART|BUY}
```

7. Database Schema

```
Table: user_embeddings (user_id, embedding_vector) // 100-dim float
Table: product_embeddings (product_id, embedding_vector)
Table: recommendations (user_id, product_id, score, model_version, refreshed_at)
Table: events (user_id, product_id, event_type, timestamp) // for training
```

8. High-Level Design (HLD)

Mobile/Web → API Gateway → Recommendation Service (vector search + real-time bias) → Vector DB + Redis cache. ML pipeline: events → Kafka → Spark training → model deploy.

9. Low-Level Design (LLD)

Hourly batch: compute top-100 recommendations per user via matrix factorization or deep learning. Store in Redis. Real-time: bias top recommendations by recent user activity (boost products in same category as last viewed).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Generate recommendations:
  1. GET /users/{id}/recommendations.
  2. Rec Service: fetch pre-computed top-100 from Redis.
  3. Apply real-time bias (boost products in last-viewed category).
  4. Filter seen + out-of-stock.
  5. Return top 10.

ML pipeline:
  1. Events (view, cart, buy) stream to Kafka.
  2. Spark job hourly: train matrix factorization + deep learning.
  3. Generate embeddings per user + product.
  4. Compute top-100 per user via vector similarity.
  5. Deploy model + cache results in Redis.
```

12. Component Explanation

- **Recommendation Service:** Real-time inference + bias.
- **Vector DB:** Faiss/Milvus for similarity search.
- **ML Pipeline:** Spark + PyTorch for training.
- **Redis:** Pre-computed recommendations cache.
- **Kafka:** Event stream for training.

13. Technology Stack

- **Language:** Python (ML), Go/Java (services).
- **ML:** PyTorch + TensorFlow.
- **Vector DB:** Faiss/Milvus.
- **Cache:** Redis.
- **Queue:** Kafka + Spark.

14. Database Choice

Vector DB for embeddings. Redis for cached recommendations. Cassandra for events.

15. Cache Strategy

Redis for pre-computed top-100 per user.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for events. Spark for batch training.

18. Scaling Strategy

Vector DB scales with embedding count. ML training scales with Spark cluster.

19. Fault Tolerance

Multi-AZ. ML model fallback (rule-based if model down).

20. Bottlenecks

- ML training cost — daily batch + hourly incremental.
- Vector similarity search — approximate nearest neighbor (ANN).
- Real-time bias latency — cache recent activity per user.

21. Trade-offs

- **Collaborative vs content-based:** CF uses behavior; content uses product features. Hybrid is best.
- **Batch vs real-time:** Batch is cheaper; real-time is fresher.
- **Matrix factorization vs deep learning:** MF is interpretable; DL is more accurate.

22. CAP Theorem Analysis

AP. Recommendations eventually consistent.

23. Security Considerations

TLS. PII handling for user behavior. Anonymize events.

24. Monitoring & Logging

Track recommendation CTR, ML model performance, training latency.

25. Deployment Strategy

Multi-region. GPU clusters for ML.

26. Interview Tips

- Discuss hybrid approach (CF + content + real-time).
- Mention vector DB for similarity search.
- Discuss batch + real-time pipeline.
- A/B test support for model variants.

27. Common Follow-up Questions

- How would you implement "frequently bought together"?
- How would you handle cold-start (new user, new product)?
- How would you implement explainable recommendations ("why this?")?
- How would you detect and remove filter bubbles?

28. Common Mistakes

- Real-time ML inference per request — too slow.
- No real-time bias — stale recommendations.
- Single model — no A/B testing.

29. Optimization Ideas

- Pre-compute top-100 per user hourly.
- Use ANN (HNSW) for fast vector search.
- Cache recent activity per user for real-time bias.
- Batch ML inference.

30. Final Summary

A recommendation engine tests ML pipeline architecture. Hybrid approach: collaborative filtering (matrix factorization) + content-based (embeddings) + real-time bias. Pre-compute top-100 per user hourly; real-time bias adjusts for recent activity. Vector DB (Faiss/Milvus) for similarity search. The architecture rewards investment in ML training pipeline + vector search infrastructure.

QUESTION 88 EXPERT

Design Search Autocomplete

1. Problem Statement

Design a search autocomplete system. As user types, return top suggestions in < 50ms. Handle typos, personalization, and trending queries.

2. Functional Requirements

- Return top N suggestions as user types.
- Typo tolerance (fuzzy matching).
- Personalization (based on user history).
- Trending queries boost.
- Multi-language support.

3. Non-functional Requirements

- Latency < 50ms.
- Availability 99.95%.
- Handle 100K QPS.
- Freshness: trending updated every 5 min.

4. Capacity Estimation

100K QPS. Suggestion index: 100M queries (top queries).

5. Back-of-the-envelope Math

Trie (prefix tree) for prefix matching. Weighted by query frequency + recency. N-gram model for typo tolerance.

6. API Design

```
GET /autocomplete?q=...&user_id=...&location=... -> [suggestions]
```

7. Database Schema

```
// Trie structure (in-memory):
TrieNode {
  children: {char: TrieNode}
  top_queries: [(query, weight)] // top 10 by frequency
}

// Query frequency DB:
Table: query_counts (query, count, last_seen, location)
```

8. High-Level Design (HLD)

Mobile/Web → API Gateway → Autocomplete Service → Trie (in-memory) + Trending Service (real-time) + Personalization Service.

9. Low-Level Design (LLD)

Build trie from top 100M queries. Per node, store top 10 queries by weight. On request: traverse trie by prefix; return top queries at node. Apply personalization (boost queries user has used before) + trending (boost recent queries).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Autocomplete:
  1. User types "iph". GET /autocomplete?q=iph&user_id=...
  2. Autocomplete Service: traverse trie to "iph" node.
  3. Get top 10 queries at node (e.g. "iphone", "iphone 15", "iphone case").
  4. Apply trending boost (queries trending in last hour).
  5. Apply personalization (boost queries user has used).
  6. Return ranked suggestions.

Trie update:
  1. Query counts stream to Kafka.
  2. Aggregator: update query counts every 5 min.
  3. Rebuild trie hourly (or incrementally update).
```

12. Component Explanation

- **Autocomplete Service:** Trie traversal + ranking.
- **Trie:** In-memory prefix tree with top queries per node.
- **Trending Service:** Real-time query count aggregation.
- **Personalization Service:** User query history.

13. Technology Stack

- **Language:** C++/Rust (low latency).
- **Data Structure:** Trie + weighted suggestions.
- **Cache:** Redis (trending, user history).
- **Queue:** Kafka (query events).

14. Database Choice

In-memory trie (no DB on critical path). Cassandra for query counts.

15. Cache Strategy

Redis for trending queries, user history. In-process cache for hot prefixes.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for query events → aggregator → trie update.

18. Scaling Strategy

Shard trie by first character. Replicate for read scale.

19. Fault Tolerance

Multi-AZ. Trie rebuilt from DB on failure.

20. Bottlenecks

- Trie memory size — shard + compress.
- Trie update frequency — hourly + incremental.
- Personalization latency — cache user history.

21. Trade-offs

- **Trie vs n-gram:** Trie is fast for prefix; n-gram handles typos.
- **Pre-compute vs on-the-fly:** Pre-compute top per node; on-the-fly for personalization.

22. CAP Theorem Analysis

AP. Suggestions eventually consistent.

23. Security Considerations

TLS. PII scrubbing (don't store sensitive queries).

24. Monitoring & Logging

Track autocomplete latency, suggestion CTR, trending update lag.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss trie data structure.
- Mention weighted suggestions per node.
- Discuss trending + personalization boosts.
- Discuss typo tolerance (n-gram or edit distance).

27. Common Follow-up Questions

- How would you handle typo tolerance?
- How would you implement multi-language autocomplete?
- How would you handle offensive query filtering?
- How would you scale to 1B queries?

28. Common Mistakes

- DB lookup per request — too slow.
- No trending boost — stale suggestions.
- No personalization — generic for all users.

29. Optimization Ideas

- Pre-compute top 10 per trie node.
- Cache user history in Redis.
- Compress trie (radix tree).
- Use edit distance for typo tolerance.

30. Final Summary

Search autocomplete tests low-latency data structures. Trie (prefix tree) enables fast prefix matching. Per node, store top 10 weighted queries. Apply trending + personalization boosts on top of trie results. The architecture rewards understanding of tries + real-time aggregation.

QUESTION 89 EXPERT

Design a Fraud Detection System

1. Problem Statement

Design a real-time fraud detection system. Score transactions in < 100ms; block high-risk; adapt to new fraud patterns via ML.

2. Functional Requirements

- Score each transaction in real-time (< 100ms).
- Block high-risk transactions.
- Adapt to new fraud patterns (online learning).
- Case management for human review.
- Rules + ML hybrid.

3. Non-functional Requirements

- Latency < 100ms per transaction.
- Availability 99.99%.
- Precision > 99% (avoid false positives).
- Recall > 90% (catch most fraud).

4. Capacity Estimation

10M transactions/day. 100K transactions/sec peak.

5. Back-of-the-envelope Math

Hybrid: rules engine (fast, deterministic) + ML model (slower, adaptive). Real-time features (user behavior in last 1h) + batch features (historical).

6. API Design

```
POST /score body: {txn_id, user_id, amount, merchant, ...} -> {score, decision:
APPROVE|REVIEW|BLOCK, reasons}
POST /feedback body: {txn_id, was_fraud} // for online learning
```

7. Database Schema

```
Table: transactions (txn_id, user_id, amount, merchant, score, decision, timestamp)
Table: user_features (user_id, txn_count_1h, txn_count_24h, avg_amount, ...)
Table: merchant_features (merchant_id, fraud_rate, ...)
Table: model_versions (version, training_date, metrics)
```

8. High-Level Design (HLD)

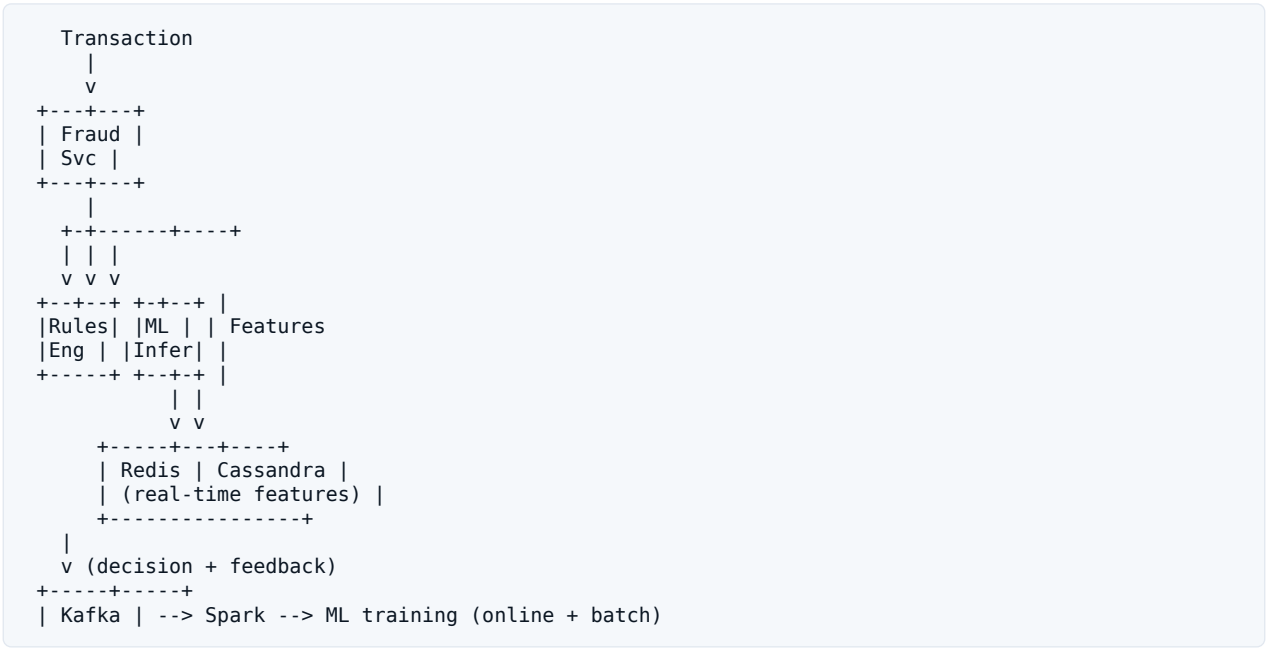
Transaction → Fraud Service (rules + ML) → Decision. Features from real-time store (Redis) + batch store (Cassandra).

9. Low-Level Design (LLD)

On transaction: fetch user features (last 1h txn count, avg amount). Fetch merchant features. Apply rules (block if amount > 10x user avg). Apply ML model (gradient boosted trees). Combine scores; return decision + reasons.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Score transaction:
  1. Transaction arrives; Fraud Service fetches features.
  2. Real-time features from Redis (txn count last 1h, avg amount).
  3. Batch features from Cassandra (user historical fraud rate).
  4. Apply rules engine (deterministic, fast).
  5. Apply ML model (gradient boosted trees).
  6. Combine scores; return decision (APPROVE/REVIEW/BLOCK) + reasons.
  7. Async: feedback (was_fraud) → Kafka → online learning.
```

12. Component Explanation

- **Fraud Service:** Rules + ML inference.
- **Rules Engine:** Deterministic rules (fast).
- **ML Inference:** Gradient boosted trees (XGBoost/LightGBM).
- **Feature Store:** Redis (real-time) + Cassandra (batch).
- **Case Management:** Human review for REVIEW decisions.

13. Technology Stack

- **Language:** Java/Python.
- **ML:** XGBoost/LightGBM (tabular), PyTorch (deep).

- **Cache:** Redis (real-time features).
- **DB:** Cassandra (batch features, transactions).
- **Queue:** Kafka (feedback, training).

14. Database Choice

Redis for real-time features (sub-ms). Cassandra for historical features.

15. Cache Strategy

Redis for real-time features (updated on every txn).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for transaction events, feedback (online learning).

18. Scaling Strategy

Fraud Service scales by txn rate. ML inference scales by GPU.

19. Fault Tolerance

Multi-AZ. Rules-only fallback if ML down.

20. Bottlenecks

- Feature fetch latency — cache aggressively.
- ML inference latency — batch + GPU.
- Online learning cost — micro-batches.

21. Trade-offs

- **Rules vs ML:** Rules are interpretable; ML is adaptive. Hybrid is best.
- **Block vs review:** Block is safer; review has fewer false positives.
- **Online vs batch learning:** Online adapts fast; batch is stable.

22. CAP Theorem Analysis

CP. Decision must be deterministic per transaction.

23. Security Considerations

TLS. PII encryption. Audit log for decisions. Model versioning.

24. Monitoring & Logging

Track precision, recall, false positive rate, model drift.

25. Deployment Strategy

Multi-AZ. GPU clusters for ML.

26. Interview Tips

- Discuss hybrid: rules + ML.
- Mention feature store (real-time + batch).
- Discuss online learning for adaptation.
- Discuss precision/recall trade-off.

27. Common Follow-up Questions

- How would you handle concept drift (fraud patterns change)?
- How would you implement graph-based fraud detection (rings)?
- How would you handle false positives (user friction)?
- How would you detect synthetic identity fraud?

28. Common Mistakes

- ML only — no interpretable rules.
- No real-time features — stale scoring.
- No feedback loop — model doesn't adapt.

29. Optimization Ideas

- Cache features per user (update on every txn).
- Batch ML inference.
- Use rules for fast path; ML for borderline.

30. Final Summary

A fraud detection system tests real-time ML + rules hybrid. Rules engine provides fast deterministic decisions; ML model adapts to new patterns. Feature store: Redis (real-time) + Cassandra (batch). Online learning via feedback loop. The system is CP — deterministic per transaction. The architecture rewards investment in feature store + ML pipeline.

QUESTION 90

EXPERT

Design a Distributed Database (CockroachDB/Spanner)

1. Problem Statement

Design a globally distributed SQL database with ACID transactions, horizontal scalability, and multi-region deployment.

2. Functional Requirements

- SQL queries with ACID transactions.
- Horizontal scaling (add nodes).
- Multi-region replication.
- Survives node + region failure.
- Serializable isolation.

3. Non-functional Requirements

- Latency: < 10ms for single-region, < 100ms for cross-region.
- Availability 99.99%.
- Survives region failure.
- Linearizable writes.

4. Capacity Estimation

10PB+ data. 1M writes/sec globally.

5. Back-of-the-envelope Math

Sharded by key range (range-based partitioning). Raft consensus per range. 2PC for cross-range transactions. TrueTime (Spanner) or HLC (CockroachDB) for ordering.

6. API Design

```
// Standard SQL API:
BEGIN TRANSACTION;
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
COMMIT;
```

7. Database Schema

```
// Internal storage (LSM-tree per range):
Range {
  start_key, end_key,
  replicas: [node_ids], // Raft group
  leader: node_id,
  data: LSM-tree
}

// Metadata:
Table: ranges (range_id, start_key, end_key, replicas, leader)
```

8. High-Level Design (HLD)

Client → Gateway Node → Range Leader (Raft) → Replicas. Cross-range transactions via 2PC with timestamp ordering (TrueTime/HLC).

9. Low-Level Design (LLD)

Each range (64MB) is a Raft group with 3 replicas across regions. Writes go to leader; replicated to followers. Cross-range transactions: 2PC with commit timestamp from TrueTime (Spanner) or HLC (CockroachDB).

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Write:
  1. Client sends write; gateway routes to range leader.
  2. Leader proposes to Raft group; replicas ack.
  3. Once quorum: commit.
  4. For multi-range: 2PC coordinator with timestamp.

Read:
  1. Client sends read; gateway routes to range leader (or replica for stale read).
  2. Leader reads from local LSM-tree.
  3. Returns result.
```

12. Component Explanation

- **Gateway Node:** Routes requests to range leaders.
- **Range Leader:** Handles writes for a key range.

- **Raft Group:** 3 replicas per range; consensus.
- **Transaction Coordinator:** 2PC for cross-range transactions.
- **TrueTime/HLC:** Global clock for ordering.

13. Technology Stack

- **Language:** Go (CockroachDB) or C++ (Spanner).
- **Algorithm:** Raft + 2PC + TrueTime/HLC.
- **Storage:** LSM-tree (RocksDB).

14. Database Choice

Custom LSM-tree per range.

15. Cache Strategy

In-memory cache for hot ranges.

16. Load Balancer Strategy

Client-side routing (range lookup).

17. Message Queue Usage

None.

18. Scaling Strategy

Add nodes; rebalance ranges. Each range is independent.

19. Fault Tolerance

Raft consensus per range. Survives node + AZ failure. Region failure: replicas in other regions promote.

20. Bottlenecks

- Cross-region write latency — place leaders in write-origin region.
- Hot ranges — split + rebalance.
- 2PC overhead for multi-range transactions.

21. Trade-offs

- **Spanner (TrueTime) vs CockroachDB (HLC):** TrueTime needs atomic clocks; HLC is software-only.
- **Strong vs eventual consistency:** Strong is safer; eventual is faster.
- **Range size:** Smaller = more parallelism; larger = less metadata.

22. CAP Theorem Analysis

CP. Strong consistency via Raft + 2PC.

23. Security Considerations

TLS. Auth. Encryption at rest. Per-range ACLs.

24. Monitoring & Logging

Track write latency, replication lag, range balance.

25. Deployment Strategy

Multi-region. Replicas across regions.

26. Interview Tips

- Discuss range-based sharding + Raft per range.
- Mention 2PC for cross-range transactions.
- Discuss TrueTime (Spanner) vs HLC (CockroachDB).

27. Common Follow-up Questions

- How would you handle hot ranges (hot keys)?
- How would you implement multi-region writes (low latency globally)?
- How would you handle schema migrations?
- How would you implement savepoints (nested transactions)?

28. Common Mistakes

- Single leader globally — high write latency.
- No 2PC for cross-range — inconsistent transactions.
- No clock skew handling — ordering violations.

29. Optimization Ideas

- Place range leaders in write-origin region.
- Use stale reads for non-critical queries.
- Batch writes within a transaction.

30. Final Summary

A distributed SQL database tests consensus + 2PC at scale. Range-based sharding (64MB per range) with Raft per range. Cross-range transactions via 2PC with TrueTime (Spanner, atomic clocks) or HLC (CockroachDB, software-only). The system is CP — strong consistency. The architecture rewards understanding of consensus protocols + distributed transactions.

QUESTION 91

EXPERT

Design a Real-Time Analytics Platform

1. Problem Statement

Design a real-time analytics platform. Ingest events (clicks, impressions), aggregate in real-time, and serve dashboards with sub-second latency.

2. Functional Requirements

- Ingest events at high throughput (1M/sec).
- Real-time aggregation (count, sum, avg).
- Dashboards with sub-second queries.
- Time-window aggregations (1min, 1h, 1d).
- Alerting on thresholds.

3. Non-functional Requirements

- Ingestion: 1M events/sec.
- Query latency < 500ms.
- End-to-end latency < 5s.
- Availability 99.9%.

4. Capacity Estimation

1M events/sec. 100B events/day. Storage: 100B × 100B = 10TB/day.

5. Back-of-the-envelope Math

Kafka for ingestion. Flink/Spark Streaming for aggregation. Druid/ClickHouse for serving.

6. API Design

```
POST /events body: [{user_id, event_type, properties, timestamp}]
GET /analytics?metric=...&group_by=...&time_range=... -> aggregated data
```

7. Database Schema

```
// Aggregated storage (Druid/ClickHouse):
Table: events_agg (
  timestamp, event_type, dimension_1, dimension_2,
  count, sum_value, avg_value
)
  PARTITION BY timestamp, event_type
```

8. High-Level Design (HLD)

Producers → Kafka → Stream Processor (Flink) → Druid/ClickHouse → Dashboard API.

9. Low-Level Design (LLD)

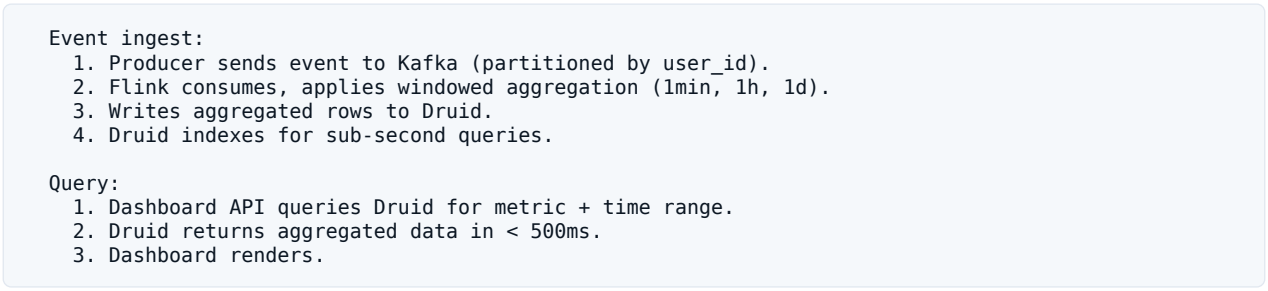
Events stream to Kafka. Flink consumes, applies windowed aggregations (1min, 1h, 1d), writes to Druid. Dashboard API queries Druid for sub-second responses.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Kafka:** Event ingestion buffer.
- **Flink:** Stream processing + windowed aggregation.
- **Druid/ClickHouse:** Columnar storage for fast queries.
- **Dashboard API:** Query + render.
- **Alerting:** Threshold-based alerts.

13. Technology Stack

- **Language:** Java/Scala (Flink), Java (Druid).
- **Queue:** Kafka.
- **Stream:** Flink or Spark Streaming.
- **Storage:** Druid or ClickHouse.

14. Database Choice

Druid (columnar, time-series optimized) for serving. Kafka for ingestion.

15. Cache Strategy

Redis for hot queries (top dashboards).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for events.

18. Scaling Strategy

Kafka scales by partition. Flink scales by parallelism. Druid scales by segment.

19. Fault Tolerance

Multi-AZ. Kafka persists during Flink outage.

20. Bottlenecks

- Event ingest throughput — Kafka partitions.
- Aggregation latency — Flink parallelism.
- Query latency — Druid segment size.

21. Trade-offs

- **Druid vs ClickHouse:** Druid is real-time optimized; ClickHouse is faster for ad-hoc.
- **Stream vs batch:** Stream is real-time; batch is cheaper.

22. CAP Theorem Analysis

AP. Eventual consistency on aggregations.

23. Security Considerations

TLS. PII scrubbing. Auth for dashboards.

24. Monitoring & Logging

Track ingestion lag, query latency, Druid segment count.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss Kafka + Flink + Druid pipeline.
- Mention windowed aggregation (1min, 1h, 1d).

- Discuss Druid for sub-second queries.

27. Common Follow-up Questions

- How would you handle late-arriving events?
- How would you implement funnel analysis?
- How would you handle schema evolution in events?

28. Common Mistakes

- DB for serving — too slow.
- No stream processor — batch only.
- No Kafka buffer — Flink crushed on burst.

29. Optimization Ideas

- Pre-aggregate common queries.
- Compress events.
- Use Druid rollups.

30. Final Summary

A real-time analytics platform tests streaming + columnar storage. Kafka ingests; Flink aggregates in windows (1min, 1h, 1d); Druid stores columnar for sub-second queries. The architecture rewards understanding of stream processing + columnar storage.

QUESTION 92

EXPERT

Design an IoT Platform

1. Problem Statement

Design an IoT platform. Millions of devices send telemetry; platform ingests, processes, and exposes for monitoring + control.

2. Functional Requirements

- Device registration + authentication.
- Telemetry ingestion (MQTT/HTTP).
- Real-time processing + alerting.
- Device control (commands).
- OTA firmware updates.

3. Non-functional Requirements

- Ingestion: 10M devices × 1 msg/sec = 10M msgs/sec.
- Latency: < 1s for alerts.
- Availability 99.9%.
- Survive device disconnection.

4. Capacity Estimation

10M devices, 10M msgs/sec. Storage: 10M × 100B × 86400 = 86TB/day.

5. Back-of-the-envelope Math

MQTT for lightweight device protocol. Kafka for ingestion. Time-series DB for storage.

6. API Design

```
// MQTT topics:
devices/{device_id}/telemetry (publish from device)
devices/{device_id}/commands (publish to device)

// HTTP API for management:
POST /devices body: {device_id, type}
GET /devices/{id}/telemetry?from=...&to=...
```

7. Database Schema

```
Table: devices (device_id, type, owner_id, status, last_seen, firmware_version)
Table: telemetry (device_id, timestamp, metrics_json) // time-series
Table: commands (command_id, device_id, type, payload, status, sent_at)
```

8. High-Level Design (HLD)

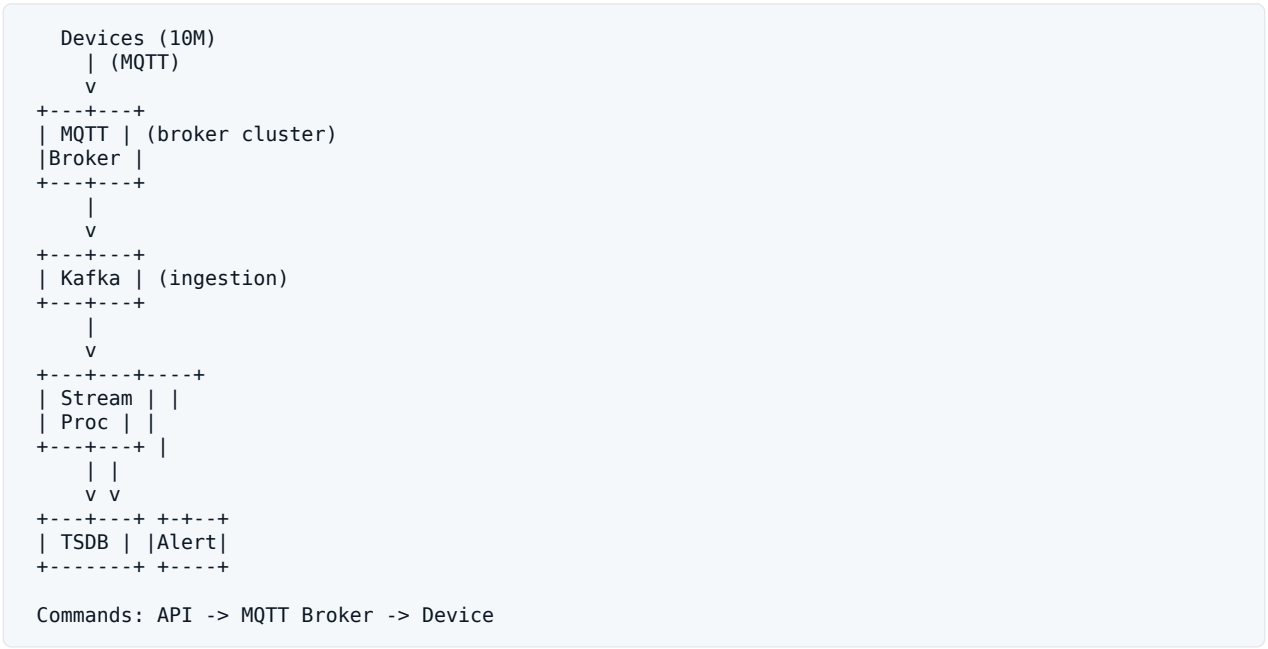
Device → MQTT Broker → Kafka → Stream Processor → Time-series DB + Alerting. Command Service → MQTT Broker → Device.

9. Low-Level Design (LLD)

Device connects to MQTT broker (per-device cert). Publishes telemetry. Broker forwards to Kafka. Stream processor aggregates + checks alert rules. Command Service publishes commands to MQTT topic for device.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Telemetry:
  1. Device publishes to MQTT topic devices/{id}/telemetry.
  2. Broker authenticates (cert), forwards to Kafka.
  3. Stream processor consumes, aggregates, checks alerts.
  4. Writes to time-series DB.
  5. If alert threshold crossed: notify via webhook/email.

Command:
  1. User triggers command via API.
  2. Command Service publishes to devices/{id}/commands.
  3. Device subscribes, receives, executes, acks.
```

12. Component Explanation

- **MQTT Broker:** Lightweight device protocol; per-device auth.
- **Kafka:** Ingestion buffer.
- **Stream Processor:** Flink for aggregation + alerting.
- **Time-Series DB:** InfluxDB/Cassandra for telemetry.
- **Command Service:** Sends commands to devices.
- **OTA Service:** Firmware updates via MQTT.

13. Technology Stack

- **Protocol:** MQTT (lightweight, built for IoT).
- **Language:** Java/Go.
- **DB:** InfluxDB or Cassandra (time-series).
- **Queue:** Kafka.
- **Stream:** Flink.

14. Database Choice

InfluxDB/Cassandra for telemetry (time-series optimized). PG for device metadata.

15. Cache Strategy

Redis for device state (last known values).

16. Load Balancer Strategy

MQTT broker cluster.

17. Message Queue Usage

Kafka for ingestion. MQTT for device comms.

18. Scaling Strategy

MQTT broker scales by connection count. Kafka scales by partition.

19. Fault Tolerance

Multi-AZ. Device reconnect on broker failure.

20. Bottlenecks

- MQTT connection count (10M) — broker cluster.
- Telemetry storage — downsample old data.
- Command delivery latency — MQTT QoS.

21. Trade-offs

- **MQTT vs HTTP:** MQTT is lightweight + persistent; HTTP is simpler.
- **Edge vs cloud processing:** Edge reduces latency; cloud is centralized.

22. CAP Theorem Analysis

AP. Telemetry eventually consistent.

23. Security Considerations

TLS. Per-device cert. OTA firmware signing. Network isolation.

24. Monitoring & Logging

Track ingestion rate, broker connections, alert latency.

25. Deployment Strategy

Multi-region for global devices.

26. Interview Tips

- Discuss MQTT for device protocol.
- Mention per-device cert auth.
- Discuss OTA firmware updates.

27. Common Follow-up Questions

- How would you handle 100M devices?
- How would you implement edge computing (process on gateway)?
- How would you detect device anomalies (ML)?

28. Common Mistakes

- HTTP for devices — too heavy.
- No per-device auth — security risk.
- No OTA — can't update firmware.

29. Optimization Ideas

- Batch telemetry (device sends every 30s, not 1s).
- Compress payloads.
- Downsample old telemetry.

30. Final Summary

An IoT platform tests lightweight protocols at scale. MQTT for device comms (lightweight, persistent, per-device cert). Kafka for ingestion. Flink for stream processing + alerting. Time-series DB for storage. The architecture rewards understanding of MQTT + stream processing.

QUESTION 93 EXPERT

Design a Multi-Region Active-Active Architecture

1. Problem Statement

Design a multi-region active-active architecture for a global SaaS application. Users in each region get low latency; data replicates across regions; survives region failure.

2. Functional Requirements

- Active-active: each region serves reads + writes.
- Cross-region replication.
- Automatic failover on region failure.
- Conflict resolution for concurrent writes.
- Per-user data residency (GDPR).

3. Non-functional Requirements

- Read latency < 50ms locally.
- Write latency < 100ms locally (async replication).
- Availability 99.99% (survives region failure).
- RPO < 1s, RTO < 60s.

4. Capacity Estimation

3 regions (US, EU, Asia). 10TB data per region. 1M writes/sec globally.

5. Back-of-the-envelope Math

Multi-leader DB per region. Async replication via log shipping. Conflict resolution via CRDT or LWW. Geographic routing via Anycast DNS.

6. API Design

```
// Standard API, routed to nearest region by Anycast DNS
PUT /data/{key} body: {value}
GET /data/{key}
```

7. Database Schema

```
// Per-region DB with replication log:
Table: data (key, value, version, origin_region, timestamp)
Table: replication_log (log_id, region, op_type, key, value, timestamp)
```

8. High-Level Design (HLD)

User → Anycast DNS → Nearest Region → Local App + DB. DB replicates async to other regions. Conflict resolver merges.

9. Low-Level Design (LLD)

Write: local DB + append to replication log. Replicator ships log to other regions. Conflict resolver merges (CRDT or LWW). Failover: DNS update redirects traffic.

10. Architecture Diagram

Architecture Diagram

```
   User (US) User (EU) User (Asia)
     | | |
     v (Anycast) v v
+---+---+ +---+---+ +---+---+
| US | | EU | | Asia |
| Region| | Region| | Region|
| App+DB| | App+DB| | App+DB|
+---+---+ +---+---+ +---+---+
     | | |
     +------+-------+-------+------+-------+--------+
            | | |
            v (async replication log shipping) v
       Conflict Resolver merges concurrent writes

Failover: Anycast DNS updates -> traffic redirects to healthy region
```

11. Data Flow Diagram

Data Flow Diagram

```
 Write:
   1. User in US writes; routes to US region.
   2. US DB writes; appends to replication log.
   3. Replicator ships log to EU + Asia (async).
   4. EU/Asia apply; conflict resolver merges concurrent writes.

 Failover:
   1. Region failure detected (health checks).
   2. Anycast DNS updates (withdraws failed region's route).
   3. Traffic redirects to healthy regions.
   4. Within 60s: full failover.
```

12. Component Explanation

- **Per-Region App + DB:** Local reads/writes.
- **Replicator:** Ships log across regions.
- **Conflict Resolver:** Merges concurrent writes (CRDT/LWW).
- **Anycast DNS:** Routes users to nearest healthy region.
- **Health Checker:** Detects region failure; triggers failover.

13. Technology Stack

- **Language:** Go/Java.
- **DB:** CockroachDB / Spanner / Cassandra (multi-region).
- **Algorithm:** CRDT or LWW for conflict resolution.
- **DNS:** Anycast BGP.

14. Database Choice

Multi-region DB (CockroachDB/Spanner/Cassandra) with built-in replication.

15. Cache Strategy

Local Redis per region.

16. Load Balancer Strategy

Anycast DNS for geographic routing.

17. Message Queue Usage

Replication log (custom or Kafka).

18. Scaling Strategy

Add regions for global coverage. Shard within region.

19. Fault Tolerance

Region failure: Anycast DNS failover. Other regions continue.

20. Bottlenecks

- Cross-region replication bandwidth — compress + batch.
- Conflict resolution cost — CRDT is O(1).
- Failover time — DNS TTL + health check interval.

21. Trade-offs

- **Multi-leader vs single-leader:** Multi has low write latency; single has strong consistency.
- **CRDT vs LWW:** CRDT preserves all writes; LWW loses concurrent.
- **Active-active vs active-passive:** Active-active uses all regions; passive wastes capacity.

22. CAP Theorem Analysis

AP. Accepts writes during partition; resolves on reconnect.

23. Security Considerations

TLS + mTLS between regions. Encryption at rest. Per-region compliance (GDPR).

24. Monitoring & Logging

Track replication lag, conflict rate, failover time.

25. Deployment Strategy

3+ regions. Anycast DNS.

26. Interview Tips

- Discuss multi-leader vs single-leader.

- Mention Anycast DNS for failover.
- Discuss conflict resolution (CRDT/LWW).

27. Common Follow-up Questions

- How would you handle strong consistency globally (Spanner)?
- How would you implement GDPR data residency?
- How would you test failover without user impact?

28. Common Mistakes

- Single-leader globally — high write latency.
- No conflict resolution — lost writes.
- Manual failover — slow RTO.

29. Optimization Ideas

- Compress replication log.
- Batch cross-region shipping.
- Use CRDT for conflict-free merging.
- Pre-warm DNS caches for fast failover.

30. Final Summary

Multi-region active-active tests global scale + conflict resolution. Multi-leader DB per region with async replication. CRDT or LWW for conflict resolution. Anycast DNS for geographic routing + failover. The system is AP — accepts writes during partition. The architecture rewards understanding of CRDTs + DNS failover.

QUESTION 94 EXPERT

Design an Event Streaming Platform (Confluent/Kafka)

1. Problem Statement

Design a complete event streaming platform: producer SDKs, broker cluster, schema registry, stream processing, and consumer SDKs. Multi-region, exactly-once.

2. Functional Requirements

- Producer SDK (multi-language).
- Broker cluster (partitioned, replicated).
- Schema registry (Avro/Protobuf).
- Stream processing (Kafka Streams/Flink).
- Consumer SDK (consumer groups).
- Exactly-once semantics.

3. Non-functional Requirements

- Throughput: 10M events/sec.
- Latency: < 10ms per event.
- Availability 99.99%.
- Multi-region replication.

4. Capacity Estimation

10M events/sec. 100TB/day. 7-day retention = 700TB.

5. Back-of-the-envelope Math

Partition = append-only log. RF=3. ISR for commit. Schema registry for compatibility. Kafka Streams for processing. Exactly-once via 2PC (transactional producer).

6. API Design

```
// Producer SDK:
producer.send(topic, key, value)

// Consumer SDK:
consumer.subscribe(topic)
for msg in consumer.poll():
    process(msg)
    consumer.commit()

// Stream processing (Kafka Streams DSL):
stream.filter(...).map(...).to(output_topic)
```

7. Database Schema

```
// Broker: per-partition log on disk.
// Schema registry: schemas + subject_versions (PG).
// Consumer offsets: internal topic __consumer_offsets.
```

8. High-Level Design (HLD)

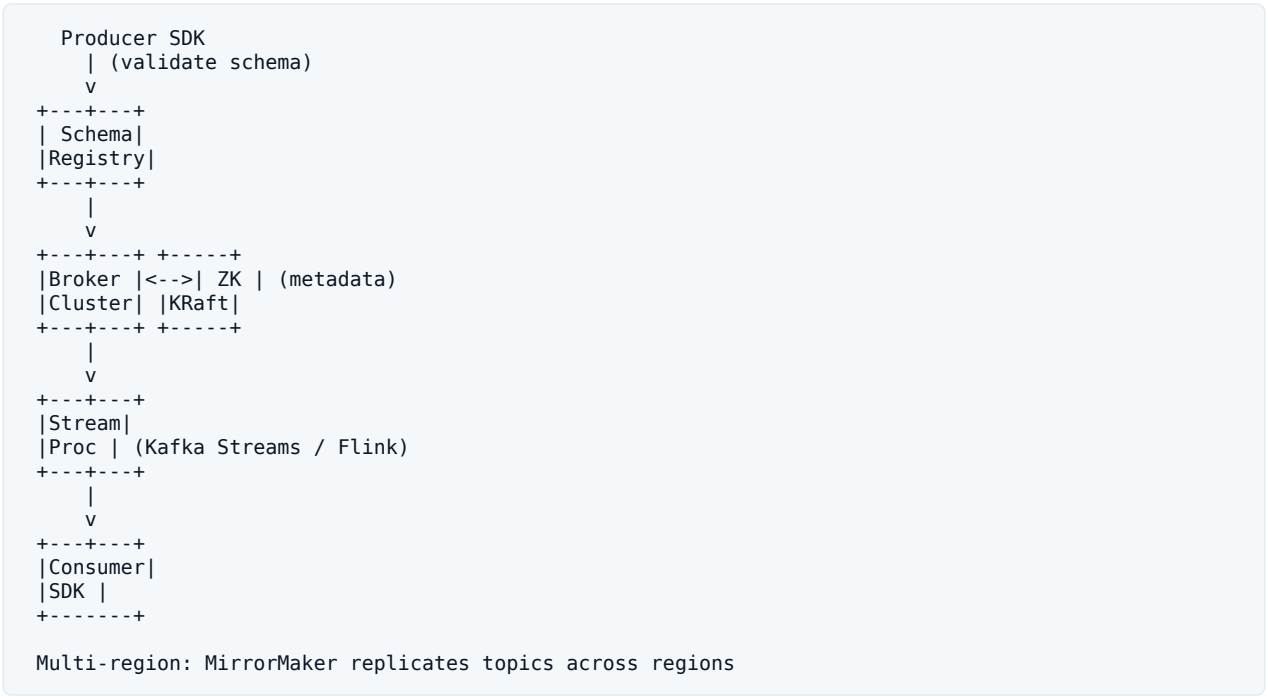
Producer → Broker Cluster → Consumer. Schema Registry validates on produce. Stream Processing (Kafka Streams) consumes + produces. Multi-region via MirrorMaker.

9. Low-Level Design (LLD)

Producer: pick partition, send to leader, ack based on acks. Broker: append + replicate to ISR, commit on quorum. Schema Registry: validate compatibility. Consumer: fetch + commit offset. Exactly-once: transactional producer + read-process-write pattern.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Produce + consume:
  1. Producer validates schema with Schema Registry.
  2. Sends to broker leader; leader replicates to ISR.
  3. On quorum ack: commit; ack to producer.
  4. Consumer fetches from leader; commits offset.

Exactly-once:
  1. Producer starts transaction.
  2. Read input topic → process → write output topic + commit offset (atomic).
  3. Transaction coordinator commits or aborts.
```

12. Component Explanation

- **Producer SDK:** Multi-language; schema validation.
- **Broker Cluster:** Partitioned, replicated logs.
- **Schema Registry:** Compatibility enforcement.
- **Stream Processing:** Kafka Streams or Flink.
- **Consumer SDK:** Consumer groups; offset tracking.
- **MirrorMaker:** Cross-region replication.

13. Technology Stack

- **Language:** Scala/Java (Kafka).
- **Storage:** Custom log files.
- **Coordination:** KRaft (built-in Raft) or ZooKeeper.
- **Schema:** Avro/Protobuf/JSON.

14. Database Choice

Custom log files. Schema Registry in PG/etcd.

15. Cache Strategy

OS page cache.

16. Load Balancer Strategy

Client-side routing by partition.

17. Message Queue Usage

This IS the message queue.

18. Scaling Strategy

Add brokers; reassign partitions. Partition count fixed at creation.

19. Fault Tolerance

Replication (RF=3). ISR-based commit. KRaft/ZK for metadata.

20. Bottlenecks

- Hot partitions — better partitioning key.
- Consumer lag — add consumers (up to partition count).
- Schema registry throughput — cache.

21. Trade-offs

- **acks=0/1/all:** 0 fastest, no durability; all slowest, full durability.
- **Exactly-once vs at-least-once:** Exactly-once has overhead; at-least-once is simpler.
- **ZK vs KRaft:** KRaft is built-in; ZK is external.

22. CAP Theorem Analysis

AP. ISR-based commit. Uncommitted may be lost.

23. Security Considerations

TLS. SASL. ACLs. Audit log.

24. Monitoring & Logging

Track throughput, consumer lag, ISR size, schema registry latency.

25. Deployment Strategy

Multi-AZ brokers. Multi-region via MirrorMaker.

26. Interview Tips

- Discuss partition log + ISR.
- Mention schema registry for compatibility.
- Discuss exactly-once via transactions.
- Mention MirrorMaker for multi-region.

27. Common Follow-up Questions

- How would you implement exactly-once semantics?
- How would you handle schema evolution (backward compat)?
- How would you implement Kafka Streams (stream processing)?
- How would you migrate from ZK to KRaft?

28. Common Mistakes

- No schema registry — incompatible producers/consumers.
- acks=1 — message loss on failover.
- No retention limit — disk fills.

29. Optimization Ideas

- Batch produces (linger.ms).
- Compress (snappy/lz4).
- Use KRaft (no ZK dependency).
- MirrorMaker 2 for multi-region.

30. Final Summary

An event streaming platform tests distributed log + ecosystem. Core: partitioned log with ISR replication. Schema Registry for compatibility. Kafka Streams for processing. Exactly-once via transactions. MirrorMaker for multi-region. The architecture rewards deep understanding of distributed logs + transactions.

QUESTION 95 EXPERT

Design an ML Inference Serving System

1. Problem Statement

Design a system for serving ML model predictions at scale. Support multiple models, A/B testing, canary deployments, and autoscaling by QPS.

2. Functional Requirements

- Load + serve multiple ML models.
- A/B testing (route traffic between model variants).
- Canary deployment (gradual rollout).
- Autoscaling by QPS + latency.
- Model versioning + rollback.

3. Non-functional Requirements

- Latency: p99 < 100ms.
- Availability 99.95%.
- Throughput: 100K predictions/sec.
- GPU utilization > 70%.

4. Capacity Estimation

100 models, 100K QPS. Model size: 1GB avg = 100GB total.

5. Back-of-the-envelope Math

Model server (TensorFlow Serving, Triton) loads models from registry. Load balancer routes by model + variant. GPU autoscaling by queue depth.

6. API Design

```
POST /predict body: {model_name, model_version, input} -> {prediction}
POST /models body: {name, version, s3_url} (deploy)
POST /experiments body: {name, variants: [{model, weight}]}
```

7. Database Schema

```
Table: models (model_id, name, version, s3_url, status: DEPLOYED|CANARY|ARCHIVED)
Table: experiments (experiment_id, model_name, variants_json, created_at)
Table: predictions (pred_id, model, variant, input_hash, output, latency, timestamp)
```

8. High-Level Design (HLD)

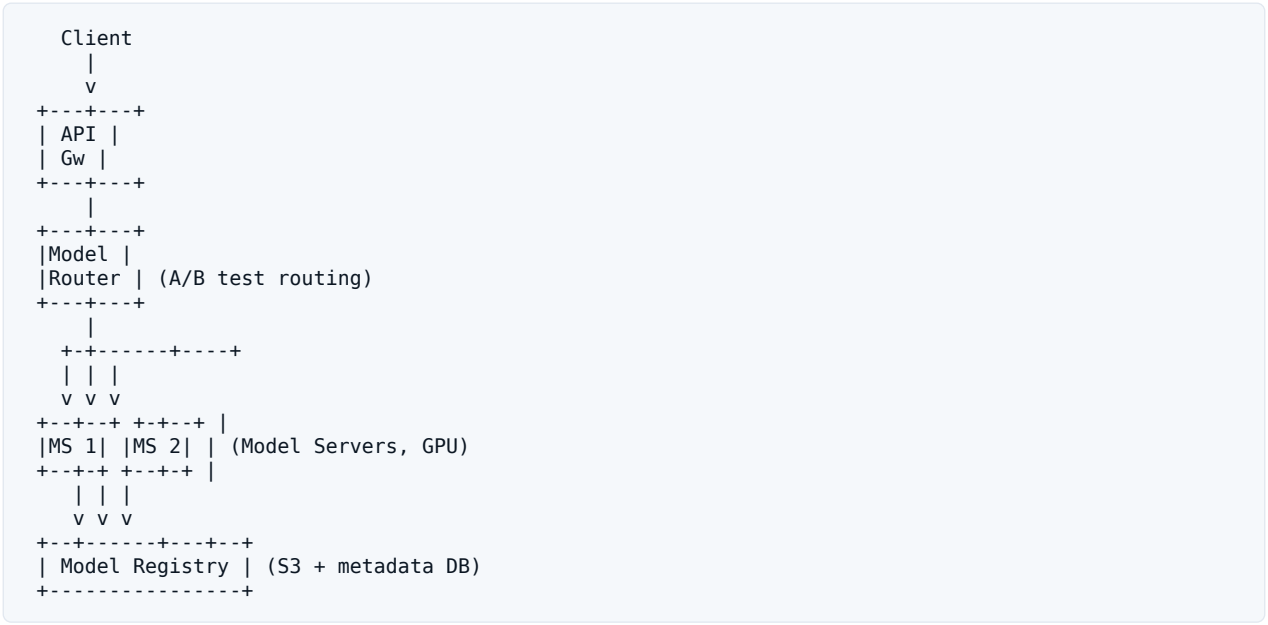
Client → API Gateway → Model Router → Model Server (GPU) → Prediction. Model Registry stores models in S3.

9. Low-Level Design (LLD)

Model server loads model from S3 into GPU memory. Router routes by model_name + variant (A/B test). Autoscaler scales GPU instances by queue depth + latency.

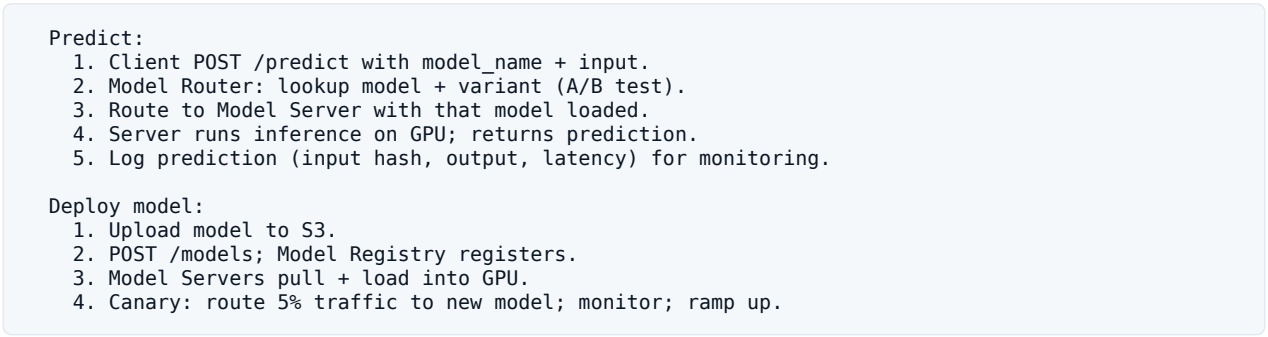
10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Model Router:** Routes by model + variant.
- **Model Server:** GPU; loads + serves models.
- **Model Registry:** S3 + metadata DB.
- **Autoscaler:** Scales GPU instances by QPS + latency.
- **Experiment Manager:** A/B test configuration.

13. Technology Stack

- **Language:** C++ (Triton) or Java (TF Serving).
- **ML:** TensorFlow, PyTorch, ONNX.
- **Storage:** S3 for models.

- **Cache:** Redis (hot predictions).

14. Database Choice

PG for model metadata + experiments. S3 for model files.

15. Cache Strategy

Redis for hot predictions (cache by input hash).

16. Load Balancer Strategy

L7 LB. Model-aware routing.

17. Message Queue Usage

Kafka for prediction logs (analytics, drift detection).

18. Scaling Strategy

GPU autoscaling by queue depth. Model servers scale by QPS.

19. Fault Tolerance

Multi-AZ. Model server failover. Model rollback on errors.

20. Bottlenecks

- GPU cost — batch inference.
- Model load time — pre-load on startup.
- Cold start — keep warm instances.

21. Trade-offs

- **GPU vs CPU:** GPU is faster for deep learning; CPU is cheaper.
- **Batch vs single inference:** Batch is higher throughput; single is lower latency.
- **Sync vs async prediction:** Sync confirms; async is faster.

22. CAP Theorem Analysis

AP. Predictions eventually consistent for A/B tests.

23. Security Considerations

TLS. Auth. Model IP protection. PII scrubbing in inputs.

24. Monitoring & Logging

Track prediction latency, GPU utilization, model drift, error rate.

25. Deployment Strategy

GPU node pools. Canary deployments.

26. Interview Tips

- Discuss model server (Triton, TF Serving).
- Mention A/B test routing.
- Discuss GPU autoscaling.
- Canary deployment for safety.

27. Common Follow-up Questions

- How would you detect model drift?
- How would you implement online learning?
- How would you handle large models (10GB+)?
- How would you serve LLMs (large language models)?

28. Common Mistakes

- CPU-only — slow inference.
- No A/B testing — risky deployments.
- No autoscaling — overprovisioned or underprovisioned.

29. Optimization Ideas

- Batch inference (multiple inputs per GPU call).
- Cache hot predictions.
- Use ONNX for cross-framework.
- Quantize models (FP16, INT8).

30. Final Summary

ML inference serving tests model deployment + serving. Model server (Triton, TF Serving) loads models from registry. Model router handles A/B test routing. GPU autoscaling by QPS + latency. Canary deployments for safe rollouts. The architecture rewards understanding of GPU infrastructure + model lifecycle.

QUESTION 96

EXPERT

Design a Vector Database (Pinecone/Milvus)

1. Problem Statement

Design a vector database for similarity search. Store + index high-dimensional vectors; return top-K nearest neighbors in < 100ms.

2. Functional Requirements

- Insert vectors with metadata.
- Search top-K nearest neighbors.
- Filter by metadata.
- Update + delete vectors.
- Multiple index types (HNSW, IVF).

3. Non-functional Requirements

- Search latency < 100ms.
- Throughput: 10K queries/sec.
- Availability 99.9%.
- Handle 1B vectors.

4. Capacity Estimation

1B vectors, 768 dims each = 1B × 768 × 4B = 3TB.

5. Back-of-the-envelope Math

HNSW (Hierarchical Navigable Small World) for approximate nearest neighbor. Sharded by vector ID. Hybrid: metadata filter + vector search.

6. API Design

```
POST /vectors body: [{id, vector, metadata}]
GET /search body: {query_vector, top_k, filter: {metadata}} -> [{id, score}]
DELETE /vectors/{id}
```

7. Database Schema

```
// Storage:
Table: vectors (id, vector_blob, metadata_json)
Index: HNSW on vector_blob // approximate nearest neighbor
Index: B-tree on metadata fields // for filtering
```

8. High-Level Design (HLD)

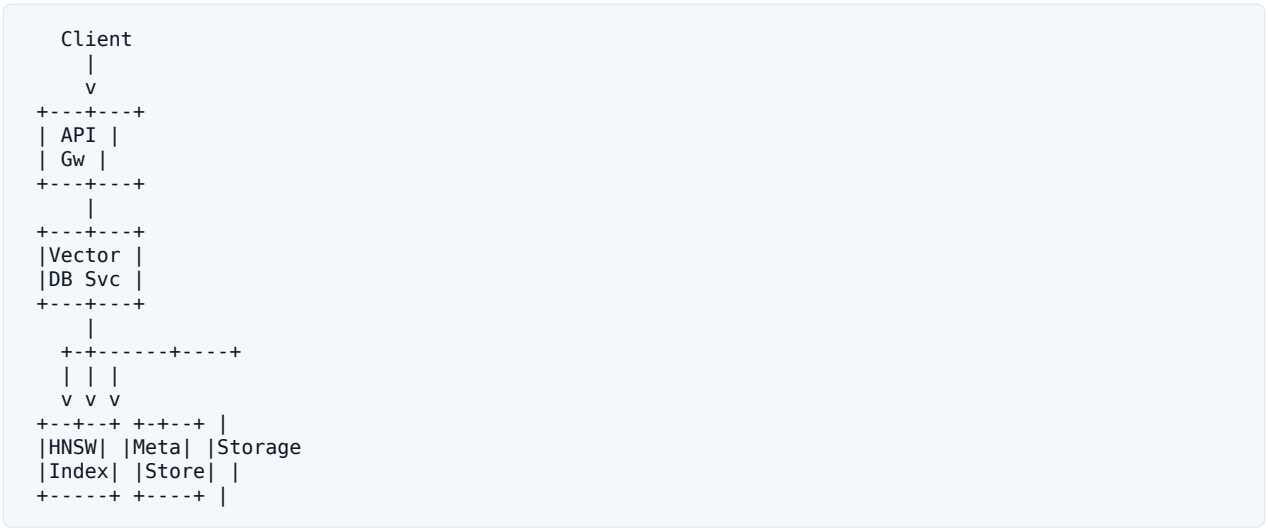
Client → API Gateway → Vector DB Service → HNSW Index (in-memory) + Metadata Store.

9. Low-Level Design (LLD)

Insert: store vector + metadata; add to HNSW index. Search: apply metadata filter (pre-filter or post-filter); query HNSW for top-K; return with scores.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Search:
  1. Client sends query vector + top_k + filter.
  2. Vector DB Service: apply metadata filter (pre-filter or post-filter).
  3. Query HNSW index for top-K nearest among filtered vectors.
  4. Return results with similarity scores.

Insert:
  1. Client sends vectors with metadata.
  2. Store vector + metadata.
  3. Add to HNSW index (incremental).
```

12. Component Explanation

- **Vector DB Service:** Coordinates search + insert.
- **HNSW Index:** Approximate nearest neighbor structure.
- **Metadata Store:** B-tree or inverted index for filtering.
- **Storage:** Disk-backed vector storage.

13. Technology Stack

- **Language:** C++ (perf-critical).
- **Algorithm:** HNSW, IVF, or ScaNN.

14. Database Choice

Custom (HNSW in-memory + disk storage).

15. Cache Strategy

In-memory HNSW index.

16. Load Balancer Strategy

L7 LB. Shard by vector ID.

17. Message Queue Usage

None.

18. Scaling Strategy

Shard by vector ID. Replicate for read scale.

19. Fault Tolerance

Multi-AZ. Replication.

20. Bottlenecks

- HNSW memory (1B × 768 × 4B = 3TB) — shard.
- Filter + search combination — pre-filter or post-filter trade-off.

21. Trade-offs

- **HNSW vs IVF:** HNSW is faster but more memory; IVF is lower memory.
- **Pre vs post filter:** Pre-filter is accurate; post-filter is faster.
- **Exact vs approximate:** Exact is slow; approximate is fast.

22. CAP Theorem Analysis

AP. Eventual consistency on inserts.

23. Security Considerations

TLS. Auth. Encryption at rest.

24. Monitoring & Logging

Track search latency, recall rate, index size.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss HNSW for ANN.
- Mention metadata filter + vector search combination.
- Discuss sharding by vector ID.

27. Common Follow-up Questions

- How would you handle 10B vectors?
- How would you implement hybrid search (vector + keyword)?
- How would you handle vector updates?

28. Common Mistakes

- Exact nearest neighbor — too slow.
- No metadata filter — limited usefulness.
- Single shard — can't scale.

29. Optimization Ideas

- Quantize vectors (FP16, INT8).
- Pre-compute filters.
- Shard by metadata (co-locate similar vectors).

30. Final Summary

A vector database tests approximate nearest neighbor search. HNSW index enables sub-100ms search on 1B+ vectors. Hybrid search (metadata filter + vector) for real-world use cases. Shard by vector ID for scale. The architecture rewards understanding of ANN algorithms + indexing.

QUESTION 97 EXPERT

Design an LLM Gateway (GenAI Platform)

1. Problem Statement

Design a gateway for serving LLMs (large language models). Handle prompt routing, token counting, rate limiting, caching, and multi-model orchestration.

2. Functional Requirements

- Route prompts to appropriate LLM (GPT, Claude, Llama).
- Token counting + billing.
- Rate limiting per user (tokens/min).
- Response caching (semantic cache).
- Multi-model orchestration (chain, ensemble).
- Streaming responses.

3. Non-functional Requirements

- Latency: < 5s for non-streaming, < 500ms for first token (streaming).
- Availability 99.9%.
- Cost optimization (route to cheaper model when possible).

4. Capacity Estimation

1M prompts/day. Avg prompt: 500 tokens. Avg response: 200 tokens.

5. Back-of-the-envelope Math

Gateway routes to model providers (OpenAI, Anthropic) or self-hosted (vLLM). Semantic cache via embedding similarity. Rate limit by tokens (not requests).

6. API Design

```
POST /completions body: {model, prompt, max_tokens, stream: true}
  -> SSE stream of tokens

POST /chat body: {messages: [...], model: "auto"} -> {response}
// model: "auto" routes to cheapest capable model
```

7. Database Schema

```
Table: prompts (prompt_id, user_id, model, prompt_text, response_text, tokens_in,
tokens_out, cost, timestamp)
Table: rate_limits (user_id, tokens_used_last_min, limit)
Table: cache (prompt_hash, prompt_text, response_text, embedding) // semantic cache
```

8. High-Level Design (HLD)

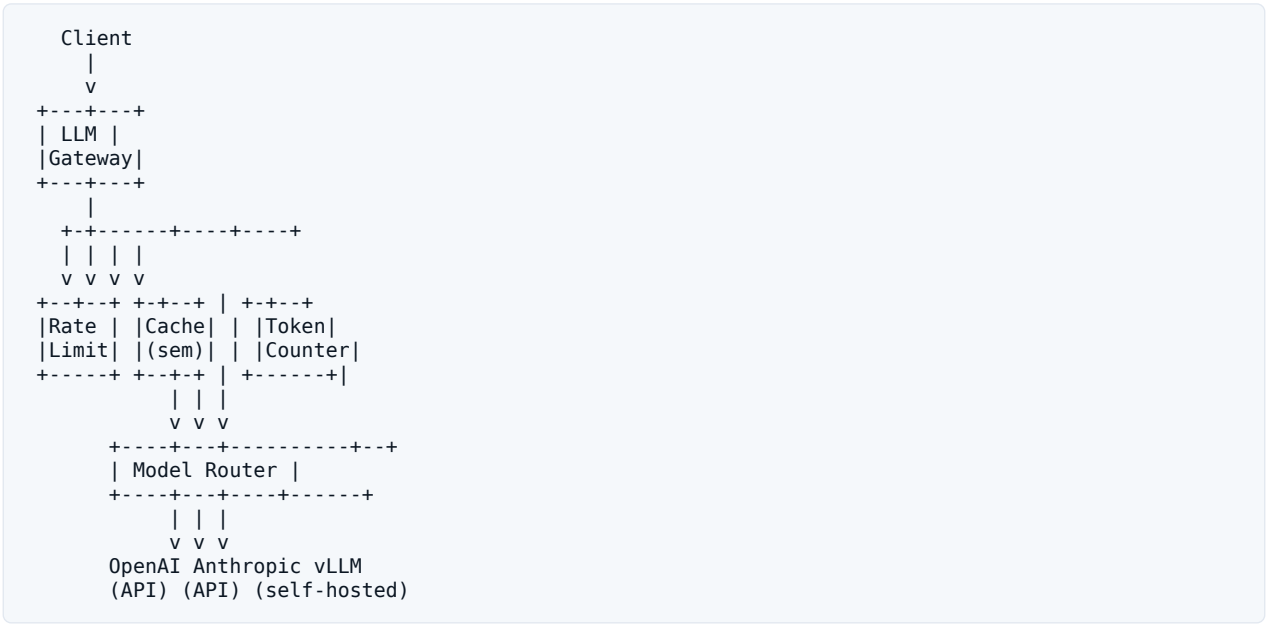
Client → LLM Gateway → Model Provider (OpenAI/Anthropic) or Self-hosted (vLLM). Semantic cache + rate limiter + token counter.

9. Low-Level Design (LLD)

On prompt: check rate limit (tokens). Check semantic cache (embedding similarity). If cache hit (similarity > 0.95): return cached response. Else: route to model, stream response, cache result.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Prompt:
  1. Client POST /completions with prompt + model.
  2. Gateway: count tokens; check rate limit.
  3. Check semantic cache (embedding similarity > 0.95).
  4. If cache hit: return cached response (free).
  5. If miss: route to model (auto = cheapest capable).
  6. Stream response; cache result; log cost.
```

12. Component Explanation

- **LLM Gateway:** Routes + caches + rate limits.
- **Model Router:** Picks model (cost vs capability).
- **Semantic Cache:** Embedding similarity for cache hits.
- **Rate Limiter:** Token-based (not request-based).
- **Token Counter:** Counts input + output tokens.
- **Model Providers:** OpenAI, Anthropic, vLLM (self-hosted).

13. Technology Stack

- **Language:** Python (FastAPI) or Go.
- **Cache:** Redis + Vector DB (semantic cache).

- **Models:** OpenAI, Anthropic APIs; vLLM for self-hosted.
- **Streaming:** SSE (Server-Sent Events).

14. Database Choice

PG for prompt logs. Redis + vector DB for semantic cache.

15. Cache Strategy

Semantic cache (vector DB) — similar prompts return cached responses.

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for prompt logs (analytics).

18. Scaling Strategy

Gateway scales horizontally. Self-hosted models scale by GPU.

19. Fault Tolerance

Multi-AZ. Model provider failover.

20. Bottlenecks

- Token counting latency — cache tokenizer.
- Semantic cache latency — vector DB ANN.
- Streaming throughput — SSE.

21. Trade-offs

- **API vs self-hosted:** API is simpler; self-hosted is cheaper at scale.
- **Semantic vs exact cache:** Semantic has more hits but may return slightly different.
- **Auto vs explicit model:** Auto saves cost; explicit is predictable.

22. CAP Theorem Analysis

AP. Cache may serve slightly stale responses.

23. Security Considerations

TLS. PII scrubbing in prompts. Per-user auth. Audit log.

24. Monitoring & Logging

Track latency, cache hit rate, cost per prompt, model usage.

25. Deployment Strategy

Multi-AZ. GPU clusters for self-hosted models.

26. Interview Tips

- Discuss semantic cache (embedding similarity).
- Mention token-based rate limiting.
- Discuss model routing (auto = cheapest capable).
- Streaming responses via SSE.

27. Common Follow-up Questions

- How would you implement multi-model orchestration (chain)?
- How would you handle long contexts (>100K tokens)?
- How would you implement guardrails (content filtering)?
- How would you fine-tune models for specific tasks?

28. Common Mistakes

- No cache — expensive + slow.
- Request-based rate limit — doesn't account for token cost.
- No streaming — poor UX for long responses.

29. Optimization Ideas

- Semantic cache (saves 30%+ cost).
- Route to cheaper model when possible.
- Batch prompts where possible.
- Compress prompts (remove redundancy).

30. Final Summary

An LLM gateway tests GenAI platform architecture. Semantic cache (embedding similarity) saves 30%+ cost. Token-based rate limiting (not request-based). Model routing (auto = cheapest capable). Streaming via SSE for UX. The architecture rewards understanding of vector search + cost optimization.

QUESTION 98 EXPERT

Design a Multi-Tenant SaaS Platform

1. Problem Statement

Design a multi-tenant SaaS platform. Each tenant (customer) has isolated data + config. Support tenant-specific customizations + per-tier resource limits.

2. Functional Requirements

- Tenant onboarding + management.
- Per-tenant data isolation.
- Per-tier resource limits (free, pro, enterprise).
- Tenant-specific customizations (themes, integrations).
- Per-tenant billing + usage tracking.

3. Non-functional Requirements

- Availability 99.95%.
- No cross-tenant data leakage.
- Per-tenant performance isolation.
- Handle 10K tenants.

4. Capacity Estimation

10K tenants. Avg 100 users/tenant = 1M users. Storage: 1TB/tenant = 10PB.

5. Back-of-the-envelope Math

Three isolation models: (1) shared DB + tenant_id column, (2) shared cluster + per-tenant schema, (3) per-tenant DB. Pick by tenant size.

6. API Design

```
POST /tenants body: {name, tier} -> {tenant_id}
GET /tenants/{id}/users
POST /tenants/{id}/config body: {theme, integrations}
GET /tenants/{id}/usage -> {api_calls, storage, users}
```

7. Database Schema

```
// Tenant isolation strategies:
// 1. Shared DB, tenant_id column (small tenants):
Table: users (user_id, tenant_id, name, ...)
  INDEX (tenant_id, user_id) -- tenant-scoped queries

// 2. Per-tenant schema (medium tenants):
Schema: tenant_{id}
  Table: users, projects, ...

// 3. Per-tenant DB (enterprise tenants):
Database: tenant_{id}
  Table: users, projects, ...
```

8. High-Level Design (HLD)

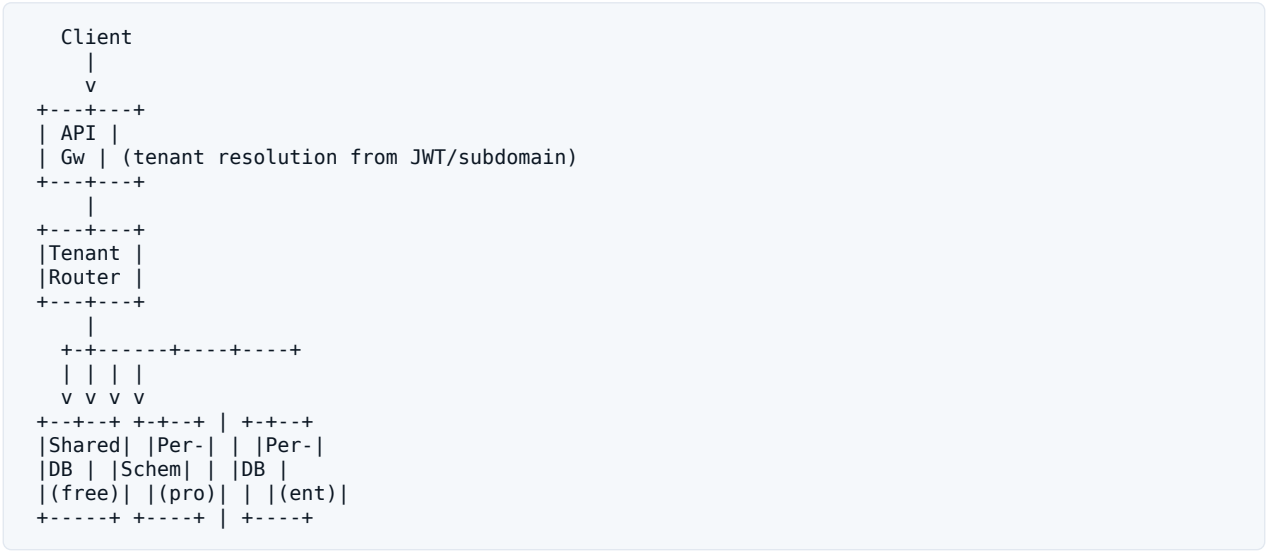
Client → API Gateway (tenant resolution) → App Service → Multi-Tenant DB. Tenant Router picks DB/schema based on tenant_id.

9. Low-Level Design (LLD)

On request: extract tenant_id from JWT/subdomain. Tenant Router decides isolation level (shared, schema, DB). Route to correct storage. Enforce resource limits.

10. Architecture Diagram

Architecture Diagram



11. Data Flow Diagram

Data Flow Diagram

```
Request:
  1. Client sends request with tenant_id (JWT or subdomain).
  2. API Gateway extracts tenant_id; passes to Tenant Router.
  3. Tenant Router: lookup tenant isolation level + resource limits.
  4. Route to correct DB/schema.
  5. Enforce limits (rate limit, storage quota).
  6. App Service executes; returns.
```

12. Component Explanation

- **API Gateway:** Tenant resolution from JWT/subdomain.
- **Tenant Router:** Picks DB/schema based on tenant tier.
- **Multi-Tenant DB:** Shared DB / per-tenant schema / per-tenant DB.
- **Quota Enforcer:** Per-tenant resource limits.
- **Billing Service:** Usage tracking + invoicing.

13. Technology Stack

- **Language:** Java/Go/Python.
- **DB:** PostgreSQL (supports all 3 isolation models).
- **Cache:** Redis (per-tenant cache keys).
- **Queue:** Kafka (per-tenant events).

14. Database Choice

PostgreSQL for all 3 isolation models (shared, schema, DB).

15. Cache Strategy

Redis with tenant_id in cache key.

16. Load Balancer Strategy

L7 LB. Tenant-aware routing.

17. Message Queue Usage

Kafka with tenant_id in event.

18. Scaling Strategy

Shared DB scales by sharding tenants. Per-tenant DB scales independently.

19. Fault Tolerance

Multi-AZ. Per-tenant backup + restore.

20. Bottlenecks

- Hot tenants (noisy neighbor) — per-tenant rate limit + isolation.
- Schema migrations across tenants — rolling per-tenant.
- Cross-tenant queries (analytics) — separate analytics DB.

21. Trade-offs

- **Shared vs isolated DB:** Shared is cheap; isolated is performant + secure.
- **Per-tenant schema vs DB:** Schema is middle ground; DB is full isolation.
- **Sync vs async provisioning:** Sync confirms; async is faster for large tenants.

22. CAP Theorem Analysis

CP. Per-tenant consistency.

23. Security Considerations

TLS. Per-tenant auth. Data isolation (no cross-tenant leakage). Encryption at rest. Per-tenant encryption keys (enterprise).

24. Monitoring & Logging

Track per-tenant latency, resource usage, errors.

25. Deployment Strategy

Multi-AZ. Per-tenant backup + restore.

26. Interview Tips

- Discuss 3 isolation models (shared, schema, DB).
- Mention tenant-aware routing.
- Discuss noisy neighbor problem.
- Per-tenant resource limits.

27. Common Follow-up Questions

- How would you migrate a tenant from shared to isolated DB?
- How would you implement per-tenant custom fields?
- How would you handle cross-tenant analytics?
- How would you implement per-tenant SSO?

28. Common Mistakes

- Single DB for all tenants — noisy neighbor.
- No tenant_id in queries — cross-tenant leakage.
- Same rate limit for all tenants — enterprise frustrated.

29. Optimization Ideas

- Tenant-aware caching (per-tenant keys).
- Per-tenant connection pools.
- Pre-provision enterprise tenant DBs.

30. Final Summary

A multi-tenant SaaS tests isolation at scale. Three models: shared DB (cheap, small tenants), per-tenant schema (medium), per-tenant DB (enterprise). Tenant Router picks model based on tier. Noisy neighbor mitigated by per-tenant rate limits + resource quotas. The architecture rewards understanding of isolation trade-offs.

QUESTION 99

EXPERT

Design a Blockchain Explorer (Etherscan)

1. Problem Statement

Design a blockchain explorer. Index on-chain data (blocks, transactions, addresses); provide search + analytics UI; handle high query volume.

2. Functional Requirements

- Index blocks + transactions in real-time.
- Search by address, block, transaction hash.
- Address analytics (balance, transaction history).
- Token + NFT tracking.
- API for developers.

3. Non-functional Requirements

- Indexing latency < 30s behind chain tip.
- Query latency < 500ms.
- Availability 99.9%.
- Handle 1B+ transactions indexed.

4. Capacity Estimation

1B transactions, 100M addresses. Storage: 1B × 1KB = 1TB transactions.

5. Back-of-the-envelope Math

Blockchain node → indexer → DB (PG + ES). Real-time updates via WebSocket. Aggregates (address balances) pre-computed.

6. API Design

```
GET /blocks/{number}
GET /transactions/{hash}
GET /addresses/{address} -> {balance, txn_count}
GET /addresses/{address}/transactions?cursor=...
GET /search?q=...
```

7. Database Schema

```
Table: blocks (number, hash, timestamp, miner, txn_count, gas_used)
Table: transactions (hash, block_number, from_addr, to_addr, value, gas, timestamp)
Table: addresses (address, balance, txn_count, first_seen, last_active)
Table: tokens (token_id, contract_addr, name, symbol, total_supply)
```

8. High-Level Design (HLD)

Blockchain Node → Indexer → PG (transactional) + ES (search) + Redis (cache). API + UI.

9. Low-Level Design (LLD)

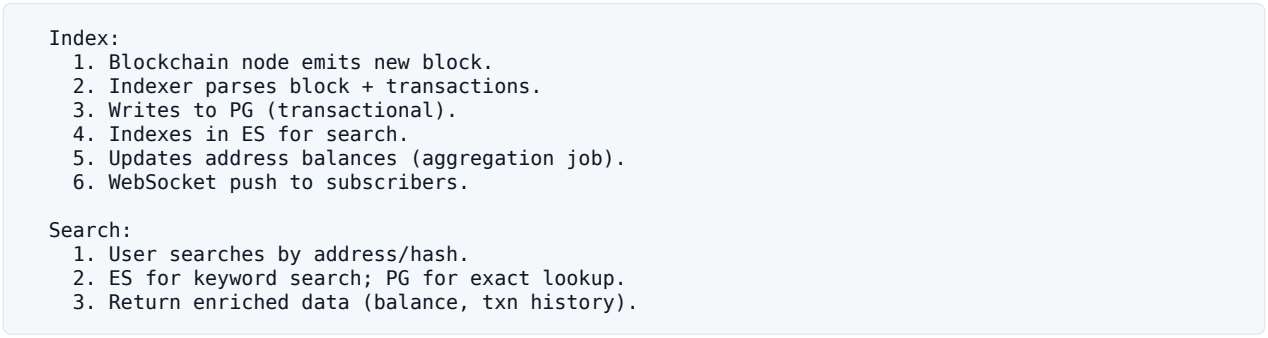
Indexer subscribes to new blocks; parses transactions; writes to PG + ES. Address balances pre-computed via aggregation job. Search via ES.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram



12. Component Explanation

- **Indexer:** Subscribes to chain; parses + stores.
- **PG:** Transactional storage (blocks, txns, addresses).
- **ES:** Search index.
- **Aggregation Job:** Pre-computes address balances.
- **WebSocket Hub:** Real-time updates.

13. Technology Stack

- **Language:** Go/Rust (indexer perf), Python (API).

- **DB:** PostgreSQL + Elasticsearch.
- **Cache:** Redis.
- **Realtime:** WebSocket.

14. Database Choice

PG for transactional data. ES for search. Redis for cache.

15. Cache Strategy

Redis for hot addresses (top 1000 by activity).

16. Load Balancer Strategy

L7 LB.

17. Message Queue Usage

Kafka for index events.

18. Scaling Strategy

Shard PG by block_number range. ES scales search.

19. Fault Tolerance

Multi-AZ. Re-index from chain on failure.

20. Bottlenecks

- Indexing throughput (chain tip speed) — parallel indexers.
- Storage growth — archive old data.
- Address aggregation — incremental updates.

21. Trade-offs

- **Real-time vs batch indexing:** Real-time is fresher; batch is cheaper.
- **PG vs ES for search:** PG is simpler; ES is faster.

22. CAP Theorem Analysis

AP. Eventual consistency on indexing.

23. Security Considerations

TLS. Read-only API. Rate limit.

24. Monitoring & Logging

Track indexing lag, query latency, storage growth.

25. Deployment Strategy

Multi-AZ.

26. Interview Tips

- Discuss indexer architecture.
- Mention ES for search.
- Discuss pre-computed address balances.

27. Common Follow-up Questions

- How would you implement token + NFT tracking?
- How would you handle chain reorganizations?
- How would you implement developer API (rate limit, billing)?

28. Common Mistakes

- PG-only search — too slow.
- No aggregation job — slow address queries.
- No reorg handling — incorrect data.

29. Optimization Ideas

- Pre-compute address balances.
- Cache hot addresses.
- Shard by block range.

30. Final Summary

A blockchain explorer tests real-time indexing + search. Indexer subscribes to chain, parses, stores in PG + ES. Pre-computed address balances for fast queries. WebSocket for real-time updates. The architecture rewards understanding of indexing pipelines + search.

QUESTION 100 EXPERT

Design a Digital Twin Platform

1. Problem Statement

Design a digital twin platform. Real-world assets (factories, cities, machines) have virtual models updated in real-time from IoT sensors; support simulation + prediction.

2. Functional Requirements

- Ingest IoT telemetry from assets.
- Maintain virtual model state per asset.
- Real-time visualization (3D).
- Simulation (what-if analysis).
- Prediction (ML on telemetry + model).

3. Non-functional Requirements

- Telemetry latency < 1s to model update.
- Visualization latency < 2s.
- Availability 99.9%.
- Handle 1M assets.

4. Capacity Estimation

1M assets × 10 sensors each = 10M sensors. 10M msgs/sec. Model state: 1M × 1MB = 1TB.

5. Back-of-the-envelope Math

IoT ingestion (MQTT + Kafka). Per-asset model state in Redis + DB. Simulation engine runs what-ifs on model. ML predictions on telemetry.

6. API Design

```
POST /assets body: {id, type, model_3d_url}
GET /assets/{id}/state -> {sensors, model_state}
POST /assets/{id}/simulate body: {scenario} -> {predicted_outcome}
// WebSocket /assets/{id}/stream for real-time state
```

7. Database Schema

```
Table: assets (asset_id, type, model_3d_url, location, status)
Table: sensors (sensor_id, asset_id, type, current_value, last_updated)
Table: telemetry (asset_id, sensor_id, value, timestamp) // time-series
Table: simulations (sim_id, asset_id, scenario, result, created_at)
```

8. High-Level Design (HLD)

IoT Devices → MQTT Broker → Kafka → Stream Processor → Asset Model Service (Redis + DB) + Simulation Engine + ML Predictor. 3D Visualization via WebSocket.

9. Low-Level Design (LLD)

Telemetry: device → MQTT → Kafka → stream processor updates asset model state in Redis (fast) + DB (durable). Visualization: WebSocket pushes state to 3D client. Simulation: engine loads model + scenario, runs physics/ML, returns predicted outcome.

10. Architecture Diagram

Architecture Diagram

11. Data Flow Diagram

Data Flow Diagram

```
Telemetry + model update:
  1. Sensor publishes to MQTT.
  2. MQTT broker → Kafka → stream processor.
  3. Stream processor updates asset model state in Redis + TSDB.
  4. WebSocket pushes state to 3D client (real-time visualization).

Simulation:
  1. User POST /simulate with scenario (e.g. "increase temperature by 10").
  2. Simulation Engine loads asset model + scenario.
  3. Runs physics simulation + ML prediction.
  4. Returns predicted outcome (e.g. "machine will fail in 2h").
```

12. Component Explanation

- **MQTT Broker:** IoT device ingestion.
- **Stream Processor:** Updates asset model from telemetry.

- **Asset Model Service:** Maintains current state per asset.
- **Simulation Engine:** Runs what-if scenarios.
- **ML Predictor:** Predicts failures/outcomes from telemetry.
- **3D Visualization:** WebSocket-based real-time client.

13. Technology Stack

- **Language:** C++ (simulation), Python (ML), Go (services).
- **Protocol:** MQTT for IoT.
- **DB:** Redis (state), InfluxDB (telemetry), PG (metadata).
- **Visualization:** Three.js / WebGL.
- **ML:** PyTorch.

14. Database Choice

Redis for current model state (fast). InfluxDB for telemetry time-series. PG for metadata.

15. Cache Strategy

Redis IS the model state cache.

16. Load Balancer Strategy

MQTT broker cluster. L7 LB.

17. Message Queue Usage

Kafka for telemetry. MQTT for IoT.

18. Scaling Strategy

MQTT scales by connection. Kafka by partition. Asset model sharded by asset_id.

19. Fault Tolerance

Multi-AZ. Model state recoverable from TSDB.

20. Bottlenecks

- Telemetry volume (10M msgs/sec) — shard Kafka.
- Simulation compute — GPU + parallel.
- 3D visualization bandwidth — level-of-detail.

21. Trade-offs

- **Real-time vs batch model update:** Real-time is fresher; batch is cheaper.
- **Edge vs cloud simulation:** Edge is faster; cloud is centralized.

22. CAP Theorem Analysis

AP. Model state eventually consistent.

23. Security Considerations

TLS. Per-device cert. OTA firmware. Network isolation.

24. Monitoring & Logging

Track telemetry lag, model update latency, simulation time.

25. Deployment Strategy

Multi-region. GPU clusters for simulation + ML.

26. Interview Tips

- Discuss IoT ingestion (MQTT + Kafka).
- Mention per-asset model state in Redis.
- Discuss simulation engine + ML predictor.
- 3D visualization via WebSocket.

27. Common Follow-up Questions

- How would you handle 100M assets?
- How would you implement predictive maintenance (ML)?
- How would you handle edge computing (process on gateway)?
- How would you implement collaborative simulation (multiple users)?

28. Common Mistakes

- DB-only model state — too slow.
- No simulation engine — just monitoring.
- No ML prediction — reactive not proactive.

29. Optimization Ideas

- Compress telemetry.
- Pre-compute common simulations.
- Level-of-detail for 3D (only render visible).
- Edge processing for low-latency decisions.

30. Final Summary

A digital twin platform tests IoT + simulation + ML. IoT telemetry (MQTT + Kafka) updates per-asset model state in Redis. Simulation engine runs what-if scenarios. ML predictor forecasts outcomes. 3D visualization via WebSocket. The architecture rewards understanding of IoT + simulation + ML pipelines.

PART VII

Comparison Tables

Side-by-side feature comparisons of the most-confused building blocks.

SQL vs NoSQL

The fundamental database choice. SQL (PostgreSQL, MySQL) provides ACID transactions and a rigid schema. NoSQL (MongoDB, Cassandra, Redis) trades some of these guarantees for horizontal scalability and flexible schemas.

Table — SQL vs NoSQL

Dimension	SQL	NoSQL
Schema	Rigid, predefined	Flexible, schema-less
Transactions	ACID	Eventual / BASE
Scaling	Vertical (primarily)	Horizontal (primarily)
Joins	Native, complex	Limited / none
Best for	Transactional, complex queries	High write throughput, flexible data
Examples	PostgreSQL, MySQL, Oracle	MongoDB, Cassandra, Redis, DynamoDB
Consistency	Strong	Tunable / eventual
Query language	SQL (standardized)	Varies (MongoDB Query, Cassandra CQL, etc.)

Takeaway

Use SQL for transactional systems (payments, inventory). Use NoSQL for high-throughput, schema-flexible workloads (events, telemetry, content).

Redis vs Memcached

Both are in-memory key-value stores, but they target different use cases. Memcached is simpler and faster for raw cache throughput; Redis offers persistence, replication, and rich data structures.

Table — Redis vs Memcached

Dimension	Redis	Memcached
Data types	Strings, lists, sets, hashes, sorted sets, streams	Strings only
Persistence	RDB snapshots + AOF log	None (in-memory only)
Replication	Master-replica, Cluster mode	None (each node independent)
Performance	~100K ops/sec	~200K ops/sec (slightly faster)
Max object size	512MB	1MB
Eviction	LRU, LFU, TTL	LRU

Dimension	Redis	Memcached
Use cases	Cache, session store, rate limiter, leaderboards	Pure cache (objects < 1MB)
Cluster	Built-in (Redis Cluster)	Client-side consistent hashing

Takeaway

Use Memcached for pure caching of small objects where durability does not matter. Use Redis for everything else (rich structures, persistence, HA).

Kafka vs RabbitMQ

Both are message brokers, but with very different models. Kafka is a durable log (consumers track their own offset); RabbitMQ is a traditional queue (messages deleted on consume).

Table — Kafka vs RabbitMQ

Dimension	Kafka	RabbitMQ
Model	Distributed commit log	Traditional message queue
Throughput	Millions/sec	~50K/sec
Message retention	Days/weeks (configurable)	Until consumed (then deleted)
Replay	Yes (consumers re-read from offset)	No (consumed messages gone)
Routing	Topic + partition	Rich (direct, topic, fanout, headers)
Ordering	Per partition	Per queue
Latency	~10ms	~1ms
Use cases	Event streaming, log aggregation, CDC	Work queues, request/reply, complex routing

Takeaway

Use Kafka for high-throughput event streaming where replay matters. Use RabbitMQ for complex routing, request/reply, or when messages can be discarded after consumption.

REST vs GraphQL

REST is the dominant API style; GraphQL solves REST's over/under-fetching problem but adds server complexity.

Table — REST vs GraphQL

Dimension	REST	GraphQL
Endpoints	Multiple (/users, /users/1, /users/1/orders)	Single (/graphql)
Payload shape	Server-defined	Client-defined (query specifies fields)
Over-fetching	Common	Eliminated
Under-fetching	Common (needs multiple calls)	Eliminated (nested queries)
Caching	HTTP cache (URL-based)	Harder (single endpoint, POST)
Versioning	URL (/v1/, /v2/)	Schema evolution (deprecate fields)
Learning curve	Low	Medium-high
Best for	CRUD, public APIs, mobile backends	Complex UIs, bandwidth-constrained clients

Takeaway

Use REST for simple CRUD and public APIs. Use GraphQL for complex UIs that need aggregated data from multiple services, or for mobile clients where minimizing requests matters.

Docker vs Kubernetes

Docker containerizes applications; Kubernetes orchestrates containers at scale. They solve different problems and are typically used together.

Table — Docker vs Kubernetes

Dimension	Docker	Kubernetes
What it is	Container runtime	Container orchestrator
Scope	Single host	Cluster of hosts
Scaling	Manual (docker run more)	Automatic (HPA, VPA, Cluster Autoscaler)
Load balancing	Docker Swarm (basic)	Built-in (Services + Ingress)
Self-healing	Restart policy	Automatic restart, reschedule on node failure
Rolling updates	Manual	Built-in (Deployments)
Service discovery	Docker DNS	Built-in (kube-dns)
Use case	Local dev, single-host prod	Multi-host prod, microservices

Takeaway

Docker packages your app; Kubernetes runs it at scale. Use Docker alone for local dev or single-host simple deployments. Use Kubernetes for production microservices.

Monolith vs Microservices

The architecture decision that defines your team structure and deployment cadence. Start monolith; extract microservices only when scale demands.

Table — Monolith vs Microservices

Dimension	Monolith	Microservices
Deployment	Single unit	Independent per service
Scaling	Whole app scales together	Per-service
Tech stack	Uniform	Polyglot (per service)
Team boundaries	Tight coupling	Loose coupling
Communication	In-process function calls	Network (HTTP, gRPC, events)
Transactions	ACID, simple	Distributed (sagas, outbox)
Operational complexity	Low	High (observability, deployment, networking)
Best for	Early-stage, small team, unclear domain	Large teams, independent scaling, stable domains

Takeaway

Start with a modular monolith. Extract microservices only when (1) team size > ~8 engineers per service, (2) scaling requirements differ across modules, or (3) deployment cadence differs.

Polling vs WebSockets

Two ways to get real-time updates from server to client. Polling is simpler but inefficient; WebSockets enable true bidirectional communication.

Table — Polling vs WebSockets

Dimension	Polling	WebSockets
Direction	Client → Server (repeated)	Bidirectional
Latency	High (depends on interval)	Low (instant push)
Overhead	High (HTTP headers per request)	Low (one handshake, then frames)
Connection	New per request	Persistent
Implementation	Simple (HTTP)	More complex (protocol upgrade, state)
Scaling	Easy (stateless)	Hard (stateful, sticky sessions)

Dimension	Polling	WebSockets
Best for	Low-frequency updates, fallback	Real-time chat, collaboration, live dashboards

> **Takeaway**
>
> Use WebSockets for real-time bidirectional communication (chat, collaboration). Use polling for low-frequency updates or when WebSockets are blocked (some corporate proxies). Long polling is a middle ground.

Sharding vs Replication

Both scale data, but in different dimensions. Sharding splits data across nodes; replication copies data across nodes.

Table — Sharding vs Replication

Dimension	Sharding	Replication
What it does	Splits dataset across nodes	Copies dataset across nodes
Scales	Write throughput + storage	Read throughput + availability
Each node holds	Subset of data	Full copy (or partial, in sharded replicas)
Routing	Shard key (hash, range)	Any replica (for reads)
Consistency	Strong per shard	Eventual (async) or strong (sync)
Failure handling	Lose one shard's data unless replicated	Continue with surviving replicas
Use cases	Write-heavy, large datasets	Read-heavy, HA

> **Takeaway**
>
> They compose: each shard has its own replicas. Sharding scales writes; replication scales reads and provides HA. Most production systems use both.

Leader-Follower vs Multi-Leader

Two replication topologies. Single-leader is simpler; multi-leader enables multi-region writes but requires conflict resolution.

Table — Leader-Follower vs Multi-Leader

Dimension	Leader-Follower	Multi-Leader
Write nodes	One (leader)	Multiple (all leaders)
Write latency	Low (local to leader)	Low (local to each leader)

Dimension	Leader-Follower	Multi-Leader
Conflict resolution	None (single source of truth)	Required (CRDT, LWW, custom)
Failover	Promote a follower	Continue with surviving leaders
Multi-region writes	No (must route to leader's region)	Yes (each region accepts writes)
Complexity	Low	High (conflict resolution)
Use cases	Most OLTP databases	Multi-region active-active, offline-first apps

Takeaway

Use single-leader when write latency to leader is acceptable. Use multi-leader for multi-region active-active or offline-first apps where conflict resolution is acceptable.

Consistency vs Availability (CAP)

During a network partition, you must choose between consistency and availability. This choice defines your system's behaviour under failure.

Table — Consistency vs Availability (CAP)

Dimension	Consistency (CP)	Availability (AP)
During partition	Reject reads/writes	Serve (possibly stale)
Steady state	Strong (linearizable)	Eventual
Use cases	Financial, configuration, locks	Social feeds, recommendations, search
Examples	Spanner, etcd, HBase	Cassandra, DynamoDB, eventually-consistent caches
Trade-off	Lower availability	Stale reads, conflict resolution
Latency	Higher (quorum)	Lower (any replica)

Takeaway

Pick CP when correctness is critical (money, locks, config). Pick AP when availability is critical and stale data is tolerable (social, search). Most systems are AP with tunable consistency.

PART VIII

Cheat Sheets

Concise reference pages for the most-asked system design concepts.

Cheat Sheet 1: Scalability Formulas

Memorize these — they appear in nearly every system design interview.

Formula	Equation	Example
Little's Law	Throughput = Concurrency / Latency	100 concurrent, 100ms latency = 1000 RPS
QPS from DAU	QPS = DAU × sessions × requests / 86400	1M DAU × 1 session × 100 reqs = ~1160 RPS avg
Peak QPS	Peak QPS = Avg QPS × 10 (typical)	1160 avg → 11,600 peak
Storage growth	Bytes/day = records/day × avg_size	1M records × 1KB = 1GB/day
5-year storage	5yr = bytes/day × 365 × 5 × 1.5 (overhead)	1GB/day × 1825 × 1.5 = 2.7TB
Bandwidth	Bits/sec = bytes/sec × 8	1GB/s = 8Gbps
Cache hit ratio	Hit ratio = hits / (hits + misses)	90% hits, 10% misses
Read:write ratio	Reads / Writes	100:1 for read-heavy (e.g. URL shortener)
Availability (nines)	Downtime/yr = (1 - availability) × 8760h	99.99% = 52min/yr
Replication factor	Quorum = (RF / 2) + 1	RF=3 → quorum=2

Cheat Sheet 2: CAP Theorem Quick Reference

Statement: During a network partition, a distributed system must choose between Consistency (C) and Availability (A). Partition tolerance (P) is non-negotiable in real distributed systems.

Choice	Behaviour	Examples
CP (Consistency + Partition tolerance)	Reject reads/writes during partition. Strong consistency.	etcd, Spanner, HBase, MongoDB (default)
AP (Availability + Partition tolerance)	Serve stale data during partition. Eventual consistency.	Cassandra, DynamoDB, eventually-consistent caches
CA (theoretical)	No partition scenario. Single-node only.	Single-node Postgres, Redis (without replication)

Common Misconception

CAP does not say "pick two forever." It says "during a network partition, you must choose between C and A." Outside of partitions, modern systems can offer both strong consistency and high availability via consensus protocols (Raft, Paxos).

Cheat Sheet 3: Load Balancing Algorithms

Algorithm	How it works	Best for
Round-robin	Cyclic distribution	Even load, identical backends
Weighted round-robin	Cyclic with weights	Heterogeneous backends (more to bigger)
Least-connections	Fewest in-flight requests	Long-lived connections (WebSocket)
Least-response-time	Lowest latency	Latency-sensitive apps
Consistent hashing	Hash key → node on ring	Session affinity, cache sharding
IP hash	Hash client IP	Sticky sessions (deprecated in favour of consistent hash)
Random	Random backend	Simple, even distribution at scale

Cheat Sheet 4: Caching Strategies

Pattern	How it works	Best for
Cache-aside	App reads cache; on miss, reads DB and writes cache	General purpose, read-heavy
Read-through	Cache fetches from DB on miss (transparent to app)	Simpler app code
Write-through	Writes go to cache + DB synchronously	Strong consistency, slower writes
Write-behind (write-back)	Writes go to cache; persisted to DB async	High write throughput, tolerates brief data loss
Refresh-ahead	Cache proactively refreshes hot items before expiry	Hot data, low latency

Cache Invalidation

Cache invalidation is one of the "two hard problems in computer science" (the other being naming). Common strategies: TTL (let it expire), explicit invalidation on write, and event-driven invalidation via pub/sub. There is no perfect solution — pick the tolerable staleness window for your use case.

Cheat Sheet 5: Database Selection Guide

Use case	Characteristics	Recommended
OLTP (transactions)	ACID, complex queries, moderate scale	PostgreSQL, MySQL
High write throughput	Time-series, events, logs	Cassandra, InfluxDB
Key-value (low latency)	Cache, session, counters	Redis, DynamoDB
Document (flexible schema)	CMS, content, profiles	MongoDB, Couchbase

Use case	Characteristics	Recommended
Search (full-text)	Search engines, autocomplete	Elasticsearch, Solr
Graph (relationships)	Social, fraud, recommendations	Neo4j, ArangoDB
Multi-region SQL	Global ACID	CockroachDB, Spanner, YugabyteDB
Blob storage	Images, videos, backups	S3, GCS, Azure Blob
Time-series	Metrics, IoT, monitoring	InfluxDB, TimescaleDB, Prometheus
In-memory	Cache, real-time analytics	Redis, Memcached

Cheat Sheet 6: Message Queue Selection

Use case	Recommended
High throughput, replay, event streaming	Kafka
Complex routing, request/reply, work queues	RabbitMQ
At-least-once, simple queue	AWS SQS
Pub/sub, fanout to many subscribers	AWS SNS, Google Pub/Sub
Low latency, in-memory	Redis Streams
Cloud-native, managed	AWS SQS/SNS, Google Pub/Sub, Azure Service Bus

Cheat Sheet 7: API Design Checklist

☐ Use nouns for resources, verbs for actions (/users, /users/1, /users/1/activate).

☐ Use HTTP methods correctly: GET (read), POST (create), PUT (full update), PATCH (partial), DELETE.

☐ Use plural nouns for collections (/users not /user).

☐ Version your API (/v1/, /v2/) — never break existing clients.

☐ Use query params for filtering, sorting, pagination (?status=active&sort;=name&page;=2).

☐ Return appropriate HTTP status codes (200, 201, 204, 400, 401, 403, 404, 409, 422, 429, 500, 503).

☐ Use idempotency keys for safe retries (POST /charges with Idempotency-Key header).

☐ Paginate all list endpoints (cursor-based preferred over offset).

☐ Include rate limit headers (X-RateLimit-Limit, X-RateLimit-Remaining, Retry-After).

☐ Use HATEOAS or document relations clearly (links to related resources).

☐ Standardize error format: {error: {code, message, details}}.

☐ Use ISO 8601 for dates (2026-07-05T14:30:00Z).

☐ Support both JSON and (optionally) Protobuf for high-throughput.

☐ Document with OpenAPI/Swagger; provide SDKs in common languages.

☐ Auth: OAuth 2.0 for third-party, JWT for first-party, API keys for server-to-server.

Cheat Sheet 8: Security Checklist

☐ TLS everywhere (no HTTP in production).

☐ mTLS for service-to-service in zero-trust networks.

☐ OAuth 2.0 for third-party access; JWT for first-party tokens (short-lived + refresh).

☐ Rate limit per user + per IP (prevent abuse).

☐ Input validation on every endpoint (schema, type, length).

☐ Output encoding (prevent XSS).

☐ Parameterized queries (prevent SQL injection).

☐ CSRF tokens for state-changing operations.

☐ CORS configured explicitly (no wildcard in production).

☐ Encryption at rest (AES-256 for databases, S3 SSE).

☐ PII encryption (separate key per user if possible).

☐ PCI DSS compliance for payment data (tokenization).

☐ Audit log for all sensitive operations.

☐ Secrets in vault (AWS Secrets Manager, HashiCorp Vault), not env vars.

☐ Rotate keys regularly (90 days for access keys, 1 year for TLS certs).

☐ WAF for public endpoints (SQL injection, XSS, rate limit).

☐ DDoS protection (Cloudflare, AWS Shield).

☐ Penetration testing annually.

☐ Bug bounty program for external researchers.

☐ Compliance: GDPR, CCPA, HIPAA, SOC 2 as applicable.

Cheat Sheet 9: Performance Optimization Checklist

☐ **Measure first:** profile before optimizing. Use p99 latency, not average.

☐ **Cache aggressively:** CDN for static, Redis for hot data, in-memory LRU for hot keys.

☐ **Batch operations:** reduce network round-trips (pipeline Redis, batch DB inserts).

☐ **Async where possible:** message queues decouple slow operations from fast requests.

☐ **Connection pooling:** reuse TCP connections (HTTP keep-alive, DB pool).

☐ **Index properly:** add indexes for WHERE/JOIN/ORDER BY columns; remove unused.

☐ **Denormalize for reads:** pre-compute joins; use materialized views.

☐ **Shard by access pattern:** co-locate data queried together.

☐ **Compress:** gzip/br for HTTP, snappy/lz4 for messages, columnar for analytics.

☐ **Lazy load:** defer non-critical work; stream large responses.

☐ **Pre-compute:** hourly/daily aggregates instead of computing on read.

☐ **Edge compute:** push logic to CDN edge (Cloudflare Workers) for low latency.

☐ **Use SSDs:** random read latency matters more than sequential throughput.

☐ **Right-size instances:** profile CPU vs memory vs network; pick balanced instance.

☐ **Auto-scale:** scale horizontally on queue depth or latency, not just CPU.

☐ **Avoid N+1 queries:** use eager loading (includes/joins).

☐ **Database read replicas:** route reads to replicas; writes to primary.

☐ **Circuit breakers:** fail fast on slow downstream (avoid cascading failures).

☐ **Backpressure:** reject requests when overloaded (load shed).

☐ **Profile regularly:** production tracing (OpenTelemetry) reveals real bottlenecks.

Cheat Sheet 10: System Design Interview Framework (5-Step Method)

Use this 5-step framework to structure any system design interview. Spend the indicated time on each step in a 45-minute interview.

1. Clarify Requirements (5 min)

Ask clarifying questions: Who are the users? What is the expected scale (DAU, QPS)? Read-heavy or write-heavy? Latency SLAs? Consistency requirements? Write down explicit functional + non-functional requirements. **Never start designing before scoping.**

2. Capacity Estimation (5 min)

Back-of-the-envelope math: QPS (avg + peak), storage (5-year), bandwidth, cache size. These numbers shape every later decision (single DB vs sharded, need CDN, etc.).

3. High-Level Design (10 min)

Draw the box-and-arrow diagram: client → CDN/LB → services → DB/cache/queue. Identify major components. Discuss API design (REST endpoints). Discuss data model (tables/collections).

4. Deep Dive (15 min)

Pick 1-2 subsystems to deep-dive. Interviewer may direct. Discuss: data flow, scaling, failure modes, trade-offs. **Name alternatives you rejected and why.**

5. Bottlenecks + Wrap-up (10 min)

Identify bottlenecks (single points of failure, hot spots). Discuss monitoring, deployment, security. Summarize the design in 30 seconds. Ask if interviewer has questions.

> **Red Flags (Avoid These)**
>
> Jumping into solution before scoping. Not calculating numbers. Picking tech without justification. No trade-off discussion. No failure-mode analysis. No monitoring/deployment plan. Running out of time without summarizing.

PART IX

Final Mock Interviews

Twenty complete interview walkthroughs with candidate thinking, interviewer feedback, and evaluation rubrics.

MOCK INTERVIEW 01

Design Twitter Timeline at Scale

Scenario

You are interviewing for a Senior SWE role at Meta. The interviewer asks you to design the home timeline for Twitter/X at 500M DAU scale.

Problem Statement

Design the home timeline system. When a user opens the app, they should see recent tweets from accounts they follow, ranked by recency + engagement. Target p99 latency < 500ms.

Candidate Thinking Process

Candidate starts by clarifying: 100:1 read:write ratio, 500M tweets/day, follow graph avg 200 followees per user. Recognizes this is read-heavy — push model makes sense. Asks about celebrity accounts — interviewer confirms >10M follower accounts need special handling.

Step-by-Step Solution

Hybrid push-pull: fanout worker pushes tweet_id to follower Redis lists on publish (normal users). For celebrities (>100K followers), skip push; pull at read time and merge. Timeline read: fetch user's Redis list (last 50) + pull celebrity tweets; rank by recency; hydrate.

Architecture Diagram

Diagram

```
Publish: Tweet -> PG + Kafka -> Fanout Worker -> Redis lists (per user)
Read: User -> Timeline Svc -> Redis list (last 50) + Pull celebrity tweets
       -> Rank -> Hydrate -> Return
```

Interviewer Feedback

Strong candidate. Clarified scale, picked hybrid push-pull with celebrity exception, discussed failure modes (Redis down → fall back to live pull). Could improve: discuss seen-set for de-duplication.

Strong Answer Example

> **Strong Answer**
>
> "I'll use a hybrid push-pull model. For normal users (< 100K followers), a fanout worker pushes tweet_ids to per-follower Redis lists on publish. For celebrities, push is too expensive, so we pull their recent tweets at read time and merge. Read path: fetch user's Redis list, merge celebrity tweets, rank by recency + engagement, hydrate. Seen-set in Redis prevents duplicates. If Redis is down, fall back to live pull from Cassandra."

Weak Answer Example

> **Weak Answer**
>
> "I'd store tweets in a database and query them on read. Maybe add Redis for caching." — Too vague, no scale awareness, no architecture.

Evaluation Rubric

Dimension	Score
Requirements clarification	15/15
Capacity estimation	12/15
High-level design	18/20
Deep dive (push-pull)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	10/15

MOCK INTERVIEW 02

Design WhatsApp Message Delivery

Scenario

Senior SWE at Meta. Design WhatsApp's message delivery system at 2B user scale.

Problem Statement

Design 1:1 chat with sub-second delivery, offline message support, and end-to-end encryption. 100B messages/day.

Candidate Thinking Process

Candidate clarifies: 100B msgs/day = ~1.2M msgs/sec. E2E encryption means server cannot read content — just route encrypted blobs. Online delivery via WebSocket; offline via push notifications.

Step-by-Step Solution

Connection layer: 50 servers × 2M WebSocket conns each. Message router: stateless, looks up recipient connection via Redis session registry. Persist to Cassandra (sharded by conversation_id). Push notifications for offline users. E2E via Signal Protocol (Double Ratchet).

Architecture Diagram

Diagram

```
Sender -> Conn Svc -> Message Router -> Persist + Fanout
Router -> Recipient Conn Svc (if online) -> Recipient
Router -> Push Svc (if offline) -> FCM/APNs
E2E: clients exchange keys via Signal Protocol
```

Interviewer Feedback

Strong. Discussed connection vs routing separation, E2E implications (no server-side search), offline delivery via push. Could improve: group chat fanout (1024 members).

Strong Answer Example

Strong Answer

"Separate connection layer (WebSocket, 50 servers × 2M conns) from message routing (stateless, scales horizontally). Router persists to Cassandra (sharded by conversation_id), looks up recipient via Redis session registry, pushes to online or queues push for offline. E2E via Signal Protocol — server only sees encrypted blobs."

Weak Answer Example

Weak Answer

"Use a database to store messages and a WebSocket to push them." — No scale, no offline handling, no E2E.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15

Dimension	Score
High-level design	19/20
Deep dive (delivery)	18/20
Trade-offs (E2E vs features)	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 03

Design Uber Surge Pricing

Scenario

Staff Engineer at Uber. Design the surge pricing system.

Problem Statement

Compute surge multipliers per geocell in real-time based on demand/supply imbalance. Update every minute. Affect pricing in the rider app.

Candidate Thinking Process

Candidate clarifies: demand = active ride requests; supply = available drivers. Compute ratio per geocell. Surge = function of ratio (e.g. ratio > 2 → 2x). Update every minute, push to rider app.

Step-by-Step Solution

Surge Service: consumes ride request events + driver location updates from Kafka. Maintains per-geocell demand + supply counters in Redis. Every minute, compute ratio → multiplier. Publish to rider app via WebSocket.

Architecture Diagram

Diagram

```
Ride requests + Driver locations -> Kafka -> Surge Svc
Surge Svc: per-geocell counters in Redis
Every 1 min: compute ratio -> multiplier
Push to rider app via WebSocket
```

Interviewer Feedback

Strong. Recognized surge as separate service with its own compute cycle. Discussed geocell-based aggregation. Could improve: how to handle geocell boundaries (smooth surge between adjacent cells).

Strong Answer Example

Strong Answer

"Surge is a separate service consuming ride requests + driver locations from Kafka. Maintains per-geocell demand + supply counters in Redis. Every minute, computes ratio per cell, derives multiplier via piecewise function. Smooths between adjacent cells to avoid cliff at boundaries. Pushes to rider app via WebSocket. Pricing applied at ride request time."

Weak Answer Example

Weak Answer

"Just increase prices when there are more riders than drivers." — No architecture, no real-time computation, no granularity.

Evaluation Rubric

Dimension	Score
Requirements clarification	13/15

Dimension	Score
Capacity estimation	12/15
High-level design	18/20
Deep dive (surge algorithm)	17/20
Trade-offs	12/15
Bottlenecks + monitoring	10/15

MOCK INTERVIEW 04

Design Netflix Recommendation Pipeline

Scenario

Senior SWE at Netflix. Design the recommendation pipeline.

Problem Statement

Generate personalized "Top Picks" rows for 250M subscribers. Refresh hourly. Sub-second serving latency.

Candidate Thinking Process

Candidate clarifies: 250M users, 50K titles. Need per-user top-N. Pre-compute offline (Spark); serve from cache. Real-time bias for recently watched.

Step-by-Step Solution

Pipeline: watch events → Kafka → Spark (hourly batch) → ML model (matrix factorization + DL) → top-100 per user → Redis. Serving: fetch from Redis; apply real-time bias (boost recently watched genre); filter seen; rank.

Architecture Diagram

Diagram

```
Watch events -> Kafka -> Spark (hourly) -> ML model
ML: top-100 per user -> Redis
Serving: User -> Rec Svc -> Redis (top-100) -> Real-time bias -> Filter seen -> Rank
```

Interviewer Feedback

Strong. Discussed offline pre-compute + real-time bias pattern. Mentioned ML pipeline. Could improve: cold-start for new users.

Strong Answer Example

Strong Answer

"Hybrid: offline pre-compute + real-time bias. Watch events stream to Kafka → Spark hourly batch → ML model (matrix factorization + deep learning) → top-100 per user cached in Redis. Serving: fetch top-100 from Redis, apply real-time bias (boost genre of recently watched), filter seen, rank. Cold-start: use popularity + demographics for new users until enough history."

Weak Answer Example

Weak Answer

"Use ML to recommend videos." — No pipeline, no serving architecture.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	19/20

Dimension	Score
Deep dive (ML pipeline)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 05

Design a Distributed Rate Limiter

Scenario

Senior SWE at Stripe. Design a distributed rate limiter for the API gateway.

Problem Statement

Limit requests per user + per API. 100K req/sec. Sub-millisecond latency. Multiple algorithms.

Candidate Thinking Process

Candidate clarifies: per-user + per-API limits. Need atomic check + increment. Redis Lua script for atomicity. Local cache for hot keys.

Step-by-Step Solution

Rate Limiter Service: stateless. Redis Cluster for shared state. Lua script for atomic check + increment (token bucket, sliding window). Local cache for hot keys (1ms TTL) reduces Redis load.

Architecture Diagram

Diagram

```
Gateway -> Rate Limiter Svc -> Redis (Lua script, atomic)
Local cache for hot keys (1ms TTL)
Algorithms: token bucket, sliding window, leaky bucket
```

Interviewer Feedback

Strong. Discussed atomicity via Lua, local cache for hot keys, multiple algorithms. Could improve: Redis failure handling (degrade to local limits).

Strong Answer Example

> **Strong Answer**
>
> "Stateless rate limiter in front of Redis Cluster. Lua script for atomic check + increment (supports token bucket, sliding window, leaky bucket). Local cache for hot keys (1ms TTL) reduces Redis load by 80%. On Redis failure: degrade to local-only limits (less accurate but available). Return 429 with Retry-After + X-RateLimit-Remaining headers."

Weak Answer Example

> **Weak Answer**
>
> "Use Redis to count requests per user." — No atomicity, no failure handling, no algorithms.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (atomicity + algorithms)	19/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 06

Design Google Drive Sync

Scenario

Senior SWE at Google. Design the file sync system.

Problem Statement

Sync files across devices in real-time. Handle conflicts. Offline edits. Up to 1TB files.

Candidate Thinking Process

Candidate clarifies: file chunking for large files, dedup by hash, conflict resolution via versioning. WebSocket for sync notifications.

Step-by-Step Solution

Files chunked into 4MB blocks; SHA-256 each; dedup. Block storage in S3. Metadata in sharded Postgres. WebSocket notifications on file change. Versioning via immutable block references.

Architecture Diagram

Diagram

```
Upload: chunk file -> hash blocks -> dedup -> S3 -> metadata PG -> notify others
Sync: WebSocket push -> other devices fetch new version
```

Interviewer Feedback

Strong. Discussed chunking + dedup (saves 30-50% storage). Versioning via immutable blocks. Could improve: collaborative editing (CRDT).

Strong Answer Example

> **Strong Answer**
>
> "Chunk files into 4MB blocks; SHA-256 each; dedup by hash (saves 30-50% storage). Blocks in S3; metadata in sharded Postgres. Upload: client asks server which blocks are new, only uploads those. Versioning: each version is a list of block hashes (immutable). WebSocket notifies other devices on change. Offline edits: client tracks locally, syncs on reconnect with version-based conflict detection."

Weak Answer Example

> **Weak Answer**
>
> "Store files in S3 and notify other devices." — No chunking, no dedup, no conflict resolution.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	19/20

Dimension	Score
Deep dive (chunking + dedup)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 07

Design Airbnb Search at Scale

Scenario

Senior SWE at Airbnb. Design the search system.

Problem Statement

Search 5M listings by location, dates, guests, filters. Sub-500ms latency. Handle date-scoped availability.

Candidate Thinking Process

Candidate clarifies: hybrid search (keyword + geo). Date-scoped inventory. Need fast radius queries.

Step-by-Step Solution

Hybrid: Elasticsearch for keyword/category + Redis GEO for location radius. Date-scoped inventory materialized per-night. Pre-compute availability summaries for fast filtering.

Architecture Diagram

Diagram

```
Search: ES for keyword + Redis GEO for radius
Merge + filter by date availability (per-night inventory)
Rank by relevance + price + reviews
```

Interviewer Feedback

Strong. Discussed hybrid search pattern. Date-scoped inventory. Could improve: dynamic pricing integration.

Strong Answer Example

Strong Answer

"Hybrid search: Elasticsearch for keyword/category matching + Redis GEO for fast radius queries. Date-scoped inventory materialized per-night (listing_id, date, available, price). Pre-compute availability summaries (next 30 days) for fast UI rendering. Merge ES + GEO results, filter by date availability, rank by relevance + price + reviews. Cache common searches for 5 min."

Weak Answer Example

Weak Answer

"Use Elasticsearch to search listings." — No geo, no date inventory.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20
Deep dive (hybrid search)	17/20

Dimension	Score
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 08

Design a Notification System

Scenario

Senior SWE at Slack. Design the notification system.

Problem Statement

Multi-channel (email, push, SMS, in-app) notifications. 10M notifications/day. No duplicates. User preferences.

Candidate Thinking Process

Candidate clarifies: idempotency via (user_id, key). Per-channel topics. Retry with backoff. User preferences (opt-out).

Step-by-Step Solution

Producer → Notification Service (validate, dedup via unique constraint) → Kafka per channel → Consumer → Provider API. Retry with exponential backoff. Dead letter queue for permanent failures.

Architecture Diagram

Diagram

```
Producer -> Notification Svc (dedup) -> Kafka (per channel)
Consumer -> Provider API (SendGrid, FCM, Twilio)
Retry with backoff. DLQ for permanent failures.
```

Interviewer Feedback

Strong. Discussed idempotency, per-channel topics, retries, DLQ. Could improve: digest emails (batch).

Strong Answer Example

Strong Answer

"Producer → Notification Service validates + dedup via (user_id, idempotency_key) unique constraint. Publishes to per-channel Kafka topics (Email, SMS, Push, InApp) — isolates failures. Consumers call provider APIs (SendGrid, FCM, Twilio) with retry + exponential backoff. Max 3 retries; then dead letter queue. User preferences cached in Redis (opt-out per channel/type). Digest emails: batch into daily summary."

Weak Answer Example

Weak Answer

"Send emails and push notifications." — No dedup, no retries, no channel isolation.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (idempotency + retries)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 09

Design a Payment Gateway

Scenario

Staff Engineer at Stripe. Design the payment gateway.

Problem Statement

Process payments for merchants. No double-charge, no lost payment. PCI compliance. 10M txns/day.

Candidate Thinking Process

Candidate clarifies: idempotency via (merchant_id, key). ACID transactions. PCI compliance (tokenization). Webhook delivery with retries.

Step-by-Step Solution

Charge Service: BEGIN TXN; check idempotency; INSERT charge (PENDING); call card network; update status; COMMIT. Webhook Service delivers events with retries. Reconciliation with banks daily.

Architecture Diagram

Diagram

```
Merchant -> API Gw -> Charge Svc (TXN + idempotency) -> Card Network
Event -> Kafka -> Webhook Svc -> Merchant (with retries)
Reconciliation: daily batch vs bank settlement
```

Interviewer Feedback

Strong. Emphasized idempotency + ACID. PCI compliance via tokenization. Could improve: fraud detection integration.

Strong Answer Example

Strong Answer

"Charge Service: BEGIN TXN; check (merchant_id, idempotency_key) unique; INSERT charge (PENDING); call card network; update status; COMMIT. Idempotency prevents double-charge on retry. PCI compliance: tokenize cards (no raw data after first charge). Webhook Service delivers events to merchants with retries (max 5, exponential backoff). Reconciliation: daily batch matches our records with bank settlement files; flag discrepancies."

Weak Answer Example

Weak Answer

"Charge the card and save the transaction." — No idempotency, no PCI, no retries.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (idempotency + ACID)	19/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 10

Design a Multi-Region Active-Active DB

Scenario

Staff Engineer at a global SaaS. Design multi-region DB.

Problem Statement

Active-active across 3 regions. Low write latency globally. Survives region failure. Conflict resolution.

Candidate Thinking Process

Candidate clarifies: multi-leader (each region accepts writes). Async replication. Conflict resolution via CRDT or LWW. Anycast DNS for routing.

Step-by-Step Solution

Multi-leader DB per region (CockroachDB or custom). Async replication via log shipping. Conflict resolver merges (CRDT preferred). Anycast DNS routes users to nearest region. Failover: DNS update on region failure.

Architecture Diagram

Diagram

```
Region A (US) <-> Region B (EU) <-> Region C (Asia)
Each: App + DB (multi-leader)
Async replication log shipping
Conflict Resolver: CRDT or LWW
Anycast DNS -> nearest region
```

Interviewer Feedback

Strong. Discussed multi-leader + CRDT/LWW. Anycast DNS for routing + failover. Could improve: strong consistency globally (Spanner).

Strong Answer Example

> **Strong Answer**
>
> "Multi-leader DB per region (CockroachDB or custom). Each region accepts writes locally (low latency). Async replication via log shipping. Conflict resolver merges concurrent writes — CRDT for conflict-free merging, or LWW for simpler cases. Anycast DNS routes users to nearest region. Failover: on region failure, withdraw its Anycast route; traffic redirects to healthy regions within 60s. For strong consistency globally, use Spanner (TrueTime + atomic clocks) — but write latency increases to cross-region RTT."

Weak Answer Example

> **Weak Answer**
>
> "Replicate the database across regions." — No conflict resolution, no failover.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15

Dimension	Score
Capacity estimation	13/15
High-level design	18/20
Deep dive (CRDT + failover)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 11

Design YouTube Video Pipeline

Scenario

Senior SWE at YouTube. Design the video upload + transcoding pipeline.

Problem Statement

500 hours uploaded/minute. Transcode to 5 resolutions. Serve via CDN. Sub-2s stream startup.

Candidate Thinking Process

Candidate clarifies: resumable upload to S3, async transcode pipeline (Kafka + ffmpeg workers), CDN for delivery. Adaptive streaming (DASH).

Step-by-Step Solution

Upload: presigned S3 URL, resumable multipart. On complete: Kafka event. Transcode workers: ffmpeg to 5 resolutions (144p, 240p, 480p, 720p, 1080p) + thumbnails. CDN for delivery. DASH manifest for adaptive streaming.

Architecture Diagram

Diagram

```
Upload: Client -> S3 (presigned, multipart) -> Kafka event
Transcode: Workers -> ffmpeg -> 5 resolutions + thumbnails -> S3
Serve: CDN (DASH manifest) -> Client (adaptive)
```

Interviewer Feedback

Strong. Discussed transcode pipeline + CDN + adaptive streaming. Could improve: AV1 codec for new uploads.

Strong Answer Example

Strong Answer

"Upload: client requests presigned S3 URL, uploads via multipart (resumable). On complete: publish Kafka event. Transcode workers consume event, run ffmpeg to produce 5 resolutions (144p to 1080p) + thumbnails + manifest. Store outputs to S3. CDN pulls from S3 on first request, caches at edge. Client requests DASH manifest; CDN serves chunks adaptively based on bandwidth. Use AV1 for new uploads (30% smaller than H.264)."

Weak Answer Example

Weak Answer

"Upload to S3 and stream via CDN." — No transcode, no adaptive.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	19/20

Dimension	Score
Deep dive (transcode pipeline)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 12

Design Instagram Feed Ranking

Scenario

Senior SWE at Meta. Design the feed ranking system.

Problem Statement

Rank posts for 1B users. Personalized by engagement + recency + affinity. Sub-1s serving latency.

Candidate Thinking Process

Candidate clarifies: pre-compute candidates (top 1000), real-time rank top 100 via ML. Push-pull hybrid for fanout.

Step-by-Step Solution

Pre-compute: per-user candidate set in Redis (top 1000 from followees). Real-time: ML model ranks top 100 on read. Push for normal users; pull for celebrities.

Architecture Diagram

Diagram

```
Pre-compute: Fanout worker -> Redis (top 1000 per user)
Read: User -> Feed Svc -> Redis (candidates) -> ML rank top 100 -> Return
Seen-set prevents duplicates
```

Interviewer Feedback

Strong. Discussed pre-compute + real-time rank pattern. Could improve: seen-set for de-duplication.

Strong Answer Example

Strong Answer

"Hybrid: pre-compute candidates + real-time rank. On post publish, fanout worker pushes tweet_id to per-follower Redis lists (top 1000). For celebrities (>100K followers), skip push; pull at read time. Read path: fetch candidates from Redis, filter seen (via seen-set), rank top 100 via real-time ML model (engagement + recency + affinity signals), hydrate post objects, return. ML model retrained daily."

Weak Answer Example

Weak Answer

"Show recent posts from followed users." — No ranking, no scale.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (ML ranking)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 13

Design Zoom Video Conferencing

Scenario

Senior SWE at Zoom. Design the video conferencing system.

Problem Statement

Real-time video for 1M concurrent meetings. Sub-200ms latency. Recording.

Candidate Thinking Process

Candidate clarifies: SFU (Selective Forwarding Unit) for many-to-many; HLS for webinars (one-to-many). WebRTC for transport.

Step-by-Step Solution

SFU per meeting: receives each participant's stream, forwards to others. WebRTC for transport. Recording: media server captures mixed stream to S3. TURN/STUN for NAT traversal.

Architecture Diagram

Diagram

```
P1, P2, P3, P4 <-> SFU (WebRTC)
Recording: SFU -> Media Server -> S3 (HLS chunks)
TURN/STUN for NAT traversal
```

Interviewer Feedback

Strong. Discussed SFU vs P2P mesh. Recording as silent participant. Could improve: simulcast for adaptive quality.

Strong Answer Example

Strong Answer

"SFU (Selective Forwarding Unit) for meetings: receives each participant's stream once, forwards to others (scales better than P2P mesh). WebRTC for transport (sub-200ms). TURN/STUN servers for NAT traversal. Recording: media server joins as silent participant, captures mixed audio + active speaker video, streams to S3 in HLS chunks. For webinars (1-to-many), use HLS via CDN instead of SFU. Simulcast: client sends multiple resolutions; SFU forwards appropriate one per viewer."

Weak Answer Example

Weak Answer

"Use WebRTC for video calls." — No SFU, no scaling, no recording.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15

Dimension	Score
High-level design	18/20
Deep dive (SFU + recording)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 14

Design a Real-Time Analytics Platform

Scenario

Senior SWE at a SaaS. Design real-time analytics.

Problem Statement

Ingest 1M events/sec. Sub-second query latency. Real-time dashboards.

Candidate Thinking Process

Candidate clarifies: Kafka for ingestion, Flink for stream processing, Druid for serving (columnar, fast queries).

Step-by-Step Solution

Producers → Kafka → Flink (windowed aggregation) → Druid (columnar) → Dashboard API. Pre-aggregate common queries.

Architecture Diagram

Diagram

```
Producers -> Kafka -> Flink (windowed) -> Druid (columnar)
Dashboard API -> Druid -> sub-second queries
Pre-aggregate common queries
```

Interviewer Feedback

Strong. Discussed Kafka + Flink + Druid pipeline. Could improve: late arriving events.

Strong Answer Example

Strong Answer

"Kafka for ingestion (1M events/sec, partitioned). Flink for stream processing with windowed aggregation (1min, 1h, 1d rollups). Druid for serving (columnar storage, sub-second queries on billions of rows). Dashboard API queries Druid. Pre-aggregate common queries (top dashboards). Late-arriving events: Flink supports watermarks; allow 1min lateness for re-aggregation. Retention: 15 days hot in Druid, 1 year downsampled in S3."

Weak Answer Example

Weak Answer

"Use a database to store events and query them." — No streaming, no columnar, too slow.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	19/20
Deep dive (streaming pipeline)	18/20

Dimension	Score
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 15

Design a Distributed Lock Service

Scenario

Staff Engineer. Design a distributed lock service (ZooKeeper/etcd style).

Problem Statement

Exclusive locks on named resources. TTL. FIFO ordering. Survives node failure.

Candidate Thinking Process

Candidate clarifies: ephemeral sequential nodes (ZooKeeper-style). Lock = lowest sequence. Watch previous for release.

Step-by-Step Solution

etcd cluster (Raft). Acquire: create ephemeral sequential node under /locks/{resource}/. Get all children; if lowest, ACQUIRED. Else watch previous. TTL: etcd deletes ephemeral on session death.

Architecture Diagram

Diagram

```
/locks/{resource}/
  001 = Client 1 (holds lock)
  002 = Client 2 (watches 001)
On 001 release/death: 002 acquires
```

Interviewer Feedback

Strong. Discussed ephemeral sequential nodes + watch. Could improve: re-entrant locks.

Strong Answer Example

Strong Answer

"etcd cluster (Raft consensus). Acquire: create ephemeral sequential node under /locks/{resource}/. Get all children sorted by sequence. If your sequence is lowest, you hold the lock. Else, watch the previous node. When previous disappears (release or session death), re-check. TTL: etcd deletes ephemeral nodes when session dies (client crash → auto-release). FIFO ordering via sequence numbers. Re-entrant: track owner + count in node data."

Weak Answer Example

Weak Answer

"Use Redis SETNX." — No TTL, no FIFO, no fair locks.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (ephemeral nodes + watch)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 16

Design a Recommendation Engine

Scenario

Senior SWE at Amazon. Design product recommendations.

Problem Statement

Personalized recommendations for 100M users. 10M products. Sub-100ms serving.

Candidate Thinking Process

Candidate clarifies: hybrid (collaborative + content-based). Pre-compute top-100 per user hourly. Real-time bias. Vector DB for similarity.

Step-by-Step Solution

Hybrid: matrix factorization (collaborative) + product embeddings (content-based). Pre-compute top-100 per user hourly in Redis. Real-time: bias by recent activity. Vector DB for "similar items".

Architecture Diagram

Diagram

```
Events -> Kafka -> Spark (hourly) -> ML model -> top-100 per user -> Redis
Serving: User -> Rec Svc -> Redis (top-100) -> Real-time bias -> Filter seen
Similar items: Vector DB (Faiss/Milvus)
```

Interviewer Feedback

Strong. Discussed hybrid approach + pre-compute + real-time bias. Could improve: cold-start for new users/products.

Strong Answer Example

Strong Answer

"Hybrid: collaborative filtering (matrix factorization) + content-based (product embeddings). Pre-compute top-100 per user hourly via Spark + ML model; cache in Redis. Serving: fetch top-100, apply real-time bias (boost products in last-viewed category), filter seen + out-of-stock, rank. Similar items: vector DB (Faiss/Milvus) with product embeddings; ANN search for "customers also bought". Cold-start: popularity + demographics for new users; content features for new products."

Weak Answer Example

Weak Answer

"Use ML to recommend products." — No pipeline, no serving.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15

Dimension	Score
High-level design	18/20
Deep dive (ML pipeline)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 17

Design a Search Autocomplete

Scenario

Senior SWE at Google. Design search autocomplete.

Problem Statement

Sub-50ms suggestions as user types. 100K QPS. Trending + personalization.

Candidate Thinking Process

Candidate clarifies: trie for prefix matching. Per node, top 10 weighted queries. Trending boost (recent). Personalization (user history).

Step-by-Step Solution

Trie (in-memory). Per node, top 10 queries by weight. On request: traverse to prefix node, return top 10. Apply trending boost (recent queries) + personalization (user's past queries).

Architecture Diagram

Diagram

```
Trie:
  /i/p/h/ -> top: ["iphone", "iphone 15", "iphone case"]
Request: "iph" -> traverse to /i/p/h/ -> top 10
Apply trending + personalization boost
```

Interviewer Feedback

Strong. Discussed trie + weighted suggestions. Could improve: typo tolerance.

Strong Answer Example

> **Strong Answer**
>
> "Trie (prefix tree) in-memory. Per node, store top 10 queries by weight (frequency × recency). On request: traverse trie to prefix node, fetch top 10. Apply trending boost (queries trending in last hour) + personalization (boost queries user has used before). Trie rebuilt hourly from query counts; incremental updates for trending. Typo tolerance: edit distance or n-gram model."

Weak Answer Example

> **Weak Answer**
>
> "Search a database of queries." — No trie, too slow.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (trie + ranking)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 18

Design a Fraud Detection System

Scenario

Staff Engineer at Stripe. Design fraud detection.

Problem Statement

Score transactions in < 100ms. Block high-risk. Adapt to new fraud patterns.

Candidate Thinking Process

Candidate clarifies: hybrid (rules + ML). Real-time features (last 1h txn count). Online learning via feedback.

Step-by-Step Solution

Rules engine (fast, deterministic) + ML model (XGBoost, adaptive). Feature store: Redis (real-time) + Cassandra (batch). Online learning via feedback loop.

Architecture Diagram

Diagram

```
Txn -> Fraud Svc -> Rules (fast) + ML (XGBoost)
Features: Redis (real-time) + Cassandra (batch)
Feedback -> Kafka -> online learning
```

Interviewer Feedback

Strong. Discussed hybrid + feature store + online learning. Could improve: graph-based fraud detection (rings).

Strong Answer Example

Strong Answer

"Hybrid: rules engine (fast, deterministic) + ML model (XGBoost, adaptive). On txn: fetch real-time features from Redis (txn count last 1h, avg amount), batch features from Cassandra (user historical fraud rate). Apply rules (block if amount > 10x user avg). Apply ML model. Combine scores; return APPROVE/REVIEW/BLOCK + reasons. Feedback (was_fraud) streams to Kafka; online learning updates model. Graph-based: detect fraud rings via shared attributes (IP, device, email)."

Weak Answer Example

Weak Answer

"Use ML to detect fraud." — No rules, no features, no feedback.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (hybrid + features)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 19

Design a Multi-Tenant SaaS

Scenario

Staff Engineer at a B2B SaaS. Design multi-tenant architecture.

Problem Statement

10K tenants. Per-tenant data isolation. Noisy neighbor mitigation. Per-tier resource limits.

Candidate Thinking Process

Candidate clarifies: 3 isolation models (shared DB, per-tenant schema, per-tenant DB). Pick by tenant size. Tenant-aware routing.

Step-by-Step Solution

3 isolation models: shared DB + tenant_id (small tenants), per-tenant schema (medium), per-tenant DB (enterprise). Tenant Router picks model based on tier. Per-tenant rate limits + quotas.

Architecture Diagram

Diagram

```
Tenant Router: tenant_id -> isolation level
Small: shared DB + tenant_id column
Medium: per-tenant schema
Enterprise: per-tenant DB
Per-tenant rate limits + quotas
```

Interviewer Feedback

Strong. Discussed 3 isolation models + tenant router. Could improve: migration between tiers.

Strong Answer Example

Strong Answer

"Three isolation models based on tenant tier: (1) Shared DB + tenant_id column for small tenants (cheap, dense), (2) Per-tenant schema for medium tenants (isolation without DB overhead), (3) Per-tenant DB for enterprise (full isolation, custom backup). Tenant Router extracts tenant_id from JWT, picks isolation model. Per-tenant rate limits + storage quotas mitigate noisy neighbor. Migration between tiers: background job copies data, cutover with dual-write."

Weak Answer Example

Weak Answer

"Add a tenant_id to every table." — No isolation, noisy neighbor.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15
High-level design	18/20

Dimension	Score
Deep dive (isolation models)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15

MOCK INTERVIEW 20

Design a Global CDN

Scenario

Staff Engineer at Cloudflare. Design a global CDN.

Problem Statement

500 PoPs globally. 1Tbps egress. Anycast DNS routing. Cache invalidation.

Candidate Thinking Process

Candidate clarifies: Anycast DNS routes to nearest PoP. Multi-tier cache (PoP → origin shield → origin). Purge via pub/sub.

Step-by-Step Solution

Anycast DNS (BGP) routes users to nearest PoP. PoP caches objects on first fetch. Origin shield dedupes origin fetches. Purge: central API enqueues; propagates to all PoPs via pub/sub.

Architecture Diagram

Diagram

```
User -> Anycast DNS -> Nearest PoP
PoP cache hit -> serve (15ms)
PoP miss -> Origin Shield (regional parent) -> Origin
Purge: Central API -> pub/sub -> all PoPs (< 30s)
```

Interviewer Feedback

Strong. Discussed Anycast + origin shield + purge. Could improve: edge compute (Cloudflare Workers).

Strong Answer Example

> **Strong Answer**
>
> "Anycast DNS (BGP-based) routes users to nearest PoP. PoP caches objects on first fetch (in-memory + SSD tier). On cache miss: fetch from Origin Shield (regional parent PoP) which dedupes origin fetches. Purge: customer calls central API; enqueues purge to all PoPs via pub/sub (NATS/Kafka); propagates globally in < 30s. Multi-tier cache: PoP → Shield → Origin maximizes hit ratio. Edge compute (Cloudflare Workers) runs custom logic at PoP. DDoS protection: rate limit + scrubbing at edge."

Weak Answer Example

> **Weak Answer**
>
> "Cache content at edge locations." — No Anycast, no origin shield, no purge.

Evaluation Rubric

Dimension	Score
Requirements clarification	14/15
Capacity estimation	13/15

Dimension	Score
High-level design	18/20
Deep dive (Anycast + caching)	18/20
Trade-offs	13/15
Bottlenecks + monitoring	11/15