

System Design — EknathaLabs Interview Guide

Platform, SRE & DevOps interview guide

Questions · spoken answers included

Online (always latest) interviewhub.eknathalabs.com/system_design_interview_guide.html

Printed 2026-10-21

Free to use and share for personal study. Answers are written for interview practice, so check command flags and versions against the official documentation before running anything in production.

1. **Walk me through your framework for approaching any system design question.**

fundamentals

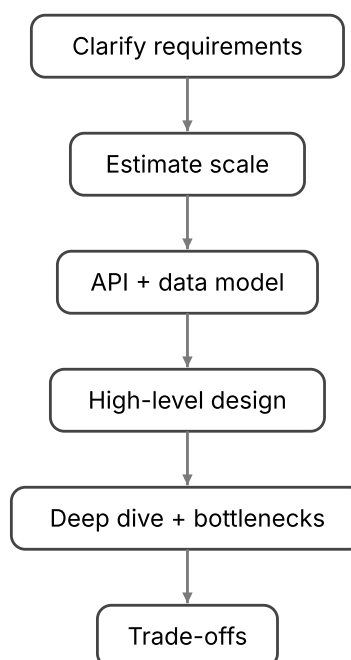
🗣 Spoken answer · ~88 s

I use a consistent framework so I don't miss anything and so the interviewer can follow my thinking. First, clarify requirements: functional ones, meaning what the system must do and the core use cases, and non-functional ones, such as scale, latency, availability, consistency, durability, and cost. I confirm scope and what's out of scope. Second, estimate scale: users, requests per second, data volume, read-to-write ratio, and storage growth, with a quick back-of-envelope calculation, because numbers drive design choices. Third, define the API and the core data model: main entities, key operations, and access patterns. Fourth, sketch a high-level architecture: clients, load balancer, stateless services, data stores, caches, queues, and external dependencies, and walk through the main flows end to end. Fifth, deep dive into the components that matter most or that the interviewer picks: data partitioning, replication, indexing, caching strategy, and consistency, discussing alternatives and trade-offs explicitly. Sixth, address reliability and scale: bottlenecks, single points of failure, failure modes, back pressure, multi-region, and capacity planning. Seventh, cover operations: monitoring, SLOs, alerting, deployment, security, and cost. Throughout, I state assumptions, think aloud, justify each choice by tying it to requirements, and keep checking in with the interviewer. I start simple and evolve the design as requirements tighten, instead of over-engineering upfront.

Likely follow-ups

- Why estimate scale before drawing the architecture?
- How do you decide where to deep dive?

🗺 Interview framework



A consistent framework prevents you from going blank and shows structured thinking. Use **RADIO**:
Requirements → Architecture → Data model → Interface (APIs) → Optimisations.

STEP-BY-STEP

1. Requirements (5 min)
 - Functional: what does the system do?
 - Non-functional: scale, latency, durability, consistency
 - Out of scope: what are we NOT building?
2. Capacity estimation (3 min)
 - DAU, RPS, storage per year, bandwidth
 - Back-of-envelope: 1M DAU × 10 req/day = ~115 RPS
3. High-level architecture (10 min)
 - Draw boxes: clients, LB, services, DB, cache, queue
 - Identify read vs write paths separately
4. Deep dives (15 min)
 - Pick 2–3 hard problems, go deep
 - Show trade-offs at each decision
5. Bottlenecks & failure modes (5 min)
 - SPOF identification
 - Scaling levers: horizontal, caching, sharding

Interview tip: Drive the conversation. Don't wait for prompts. Say "I'll now focus on the write path since that's the harder problem — is that OK?" Shows Principal-level ownership.



Official docs

- [Google SRE Book: Effective Troubleshooting](#)

2. How do you estimate scale requirements? Walk through a back-of-envelope calculation.

fundamentals

Spoken answer · ~83 s

Back-of-envelope estimation turns vague requirements into numbers that drive design. I start with users and usage: say 100 million daily active users, each making 10 requests a day, giving a billion requests a day. Dividing by roughly 100,000 seconds per day, which is about 86,400 rounded, gives around 10,000 requests per second on average; peak is usually two to five times higher, so plan for 30,000 to 50,000. Then the read-write ratio, for example 10 to 1, splits that into reads and writes. For storage: if each record is 1 kilobyte and we create 10 million a day, that's 10 gigabytes a day, about 3.6 terabytes a year, and with replication factor of three, around 11 terabytes; multiply by years of retention. For bandwidth: requests per second times average payload, for instance 10,000 times 10 kilobytes is 100 megabytes per second. For memory and caching: apply the 80/20 rule, caching the hottest 20 percent of daily data. I use powers of two and round aggressively: a kilobyte is a thousand bytes, a million kilobytes is a gigabyte. Useful latencies: memory about 100 nanoseconds, SSD around 100 microseconds, a cross-datacentre round trip around 500 microseconds, a cross-continent one around 150 milliseconds.

Likely follow-ups

- How do you convert daily requests to requests per second?
- Why plan for peak rather than average?

Interviewers want to see whether you can reason about numbers, not whether you memorise them. Use round numbers and narrate your logic.

KEY CONSTANTS TO MEMORISE

```
1 million requests/day ≈ 12 RPS
1 billion requests/day ≈ 11,600 RPS ≈ ~12K RPS
1 KB × 1M events/day ≈ 1 GB/day ≈ 365 GB/year
SSD sequential read: ~500 MB/s
Network (1GbE): ~125 MB/s
RAM access: ~100 ns | SSD: ~100 μs | HDD: ~10 ms
```

EXAMPLE: TWITTER-SCALE TIMELINE SERVICE

```
DAU: 300M users, 20% active daily = 60M active
Avg tweets/day: 60M users × 5 tweets = 300M tweets/day
Write RPS: 300M / 86400 ≈ 3,500 writes/sec
Read RPS: 10× write ratio ≈ 35,000 reads/sec
Storage: 300M tweets × 300 bytes = 90 GB/day → ~33 TB/year (text only)
```

Interview tip: After estimation, say "Given 35K read RPS, a single DB won't cut it — so we'll need read replicas and a cache layer." Estimation should drive architectural decisions, not be an isolated exercise.

Official docs

- [Latency numbers every programmer should know](#)

3. Explain CAP theorem and how it affects real-world database choices.

tradeoffs

Spoken answer · ~85 s

The CAP theorem says that in a distributed data store, when a network partition occurs, you must choose between consistency and availability; you cannot have both. Consistency here means every read sees the most recent write, as if there were a single copy; availability means every request to a non-failing node gets a response. Partition tolerance isn't really optional, because networks do fail, so the practical choice is between CP and AP behaviour during a partition. CP systems, like etcd, ZooKeeper, HBase, or Spanner, refuse or delay requests on the minority side to avoid returning stale or conflicting data, so they suit coordination, configuration, and financial ledgers. AP systems, like Cassandra, DynamoDB in its default mode, and Riak, keep serving requests on both sides and reconcile later, accepting eventual consistency, so they suit shopping carts, social feeds, and metrics, where availability matters more than perfectly fresh reads. When there's no partition, the PACELC extension notes a further trade-off between latency and consistency. In practice the choice is tunable per operation: quorum reads and writes, with $R + W > N$, give strong consistency at the cost of latency and availability, while lower quorums favour availability. Also, consistency models range between strong, causal, and eventual.

Likely follow-ups

- What does PACELC add?
- How do quorums let you tune the trade-off?

CAP during a partition

CP: consistent	AP: available
Rejects writes on minority side	Both sides keep serving, reconcile later
Examples: etcd, ZooKeeper, HBase	Examples: Cassandra, DynamoDB, Riak

CAP: a distributed system can guarantee at most two of Consistency, Availability, Partition Tolerance. Since network partitions always occur in practice, the real choice is **CP vs AP**.

CP systems (consistent when partitioned – may reject writes):

PostgreSQL (single primary), HBase, Zookeeper, etcd

→ Use when: financial data, inventory counts, leader election

AP systems (available when partitioned – may serve stale data):

Cassandra, DynamoDB, CouchDB, DNS

→ Use when: social feeds, metrics, user preferences, caches

CA systems (no partition tolerance – theoretical only):

Single-node RDBMS – not distributed

Principal answer: CAP is binary and too coarse for real decisions. Use **PACELC** (Partition: Availability vs Consistency; Else: Latency vs Consistency) — it captures the latency trade-off that matters even during normal operation. Cassandra is PA/EL; PostgreSQL is PC/EC.

Official docs

- [Jepsen: Consistency models](#)

4. When do you choose SQL vs NoSQL? Give concrete decision criteria.

tradeoffs

Spoken answer · ~83 s

I choose based on data model, access patterns, consistency needs, and scale, not fashion. Choose SQL when data is relational with well-defined structure and many relationships; you need multi-row ACID transactions, such as payments or inventory; you need flexible ad hoc queries, joins, and aggregations; the schema is relatively stable; and the scale fits a well-tuned primary with read replicas, which covers most workloads, often into terabytes and tens of thousands of requests per second. Choose NoSQL when a specific need outweighs relational strengths. Key-value stores like DynamoDB or Redis for massive throughput, simple lookups by key, and predictable low latency. Document stores like MongoDB for hierarchical, evolving data read as a unit. Wide-column stores like Cassandra or Bigtable for enormous write volumes, time-series or event data, and horizontal scaling across regions. Graph databases for deeply connected data. Search engines for full-text. Key criteria: Do you need joins and transactions across entities? What are the read and write patterns and are they known upfront? What is the scale in data size and throughput, and does it need horizontal sharding? What consistency and availability does the business require? How flexible is the schema? What does the team know how to operate?

Likely follow-ups

- When does a single SQL primary stop being enough?
- What complexity does NoSQL move into the application?

Neither is universally better. Choose based on data model, access patterns, and operational requirements.

Choose SQL (PostgreSQL, MySQL, Aurora) when:

- ✓ Complex joins between entities
- ✓ ACID transactions required (payments, bookings)
- ✓ Schema is stable and well-defined
- ✓ Reporting / ad-hoc analytics
- ✓ Team has strong SQL expertise

Choose NoSQL when:

- ✓ Document model (MongoDB): nested JSON, flexible schema
- ✓ Key-value (Redis, DynamoDB): simple lookups at high RPS
- ✓ Wide-column (Cassandra): time-series, write-heavy, multi-region
- ✓ Graph (Neo4j): social graphs, recommendation engines
- ✓ Search (Elasticsearch): full-text, faceted search

Interview tip: Say "I'd choose PostgreSQL as the source of truth for user accounts and payments — it gives us transactions and referential integrity. For the activity feed (high write RPS, time-ordered, no joins needed), I'd use Cassandra." Showing you use both is better than picking a side.



Official docs

- [Azure: Choose a data store \(relational vs NoSQL\)](#)

5. What is consistent hashing and why do distributed systems use it?

fundamentals

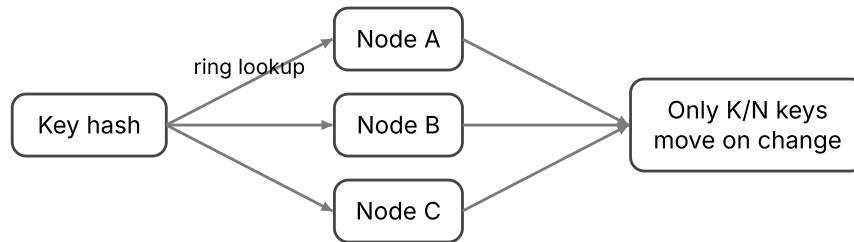
 Spoken answer · ~87 s

Consistent hashing is a way to distribute keys across a changing set of nodes so that adding or removing a node moves only a small fraction of the keys. With the naive approach, $\text{hash}(\text{key}) \bmod N$, changing N remaps almost every key, causing a massive cache miss storm or data migration. In consistent hashing, both nodes and keys are hashed onto a circular space, the ring. Each key is assigned to the first node found moving clockwise from its position. When a node is added, it takes over only the keys between itself and its predecessor; when a node is removed, its keys pass to the next node clockwise; on average only about K divided by N keys move. A weakness is uneven distribution: with few nodes, arcs vary in size, and a failed node dumps all its load on one neighbour. The fix is virtual nodes: each physical node is placed at many points on the ring, often 100 to 200, which smooths out the load, lets heterogeneous machines take proportionally more, and spreads a failed node's keys across many peers. For replication, a key is stored on the next several distinct nodes clockwise. Used by Dynamo, Cassandra, memcached clients, CDNs, and load balancers for sticky routing.

Likely follow-ups

- Why are virtual nodes needed?
- What fraction of keys moves when a node joins?

🚩 Consistent hashing ring



Consistent hashing maps keys and nodes to a ring. When a node is added/removed, only K/N keys need remapping (vs all keys in naive modulo hashing). Critical for cache clusters, distributed DBs, and load balancers.

Ring: 0 — 90 — 180 — 270 — 360
Node A Node B Node C Node D

Key "user:123" hashes to position 95 → assigned to Node B (next node clockwise)
Remove Node B → keys 90–180 shift to Node C only. All others unaffected.

VIRTUAL NODES (VNODES)

Each physical node is mapped to multiple positions on the ring. Improves load distribution. Used by Cassandra (256 vnodes per node), DynamoDB, Riak.

Without vnodes: 3 nodes, each owns 1/3 of ring
With vnodes: 3 nodes × 256 = 768 positions → much more even spread

Interview tip: Mention that consistent hashing is used in Redis Cluster, Cassandra, Memcached, and CDN edge routing. Examiners love when you connect theory to real systems.

📖 Official docs

- [Microsoft: Sharding_pattern](#)

6. Design a URL shortener (like bit.ly). Handle 100M URLs, 10B redirects/day.

scalability

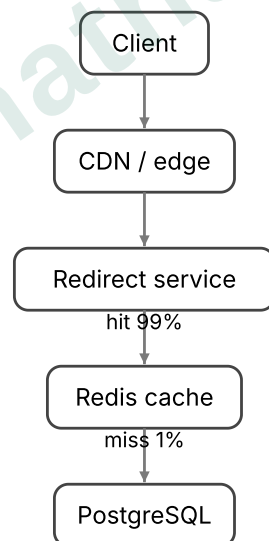
🗣 Spoken answer · ~75 s

Requirements: shorten a long URL to a short code, redirect quickly, with 100 million stored URLs and 10 billion redirects a day, which is about 115,000 redirects per second on average, with peaks perhaps 300,000. It's extremely read-heavy, around 100 to 1 or more. API: POST /shorten with the long URL and an optional custom alias or expiry returns a short URL; GET /{code} returns a 301 or 302 redirect. A 302 lets you count clicks, and 301 allows browser caching, so choose by the need for analytics. Short code generation: base62 encoding gives seven characters for about 3.5 trillion combinations. Options: a counter or a distributed ID generator, such as ranges allocated to servers from a coordinating service, encoded in base62, which guarantees uniqueness with no collisions; or hashing the URL and taking a prefix, which needs collision handling. Storage: a key-value store, such as DynamoDB or Cassandra, keyed by code, storing the long URL, owner, creation time, and expiry; 100 million records at around 500 bytes is only 50 gigabytes, so size isn't the issue, throughput is.

Likely follow-ups

- Why choose 302 over 301 for redirects?
- How do you guarantee unique short codes without collisions?

🗣 URL shortener request path



Classic design problem. Key challenges: unique ID generation, low-latency redirects, analytics.

```
Client → CDN / Edge Cache
      ↓ (cache miss)
LB → Redirect Service (stateless, horizontal scale)
      ↓
Cache (Redis) ← hot URLs served here (95% hit rate)
      ↓ (cache miss)
DB (PostgreSQL) – short_code → long_url, created_at, user_id
```

↓
Analytics Queue (Kafka) → Click Aggregator → Analytics DB

ID GENERATION — BASE62 ENCODING

```
# 7 chars × 62 possibilities = 62^7 = 3.5 trillion unique URLs
# Generate unique int ID (Snowflake / DB sequence), encode to Base62
def encode_base62(n):
    chars = "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
    result = []
    while n:
        result.append(chars[n % 62])
        n //= 62
    return ''.join(reversed(result))
```

SCALE: 10B REDIRECTS/DAY = ~115K RPS

Redis cache: 99% hit rate → 115K RPS handled in memory (~1μs/lookup)
Cache miss (1%): 1,150 RPS to PostgreSQL — easily handled by single read replica
Write rate: 100M URLs / (365 days × 86400 sec) ≈ 3 writes/sec — trivial

Principal answer: The redirect path must be P99 < 10ms. Serve from CDN edge where possible — cache 301 Permanent vs 302 Temporary based on whether the URL can change. Use 302 for analytics (forces redirect through your service every time). Use 301 for permanent links (browser caches, fewer requests).

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

7. Design a rate limiter. Support per-user, per-IP, and per-API-key limits.

scalability

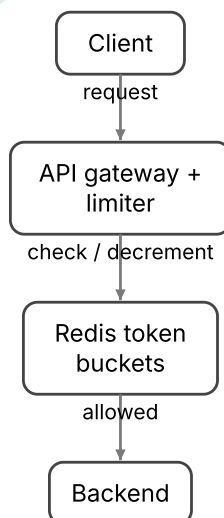
Spoken answer · ~87 s

Requirements: limit the requests per client, identified by user, IP, or API key, with configurable limits per rule, low added latency, and accurate enough enforcement across many servers. Algorithms: a token bucket allows bursts up to a bucket size while enforcing an average refill rate, widely used; a leaky bucket smooths output; fixed window counters are simple but allow double bursts at window edges; sliding window log is accurate but memory hungry; sliding window counter approximates it cheaply. Architecture: place the limiter at the API gateway or as middleware in front of services. Rules are stored in a configuration service and cached locally, each rule defining the dimension, such as user, IP, or key, the limit, and the window. For distributed enforcement, keep counters in a shared fast store like Redis, using atomic operations, such as INCR with expiry or a Lua script implementing the token bucket, to avoid race conditions. Keys look like `ratelimit:{apikey}:{window}`. To reduce latency and load, combine local in-memory counters with periodic synchronisation, accepting slight overshoot, or shard counters by key across a Redis cluster. When exceeded, return HTTP 429 with `Retry-After` and `X-RateLimit` headers. Consider failure behaviour: if Redis is unavailable, fail open for availability or fail closed for protection, depending on the endpoint.

Likely follow-ups

- How does a token bucket allow bursts?
- Should the limiter fail open or closed?

Rate limiter



Rate limiting prevents abuse, enforces SLAs, and protects downstream services. Key question: where to enforce (edge, API gateway, service)?

ALGORITHM CHOICES

Token Bucket: allow bursting, refill at constant rate
→ Best for: API gateways, per-user limits

Fixed Window: simple, cheap – but allows 2× burst at window boundary
Sliding Window Log: precise, high memory (store every request timestamp)
Sliding Window Counter: interpolate between two fixed windows – good balance

IMPLEMENTATION: REDIS SLIDING WINDOW

```
# Redis Lua script (atomic – no race condition)
local key = KEYS[1]          -- "rl:user:123:endpoint:/api/search"
local limit = tonumber(ARGV[1]) -- 100
local window = tonumber(ARGV[2]) -- 60 (seconds)
local now = tonumber(ARGV[3])

redis.call('ZREMRANGEBYSCORE', key, 0, now - window * 1000)
local count = redis.call('ZCARD', key)
if count < limit then
    redis.call('ZADD', key, now, now)
    redis.call('EXPIRE', key, window)
    return 1 -- allowed
end
return 0 -- rejected → HTTP 429
```

Multi-tier rate limiting:

- Layer 1: CDN/Edge (IP-based, coarse – blocks DDoS)
- Layer 2: API Gateway (per-API-key – SLA enforcement)
- Layer 3: Service level (per-user, per-resource – business logic)

Interview tip: Mention distributed rate limiting across multiple API gateway pods — Redis is the central store. Add response headers: X-RateLimit-Limit, X-RateLimit-Remaining, Retry-After. These are required for good API UX.

Official docs

- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [Stripe Engineering: Scaling your API with rate limiters](#)

8. Design a notification system (push, email, SMS) for 500M users.

scalability

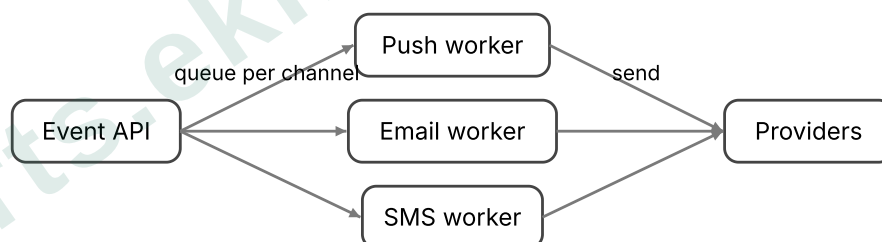
🗣 Spoken answer · ~85 s

Requirements: send push, email, and SMS notifications to 500 million users, supporting transactional messages that must be reliable and timely, and marketing broadcasts that are high volume, with user preferences, deduplication, retries, and delivery tracking. Architecture: producers, such as services and campaign tools, call a notification API, which validates, applies user preferences and opt-outs, rate limits, and templates, then publishes events to a message queue, such as Kafka, partitioned by user, with separate topics or priorities for each channel and for urgent versus bulk. Channel workers consume from the queues, render the content using templates and localisation, and call third-party providers: APNs and FCM for push, SES or SendGrid for email, Twilio for SMS. Store device tokens and user contact details and preferences in a database. Reliability: at-least-once delivery with idempotency keys to deduplicate, retries with exponential backoff and a dead-letter queue, provider failover, and circuit breakers. Throttle per provider limits and per user to avoid spamming. Scale: horizontally scaled workers, bulk jobs fanned out through batch pipelines that read audience segments, and priority queues so a marketing blast doesn't delay a password reset. Track delivery status through callbacks and webhooks into an analytics store, and handle unsubscribe and bounces, removing invalid tokens.

Likely follow-ups

- Why separate priority queues for transactional and marketing messages?
- How do you avoid duplicate notifications?

🗣 Notification fan-out



Notification systems are deceptively hard: delivery guarantees, user preferences, throttling, channel failover, and deduplication.

Producers (events) → Kafka Topics (by channel type)

↓

Notification Service

- └ Preference filter (user opted out? → discard)
- └ Dedup check (Redis TTL key on notification_id)
- └ Priority queue (critical > normal > marketing)
- └ Channel router
 - └ Push → APNs (iOS) / FCM (Android)
 - └ Email → SendGrid / SES → SMTP
 - └ SMS → Twilio / SNS

KEY DESIGN DECISIONS

1. At-least-once delivery via Kafka (consumer group, manual offset commit)
2. Idempotency: notification_id deduplicated in Redis (TTL = 24h)
3. Retry with exponential backoff: 1s → 2s → 4s → 16s → DLQ
4. User preference DB: per-channel opt-out, quiet hours, frequency caps
5. Rate limiting: max 5 marketing emails/user/day regardless of events

Principal answer: Separate the *event bus* (Kafka) from the *sending workers*. This lets you: replay failed notifications, add new channels without touching producers, and independently scale push vs email workers (push is CPU-light, email involves template rendering). Store every notification in a NoSQL log (DynamoDB) for inbox features.

Official docs

- [Microsoft: Competing Consumers pattern](#)

9. Design a distributed job queue / task scheduler. Handle millions of jobs/day.

scalability

Spoken answer · ~83 s

Requirements: accept jobs, run them reliably at millions per day, with scheduling, priorities, retries, and visibility. Components: an API for submitting jobs with payload, priority, and schedule; a durable job store, such as a database or Kafka-backed log, holding state; a scheduler that moves due jobs into ready queues; queues partitioned by priority or type; and a pool of stateless workers that pull jobs, execute them, and report results. Workers claim a job with a lease or visibility timeout: if a worker crashes or doesn't heartbeat, the lease expires and the job is retried by another worker, giving at-least-once execution; so tasks must be idempotent, or use idempotency keys. Retries use exponential backoff with jitter, a maximum attempt count, and a dead-letter queue for poison jobs. For delayed and recurring jobs, a time-bucketed index, such as sorted sets keyed by run time or a database table indexed on next-run time, is polled by the scheduler, which is itself sharded or uses leader election per shard to avoid double scheduling. Scaling: partition by job ID or tenant, autoscale workers on queue depth, apply rate limits and fair sharing so one tenant can't starve others, and concurrency limits per job type.

Likely follow-ups

- How do leases prevent lost jobs?
- Why must job handlers be idempotent?

Job queue lifecycle



Core requirements: reliable delivery, at-least-once execution, delayed jobs, priority, retries, observability.

Producers → Job Queue (Redis / Kafka / SQS)



Dispatcher / Broker



Worker Pool (horizontal scale)



└─ Success → mark complete in Job Store

└─ Failure → retry with backoff → DLQ after N attempts

DELAYED JOBS (SCHEDULED TASKS)

```
# Redis sorted set – score = execution timestamp
ZADD delayed_jobs 1735689600 "job:abc123" # execute at Unix time

# Poller (every second):
jobs = ZRANGEBYSCORE delayed_jobs 0 NOW LIMIT 0 100
for job in jobs:
    push to ready_queue
    ZREM delayed_jobs job
```

AT-LEAST-ONCE VS EXACTLY-ONCE

At-least-once: Ack only after successful processing

Idempotency key on the job consumer side

Exactly-once: Impossible in distributed systems without 2PC overhead

→ Design jobs to be idempotent instead

Interview tip: Celery + Redis/RabbitMQ is fine for most workloads. For 10M+ jobs/day with complex dependencies: Temporal (workflow orchestration). Mention visibility timeout — job becomes visible again if worker crashes without acking. Must be > expected max job duration.



Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

10. Design a web crawler at Google scale. Crawl 1 billion pages per day.

scalability

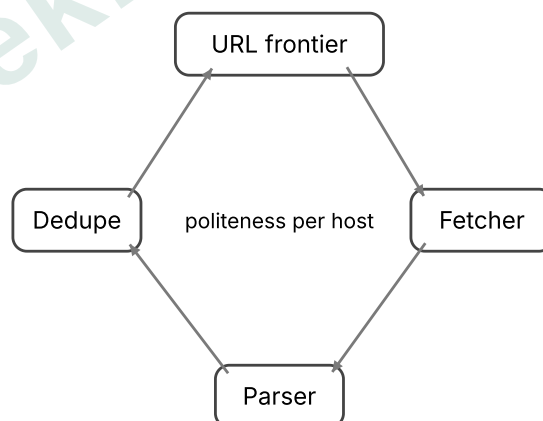
Spoken answer · ~88 s

Requirements: crawl one billion pages a day, which is about 11,500 pages per second, politely, without duplicates, and keep content fresh. Components. A URL frontier: the prioritised queue of URLs to fetch, with priority by importance and freshness, and politeness by per-host queues so you don't hammer a site, enforcing delays and robots.txt rules. Fetchers: a large distributed fleet of workers, each pulling URLs from the frontier, resolving DNS through a caching resolver, since DNS is a bottleneck, and downloading pages with timeouts, connection reuse, and limits on size. Parser: extracts text, metadata, and outgoing links, and normalises URLs. Duplicate detection: a URL-seen check using a Bloom filter backed by a distributed store, and content-level deduplication via hashes or SimHash for near-duplicates. Storage: raw pages in a distributed object store or file system, such as HDFS or GCS, and metadata in a scalable database; extracted links feed back into the frontier. Scaling and partitioning: shard the frontier and fetchers by host hash, so one worker owns each host's politeness state. Handle traps like infinite calendars with depth limits, URL pattern detection, and page budgets per domain. Recrawl scheduling uses change frequency estimates. Handle JavaScript rendering for dynamic pages with a headless-browser pool for a subset. Monitor throughput, error rates, and freshness.

Likely follow-ups

- Why shard the frontier by host?
- How do you detect near-duplicate pages?

Crawler loop



Hard problems: politeness (don't hammer a single domain), deduplication, dynamic content, distributed coordination.

Seed URLs → URL Frontier (priority queue)

↓

Dispatcher → Fetcher Workers (1000s, distributed)

↓

DNS Resolver Cache

↓

HTML Parser

↓
Content Seen? → Yes: discard
↓ No
Content Store (GCS / S3)
↓
URL Extractor → URL Normaliser → URL Seen Filter → URL Frontier

DEDUPLICATION

```
# URL dedup: Bloom filter (fast, probabilistic, no false negatives)
bloom.add(url)
if bloom.check(url): skip # 1% false positive rate acceptable

# Content dedup: SimHash / MinHash to detect near-duplicate pages
# Exact dedup: SHA-256 of content stored in distributed cache
```

POLITENESS

```
# Per-domain queue + delay
# Respect robots.txt – cache it per domain (1h TTL)
# Back-pressure: rate limit to 1 req/sec per domain by default
# Crawl-Delay directive in robots.txt: honour it
```

Principal answer: URL Frontier is a priority queue — prioritise high-PageRank domains. Use a multi-level queue: fresh (recrawl changed pages quickly) vs stale. DNS is the #1 bottleneck at scale — maintain a local DNS cache with thousands of entries. 1 DNS lookup per URL at 1B pages/day = ~11K DNS queries/sec.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

11. **Design a system that achieves 99.99% availability. What does that mean operationally?**

reliability

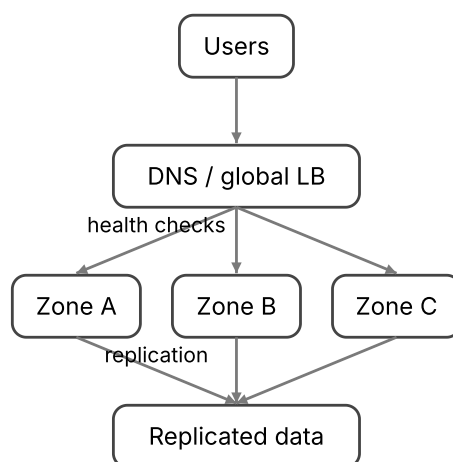
🗣 Spoken answer · ~90 s

99.99 percent availability, four nines, allows about 52 minutes of downtime a year, or roughly 4.3 minutes a month. Operationally that's very demanding: a single slow human-led incident can use the whole annual budget, so detection, failover, and recovery must be automated and take seconds to minutes. Design principles. Eliminate single points of failure at every layer: redundant load balancers, multiple stateless application instances across availability zones, and replicated data stores with automatic failover. Multi-AZ is the baseline; multi-region for regional failure tolerance, with active-active or active-passive and tested failover. Remember availability multiplies across serial dependencies: three components each at 99.99 percent give about 99.97, so reduce critical-path dependencies and make non-critical ones degrade gracefully. Use health checks, automated failover, and self-healing orchestration. Protect against overload with rate limiting, load shedding, timeouts, retries with backoff and jitter, circuit breakers, and bulkheads. Deploy safely: canary and blue-green releases with automatic rollback, feature flags, backward-compatible migrations, because changes cause most outages. Observability: SLOs, error budgets, alerts on burn rate, tracing. Process: on-call, runbooks, blameless postmortems, game days and chaos testing, and capacity planning. Data durability through backups and replication, with tested restores. Also avoid maintenance windows; use rolling upgrades. Finally, agree what counts as downtime: measure from the user's perspective, excluding planned exclusions only if contractually defined.

Likely follow-ups

- How do serial dependencies affect overall availability?
- Why do deployments cause most outages?

🚧 Availability by redundancy



99.99% = 52 minutes downtime/year. 99.999% = 5 minutes/year. Every architectural decision must be justified against this budget.

Availability budget breakdown (99.99% = 52 min/year):

Planned maintenance: <10 min (rolling deploys, zero-downtime)

Incident response: <42 min (~4 incidents × 10 min MTTR each)

MTTR must be < 10 minutes → requires:

- Automated alerting (PagerDuty, < 2 min to page)
- Runbooks for top 10 failure modes
- Feature flags for instant rollback
- Health checks + auto-restart (Kubernetes liveness probe)

ARCHITECTURAL REQUIREMENTS FOR 99.99%

1. No single points of failure
 - Multi-AZ deployment (lose 1 AZ, still serving)
 - N+2 redundancy for stateless services
2. Graceful degradation
 - Circuit breakers (fail fast, don't cascade)
 - Fallback: serve stale cache instead of error
3. Load shedding
 - Reject low-priority requests under overload
 - Rate limiting protects against thundering herd
4. Chaos engineering
 - Intentionally kill instances to verify recovery
 - GameDay exercises quarterly

Interview tip: Availability = $MTBF / (MTBF + MTTR)$. To improve: increase MTBF (fewer failures) OR decrease MTTR (faster recovery). Recovery automation gives 10× more leverage than trying to make systems never fail.



Official docs

- [Google SRE Book: Embracing Risk \(capacity & availability math\)](#)

12. Design a distributed cache system. Handle 1M RPS with sub-millisecond latency.

reliability

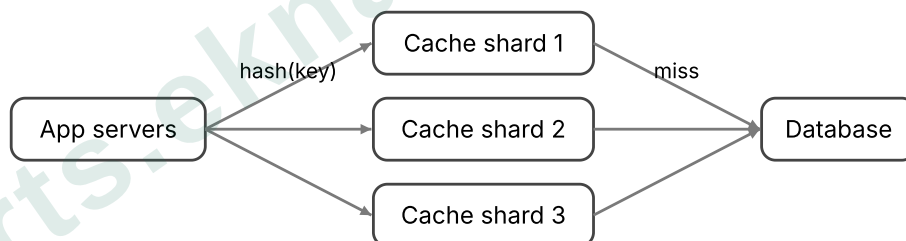
Spoken answer · ~75 s

Requirements: a distributed in-memory cache handling one million requests per second at sub-millisecond latency. A single node can serve perhaps 100,000 or more simple operations per second, so we need a cluster of ten or more nodes plus headroom. Architecture: shard the key space across nodes using consistent hashing with virtual nodes, or Redis Cluster's hash slots, so adding nodes moves little data; clients use a smart client library or a proxy such as Twemproxy or Envoy to route requests directly to the right shard with minimal hops, keeping connections pooled and using pipelining. Replication: each shard has one or more replicas for availability and read scaling, with automatic failover; asynchronous replication means a small window of data loss, acceptable for a cache. Latency: keep data in RAM, use efficient serialisation, colocate caches near the application in the same zone, and avoid cross-region calls; tune network, and use a local in-process L1 cache for ultra-hot keys. Eviction: LRU or LFU with TTLs, with a memory limit per node. Consistency strategy: cache-aside with explicit invalidation or short TTLs; write-through for stronger freshness.

Likely follow-ups

- How do you protect against a cache stampede?
- How do you handle a hot key?

Diagram: Distributed cache sharding



Caching is the single highest-leverage performance optimisation. Get the design right or you'll fight cache stampedes and inconsistency bugs forever.

Client → Cache Cluster (Redis / Memcached, consistent hashing)

↓ miss

Primary DB → write result to cache → return to client

CACHE PATTERNS

Cache-aside (lazy loading):

App checks cache → miss → fetch DB → write cache → return

Pros: only cache what's used. Cons: cold start latency.

Write-through:

App writes DB and cache atomically

Pros: cache always fresh. Cons: write penalty, caches unused data.

Write-behind (write-back):

App writes cache only → async flush to DB

Pros: very fast writes. Cons: data loss if cache crashes before flush.

Read-through:

Cache fetches from DB on miss (cache library handles it)

Pros: simpler app logic.

CACHE STAMPEDE PREVENTION

Problem: key expires, 10K requests hit DB simultaneously

Solution 1: probabilistic early expiration

```
if time_to_expire < threshold and random() < probability:  
    refresh_in_background()
```

Solution 2: Redis lock on cache miss

```
if not redis.set(lock_key, 1, nx=True, ex=5):  
    sleep(0.05); return get_from_cache() # wait for first request to populate
```

Principal answer: Redis Cluster for 1M RPS: 6 nodes (3 primary + 3 replica), consistent hashing across slots. Each node handles ~167K RPS — well within Redis's 500K+ RPS single-node ceiling. Use pipelining for batch gets. Watch out for hot keys — a viral post accessed by millions needs key sharding or a local L1 in-process cache.



Official docs

- [Microsoft: Cache-Aside pattern](#)

13. Design a circuit breaker. When and how does a service fail fast?

reliability

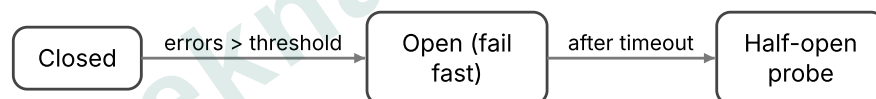
🗣 Spoken answer · ~85 s

A circuit breaker protects a service from repeatedly calling a failing or slow dependency, letting it fail fast and giving the dependency time to recover, which prevents cascading failures and resource exhaustion from threads waiting on timeouts. It has three states. Closed: requests flow normally, and the breaker counts failures and slow calls over a sliding window. When the failure rate or slow-call rate exceeds a threshold, for example 50 percent over at least 20 calls, it trips to Open. Open: calls fail immediately, without touching the dependency, returning an error or a fallback; after a cool-down period it moves to Half-Open. Half-Open: a limited number of trial requests are allowed through; if they succeed, the breaker closes; if they fail, it returns to open, perhaps with a longer wait. Design details: configure thresholds, window size, minimum call volume, wait duration, and what counts as failure, such as timeouts and 5xx errors but not client errors; provide fallbacks, like cached data, defaults, or degraded features; combine with timeouts, retries with backoff and jitter, bulkheads, which isolate resource pools per dependency, and rate limiting. Place it in the client library, such as Resilience4j or Polly, or in a service mesh like Envoy or Istio outlier detection.

Likely follow-ups

- What happens in the half-open state?
- How do bulkheads complement circuit breakers?

🗺 Circuit breaker states



Circuit breakers prevent cascade failures — when a downstream service is slow or down, fail fast instead of queuing up requests that time out.

State machine:

```
CLOSED (normal) —[failure threshold exceeded]→ OPEN (fail fast)
      ↑                               ↓
      └─[success threshold met]— HALF-OPEN ←[timeout elapsed]
```

IMPLEMENTATION

```

class CircuitBreaker:
    CLOSED, OPEN, HALF_OPEN = 0, 1, 2

    def __init__(self, failure_threshold=5, timeout=30, success_threshold=2):
        self.state = self.CLOSED
        self.failure_count = 0
        self.last_failure_time = None

    def call(self, fn, *args):
        if self.state == self.OPEN:
            if time.time() - self.last_failure_time > self.timeout:
                self.state = self.HALF_OPEN
            else:
                raise CircuitOpenError("Circuit open – failing fast")

        try:
            result = fn(*args)
            self._on_success()
            return result
        except Exception as e:
            self._on_failure()
            raise

    def _on_failure(self):
        self.failure_count += 1
        self.last_failure_time = time.time()
        if self.failure_count >= self.failure_threshold:
            self.state = self.OPEN

    def _on_success(self):
        if self.state == self.HALF_OPEN:
            self.state = self.CLOSED
        self.failure_count = 0

```

Interview tip: Libraries: Resilience4j (Java), Polly (.NET), Hystrix (deprecated). In cloud-native: use Envoy/Istio service mesh — circuit breaker at the proxy layer, no code changes. Set fallback: return cached data, default response, or degrade gracefully ("show general recommendations instead of personalised").

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [Microsoft: Cloud design pattern – Circuit Breaker](#)

14. Design a distributed transaction system. Handle payments across microservices.

reliability

Spoken answer · ~77 s

Distributed transactions across microservices can't rely on a single ACID database, and two-phase commit has drawbacks: blocking coordinators, poor availability, and tight coupling, so payments typically use the saga pattern. A saga breaks a business transaction into a sequence of local transactions, each in one service, with a compensating action to undo each step if a later step fails. Example: place an order, reserve inventory, charge payment, and ship. If payment fails, run compensations: release inventory and cancel the order. Two coordination styles. Orchestration: a central saga orchestrator, such as a workflow engine like Temporal or AWS Step Functions, tells each service what to do and tracks state, which makes the flow explicit and easier to reason about and monitor. Choreography: services react to events published by others, which is decoupled but harder to follow as flows grow. Reliability essentials: idempotent operations and idempotency keys, since retries are inevitable; the transactional outbox pattern, writing the state change and the event to the database in one local transaction and publishing afterwards, to avoid losing events; deduplication of consumers; timeouts and retries; and persistent saga state.

Likely follow-ups

- Why avoid two-phase commit across microservices?
- What does the outbox pattern solve?

Saga across services



Distributed transactions are notoriously hard. 2PC is slow and blocking. The practical answer for most systems is Saga pattern.

2PC (TWO-PHASE COMMIT) — AVOID IN MICROSERVICES

Phase 1 (Prepare): Coordinator asks all participants "can you commit?"

Phase 2 (Commit): If all say yes → commit. If any says no → rollback.

Problems:

- Coordinator is SPOF
- Blocking: participants hold locks during Phase 1
- Slow: 2 round trips of network calls minimum

SAGA PATTERN — THE PRACTICAL ANSWER

Choreography-based Saga (event-driven):

OrderService → emit OrderCreated

PaymentService → listen, charge card → emit PaymentCompleted

InventoryService → listen, reserve stock → emit StockReserved

ShippingService → listen, schedule delivery

Failure: each service publishes compensating event

PaymentFailed → OrderService listens → cancel order

StockInsufficient → PaymentService listens → refund payment

Outbox pattern – prevents dual-write problem

In same DB transaction:

BEGIN;

INSERT INTO orders (id, status) VALUES (123, 'pending');

INSERT INTO outbox (event_type, payload) VALUES ('OrderCreated', '{...}');

COMMIT;

Background process reads outbox → publishes to Kafka → marks as sent

Principal answer: Choreography works well for simple flows. Orchestration (Temporal, AWS Step Functions) is better when: you need visibility into flow state, workflows are complex, or you need timeouts and compensation logic in one place. Always implement idempotency — payment services must handle duplicate events without double-charging.

Official docs

- [Microsoft: Saga pattern](#)
- [Martin Fowler: Microservices](#)

15. Design a time-series metrics storage system (like Prometheus at scale).

data

🗣 Spoken answer · ~85 s

Requirements: ingest millions of samples per second from many sources, store compressed time series, support fast range queries and aggregations, downsampling, retention, and alerting. Data model: a series identified by a metric name and a set of labels, with timestamped float values. Ingestion path: agents scrape or push to a distributed ingestion layer; a stateless distributor validates and shards by hash of the series ID, replicates to several ingesters for durability, and the ingesters buffer recent samples in memory with a write-ahead log, flushing in time blocks, for example two hours, to long-term storage. Storage: time-based blocks in object storage such as S3 or GCS, with each block containing compressed chunks, using delta-of-delta timestamp encoding and XOR float compression like Gorilla, achieving about 1 to 2 bytes per sample, plus an inverted index from label values to series for fast lookup. Querying: a query frontend splits and caches queries by time range, queriers fetch from ingesters for recent data and from object storage for historical blocks, and merge results; compactors merge small blocks and apply downsampling to five-minute and one-hour resolutions for older data. Cardinality is the main danger: high-cardinality labels explode series counts, so enforce limits per tenant and drop or aggregate problem labels.

Likely follow-ups

- Why is label cardinality the main scaling risk?
- What does downsampling do for old data?

Time-series data has unique characteristics: always append-only, high cardinality, queries are usually recent data, compression is very effective.

Agents (Node Exporter, cAdvisor)

- Prometheus scrape / push gateway
- Ingest Service
- Write path: WAL → In-memory chunk → Flush to TSDB (disk blocks)
- Long-term: Thanos / Cortex → Object storage (S3/GCS)
- Query path: PromQL → Query Planner → Block reader → Result

STORAGE DESIGN

```
# Time-series: (metric_name, labels_map) → [(timestamp, value)]
# High cardinality problem: {user_id="..."} label → millions of series

# Delta encoding: store differences, not absolute values
# e.g., [100, 101, 103, 106] → [100, +1, +2, +3]

# XOR compression (Gorilla by Facebook):
# Consecutive float values differ in only a few bits → 1.37 bytes/sample
# vs 16 bytes uncompressed → 12× compression ratio
```

THANOS FOR GLOBAL SCALE

Per-cluster Prometheus (local retention: 2h)

- Thanos Sidecar → Object store (S3, GCS)
- Thanos Query → fan-out queries across all Prometheus + store
- Thanos Compactor → downsampling (5m, 1h resolution for old data)

Interview tip: High cardinality is the #1 killer of Prometheus performance. Each unique label combination = a separate time series. Never use user IDs or request IDs as labels. Use histograms for latency (not gauges per request). Retention policy: 15 days local, compress to 5m resolution after 7 days, 1h resolution after 30 days.

Official docs

- [Prometheus: Overview](#)

16. Design a real-time log aggregation pipeline. Handle 1TB/day of logs.

data

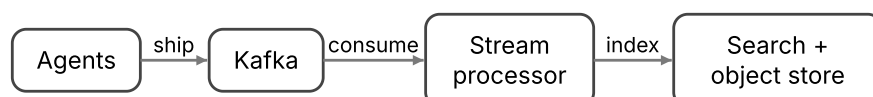
Spoken answer · ~83 s

Requirements: ingest about one terabyte a day of logs, which averages about 12 megabytes per second and perhaps several times that at peak, make them searchable within seconds, retain for weeks, with cost control. Pipeline: collection agents on each host, such as Fluent Bit, Vector, or Filebeat, tail container and application logs, add metadata like service, host, and Kubernetes labels, and forward with local buffering and backpressure handling. A durable buffer, Kafka, decouples producers from consumers, absorbs bursts, and allows replay; partition by service or hash for throughput, with retention of a day or two. A processing tier, consumers such as Logstash, Vector, or Flink, parses structured and unstructured logs, normalises fields, filters noise, redacts sensitive data, and routes. Storage and indexing: for hot searchable data, either Elasticsearch or OpenSearch, with time-based indices, sharded and replicated, with index lifecycle management moving data from hot to warm to cold tiers; or Loki, which indexes only labels and stores compressed chunks in object storage, much cheaper at this volume. Archive raw logs to object storage for compliance, and cheap long-term retention, in Parquet for later analytics. Query layer: Kibana or Grafana, with access control and saved searches; alerts on log patterns.

Likely follow-ups

- Why put Kafka between agents and storage?
- Why is Loki cheaper than Elasticsearch at this volume?

Log pipeline



Log pipelines must be: reliable (no log loss), scalable, queryable, and cost-controlled.

App instances → Fluent Bit (agent, lightweight)

↓

Kafka (buffer + replay capability)

- └─ Topic: logs.app (structured JSON)
- └─ Topic: logs.infra (system metrics)
- └─ Topic: logs.audit (compliance, long retention)

↓

Stream Processor (Flink / Spark Streaming)

- └─ Parse & enrich (add k8s metadata, geo-IP)
- └─ Detect anomalies (error spike alerting)
- └─ Route by log level

↓

- └─ Hot storage: Elasticsearch / OpenSearch (30 days, queryable)
- └─ Cold storage: S3 + Parquet (1 year, Athena queryable)

COST CONTROL — 1TB/DAY

- # Sampling: DEBUG logs sampled at 1%, INFO at 10%, WARN/ERROR at 100%
- # Compression: gzip on Kafka messages → ~5× reduction → 200 GB/day effective
- # Tiering: Elasticsearch for 30 days, then archive to S3 Glacier
- # Estimated monthly cost: ~\$500 ES + ~\$150 S3 for 1TB/day

Principal answer: Kafka is the critical buffer — it absorbs spikes (deploy wave, incident) and allows consumer lag without data loss. Set retention to 72h on Kafka topics. If Elasticsearch is down, consumers lag but Kafka retains. Never write directly from agent to Elasticsearch — you'll lose logs on ES restarts.



Official docs

- [Google SRE Book: Distributed tracing & logs \(Monitoring chapter\)](#)

17. Design a search autocomplete / typeahead system. Return results in <100ms.

data

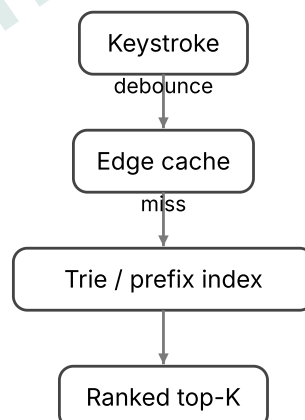
Spoken answer · ~88 s

Requirements: return top suggestions for a prefix within 100 milliseconds, as the user types, with high query volume and fresh trending terms. Data structure: a trie, or prefix tree, where each node represents a character and stores the top-K suggestions for that prefix, precomputed, so a lookup walks down the prefix, $O(\text{length})$, and returns the cached list immediately, with no subtree traversal. Memory for a large vocabulary is heavy, so compress using radix trees or finite state transducers, or store only popular terms; alternatively use a search engine's prefix index, such as Elasticsearch completion suggesters. Ranking: by query frequency, recency, personalisation, and location, with scores updated by an offline pipeline. Architecture: a data pipeline collects query logs, aggregates counts in batches, with MapReduce or Spark and a streaming layer for trending queries, builds the trie or index, and publishes snapshots to serving nodes; serving nodes load the structure in memory, sharded by prefix range or hash when it doesn't fit, with replication. A caching layer and CDN cache responses for popular prefixes, since the first one or two characters have huge repeat rates. Client-side optimisations: debounce keystrokes, cancel stale requests, cache previous results, and prefetch. Filter offensive or blocked terms. Handle typo tolerance with fuzzy matching on a secondary path.

Likely follow-ups

- Why precompute top-K suggestions at each trie node?
- How do you keep suggestions fresh for trending queries?

Typeahead path



Typeahead requires extremely low latency (every keystroke). The system must handle billions of queries and millions of distinct prefixes.

User types "plat" →

Client (debounce 100ms) → CDN edge cache (popular prefixes)

↓ miss

Trie service → Redis prefix cache

↓ miss

Precomputed prefix table (top-10 per prefix)

↓
Return ranked suggestions

DATA STRUCTURE: TRIE WITH TOP-K PER NODE

```
# Each trie node stores top-10 completions sorted by frequency
# "pl" → ["platform engineer", "platform team", "playbook", ...]
# "pla" → ["platform engineer", "platform team", "plan", ...]

# Precomputed and stored in Redis:
redis.setex("prefix:pla", 3600, json.dumps(top_10_results))

# Update frequency (offline, batch):
# Kafka → Spark aggregation (hourly) → recompute top-10 → Redis update
```

LATENCY BUDGET FOR 100MS

```
Network (client to edge CDN): ~5ms
CDN cache hit: ~1ms (serve cached top prefixes)
Cache miss → Redis lookup: ~2ms
Redis miss → DB lookup: ~10ms
Total (cache hit): ~6ms ✓
Total (cache miss): ~17ms ✓
```

Interview tip: Cache hit ratio for typeahead is very high — top 1000 prefixes cover ~80% of queries. Pre-warm the cache on deploy. Handle personalisation separately: blend global top-10 with user's recent searches client-side to avoid per-user cache fragmentation.

Official docs

- [Cloudflare Learning: What is a CDN?](#)
- [Redis documentation](#)

18. Design a distributed ID generation system (like Twitter Snowflake).

data

🗣 Spoken answer · ~73 s

Requirements: generate unique, roughly time-ordered 64-bit IDs across many machines, with high throughput, no central bottleneck, and no coordination on each request. Twitter's Snowflake layout packs the ID into 64 bits: one unused sign bit, 41 bits of timestamp in milliseconds since a custom epoch, giving about 69 years; 10 bits for a machine or worker ID, supporting 1,024 workers; and 12 bits for a per-millisecond sequence number, allowing 4,096 IDs per worker per millisecond. Each worker generates IDs locally: it takes the current timestamp, and if it equals the last timestamp, increments the sequence; if the sequence overflows, it waits for the next millisecond; otherwise the sequence resets to zero. The IDs are k-sortable by time, fit in a bigint, index well in B-trees, and need no network call. Operational concerns: assigning unique worker IDs, via ZooKeeper, etcd, or a configuration service at startup; clock skew and backwards clocks, handled by refusing to generate until the clock catches up, using monotonic checks, and NTP configuration; a restart must not reuse a timestamp-sequence combination.

Likely follow-ups

- What happens if the system clock moves backwards?
- How are worker IDs assigned?

IDs must be: globally unique, sortable by time (for pagination), high throughput (100K+/sec), no central bottleneck.

SNOWFLAKE ID STRUCTURE (64-BIT INTEGER)

Bit layout:

- [0] – sign bit (always 0)
- [1–41] – timestamp (ms since epoch) → ~69 years of unique IDs
- [42–51] – machine/datacenter ID (10 bits → 1024 machines)
- [52–63] – sequence number (12 bits → 4096 IDs/ms/machine)

Max throughput: $1024 \text{ machines} \times 4096 \text{ IDs/ms} = 4.2 \text{ billion IDs/second}$

```

class SnowflakeGenerator:
    EPOCH = 1288834974657 # Twitter epoch (Nov 4, 2010)

    def __init__(self, machine_id):
        self.machine_id = machine_id # 0-1023
        self.sequence = 0
        self.last_ms = -1

    def generate(self):
        ms = int(time.time() * 1000) - self.EPOCH
        if ms == self.last_ms:
            self.sequence = (self.sequence + 1) & 0xFFF
            if self.sequence == 0:
                while ms <= self.last_ms: # busy wait for next ms
                    ms = int(time.time() * 1000) - self.EPOCH
        else:
            self.sequence = 0
        self.last_ms = ms
        return (ms << 22) | (self.machine_id << 12) | self.sequence

```

Alternatives: UUID v4 (random, not sortable, 128-bit), ULID (sortable UUID), database sequences (single SPOF). For most services, a database auto-increment sequence is fine up to ~50K inserts/sec. Use Snowflake when you need sortable IDs across shards without a central DB sequence.

Official docs

- [Twitter Snowflake \(ID generation\)](#)

19. **Design a multi-region Kubernetes platform. Handle active-active across 3 regions.**

platform

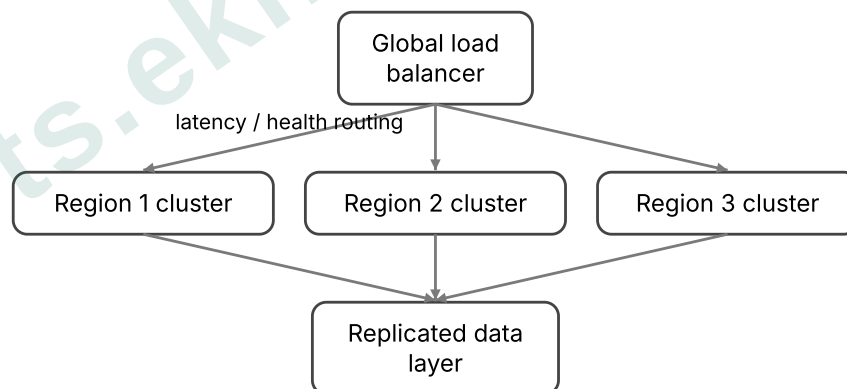
🔊 Spoken answer · ~79 s

Requirements: run Kubernetes across three regions in active-active mode, so each region serves live traffic and survives the loss of another region, with consistent configuration and data strategy. Cluster topology: independent regional clusters, not one stretched control plane, since etcd is latency-sensitive; each cluster spans three availability zones with node pools per zone, a highly available control plane, and its own autoscaling. Traffic: global load balancing via anycast or DNS-based geo and latency routing, such as Cloud Load Balancing, Route 53 latency records, or Cloudflare, with health checks that remove an unhealthy region automatically; users route to the nearest healthy region; capacity in each region must absorb a failed region's share, typically running at 60 to 70 percent. Configuration: GitOps with Argo CD or Flux applying the same manifests to all clusters, using ApplicationSets or fleet management with per-region overlays; policy as code for consistency; progressive rollouts one region at a time, with automated checks, so a bad release doesn't hit every region at once. Service connectivity: a multi-cluster service mesh or multi-cluster services for cross-region calls when needed, but prefer region-local calls to avoid latency and failure coupling.

Likely follow-ups

- Why avoid stretching one Kubernetes control plane across regions?
- How do you size capacity for a regional failure?

🚧 Multi-region platform



Multi-region Kubernetes is the hardest class of PE design question. Focus on: traffic routing, data consistency, failure isolation, and operational complexity.

Global Traffic Layer:

Anycast IP (Global Accelerator / Cloud Armor LB)
→ Route to nearest healthy region (health check + latency-based routing)

Per-region stack:

GKE / EKS cluster
└─ App workloads (stateless – active in all regions)

- └─ Local PostgreSQL primary + cross-region read replica
- └─ Redis cluster (eventually consistent cross-region via CRDTs)

Control plane:

- ArgoCD (one per region, sync from same Git repo)
- Fleet management via GKE Hub / EKS anywhere

DATA STRATEGIES PER TIER

- Stateless services: Deploy identical in all regions. No data concerns.
- User sessions: Redis CRDT (Riak) or route user to same region (sticky DNS)
- Transactional data: Primary in one region, failover via Aurora Global / Spanner
- Analytics/events: Multi-region Kafka MirrorMaker 2 → eventual consistency
- User-generated content: S3/GCS with CRR (cross-region replication)

Principal answer: Don't default to active-active. It's 3–5× harder than active-passive. Ask: what's the actual RTO/RPO requirement? For most companies, active-passive (RPO < 1min, RTO < 15min) via Aurora Global DB + standby cluster is correct. Active-active only when regulatory requirements demand data residency in each region simultaneously, or when single-region latency is unacceptable for global users.

Official docs

- [AWS Well-Architected: Reliability pillar](#)
- [Google Cloud: Multi-region deployment archetypes](#)

20. Design a CI/CD platform for 500 microservices and 200 engineers.

platform

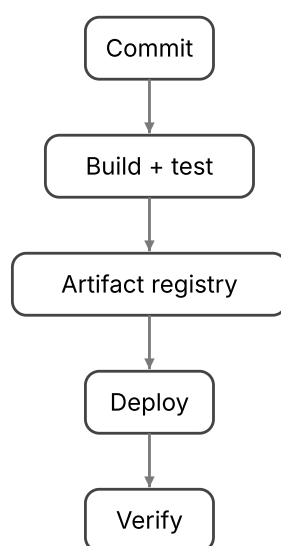
🗣 Spoken answer · ~88 s

Requirements: a platform serving 500 microservices and 200 engineers, with fast builds, safe deployments, self-service, and governance. Source and triggers: Git repositories, mostly polyrepo per service or a few monorepos, with webhooks triggering pipelines. Build: autoscaling ephemeral runners on Kubernetes, with remote caching of dependencies and layers, and a shared build service producing container images once, tagged by commit SHA, pushed to a registry with vulnerability scanning, SBOM generation, and signing. Pipeline definition: standardised templates or reusable workflows, so each service has a short file referencing the golden path, with escape hatches; consistent stages: lint, test, build, scan, publish. Test: parallelised unit tests, contract tests between services, and ephemeral environments for integration tests. Deploy: GitOps with Argo CD, each service's manifests in a configuration repository or folder, promotion from dev to staging to production by pull request or automated policy; progressive delivery with canary analysis through Argo Rollouts; automatic rollback on SLO regressions. Governance: policy as code, admission control requiring signed images, secrets via a secret manager and OIDC, no static credentials, RBAC per team. Developer experience: service templates, a portal such as Backstage showing build and deployment status and ownership, deployment dashboards, and notifications. Scale concerns: runner capacity and queue time, registry throughput, monorepo affected-service detection, and the platform's own SLOs.

Likely follow-ups

- How do reusable pipeline templates keep 500 services consistent?
- How does progressive delivery fit in?

🚧 CI/CD platform



At this scale, the pipeline itself becomes a product. Engineers are your customers. Standardisation, speed, and self-service are the design goals.

Developer push → GitHub

- GitHub Actions (or Tekton on-cluster)
- ├─ Lint + Test (parallel, cached dependencies)
- ├─ Container build (Buildkit, layer caching)
- ├─ Security scan (Trivy, Gype – fail on CRITICAL)
- ├─ Push to Artifact Registry (tagged by git SHA)
- ├─ Update GitOps repo (PR to k8s-config repo)

↓

ArgoCD (per-environment)

- ├─ Staging: auto-sync on PR merge
- ├─ Prod: sync on manual approval + canary analysis
- ├─ Rollback: git revert → ArgoCD auto-applies

STANDARDISATION VIA GOLDEN PATH

```
# Backstage software template:
# Engineer runs: `npx @backstage/create-app`
# Generates: Dockerfile, GitHub Actions workflow, Helm chart,
#             ArgoCD Application, PagerDuty integration, Datadog monitors
# 80% of services use the golden path. 20% get custom support.
```

METRICS THAT MATTER

DORA metrics:

Deployment frequency:	target daily or on-demand per service
Lead time for change:	commit → production < 30 minutes
Change failure rate:	< 5% of deployments cause incidents
MTTR:	< 1 hour for P1 incidents

Principal answer: At 500 services, the biggest problem isn't tooling — it's standardisation. Platform team builds golden paths (opinionated templates) and scores services on adoption. Enforce via Scorecard in Backstage: security scan ✓, SLO defined ✓, runbook linked ✓. Teams that score <70% get Platform team office hours, not mandates.



Official docs

- [Martin Fowler: Microservices](#)

21. Design a feature flag system for safe production rollouts.

platform

🗣 Spoken answer · ~86 s

Requirements: toggle features at runtime without deploying, target subsets of users, roll out gradually, and kill features instantly, with negligible latency impact and high availability even if the flag service fails. Components. A management UI and API for creating flags, defining targeting rules, such as percentage rollouts, user attributes, segments, and environments, with approvals and audit logs, stored in a database. A flag delivery layer: SDKs embedded in applications that fetch the full ruleset on startup, cache it in memory, and receive updates via streaming, using server-sent events, WebSockets, or polling, so evaluation happens locally in microseconds with no network call per check. Evaluation: deterministic hashing of the user ID plus the flag key into a bucket, so a given user consistently sees the same variation and percentage rollouts are sticky; rules evaluated in order. Resilience: SDKs keep the last known values and fall back to safe defaults compiled into code if the service is unreachable; a CDN or relay proxy distributes configuration to scale to many clients; client-side SDKs for browsers and mobile get only the evaluated values, not rules, to avoid leaking targeting data. Observability: record flag exposure events for analytics and experimentation, and integrate with monitoring so flag changes appear as annotations on dashboards.

Likely follow-ups

- Why evaluate flags locally in the SDK?
- What happens if the flag service is down?

Feature flags decouple deployment from release. Critical for: canary rollouts, A/B testing, kill switches, and dark launches.

Flag Config Store (PostgreSQL + Redis cache)

- ├─ flag_key, enabled, rollout_percentage, targeting_rules, variants
- └─ Admin UI → flag management

SDK (embedded in each service):

- Fetch flags on startup → local cache (in-memory)
- Refresh every 30s via polling or SSE (Server-Sent Events)
- Evaluate: is user X in rollout bucket for flag Y?

Evaluation logic:

1. Is flag globally off? → return default
2. Is user in targeting rule? (beta_users group, specific user IDs) → return variant
3. $\text{Hash}(\text{user_id} + \text{flag_key}) \bmod 100 < \text{rollout_pct?}$ → return variant
4. Return default

CONSISTENT BUCKETING

```
def get_bucket(user_id: str, flag_key: str) -> int:
    # Same user always gets same bucket for same flag
    hash_input = f"{user_id}:{flag_key}"
    return int(hashlib.sha256(hash_input.encode()).hexdigest(), 16) % 100

# user_id="abc123", flag="new_checkout" → bucket 47
# Always 47 → consistent experience across sessions and services
```

Principal answer: Use flag SDK with <1ms evaluation latency — never make a network call per flag evaluation in the request path. Cache flags locally, stream updates via SSE. Log every flag evaluation to a data warehouse for analytics (who saw which variant, what was their conversion rate). This is the foundation of experimentation infrastructure.

Official docs

- [Kubernetes: Deployments](#)
- [Martin Fowler: Feature Toggles](#)

22. Design an Internal Developer Platform (IDP). What components does it need?

platform

Spoken answer · ~74 s

An internal developer platform gives engineers self-service access to the capabilities they need to build, ship, and run software, through paved paths. Components. Developer portal: a front door such as Backstage with a software catalogue showing services, owners, dependencies, documentation, and scorecards; templates for scaffolding new services; and links to dashboards. Source and CI/CD: standardised pipelines, reusable templates, artefact registry, scanning, and signing; GitOps-based deployment with environment promotion and progressive delivery. Runtime: managed Kubernetes or a serverless layer, with namespaces or tenants per team, ingress, autoscaling, and a service mesh if needed. Infrastructure self-service: a way to provision databases, queues, buckets, and caches through declarative requests, using Terraform modules, Crossplane, or an API, with guardrails and policy built in. Environments: ephemeral preview and test environments on demand. Security and compliance: secrets management, identity and access, policy as code, vulnerability management, and audit logging, embedded by default. Observability: standard logging, metrics, tracing, dashboards, and alerting pre-wired for each service, plus SLO tooling. Cost visibility: per-team usage and budgets. Platform APIs and CLI for automation, plus documentation and support channels.

Likely follow-ups

- What does the software catalogue provide?
- Why build golden paths that are optional?

An IDP reduces cognitive load for developers — they shouldn't need to understand Kubernetes, Terraform, or CI/CD to ship software safely.

Developer-facing:

- └─ Service Catalogue (Backstage)
 - └─ Software templates (golden paths)
 - └─ Tech docs (docs-as-code, MkDocs)
 - └─ Scorecards (compliance, security posture)
- └─ Self-service portal
 - └─ Provision environment (namespace + resources)
 - └─ Manage secrets (Vault UI)
 - └─ View logs, traces, metrics (linked from service page)
- └─ CLI tool
 - └─ `platform new service` / `platform env create` / `platform deploy`

Platform-facing (hidden from devs):

Kubernetes + ArgoCD + Terraform + Vault + Crossplane

THE GOLDEN PATH CONTRACT

Developer provides:

- └─ service name
- └─ tech stack (Node/Python/Go/Java)
- └─ resource requirements (CPU, memory, replicas)
- └─ environment variables (plain) + secrets (references to Vault paths)

Platform provides automatically:

- └─ Kubernetes namespace + RBAC
- └─ CI/CD pipeline (GitHub Actions workflow)
- └─ Ingress + TLS certificate
- └─ Observability (metrics, logs, traces pre-wired)
- └─ PagerDuty integration
- └─ Cost attribution labels

Principal answer: Measure IDP success by: time-to-first-deployment for a new service (target <1 day from zero), % services on golden path (target >80%), and developer satisfaction score (quarterly survey). Don't build what you can buy — Backstage for catalogue, Vault for secrets, Crossplane for cloud resources. Build the integration layer, not the primitives.

Official docs

- [Backstage: What is Backstage?](#)

23. Design a global CDN architecture. How does edge caching work at scale?

networking

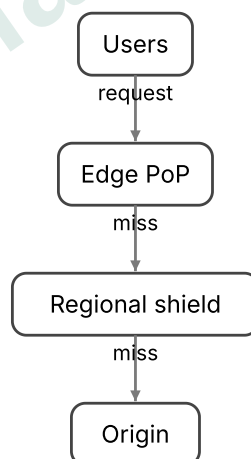
Spoken answer · ~85 s

A CDN caches content on edge servers close to users, reducing latency, offloading the origin, and absorbing traffic spikes and attacks. Architecture. A global network of points of presence, each with edge servers, typically with tiers: edge caches near users, regional or mid-tier caches that consolidate misses, and finally the origin or origin shield, which reduces origin load. Request routing steers users to the best edge using anycast IP, so BGP routes to the nearest location, or DNS-based geo and latency steering, along with health and load awareness. Caching behaviour: on a request, the edge checks its cache by key, built from the URL and selected headers; on a hit it serves directly; on a miss it fetches from the next tier or origin, stores it according to Cache-Control, Expires, and ETag headers, and serves the client; request coalescing collapses simultaneous misses for the same object into one origin request. Eviction is LRU-like per node, with large objects handled specially. Static assets cache for long periods with versioned URLs for cache busting; dynamic content can use short TTLs, stale-while-revalidate, or edge compute. Invalidation: purge by URL, tag, or prefix through an API that propagates across the network quickly, or avoid it with content-hashed filenames.

Likely follow-ups

- What is an origin shield?
- How do versioned filenames reduce the need for purging?

CDN caching



CDNs reduce origin load and user latency by serving content from edge nodes close to users. Design involves: cache hierarchy, cache invalidation, origin protection, and routing.

```
User (Mumbai) → Anycast DNS → Nearest PoP (Mumbai edge)
                        ↓ cache miss
                Regional hub (Singapore)
                        ↓ cache miss
                Origin shield (single region close to origin)
```

↓ cache miss
Origin servers

CACHE KEY DESIGN

```
# Default cache key: URL only
# Vary by: Accept-Encoding (gzip vs br), Accept-Language, device type

# Cache-Control headers (origin controls caching):
Cache-Control: public, max-age=31536000, immutable      # versioned assets
Cache-Control: public, max-age=300, stale-while-revalidate=60 # API responses
Cache-Control: private, no-store                        # user-specific data
```

CACHE INVALIDATION STRATEGIES

1. URL versioning (best): /app.v1.2.3.js → never expires, change URL to invalidate
2. Cache purge API: cdn.purge("/api/products/123") → instant but expensive at scale
3. Surrogate keys: tag responses with logical keys → purge by tag
4. Short TTL: max-age=60 → accept 60s staleness, no active invalidation

Principal answer: Origin Shield is critical for large CDNs — it collapses all cache misses from all edge PoPs into a single PoP before hitting origin. Without it, a cache invalidation for a popular asset triggers N simultaneous origin requests (N = number of edge PoPs). With Origin Shield: exactly 1 origin request per object per cache refresh.

Official docs

- [Cloudflare Learning: What is a CDN?](#)
- [Microsoft: Cache-Aside pattern](#)

24. Design a load balancer. Explain L4 vs L7 and when to use each.

networking

🗣 Spoken answer · ~64 s

A load balancer distributes incoming traffic across backend servers to improve availability, scalability, and performance. Layer 4 load balancers operate at the transport level, TCP and UDP, making routing decisions from IP addresses and ports without inspecting the payload. They're very fast and efficient, handle millions of connections, support any protocol, and can preserve client IPs or use direct server return, but they can't route on URLs or headers, and can't terminate HTTP-aware features. Layer 7 load balancers operate at the application level, parsing HTTP, so they can route by host, path, header, or cookie, terminate TLS, rewrite requests, apply authentication, compression, caching, rate limiting, and WAF, and provide richer observability, at a higher resource cost and somewhat more latency. Use L4 for raw performance, non-HTTP protocols such as databases or gaming, TLS passthrough, and as the first tier in front of L7 proxies; use L7 for microservice routing, canary releases, and content-based decisions.

Likely follow-ups

- Which algorithm would you use for sticky routing?
- How do you make the load balancer itself highly available?

🔗 L4 vs L7

L4 load balancer	L7 load balancer
Routes on IP and port	Routes on host, path, headers
Fast, protocol agnostic	TLS termination, retries
No content awareness	More CPU per request

Load balancers distribute traffic across backend instances. L4 operates at transport layer (TCP/UDP); L7 operates at application layer (HTTP/gRPC/WebSocket).

L4 Load Balancer (TCP):

- └ Sees: IP, port, TCP flags – not HTTP content
- └ Very fast: ~100ns decision (hardware), ~10µs (software)
- └ Algorithms: IP hash, round-robin, least connections
- └ Use: any TCP traffic, non-HTTP, highest throughput needs

L7 Load Balancer (HTTP):

- └ Sees: URL, headers, cookies, body, TLS cert
- └ Slower: ~1ms per request (TLS + HTTP parsing overhead)
- └ Features: content routing, auth, rate limiting, WAF, health checks
- └ Use: HTTP APIs, A/B testing, canary routing, session affinity

HEALTH CHECKING

```
# L4: TCP connect probe (is the port open?)
# L7: HTTP GET /health → expect 200 + healthy body
# Passive: monitor real request error rates → remove if error rate > threshold

# Remove from pool: 3 consecutive failures in 10s
# Add back to pool: 2 consecutive successes (conservative)
```

CONSISTENT HASHING FOR SESSION AFFINITY

```
# Sticky sessions: route user to same backend
# Option 1: Cookie (LB sets __lb_session=backend_id)
# Option 2: IP hash (fragile – changes with proxies/CG-NAT)
# Option 3: Consistent hashing on session token (best for scale)
```

Principal answer: Use both in sequence. L4 (NLB/IPVS) at the front for TLS termination and connection distribution. L7 (NGINX/Envoy) behind it for routing, auth, and observability. This two-tier approach scales better than a single L7 handling all TLS connections (TLS handshakes are CPU-intensive).

Official docs

- [Cloudflare Learning: OSI model](#)
- [Cloudflare Learning: Load balancing](#)

25. **Design a WebSocket / real-time messaging system. Handle 1M concurrent connections.**

networking

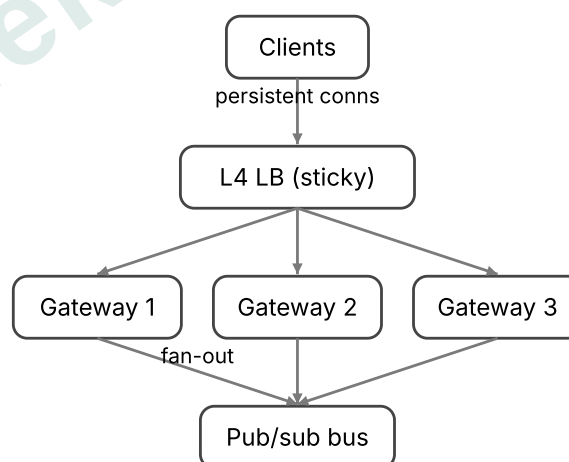
🗣 Spoken answer · ~82 s

Requirements: hold one million concurrent WebSocket connections, deliver messages in real time with low latency, and scale horizontally. A single modern server can hold roughly 50,000 to several hundred thousand idle connections if tuned, so 1 million needs perhaps 10 to 20 gateway servers with headroom. Architecture: clients connect through a load balancer, L4 or L7 with WebSocket support, to stateless-ish connection gateway servers that terminate WebSockets and authenticate on connection. Because connections are stateful and long-lived, the gateway maintains a mapping of user to connection, kept in memory and mirrored in a shared registry such as Redis, mapping user ID to gateway node, for routing. To send a message to a user, the sender's service publishes to a message bus, such as Kafka, Redis pub/sub, or NATS, and the gateway holding that user's connection delivers it; or look up the registry and forward to the right gateway. Groups and channels use pub/sub topics, with gateways subscribing to channels that have local members. Reliability: heartbeats and ping-pong to detect dead connections, client reconnection with exponential backoff and jitter, message sequence numbers and acknowledgements for resend after reconnect, and offline storage for undelivered messages with push notification fallback.

Likely follow-ups

- How does a message find the gateway that holds a user's connection?
- Why use jitter for client reconnects?

🚧 **WebSocket scale-out**



Long-lived connections at scale require stateful servers — which breaks horizontal scaling assumptions.

```
Client —WebSocket—> WS Gateway (stateful, sticky LB)
                        ↓ (Pub/Sub)
                    Redis Pub/Sub
                        ↑
            Message Producer Services
```

(Chat, Notifications, etc.)

WS Gateway publishes received messages → Redis → all gateways
→ Each gateway fans out to its local connected clients

CONNECTION CAPACITY PLANNING

```
# 1M concurrent connections:
# Each idle WebSocket: ~40KB memory (kernel TCP state + buffer)
# 1M × 40KB = ~40GB RAM just for connections
# Distribute: 50 servers × 20K connections each → 800MB RAM/server ✓

# Message fan-out: if 1M users in 10K channels (100 users/channel avg)
# Send to channel: 1 Redis publish → 50 WS servers receive → each fans to local subs
# Cost: O(channels × servers) not O(users) – scalable
```

SCALING CHALLENGES

1. Sticky sessions: WS connections are long-lived → LB must route to same server
2. Reconnection: client disconnects → reconnect with exponential backoff (1s, 2s, 4s...)
3. State recovery: on reconnect, send all missed messages since last_seen_id
4. Heartbeat: ping every 30s, close if no pong in 10s (detect zombie connections)

Interview tip: Redis Pub/Sub is eventually consistent — some messages may be lost on Redis restart. For guaranteed delivery use Redis Streams or Kafka. Distinguish: fire-and-forget (notifications) vs guaranteed delivery (chat history). For chat, store every message in Cassandra; WebSocket is just the delivery mechanism, not the source of truth.



Official docs

- [RFC 6455: The WebSocket Protocol](#)

Spoken answer · ~88 s

I choose based on coupling, timing, and delivery semantics. A direct API call, synchronous request-response over HTTP or gRPC, fits when the caller needs an immediate result to continue, such as fetching a user profile or validating a payment; it's simple and easy to reason about, but couples availability and latency: if the callee is down or slow, the caller suffers, so you need timeouts, retries, and circuit breakers. A message queue, like SQS or RabbitMQ, decouples producer and consumer in time: the producer enqueues work and moves on, consumers process at their own pace, with buffering during spikes, retries, and dead-letter queues; each message is typically consumed by one worker, ideal for task distribution, background jobs, and smoothing load. Event streaming, like Kafka, Pulsar, or Kinesis, is a durable, ordered, replayable log of events that many independent consumers read at their own offsets; ideal for event-driven architectures, data pipelines, audit trails, change data capture, and when new consumers need to reprocess history. Decision criteria: need an immediate answer, use a direct call; work can be done later and each task handled once, use a queue; multiple systems care about the same event, ordering within a key matters, or replay is needed, use a stream. Consider throughput, retention, ordering, and operational complexity.

Likely follow-ups

- When is replay a deciding factor?
- How do queues smooth load spikes?

This decision defines service coupling, reliability, and operational complexity. Get it wrong and you'll fight it for years.

Direct API call (synchronous, point-to-point):

- ✓ Simple, request-response, immediate feedback
- ✓ Use when: caller needs the result, operation is <100ms, failure = retry immediately
- ✗ Tight coupling: if downstream is down, upstream fails
- ✗ No buffering: traffic spike propagates directly to consumer
- Examples: user login validation, inventory check before checkout

Message queue (async, point-to-point):

- ✓ Decouples producer from consumer availability
- ✓ Natural load levelling (workers drain at their own pace)
- ✓ Use when: task can be deferred, producer shouldn't wait, work is heavy
- ✗ Not real-time; adds latency (ms to seconds)
- Examples: send email, resize image, process payment confirmation
- Tools: SQS, RabbitMQ, Celery

Event streaming (pub/sub, many consumers):

- ✓ Multiple independent consumers process same event
- ✓ Replay: process historical events for new consumers
- ✓ Use when: audit log, fan-out, event sourcing, real-time analytics
- ✗ Higher operational complexity (Kafka cluster management)

- Examples: order events consumed by inventory + analytics + notifications
- Tools: Kafka, Kinesis, Pub/Sub

Principal decision rule: Start with synchronous API calls. When you see: retry queues appearing in code, caller blocking for long operations, fan-out to multiple consumers, or cross-team dependencies — introduce async. Don't over-engineer with Kafka before you need it.

Official docs

- [Apache Kafka documentation \(streaming ingestion\)](#)
- [Azure: Asynchronous messaging options](#)

27. Explain database sharding strategies and their trade-offs.

tradeoffs

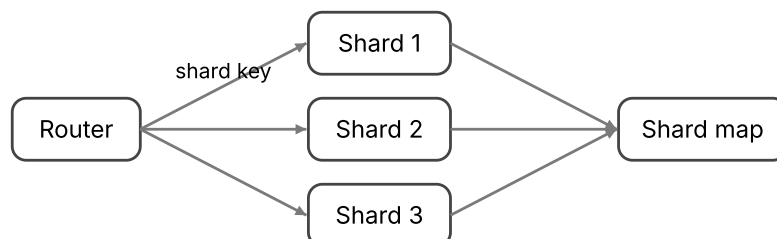
Spoken answer · ~65 s

Sharding splits a dataset across multiple database nodes so that no single machine holds everything, scaling storage and throughput horizontally. The shard key choice is critical. Strategies. Range-based: contiguous ranges of the key, like user IDs 1 to 1 million, to shard one. Efficient range queries and simple, but prone to hotspots with monotonic keys like timestamps, where the newest shard takes all writes, and to uneven distribution. Hash-based: $\text{shard} = \text{hash}(\text{key}) \bmod N$ or via consistent hashing. Even distribution and no hotspot for random keys, but range queries scatter across all shards, and resharding with mod N moves most data, which consistent hashing mitigates. Directory-based: a lookup service maps keys to shards, offering flexibility to move tenants or rebalance, at the cost of an extra lookup and a critical component that must be highly available. Geographic sharding by region for data residency and latency. Entity or tenant-based sharding in multi-tenant systems keeps a tenant's data together.

Likely follow-ups

- Why do monotonic keys cause hotspots with range sharding?
- What are virtual shards?

Sharded database



Sharding partitions data across multiple database instances. The sharding key choice is the most consequential schema decision you'll ever make.

SHARDING STRATEGIES

Range sharding: Users A–M → Shard 1, N–Z → Shard 2

Pros: range queries efficient, easy to add shards at end

Cons: hot spots (all new users go to last shard), uneven distribution

Hash sharding: $\text{Shard} = \text{hash}(\text{user_id}) \% N$

Pros: even distribution, no hot spots

Cons: range queries hit all shards, resharding requires moving half the data

Directory sharding: lookup table maps `user_id` → `shard_id`

Pros: flexible redistribution, no resharding math

Cons: lookup table is SPOF, extra network round-trip per query

Geography sharding: EU users → EU shard, US users → US shard

Pros: data residency compliance, lower latency for regional users

Cons: cross-region queries complex, uneven growth if one region grows faster

RESHARDING — THE HARD PROBLEM

Online resharding without downtime:

1. Add new shard (start empty)
2. Dual-write: new writes go to old + new shard
3. Backfill: migrate existing data from old to new shard
4. Verify: compare checksums
5. Cut over: reads switch to new shard
6. Remove dual-write, decommission old shard

Principal answer: Shard only when you must. Most systems never need sharding — a well-tuned PostgreSQL on high-memory hardware handles 100K+ RPS with proper indexing and read replicas. Sharding adds enormous operational complexity: cross-shard joins become application logic, distributed transactions across shards require Sagas, and resharding can take months. Consider Vitess (MySQL) or Citus (PostgreSQL) before rolling your own sharding.



Official docs

- [Microsoft: Sharding pattern](#)

28. Design a leader election system for distributed coordination.

tradeoffs

🗣 Spoken answer · ~75 s

Leader election chooses one node among many to act as coordinator, for tasks like scheduling, writing to a primary, or holding a lock, with automatic failover when it fails. The key requirements are safety, at most one leader at a time, and liveness, a new leader is chosen quickly after failure. Approaches. Use a consensus-based coordination service such as ZooKeeper, etcd, or Consul rather than building your own. With etcd, candidates try to create a key with a lease in a transaction, or use the election API; the one that succeeds is leader and must keep renewing its lease via heartbeats; if it crashes or is partitioned, the lease expires, the key disappears, and another candidate wins. ZooKeeper uses ephemeral sequential znodes: each candidate creates one, the lowest sequence number is the leader, and the others watch their predecessor. Underlying consensus algorithms, Raft and Paxos, elect leaders by terms and majority votes: a node that doesn't hear from a leader within a randomised timeout becomes a candidate, requests votes, and wins with a majority, and the term number prevents stale leaders.

Likely follow-ups

- What is a fencing token?
- How does Raft avoid two leaders in the same term?

Leader election ensures exactly one node acts as coordinator. Required for: primary DB selection, distributed cron (run job once), cluster membership.

OPTION 1: ETCD / ZOOKEEPER (PURPOSE-BUILT)

```
# etcd lease-based leader election
import etcd3

client = etcd3.client()
# Try to acquire lease (TTL = 10s – leader must renew every 5s)
lease = client.lease(ttl=10)

# Atomic write: only succeeds if key doesn't exist
success, _ = client.transaction(
    compare=[client.transactions.version('/election/leader') == 0],
    success=[client.transactions.put('/election/leader', node_id, lease=lease)],
    failure=[]
)
if success:
    print("I am the leader")
    # Keep renewing lease – stop renewing = abdicate leadership
    while True:
        lease.refresh()
        time.sleep(5)
```

OPTION 2: REDIS (SIMPLER, LESS RELIABLE)

```
# SET NX (set if not exists) + TTL
acquired = redis.set('leader', node_id, nx=True, ex=30)
if acquired:
    # Renew: must reset TTL before it expires
    redis.expire('leader', 30)
```

Split-brain problem: two nodes think they're leader simultaneously

Cause: network partition – leader A can't see quorum but still holds lease

Solution: Raft consensus (etcd uses it)

- Leader needs majority vote to remain leader
- If A is isolated from majority, it steps down

Interview tip: For Kubernetes: use Lease objects (leader-election library built into controller-runtime). For general use: etcd is the gold standard. Redis leader election is fine for advisory locks (best-effort) but not safe for cases where dual-leadership causes data corruption.



Official docs

- [Google SRE Book: Managing Critical State \(consensus\)](#)
- [Raft consensus algorithm](#)
- [Microsoft: Leader Election pattern](#)

29. Design a multi-tenant SaaS platform. How do you isolate tenants?

tradeoffs

Spoken answer · ~88 s

Multi-tenant SaaS serves many customers from shared infrastructure, and the design question is the isolation level, traded against cost and operational simplicity. Three models for data. Shared database, shared schema, with a tenant ID column on every table: cheapest and simplest to scale to many tenants, but weakest isolation, so enforce tenant filtering centrally via the data access layer, row-level security in PostgreSQL, and tests, since a missing filter leaks data. Shared database with a schema per tenant: stronger logical separation, easier tenant-specific changes, but schema migrations multiply and limits arise at thousands of tenants. Database per tenant: strongest isolation, per-tenant backup, restore, encryption keys, and data residency, and easy to move large tenants, but higher cost and operational overhead. A tiered approach is common: small tenants share, large or regulated tenants get dedicated resources. Compute isolation: shared stateless services with tenant context carried through authentication tokens; noisy-neighbour protection through per-tenant rate limits, quotas, fair-share queues, and resource limits; dedicated cells or clusters for premium tenants. Security: tenant-aware authorisation everywhere, per-tenant encryption keys, audit logs, and no cross-tenant caching leaks, keying caches by tenant. Operations: per-tenant metrics, cost attribution, tenant onboarding automation, and safe rollouts by tenant cohort. Cell-based architecture limits blast radius by deploying independent copies and routing tenants to cells.

Likely follow-ups

- How does row-level security protect shared tables?
- How do you contain noisy neighbours?

Tenancy model choice affects: cost, isolation strength, compliance, and operational complexity. There's no universally right answer — it depends on customer tier and requirements.

Model 1: Shared everything (silo = false)

One DB, one app cluster, tenant_id column on every table

✓ Cheapest, easiest to operate

✗ Noisy neighbour, blast radius = all tenants, compliance risk (HIPAA?)

→ Use for: SMB SaaS, free tier, <10K tenants

Model 2: Shared app, separate DB per tenant

One app cluster, DB per tenant (separate schemas or instances)

✓ Data isolation, easier compliance

✗ DB sprawl (1000 tenants = 1000 DBs), connection pool explosion

→ Use for: mid-market, regulated industries

Model 3: Dedicated cluster per tenant

Separate Kubernetes namespace or cluster per tenant

✓ Full isolation, custom configuration per tenant

✗ Very expensive, high operational overhead

→ Use for: enterprise, government, customers who require it contractually

Free / Starter tier: shared pool (Model 1), noisy neighbour acceptable
 Professional tier: shared app + separate DB schema (Model 2)
 Enterprise tier: dedicated namespace or cluster (Model 3)
 # Migrate tenant between tiers: move data, update routing config, no app code change

Principal answer: Implement tenant routing at the gateway layer — every request carries tenant_id (from JWT or subdomain). Gateway resolves tenant_id → connection string → routes to correct DB. App code never deals with routing. This makes tier migration transparent to the application.

Official docs

- [Kubernetes: Multi-tenancy](#)

30. You inherit a monolith. How do you migrate it to microservices safely?

tradeoffs

Spoken answer · ~85 s

Migrating a monolith safely means doing it incrementally, not as a big-bang rewrite, which usually fails. Use the strangler fig pattern: put a routing layer, such as an API gateway or proxy, in front of the monolith, then gradually extract functionality into new services and route those requests to them, while the monolith keeps handling the rest, until it shrinks away. Steps. First, understand the system: map domains, dependencies, and data ownership, using domain-driven design to find bounded contexts, and use observability to see real call patterns. Make the monolith modular internally, with clear boundaries, since extracting a poorly separated module is hard. Choose the first candidates carefully: capabilities with clear boundaries, low coupling, high business value or pain, such as a scaling bottleneck or frequent changes; avoid the most entangled core first. Extract the service, and handle data, the hardest part: move to separate databases per service, by first duplicating or synchronising data, using change data capture or events, then switching ownership; avoid shared databases long term. Run old and new in parallel, with shadow traffic, feature flags, and gradual percentage-based routing, comparing results before cutover; keep the ability to roll back. Build supporting platform capabilities first: CI/CD, observability, service discovery, and API standards.

Likely follow-ups

- Why is the data the hardest part of the migration?
- How does the strangler fig pattern reduce risk?

The strangler fig pattern: gradually extract services from the monolith without a big-bang rewrite. The monolith keeps running and shrinks over time.

Phase 1: Observe (month 1–2)

Map the monolith: bounded contexts, DB tables, traffic patterns

Identify extraction candidates: high-change rate, clear boundaries, team ownership

Add instrumentation: trace every module call (basis for SLO baseline)

Phase 2: Strangle at the edges (month 3–9)

Extract peripheral services first (auth, notifications, image resizing)
Use API gateway to route: /auth → new service, everything else → monolith
Share the DB initially – acceptable short-term compromise

Phase 3: Extract core services (month 9–18)

Separate DB per service (Database Decomposition pattern)
Dual-write during transition → cut over → remove monolith path
Beware: distributed joins now need API calls or data denormalisation

Phase 4: Retire monolith (month 18–24)

Shell is empty – decommission

WHAT MAKES MIGRATIONS FAIL

- × Big bang rewrite (6 months, then nobody uses the new system)
- × Extracting too many services simultaneously (coordination overhead explodes)
- × Shared DB for too long (services can't evolve schemas independently)
- × No SLO baseline before extraction (can't tell if new service is worse)
- × Wrong first extraction (core transactional service → high risk)

Principal answer: The decision to migrate should be driven by pain, not hype. If the monolith deploys weekly with a 5-person team and has 99.9% uptime — don't migrate. Migrate when: deploy takes 2+ hours blocking all teams, a bug in billing affects the feed algorithm, or you need to scale one component 10× without scaling all. Team topology (Conway's Law) should drive service boundaries, not technical architecture.

Official docs

- [Martin Fowler: Microservices](#)

31. **Design a news feed / timeline system (like Twitter or Instagram). Handle fan-out at scale.**

scalability

🗣 Spoken answer · ~66 s

Requirements: users follow others and see a ranked feed of their posts, with low read latency, handling very high read volume and the celebrity problem. The core design choice is fan-out. Fan-out on write, push model: when a user posts, the system writes the post ID into the feed cache of every follower, for instance a Redis list per user. Reads are then very fast, a single cache lookup. The cost is write amplification: a user with 10 million followers triggers 10 million writes, delaying delivery and wasting work on inactive users. Fan-out on read, pull model: at read time, fetch recent posts from everyone the user follows and merge. Writes are cheap, but reads are slow and expensive for users following many accounts. The practical answer is a hybrid: push for normal users, since most have modest followings, and pull for celebrities, merging their recent posts into the feed at read time; also push only to active followers.

Likely follow-ups

- Why does fan-out on write fail for celebrities?
- How does the hybrid approach decide who gets pushed?

🚩 Feed fan-out



The core tension is **fan-out-on-write** (push) vs **fan-out-on-read** (pull). The right answer is a hybrid driven by follower counts.

Write path (user posts):

Post Service → append to user's own timeline (DB)

→ enqueue fan-out job (Kafka)

Fan-out Worker → for each follower, push post_id into their feed cache (Redis list)

Read path (user opens app):

Feed Service → read pre-computed feed from Redis (O(1), already merged)

→ hydrate post_ids → full posts (batch get)

FAN-OUT-ON-WRITE VS ON-READ

Fan-out-on-WRITE (push):

- + Reads are cheap (feed pre-computed)
- Writes expensive: a celebrity with 100M followers = 100M cache writes

Fan-out-on-READ (pull):

- + Writes cheap (just store the post once)
- Reads expensive: merge N followees' timelines on every open

HYBRID (WHAT TWITTER ACTUALLY DOES)

- Normal users (< ~10K followers): fan-out-on-write (push to followers)
- Celebrities (> threshold): fan-out-on-read (followers pull celeb posts at read time)
- Final feed = merge(pushed posts, pulled celeb posts) sorted by rank/time

Principal answer: The celebrity problem is the whole interview. Naively pushing a celeb's tweet to 100M feeds creates a multi-minute write storm and thundering-herd reads. The hybrid bounds write amplification. Mention feed ranking (ML score, not pure chronological) and that you store post_id references in the feed cache, not full posts — keeps the cache small and lets you re-rank cheaply.



Official docs

- [Go concurrency patterns: Pipelines and cancellation](#)
- [Microsoft: Competing Consumers pattern](#)

32. **Design a distributed cache (like Redis Cluster / Memcached). Cover sharding, eviction, and consistency.**

scalability

🗣 Spoken answer · ~79 s

Requirements for a distributed cache like Redis Cluster or Memcached: low-latency key-value access, horizontal scale, high availability, and bounded memory. Sharding: partition the key space across nodes. Redis Cluster divides keys into 16,384 hash slots, assigned to masters, with CRC16 of the key determining the slot; clients cache the slot map and are redirected with MOVED responses on changes; resharding moves slots between nodes online. Memcached typically relies on client-side consistent hashing. Replication: each master has replicas that take over on failure, via cluster consensus on node failure detection, accepting asynchronous replication and the potential loss of recent writes, which is fine for a cache but notable if used as a store. Eviction when memory is full: policies such as LRU, LFU, random, or TTL-based, applied approximately by sampling for efficiency; set maxmemory and choose allkeys-lru for general caches. TTLs expire entries, both lazily on access and with periodic sampling. Consistency: caches are typically eventually consistent with the source of truth. Patterns: cache-aside, where the application loads on miss and writes the cache; write-through; write-behind; invalidation on writes or short TTLs. Problems: hot keys, stampedes on expiry, and big keys.

Likely follow-ups

- How do hash slots simplify resharding?
- What is a hash tag used for?

A cache layer absorbs read load and cuts latency. At scale you need it distributed across nodes with predictable key placement.

```
Client → consistent-hash(key) → cache node
      (client-side routing OR proxy like twemproxy/Envoy)
```

```
Node A   Node B   Node C   Node D
[shard] [shard] [shard] [shard]
  ↑      ↑      ↑      ↑
  replica replica (for HA reads / failover)
```

KEY DESIGN DECISIONS

```
Sharding:    consistent hashing + vnodes → minimal reshuffle on node change
Eviction:    LRU (default) | LFU (better for skewed access) | TTL-based
Replication: primary-replica per shard; promote replica on failure
Consistency: cache-aside is eventually consistent – handle staleness explicitly
```

CACHING PATTERNS

```
Cache-aside (lazy): app reads cache → miss → read DB → populate cache
                    Most common. Risk: stampede on hot-key miss.
Write-through:      write to cache + DB synchronously. Strong-ish, slower writes.
Write-behind:        write cache, async flush to DB. Fast, risk of data loss.
```

- # Thundering herd / cache stampede
 - request coalescing (single-flight): only 1 request rebuilds a hot key
 - add jitter to TTLs so keys don't all expire together
- # Hot key (one key = huge traffic)
 - replicate the hot key across nodes, or local in-process cache (L1)
- # Cache penetration (queries for non-existent keys)
 - cache negative results / bloom filter to short-circuit

Interview tip: Always name the three failure modes — stampede, hot key, penetration — and a mitigation for each. Mention a two-tier cache (L1 in-process + L2 Redis) for the hottest keys to cut network hops. State your staleness budget: "a 30s stale feed is fine; a stale account balance is not."

Official docs

- [Kubernetes: Node-pressure eviction](#)
- [Microsoft: Cache-Aside pattern](#)
- [Microsoft: Sharding pattern](#)

33. Design a typeahead / autocomplete search system. Sub-50ms suggestions at scale.

scalability

Spoken answer · ~82 s

The same core design applies as for typeahead in general, with a stricter latency target of under 50 milliseconds, which pushes the design to keep everything in memory and close to the user. Data structure: a trie with the top suggestions precomputed at each node, so a lookup is a short walk plus a list read, taking microseconds. Compress with radix trees or finite state automata to fit large vocabularies in RAM. Rank offline using query frequency, recency, and personalisation signals, with an hourly or daily batch pipeline plus a streaming layer for trending. Serving: replicate the structure across many nodes, sharded by prefix when needed, and place serving nodes at the edge or in regions close to users to cut network time, as network latency often dominates the 50 milliseconds. Caching: edge and CDN caching of responses for short prefixes, which are enormously repetitive; browser caching with appropriate headers. Client side: debounce keystrokes by 50 to 100 milliseconds, cancel in-flight requests when a newer keystroke arrives, cache results for earlier prefixes, and reuse the previous results' filtering locally as the user types more characters. Protocol optimisations: keep-alive connections, HTTP/2 or HTTP/3, small payloads, and compressed responses.

Likely follow-ups

- Why does network time often dominate this latency budget?
- How do you swap in a new index without downtime?

Autocomplete must return ranked suggestions in < 50ms as the user types. Pre-computation and in-memory tries are the core.

```
Client (debounced, 100–200ms) → API → Suggestion Service
                                     ↓
                               In-memory Trie (per shard, by prefix)
                                     ↓ (offline)
                    Aggregation Pipeline ← query logs (Kafka)
                                     ↓
                               Build/rank tries nightly + hourly deltas
```

DATA STRUCTURE: RANKED TRIE

```
# Each trie node stores top-K completions for that prefix (pre-computed)
# "sys" → ["system design", "system call", "systemd"] cached AT the node
# Lookup = walk prefix (O(prefix length)) → return node's top-K. No sorting at query time.
```

RANKING & FRESHNESS

```
Rank by: query frequency (popularity) + recency + personalization
Freshness: trending terms via streaming counts; merge hourly deltas into base trie
Sharding: by first 1–2 chars of prefix → route "sy*" to shard owning 's'
```

Principal answer: Don't query a DB per keystroke. Pre-compute top-K per prefix and serve from RAM. Debounce client-side and cache popular prefixes at the CDN/edge. For typo tolerance, layer an edit-distance / fuzzy match (e.g. Elasticsearch completion suggester) behind the exact-prefix trie. Bound suggestion size and use HTTP caching headers on common prefixes.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

34. Design a global CDN. How does content get cached and invalidated worldwide?

scalability

🗣 Spoken answer · ~85 s

A global CDN caches content at edge locations worldwide. How content gets cached: the first request for an object at an edge misses; the edge fetches it from the origin, possibly via a regional cache or origin shield, stores it keyed by URL and chosen request headers, and serves later requests directly, according to the Cache-Control headers, TTLs, and ETags. Popular objects stay in cache by LRU-like eviction. Request coalescing prevents a flood of identical origin fetches. Routing: anycast or DNS steering sends users to a nearby healthy point of presence. Invalidation is the hard part in a global system. Approaches. Versioned or content-hashed URLs, such as `app.3f9a1c.js`, with a very long TTL: a new deployment uses a new URL, so no invalidation is needed, which is the most reliable method for static assets. TTL expiry with stale-while-revalidate and stale-if-error, which keep serving cached content while refreshing in the background. Explicit purge APIs: purge by URL, path prefix, or surrogate key or cache tag, which lets you invalidate all objects related to a product with a single call; the CDN propagates the purge to all edges, typically in seconds, using a control-plane messaging system. Soft purge marks content stale instead of deleting it.

Likely follow-ups

- Why are content-hashed URLs the most reliable invalidation strategy?
- What are surrogate keys?

A CDN serves content from edge PoPs near users, cutting latency and origin load. Key topics: routing, cache hierarchy, and invalidation.

```
User → DNS/Anycast → nearest Edge PoP
                        ↓ (cache miss)
                    Regional Shield / Mid-tier cache
                        ↓ (cache miss)
                    Origin (your servers / object store)
```

REQUEST ROUTING

Anycast: same IP announced from many PoPs; BGP routes to nearest
DNS-based: geo/latency DNS returns the closest edge IP
The edge tier absorbs the vast majority of requests for cacheable content

CACHING & INVALIDATION

Cache key: URL + select headers (Vary) + query params you choose
TTL: Cache-Control max-age / s-maxage; stale-while-revalidate for smoothness
Invalidation:
- TTL expiry (simplest, eventual)
- Purge API (push invalidation to all PoPs) – for urgent updates
- Versioned URLs (`/app.v3.js`) – best: immutable assets, never invalidate

- Dynamic content: cache micro-TTL or use edge compute (workers)
- Large files/video: range requests + segment caching (HLS/DASH chunks)
- Origin shield reduces origin load by funneling misses through a mid-tier

Interview tip: Push hard on invalidation — it's the famously hard part. Recommend versioned/immutable URLs for static assets (cache forever, zero invalidation) and reserve purge APIs for emergencies. Mention stale-while-revalidate so users never wait on a refresh, and that cache-hit ratio is the metric that determines whether the CDN is actually helping.

Official docs

- [Cloudflare Learning: What is a CDN?](#)
- [AWS Builders' Library: Caching challenges and strategies](#)
- [Microsoft: Cache-Aside pattern](#)

35. Design a notification system (push, email, SMS) delivering billions of messages reliably.

scalability

Spoken answer · ~87 s

Requirements: deliver billions of push, email, and SMS messages reliably, honouring user preferences, with deduplication, retries, and tracking. Architecture. An ingestion API accepts notification requests from internal services and campaign systems, validates them, checks opt-outs and preferences, applies per-user and per-tenant rate limits, and writes events to a durable log, such as Kafka, partitioned for parallelism, with priority tiers so password resets bypass bulk campaigns. A template and personalisation service renders content in the user's language. Channel-specific workers consume from topics and deliver through providers: APNs and FCM for push, email service providers for email, and SMS gateways, each with its own rate limits and failure behaviour; multiple providers per channel allow failover and cost optimisation. Reliability for billions of messages: at-least-once delivery from the queue, with idempotency keys to avoid duplicates; retries with exponential backoff and jitter and a dead-letter queue; circuit breakers on provider failure; the transactional outbox pattern so notifications are not lost when the originating transaction commits. Scale: horizontally scaled consumers, batching API calls to providers, and for broadcast campaigns, a fan-out pipeline that reads audience segments in chunks rather than materialising billions of messages at once. Tracking: delivery receipts and webhooks, bounce and complaint handling, invalid token cleanup, and analytics on opens and clicks.

Likely follow-ups

- Why use idempotency keys with at-least-once delivery?
- How do you avoid materialising billions of messages at once?

A notification platform must be multi-channel, deduplicated, rate-limited per user, and resilient to provider failures.

```

Producers (services) → Notification API → Kafka (ingest)
    ↓
Processor: template render + user prefs + dedup + rate-limit + scheduling
    ↓
Channel queues: [push] [email] [SMS] (per-channel workers)
    ↓
Provider adapters: APNs/FCM | SES/SendGrid | Twilio
    ↓
Delivery tracking + retries (DLQ) → status store

```

CORE REQUIREMENTS

```

Dedup:      idempotency key per (user, event) → drop duplicates
Preferences: per-user channel opt-in, quiet hours, frequency caps
Rate limit:  don't spam a user — cap N notifications/hour/channel
Prioritize:  OTP/security (high) vs marketing (low) in separate queues
Reliability: at-least-once delivery + retries with backoff + DLQ

```

FAN-OUT & SCHEDULING

```

# Bulk campaign to 50M users → chunk into batches, spread over time window
# Scheduled notifications → time-bucketed store, worker polls due buckets
# Provider failover → if APNs errors spike, retry / circuit-break to backup path

```

Principal answer: Separate transactional (OTP, alerts — low latency, high priority) from bulk/marketing (throughput-oriented) into different queues and worker pools so a marketing blast can't delay a login OTP. Make the pipeline at-least-once with idempotent delivery; track per-message state for observability. Respect user prefs and frequency caps before render to avoid wasted provider cost.



Official docs

- [Microsoft: Competing Consumers pattern](#)

36. **Design a ride-sharing / location-matching system (like Uber). Match riders to nearby drivers.**

scalability

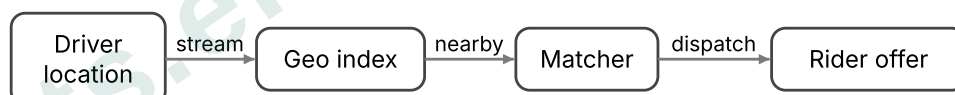
🗣 Spoken answer · ~88 s

Requirements: riders request a ride, the system finds nearby available drivers, matches them quickly, and tracks locations in real time, at the scale of millions of drivers sending location updates every few seconds. Location ingestion: drivers send GPS updates every 3 to 5 seconds, which at a million drivers is hundreds of thousands of writes per second; handle through a streaming pipeline, writing the latest position to an in-memory geospatial store, as the current location is what matters for matching, with history archived separately. Spatial indexing: divide the map into cells using geohash, Google S2, or Uber's H3 hexagonal grid; each driver belongs to a cell, and a proximity search queries the rider's cell and its neighbours, with the radius expanded progressively. Redis geo commands or in-memory sharded indexes by region provide fast lookups. Matching service: when a ride is requested, find candidate drivers in nearby cells, filter by availability and vehicle type, rank by ETA computed with routing and traffic data, not just straight-line distance, and send offers to the best candidates, with timeouts and fallback to the next. Use a dispatch approach that optimises globally, batching requests over a short window to reduce overall wait. Ensure a driver isn't assigned two rides, using atomic state transitions or locks per driver.

Likely follow-ups

- Why use geohash, S2, or H3 cells?
- How do you avoid assigning a driver to two rides?

🚗 Ride matching



The hard parts are real-time geospatial indexing of moving drivers and low-latency matching under high update rates.

Driver app → location pings (every ~4s) → Location Ingest (Kafka)

↓

Geospatial Index (Redis GEO / QuadTree / S2 / H3 cells) – updated continuously

↑ query "drivers near (lat,lng)"

Rider request → Matching Service → rank candidates → dispatch → Trip Service

GEOSPATIAL INDEXING

Geohash / S2 / H3: map earth into cells; index drivers by cell id

→ "find nearby" = look up rider's cell + neighbor cells

Redis GEO (sorted set on geohash) gives radius queries out of the box

Drivers move constantly → updates dominate; keep the index in memory

MATCHING

Candidate set = drivers in rider's cell + adjacent cells, status=available
Rank by: ETA (road network, not straight-line), driver rating, fairness
Dispatch: offer to top driver; timeout → next; lock driver during offer
Surge: per-cell supply/demand ratio → dynamic pricing signal

Interview tip: Emphasize that this is write-heavy (millions of location pings/sec) — the geo index must handle high update throughput, so partition by region/cell. Use ETA via a routing engine, not Euclidean distance. Mention that matching must lock a driver atomically to avoid double-dispatch, and that you'd shard the whole system by geography since a NYC rider never matches a London driver.

 **Start here (official docs)**

- [Azure Architecture Center: cloud design patterns](#)

37. Design a distributed job scheduler / cron system. Run millions of scheduled tasks reliably.

scalability

 Spoken answer · ~86 s

Requirements: run millions of scheduled tasks, one-off and recurring, at the right time, reliably, and exactly once in effect. Data model: jobs stored in a durable database with schedule definitions, such as cron expressions, next run time, payload, owner, and retry policy. Scheduling mechanism: instead of scanning the whole table, maintain an index on next run time, and have scheduler instances poll for jobs due in the next short window, such as the next minute, or use time-bucketed queues or a hierarchical timing wheel in memory. Shard the schedule across multiple scheduler nodes, partitioned by job ID hash, with each shard owned by one active scheduler via leader election or a lease, to avoid duplicate triggers. When a job is due, the scheduler enqueues a task into a work queue with an execution ID; workers pull it, execute it, and report completion. Reliability: at-least-once triggering with idempotent execution using the execution ID or deduplication keys; leases and heartbeats so jobs from crashed workers are retried; retry with backoff; dead-letter queues; and updating the next run time transactionally after enqueueing to avoid missing or double-firing after failover. Handle missed runs after downtime with a policy: run immediately, skip, or catch up. Support time zones and daylight saving carefully.

Likely follow-ups

- Why avoid scanning the whole job table?
- How do you prevent double triggers after a scheduler failover?

A scheduler must trigger jobs at the right time, exactly-once-ish, survive node failures, and scale to millions of schedules.

```
Schedule Store (DB)  → Scheduler (leader-elected / partitioned)
(cron expr, next_run)    ↓ poll "due" jobs each tick
                        Dispatch → Task Queue (Kafka/SQS)
                        ↓
```

Workers (idempotent execution)

↓

Status store + retry/DLQ

FINDING DUE JOBS AT SCALE

```
# Don't scan all jobs every second.  
# Time-bucketed / sorted index by next_run_time (e.g. Redis ZSET, or DB index)  
# Each tick: SELECT jobs WHERE next_run <= now LIMIT N → enqueue → compute next_run  
# Partition the schedule space across scheduler nodes (by hash of job_id)
```

RELIABILITY & CORRECTNESS

Exactly-once trigger: claim job atomically (compare-and-set next_run) before dispatch
Leader election / partition ownership: avoid two schedulers firing the same job
Missed runs (downtime): catch-up policy – run once vs skip vs run-all
Idempotent workers: dedup by (job_id, scheduled_time) so retries are safe

Principal answer: True exactly-once is impossible across failures, so aim for exactly-once *scheduling* (atomic claim of the time slot) + idempotent execution. Separate the scheduler (decides WHEN) from workers (do the WORK) via a durable queue, so slow jobs never delay scheduling. Define the catch-up policy explicitly — for a billing job you run missed slots; for a cache-refresh you skip them.



Official docs

- [Apache Kafka documentation](#)

38. **Design a real-time chat / messaging system (like WhatsApp). Cover delivery, ordering, and presence.**

scalability

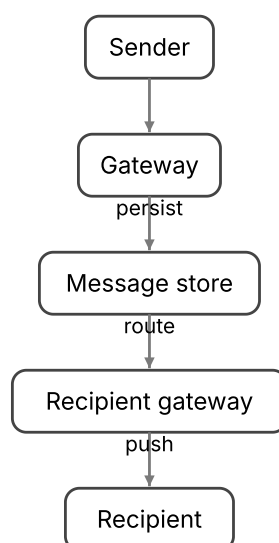
📢 Spoken answer · ~81 s

Requirements: one-to-one and group chat, reliable delivery, ordering, presence, multi-device sync, and low latency for hundreds of millions of users. Connections: clients keep persistent WebSocket or similar connections to chat gateway servers, which authenticate users and track user-to-gateway mappings in a distributed registry. Send flow: the sender's gateway assigns the message a unique ID and a per-conversation sequence number, persists it, and routes it to the recipient's gateway through a message bus if the recipient is online, or stores it for later delivery and sends a push notification if offline. Delivery guarantees: clients send acknowledgements; the server retries until acknowledged; message IDs enable deduplication, giving effectively exactly-once display. Receipts: sent, delivered, and read states flow back as small messages. Ordering: sequence numbers per conversation, assigned by a single owner for that conversation, such as the partition owner in Kafka keyed by conversation ID, let clients order messages and detect gaps, requesting missing ones. Storage: a write-optimised, horizontally scalable store such as Cassandra or HBase, partitioned by conversation, with recent messages cached; media goes to object storage with a CDN. Group chats: fan-out on the server for small groups, with limits on size, delivering per member.

Likely follow-ups

- How do sequence numbers help clients detect missing messages?
- Why limit presence subscriptions?

🚀 Chat delivery



Chat needs persistent connections, guaranteed delivery, message ordering, and presence — at billions of connections.

Client $\rightleftharpoons$ (WebSocket) $\rightleftharpoons$ Connection/Gateway Servers (stateful, hold the socket)

↓

```
Message Service → persist (per-conversation store)
↓
Routing: find recipient's gateway → push
↓ (offline)
Store + push notification (APNs/FCM)
```

DELIVERY & STATE MACHINE

```
States: sent → delivered → read (the double ticks)
Online: push over recipient's live WebSocket
Offline: persist, send push notification, deliver on reconnect (sync since last_seen)
Ack flow: client acks receipt → server marks delivered → notify sender
```

ORDERING & STORAGE

```
Ordering: per-conversation sequence number (monotonic), not global clock
Storage: partition by conversation_id; messages are append-only, time-ordered
        → Cassandra/HBase fit (write-heavy, wide rows per conversation)
Connection registry: which gateway holds which user's socket (Redis)
```

PRESENCE (ONLINE/TYPING)

```
Heartbeat over the socket → TTL key in Redis (e.g. 30s)
Missed heartbeats → mark offline. Don't broadcast presence to all contacts;
fan-out presence only to users currently viewing the chat (bounded).
```

Interview tip: The stateful-connection layer is the crux: millions of long-lived WebSockets need a registry mapping user → gateway so you can route a message to the right box. Use per-conversation sequence numbers for ordering (global ordering doesn't scale). For E2E encryption, mention the server only routes ciphertext. Presence is deceptively expensive — bound the fan-out.

Official docs

- [RFC 6455: The WebSocket Protocol](#)

39. **Design a payment / transaction system. Guarantee correctness with money on the line.**

data

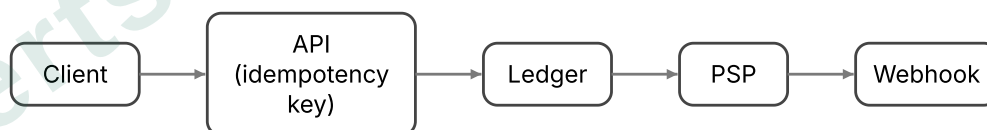
Spoken answer · ~86 s

With money involved, correctness outranks latency, so the design emphasises consistency, auditability, and idempotency. Core principles. Idempotency: every payment request carries a client-generated idempotency key; the server stores the key with the result and returns it on retries, so network failures and double clicks never double-charge. Ledger: record every movement as immutable double-entry journal entries, debit one account and credit another, in an append-only store, with balances derived from entries or maintained transactionally; never update or delete financial records; corrections are new reversing entries. Strong consistency: use a relational database with ACID transactions for the ledger, serialisable or carefully designed isolation levels, and constraints preventing negative balances where required; for scale, shard by account while keeping a transfer's participants consistent, or use sagas with careful compensation. State machine: payments move through explicit states, such as created, authorised, captured, settled, failed, refunded, with allowed transitions enforced. External providers: calls to card networks or banks are asynchronous and can time out, leaving an unknown outcome; handle with a pending state, retries using the same idempotency key, webhooks, and reconciliation jobs that compare internal records with provider settlement files daily to find discrepancies. Reliability: outbox pattern for events, durable queues, replication with synchronous commit in a region, and tested backups.

Likely follow-ups

- Why is an append-only ledger preferred?
- What is reconciliation and why does it matter?

Idempotent payment



Money demands strong consistency, idempotency, auditability, and reconciliation. "Eventually consistent" is unacceptable for balances.

```
Client → Payment API (idempotency key)
↓
Ledger Service (double-entry, ACID DB – Postgres/Spanner)
↓
Payment Orchestrator → external PSP (Stripe/bank) via saga
↓
Reconciliation job ← compare ledger vs PSP statements (daily)
```

NON-NEGOTIABLES

Idempotency: client sends idempotency_key; same key = same result, never double-charge
Double-entry ledger: every txn = debit + credit; balances are DERIVED, immutable log
ACID: use a transactional DB for the ledger; no eventual consistency here
Audit: append-only ledger, never UPDATE/DELETE a posted entry

DISTRIBUTED PAYMENT FLOW (SAGA)

1. Reserve/authorize funds (pending entry)
2. Call external PSP
3a. Success → capture (post entry)
3b. Failure/timeout → compensating action (reverse the reservation)
State machine per payment: PENDING → AUTHORIZED → CAPTURED | FAILED | REVERSED

HANDLING THE UNKNOWN

PSP call times out – did it succeed? Don't guess.
→ store as PENDING, query PSP status (idempotent), reconcile.
Never assume failure on timeout → that's how you double-pay or lose money.

Principal answer: Three pillars: (1) idempotency keys end-to-end so retries can't double-charge, (2) an immutable double-entry ledger where balances are computed from entries, never mutated in place, and (3) reconciliation against the PSP to catch drift. Use a saga with compensating transactions across the external boundary, and treat timeouts as "unknown" — verify, never assume. This is one place to explicitly reject eventual consistency.

Official docs

- [RFC 9110: HTTP Semantics \(status codes, methods\)](#)
- [Microsoft: Saga pattern](#)

40. **Design a video streaming platform (like YouTube/Netflix). Cover upload, transcoding, and playback.**

scalability

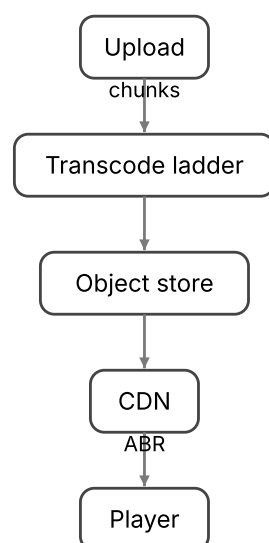
🗣 Spoken answer · ~85 s

Requirements: users upload videos, which are processed into multiple formats, and streamed smoothly worldwide on varied devices and networks. Upload: clients upload directly to object storage using resumable, chunked uploads through pre-signed URLs, avoiding the application servers; an upload completion event triggers processing via a message queue. Transcoding: a pipeline of workers splits the source into segments, so many workers transcode chunks in parallel, and encodes into multiple resolutions and bitrates, such as 240p to 4K, with codecs like H.264, VP9, or AV1, plus audio tracks, thumbnails, and subtitles; orchestrated by a workflow engine, with a DAG of tasks and retries; transcoding is CPU or GPU heavy, so autoscale worker pools, possibly with spot instances. Output formats are packaged for adaptive bitrate streaming, HLS or MPEG-DASH, with a manifest listing the renditions and short segments of two to ten seconds. Storage: processed segments in object storage, metadata such as title, owner, and status in a database, with search indexing separately. Playback: the player fetches the manifest and picks a bitrate according to measured bandwidth, switching adaptively; segments are served via a CDN, which caches popular content at the edge, with origin shielding; popular titles may be pre-positioned in ISP-embedded caches, as Netflix does.

Likely follow-ups

- Why split videos into segments for transcoding?
- How does adaptive bitrate streaming work?

🚧 Video pipeline



Two very different paths: a heavy async ingest/transcode pipeline, and a globally-cached, adaptive playback path.

UPLOAD :

Creator → upload (resumable, to object store) → Transcode Queue

- Transcoding workers: produce multiple renditions (240p...4K) + segments (HLS/DASH)
- store segments in object store → distribute to CDN

PLAYBACK:

Viewer → manifest (.m3u8) → CDN edge → video segments (adaptive bitrate)

TRANSCODING PIPELINE

- Split source into chunks → parallel transcode (massively parallel)
- Output ABR ladder: 240p/480p/720p/1080p/4K, each segmented (~2–10s chunks)
- Generate manifests (HLS m3u8 / DASH mpd) listing segments per bitrate
- This is async – user gets "processing" then "ready"

ADAPTIVE BITRATE (ABR) PLAYBACK

Player measures bandwidth → requests next segment at suitable bitrate
Drop quality on congestion, raise on recovery → no rebuffering
Segments are static files → perfectly CDN-cacheable (huge hit ratio)

SCALE LEVERS

Storage: object store (S3/GCS) for source + renditions; cold tiers for old/rare
Delivery: CDN does the heavy lifting – video is the ultimate cacheable static asset
Popularity skew: pre-warm CDN for anticipated hits; long-tail served from origin

Interview tip: Cleanly separate the asynchronous upload/transcode pipeline from the read-optimized playback path. Stress that video segments are static and thus ideal for CDN caching (that's how Netflix serves a third of internet traffic). Mention ABR for smooth playback, chunked parallel transcoding for fast processing, and storage tiering for the massive long tail of rarely-watched content.

Official docs

- [Cloudflare Learning: What is a CDN?](#)
- [Microsoft: Cache-Aside pattern](#)

41. **Design a distributed unique ID generator. Compare UUID, Snowflake, and DB ticket servers.**

data

🗣 Spoken answer · ~85 s

Distributed ID generation compares several approaches. UUIDs, version 4, are 128-bit random values generated locally with no coordination and negligible collision probability. They're simple and scale infinitely, but are large, not sortable, and cause index fragmentation and poor cache locality in B-tree databases. Time-ordered variants, UUIDv7 and ULID, put a timestamp first, so they sort roughly by creation time and index much better. Snowflake-style IDs pack a timestamp, a worker ID, and a sequence number into 64 bits: compact, roughly time-ordered, generated locally at very high throughput, but needing unique worker ID assignment and robust handling of clock drift or backwards clocks; they reveal creation time, which may matter for privacy. Database ticket servers, as at Flickr, use a dedicated database with an auto-increment column, or a pair of servers handing out odd and even numbers for redundancy, to produce small sequential IDs. They're simple and compact, but a central dependency and potential bottleneck, adding a network call to each ID; mitigated by handing out blocks of IDs to each application server, which then allocates locally, at the cost of gaps and loss of strict ordering. Choose by needs: strict global ordering is expensive and rarely necessary; for most systems, time-ordered, roughly sortable IDs suffice.

Likely follow-ups

- Why do random UUIDs hurt B-tree indexes?
- How do ticket servers avoid becoming a bottleneck?

Many systems need unique IDs without a single bottleneck. The classic answer is Snowflake; know the trade-offs of each approach.

OPTIONS COMPARED

UUID v4 (random):

- + Fully decentralized, zero coordination
- 128-bit, not sortable, poor DB index locality (random inserts)

DB auto-increment / ticket server:

- + Simple, sequential
- Single point of contention; sharded ticket servers add complexity

Snowflake (64-bit): [timestamp | machine_id | sequence]

- + Roughly time-sortable, no central coordination, compact
- Needs unique machine_id assignment + clock sync

SNOWFLAKE LAYOUT (64 BITS)

```
1 bit : unused (sign)
41 bits: timestamp (ms since custom epoch) → ~69 years
10 bits: machine/worker id → 1024 nodes
12 bits: per-ms sequence → 4096 ids per node per ms
→ ~4M ids/sec/node, time-ordered, no coordination on the hot path
```

THE CLOCK PROBLEM

```
# Clock skew / NTP step backwards → could repeat timestamps
# Mitigation: refuse to issue (wait) if clock moves backward;
#             monotonic clock; alert on large skew.
```

Principal answer: Default to Snowflake-style IDs: time-sortable (great for index locality and pagination), compact (64-bit fits in BIGINT), and coordination-free per node. Call out the two operational gotchas — assigning unique worker IDs (via Zookeeper/etcd or config) and handling clock moving backwards. Use UUIDs only when you truly need client-side generation with zero infrastructure; their index fragmentation hurts write-heavy tables.

Official docs

- [Twitter Snowflake \(ID generation\)](#)

42. Design a search engine / full-text search system. Cover indexing and ranking.

data

Spoken answer · ~87 s

A search engine has two halves: indexing and querying. Indexing pipeline: documents are collected, by a crawler or from a database through change data capture, parsed, and analysed: tokenised, lowercased, stripped of stop words, and stemmed or lemmatised. The core structure is an inverted index mapping each term to a postings list of the documents containing it, with term frequencies, positions for phrase queries, and field information. Postings are stored compressed with delta encoding and sorted by document ID for efficient intersection. The index is divided into immutable segments, merged in the background, as in Lucene, enabling fast writes and near-real-time search. Scale by sharding the index, by document or by term, usually by document, so each shard holds a subset and every query fans out to all shards, with results merged; replicas provide availability and read throughput. Querying: parse the query, apply the same analysis, retrieve matching postings by intersecting or unioning lists, and score candidates. Ranking: classic relevance with TF-IDF or BM25, which weigh term frequency against rarity and document length, combined with signals such as freshness, link-based authority like PageRank, popularity, personalisation, and increasingly learned ranking models, often in two stages: a cheap retrieval step, then re-ranking the top few hundred with a heavier model.

Likely follow-ups

- Why is the inverted index so efficient for search?
- What is the two-stage retrieval and ranking approach?

Full-text search at scale = an inverted index, distributed across shards, with relevance ranking. Elasticsearch is the canonical implementation.

INDEXING (write path):

Docs → Ingest → analyze (tokenize, lowercase, stem, stop-words)
→ build inverted index: term → [doc_ids + positions + freq]
→ shard by doc_id, replicate per shard

QUERY (read path):

Query → analyze same way → look up terms in inverted index
→ scatter to all shards → gather + rank → top-K → merge

INVERTED INDEX

term "design" → [doc1:tf=3, doc7:tf=1, doc9:tf=5, ...]
Query "system design" = intersect postings lists for both terms
Positions stored too → phrase queries ("system design" adjacent)

RANKING

TF-IDF / BM25: term frequency × inverse document frequency (classic relevance)
+ signals: freshness, popularity, click-through, personalization
+ optional: vector / semantic search (embeddings + ANN) for meaning, not just keywords

SCATTER-GATHER AT SCALE

Query hits ALL shards (term could be anywhere) → each returns local top-K
→ coordinator merges → global top-K. Latency = slowest shard (tail latency matters).
Replicas serve reads in parallel; index refresh is near-real-time, not instant.

Interview tip: Lead with the inverted index — it's the heart of the answer. Explain BM25 for ranking, and that search is scatter-gather (every shard queried, results merged), making tail latency the enemy. Distinguish indexing (write-heavy, async, eventually searchable) from querying (read-heavy, latency-sensitive). Bonus points for mentioning semantic/vector search as a modern layer over keyword search.

Official docs

- [PostgreSQL: Indexes](#)

43. Design a system for exactly-once message processing. Is it even possible?

reliability

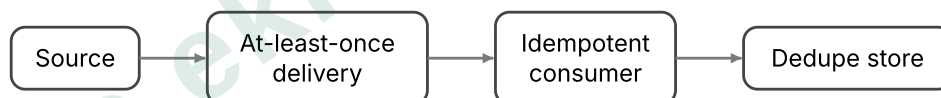
🗣 Spoken answer · ~81 s

Exactly-once processing in the strict sense, a message being delivered and processed exactly once across failures, is impossible in a general distributed system, because you can't distinguish a lost request from a lost acknowledgement; that's the two generals problem. What is achievable is exactly-once effect: at-least-once delivery combined with idempotent processing, or transactional processing within a system, so the observable outcome is as if each message were processed once. Techniques. Idempotent consumers: attach a unique message ID, and record processed IDs in the same database transaction as the business update, or make the operation naturally idempotent, such as setting a value instead of incrementing; on redelivery, the duplicate is detected and skipped. Transactional outbox and inbox patterns: write the state change and the outgoing event atomically in one local transaction, and a relay publishes it; the consumer writes to an inbox table with a unique constraint for deduplication. Kafka offers idempotent producers, which deduplicate retries within a partition, and transactions, which atomically write output messages and commit consumer offsets, giving exactly-once semantics for read-process-write pipelines entirely within Kafka; stream processors like Flink use checkpointing to achieve it with replayable sources and transactional or idempotent sinks.

Likely follow-ups

- Why is exactly-once delivery impossible in general?
- What does the inbox table do?

🔧 Effectively-once processing



True end-to-end exactly-once delivery is impossible in a distributed system (two-generals problem). What you achieve is **at-least-once delivery + idempotent processing = effectively-once**.

THE THREE DELIVERY GUARANTEES

At-most-once:	fire and forget. May lose messages. (metrics, logs)
At-least-once:	retry until acked. May duplicate. (most systems)
Exactly-once:	no loss, no dup. Impossible as pure delivery – achieved as at-least-once + dedup/idempotency.

ACHIEVING EFFECTIVELY-ONCE

1. Producer: idempotent producer (dedup by sequence number, e.g. Kafka)
2. Consumer: idempotent processing
 - dedup table: store processed message_id, skip if seen
 - or naturally idempotent ops (SET x=5, not x+=5)
3. Atomic: commit offset + side-effect together (transactional outbox)

```
# Problem: write to DB AND publish to Kafka atomically (no XA across them)
# Solution: in ONE DB txn, write business row + an "outbox" row
#           separate relay reads outbox → publishes to Kafka → marks sent
# Guarantees the event is published iff the DB change committed.
```

Principal answer: Start by stating exactly-once delivery is impossible, then pivot to how it's done in practice: at-least-once + idempotency keys + a dedup store, or naturally idempotent operations. Mention the transactional outbox for the dual-write problem (DB + message broker) and Kafka's idempotent producer + transactions. The mature signal is knowing it's an end-to-end property, not something one component provides.

Official docs

- [Apache Kafka documentation](#)

44. How do you handle database sharding? Cover shard keys, resharding, and cross-shard queries.

data

Spoken answer · ~85 s

Sharding splits data across nodes, and doing it well hinges on the shard key, resharding, and handling cross-shard operations. Shard key: choose a key with high cardinality and even distribution, that appears in most queries so they hit a single shard, and that isn't monotonic. For a multi-tenant app, tenant ID is common; for users, user ID. Avoid keys like timestamp or country with skew, which cause hotspots. Sometimes use a compound or hashed key to balance distribution. Mapping strategy: hash-based with consistent hashing, range-based, or directory lookup; I prefer many logical shards, say 1,024, mapped to fewer physical nodes, so resharding means moving whole logical shards instead of rehashing keys. Resharding: add nodes and migrate shards online. Steps: copy data to the new location while the old continues serving, using snapshot plus change capture to catch up, then switch routing atomically, verify, and delete from the old; use double writes or a proxy during transition; throttle migration to protect production. Cross-shard queries: scatter-gather across shards and merge is expensive and increases tail latency, so design data models so that common queries are single-shard; maintain denormalised copies or secondary indexes, global indexes in a separate store, or a search system for non-key queries.

Likely follow-ups

- Why choose many logical shards over few?
- How do you query data not tied to the shard key?

Sharding splits data horizontally across DBs when one node can't hold the data or handle the load. The shard key choice makes or breaks it.

SHARDING STRATEGIES

Range-based: shard by key range (A–M | N–Z)
+ range queries easy - hotspots if keys skew (sequential ids)
Hash-based: shard by hash(key)
+ even distribution - range queries hit all shards
Directory/lookup: explicit key→shard map (flexible, extra hop)
Geo/entity: shard by region or tenant (data locality)

CHOOSING THE SHARD KEY

Good key: high cardinality, even access, matches query patterns
Bad key: low cardinality (gender), monotonic (timestamp → hotspot on newest shard)
Example: shard by user_id → user's data co-located, queries scoped to one shard

THE HARD PROBLEMS

Cross-shard queries: scatter-gather (slow) – design schema to avoid them
Cross-shard transactions: 2PC (slow/fragile) or saga – avoid if possible
Resharding: use consistent hashing / vnodes so adding a shard moves minimal data
Celebrity/hot shard: a single popular key overwhelms one shard → sub-partition it

Interview tip: Spend your time on shard-key selection — it determines whether queries stay single-shard (fast) or become scatter-gather (slow). Warn against monotonic keys (they hotspot the newest shard). For resharding, consistent hashing minimizes data movement. State plainly: the goal is to make the common queries hit exactly one shard, and to denormalize so you rarely need cross-shard joins.



Official docs

- [Microsoft: Sharding pattern](#)

45. **Design a distributed locking system. When do you need one and what are the pitfalls?**

reliability

🗣 Spoken answer · ~86 s

A distributed lock gives mutual exclusion across processes on different machines, so only one worker performs a critical action at a time. When do you need one: protecting a resource that lacks its own concurrency control, such as ensuring one instance runs a scheduled job, preventing concurrent modification of an external system, or leader election. First ask whether you can avoid one: use database transactions, optimistic concurrency with version numbers, idempotent operations, or queues with a single consumer per partition, which are simpler and safer. Implementations. Redis: SET key value NX PX ttl acquires with expiry; release with a Lua script that checks the owner token. The Redlock algorithm across several instances is debated. ZooKeeper or etcd: ephemeral nodes or leases built on consensus, offering stronger guarantees. Database row locks or advisory locks, simple if you already use the database. Pitfalls. Lease expiry and pauses: a client may hold the lock, stall due to a GC pause or network delay, the lock expires, another client acquires it, and then the first resumes, believing it still holds the lock, causing both to act. The fix is fencing tokens: the lock service issues an increasing token with each grant, and the protected resource rejects requests with older tokens.

Likely follow-ups

- What problem do fencing tokens solve?
- What can replace a distributed lock in many designs?

A distributed lock coordinates exclusive access to a resource across processes/nodes. It's subtle — naive implementations are unsafe.

WHEN YOU NEED IT

- Leader election (one active scheduler/worker)
 - Preventing duplicate work (only one node processes job X)
 - Guarding a non-idempotent external action
- # Often you can AVOID locks via idempotency or single-writer partitioning – prefer that.

IMPLEMENTATIONS

Redis (SET key val NX PX ttl): fast, simple

- single-instance: not fault-tolerant
- Redlock (multi-instance): debated safety under GC pauses/clock skew

etcd / Zookeeper: consensus-backed (Raft/Zab), lease + watch

- + safer, linearizable; - higher latency

DB row lock / advisory lock: simple if you already have the DB

THE CLASSIC PITFALLS

1. No TTL → holder dies → lock held forever (deadlock). Always expire.
2. TTL too short → lock expires mid-work → two holders.
3. Process pause (GC/STW) → lock expires → another acquires → BOTH act.
→ Fencing token: lock issues monotonic token; resource rejects stale tokens.
4. Releasing someone else's lock → check-and-delete must be atomic (Lua/CAS).

Principal answer: First ask whether you can design the lock away (idempotency, single-writer-per-partition). If you truly need one, use a consensus store (etcd/Zookeeper) with leases for safety-critical locks, and always pair it with a **fencing token** so a paused-then-resumed old holder can't corrupt the resource — the resource rejects any token lower than the highest it has seen. TTLs alone are not safe under process pauses.

Official docs

- [Microsoft: Leader Election pattern](#)

46. Design a feature flag / experimentation platform. Support gradual rollouts and A/B tests.

platform

Spoken answer · ~85 s

A feature flag and experimentation platform lets teams change behaviour without deploying, and measure the impact. Flag management: a control plane with a UI and API to define flags, targeting rules, percentage rollouts, environments, approvals, and audit logs. Delivery: SDKs download the ruleset, cache it in memory, and receive updates by streaming or polling, so each evaluation is a local, microsecond operation, with safe defaults if the service is unreachable. Assignment is deterministic: hash the user ID with the flag or experiment key into a bucket, so a user always sees the same variant, and a gradual rollout from 1 to 10 to 50 percent only adds users. Experiments add more: a hypothesis, a primary metric, guardrail metrics like errors and latency, traffic allocation, and mutual exclusion layers so overlapping tests don't contaminate each other. Every assignment logs an exposure event, which flows through a streaming pipeline into a warehouse, where metric computation joins exposures with outcomes. Statistics matter: pre-compute sample size, avoid peeking until significance, correct for multiple comparisons, and watch for sample ratio mismatch as a sign of bugs. Safety: automatic rollback when guardrails regress, flag owners, expiry dates, and cleanup to avoid stale flags. Separate rollout flags from experiments from permanent configuration.

Likely follow-ups

- Why log an exposure event instead of just the assignment?
- What is sample ratio mismatch?

Feature flags decouple deploy from release. The platform must evaluate flags fast, support targeting, and enable experiments.

Flag Config (dashboard) → Config Store → distributed to SDKs

↓

App (SDK) evaluates locally: rules + targeting + % rollout → on/off

↓
Exposure events → analytics → experiment results

EVALUATION MUST BE LOCAL & FAST

```
# Don't call a service per flag check (latency + SPOF).  
# SDK pulls/streams ruleset, evaluates IN-PROCESS (microseconds).  
# Streaming updates (SSE/poll) propagate changes in seconds.
```

TARGETING & ROLLOUT

```
Rules: by user attribute, segment, geo, app version  
Gradual rollout: hash(user_id) % 100 < rollout_pct → stable bucketing  
    (same user stays in-bucket as % grows – no flip-flapping)  
Kill switch: instant flag off → disable a bad feature without redeploy
```

EXPERIMENTATION (A/B)

```
Deterministic assignment: hash(user_id + experiment_id) → variant  
Log exposures + metrics → statistical significance analysis  
Guardrail metrics + automated rollback if a variant harms key metrics
```

Interview tip: The key insight is local in-process evaluation — flag checks happen millions of times, so they can't be network calls. Stress consistent bucketing via hashing so a user doesn't flip variants as you ramp the percentage. Mention the kill switch (the operational reason flags exist) and that exposure logging is what turns a flag into an experiment.

Official docs

- [Kubernetes: Deployments](#)
- [Martin Fowler: Feature Toggles](#)

47. Design an analytics / metrics ingestion pipeline. Handle billions of events per day.

data

🗣 Spoken answer · ~77 s

Requirements: ingest billions of events a day, which is tens of thousands per second on average and several times that at peak, with durability, low ingest latency, and flexible analysis.

Collection: lightweight client SDKs batch events and send them to a stateless, horizontally scaled ingestion API behind a load balancer, which validates against a schema, enriches with server-side data such as geo and timestamps, and immediately appends to a durable log like Kafka, partitioned by a well-distributed key. The ingestion tier does nothing slow, so it stays fast and returns quickly. Processing: a stream processor such as Flink or Spark Streaming deduplicates by event ID, handles late and out-of-order events using event time and watermarks, and computes real-time aggregates; in parallel, raw events land in object storage as partitioned columnar files, Parquet, forming the data lake and the source of truth for replays. Serving: an OLAP store such as ClickHouse, Druid, or Pinot ingests from Kafka for sub-second dashboards, and a warehouse handles heavy ad hoc analysis. This is the lambda or kappa style trade-off: one streaming path with replay versus separate batch correction.

Likely follow-ups

- How do you handle late-arriving events?
- Why keep raw events in object storage?

High-volume event analytics splits into a real-time path (fast, approximate) and a batch path (complete, accurate) — the classic lambda/kappa choice.

```
Producers → Collector (validate, enrich) → Kafka (buffer)
├─ Stream processor (Flink/Spark) → real-time aggregates → fast store
└─ Batch / data lake (object store) → warehouse (BigQuery/Snowflake) → reports
    ↓
Dashboards / OLAP queries
```

INGESTION AT SCALE

Buffer with Kafka → absorbs spikes, decouples producers from processors
Schema + validation at edge; drop/dead-letter malformed events
Partition by event-type or tenant for parallel consumption
Sampling for ultra-high-volume, low-value streams

STORAGE BY ACCESS PATTERN

Time-series DB (Prometheus/InfluxDB): operational metrics, recent, fast
Columnar warehouse (BigQuery/Redshift/ClickHouse): ad-hoc analytics over history
Data lake (Parquet on S3): cheap raw retention, reprocessing
Pre-aggregation / rollups: hourly/daily cubes for common queries

LAMBDA VS KAPPA

Lambda: separate batch (accurate) + speed (fast) layers, merge at query
Kappa: single stream-processing path, reprocess by replaying the log (simpler)

Principal answer: Separate by access pattern: real-time approximate aggregates (stream processor → fast store) for live dashboards, and a batch path into a columnar warehouse for accurate historical analysis. Kafka is the shock absorber. Use pre-aggregation/rollups so dashboards don't scan raw events. Mention Kappa (replayable log, one code path) as the modern simplification over classic Lambda.

Official docs

- [Apache Kafka documentation \(streaming ingestion\)](#)

48. Design a system to handle the 'thundering herd' and cache stampede problems.

reliability

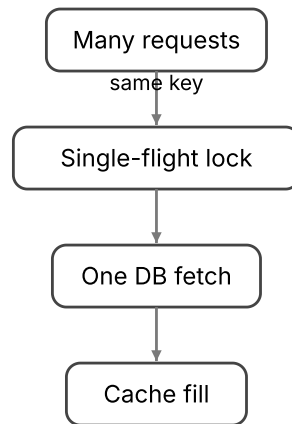
Spoken answer · ~86 s

A cache stampede, or thundering herd, happens when a popular cache entry expires or is evicted and many requests simultaneously miss and hit the backend to recompute it, which can overload the database and cascade. Related problems are a cold cache after a restart, and penetration, where requests for non-existent keys always miss. Defences. Request coalescing, or single flight: only one request per key recomputes while the others wait for its result, within a process or via a distributed lock with a short timeout. Stale-while-revalidate: serve the expired value while one background refresh runs. Probabilistic early expiration: each request has a small, increasing chance of refreshing before the TTL, spreading recomputation. TTL jitter: add random variation to expiry times so keys that were loaded together don't expire together. Pre-warming and background refresh for known hot keys, and warming a new cache before shifting traffic. Negative caching with short TTLs, and Bloom filters, to stop penetration. Protect the backend as the last line: rate limiting, load shedding, circuit breakers, and bounded concurrency, so even a herd can't take the database down. Also use layered caches, local plus distributed, so one layer absorbs misses from the other. Monitor miss spikes and backend latency, and load test expiry scenarios.

Likely follow-ups

- What is single-flight request coalescing?
- Why add jitter to TTLs?

Stampede protection



A thundering herd happens when many clients hit the origin simultaneously — typically after a cache expiry, a cold start, or a recovery. Left unmanaged, it can cascade into an outage.

WHERE HERDS FORM

- Hot cache key expires → thousands of misses stampede the DB at once
- Service restarts cold → every request rebuilds expensive state
- Recovery after outage → all clients retry simultaneously

MITIGATIONS

1. Request coalescing (single-flight):
only ONE request recomputes a missing key; others wait for its result.
2. TTL jitter:
add randomness to expiries so keys don't all expire at the same instant.
3. Early/probabilistic refresh:
refresh hot keys slightly BEFORE expiry, in the background.
4. stale-while-revalidate:
serve the stale value while one worker refreshes async.
5. Exponential backoff + jitter on client retries (prevents retry storms).

DEFENSIVE BACKPRESSURE

- Concurrency limits / load shedding at the origin (protect the DB)
- Circuit breaker: stop hammering a struggling dependency
- Queue + rate-limit the rebuild work

Interview tip: Name the specific pattern for each cause: single-flight/coalescing for hot-key misses, TTL jitter so keys don't expire in lockstep, stale-while-revalidate so users never block on a refresh, and exponential backoff *with jitter* for retry storms (jitter is the part people forget). Top it with load shedding so the origin sheds excess load rather than collapsing.

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Caching challenges and strategies](#)

49. Design a time-series database / monitoring data store. Why not just use PostgreSQL?

data

Spoken answer · ~88 s

A time-series database stores timestamped measurements, mostly appended in time order, queried by time range and aggregated. Why not just PostgreSQL? You can, up to a point, and for modest volumes with TimescaleDB it is a fine choice. But at high ingest, a general-purpose row store struggles: every sample is a row with tuple overhead and index maintenance, so write throughput and storage costs are poor; a typical time-series engine achieves one to two bytes per sample through delta-of-delta timestamp and XOR value compression, versus tens of bytes in a row store. Design of a purpose-built store: an in-memory buffer with a write-ahead log, flushed into immutable time-partitioned blocks; columnar or chunked layout, so a range scan reads contiguous compressed data; an inverted index from labels to series for fast filtering; automatic retention by dropping whole old partitions, instead of expensive deletes; downsampling and rollups to keep long-term data cheap; and query functions like rate, windowed aggregation, and interpolation. Challenges are high cardinality, out-of-order writes, and horizontal scaling by sharding on series and time. Trade-offs: specialised stores have limited joins and transactions, so keep business data relational. Choose Postgres with an extension for moderate scale and SQL familiarity, and a dedicated engine like Prometheus, VictoriaMetrics, or InfluxDB for heavy metrics workloads.

Likely follow-ups

- How does retention work without row deletes?
- When is TimescaleDB enough?

Time-series workloads (metrics, IoT, monitoring) have a distinctive shape: massive append-only writes, time-ordered, recent-data-heavy reads, and aggregation queries. Purpose-built TSDBs exploit this.

WHY A GENERAL RDBMS STRUGGLES

- Write volume: millions of points/sec overwhelm B-tree index updates
- Storage: raw rows are huge; TS data compresses ~10x with the right encoding
- Retention: old data must auto-expire; deletes in RDBMS are expensive
- Query shape: "avg over 5-min buckets, last 24h" wants columnar + downsampling

TSDB DESIGN CHOICES

Storage: columnar + time-partitioned chunks (write to newest, read recent)
Compression: delta-of-delta on timestamps, XOR on float values (Gorilla)
Downsampling: roll up raw → 1m → 1h → 1d as data ages

Retention: drop whole old partitions (cheap), tiered storage for history
Cardinality: the killer – each unique label combo = a series; bound it!

THE CARDINALITY TRAP

```
# label like user_id or request_id → millions of series → memory blowup
# Rule: labels must be low-cardinality (region, service, status_code)
# High-cardinality dimensions belong in logs/traces, not metric labels.
```

Principal answer: Time-series data is append-mostly, time-ordered, and read recent-first — so a TSDB uses time-partitioned columnar chunks, aggressive compression (Gorilla-style delta + XOR), downsampling as data ages, and partition-drop retention. The interview gotcha is cardinality: warn that putting a high-cardinality field (user_id, UUID) in a metric label explodes the series count and OOMs the DB — that's what kills Prometheus in production.

Official docs

- [Prometheus: Metric and label naming](#)
- [Prometheus: Storage \(TSDB\)](#)

50. Design a collaborative document editor (like Google Docs). Handle concurrent edits.

scalability

Spoken answer · ~88 s

Collaborative editing needs concurrent edits from many users to converge to the same document without losing intent. Two main techniques. Operational transformation, used in early Google Docs: each edit is an operation like insert character at position five; when two users edit concurrently, a central server transforms operations against each other so positions shift correctly, and orders them canonically. It works well but the transformation functions are intricate and generally rely on a central server. CRDTs, conflict-free replicated data types, such as sequence CRDTs like Yjs and Automerge, give every character a unique, ordered identifier so operations commute; replicas can apply edits in any order and still converge, which suits peer-to-peer and offline editing, at the cost of metadata overhead and the need for garbage collection. Architecture: clients connect over WebSocket to a collaboration server that is stateful per document, so route all editors of one document to the same node, using consistent hashing on the document ID; apply edits optimistically on the client for instant feedback, then reconcile with server acknowledgements; broadcast changes to others. Persistence: append operations to a log, snapshot periodically, and store snapshots in a database or object store, so loading means snapshot plus recent operations, and enabling version history. Presence: cursors and selections as ephemeral, throttled messages.

Likely follow-ups

- Why must editors of one document reach the same server in OT?
- What are the trade-offs of CRDTs?

Real-time collaboration needs concurrent edits to converge to the same state without locking. Two families: OT and CRDT.

```
Client A ⇌ WebSocket ⇌ Collab Server ⇌ WebSocket ⇌ Client B
(local edit applied instantly = optimistic)
  ↓ send op
Server orders/transforms ops → broadcasts → clients converge
  ↓
Persist op log + periodic snapshots
```

OT VS CRDT

Operational Transformation (OT):

Transform concurrent ops against each other so they apply consistently.

Used by Google Docs. Needs a central server to order/transform. Complex but compact.

CRDT (Conflict-free Replicated Data Type):

Data structure mathematically guarantees convergence regardless of order.

Peer-friendly, offline-friendly. Metadata overhead (tombstones, ids).

KEY MECHANICS

Optimistic local apply: editor feels instant; reconcile when server confirms

Ordering: server assigns a total order (or version vectors for CRDT)

Presence/cursors: broadcast lightweight cursor positions to collaborators

Persistence: append ops to a log; snapshot periodically to bound replay

Interview tip: Frame the core problem as *conflict resolution without locking*, then contrast OT (server-centric, used by Google Docs) vs CRDTs (convergence by construction, great for offline/P2P). Mention optimistic local application for responsiveness, an op log + snapshots for persistence, and that cursor/presence is a separate low-stakes broadcast. Picking one and justifying it (OT for a central server, CRDT for offline-first) shows depth.

Official docs

- [RFC 6455: The WebSocket Protocol](#)

51. **How do you design for high availability? Walk through redundancy at every layer.**

reliability

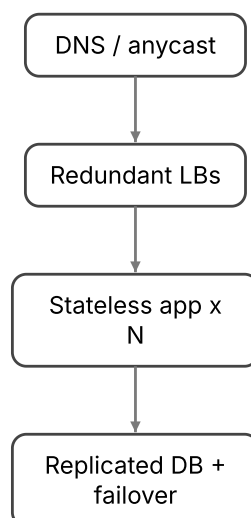
🗣 Spoken answer · ~86 s

High availability means eliminating single points of failure at every layer and automating recovery. Walk down the stack. DNS and edge: multiple DNS providers or anycast, a CDN, and global load balancing with health checks to steer around failures. Network: redundant load balancers across zones, redundant links, and no single firewall or gateway. Compute: stateless application instances spread across at least three availability zones, in autoscaling groups or Kubernetes with pod disruption budgets and anti-affinity, with health checks that replace unhealthy instances automatically. Data: replication with automatic failover, such as primary plus synchronous standby in another zone, or a quorum-based system; backups and tested restores; avoid shared single storage. Dependencies: timeouts, retries with backoff, circuit breakers, bulkheads, and graceful degradation for non-critical features, because availability multiplies across serial dependencies. Capacity: headroom to absorb the loss of a zone, so run N plus one or N plus two, and load shedding for overload. Region level: multi-region active-active or active-passive, depending on the target. Change management: canary releases, automatic rollback, and feature flags, since most outages follow changes. Operations: SLOs, error budgets, monitoring, runbooks, on-call, chaos experiments, and blameless postmortems. State the target, such as 99.95 versus 99.99 percent, because each extra nine increases cost and complexity sharply.

Likely follow-ups

- Why does availability multiply across serial dependencies?
- What headroom do you keep for zone failure?

🚧 HA at every layer



HA means no single point of failure and graceful handling of component loss. You design redundancy at every tier and target a measurable availability SLO.

AVAILABILITY MATH

99.9% (three 9s) = 8.7 hours downtime/year = 43 min/month
99.99% (four 9s) = 52 minutes/year = 4.3 min/month
99.999% (five 9s) = 5.3 minutes/year
Serial dependencies MULTIPLY: two 99.9% services in series = 99.8%
Redundant (parallel) components ADD nines.

REDUNDANCY AT EACH LAYER

DNS: multiple providers, health-checked failover
LB: redundant LBs, multi-AZ
App tier: stateless, N+1 instances across AZs, auto-scaling
Data tier: primary + replicas, automated failover, multi-AZ
Region: multi-region (active-active or active-passive) for disaster
Dependencies: timeouts, retries, circuit breakers, fallbacks

ELIMINATING SPOFS

- No single instance, AZ, or region the whole system depends on
- Health checks + automated failover (manual failover is too slow)
- Test it: chaos engineering / game days to prove failover actually works

Interview tip: Start with the SLO target (it dictates cost — five 9s is wildly more expensive than three). Show you understand serial vs parallel availability math, then walk redundancy tier by tier. The mature point: redundancy you haven't tested isn't redundancy — mention game days / chaos testing to verify failover works before a real incident.

Official docs

- [Google SRE Book: Embracing Risk \(capacity & availability math\)](#)

52. Design a circuit breaker and explain graceful degradation under dependency failure.

reliability

🗣 Spoken answer · ~87 s

A circuit breaker wraps calls to a dependency and tracks failures; when failures cross a threshold, it opens and fails fast instead of waiting on a dying service, then probes recovery with a half-open state. That protects threads, connections, and queues in the caller from filling up, which is how a slow dependency turns into a cascading outage. But failing fast is only half the story; graceful degradation decides what the user experiences. Options when the circuit is open: return cached or stale data, such as the last known product recommendations; use sensible defaults, like omitting personalisation; hide or disable non-essential features, such as reviews or a recommendations carousel, while the core flow, browsing and checkout, keeps working; queue the work for later, as with sending an email or recording analytics; or route to a secondary provider. Classify dependencies as critical or non-critical up front, and design each non-critical one with a fallback. Supporting patterns: timeouts shorter than the caller's own deadline, bulkheads to isolate resource pools per dependency, retries with budgets and jitter, load shedding, and feature flags that can switch features off manually. Make degraded state visible with metrics and banners, and test it with fault injection or game days, because untested fallbacks fail when needed.

Likely follow-ups

- How do you decide which dependencies are non-critical?
- Why test fallbacks with fault injection?

A circuit breaker stops a service from repeatedly calling a failing dependency, preventing cascading failure and giving the dependency room to recover.

CIRCUIT BREAKER STATES

```
CLOSED:    calls pass through. Count failures.
            | failure rate > threshold
            ▼
OPEN:      calls fail fast (no call made). Return fallback immediately.
            | after cooldown timer
            ▼
HALF-OPEN: allow a few trial calls.
            ├── success → CLOSED (recovered)
            └── failure → OPEN (keep waiting)
```

WHY IT MATTERS

```
# Without it: dependency slows → threads pile up waiting → caller exhausts
#             its thread pool → caller fails too → cascade up the stack.
# With it: caller fails fast, sheds load off the sick dependency, stays alive.
```

GRACEFUL DEGRADATION

When a dependency is down, degrade – don't crash:

- serve stale cache instead of live data
- hide a non-essential feature (recommendations) but keep core flow
- default/empty response with a clear partial-result indicator

Example: product page loads even if the reviews service is down.

Principal answer: Pair the circuit breaker with timeouts (a slow dependency is as dangerous as a dead one), bulkheads (isolate thread pools per dependency so one can't starve the others), and a defined fallback per call site. The design goal is partial availability: a single failed dependency should degrade one feature, never take down the whole request path. That's the difference between an incident and an outage.

Official docs

- [Google SRE Book: Handling Overload](#)
- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Avoiding fallback in distributed systems](#)

53. **Synchronous vs asynchronous communication: when do you use each between services?**

tradeoffs

Spoken answer · ~88 s

Choose by whether the caller needs the result now, and how much coupling you can tolerate. Synchronous communication, REST or gRPC request-response, fits when the caller must have an answer to proceed, like validating a token, fetching a price, or reading a profile to render a page. It's simple, easy to debug and trace, and gives immediate feedback, but couples availability and latency: a slow or failing downstream directly hurts the caller, and chains of calls multiply latency and failure probability. Mitigate with timeouts, retries with backoff, circuit breakers, and caching. Asynchronous communication, through queues or event streams, fits when work can complete later, when multiple consumers care about an event, or when you need to absorb spikes: sending notifications, order fulfilment, analytics, and cross-service data propagation. It decouples services in time and availability, supports buffering and independent scaling, and enables retries, but adds complexity: eventual consistency, ordering, duplicate delivery requiring idempotency, harder debugging, and the need for correlation IDs and dead-letter handling. A practical pattern is a synchronous edge for user-facing requests that returns fast, such as accepted with 202, with asynchronous processing behind it and events published for downstream consumers. Use sync for queries and strongly dependent steps, async for commands that can be deferred and for notifications of facts.

Likely follow-ups

- Why do long synchronous call chains hurt reliability?
- What does returning 202 Accepted signal?

Choosing sync (request-response) vs async (messaging/events) per interaction is one of the most consequential architecture decisions.

SYNCHRONOUS (REST/gRPC, caller waits):

- + simple, immediate result, easy to reason about
- tight coupling, caller blocked, failure propagates, latency adds up
- use when: you need the result NOW to continue (read a user profile)

ASYNCHRONOUS (queue/event, fire-and-continue):

- + decoupled, resilient (buffer absorbs spikes), scalable
- eventual consistency, harder to trace, ordering/dedup concerns
- use when: work can happen later (send email, process upload, fan-out)

DECISION HEURISTICS

Need immediate response to proceed?	→ sync
Can it be processed later / in background?	→ async
Want to decouple producer/consumer scale?	→ async
Multiple consumers of one event?	→ async (pub/sub)
Long-running work (transcode, report)?	→ async + status polling

HYBRID: THE COMMON REAL PATTERN

```
# Sync ack + async work:
# POST /upload → 202 Accepted (sync) + enqueue processing (async)
# Client polls /status or gets a webhook/websocket update when done.
```

Interview tip: Don't pick a side globally — decide per interaction. The strongest answer is the hybrid: respond synchronously with an acknowledgment, then do heavy work asynchronously. Call out async's costs honestly (eventual consistency, harder debugging, need for idempotency) so it's clear you're not cargo-culting "async = scalable."



Official docs

- [Azure: Asynchronous messaging options](#)

54. Message queue vs event streaming (e.g. RabbitMQ/SQS vs Kafka). How do you choose?

tradeoffs

🗣 Spoken answer · ~85 s

Both move messages between services, but they model different things. A traditional message queue, such as RabbitMQ or SQS, is about work distribution: producers add tasks, competing consumers pull them, and each message is removed once acknowledged. It offers per-message routing, priorities, delayed delivery, dead-letter queues, and simple operations; consumers scale by adding workers. Ordering is limited and messages aren't replayable after consumption. An event streaming platform, such as Kafka, Pulsar, or Kinesis, is a durable, partitioned, append-only log. Messages are retained by time or size, not deleted on read; many consumer groups each read independently at their own offsets; ordering is guaranteed within a partition; and consumers can replay history. It handles very high throughput and suits event sourcing, change data capture, stream processing, and feeding multiple downstream systems. Costs: more operational complexity, partition planning, and consumer lag management; per-message acknowledgement and priority semantics are weaker. How to choose: use a queue when you have tasks to be processed once by a pool of workers, need per-message retry and delay semantics, or want the simplest managed option. Use a stream when multiple independent consumers need the same events, ordering per key matters, throughput is very high, or you need replay and reprocessing.

Likely follow-ups

- What does replay enable that a queue doesn't?
- How does Kafka guarantee ordering?

🗖 Queue vs log

Message queue	Event log
Consume and delete	Retain and replay
Per-message ack	Offset per consumer group
RabbitMQ, SQS	Kafka, Kinesis

Both decouple producers and consumers, but their models differ fundamentally — queues are about task distribution, logs are about durable replayable streams.

Traditional Queue (RabbitMQ, SQS):

- message consumed → removed from queue
- work distribution among competing consumers
- per-message ack, routing, priorities, DLQ
- jobs/tasks: "process this once, by someone"

Log/Stream (Kafka, Kinesis, Pulsar):

- append-only log; messages RETAINED (by time/size)
- consumers track their own offset; can replay
- ordered within a partition; high throughput
- events: "this happened; many consumers, replayable"

CHOOSE A QUEUE WHEN

- Task/work distribution (one job done once)
- Need per-message priority, complex routing, or visibility timeouts
- Don't need replay or multiple independent consumer groups

CHOOSE A LOG/STREAM WHEN

- Event sourcing, multiple consumers of the same events
- Need replay (reprocess history, bootstrap a new consumer)
- Very high throughput, ordered per key (partition)
- Stream processing / analytics pipelines

Principal answer: The deciding question is replay + multiple consumers. If many independent consumers need the same events, or you need to reprocess history, that's Kafka (the log is retained and each consumer owns its offset). If it's competing workers draining a task queue once, that's RabbitMQ/SQS. Mention Kafka's per-partition ordering and that it's not ideal when you need per-message priority or fine-grained routing.

Official docs

- [Amazon SQS](#)
- [Apache Kafka documentation \(streaming ingestion\)](#)
- [RabbitMQ documentation](#)

55. Design an API gateway. What responsibilities belong there vs in services?

networking

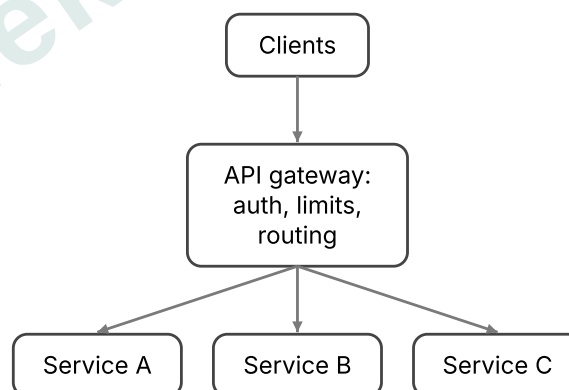
🗣 Spoken answer · ~88 s

An API gateway is the single entry point for clients, handling cross-cutting concerns so services don't each reimplement them. Responsibilities that belong in the gateway: routing requests to the right backend, TLS termination, authentication of tokens and coarse authorisation, rate limiting and quotas, request validation against schemas, protocol translation such as REST to gRPC, response caching, CORS, request and response transformation, API key management, logging, metrics, and tracing initiation, plus WAF integration and canary routing. It can also aggregate several backend calls into one for clients, a backend-for-frontend pattern, though keep that thin or have separate BFFs per client type. What should stay in services: business logic and domain rules, fine-grained authorisation that depends on data, data validation involving business invariants, and orchestration of complex workflows. Putting business logic in the gateway creates a distributed monolith bottleneck owned by a central team. Design concerns: the gateway is on the critical path, so it must be highly available, horizontally scaled, low-latency, and stateless; avoid it becoming a single point of failure; keep configuration declarative and versioned; protect against a noisy tenant. Distinguish north-south traffic, between clients and services, handled by the gateway, from east-west traffic between services, better handled by a service mesh. Options include Kong, Envoy-based gateways, and cloud gateways.

Likely follow-ups

- Why keep business logic out of the gateway?
- How is a gateway different from a service mesh?

🚩 API gateway



An API gateway is the single entry point for clients, handling cross-cutting concerns so individual services don't reimplement them.

Clients → API Gateway → [auth, rate-limit, routing, TLS, transform]

↓

Service A Service B Service C

↓

(service mesh handles service↔service)

GATEWAY RESPONSIBILITIES (EDGE CONCERNS)

- TLS termination, request routing / path-based dispatch
- AuthN (validate token) – coarse authZ
- Rate limiting / quota enforcement per API key
- Request/response transformation, protocol translation (REST→gRPC)
- Aggregation (BFF: combine several services for one client call)
- Observability: logging, tracing injection, metrics

WHAT STAYS IN SERVICES

- Business logic (always)
 - Fine-grained authorization (resource-level permissions)
 - Domain validation
- # Gateway = cross-cutting plumbing; services = domain. Don't leak logic into the gateway.

PITFALLS

- Gateway becomes a SPOF → run it redundantly, multi-AZ
- Fat gateway with business logic → "distributed monolith" anti-pattern
- Latency: it's an extra hop → keep it lean

Interview tip: Draw the line clearly: the gateway owns cross-cutting edge concerns (auth, rate limiting, routing, TLS), services own business logic and fine-grained authorization. Mention the Backend-for-Frontend variant for client-specific aggregation, and warn against the common failure mode — stuffing business logic into the gateway until it becomes a bottleneck and a deployment chokepoint.



Official docs

- [Amazon API Gateway](#)

Spoken answer · ~87 s

A load balancer spreads traffic across backends. Layer 4 balancers work on TCP or UDP connections using addresses and ports: very fast, protocol-agnostic, handle huge connection counts, and can use direct server return, but they can't see HTTP. Layer 7 balancers parse HTTP, so they can route by host, path, headers, or cookies, terminate TLS, retry, compress, authenticate, and give detailed metrics, at higher cost. Common deployment: an L4 tier, anycast or ECMP, in front of an L7 proxy tier. Algorithms. Round robin: simple and fair when requests are similar. Weighted round robin: for unequal capacity. Least connections: better when request durations vary, sending traffic to the least busy backend. Least response time or peak EWMA: adapts to latency. Power of two choices: pick two random backends and choose the less loaded, giving near-optimal balance with little coordination. IP hash or consistent hashing: sticky routing for caches and session affinity, with consistent hashing limiting disruption when backends change. Random is often surprisingly adequate. Supporting features: active and passive health checks, connection draining, slow start for new backends, and circuit breaking or outlier ejection. Avoid sticky sessions where possible by keeping services stateless. Make the balancer itself redundant, and consider global load balancing with DNS or anycast across regions.

Likely follow-ups

- Why is power of two choices effective?
- When would you use consistent hashing?

Load balancers distribute traffic across backends for scale and availability. The L4 vs L7 distinction and algorithm choice are the core of the answer.

L4 (transport): routes by IP:port, no payload inspection

- + very fast, protocol-agnostic, low overhead
- can't route by URL/header, no content awareness

L7 (application): understands HTTP – routes by path/host/header/cookie

- + smart routing, TLS termination, sticky sessions, rewrites
- more CPU per request

ALGORITHMS

Round robin:	simple, even – ignores backend load
Weighted RR:	bigger boxes get more traffic
Least connections:	send to least-busy backend (good for long-lived conns)
Least response time:	factor in latency
Consistent hashing:	same client/key → same backend (cache affinity, sticky)
IP hash:	session stickiness without cookies

HEALTH & RESILIENCE

Active health checks → eject unhealthy backends automatically
Graceful drain on deploy (connection draining)
Outlier detection: eject a backend that starts erroring
Multi-LB + DNS/anycast so the LB itself isn't a SPOF

Principal answer: Use L4 when you need raw throughput and protocol-agnostic routing (e.g. in front of L7 LBs, or for non-HTTP), and L7 for content-based routing, TLS termination, and canary/path splitting. Default to least-connections over round-robin for uneven workloads, and consistent hashing when backend cache affinity matters. Always pair with health checks + connection draining — a dumb LB sending traffic to a dead node is worse than no LB.

Official docs

- [Cloudflare Learning: OSI model](#)
- [Cloudflare Learning: Load balancing](#)

57. Push vs pull architectures: when do clients pull and when does the server push?

tradeoffs

Spoken answer · ~87 s

The question is who initiates data transfer. Pull: the client asks when it wants data. It's simple, works through firewalls and proxies, is cacheable, and the client controls its rate, but it wastes requests when nothing has changed and has latency up to the polling interval. Variants: short polling, and long polling, where the server holds the request open until data is available or a timeout expires. Push: the server sends data when something changes, through WebSockets, server-sent events, or mobile push notifications. It gives low latency and fewer wasted requests, but requires holding connections, handling reconnects, scaling stateful gateways, dealing with slow consumers, and fallbacks when networks block it. Choose pull when updates are infrequent, freshness within seconds or minutes is fine, the data is highly cacheable, or you need simplicity: dashboards refreshing every 30 seconds, batch consumers. Choose push when latency matters and events are sparse and unpredictable: chat, live scores, collaborative editing, alerts. Server-sent events suit one-way server-to-client streams; WebSockets suit bidirectional. In backend systems, the same trade-off appears: consumers pulling from Kafka control their own pace and get natural backpressure, whereas pushing to subscribers risks overwhelming them. A hybrid works well: push a lightweight notification that something changed, and let the client pull the details.

Likely follow-ups

- Why does pull give natural backpressure?
- What is the push-a-notification, pull-the-details pattern?

How does a client get new data — does it ask (pull) or does the server notify it (push)? This decision shapes latency, load, and complexity.

PULL (polling):

Client asks repeatedly: "anything new?"

- + simple, stateless, client controls rate
- wasteful (empty polls), latency = poll interval

PUSH (server → client):

Server sends when something changes

- + real-time, efficient (no empty checks)
- server holds connection state, harder to scale

PUSH MECHANISMS

Long polling: client request held open until data or timeout (easy, works everywhere)

Server-Sent Events: one-way server-client stream over HTTP (dashboards, feeds)

WebSocket: full-duplex persistent connection (chat, games, collab)

Webhooks: server-to-server push (callback URL)

CHOOSING

Low-frequency / tolerant of delay → polling (simplest)

Real-time, one-way (notifications, prices) → SSE

Real-time, two-way (chat, multiplayer) → WebSocket

Hybrid: pull for initial state + push for live deltas (very common)

Interview tip: Match the mechanism to the need: don't reach for WebSockets when SSE or even long-polling suffices — persistent connections are expensive to hold at scale (millions of open sockets need a stateful gateway tier). The common production pattern is pull-then-push: fetch initial state via a normal request, then subscribe to live deltas. Mention the scaling cost of stateful connections explicitly.



Official docs

- [Prometheus: When to use the Pushgateway](#)

58. **Design backpressure and load shedding. How does a system protect itself from overload?**

reliability

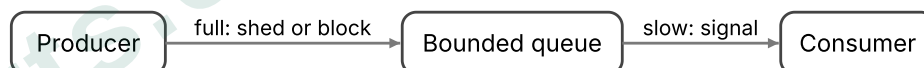
🗣 Spoken answer · ~89 s

Backpressure and load shedding keep a system useful when demand exceeds capacity, instead of collapsing. Backpressure propagates the signal 'slow down' upstream: bounded queues that block or reject when full, pull-based consumers that fetch at their own pace, flow control in TCP and HTTP/2, reactive streams, and 429 or 503 responses with Retry-After so clients back off. Unbounded queues are dangerous: they hide overload, grow memory, and make latency enormous, so a request answered after the client has already timed out is wasted work. Load shedding deliberately drops excess work early and cheaply to protect the rest: reject at the edge or gateway with rate limits, concurrency limits per instance, and adaptive limits that track latency, as in TCP-style AIMD; shed by priority, keeping critical traffic like checkout and health checks while dropping low-value requests like analytics and prefetch; drop requests whose deadlines have already passed, using deadline propagation. Related protections: timeouts, retries with budgets and jitter, since retries multiply load, circuit breakers, and bulkheads so one tenant or dependency can't consume all resources. Autoscaling helps but is too slow for sudden spikes, so shedding covers the gap. Design for graceful degradation, and test with load tests that exceed capacity. Monitor saturation, queue depth, and rejection rates, and make shedding behaviour visible to clients.

Likely follow-ups

- Why are unbounded queues dangerous?
- How do retry budgets prevent retry storms?

🚩 **Backpressure**



When demand exceeds capacity, a system must degrade gracefully rather than collapse. Backpressure and load shedding are how it protects itself.

BACKPRESSURE

Signal upstream to slow down when you're saturated:

- bounded queues (reject/block when full, don't grow unboundedly)
- TCP flow control / HTTP 429 with Retry-After
- reactive streams (request N items at a time)

Unbounded queues are a trap: they hide overload, add latency, then OOM.

LOAD SHEDDING

When you can't keep up, drop work deliberately:

- reject low-priority requests first (shed by priority)
- admission control: cap concurrent requests
- fail fast with 503 rather than queueing into oblivion

Better to serve 90% well than 100% badly (or crash and serve 0%).

RELATED PROTECTIONS

Rate limiting: bound input rate per client

Concurrency limits: cap in-flight work (e.g. semaphore per dependency)

Bulkheads: isolate resource pools so one tenant can't starve others

Autoscaling: add capacity – but it lags; shedding handles the gap

Principal answer: The mental model is: autoscaling handles sustained growth, but it lags by minutes — so you need load shedding to survive the seconds-scale spike. Use bounded queues (unbounded queues just defer the crash via OOM), shed by priority so critical traffic survives, and fail fast with 503 + Retry-After instead of queueing requests the client has already given up on. Protecting throughput for the majority beats trying to serve everyone and toppling.



Official docs

- [Google SRE Book: Handling Overload](#)
- [AWS Builders' Library: Using load shedding to avoid overload](#)

59. **Design a data replication strategy. Compare sync, async, and quorum replication.**

data

🗣 Spoken answer · ~88 s

Replication keeps copies of data on multiple nodes for durability, availability, and read scaling, and the key choice is when a write is acknowledged. Synchronous replication: the primary waits for one or more replicas to confirm before acknowledging. No data loss on failover, so RPO is zero, but higher write latency, and availability suffers if a replica is slow or down, so cross-region synchronous replication is costly. Asynchronous replication: the primary acknowledges after its own commit and ships changes afterwards. Low latency and high availability, but replicas lag, reads from them can be stale, and a failover can lose recently committed writes. Semi-synchronous: wait for at least one replica, balancing the two. Quorum replication, as in Dynamo-style systems and Cassandra: with N replicas, a write needs W acknowledgements and a read consults R ; if R plus W exceeds N , the read set overlaps the latest write, giving strong consistency, such as $N=3, W=2, R=2$. Tuning W and R trades latency and availability. Consensus-based replication, Raft or Paxos, as in etcd or Spanner, commits when a majority persists the entry, giving linearizability with automatic leader failover. Topology: single-leader is simplest; multi-leader and leaderless need conflict resolution through last-write-wins, vector clocks, or CRDTs. Choose by RPO, latency, and consistency needs, and monitor replication lag.

Likely follow-ups

- What does R plus W greater than N guarantee?
- What is the risk of async replication on failover?

🚩 Replication modes

Synchronous	Asynchronous
Waits for replica ack	Returns after primary write
No data loss	Replica lag
Higher latency	Possible loss on failover

Replication copies data across nodes for durability, availability, and read scaling. The sync/async/quorum choice trades consistency against latency and availability.

Synchronous: primary waits for replica(s) to ack before confirming write
+ no data loss on primary failure - higher write latency, availability risk

Asynchronous: primary acks immediately, replicates in background
+ low latency, high availability - data loss window if primary dies

Quorum ($W+R > N$): write to W replicas, read from R ; tune consistency
+ flexible balance - more complex, still needs conflict handling

QUORUM MATH (DYNAMO-STYLE)

N = replicas, W = write quorum, R = read quorum
 $W + R > N$ → strong consistency (read sees latest write)
 $W=N, R=1$ → fast reads, slow/fragile writes
 $W=1, R=N$ → fast writes, slow reads
 $W=R=2, N=3$ → common balanced choice (survive 1 node loss)

TOPOLOGIES & CONFLICTS

Single-leader: one primary, replicas follow (most common; failover needed)
Multi-leader: writes to multiple regions → conflict resolution required
Leaderless: any node accepts writes (Dynamo/Cassandra) → quorum + read-repair
Conflict resolution: last-write-wins (lossy), vector clocks, CRDTs

Interview tip: Tie the choice to the data: synchronous for can't-lose data (payments) accepting the latency, async for read-scaling and availability where a small loss window on failover is tolerable. For leaderless/multi-region, show the quorum formula $W+R>N$ and that you still need conflict resolution (LWW loses data; vector clocks/CRDTs preserve it). Mention replication lag as the source of read-your-writes anomalies.



Official docs

- [Google SRE Book: Managing Critical State \(consensus\)](#)
- [Raft consensus algorithm](#)
- [PostgreSQL: High availability, load balancing, replication](#)

60. How do you achieve read-your-writes and other consistency guarantees with replication lag?

tradeoffs

🗣 Spoken answer · ~88 s

With asynchronous replication, a user can write to the primary and then read from a lagging replica and not see their own change, which is confusing. Guarantees and techniques. Read-your-writes: ensure a user sees their own updates. Approaches: read from the primary for data the user recently modified, for example for a minute after a write, or for the profile owner's own data; track the user's last write timestamp or log position and only read from replicas that have caught up to it; route a user's reads to the same replica or the primary via sticky sessions; or use a client-held token containing the commit position. Monotonic reads: a user never sees data go backwards in time when consecutive reads hit different replicas; fix by pinning each user to one replica, for example by hashing the user ID. Consistent prefix reads: causally related writes appear in order, which matters in partitioned databases; fix by writing related data to the same partition or using causal consistency tracking. Causal consistency: operations that depend on one another are seen in order, tracked using version vectors. Stronger options: synchronous replication, quorum reads, or a consensus-based database for linearizability, at higher latency. Choose per feature: critical flows like checkout use the primary; feeds and counts tolerate lag.

Likely follow-ups

- How can a log position token enforce read-your-writes?
- Why does cross-device use complicate read-your-writes?

With async replicas, a user can write to the primary then read a stale replica — seeing their own change vanish. Several consistency models address such anomalies.

THE CONSISTENCY SPECTRUM

Strong / linearizable: every read sees the latest write (expensive)
Read-your-writes: you always see YOUR own writes
Monotonic reads: you never see time go backwards (no flicker)
Consistent prefix: reads see writes in causal order
Eventual: replicas converge "eventually" (weakest)

TECHNIQUES TO FIX LAG ANOMALIES

Read-your-writes:
- route a user's reads to the primary for a short window after their write
- or pin reads to a replica known to have caught up (track write timestamp/LSN)
Monotonic reads:
- pin a user to the same replica (sticky) so they don't bounce to a staler one
Version tokens:
- client passes last-seen version; system serves a replica $\geq$ that version

PRACTICAL PATTERN

```
# After write, return a write-timestamp/LSN to the client.  
# On read, client sends it; router picks a replica that has applied >= that LSN,  
# else falls back to primary. Gives read-your-writes without all-strong cost.
```

Principal answer: Don't reach for global strong consistency to fix one anomaly — it's expensive. Identify the specific guarantee the use case needs (usually read-your-writes + monotonic reads) and solve it surgically: route post-write reads to the primary or to a caught-up replica via a version token, and use sticky replica routing to prevent reads going backwards. This gives the UX of consistency at a fraction of the cost of linearizability.

Official docs

- [PostgreSQL: High availability, load balancing, replication](#)

61. Design a multi-region active-active architecture. How do you handle data and failover?

platform

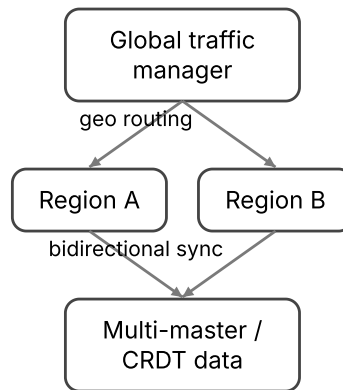
Spoken answer · ~80 s

Active-active means multiple regions serve live traffic at once, giving low latency to nearby users and resilience to a regional outage. Traffic: global load balancing via anycast or latency-based DNS with health checks directs users to the nearest healthy region; each region runs the full stack and is sized to absorb a failed neighbour's share. Data is the hard part. Options. Partition by home region: each user or tenant has a home region that handles writes, with replicas elsewhere for reads and disaster recovery; simple, avoids conflicts, but cross-region users pay latency. Multi-writer databases, such as Spanner, CockroachDB, or DynamoDB global tables, accept writes in any region; strongly consistent ones use consensus across regions and pay inter-region latency on writes, while eventually consistent ones replicate asynchronously and need conflict resolution, last-writer-wins, application merge logic, or CRDTs. Design data so conflicts are rare or mergeable, and avoid global uniqueness constraints that need coordination. Caches are regional; queues and events replicate through cross-region streams. Failover: health checks remove the failed region from DNS or load balancers, and since all regions are live there's no cold start; verify that data replication lag defines the potential loss.

Likely follow-ups

- How do you avoid write conflicts in active-active?
- Why avoid cross-region calls in the request path?

🚩 Active-active regions



Active-active serves traffic from multiple regions simultaneously for low latency and disaster resilience. The hard part is data consistency across regions.

Users → GeoDNS / Anycast → nearest region

Region US	Region EU	Region APAC
[full stack]	[full stack]	[full stack]
↓	↓	↓
cross-region data replication / globally-distributed DB		

ACTIVE-ACTIVE VS ACTIVE-PASSIVE

Active-passive: one region serves, standby on failover

+ simpler, no write-conflict problem - wasted capacity, failover lag

Active-active: all regions serve traffic

+ low latency everywhere, instant failover - cross-region write conflicts

DATA STRATEGIES

1. Partition by geo (data residency): EU users' data lives in EU
→ most writes stay regional, no conflict. Great when data is user-scoped.
2. Globally-distributed DB (Spanner/CockroachDB/DynamoDB global tables)
→ handles replication + consistency for you (at a cost).
3. Single write-region + read replicas elsewhere (active-active reads only)

FAILOVER & CONFLICTS

Health-checked GeoDNS reroutes away from a dead region automatically

Conflict resolution for multi-region writes: LWW, CRDTs, or geo-partitioning

Watch: cross-region latency (~100ms+) makes synchronous global writes painful

Interview tip: The cleanest active-active answer is geo-partitioning: shard data so a user's writes stay in their home region, eliminating cross-region write conflicts entirely. If data must be global and strongly consistent, name Spanner/CockroachDB and accept the latency cost of cross-region consensus. Always cover automatic failover via health-checked GeoDNS, and call out that ~100ms inter-region latency makes naive synchronous global writes a non-starter.

Official docs

- [AWS Well-Architected: Reliability pillar](#)
- [Google Cloud: Multi-region deployment archetypes](#)

62. Design a service discovery system. How do services find each other dynamically?

platform

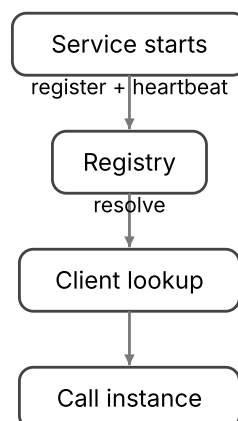
Spoken answer · ~78 s

Service discovery lets services find the current network locations of other services, which change constantly in dynamic environments. Registry-based models: a service registry such as Consul, etcd, ZooKeeper, or Eureka keeps a list of healthy instances. Instances register on startup, with a heartbeat or health check, and deregister on shutdown or on failure. Client-side discovery: the caller queries the registry, caches the list, and load-balances itself, as in Netflix Ribbon or gRPC client-side balancing; it avoids an extra hop but puts logic in every client and language. Server-side discovery: the caller sends to a load balancer or proxy that consults the registry, simplifying clients at the cost of a hop and another component to run. DNS-based discovery: service names resolve to instance IPs or virtual IPs; simple and universal, but DNS caching and TTLs can serve stale endpoints. Kubernetes does this natively: Services give a stable virtual IP and DNS name, backed by EndpointSlices updated from pod readiness, while kube-proxy or a CNI performs balancing. A service mesh moves discovery and load balancing into sidecar or node proxies that get endpoint updates from a control plane.

Likely follow-ups

- What are the trade-offs of client-side versus server-side discovery?
- How does Kubernetes handle discovery?

Service discovery



In dynamic environments services come and go (autoscaling, deploys, failures), so hard-coded addresses don't work. Service discovery maintains a live registry of healthy instances.

Service instances → register (with health) → Service Registry
(etcd/Consul/Eureka)

Caller → query registry → get healthy instances → call one (with LB)
↑ registry health-checks instances, removes dead ones

DISCOVERY PATTERNS

Client-side discovery: client queries registry, picks instance, calls directly
+ fewer hops - discovery logic in every client/language
Server-side discovery: client calls a LB/router that consults the registry
+ clients stay dumb - extra hop
DNS-based (K8s Services): query a stable DNS name → resolves to healthy pods

HEALTH & CONSISTENCY

Registration: self-register on startup OR third-party (orchestrator) registers
Health checks: registry probes instances; remove on failure (with grace)
The registry itself must be HA + consistent → backed by Raft (etcd/Consul)
Stale entries are the enemy: a removed-too-slowly dead node gets traffic

Principal answer: In Kubernetes this is mostly solved for you — Services + kube-dns/CoreDNS give a stable virtual IP/DNS name that load-balances across healthy pods, with the control plane handling registration and health. Explain the underlying model anyway (registry + health checks + Raft-backed consistency) and the client-side vs server-side trade-off. The recurring failure mode to mention: stale registry entries routing traffic to dead instances, which is why aggressive health-checking matters.



Official docs

- [HashiCorp Consul: Service discovery](#)

63. How do you prevent and handle cascading failures in a microservices architecture?

reliability

🗣 Spoken answer · ~88 s

A cascading failure begins when one component's trouble spreads: a slow dependency makes callers hold threads and connections, they time out and retry, load grows, queues fill, and failures propagate upstream and sideways. Prevention. Timeouts everywhere, with deadlines propagated across calls, so work doesn't continue after the caller has given up. Bounded retries with exponential backoff, jitter, and retry budgets, since naive retries multiply load. Circuit breakers to fail fast, and bulkheads to isolate thread pools, connection pools, or instances per dependency or tenant. Load shedding and concurrency limits to reject excess work early. Rate limiting and backpressure, with bounded queues. Graceful degradation with fallbacks, so non-critical dependency failure doesn't break the core flow. Avoid synchronous call chains and shared fate, such as every service sharing one database or cache. Capacity headroom and tested autoscaling; avoid dependency on a cold cache. Staggered deployments and canaries so a bad change doesn't hit everything at once. Handling when it's happening: identify the origin with distributed tracing and dashboards, then stop the amplification: shed load, disable retries or non-critical features with flags, scale or restart the bottleneck, and bring traffic back gradually to avoid a thundering herd. Afterwards, run a blameless postmortem and add chaos experiments and load tests to verify protections work.

Likely follow-ups

- How do retry budgets limit amplification?
- What does a bulkhead isolate?

A cascading failure is when one service's failure propagates and topples others, turning a small incident into a system-wide outage. Microservices are especially prone to it.

HOW CASCADES HAPPEN

Service C slows → callers (B) block on it → B's threads exhaust →
B's callers (A) block on B → A exhausts → the whole chain is down.
Or: retry storms – failures trigger retries → amplify load → make it worse.

DEFENSES

Timeouts: never wait forever on a dependency (the #1 cause)
Circuit breakers: stop calling a failing dependency, fail fast
Bulkheads: isolate thread/connection pools per dependency
Retry budgets: cap retries (e.g. ≤10% extra) + backoff + JITTER
Load shedding: drop excess load to protect core capacity
Fallbacks: degrade gracefully (stale cache, default response)

RETRY STORMS (THE AMPLIFIER)

```
# Naive retries during an outage MULTIPLY load on the struggling service.
# Fix: exponential backoff + jitter, retry budgets, and circuit breakers
#      that stop retrying once a dependency is clearly down.
```

Interview tip: Name timeouts as the root cause first — unbounded waits are what let a slow dependency exhaust caller threads. Then layer circuit breakers (fail fast), bulkheads (contain the blast), and retry budgets with jitter (so retries don't become a self-inflicted DDoS). The senior insight: retries are a double-edged sword — they help with transient blips but amplify sustained outages, so they must be bounded and paired with circuit breaking.

Official docs

- [Google SRE Book: Addressing Cascading Failures](#)
- [Martin Fowler: Microservices](#)

64. Monolith vs microservices: how do you actually decide, and what's the cost of microservices?

tradeoffs

Spoken answer · ~86 s

I decide based on team size, domain clarity, and scaling needs, not fashion. A well-structured modular monolith is often the right start: one deployable unit with strong internal module boundaries, simple local development, in-process calls, ACID transactions, easy refactoring, and low operational overhead. Microservices add independent deployment and scaling, team autonomy, fault isolation, and technology choice, but they bring a real cost: distributed-systems problems such as network latency and partial failure, data consistency without cross-service transactions, complex testing, operational overhead for deployment, observability, and service discovery, versioning of APIs, and cognitive load. Signals to split: many teams blocking each other on a shared release cycle; a component with very different scaling or availability needs; clearly bounded domains with stable interfaces; and regulatory or security isolation. Signals not to: a small team, an unclear domain where boundaries will change often, and no mature platform for CI/CD and monitoring. Premature microservices produce a distributed monolith, with the worst of both worlds: tight coupling plus network failures. Approach: build a modular monolith, enforce boundaries, extract services incrementally using the strangler pattern when a concrete pain justifies it, each with its own data. Align services with bounded contexts and team ownership, per Conway's law, and invest in the platform first.

Likely follow-ups

- What is a distributed monolith?
- What signals justify extracting a service?

Microservices are not automatically better. The decision hinges on team size, domain maturity, and operational capability — not hype.

MONOLITH:

- + simple deploy, easy local dev, no network between modules, easy transactions
- scaling couples everything, large codebase, one deploy pipeline

MICROSERVICES:

- + independent scaling & deploys, team autonomy, tech diversity, fault isolation
- distributed-systems complexity: network, partial failure, data consistency, observability, deployment/infra overhead, harder transactions

WHEN MICROSERVICES PAY OFF

- Multiple teams stepping on each other in one codebase (org scaling)
- Components with very different scaling/availability needs
- Mature domain boundaries you understand well
- You have the ops maturity: CI/CD, observability, on-call, infra automation

WHEN TO STAY MONOLITH

- Small team / early product (boundaries still shifting)
 - You can't yet operate distributed systems well
- Start with a well-modularized monolith; extract services when pain is real.

Principal answer: Microservices solve an *organizational* scaling problem more than a technical one — they let many teams ship independently. They impose a real tax: distributed transactions, partial failure, network latency, and a much higher observability/ops bar. The pragmatic path is a modular monolith first, then extract services along proven seams when team friction or divergent scaling justifies the cost. Beware the "distributed monolith" — microservices that must deploy together give you all the cost and none of the benefit.

Official docs

- [Microsoft: Strangler Fig pattern](#)
- [Martin Fowler: Microservices](#)
- [Martin Fowler: Monolith First](#)

65. **Design an event sourcing + CQRS system. What problems does it solve and what does it cost?**

data

🗣 Spoken answer · ~83 s

Event sourcing stores state as an append-only sequence of immutable events, such as `OrderPlaced` and `ItemAdded`, instead of just the current state. Current state is derived by replaying events, usually from periodic snapshots. CQRS, command query responsibility segregation, separates the write model, which validates commands and emits events, from one or more read models, projections built from the events and optimised for queries, such as denormalised views or search indexes. Architecture: commands go to an aggregate, which checks invariants against its state and appends events to the event store, with optimistic concurrency on the aggregate's version; events are published, often via the outbox or the store's change feed, to projectors that update read databases, asynchronously. Benefits: a complete audit trail, the ability to rebuild state or new read models by replaying history, temporal queries, debugging by reproducing past states, natural integration through events, and independent scaling and optimisation of reads and writes. Costs: significant complexity and learning curve; eventual consistency between write and read sides, which affects user experience; event schema evolution, requiring versioning and upcasting; replay time and snapshot management; difficulty deleting data, which conflicts with privacy rules like GDPR, often solved with crypto-shredding; and harder ad hoc querying.

Likely follow-ups

- How do you evolve event schemas over time?
- How do you handle deletion requests with immutable events?

🔗 Event sourcing + CQRS



Event sourcing stores state as an immutable log of events; CQRS separates the write model from read models. Powerful for audit and complex domains, but not free.

```
Command → validate → append EVENT to log (source of truth)
                        ↓ (events published)
Read model A (SQL view)  Read model B (search index)  Read model C (cache)
    ↑ queries              ↑ queries                  ↑ queries
Current state = fold(events). Read models are projections, rebuildable from the log.
```

WHAT IT GIVES YOU

- Full audit trail / history (every change is an event, immutable)
- Time travel: reconstruct state at any past point
- Multiple read models optimized per query (CQRS)
- Natural fit for event-driven integration

WHAT IT COSTS

- Complexity: eventual consistency between write log and read models
- Event schema evolution / versioning is hard (events live forever)
- Replay/rebuild needs snapshots to bound cost
- Steeper learning curve; over-engineering risk for simple CRUD

SNAPSHOTS

- # Don't replay millions of events to get current state.
- # Periodically snapshot aggregate state; replay only events since last snapshot.

Interview tip: Reach for event sourcing + CQRS only where its strengths matter — strong audit requirements (finance, healthcare), complex domains, or wildly divergent read patterns. For simple CRUD it's massive over-engineering. Be honest about the costs: eventual consistency between write and read sides, event versioning, and the need for snapshots. Showing you know *when not* to use it signals more maturity than knowing how.

Official docs

- [Microsoft: CQRS pattern](#)
- [Microsoft: Event Sourcing pattern](#)

66. Design a disaster recovery strategy. Explain RTO, RPO, and the DR tiers.

reliability

Spoken answer · ~88 s

Disaster recovery is the plan to restore service after a major failure, such as losing a region. Two objectives drive it. RTO, recovery time objective, is how long you can be down. RPO, recovery point objective, is how much data loss you can tolerate, measured in time. Both come from business impact analysis, and tighter targets cost sharply more. Typical DR tiers, from cheapest to most expensive. Backup and restore: periodic backups copied to another region; recovery means rebuilding infrastructure and restoring data, so RTO is hours to days and RPO is the backup interval. Pilot light: the data tier is continuously replicated and a minimal core of infrastructure is ready, with compute scaled up on failover, giving an RTO in tens of minutes to hours. Warm standby: a scaled-down but fully functional copy runs in the second region, receiving replicated data, scaling up at failover, with RTO in minutes. Multi-site active-active: all regions serve traffic, with near-zero RTO and low RPO, at the highest cost and complexity. Essentials: infrastructure as code to rebuild reliably, cross-region replication and immutable backups protected from deletion and ransomware, automated failover runbooks, DNS and traffic switching, secrets and identity available in the DR region, and, critically, regular testing through game days, because untested DR plans fail.

Likely follow-ups

- How do you choose a DR tier for a given system?
- Why are immutable backups important?

DR tiers

Backup and restore	Warm / hot standby
High RTO and RPO	Low RTO and RPO
Cheapest	Costly, needs failover drills

Disaster recovery plans for catastrophic loss (region down, data corruption). Two numbers drive every decision: RTO and RPO.

THE TWO KEY METRICS

RTO (Recovery Time Objective): how long until you're back up?

→ drives your failover architecture

RPO (Recovery Point Objective): how much data can you afford to lose?

→ drives your backup/replication frequency

"RTO 5 min, RPO 0" is wildly more expensive than "RTO 24h, RPO 1h."

DR STRATEGY TIERS (COST VS RTO/RPO)

Backup & restore: cheap; RTO hours-days, RPO = backup interval

Pilot light: minimal standby (DB replicating, app off); scale up on DR

Warm standby: scaled-down full stack running; promote on DR

Hot / active-active: full second region serving; near-zero RTO/RPO (costliest)

WHAT A REAL PLAN INCLUDES

- Cross-region backups (and test RESTORES – untested backups aren't backups)
- Data corruption protection: point-in-time recovery, immutable backups
- Runbooks for failover (don't improvise during a disaster)
- Regular DR drills / game days to validate RTO/RPO are actually met

Principal answer: Start by asking the business for RTO and RPO — they're requirements, not technical choices, and they determine cost. Map them to a tier: backup/restore, pilot light, warm standby, or hot active-active. The two points interviewers reward: backups are worthless unless restores are tested regularly, and DR plans are worthless unless drilled — a plan that's never been exercised will fail when you need it. Also distinguish DR (disaster) from HA (component failure); they're different problems.

Official docs

- [AWS Disaster Recovery of workloads \(whitepaper\)](#)
- [Google SRE Book: Data Integrity](#)

67. Design a container orchestration platform's scheduler (like the Kubernetes scheduler).

platform

🔊 Spoken answer · ~78 s

A scheduler assigns pending workloads to nodes. In Kubernetes, the scheduler watches for pods with no node and runs a two-phase cycle for each. Filtering eliminates nodes that cannot run the pod: insufficient CPU or memory against resource requests, taints without tolerations, node selector or affinity mismatches, and volume topology constraints. Scoring ranks the feasible nodes using plugins, such as spreading pods evenly, balanced resource allocation, and preferring nodes that already have the image. The highest score wins, and the scheduler binds the pod by writing the node name to the API server, after which the kubelet starts it. For a custom scheduler, I'd consider scale, which means caching cluster state from watches, scheduling concurrently, and binding asynchronously with optimistic assumptions; priority and preemption, so a high-priority pod can evict lower-priority ones when nothing fits; the resource model, including bin packing versus spreading and extended resources such as GPUs; constraints such as anti-affinity and zone spread; gang scheduling for jobs that need all their pods at once; and fault handling, with leader election so only one instance is active. The core trade-off is scheduling latency against placement quality.

Likely follow-ups

- What is the difference between filtering and scoring?
- How does preemption work?

🔧 Scheduler loop



The scheduler decides which node each workload runs on, balancing resource fit, constraints, and spreading — at scale, efficiently.

Pending pod → Scheduler:

1. FILTER (predicates): which nodes CAN run this pod?
resources fit? node selector / affinity? taints tolerated? volumes?
2. SCORE (priorities): rank feasible nodes
least-loaded? spread across zones? affinity preferences? image locality?
3. BIND: assign pod → highest-scoring node → kubelet runs it

FILTERING (HARD CONSTRAINTS)

- Sufficient CPU/memory (requests fit remaining capacity)
- nodeSelector / nodeAffinity match
- Taints/tolerations (node says "only these pods")
- Pod affinity/anti-affinity (co-locate or spread relative to other pods)

SCORING (SOFT PREFERENCES)

- Spread across nodes/zones for availability (topology spread)
- Bin-packing vs spreading (cost efficiency vs blast-radius)
- Image locality (node already has the image → faster start)
- Preferred affinity weights

SCALE & ADVANCED

Resource model: requests (scheduling) vs limits (runtime cap)
 Preemption: evict lower-priority pods to fit a higher-priority one
 Descheduling/rebalancing: move pods when the cluster drifts from optimal
 Scale: schedule thousands of pods/sec → efficient filtering, caching node state

Interview tip: Frame it as the two-phase filter-then-score pipeline — hard constraints eliminate infeasible nodes, soft priorities rank the rest. Discuss the bin-packing (cost) vs spreading (availability) tension as a scoring trade-off. Mention requests-vs-limits (scheduling uses requests), preemption for priority, and that at scale the scheduler caches node state and processes a queue rather than rescanning the cluster per pod.

Official docs

- [Kubernetes: Assigning Pods to Nodes](#)

68. Strong vs eventual consistency: how do you decide per feature, with examples?

tradeoffs

Spoken answer · ~87 s

The decision is per feature: ask what the business cost of a stale or conflicting read is, versus the cost in latency and availability of preventing it. Strong consistency means that once a write completes, every subsequent read sees it, as if there were one copy. It costs coordination: higher latency, lower availability under partitions, and limited cross-region scale. Use it where correctness is critical: account balances and payments, inventory for the last item, uniqueness constraints like usernames, authorisation and permission changes, and leader election or locks. Eventual consistency means replicas converge if writes stop, but reads may be stale for a while. It gives lower latency, higher availability, and easier scale. Use it where staleness is harmless or tolerable: social feeds, like and view counts, recommendations, search indexes, analytics, caches, and notification counts. Many systems need a mix, and intermediate models help: read-your-writes for a user's own posts, causal consistency for comments appearing after the posts they reply to, and bounded staleness, a guaranteed maximum lag. Design tactics: keep the system of record strongly consistent and derive eventually consistent views from it; make operations idempotent and commutative; communicate pending states in the UI; and use compensating actions to repair inconsistencies. Example: shopping cart eventual, checkout stock decrement strong.

Likely follow-ups

- Which intermediate consistency models would you offer?
- Why keep one strongly consistent source of truth?

Consistency is a per-feature decision, not a system-wide one. Strong consistency is simpler to reason about but costs latency and availability; eventual scales but admits anomalies.

THE TRADE-OFF

Strong: every read sees the latest write

- + correct & simple to reason about
- higher latency, reduced availability under partition (CAP: CP)

Eventual: replicas converge over time

- + low latency, high availability, scales (CAP: AP)
- reads may be stale; needs conflict handling

DECIDE BY COST OF BEING WRONG

Needs STRONG (correctness critical):

- account balance / payments / inventory "last item"
- unique username claim, booking the same seat
- anything where stale data causes money loss or double-allocation

Fine with EVENTUAL (stale is acceptable):

- social feed, like counts, view counts
- user profile, recommendations, search index freshness
- analytics, notifications

MIXING WITHIN ONE PRODUCT

- # E-commerce checkout:
- # inventory decrement & payment → strong (can't oversell / double-charge)
- # product reviews, "customers also bought" → eventual (stale is fine)
- # Use the right model per data type; don't make everything strong (too slow)
- # or everything eventual (bugs with money).

Principal answer: Frame it as cost-of-staleness. Ask: if a read is a few seconds stale, what breaks? If the answer is "money or double-allocation," go strong (and accept the latency/availability hit). If it's "a slightly old like count," go eventual and reap the scale. The strongest answers explicitly mix models in one system — strong for the checkout/balance path, eventual for feeds and counts — showing you don't apply one hammer everywhere.

Official docs

- [Jepsen: Consistency models](#)

69. **Design rate limiting at scale across a distributed fleet. How do limits stay accurate?**

networking

🗣 Spoken answer · ~88 s

Rate limiting across a fleet needs a shared view of usage, while avoiding a central bottleneck. Options. Centralised counters in a fast store such as Redis, with atomic operations or Lua scripts implementing token bucket or sliding window: accurate, simple, but adds a network hop per request and makes Redis a critical dependency; shard keys across a Redis cluster for scale. Local limiting per instance, dividing the global limit by the instance count: no network cost, but inaccurate when traffic is uneven or the fleet size changes. Hybrid: each instance keeps a local token bucket and periodically syncs with a central store, borrowing batches of tokens, such as requesting 100 tokens at a time; this reduces central calls by orders of magnitude, accepting a small overshoot. Gossip or eventually consistent counters between nodes, as in some CRDT-based approaches. Sticky routing, via consistent hashing of the client key to a specific limiter node, makes local counters accurate without shared state. Accuracy versus performance is the core trade-off; for abuse prevention, approximate is fine, while billing quotas need exactness. Handle edge cases: clock differences, using server-side time; failure mode, deciding to fail open or closed; hot keys; multi-region limits, which may be split per region with periodic rebalancing; and fairness across tenants.

Likely follow-ups

- How does token borrowing reduce central calls?
- When must a limiter be exact versus approximate?

Single-node rate limiting is easy; the hard problem is enforcing a global limit across many gateway/server instances without a bottleneck or wild inaccuracy.

THE DISTRIBUTED CHALLENGE

```
# Limit: 100 req/min per user. Traffic spread across 10 gateway pods.
# Naive: each pod allows 100 → user actually gets 1000. Wrong.
# Need shared state OR a smart approximation.
```

APPROACHES

Centralized (Redis) counter:

all pods INCR a shared key (atomic, Lua sliding window)
+ accurate - Redis is a hot dependency / potential bottleneck

Local + sync (token distribution):

each pod gets a share of the budget; periodically rebalance
+ fast, no per-request network hop - approximate near boundaries

Tiered: coarse local limit (fast) + global Redis check for precision

ALGORITHMS (RECAP)

Token bucket: allows bursts, smooth refill – most common for APIs
Sliding window counter: accurate, bounded memory
Return headers: X-RateLimit-Remaining, Retry-After (good API citizenship)

RESILIENCE

```
# If the rate-limit store (Redis) is down – fail OPEN or CLOSED?  
#   fail open: don't block legit traffic (risk abuse)  
#   fail closed: protect backend (risk blocking everyone)  
# Usually fail open with a conservative local fallback limit.
```

Interview tip: The crux is shared state vs accuracy. A central Redis counter with an atomic Lua sliding window is accurate but a hot dependency; local budgets with periodic sync are fast but fuzzy at boundaries. The pragmatic answer is tiered: a cheap local limit for the common case plus a global check for precision. Always address the failure mode — what happens when the limiter store is down — and lean fail-open with a conservative local cap.



Official docs

- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [Stripe Engineering: Scaling your API with rate limiters](#)

70. Design an internal developer platform (IDP) with golden paths and self-service.

platform

🗣 Spoken answer · ~88 s

An internal developer platform provides self-service capabilities packaged as golden paths: opinionated, supported routes from idea to production. Design around the developer journey. Entry point: a portal, like Backstage, with a catalogue of services, owners, docs, and templates; the new service template creates a repository, CI pipeline, deployment manifests, observability dashboards, and alerts in minutes, already secure and compliant. Delivery: shared pipeline templates, an artefact registry with scanning and signing, GitOps deployment with progressive delivery, and preview environments. Runtime: managed clusters or serverless, namespaces per team, standard ingress, autoscaling, and secrets integration. Infrastructure self-service: a catalogue of vetted modules or custom resources, through Terraform modules or Crossplane, for databases, queues, and buckets, with guardrails like quotas, tagging, and policies embedded. Observability and reliability defaults: logs, metrics, traces, and SLO templates wired automatically. Security by default: identity, policy as code, and scanning in the path. Golden paths should be easy to follow and optional to leave: provide escape hatches, but make the paved road the easiest choice. Platform as a product: interview developers, define a roadmap, measure adoption, time to first deploy, lead time, change failure rate, and developer satisfaction; provide support and documentation; version APIs and migrate users carefully. Start with the biggest pain points, not a grand all-in-one build.

Likely follow-ups

- Why make golden paths optional?
- Which metrics show the platform is working?

An IDP abstracts infrastructure so developers ship without becoming Kubernetes/cloud experts — paved roads with guardrails, self-service from day one.

```
Developer → Portal/CLI (self-service) →  
  Golden-path templates (service scaffold, CI/CD, observability baked in)  
  → provision via IaC (Terraform/Crossplane) on K8s/cloud  
  → guardrails: policy-as-code, quotas, security defaults  
  ← golden signals dashboards auto-wired
```

CORE CAPABILITIES

```
Golden paths: opinionated templates for the 80% common case  
  → "new service" = repo + pipeline + dashboards + alerts in minutes  
Self-service: provision DBs, queues, environments without filing tickets  
Guardrails: policy-as-code (OPA/Kyverno), secure defaults, cost quotas  
Abstraction: hide YAML/cloud details behind a simple interface
```

DESIGN PRINCIPLES

- Paved road, not a cage: golden path is easy; deviation is possible but owned
- Reduce cognitive load: devs think in services, not pods/IAM/VPCs
- Day-2 included: observability, scaling, secrets, on-call wired by default
- Treat the platform as a PRODUCT: devs are customers; measure adoption

MEASURING SUCCESS

Lead time to first deploy (new service → prod)
Deployment frequency, change-fail rate (DORA metrics)
Golden-path adoption %, ticket reduction, developer satisfaction

Principal answer: The philosophy is paved road, not cage: make the right way the easy way via golden-path templates that bake in CI/CD, observability, and security, while allowing (owned) deviation. Treat the platform as a product with developers as customers — measure adoption and DORA metrics, not just uptime. Backstage for the portal, Crossplane/Terraform for provisioning, OPA/Kyverno for guardrails. The goal is reducing developer cognitive load so they think in services, not infrastructure primitives.



Official docs

- [Backstage: What is Backstage?](#)
- [CNCF: Platforms white paper](#)
- [Spotify Engineering: Golden Paths](#)

71. Design a distributed web crawler. Crawl billions of pages politely and efficiently.

scalability

🗣 Spoken answer · ~86 s

A web crawler downloads pages starting from seed URLs, extracts links, and repeats, at the scale of billions of pages. Core components. URL frontier: a prioritised, politeness-aware queue; it has front queues ordered by priority, such as page importance and expected change rate, and back queues, one per host, so each site is accessed at a controlled rate, respecting robots.txt crawl delays. Fetchers: a large distributed fleet; assign hosts to workers by hashing the hostname, so politeness state stays local; use asynchronous I/O, connection reuse, timeouts, size limits, and a caching DNS resolver, as DNS lookups are a bottleneck. Parsing and link extraction: normalise URLs, resolve relatives, strip tracking parameters, and filter by allowed schemes and domains. Deduplication: a URL-seen test using a Bloom filter or distributed key-value store, plus content dedup by checksums and SimHash for near-duplicates. Storage: raw content in a distributed file or object store, metadata and link graph in a scalable database, and a pipeline feeding the indexer. Scheduling recrawls by estimating each page's change frequency, so news sites refresh often and static pages rarely. Robustness: avoid crawler traps with depth and per-domain budgets, handle redirects and errors with backoff, detect soft 404s, and render JavaScript for selected pages using headless browsers.

Likely follow-ups

- Why does a crawler need per-host queues?
- How do you detect near-duplicate content?

A web crawler must fetch billions of pages, avoid re-crawling, respect politeness, and distribute work — without overwhelming any single site or itself.

```
Seed URLs → URL Frontier (priority + politeness queues)
    ↓
Fetchers (distributed) → download page
    ↓
Parser → extract links + content
    ↓ new links          ↓ content
Dedup (seen?) ——— Content store + index
    ↓ unseen
back to Frontier
```

THE URL FRONTIER (THE HEART)

```
Politeness: per-domain queue + delay (don't hammer one host)
Priority: crawl important/fresh pages first (PageRank-ish, change rate)
Partition by domain → one host's pages handled by one worker (rate control)
```

DEDUP & FRESHNESS

Seen-URL set: billions of URLs → bloom filter (memory-efficient, probabilistic)
Content dedup: hash/simhash to skip near-duplicate pages
Re-crawl: schedule by observed change frequency (news daily, archive rarely)
robots.txt: fetch + cache per domain; respect crawl-delay and disallow

SCALE & TRAPS

- Crawler traps (infinite calendar URLs) → depth limits, URL pattern detection
- DNS load → cache DNS aggressively
- Distributed coordination → partition URL space by domain hash

Interview tip: The URL frontier is the design — it encodes both politeness (per-domain rate limiting so you're a good citizen) and prioritization (crawl valuable/changing pages first). Use a bloom filter for the billions-scale seen-set, partition by domain so one worker controls a host's crawl rate, and respect robots.txt. Mention crawler traps and re-crawl scheduling by change rate to show production awareness.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

72. Design a recommendation system. Cover candidate generation and ranking at scale.

data

Spoken answer · ~89 s

Recommendation systems are usually a multi-stage funnel, because scoring every item for every user is far too expensive. Stage one: candidate generation, which narrows millions of items to hundreds or a few thousand cheaply. Sources include collaborative filtering, such as matrix factorisation or item-item similarity; embedding-based retrieval with a two-tower model, searched by approximate nearest neighbour indexes like HNSW or FAISS; popularity and trending lists; content-based matches; and recent activity or follow relationships. Combining several candidate sources improves coverage. Stage two: ranking, where a heavier model scores each candidate using rich features: user history, item attributes, context like time and device, and cross features, predicting click, watch, purchase, or a blended objective; gradient boosted trees or deep neural networks serve these, with a feature store providing consistent features online and offline. Stage three: re-ranking, applying business rules, diversity, freshness, deduplication, fairness, and policy filters. Infrastructure: offline pipelines compute embeddings and train models on logged interactions; online serving must respond within about 100 milliseconds, using precomputed candidates, caches, and low-latency feature lookup. Cold start for new users and items uses popularity, content features, and exploration, such as bandit algorithms. Evaluate offline with precision at K and NDCG, and online with A/B tests; beware feedback loops and position bias. Monitor drift, and log impressions for training.

Likely follow-ups

- Why split candidate generation from ranking?
- How do you handle the cold-start problem?

Recommenders predict what a user will engage with. At scale the standard architecture is a two-stage funnel: cheap candidate generation, then expensive ranking.

Millions of items

- ↓ CANDIDATE GENERATION (fast, approximate) → ~hundreds
- ↓ RANKING (heavy ML model, precise) → ~dozens
- ↓ RE-RANK / business rules (diversity, freshness, dedup) → final list

CANDIDATE GENERATION

Collaborative filtering: users like you liked X (matrix factorization)
Content-based: items similar to what you engaged with (embeddings + ANN)
Graph / co-occurrence: "bought together", session sequences
→ Reduce millions → a few hundred candidates cheaply (often via vector ANN)

RANKING

Heavy model scores each candidate with rich features:
user features + item features + context (time, device) + cross features
Predicts $P(\text{click})/P(\text{engage})$; sort by expected value.
Too expensive for millions of items → that's why stage 1 narrows first.

SERVING & FRESHNESS

Offline: train models, precompute embeddings/candidate lists (batch)
Online: real-time features + low-latency ranking (feature store + model server)
Feedback loop: log impressions/clicks → retrain; watch for filter bubbles
Cold start: new user/item → fall back to popularity/content features

Principal answer: The two-stage funnel is the key insight — you cannot run a heavy ranking model over millions of items per request, so cheap candidate generation (vector ANN / collaborative filtering) narrows to hundreds, then a rich model ranks those. Mention the offline/online split (batch training + precompute, real-time serving with a feature store), cold-start fallbacks, and re-ranking for diversity so you don't show ten near-identical items. The feedback loop and its bias risks (filter bubbles) show senior awareness.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

73. **Design a stock-trading / order-matching engine. Low latency, strict correctness.**

scalability

🔊 Spoken answer · ~83 s

A matching engine pairs buy and sell orders with strict price-time priority, requiring microsecond latency and absolute correctness. Core structure: an order book per instrument, with bids sorted by price descending and asks ascending, and a FIFO queue at each price level; typical data structures are sorted maps or arrays of price levels plus hash maps by order ID for cancels. Matching: an incoming limit order crosses against the best opposite prices until filled or no longer marketable, then rests on the book; market orders sweep the book. Determinism is the key design principle: the engine is a single-threaded, deterministic state machine per instrument, so no locks and no race conditions; all inputs are sequenced into a totally ordered log by a sequencer, and the same log replayed gives the same state, which enables recovery and audit. Performance: keep everything in memory, avoid garbage collection and allocation in the hot path, use pre-allocated memory, lock-free ring buffers like the LMAX Disruptor, kernel-bypass networking, and CPU pinning. Scale by sharding instruments across engines, not by parallelising one book. Reliability: write the event log durably, replicate it to a hot standby by consuming the same sequenced stream, with fast failover.

Likely follow-ups

- Why is the matching engine single-threaded per instrument?
- How does event-log replay help recovery and audit?

An order-matching engine pairs buy and sell orders with strict price-time priority, extreme low latency, and perfect correctness — a uniquely demanding workload.

```
Orders → Gateway (validate, risk check) → Matching Engine (per symbol)
                                         ↓ in-memory order book
                                         match by price-time priority
                                         ↓
Trades → market data feed + settlement
                                         ↓
                                         Event log (sequenced, replayable)
```

THE ORDER BOOK

Two sorted sides: bids (desc price), asks (asc price)
Match: incoming buy $\geq$ best ask $\rightarrow$ trade at resting price
Priority: price first, then time (FIFO at each price level)
Kept fully IN MEMORY – disk/network in the hot path is too slow

LATENCY & CORRECTNESS

Single-threaded per symbol $\rightarrow$ no locks, deterministic ordering
(LMAX Disruptor pattern: one thread, ring buffer, micro/nanosecond latency)
Sequence every event $\rightarrow$ deterministic replay $\rightarrow$ audit + recovery
Partition by symbol $\rightarrow$ AAPL engine independent of TSLA engine (scale out)

Replicated event log → hot standby replays the same sequence → instant failover
 Determinism is everything: same inputs → same outputs on the replica
 Risk checks pre-trade (limits, fat-finger) at the gateway, off the hot path

Interview tip: Counter-intuitively, the matching engine is single-threaded per symbol — concurrency would break deterministic price-time ordering and add lock latency. Cite the LMAX Disruptor model (one thread, in-memory ring buffer, no locks → microsecond latency). Everything is sequenced and in-memory for determinism, which also gives you replay-based recovery and a hot standby. Partition by symbol to scale horizontally. This is a case where strong consistency and low latency are both non-negotiable.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

74. Design a system to store and serve large files / object storage (like S3).

data

Spoken answer · ~87 s

Object storage like S3 stores immutable blobs addressed by key, designed for enormous scale, high durability, and low cost per byte. Architecture: separate metadata from data. A metadata service maps bucket and key to object location, size, checksum, and version, in a sharded, replicated key-value store, partitioned by key prefix hash for even load, with listing supported through ordered indexes. The data plane stores bytes on many storage nodes. Large objects are split into chunks, with multipart upload letting clients upload parts in parallel and resume. Durability, such as eleven nines, comes from replication across zones, typically three copies, or erasure coding, such as splitting into data and parity fragments, say 10 plus 4, which tolerates multiple failures at about 1.4 times storage overhead instead of three times. Background processes verify checksums, detect bit rot, and repair lost fragments. Requests flow through front-end servers that authenticate, authorise through policies and signed URLs, look up metadata, and stream data. Consistency: modern stores offer strong read-after-write consistency, via careful metadata design. Features: versioning, lifecycle rules moving data to colder tiers, replication across regions, encryption at rest with managed keys, and event notifications. Scale: add nodes and rebalance; avoid hot partitions with randomised key prefixes; serve popular objects through a CDN.

Likely follow-ups

- Why use erasure coding instead of three replicas?
- Why separate metadata from data?

Object storage holds vast numbers of large immutable blobs with high durability and scalable throughput — a fundamentally different design from a database.

Client → API (PUT/GET) →

Metadata service (object → location, size, checksum, version)

↓
Storage nodes: object split into chunks, replicated/erasure-coded across nodes
↓
Background: repair, rebalance, integrity scrub, lifecycle/tiering

DURABILITY: REPLICATION VS ERASURE CODING

Replication (3x): store 3 copies → simple, 200% storage overhead
Erasure coding (e.g. 10+4): split into 10 data + 4 parity shards
→ survive 4 shard losses with only 40% overhead (vs 200%)
→ S3-scale uses erasure coding for cost; 11 nines of durability

METADATA & SCALE

Separate metadata plane (object→chunk locations) from data plane (bytes)
Metadata is small but high-QPS → sharded, replicated KV store
Large uploads: multipart upload (parallel chunks, resumable)
Immutability + versioning: objects are versioned, not edited in place

OPERATIONAL

Background scrubbing: detect bit-rot via checksums, re-replicate/rebuild
Rebalancing: spread load as nodes are added/fail
Lifecycle/tiering: hot → cold → archive (cost optimization)

Principal answer: Two big design points: (1) separate the metadata plane from the data plane — metadata is small/high-QPS (sharded KV), data is huge (storage nodes); (2) use erasure coding rather than naive 3x replication at scale — it gives the same durability (11 nines) at a fraction of the storage cost. Add multipart upload for large/resumable transfers, background integrity scrubbing for bit-rot, and lifecycle tiering for cost. Objects are immutable + versioned, never edited in place.



Official docs

- [Git LFS](#)
- [Amazon S3 User Guide](#)

75. How do you approach a system design question when you've never seen the domain before?

fundamentals

🗣 Spoken answer · ~88 s

Facing an unfamiliar domain, the process matters more than prior knowledge. First, say so honestly and ask clarifying questions to learn the domain: who the users are, what the core workflow is, what the success criteria are, and what the key terms mean; interviewers expect and value this. Second, reduce the unfamiliar problem to familiar building blocks: almost every system consists of ingesting data, storing it, processing it, serving it, and handling failure, so map the domain's entities to data models, its actions to APIs, and its traffic to read and write patterns. Third, establish requirements and scale with numbers, even rough ones, stating assumptions explicitly so the interviewer can correct them. Fourth, identify what makes this problem hard: the dominant constraint, such as latency, consistency, volume, or fan-out, and focus the design on that. Fifth, draw an end-to-end simple design first, then deepen the riskiest components. Sixth, reason from first principles and analogies to systems you do know: a ride matching service resembles a geospatial index plus a dispatcher; an ad auction resembles a low-latency real-time bidding pipeline. Narrate trade-offs, name alternatives, and acknowledge uncertainty without bluffing. Invite the interviewer's domain knowledge: asking whether a constraint is real demonstrates collaboration. Finally, summarise the design, risks, and what you'd validate next.

Likely follow-ups

- How do you reduce an unfamiliar domain to known building blocks?
- Why is stating assumptions valuable?

Interviewers often pick unfamiliar domains deliberately — they're testing your *process*, not domain trivia. A repeatable method keeps you composed.

THE METHOD (REDUCE THE UNKNOWN TO KNOWN)

1. Clarify relentlessly: turn the vague prompt into concrete requirements.
"Design X" → who uses it, what are the core flows, what scale, what's out of scope?
2. Map to primitives you DO know:
nearly everything decomposes into reads, writes, storage, queues, caches, fan-out, search, real-time push — patterns you've seen before.
3. Identify the hard part: what makes THIS problem non-trivial? Optimize the design around it.
4. Start simple, then scale: a working simple design beats a half-drawn complex one.

PATTERN-MATCH THE CORE CHALLENGE

High write volume?	→ queue + partition + async
Low-latency reads?	→ cache + CDN + pre-compute
Real-time updates?	→ push (WebSocket/SSE) + pub-sub
Massive storage?	→ object store + tiering
Search/discovery?	→ inverted index / vector search
Strict correctness?	→ strong consistency + idempotency

Interview tip: Say this out loud: "I haven't built exactly this, so let me clarify the requirements and decompose it into patterns I know." Then narrate the mapping. Unfamiliar domains are won by process — clarify, decompose into primitives (reads/writes/cache/queue/search), find the one hard part, and design around it. Drive the conversation; don't freeze waiting to recall a domain you've never touched.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

76. Design a counter / metrics aggregation system for things like view counts and likes at scale.

data

Spoken answer · ~84 s

A counter for views and likes looks trivial, but at scale a single hot row becomes a write bottleneck. Naive approach: UPDATE counters SET $n = n + 1$, which serialises on a row lock and collapses under a viral video. Better approaches. Sharded counters: split one logical counter into N sub-counters, say 20 to 100, increment a random one, and sum them on read; writes scale linearly, reads cost a few extra lookups, and the total can be cached. Buffering and batching: aggregate increments in memory in the application or in Redis, then flush periodic deltas to the database, trading durability of the last few seconds for far fewer writes. Event-based: log each view or like as an event into Kafka, and a stream processor aggregates counts in time windows and updates the counter store; this also enables deduplication, unique viewers, and analytics, with approximate structures like HyperLogLog for unique counts at tiny memory cost. Accuracy requirements differ: view counts can be eventually consistent and approximate, shown as rounded values; likes need to track per-user state, so a user can't like twice, stored in a set or table keyed by user and item, with the count derived or updated asynchronously.

Likely follow-ups

- How do sharded counters avoid hot rows?
- Why use HyperLogLog for unique viewers?

Counting seems trivial until it's billions of increments/sec on hot items. Naive UPDATE counter SET $c=c+1$ on a row becomes a contention nightmare.

WHY NAIVE COUNTING FAILS

```
# A viral video gets 1M likes/min → 1M writes to ONE row →  
# row lock contention, the single row is a hotspot, DB melts.
```

SCALING TECHNIQUES

1. Sharded/striped counters:
split one logical counter into N sub-counters; increment a random shard;
total = SUM(shards). Spreads write contention by N .
2. In-memory aggregation + flush:

increment in Redis (atomic INCR) → periodically flush to durable store.

3. Stream aggregation:

emit "like" events → Kafka → stream processor aggregates → store rollups.
Decouples write spikes from the counter store.

ACCURACY VS COST

Exact counts: striped counters / stream aggregation (eventual, but exact total)

Approximate (huge cardinality, e.g. unique viewers):

HyperLogLog → count distinct in ~1.5KB with ~2% error

Display counts can be eventually consistent – nobody needs a like count to the ms.

Principal answer: Name the hot-row problem immediately, then solve write contention with striped counters (increment one of N shards, sum to read) or in-memory INCR in Redis flushed asynchronously. For unique counts (distinct viewers) at scale, reach for HyperLogLog — exact distinct-count storage is infeasible, and ~2% error is fine for a view counter. The framing that impresses: display counters tolerate eventual consistency, so trade exactness/latency for write scalability deliberately.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

77. Design health checks and self-healing for a large service fleet.

reliability

Spoken answer · ~80 s

Health checks tell the platform which instances can serve traffic and which need to be replaced; self-healing acts on that automatically. Types, as in Kubernetes. Liveness: is the process stuck or deadlocked, so should it be restarted. Readiness: can it serve requests right now; failing readiness removes it from load balancing without killing it. Startup: allow slow initialisation before the other probes begin. Design: keep checks cheap, fast, and local; liveness should not depend on downstream services, otherwise a database blip restarts the whole fleet. Readiness may check essential local conditions, such as caches loaded and connections established, but be careful with deep dependency checks, which can cascade by emptying the load balancer. Add timeouts and thresholds, requiring consecutive failures, to avoid flapping. Fleet level: orchestrators and autoscaling groups replace failed instances automatically; use pod disruption budgets and a maximum number of simultaneous replacements; apply backoff for crash loops, with alerts so crash loops get human attention; use auto-repair for nodes, draining and replacing them. Add outlier detection at the proxy, removing instances with elevated error rates or latency compared with peers. Protect against mass-ejection with a minimum healthy percentage, panic mode.

Likely follow-ups

- Why shouldn't liveness probes depend on downstream services?
- What protects against mass ejection of instances?

At fleet scale, instances fail constantly; the system must detect and replace them automatically. Good health checking is subtler than a single /health endpoint.

TYPES OF HEALTH CHECKS

Liveness: is the process alive? failing → RESTART it (deadlock, hang)
Readiness: can it serve traffic NOW? failing → remove from LB (warming up, deps down)
Startup: slow-starting app → don't kill it prematurely
Conflating liveness & readiness is a classic bug: a dependency blip
flips readiness (fine) but if wired to liveness it RESTARTS healthy pods (bad).

DEEP VS SHALLOW CHECKS

Shallow: "process responds" – cheap, but misses broken dependencies
Deep: "can I actually query the DB / serve a real request" – catches more,
but risk: a shared-dependency blip fails ALL instances at once →
mass eviction → outage. Tune carefully; don't cascade.

SELF-HEALING LOOP

Detect (health check fails) → react (restart / replace / reschedule) →
verify (new instance healthy) → rebalance.
Autoscaler replaces failed nodes; orchestrator reschedules pods.
Guard rails: don't replace so aggressively you mask a real bug or thrash.

Interview tip: Distinguish liveness (restart) from readiness (depool) from startup probes — conflating them causes the infamous failure where a transient dependency hiccup restarts your entire healthy fleet. Discuss the deep-vs-shallow trade-off: deep checks catch more but can cause correlated mass eviction during a shared-dependency blip. Self-healing is a detect→react→verify loop with guardrails so it doesn't thrash or hide bugs.



Official docs

- [AWS Builders' Library: Implementing health checks](#)
- [Argo CD: ApplicationSet](#)
- [Argo CD documentation](#)

78. Design a secrets management system. How are credentials stored, rotated, and accessed?

platform

🗣 Spoken answer · ~83 s

A secrets management system stores credentials, keys, and tokens securely and controls who and what can access them. Storage: secrets are encrypted at rest with a key hierarchy, where a master key, protected in an HSM or cloud KMS, wraps data encryption keys; the master key is never stored in plaintext, and unsealing may require a quorum of key shares, as in Vault. Access: every client authenticates with a strong, non-secret identity, such as a Kubernetes service account token, a cloud IAM role, or workload identity via SPIFFE, and policy grants least-privilege access to specific paths; humans use SSO with MFA. Avoid the secret zero problem by relying on platform-attested identity instead of a bootstrap password. Delivery: applications fetch secrets at runtime through an agent, sidecar, or CSI driver, or an operator syncs them into the cluster, keeping them in memory or tmpfs, never in images, Git, or environment dumps where practical. Rotation: automate it, with short-lived dynamic secrets generated on demand, such as database credentials with a TTL that are revoked automatically, which limits the blast radius of leaks; support overlapping validity so rotation causes no downtime. Auditing: log every read and write with identity, and alert on anomalies.

Likely follow-ups

- What is the secret zero problem?
- Why prefer short-lived dynamic credentials?

Secrets (DB passwords, API keys, certs) must never live in code or plain config. A secrets manager centralizes secure storage, access control, rotation, and audit.

```
App → authenticate (workload identity) → Secrets Manager (Vault/KMS/cloud)
      ↓ authorized?
      return secret (short-lived ideally) + audit log entry
      ↑
      Encryption at rest (master key in HSM/KMS), encryption in transit
```

CORE REQUIREMENTS

Encryption: at rest (envelope encryption, master key in HSM/KMS) + in transit
Access control: least-privilege; identity-based (not a shared static token)
Audit: every access logged (who, what, when) – essential for compliance
No secrets in code/images/env files committed to git

ROTATION & DYNAMIC SECRETS

Static secret rotation: change credential + update consumers without downtime
Dynamic secrets (Vault): generate short-lived, per-request DB creds that auto-expire
→ drastically shrinks the blast radius of a leak
Lease + revoke: secrets have TTLs; can be revoked instantly if compromised

WORKLOAD IDENTITY

The bootstrapping problem: how does an app prove who it is to GET secrets?
Use platform identity (K8s ServiceAccount, cloud IAM role / IRSA) –
NOT a long-lived token stored somewhere (which is itself a secret to protect).

Principal answer: Three pillars: encryption (envelope encryption with the master key in an HSM/KMS), identity-based least-privilege access with full audit, and rotation. The advanced move is dynamic, short-lived secrets (Vault generating per-request DB credentials that auto-expire) so a leaked credential is useless within minutes. Solve the bootstrapping problem with workload identity (IAM roles / K8s service accounts) rather than a static token — otherwise you just have a secret guarding your secrets.

Official docs

- [AWS KMS](#)

79. Design a system using a service mesh. What does it add over plain service-to-service calls?

networking

Spoken answer · ~88 s

A service mesh moves cross-cutting network concerns out of application code into infrastructure, typically a sidecar proxy such as Envoy next to each service, managed by a control plane like Istio or Linkerd; newer ambient designs use per-node proxies instead. What it adds over plain service-to-service calls. Security: automatic mutual TLS between services with certificates issued and rotated by the mesh, plus identity-based authorisation policies. Traffic management: load balancing, retries, timeouts, circuit breaking, outlier detection, rate limiting, and fine-grained routing for canary releases, traffic shifting by percentage, header-based routing, and mirroring, all configured declaratively without code changes. Observability: uniform metrics such as request rate, errors, and latency, distributed tracing spans, and access logs for every call, across all languages. Resilience: consistent policies instead of each team's library. Costs: extra latency per hop, resource overhead of proxies in every pod, operational complexity in upgrading the control plane and data plane, a steep learning curve, debugging difficulty with another layer in the path, and the risk of a misconfiguration affecting all traffic. When it's worth it: many services, polyglot environments, strict security requirements like zero trust, and a need for consistent traffic policy. When not: a handful of services, where a library or gateway is enough. Start with the mesh features you actually need.

Likely follow-ups

- What does mutual TLS in a mesh give you?
- When is a service mesh overkill?

A service mesh moves cross-cutting network concerns (security, reliability, observability) out of application code into a sidecar proxy layer.

Service A [app + sidecar proxy] $\rightleftharpoons$ [sidecar proxy + app] Service B
↑ data plane (Envoy proxies handle the traffic)

Control plane (Istio/Linkerd): config, certs, policy, telemetry

WHAT THE MESH PROVIDES

Security: automatic mTLS between services (encrypted + authenticated), zero code change
Reliability: retries, timeouts, circuit breaking, load balancing in the proxy
Observability: uniform metrics/traces/logs for ALL service-to-service traffic
Traffic control: canary, blue-green, traffic splitting, fault injection

SIDECAR VS SIDECAR-LESS

Sidecar (Istio classic): a proxy per pod → flexible but resource overhead + latency
Sidecar-less / ambient / eBPF (Cilium): node-level proxying → less overhead
Trade-off: capability/isolation vs resource cost and per-hop latency.

WHEN IT'S WORTH IT

Worth it: many services, need uniform mTLS + observability + traffic policy
Overkill: a handful of services – the operational complexity outweighs benefit
A mesh is powerful but adds a whole control plane to operate.

Interview tip: The pitch is: take retries, timeouts, mTLS, and telemetry out of every app (in every language) and standardize them in a sidecar/proxy layer with a central control plane. The big wins are automatic mTLS and uniform observability for free. Be balanced — call out the costs (sidecar latency/resource overhead, a control plane to operate) and that it's overkill for a few services. Mention eBPF/ambient mesh as the lower-overhead modern direction.

Official docs

- [Istio documentation](#)

80. Design a leaderboard / ranking system for millions of users in real time.

scalability

🗣 Spoken answer · ~88 s

A leaderboard ranks millions of users by score, supports updating scores, showing the top N, and finding a given user's rank, all in real time. A relational query with ORDER BY and COUNT is too slow at scale. The standard answer is Redis sorted sets: ZADD updates a user's score, ZREVRANGE returns the top N with scores, and ZREVRANK gives a user's rank, each in $O(\log N)$, backed by a skip list. One sorted set can hold many millions of members and tens of thousands of updates per second. Scaling beyond one node: shard by score range, so the top shard holds the highest scores and rank equals offset plus rank within the shard, or by user hash with scatter-gather for top N, merging each shard's top lists; for approximate global rank at huge scale, use score buckets with counts and estimate percentile. Durability: the sorted set is a derived view; persist scores in a database or event log and rebuild the leaderboard if lost, and replicate for availability. Variations: time-windowed boards, daily, weekly, and monthly, by using separate keys that expire; friends-only leaderboards, by fetching friends' scores and sorting; ties, broken by timestamp encoded in the score. Updates arrive through a stream, with batching and coalescing for high-frequency changes.

Likely follow-ups

- Why are Redis sorted sets a good fit?
- How do you rank a user when the leaderboard is sharded?

A real-time leaderboard needs fast rank queries (top-N and a user's own rank) under constant score updates — a poor fit for naive SQL ordering.

WHY SQL STRUGGLES

```
# "SELECT ... ORDER BY score DESC LIMIT 10" on 50M rows on every request,  
# plus "what's MY rank" = COUNT(scores > mine) → expensive at scale + churn.
```

REDIS SORTED SET (THE STANDARD ANSWER)

```
ZADD board score user      # update score:  $O(\log N)$   
ZREVRANGE board 0 9        # top 10:  $O(\log N + 10)$   
ZREVRANK board user        # a user's rank:  $O(\log N)$   
ZINCRBY board +10 user     # atomic increment  
# Sorted set keeps everything ranked incrementally – no full re-sort.
```

SCALING BEYOND ONE NODE

Sharding: partition by region/segment; merge top-Ns for a global board
Approximate global rank: exact rank among 50M is costly – bucket/percentile
("top 5%") is often enough for display
Time windows: daily/weekly boards → separate keys with TTL
Durability: Redis is the serving layer; persist authoritative scores in a DB too

Principal answer: Redis sorted sets are purpose-built for this — $O(\log N)$ updates and rank queries, atomic increments, no re-sorting. For global scale, shard by segment and merge top-Ns; provide approximate/percentile

rank for the long tail since exact rank among millions is expensive and rarely needed for display. Use separate TTL'd keys for time-windowed boards, and keep a durable copy of authoritative scores in a DB behind the Redis serving layer.

Official docs

- [Redis documentation](#)

81. How do you design APIs for scale and evolution? REST vs gRPC vs GraphQL, and versioning.

tradeoffs

Spoken answer · ~88 s

Designing APIs for scale and evolution means choosing the right style and planning for change. REST over HTTP and JSON is ubiquitous, cache-friendly, and easy to consume: great for public APIs and resource-oriented CRUD; weaknesses are over-fetching or under-fetching and multiple round trips. gRPC uses HTTP/2 and Protocol Buffers: compact binary, strongly typed contracts, code generation, streaming, and low latency, ideal for internal service-to-service calls; weaker for browsers without a proxy and harder to debug by hand. GraphQL lets clients specify exactly the fields they need in one request, which suits diverse clients, such as mobile and web, and aggregating many backends; costs include caching complexity, risk of expensive queries requiring depth and cost limits, and the N+1 problem needing batching like DataLoader. Scaling concerns across all: pagination, preferably cursor-based; rate limiting; idempotency keys for unsafe operations; caching headers and ETags; compression; timeouts; and consistent error formats. Evolution: prefer backward-compatible changes, adding optional fields and never repurposing or removing; tolerant readers; versioning when breaking changes are unavoidable, via URL path, header, or content negotiation; deprecation policies with sunset headers, usage monitoring, and long migration windows; schema registries and contract tests for consumer-driven compatibility. With protobuf, never reuse field numbers. Document with OpenAPI or schemas, and design APIs around client use cases.

Likely follow-ups

- Why prefer cursor-based pagination at scale?
- What kinds of changes are backward compatible?

API design choices affect performance, client ergonomics, and how painlessly you can evolve without breaking consumers.

```
REST/JSON:  ubiquitous, cacheable, human-readable
  → public APIs, broad client support
gRPC/protobuf: binary, fast, streaming, strong contracts
  → internal service-to-service, low latency, polyglot
GraphQL:    client picks exactly the fields → no over/under-fetching
  → varied clients (mobile/web) with different data needs; complex graphs
```

CHOOSING

Internal high-perf RPC → gRPC
Public, cache-friendly, simple → REST
Many clients, flexible queries → GraphQL (mind N+1 and query-cost limits)

VERSIONING & EVOLUTION

Backward compatibility first: add fields, don't remove/rename (additive changes)
Versioning: URI (/v2/), header, or content negotiation
Deprecation: announce → dual-run → sunset with timelines + metrics on old version
protobuf: reserve field numbers; never reuse → safe wire-compatible evolution

SCALE CONCERNS

Pagination (cursor > offset for large sets), filtering, field selection
Rate limiting + quotas, idempotency keys for writes
GraphQL: depth/complexity limits to prevent abusive expensive queries

Interview tip: Match protocol to use case — gRPC for internal low-latency RPC, REST for public/cacheable APIs, GraphQL when diverse clients need flexible field selection (while guarding against N+1 and runaway query cost). On evolution, emphasize additive, backward-compatible changes as the default (add fields, never repurpose them), cursor-based pagination for scale, and a real deprecation process with dual-running and usage metrics before sunsetting a version.



Official docs

- [gRPC documentation](#)

82. Design a system that needs both OLTP and OLAP. How do you avoid analytics killing production?

data

🗣 Spoken answer · ~83 s

OLTP handles many small transactional reads and writes with low latency; OLAP runs heavy scans and aggregations over large data. Running analytics directly on the production database can lock tables, evict cache, and saturate I/O and CPU, hurting customers. Approaches, from simple to robust. Read replica: point reporting queries at a replica, which isolates load from the primary, but replicas share the row-store design, so heavy queries are slow, and long queries can conflict with replication. Offload to a warehouse: stream changes from the OLTP database using change data capture, such as Debezium reading the write-ahead log, through Kafka into a columnar warehouse or lakehouse like BigQuery, Snowflake, or ClickHouse; analytics scale independently, columnar storage and compression make scans efficient, and production is untouched except for lightweight log reading. The trade-off is data freshness, from seconds to minutes. Batch ETL on a schedule is simpler but staler. HTAP databases serve both from one system, using a row store and a column store with replication between them, offering fresh data at the cost of vendor-specific complexity. Add controls: resource governance, query timeouts, and separate credentials; pre-aggregated tables or materialised views for dashboards; and data modelling in the warehouse with star schemas.

Likely follow-ups

- Why is change data capture preferable to nightly dumps?
- What is the cost of using a read replica for analytics?

Transactional (OLTP) and analytical (OLAP) workloads have opposite needs; running heavy analytics on your production DB will degrade or crash it. You separate them.

OLTP VS OLAP

OLTP: many small, fast read/write txns (orders, logins)
→ row store, normalized, low latency, high concurrency (Postgres/MySQL)
OLAP: few huge scan/aggregate queries (revenue by region last year)
→ columnar, denormalized, throughput-oriented (BigQuery/Snowflake/ClickHouse)
A giant analytical scan on the OLTP DB locks/I/O-starves production.

SEPARATION PATTERNS

1. Read replica for analytics: cheap isolation; replica absorbs heavy reads (still row-store – OK for moderate analytics, not big aggregations)
2. ETL/ELT to a warehouse: batch export OLTP → columnar warehouse
3. CDC (change data capture): stream OLTP changes (Debezium → Kafka) → warehouse / lake in near-real-time, zero load on the primary
4. HTAP systems (TiDB, SingleStore): both in one – convenient, niche trade-offs

MODERN PATTERN

CDC from OLTP → Kafka → stream/batch into a columnar lakehouse.
Keeps production isolated, gives analysts a query-optimized copy, near-real-time.

Principal answer: Never run OLAP on your OLTP database — the access patterns are opposite (row vs columnar, point vs scan). Isolate with a read replica for light analytics, or for real scale, stream changes via CDC (Debezium/Kafka) into a columnar warehouse or lakehouse. CDC is the modern default: near-real-time, zero query load on the primary, and the analysts get a store actually optimized for big aggregations. Mention HTAP databases as a unified-but-niche alternative.

Official docs

- [PostgreSQL: Logical replication](#)

83. Design a deployment system supporting canary, blue-green, and instant rollback.

reliability

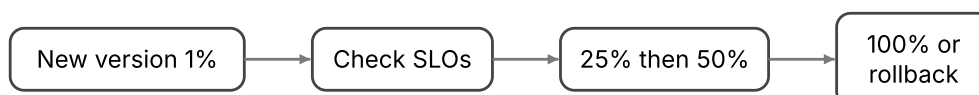
Spoken answer · ~85 s

A deployment system should make releases routine and reversible. Foundation: immutable, versioned artefacts, one container image per commit, promoted unchanged through environments, with configuration externalised. Rolling update: replace instances gradually, simple, but old and new versions coexist and rollback means another rollout. Blue-green: run two full environments; deploy to the idle green, test it, then switch traffic at the load balancer or service selector; rollback is flipping back instantly, at the cost of double capacity and care with shared databases. Canary: send a small percentage of traffic, say 1 then 5 then 25 then 100 percent, to the new version while comparing metrics against the baseline, such as error rate, latency, and business KPIs; automated analysis, as in Argo Rollouts or Flagger, promotes or aborts, limiting the blast radius. Instant rollback: keep the previous version warm and traffic-shiftable, automate rollback on SLO breach, and use feature flags to disable functionality without redeploying. Database changes are the hard part: use the expand and contract pattern, so schema changes are backward compatible with both versions, deployed separately from code, never dropping or renaming in the same release. Control plane: GitOps for desired state, approvals and policies, deployment freezes, audit logs, and a history of versions.

Likely follow-ups

- Why must schema changes use expand and contract?
- How does automated canary analysis decide to abort?

Canary rollout



Safe deployment minimizes blast radius and enables fast recovery. The core strategies trade infrastructure cost against risk control.

Rolling: replace instances gradually (default; some mixed-version time)
Blue-Green: full parallel env (green), switch traffic at once, old (blue) on standby
+ instant rollback (flip back) - 2x infra during deploy
Canary: route a small % to the new version, watch metrics, ramp gradually
+ smallest blast radius, real-traffic validation - slower, needs good metrics

CANARY FLOW (THE GOLD STANDARD)

1. Deploy new version alongside old
2. Route 1% → 5% → 25% → 100% in steps
3. At each step: compare canary vs baseline (error rate, latency, business KPIs)
4. Automated analysis → promote if healthy, auto-rollback if not

ROLLBACK & SAFETY

Instant rollback: keep previous version ready (blue-green flip, or redeploy prev image)
DB migrations: must be backward-compatible (expand/contract) – you can't roll back data
→ decouple schema change from code: add column → deploy code → backfill → drop later
Feature flags: decouple release from deploy → flip off without redeploying

Interview tip: Canary is the strongest answer — validate on real traffic with automated metric comparison and auto-rollback, minimizing blast radius. The detail that separates seniors: database migrations break the rollback story, so use the expand/contract (backward-compatible) pattern and decouple schema changes from code deploys. Pair deployments with feature flags so you can disable a feature instantly without a redeploy. Always have a tested, fast rollback path.

Official docs

- [Kubernetes: Deployments](#)
- [Google SRE Workbook: Canarying Releases](#)
- [Argo Rollouts](#)

84. **Design a multi-tenant SaaS platform. Compare isolation models and the noisy-neighbor problem.**

platform

🗣 Spoken answer · ~88 s

Multi-tenant platforms trade isolation against cost. Isolation models. Pooled: all tenants share the same application and database tables, distinguished by a tenant ID; cheapest, scales to many small tenants, easiest to operate, but weakest isolation, so a bug can leak data, and one tenant can degrade others. Mitigate with row-level security, enforced tenant context in the data layer, and per-tenant keys. Bridge: shared compute with a schema or database per tenant; stronger data separation, per-tenant backup and restore, and easier custom changes, but migrations multiply. Silo: dedicated stack or cluster per tenant; strongest isolation, compliance and data residency friendly, and a tenant-specific performance guarantee, at much higher cost and operational burden. Most platforms are tiered: small customers pooled, enterprise customers siloed. The noisy-neighbour problem: one tenant's heavy usage consumes shared resources. Controls: per-tenant rate limits and quotas, fair queueing and weighted scheduling in job systems, resource limits in containers, connection pool caps per tenant, query timeouts, and separate queues or worker pools for large tenants; monitor usage per tenant and alert on anomalies; consider cell-based architecture, deploying independent cells and assigning tenants to cells, which limits blast radius and enables gradual rollouts. Also tenant-aware caching keys, onboarding automation, per-tenant cost attribution, and the ability to move a tenant between tiers or shards.

Likely follow-ups

- What does cell-based architecture give you?
- How do you prevent cross-tenant data leaks in a pooled model?

Multi-tenancy serves many customers from shared infrastructure. The central trade-off is isolation (security, blast radius, noisy-neighbor) vs cost/operational efficiency.

Isolation models (most shared → most isolated):

Shared everything (pooled): one DB, tenant_id column on every row

+ cheapest, easiest to operate - weakest isolation, noisy-neighbor risk

Shared app, DB-per-tenant (bridge):

+ good data isolation, per-tenant backup/restore - more DBs to manage

Silo (stack per tenant):

+ strongest isolation, per-tenant customization - costliest, ops-heavy

TENANT ISOLATION CONCERNS

Data: enforce tenant_id everywhere (a missed filter = cross-tenant leak!)

→ row-level security, scoped data-access layer, tested rigorously

Noisy neighbor: one tenant's load degrades others

→ per-tenant quotas, rate limits, resource caps, fair scheduling

Blast radius: a bad deploy/incident shouldn't take down all tenants

→ cell-based architecture (shard tenants into isolated cells)

- # Partition tenants into independent "cells" (each a full stack serving a subset).
- # A cell failure affects only its tenants. Big tenants can get a dedicated cell.
- # Best balance of isolation + efficiency at scale.

Principal answer: Frame it as an isolation/cost spectrum (pooled → bridge → silo) and pick per requirements — enterprise customers may demand a dedicated DB or cell; SMB tenants pool cheaply. The two failure modes to address explicitly: cross-tenant data leakage (one missed tenant_id filter is a breach — enforce via row-level security and a scoped data layer) and noisy neighbors (per-tenant quotas/rate limits). At scale, cell-based architecture caps blast radius while keeping efficiency — the answer that signals real SaaS experience.

Official docs

- [Kubernetes: Multi-tenancy](#)

85. Design a photo/media-sharing service (like Instagram). End-to-end from upload to feed.

scalability

Spoken answer · ~85 s

Requirements: users upload photos, follow others, view a ranked feed, and like and comment; read-heavy at hundreds of millions of users. Upload path: the client requests a pre-signed URL and uploads directly to object storage, avoiding the application tier; an event triggers an asynchronous pipeline that validates the file, strips sensitive metadata, generates multiple resolutions and thumbnails, runs content moderation, and records metadata. Storage: images in object storage fronted by a CDN, which serves nearly all image bytes; metadata such as owner, caption, and time in a sharded database, partitioned by user ID; the social graph in a store optimised for follower lookups; likes and comments in separate, write-optimised stores with counters sharded. Feed: hybrid fan-out; on posting, push the post ID to followers' feed caches for normal users, while celebrities' posts are pulled and merged at read time; the feed service reads the ID list, hydrates posts from caches, and applies ranking by recency, relationship strength, and predicted engagement. Pagination uses cursors. Other components: search and hashtags via an inverted index, notifications through queues, and discovery and recommendations. Scale: image storage grows fast, so use lifecycle tiering and compression such as WebP or AVIF; the CDN hit ratio is the key cost lever.

Likely follow-ups

- Why upload directly to object storage?
- How does the CDN reduce cost and latency?

A media-sharing service combines object storage, async processing, CDN delivery, and a feed system — a great integrative question.

Upload: client → presigned URL → object store (original)
→ async: generate thumbnails/sizes, extract metadata, moderation

→ store renditions → distribute to CDN

Serve: feed request → metadata (posts) + CDN URLs for media
→ client loads images from nearest CDN edge

UPLOAD PATH

Direct-to-storage via presigned URL (don't proxy big files through app servers)
Async pipeline: resize to multiple resolutions, strip EXIF, content moderation
Metadata (caption, author, timestamp, media refs) → DB

STORAGE & DELIVERY

Originals + renditions → object store (S3/GCS), tiered by age
Serve via CDN (images are static, cacheable → huge hit ratio)
Responsive: deliver the right size per device (don't send 4K to a phone)

FEED (TIES TO FAN-OUT DESIGN)

Feed = post metadata + CDN media URLs, hybrid fan-out (push/pull for celebs)
Metadata DB sharded by user; media is just references → small feed payloads
Counts (likes/views) via striped counters; comments paginated separately

Interview tip: Hit the four pillars: (1) direct-to-storage upload via presigned URLs (never proxy large media through app servers), (2) an async processing pipeline for renditions + moderation, (3) CDN delivery since images are the ideal cacheable static asset, and (4) a feed that stores only metadata + media references with hybrid fan-out. This question rewards composing patterns from earlier problems — call those connections out explicitly.



Official docs

- [Cloudflare Learning: What is a CDN?](#)
- [Microsoft: Cache-Aside pattern](#)

86. How do you estimate cost and design for cost-efficiency (FinOps) in a system design?

tradeoffs

🗣 Spoken answer · ~82 s

Cost is a design dimension, so I estimate and optimise it alongside performance. Estimate: translate the load model into resources, compute hours, storage, network egress, and managed-service requests, using unit prices; for instance, 10 TB of stored data, 20 TB a month of egress, and a given instance fleet, and sum into a monthly figure, noting which line dominates. Typically the big drivers are data transfer, especially cross-zone and egress, storage growth, and over-provisioned compute. Design for efficiency. Right-size instances and autoscale on demand instead of provisioning for peak; use commitments such as savings plans or reserved capacity for steady baseline, and spot instances for fault-tolerant batch and stateless work. Choose storage tiers with lifecycle rules, and compression and columnar formats; set retention policies for logs and metrics, and sample traces. Reduce data movement: keep chatty services in the same zone, cache and use a CDN to cut origin egress, and compress payloads. Pick architectures sensibly: serverless suits spiky, low-volume workloads, but can be pricey at sustained high volume; managed services reduce operational cost but cost more per unit. Avoid over-engineering for scale you don't have, such as multi-region when single-region suffices for the required availability.

Likely follow-ups

- Which cost drivers are most often overlooked?
- What are unit economics in this context?

Senior designs account for cost, not just correctness and scale. Showing cost-awareness distinguishes a principal-level answer.

WHERE THE MONEY GOES

Compute: often the biggest line; right-size, autoscale, use spot for fault-tolerant work
Storage: tier aggressively (hot/warm/cold/archive); lifecycle policies
Data transfer: egress (esp. cross-region & internet) is sneakily expensive
Managed services: convenient but premium-priced; build-vs-buy math
Idle/over-provisioned resources: the silent budget killer

COST-EFFICIENCY LEVERS

Caching/CDN: cuts compute + egress (cache hit is far cheaper than origin)
Autoscaling: match capacity to demand; scale to zero for spiky/dev workloads
Spot/preemptible: 60–90% cheaper for interruptible batch/stateless work
Storage tiering: move cold data to cheap tiers automatically
Data locality: keep traffic in-region/in-AZ to dodge egress charges
Right-sizing: measure actual usage; trim over-provisioned requests

THE TRADE-OFFS

Cost vs reliability: spot saves money but can be reclaimed → only for tolerant work
Cost vs latency: cheaper cold storage = slower access
Cost vs simplicity: a hand-rolled cheaper solution may cost more in eng time
Optimize for TOTAL cost of ownership, including engineering effort.

Principal answer: Weave cost in as a first-class constraint: do a rough cost estimate (compute + storage + egress are the big three) the same way you do a capacity estimate. Lean on caching/CDN (cuts both compute and pricey egress), autoscaling, spot instances for interruptible work, and storage tiering. Flag data egress as the commonly-overlooked cost. Frame every cost lever as a trade-off (spot vs reliability, cold storage vs latency) and optimize total cost of ownership, engineering time included.

Official docs

- [AWS Well-Architected: Cost optimization pillar](#)

87. Design observability for a large distributed system. Logs, metrics, and traces together.

reliability

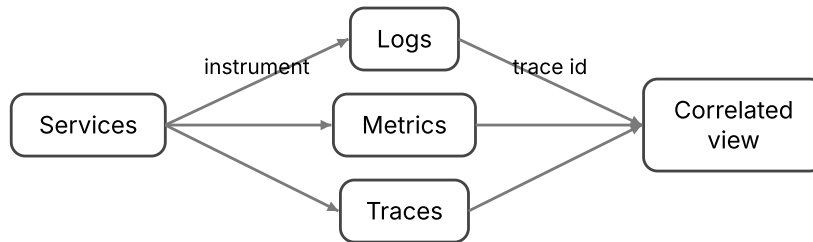
Spoken answer · ~85 s

Observability lets you understand a system's internal state from its outputs, using three complementary signals. Metrics: numeric time series, cheap to store, ideal for dashboards, alerting, and SLOs; use the RED method for services, rate, errors, duration, and USE for resources. Logs: detailed event records, structured as JSON with consistent fields, including trace IDs, giving context for debugging specific events. Traces: follow a request across services as spans, showing latency breakdown and dependency paths, using context propagation through standards like OpenTelemetry. The power comes from correlation: an alert on a metric leads to exemplars or a time window, then to traces showing the slow service, then to logs for the exact error, all linked by trace and request IDs. Architecture: instrument with OpenTelemetry SDKs and auto-instrumentation; collectors or agents gather and process telemetry, adding metadata, batching, filtering, and sampling; backends store each signal: Prometheus-compatible store for metrics, Loki or Elasticsearch for logs, Tempo or Jaeger for traces; a unified UI like Grafana. Scale and cost controls: limit metric label cardinality, use tail-based sampling to keep error and slow traces, tier log retention, and drop noisy debug logs. Alerting should be based on symptoms and SLO burn rate, not every cause, with runbooks linked from alerts.

Likely follow-ups

- How do trace IDs tie the three signals together?
- Why use tail-based sampling?

🚧 Three pillars



Observability is the ability to understand a system's internal state from its outputs — essential for operating distributed systems where failures are emergent. The three pillars work together.

```
Metrics (what & how much): aggregated numbers, cheap, alerting
Logs (what happened): discrete events, detailed, debugging
Traces (where & why slow): request path across services, latency breakdown

    ↓           ↓           ↓
Time-series DB  Log store  Trace store
    ↓           ↓           ↓
Unified query / dashboards / alerting
```

WHEN YOU USE EACH

```
Metrics: dashboards + alerting ("error rate > 1%", "P99 > 500ms") – start here
Traces: "this request is slow – WHICH service/hop?" – distributed latency
Logs: "what exactly happened in that failing request" – deepest detail
Flow: alert (metric) → trace (locate) → logs (root cause). They chain.
```

KEY PRACTICES

```
Correlation IDs: propagate a request/trace id through all services → tie the 3 together
Structured logs: JSON, queryable, not free-text grep
Golden signals: latency, traffic, errors, saturation (alert on these)
Sampling traces: 100% is expensive → head/tail sample (keep slow/error traces)
Cardinality discipline: high-cardinality labels blow up metric stores
```

Interview tip: Explain how the three pillars chain in an incident: a metric alert tells you something is wrong, a trace localizes which service/hop, and logs give the root cause. The glue is correlation/trace IDs propagated everywhere so you can pivot between them. Alert on the four golden signals, use structured logs, sample traces (keep errors/slow ones), and watch metric cardinality. OpenTelemetry as the vendor-neutral instrumentation standard is a good name-drop.

📖 Official docs

- [OpenTelemetry documentation \(observability design\)](#)

88. Design a system with WebSockets for millions of concurrent connections.

networking

🗣 Spoken answer · ~85 s

Supporting millions of concurrent WebSocket connections is mainly a problem of connection state and fan-out. A tuned server can hold hundreds of thousands of mostly idle connections, limited by memory per connection, file descriptors, and kernel settings, so one million connections needs on the order of ten to twenty gateway nodes with headroom, and ten million needs a hundred or more. Gateway tier: lightweight, event-driven servers in Go, Java with Netty, Elixir, or Node, doing little more than terminating the connection, authenticating, and relaying messages; keep business logic in separate backend services. Load balancing: L4 or L7 balancers with WebSocket support, using least-connections; deployments must drain connections gradually to prevent reconnect storms. Routing: maintain a connection registry in a shared store such as Redis, mapping user to gateway node, and use a pub/sub or message bus so backend services can deliver messages to the right node, or have each gateway subscribe to topics for its local users. Reliability: heartbeats to detect dead connections, client reconnect with exponential backoff and jitter, message IDs and acknowledgements with resume from the last seen offset, and offline queues. Protection: per-connection rate limits, message size limits, backpressure for slow consumers by dropping or disconnecting, and authentication with token refresh.

Likely follow-ups

- Why do rolling deployments cause reconnect storms?
- How do you deliver a message to a user on a specific gateway?

Holding millions of persistent WebSocket connections is a distinct scaling challenge: the connections themselves are the constrained resource, and the connection tier is stateful.

```
Clients ⇌ (WSS) ⇌ Connection/Gateway tier (stateful, holds sockets)
                        ↓ registry: user → which gateway holds the socket (Redis)
                        Pub/Sub (Redis/Kafka) for cross-gateway message routing
                        ↓
                        Backend services (stateless) publish events
```

THE SCALING CHALLENGES

Each connection consumes memory + a file descriptor → tune OS limits (ulimit, ephemeral ports)
Connections are LONG-LIVED → LB must support sticky/WS, can't freely rebalance
State: a gateway holds specific users' sockets → need a registry to find them
A single box maxes out around tens-hundreds of K conns → must scale horizontally

CROSS-NODE MESSAGE ROUTING

```
# User A (on gateway 1) messages User B (on gateway 3).
# Gateway 1 publishes to pub/sub → gateway 3 (subscribed for B) pushes to B's socket.
# Registry maps user → gateway so you route to the right node.
```

Heartbeats/ping-pong → detect dead connections, free resources
 Reconnect with backoff + resume (sync missed messages since last ack)
 Graceful gateway drain on deploy (migrate connections, don't drop all at once)

Principal answer: The key realization is that the connection tier is stateful and the sockets are the bottleneck resource — so you scale it as a dedicated horizontal layer, tune OS limits, and use a registry (user→gateway) plus pub/sub to route messages between nodes. Handle the operational realities: sticky LB for long-lived connections, heartbeats to reap dead ones, reconnect-with-resume for mobile clients, and graceful drain on deploy so a rollout doesn't drop millions of connections at once.

Official docs

- [RFC 6455: The WebSocket Protocol](#)

89. Walk through a build-vs-buy decision for a major system component.

tradeoffs

Spoken answer · ~85 s

A build-versus-buy decision weighs strategic value against cost and risk. Start by asking whether the component is a differentiator: if it directly creates competitive advantage, building has a stronger case; if it's commodity infrastructure, like authentication, payments, email delivery, or search, buying or adopting open source usually wins, because specialists do it better and faster. Evaluate total cost of ownership, not licence price alone: for build, engineering time, ongoing maintenance, on-call, security patching, scaling, and the opportunity cost of what those engineers would otherwise build; for buy, licence or usage fees, integration work, customisation limits, and growth in price as volume increases. Consider time to market: buying is usually faster, and speed can decide. Assess fit: do the requirements match the product, how much customisation is needed, and what is the vendor's roadmap. Risks of buying: vendor lock-in, outages outside your control, data residency and compliance, price changes, and the vendor disappearing; mitigate with abstraction layers, data export, contracts, and exit plans. Risks of building: underestimated effort, key-person dependency, and a slowly rotting internal tool. Intermediate options: open source self-hosted, managed open source, or buying and extending. Decide with a documented proposal, a small proof of concept, a reversibility assessment, and explicit criteria.

Likely follow-ups

- How do you assess total cost of ownership for building?
- How do you reduce lock-in when buying?

Senior engineers don't build everything — knowing when to buy/adopt vs build is a key judgment. It's a TCO and focus decision, not an ego one.

THE DECISION FRAMEWORK

Is it core/differentiating to YOUR business? → lean BUILD (it's your edge)
Is it commodity (auth, queues, search, monitoring)? → lean BUY/adopt
What's the TOTAL cost: build + maintain + operate forever, vs license + integrate?
Time-to-market: buy is faster now; build is a long-term investment
Opportunity cost: every engineer building commodity infra isn't building product

HIDDEN COSTS OF BUILD

It's not just initial dev – it's maintenance, on-call, security patching, scaling, docs, and the bus-factor risk, forever. Most "we'll build it" underestimates the perpetual carrying cost by a lot.

HIDDEN COSTS OF BUY

Vendor lock-in, pricing at scale, customization limits, integration effort, dependency on vendor reliability/roadmap, data egress/portability.
→ Mitigate with open standards / abstraction layers / exit strategy.

Interview tip: Anchor on "is this core to our differentiation?" Build the thing that IS your product; buy or adopt open-source for commodity infrastructure (don't build your own message queue or monitoring stack to save a license fee — you'll pay far more in perpetual maintenance and opportunity cost). Be explicit that build cost is forever (maintenance, on-call, security), and de-risk buy decisions against lock-in with abstraction layers and an exit plan. This signals business judgment, not just engineering.



Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

90. **You're given vague requirements and limited time in the interview. How do you scope and drive the design?**

tradeoffs

🗣 Spoken answer · ~86 s

With vague requirements and limited time, my job is to create structure quickly and show judgement. First, spend the opening minutes clarifying: ask a handful of high-leverage questions about the core use case, who the users are, the scale, and the key constraint such as latency, consistency, or availability; if the interviewer says to decide, state assumptions explicitly, such as 10 million daily users and a read-heavy workload, and move on. Second, define scope out loud: list the must-have features, name the ones I'm deliberately leaving out, and confirm. Third, agree on a time plan: for example, five minutes on requirements, ten on high-level design, fifteen on a deep dive, and the rest on scaling, failure, and trade-offs, adjusting to the interviewer's interests. Fourth, draw a simple end-to-end design early, so there's something concrete, then ask which area to go deeper on, or pick the riskiest component myself. Throughout, narrate reasoning, compare options with trade-offs, and tie decisions to the stated requirements. Keep a visible list of open issues, and park them with a note such as 'we can come back to multi-region if time permits'. If I'm running out of time, summarise the design, the main trade-offs, the known limitations, and what I'd do next.

Likely follow-ups

- What are the most valuable clarifying questions?
- How do you recover when you realise a design choice was wrong?

Most system design interviews are deliberately open-ended and time-boxed. Managing scope and driving the conversation is itself a graded skill — often the difference between levels.

DRIVE, DON'T WAIT

1. Clarify & scope FIRST (5 min): functional + non-functional reqs, scale, and EXPLICITLY state what's out of scope. Get agreement.
2. Propose a plan: "I'll do estimation, then high-level design, then deep-dive the 2 hardest parts. Sound good?" → you're steering.
3. Time-box yourself: don't spend 20 min on the easy CRUD part.
4. Signpost trade-offs as you go: every choice has alternatives – say them.

MANAGE SCOPE ACTIVELY

Breadth first (a complete simple design) THEN depth (deep-dive hard parts).
A finished simple system > an unfinished elaborate one.
Ask "is it more valuable to go deeper here or cover X?" → collaborative scoping.
Watch the interviewer's cues – they'll steer you to what they want to probe.

SENIORITY SIGNALS

- Owning the structure (not waiting to be prompted)
- Stating assumptions explicitly when info is missing (then proceeding)
- Naming trade-offs and justifying choices, not presenting one "right" answer
- Knowing what to leave out and saying why

Principal answer: Treat the ambiguity as the test. Open by clarifying requirements and explicitly bounding scope, then propose and narrate a plan so you're visibly driving. Go breadth-first to a complete simple design, then deep-dive the genuinely hard parts, time-boxing the trivial bits. Continuously surface trade-offs and stated assumptions rather than hunting for one "correct" architecture. Senior interviewers are grading whether you can impose structure on an under-specified problem — so impose it.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

91. **How do you define functional and non-functional requirements in the first five minutes of a design interview?**

fundamentals

 Spoken answer · ~57 s

I begin with a short clarification phase. Functional requirements are the core actions users perform; I list three to five and agree which are in scope. Then I ask about non-functional requirements, which usually shape the architecture more: expected users and request rates, read-to-write ratio, latency targets at the ninety-ninth percentile, availability goal, data durability, consistency needs and geographic spread. I turn vague answers into numbers, for example ten million daily users and a hundred thousand requests per second at peak. I state assumptions out loud and note what is out of scope so the interviewer can correct me. This prevents building the wrong system and gives me criteria to justify later choices, such as choosing eventual consistency because availability matters more. I keep the notes visible on the whiteboard and refer back when making trade-offs.

Likely follow-ups

- Which non-functional requirement drives architecture most?
- How do you handle vague answers?

Split what the system does from how well it must do it, and confirm both before drawing anything.

Functional: post, read, search, delete
Non-functional: latency p99, availability, durability,
consistency, scale (QPS, GB), cost
Out of scope: say it explicitly

Principal answer: Write numbers next to every non-functional requirement.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

 Spoken answer · ~60 s

After sketching a high-level design, I walk the request path and estimate the load each component must handle, comparing it with realistic capacity: a single database node might handle a few thousand writes per second, a cache hundreds of thousands of reads, a network link a fixed bandwidth. The component that exceeds its capacity first is the bottleneck and gets my attention. Stateless application servers scale horizontally and are rarely the problem; databases, hot keys, single leaders and shared queues usually are. I propose targeted remedies, like caching reads, sharding writes, batching or moving work to asynchronous processing, and state their cost. I also consider the second bottleneck that appears once the first is fixed. Defending the choice means showing the arithmetic and explaining what happens at ten times the load, which demonstrates that the design has headroom and a path to scale.

Likely follow-ups

- Which components usually bottleneck first?
- What do you do after fixing the first bottleneck?

Estimate load per component, find the one that saturates first, and focus the design effort there.

```
QPS -> app tier (stateless, scales out)
      -> cache (hit rate?)
      -> DB (writes: single leader limit)
Bottleneck = first component to hit its ceiling
```

Principal answer: Name the bottleneck before you optimise.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

93. **What latency numbers should every engineer know, and how do you use them in estimates?**

fundamentals

 Spoken answer · ~62 s

Approximate latency orders of magnitude let me reason about feasibility quickly. Reading from main memory costs around a hundred nanoseconds, an SSD random read about a hundred microseconds, a round trip inside a data centre roughly half a millisecond, a spinning disk seek around ten milliseconds, and a trip between continents a hundred milliseconds or more. From these I derive design implications. A page that requires twenty sequential cross-region calls cannot meet a hundred millisecond budget. Serving hot data from memory is a thousand times faster than from disk, which justifies caches. Sequential reads are much faster than random ones, which favours log-structured storage. Network calls dominate, so I reduce round trips by batching and parallelising. I also remember that the tail matters: with many dependencies, the slowest one defines the response, so I discuss timeouts, hedged requests and fan-out limits. The exact figures matter less than the ratios.

Likely follow-ups

- Why do tail latencies amplify with fan-out?
- What do these numbers imply for caching?

Memory is nanoseconds, SSD microseconds, network within a datacenter sub-millisecond, cross-region tens to hundreds of milliseconds.

L1/L2 cache	~1-10 ns
RAM	~100 ns
SSD read	~100 us
Same-DC RTT	~0.5 ms
HDD seek	~10 ms
Cross-region	~50-150 ms

Principal answer: Use them to rule out designs fast.

 Official docs

- [Azure Well-Architected: Performance efficiency](#)

94. Explain the outbox pattern. Why is it used to publish events reliably?

tradeoffs

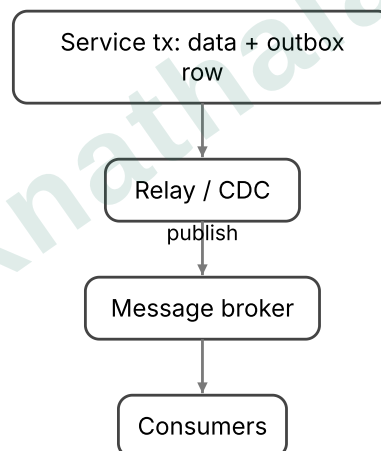
🗣 Spoken answer · ~63 s

A service often needs to update its database and publish an event. Doing both separately risks inconsistency: the database commits but the publish fails, or the reverse. The outbox pattern fixes this. In one local transaction the service writes the business data and also inserts an event row into an outbox table in the same database. A separate relay process, either a poller or a change data capture connector reading the log, reads new outbox rows and publishes them to the message broker, marking them as sent. Because the event is persisted atomically with the change, it is never lost; if the relay crashes it resumes. The delivery is at least once, so duplicates can occur, and consumers must be idempotent, for example by tracking event identifiers. The cost is extra latency and an additional component, and the outbox table needs cleanup. It is a standard building block for reliable event-driven systems.

Likely follow-ups

- Why can't you just publish after commit?
- How do consumers handle duplicates?

🗣 Outbox pattern



Write the business change and the event in one local transaction, then a relay publishes outbox rows to the broker, avoiding dual-write inconsistency.

```
Service --(1 tx)--> DB: orders + outbox
Relay/CDC --(2)--> Broker --> consumers
at-least-once => consumers idempotent
```

Principal answer: Dual writes cannot be made atomic.

📖 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

95. **Compare orchestration and choreography for coordinating microservices.**

tradeoffs

 Spoken answer · ~59 s

In orchestration, a central workflow engine or service explicitly calls each participant in sequence and tracks state, handling retries, timeouts and compensation. The advantages are clarity, easy monitoring of where a process is, and simpler reasoning about failures; the drawbacks are a central component that can become a bottleneck or accumulate logic, and tighter coupling to it. In choreography, services publish events and others react independently, with no central controller. That gives loose coupling and extensibility, since new consumers can join without changing producers, but the overall flow becomes implicit, hard to trace and hard to change, and cyclic dependencies can appear. I choose choreography for simple, fan-out style reactions, such as sending a notification after an order, and orchestration for long, multi-step business transactions with compensation, such as booking and payment. Tools like Temporal or Step Functions provide durable orchestration.

Likely follow-ups

- When does choreography become hard to manage?
- What does a workflow engine give you?

Orchestration uses a central coordinator that directs steps; choreography has services react to events. Choose by visibility versus coupling.

Orchestration: Orchestrator -> A -> B -> C (central state)
Choreography: A --event--> B --event--> C (no owner)

Principal answer: Complex flows need a visible owner.

 Official docs

- [Martin Fowler: Microservices](#)

96. What is the saga pattern and how do you handle compensation?

tradeoffs

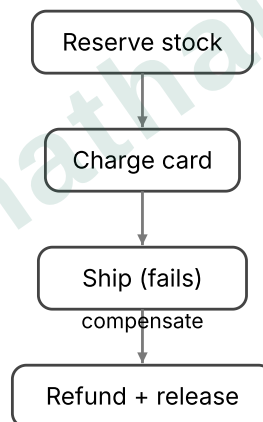
🗣 Spoken answer · ~59 s

Distributed transactions across services with two-phase commit are slow and fragile, so sagas model a business process as a sequence of local transactions. Each step commits independently and publishes an outcome. If a later step fails, previously completed steps are undone by compensating transactions, such as refunding a payment or releasing reserved inventory. Compensation is a semantic undo, not a database rollback, so it must be designed carefully: it may not restore the exact previous state, some actions cannot be undone, such as an email already sent, and compensations can fail and need retries. Every step and compensation must be idempotent, and the saga state must be persisted so it can resume after crashes. Isolation is weaker than in ACID: other transactions might see intermediate states, so I use techniques like pending states and reservations. Sagas can be orchestrated or choreographed.

Likely follow-ups

- Why must compensations be idempotent?
- What isolation problems do sagas have?

🗣 Saga compensation



A saga splits a distributed transaction into local transactions with compensating actions to undo completed steps on failure.

```
Reserve stock -> Charge card -> Create shipment
fail at 3 => Refund card -> Release stock
(each step idempotent, each has a compensation)
```

Principal answer: Compensation is semantic undo, not rollback.

📖 Official docs

- [Microsoft: Saga pattern](#)

97. **Compare batch processing, stream processing and the Lambda and Kappa architectures.**

tradeoffs

 Spoken answer · ~59 s

Batch processing handles large bounded datasets periodically, with high throughput and easy correctness, but results are delayed. Stream processing handles events continuously with low latency but needs careful handling of ordering, late data and state. The Lambda architecture runs a batch layer for accurate recomputation and a speed layer for fresh approximate results, merging them at query time, which gives both properties but duplicates logic in two codebases, a maintenance burden. The Kappa architecture simplifies by treating everything as a stream, with a durable replayable log such as Kafka; when logic changes, you reprocess history by replaying the log through the new job. Modern engines like Flink and Spark Structured Streaming can handle both bounded and unbounded data with the same API, making Kappa-style designs practical. I choose based on latency requirements, data volume, team skills and whether reprocessing is needed.

Likely follow-ups

- What is the drawback of Lambda?
- How does replay support Kappa?

Batch is simple and accurate over bounded data; streaming gives low latency; Lambda runs both, Kappa uses streaming with replay only.

Lambda: events -> batch layer + speed layer -> serving

Kappa: events -> log (replayable) -> stream job -> serving

Principal answer: Prefer one code path if you can.

 Official docs

- [AWS Lambda Developer Guide](#)

98. **Long polling, Server-Sent Events, WebSockets and webhooks: how do you choose?**

tradeoffs

 Spoken answer · ~57 s

Simple polling is easy but wasteful and slow. Long polling holds a request open until data is available, giving near real-time updates over plain HTTP, and works through almost any proxy, but costs a connection per client and repeated reconnects. Server-Sent Events provide a one-way stream from server to browser over HTTP with automatic reconnection and event identifiers, ideal for notifications, dashboards and feeds. WebSockets give a persistent bidirectional channel with low overhead, suited to chat, collaborative editing and games, but require stateful connection handling, load balancer support, heartbeats, and a reconnection and resume strategy. Webhooks are for server-to-server notification: the provider calls a customer's URL, needing retries, signatures and idempotency. At scale, connection count and fan-out matter, so I plan connection gateways, presence storage and pub/sub backplanes, and choose the simplest mechanism that meets the requirement.

Likely follow-ups

- When is SSE better than WebSocket?
- What must webhook receivers handle?

Match direction and frequency: SSE for one-way server push, WebSockets for bidirectional low-latency, long polling as fallback, webhooks for server-to-server.

Poll: client asks repeatedly
Long poll: server holds request
SSE: one-way stream over HTTP
WebSocket: full duplex
Webhook: server calls your URL

Principal answer: Consider proxies, scale and reconnection.

 Official docs

- [RFC 6455: The WebSocket Protocol](#)

99. **How do you choose between a relational database and a document store for a new service?**

tradeoffs

 Spoken answer · ~57 s

I start from how the data will be accessed. If the domain has many relationships, needs multi-row transactions, constraints, and flexible ad hoc queries and reporting, a relational database is the safer default and scales further than many assume with replicas and partitioning. A document store fits when the main access pattern is fetching a self-contained aggregate by key, such as a user profile or product page, when the schema varies between records, and when I need easy horizontal scaling through built-in sharding. Document stores make cross-document transactions and joins harder, and denormalisation leads to update anomalies that the application must manage. I also weigh the team's experience, operational tooling and managed service availability. Many systems use both, with the relational database as the source of truth and a document or search store as a read-optimised projection.

Likely follow-ups

- What does a document store make harder?
- When is a relational database enough at scale?

Pick by access patterns, relationships, consistency needs and query flexibility, not by trend.

Relational: joins, constraints, transactions, ad-hoc queries
Document: aggregate-per-read, flexible schema, horizontal scale

Principal answer: Model reads first.

 Official docs

- [Azure: Choose a data store \(relational vs NoSQL\)](#)

100. What are the trade-offs of using GraphQL, REST and gRPC for service APIs?

tradeoffs

 Spoken answer · ~55 s

REST uses resources and standard HTTP semantics, is easy to understand, cache and debug, but can lead to over- or under-fetching and many round trips for complex screens. GraphQL lets clients request exactly the fields they need in one request and evolves without versioning, which suits varied front ends, but introduces complexity: query cost control, caching difficulty, N+1 resolver problems and authorisation per field. gRPC uses protocol buffers over HTTP/2, giving compact messages, strong typing with generated clients, streaming and low latency, which is ideal for internal service communication, but is less convenient for browsers and harder to inspect. A common architecture uses REST or GraphQL at the edge, perhaps through a backend-for-frontend, and gRPC internally. I weigh client diversity, performance, tooling, team familiarity, and the need for caching and public accessibility.

Likely follow-ups

- What is the N+1 problem in GraphQL?
- Why use gRPC internally?

REST is simple and cacheable; GraphQL lets clients shape responses; gRPC is efficient and typed for internal calls.

Public/web: REST or GraphQL (BFF)

Internal service-to-service: gRPC (protobuf, streaming)

Principal answer: Different layers can use different styles.

Official docs

- [gRPC documentation](#)

101. **How do you choose a cache strategy: cache-aside, read-through, write-through or write-back?**

tradeoffs

 Spoken answer · ~58 s

With cache-aside, the application checks the cache, loads from the database on a miss and populates the cache; writes go to the database and invalidate or update the entry. It is simple, resilient to cache failure and the most common. Read-through hides this logic inside the cache layer. Write-through writes to the cache and database synchronously, so the cache is always fresh for written keys, at the cost of write latency and caching data that may never be read. Write-back writes to the cache first and flushes to the database asynchronously, giving excellent write performance and batching, but data can be lost if the cache fails before flushing, so it needs replication or durable logs. I choose based on read-write ratio, tolerance for staleness, durability needs and failure behaviour, and add TTLs as a safety net, plus protection against stampedes.

Likely follow-ups

- Why is write-back risky?
- What does cache-aside do on a write?

 Cache strategies

Cache-aside	Write-through
App reads cache, then DB	Write cache and DB together
Simple, stale risk	Fresh, slower writes

Cache-aside is the flexible default; write-through gives consistency at write cost; write-back is fast but risks data loss.

```
cache-aside: app -> cache miss -> DB -> fill cache
write-through: app -> cache -> DB (sync)
write-back: app -> cache -> async flush to DB
```

Principal answer: Decide how stale reads can be.

 Official docs

- [AWS Builders' Library: Caching challenges and strategies](#)
- [Microsoft: Cache-Aside pattern](#)

102. **How do you decide between vertical and horizontal scaling and when does each break down?**

tradeoffs

 Spoken answer · ~61 s

Vertical scaling means using a larger machine, which keeps the architecture simple and avoids distributed system problems; it works well for databases up to a surprisingly large size. Its limits are hardware ceilings, rising cost per unit of capacity, longer maintenance and recovery times, and a single point of failure. Horizontal scaling adds more machines behind a load balancer, which scales further and improves availability, but requires that state is externalised or partitioned, and introduces consistency, coordination and operational complexity. Stateless application tiers scale horizontally with ease. Stateful systems need replication for reads and sharding for writes, which is much harder. In practice I start with the simplest approach that meets the requirement, scale vertically while it is economical, add read replicas and caching, and design the data model with sharding keys in mind so that the move to horizontal scaling later is not a rewrite.

Likely follow-ups

- What makes a tier easy to scale horizontally?
- When is vertical scaling the right answer?

Vertical is simple but capped and a single point of failure; horizontal needs statelessness or partitioning but scales further.

Vertical: bigger box (easy, limit, SPOF)
Horizontal: more boxes + LB + shared/partitioned state

Principal answer: Scale the simple thing first, but plan the exit.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

103. **Design a file storage and sync service like Dropbox. How do you handle large files and sync efficiently?**

scalability

 Spoken answer · ~61 s

The client splits each file into fixed or content-defined chunks, around four megabytes, and computes a hash for each. It asks the server which chunks it already has, and uploads only missing ones, which saves bandwidth for edits and enables deduplication across files and users. Chunks are stored in object storage; a metadata service records the file tree, versions and the ordered list of chunk hashes in a database. When a change is committed, a notification service, using long polling or push, tells the user's other devices, which fetch the new metadata and download only the changed chunks. Conflicts are handled by keeping both versions with conflict markers. Resumable uploads handle poor networks. Security adds encryption in transit and at rest, and access control by sharing metadata. Scaling uses sharding of metadata by user, and a CDN or regional storage for downloads. Garbage collection removes unreferenced chunks.

Likely follow-ups

- Why chunk and hash files?
- How do you resolve edit conflicts?

Split files into content-addressed chunks, upload only changed chunks, keep metadata in a database and sync via notifications.

```
Client -> chunker (4MB, hash) -> upload new chunks -> Blob store
Metadata DB: file -> [chunk hashes], version
Notify service -> other devices -> pull diff
```

Principal answer: Deduplicate by content hash.

 Official docs

- [Git LFS](#)

104. **Design a hotel or ticket booking system that prevents double booking under high concurrency.**

scalability

 Spoken answer · ~62 s

I separate browsing from booking. Search and availability views read from caches and replicas and can be slightly stale. The booking path must be strictly correct. When a user selects a room or seat, the system creates a temporary hold using an atomic conditional update, for example decrementing availability only if it is greater than zero, or inserting a unique reservation row, inside a database transaction. Pessimistic locking works for low contention, optimistic concurrency with version checks scales better. Holds expire after a few minutes through a TTL and a background job releases them. Payment is handled with idempotency keys so retries do not charge twice, and the booking is confirmed only after payment succeeds, with a saga to release inventory on failure. For flash sales, I use a queue to serialise and a waiting room to protect the database. Sharding by event or hotel isolates hot items.

Likely follow-ups

- How do holds prevent double booking?
- How do you protect the DB in a flash sale?

Use transactional inventory with row locking or optimistic concurrency, short-lived holds with expiry, and idempotent payments.

Search (read replica/cache) -> Hold (TTL 10m) -> Pay -> Confirm
Inventory DB: UPDATE ... WHERE available>0 (atomic)
Expiry job releases holds

Principal answer: Correctness on the write path, speed on the read path.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

105. **Design a maps and routing service. How do you compute fastest routes at scale?**

scalability

 Spoken answer · ~60 s

The road network is a weighted graph where edge weights represent travel time. Plain Dijkstra over a continent-sized graph is too slow, so I preprocess the graph offline into a hierarchy, for example contraction hierarchies, which add shortcut edges so queries explore only a tiny fraction of nodes and run in milliseconds. The graph is partitioned geographically so each routing server holds part of it. Map display is separate: pre-rendered or vector tiles stored in object storage and served through a CDN at multiple zoom levels. Live traffic comes from anonymised location probes from phones, processed by a stream pipeline that updates edge weights; routing blends historical patterns and current speeds, and predicted traffic for the arrival time. Results are cached for popular origin and destination pairs, and the system supports rerouting via periodic updates. Geospatial indexes such as geohash or S2 support nearby-place queries.

Likely follow-ups

- Why preprocess the graph?
- How is live traffic incorporated?

Model roads as a graph, use hierarchical preprocessing like contraction hierarchies, tile the map, and blend live traffic.

Map tiles (CDN) <- tile service
Routing: graph partitions -> contraction hierarchies -> A*
Traffic: GPS probes -> stream -> edge weights

Principal answer: Preprocess offline, query online.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

106. Design an email service that sends and receives mail for millions of users.

scalability

Spoken answer · ~59 s

Inbound mail arrives at SMTP gateways found through MX records, which validate SPF, DKIM and DMARC, run spam and virus filters, and place accepted messages on a queue. Workers write them to a mailbox store, typically a wide-column or object-based system keyed by user, plus a search index for full-text queries, and update counters and notifications. Outbound sending goes through a queue so spikes do not overload senders; sender pools handle retries with exponential backoff for temporary failures, per-destination rate limits to respect receiving servers, and bounce processing. Deliverability depends on sender reputation, so I manage dedicated IP pools, warm up new IPs gradually, sign messages and handle complaints through feedback loops and suppression lists. Storage is replicated across zones, with quotas per user. Scaling comes from sharding mailboxes by user, and separating hot recent mail from older archives on cheaper storage.

Likely follow-ups

- What do SPF, DKIM and DMARC do?
- Why warm up new sending IPs?

Separate SMTP ingestion, queued delivery with retries, spam filtering, mailbox storage and search, with attention to reputation.

Inbound: MX -> SMTP gw -> spam/AV -> queue -> mailbox store + index
Outbound: API -> queue -> sender pool (IP reputation) -> remote MX
Bounces/complaints -> feedback loop

Principal answer: Deliverability is the hard part.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

107. Design a distributed file system like GFS or HDFS.

scalability

 Spoken answer · ~61 s

Files are split into large blocks, such as 128 megabytes, stored on many data nodes with three replicas placed across racks. A master node, called the NameNode in HDFS, stores the namespace and the mapping from files to blocks in memory for speed, persisting changes to a log with checkpoints, and a standby for high availability. Clients ask the master where blocks live and then read or write directly with data nodes, so the master is not a data bottleneck. Writes flow through a pipeline of replicas. Data nodes send heartbeats and block reports; when one dies, the master schedules re-replication of its blocks. Large blocks reduce metadata and favour sequential throughput, making it ideal for batch analytics but poor for small files and low-latency random access. Erasure coding can replace replication for cold data to save space. Consistency is simple because files are mostly append-only.

Likely follow-ups

- Why large block sizes?
- What is the NameNode's limitation?

A metadata master tracks file-to-block mappings while chunk servers store replicated blocks; clients stream data directly from chunk servers.

```
Client -> NameNode/Master (metadata, leases)
      -> DataNodes: block A (3 replicas), block B ...
Heartbeats + block reports; re-replicate on failure
```

Principal answer: Keep the master out of the data path.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

108. Design a Kafka-like distributed log or message broker.

scalability

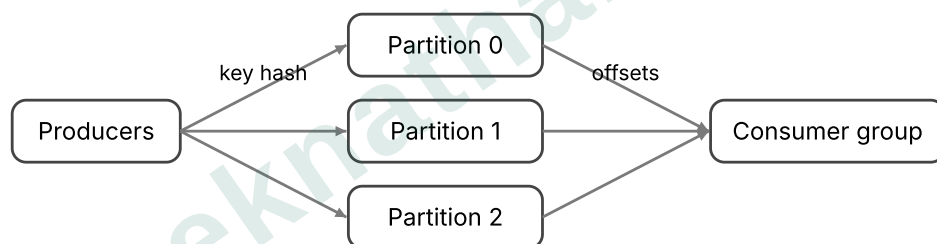
🗣 Spoken answer · ~59 s

A topic is divided into partitions, each an ordered, append-only log stored as segment files on disk. Producers choose the partition, often by hashing a key so related events stay ordered. Each partition has a leader that serves reads and writes and followers that replicate; the set of in-sync replicas determines when a write is acknowledged, trading durability for latency through the acknowledgement setting. Throughput comes from sequential disk writes, batching, compression, zero-copy transfer and the operating system page cache. Consumers in a group divide partitions among themselves and track their position as an offset, which allows replay, independent consumers and at-least-once processing. Retention is by time or size, not by consumption. A controller or consensus layer manages metadata and leader election. Scaling means adding partitions and brokers, but ordering is only guaranteed within a partition, and hot keys can skew load.

Likely follow-ups

- Why is Kafka fast on spinning disks?
- What guarantees ordering?

🗣 Kafka-like log



Partitioned, replicated append-only logs with consumer offsets give high throughput, ordering per partition and replayability.

Topic -> partitions (leader + followers, ISR)
Producer -> partition by key -> append to log (segment files)
Consumer group: each partition -> one consumer, tracks offset

Principal answer: Sequential disk I/O is the secret.

📖 Official docs

- [Apache Kafka documentation](#)
- [RabbitMQ documentation](#)

109. **Design an ad click aggregation system that counts billions of clicks accurately.**

scalability

 Spoken answer · ~58 s

Clicks arrive at stateless collectors that validate and write them to a durable log such as Kafka, partitioned by ad identifier to localise aggregation. A stream processor like Flink reads the log, deduplicates using event identifiers within a time window, and computes counts in tumbling or sliding windows, handling late events with watermarks and allowed lateness. Results are written to a fast analytics store for dashboards and billing queries, in pre-aggregated rollups at minute, hour and day granularity. The raw events also land in object storage so a periodic batch job can recompute exact numbers and reconcile any differences caused by late data or failures, which matters for billing. Hot ads cause skewed partitions, mitigated by sub-partitioning and two-stage aggregation. Fraud detection runs on the same stream. Correctness uses checkpointing and idempotent or transactional writes to approximate exactly-once results.

Likely follow-ups

- How do you handle late events?
- Why reconcile with batch?

Ingest clicks through a durable log, aggregate in tumbling windows with stream processing, deduplicate, and reconcile with batch.

Click -> collector -> Kafka -> Flink (dedupe, window 1m)
-> OLAP store (rollups) + raw to data lake -> batch reconcile

Principal answer: Exactly-once effects via idempotent sinks.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

110. Design a trending topics or top-K system for a social platform.

scalability

 Spoken answer · ~60 s

Hashtags or terms are extracted from the event stream and counted within sliding windows. Exact counting of billions of distinct terms is expensive, so each processing shard keeps an approximate frequency structure such as a count-min sketch plus a small heap of the current top candidates. Shards periodically send their local top lists to an aggregator that merges them into a global ranking. To capture recency, I use time-bucketed counters and weight newer buckets more heavily or apply exponential decay, so a burst of activity rises quickly and then fades. Trends are also normalised against baseline popularity so that always-popular terms do not dominate, and spam and bot activity are filtered. Results are refreshed every minute or so and stored in a cache for the API, with personalisation or regional variants computed separately. Accuracy needs are modest, so memory and latency are prioritised.

Likely follow-ups

- Why use a count-min sketch?
- How do you detect a sudden spike?

Count events in sliding windows with approximate structures, merge partial results, and decay old counts.

```
Posts -> stream -> per-shard Count-Min Sketch + heap (window 1h)
      -> merge top-K -> cache -> API
Decay older windows
```

Principal answer: Approximate is fine for trends.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

111. **Design a flash sale or high-demand ticket release system that survives a traffic spike.**

scalability

 Spoken answer · ~60 s

The goal is to protect the system and give fairness. Static assets and the product page are served from a CDN so only essential requests reach the backend. A virtual waiting room admits users at a controlled rate, issuing signed tokens, which smooths the spike. For inventory, I use an atomic counter in a fast store like Redis to reserve stock, rejecting requests immediately once it reaches zero, which prevents overselling and keeps the database out of the hot path. Accepted reservations go on a queue and are turned into orders asynchronously, with payment and confirmation following, and unpaid reservations expire and return stock. Rate limiting, bot detection and per-user limits prevent abuse. Capacity is pre-scaled before the event, and graceful degradation disables non-essential features. Load testing at expected peak plus margin and a rehearsed runbook for rollback are part of the plan.

Likely follow-ups

- How does the waiting room help?
- How do you avoid overselling?

Use a waiting room queue, cached static content, atomic inventory with strict limits, and asynchronous order processing.

CDN (static) -> waiting room (token) -> API
-> Redis atomic stock decr -> order queue -> DB
Rate limit + bot protection

Principal answer: Protect the inventory, shed the rest.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

 Spoken answer · ~60 s

The streamer sends video to an ingest server over RTMP or SRT. Live transcoders produce several resolutions and bitrates and a packager cuts them into short segments with manifests for HLS or DASH, written to an origin. A multi-tier CDN caches segments close to viewers, so the origin serves each segment only a few times regardless of audience size. Adaptive bitrate lets players switch quality as bandwidth changes. Typical latency is several seconds; lower latency variants such as LL-HLS or WebRTC cut it further at higher cost and complexity. Chat is a separate system: clients connect over WebSockets to gateways, messages are published to a pub/sub backbone partitioned by channel, and gateways fan out to subscribers, with rate limiting, moderation and sampling for very large rooms. Metadata, authentication and recordings are handled by separate services, and recordings are stored for video on demand.

Likely follow-ups

- Why do segments scale so well on a CDN?
- How do you reduce live latency?

Ingest via RTMP/SRT, transcode to adaptive bitrate segments, distribute over a CDN, and scale chat with a separate pub/sub fan-out.

```
Streamer -> ingest -> transcoder (ABR ladder) -> packager (HLS/DASH)
        -> origin -> CDN -> viewers
Chat: WebSocket gw -> pub/sub -> fan-out
```

Principal answer: Latency vs scale: pick the protocol deliberately.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

113. **Design a distributed build system or remote build cache for a large monorepo.**

scalability

 Spoken answer · ~58 s

A build is a graph of actions, each with declared inputs, a command and outputs. For caching to be correct, builds must be hermetic: the output depends only on declared inputs and a pinned toolchain. The action key is a hash of the inputs, command and environment; if a result for that key exists in the action cache, its outputs are fetched from a content-addressable store instead of being rebuilt. The scheduler distributes cache-missing actions across a pool of remote workers, running in sandboxes, with data locality to reduce transfers. Developers and CI share the cache, so a change rebuilds only affected targets. Challenges include non-determinism, large artifact transfer, cache poisoning protection through write restrictions, and eviction policies. Systems like Bazel with remote execution work this way, delivering large speedups at the cost of migrating to hermetic build definitions.

Likely follow-ups

- Why must builds be hermetic?
- How do you prevent cache poisoning?

Hash inputs to cache action results, distribute actions to a worker pool, and share artifacts through a content-addressable store.

Client -> scheduler -> workers (hermetic sandbox)
Action key = hash(inputs + command + toolchain)
CAS (artifacts) + action cache

Principal answer: Hermetic builds make caching safe.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

114. Design an alerting and monitoring system that evaluates millions of rules.

scalability

 Spoken answer · ~61 s

Rules are distributed across a fleet of evaluators using consistent hashing, each running its assigned rules at their intervals against the time-series database. Query cost is critical, so recording rules precompute expensive expressions, and queries are batched and limited. Each rule has a state machine: inactive, pending for the required duration, and firing, with state persisted so evaluator restarts do not cause missed or duplicate alerts, and with replicated evaluators for high availability and deduplication downstream. Firing alerts go to a notification service that groups related alerts, applies inhibition and silences, routes by labels to the right team and channel, and retries delivery. The system must be more reliable than the systems it watches, so it is deployed independently, across zones, with a watchdog alert that proves the pipeline works end to end. Scaling concerns include rule explosion and cardinality, which I control with limits and ownership.

Likely follow-ups

- Why persist alert state?
- What is a watchdog alert?

Shard rule evaluation, query a time-series store efficiently, deduplicate and route alerts, and monitor the monitor.

Rules -> sharded evaluators -> TSDB queries
-> alert state machine (pending/firing)
-> grouping/dedup -> routing -> pager/chat
Watchdog heartbeat

Principal answer: Alerting must be more reliable than what it watches.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

115. Design a webhook delivery system for third-party integrations.

scalability

 Spoken answer · ~61 s

When an event occurs, the system finds matching subscriptions and enqueues a delivery task per endpoint. Workers send an HTTPS POST with a signature header computed with a shared secret and a timestamp, so receivers can verify authenticity and reject replays, along with a unique event identifier for idempotency. Short timeouts prevent hanging. Failures are retried with exponential backoff and jitter over hours or days, and after the limit, events go to a dead letter queue and the endpoint may be disabled with notification to the owner. Because slow or failing customer endpoints could block the system, I isolate queues per endpoint or tenant, apply per-endpoint concurrency limits and circuit breakers, so one bad receiver cannot affect others. Delivery is at least once and ordering is not guaranteed unless documented. A dashboard shows delivery attempts, response codes and allows manual replay, which greatly reduces support burden.

Likely follow-ups

- How do receivers verify authenticity?
- How do you stop one bad endpoint affecting others?

Queue events per subscription, deliver with signed requests, retry with exponential backoff, isolate slow endpoints and give visibility.

Event -> subscription match -> per-endpoint queue
-> delivery workers (timeout 10s, HMAC sig)
retry backoff -> DLQ; circuit breaker per endpoint; delivery log UI

Principal answer: Customers' endpoints are unreliable by default.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

116. **Design a pub/sub notification fan-out for a celebrity account with 100 million followers.**

scalability

🗣 Spoken answer · ~56 s

A naive fan-out on write would create a hundred million writes for one post, causing delays and load spikes. So I treat celebrity accounts differently. For ordinary users, posts are pushed to followers' timelines at write time. For very high-follower accounts, the system does not write to every follower; instead their posts are stored once, and at read time the timeline service merges the precomputed feed with recent posts from followed celebrities. For notifications, I prioritise delivery to recently active users first, using batches processed by many workers and rate limiting against push providers, and deliver to the long tail progressively or on app open. Partitioned queues prevent one large fan-out from starving others, and idempotent tasks make retries safe. Monitoring tracks fan-out lag. This hybrid keeps write cost bounded while keeping reads fast.

Likely follow-ups

- Why not fan out on write for celebrities?
- How do you prioritise recipients?

Use hybrid fan-out: push to active followers' feeds in batches, pull for the rest, and prioritise by activity.

Post -> fanout service -> batch workers -> active followers' inboxes
Inactive followers: pull on open
Celebrity: no per-follower write; merge at read

Principal answer: Avoid write amplification for hot publishers.

📖 **Official docs**

- [Pub/Sub](#)
- [Go concurrency patterns: Pipelines and cancellation](#)
- [Microsoft: Competing Consumers pattern](#)

117. How do you design retries, timeouts and budgets so retries don't cause retry storms?

reliability

🗣 Spoken answer · ~62 s

Retries help with transient errors but can turn a small incident into an outage: when a dependency slows down, every caller retries, multiplying load, and layered services multiply it again. I set explicit timeouts at every call, nested so that inner timeouts are shorter than outer ones. Retries apply only to idempotent operations and only for retryable errors such as timeouts and service unavailable, with exponential backoff and random jitter so clients do not synchronise. I limit attempts to a small number and use a retry budget, for example allowing retries up to ten percent of the request volume, so retry traffic is bounded. Circuit breakers stop calling a failing dependency, and load shedding at the server protects it. Idempotency keys make retried writes safe. Hedged requests can reduce tail latency for reads but add load, so I use them sparingly. Metrics on retry rates reveal trouble early.

Likely follow-ups

- What is a retry budget?
- Why add jitter?

🔧 Safe retries



Set timeouts shorter than the caller's, retry only idempotent calls with backoff and jitter, and cap retries with budgets and circuit breakers.

Client timeout 1s < server SLA
Retry: max 2, backoff+jitter, only on 5xx/timeout
Retry budget: <=10% of requests
Circuit breaker opens on sustained failure

Principal answer: Retries multiply load by the retry count at every layer.

📖 Official docs

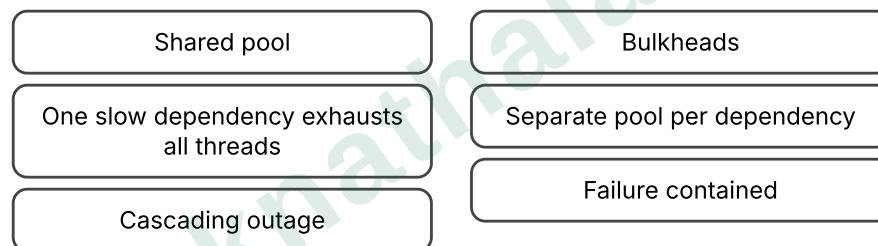
- [Google SRE Book: Addressing Cascading Failures](#)
- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)
- [Microsoft: Cloud design pattern – Retry](#)

 Spoken answer · ~57 s

The name comes from ship bulkheads that stop a leak flooding the whole hull. In software, a bulkhead separates resources so that one overloaded or failing component cannot consume everything. Examples are separate thread pools or connection pools per downstream dependency, so a slow reporting database cannot take all threads needed for checkout; separate queues or consumer groups per priority or tenant; dedicated instances or clusters for critical customers; and cell-based architectures that split users into independent cells, limiting the blast radius of a bad deployment to a fraction of traffic. Bulkheads work together with timeouts and circuit breakers. The cost is lower utilisation efficiency because capacity is partitioned, and more configuration to maintain, so I apply them where failure isolation matters most, identified by thinking about which shared resource would be exhausted in a failure.

Likely follow-ups

- What is a cell-based architecture?
- What is the cost of bulkheads?

 Bulkhead

Partition resources such as thread pools, connection pools or instances so one failing workload cannot exhaust shared capacity.

```
Service -> pool A (checkout, 20 threads)
        -> pool B (search, 20 threads)
        -> pool C (reports, 5 threads)
```

Principal answer: Name the shared resource that would be exhausted.

 Official docs

- [AWS: Disaster recovery & availability concepts \(Fault isolation boundaries\)](#)
- [Google SRE Book: Addressing Cascading Failures](#)
- [Microsoft: Cloud design pattern – Bulkhead](#)

119. **Design graceful degradation: what do you turn off first when the system is overloaded?**

reliability

 Spoken answer · ~58 s

Before an incident I classify features into tiers by business criticality. The core path, such as authentication and checkout, gets the strongest protection, with reserved capacity and priority in queues. Optional features, like personalised recommendations, recently viewed lists or analytics, are tiered lower and designed with fallbacks: serve cached or static content, return defaults, or skip them entirely when a dependency is slow. Load shedding at the edge rejects the lowest-priority requests first with clear responses and retry hints, using request classification, per-client limits and adaptive concurrency limits. Feature flags let operators switch off expensive features quickly, and some can be automated based on latency or error signals. Pre-computed fallbacks and stale-while-error caching keep pages useful. I rehearse degradation in game days, because untested fallbacks often fail, and I communicate degraded mode to users honestly in the interface.

Likely follow-ups

- What do you protect first?
- Why test fallbacks?

Rank features by criticality, shed optional load first, serve stale or cached data, and keep the core path alive.

Tier 0: login, checkout, payments (protect)
Tier 1: search, browse
Tier 2: recommendations, analytics (shed first)
Fallbacks: cache, defaults, static

Principal answer: Decide degradation tiers before the incident.

 Official docs

- [Google SRE Book: Handling Overload](#)
- [AWS Builders' Library: Using load shedding to avoid overload](#)
- [AWS Builders' Library: Avoiding fallback in distributed systems](#)

120. Explain quorum reads and writes and how you tune N, R and W.

reliability

 Spoken answer · ~63 s

In a replicated store each item lives on N nodes. A write succeeds when W replicas acknowledge, and a read consults R replicas and returns the most recent version. If R plus W is greater than N, the read set and the latest write set must overlap in at least one node, so the read sees the latest committed value. A common configuration is N three, W two, R two, tolerating one node failure for both reads and writes. Lower W makes writes faster and more available but risks losing data if the few acknowledging nodes fail; lower R speeds reads but allows stale results. Setting W to N favours read speed but writes fail if any replica is down. Systems like Cassandra and Dynamo expose these settings per request. Quorums still do not give full linearizability without extra mechanisms, because of concurrent writes and sloppy quorums, so conflict resolution matters.

Likely follow-ups

- What does $R + W > N$ guarantee?
- What happens with sloppy quorums?

With N replicas, require W acks on write and R on read; $R + W > N$ guarantees overlap with the latest write.

N=3, W=2, R=2 -> overlap, consistent reads

N=3, W=1, R=1 -> fast, eventual

N=3, W=3, R=1 -> slow writes, fast reads

Principal answer: Tune per use case, not globally.

Official docs

- [Google SRE Book: Managing Critical State \(consensus\)](#)
- [Raft consensus algorithm](#)

121. **How does the Raft consensus algorithm work at a high level, and when do you need it?**

reliability

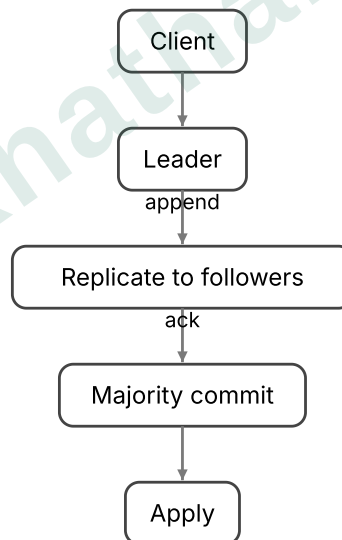
🗣 Spoken answer · ~62 s

Raft makes a cluster of nodes agree on an ordered log of commands, so a replicated state machine behaves like a single reliable server. Time is divided into terms. A node that hears no heartbeat from a leader becomes a candidate and requests votes; the candidate with a majority becomes the leader. The leader accepts client commands, appends them to its log and replicates to followers; once a majority has stored an entry it is committed and applied. A rule that only candidates with up-to-date logs can win elections preserves committed entries across leader changes. The cluster tolerates the failure of a minority, so three nodes survive one failure and five survive two, but writes stall without a majority. I need consensus for metadata, locks, leader election and configuration stores, as in etcd and Consul, not for bulk data. I rely on proven implementations and keep clusters small.

Likely follow-ups

- Why odd cluster sizes?
- What happens when a leader fails?

📄 Raft write



A leader is elected by majority vote and replicates a log to followers; entries commit when a majority has them.

Follower -> timeout -> Candidate -> majority votes -> Leader
Leader: append entry -> replicate -> commit on majority
3 nodes tolerate 1 failure; 5 tolerate 2

Principal answer: Use a library, never write your own.

Official docs

- [Google SRE Book: Managing Critical State \(consensus\)](#)
- [Raft consensus algorithm](#)

122. Design zero-downtime deployments for a stateful service with long-lived connections.

reliability

Spoken answer · ~62 s

For stateful services like WebSocket gateways or databases, restarting nodes abruptly breaks sessions. The process starts by marking a node as draining in the load balancer so it receives no new connections. Existing connections are allowed to finish within a window, or clients are sent a reconnect instruction with jitter to avoid a thundering herd, and they resume from the last acknowledged position using session tokens. State held in memory is either externalised to a shared store or replicated, or handed off to another node before shutdown. Deployments proceed in small batches, with health and error-rate checks gating each step, so a bad version stops early. During the rollout, old and new versions run together, so protocols and schemas must be backward and forward compatible, using expand and contract for database changes. Capacity includes surge nodes so draining does not reduce headroom, and rollback uses the same drain procedure.

Likely follow-ups

- How do you avoid a reconnect storm?
- Why must versions interoperate?

Drain connections gradually, version the protocol, hand off state, deploy in small batches and keep rollback ready.

```
LB marks node draining -> no new conns
Existing conns finish / migrate (reconnect hints)
Rolling batch -> health gate -> next
Backward-compatible protocol
```

Principal answer: Both versions must interoperate during rollout.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

123. **How do you design for data durability: replication, backups and protection against corruption and deletion?**

reliability

🗣 Spoken answer · ~57 s

Durability is layered. Replication across disks, zones and regions protects against hardware and site failure, but it faithfully replicates accidental deletes and corruption, so it is not a backup. Backups with point-in-time recovery let me restore to just before a mistake. They must be stored separately from production, in a different account with restricted permissions, and made immutable or versioned with retention locks to resist ransomware and human error. Data integrity needs checksums on write and read, and background scrubbing that detects silent corruption and repairs from replicas. For critical systems, soft deletes and delayed physical deletion give a recovery window. I define recovery point and recovery time objectives, and prove them by running restore drills regularly, because an untested backup is unreliable. Monitoring covers backup success, age of the latest restorable point and replication health.

Likely follow-ups

- Why is replication not a backup?
- How do you protect backups from ransomware?

Replicate across failure domains, take immutable versioned backups, verify checksums, and test restores; replication alone doesn't protect against deletion.

Primary + sync replica (AZ-b) + async replica (region-2)
PITR backups (immutable, cross-account)
Checksums + scrubbing + restore drills

Principal answer: Replication copies mistakes instantly.

📖 **Official docs**

- [PostgreSQL: High availability, load balancing, replication](#)

124. Design a ledger or wallet system where balances must never be wrong.

data

 Spoken answer · ~61 s

Money systems use double-entry bookkeeping: every transfer creates immutable ledger entries that debit one account and credit another so that the sum is always zero. Balances are derived from entries, often cached in a materialised column updated in the same transaction, and entries are never edited, only reversed by new entries. The transfer runs in a single ACID transaction on one database shard, or through a saga with strong guarantees if accounts are in different shards, with idempotency keys to make retries safe and unique constraints preventing duplicates. Concurrency is controlled with row locks or optimistic versions, and overdraft rules are enforced inside the transaction. Every state change is auditable. Continuous reconciliation compares internal totals with external sources such as banks and payment providers, and alerts on any mismatch. Strong consistency is chosen over availability for the write path, with careful operational procedures and access controls.

Likely follow-ups

- Why derive balances from entries?
- How do you make transfers idempotent?

Use an append-only double-entry ledger, derive balances, enforce transactional invariants and idempotency, and reconcile continuously.

```
Transfer(idempotency_key) -> tx:
  debit A entry + credit B entry (sum=0)
Balances = materialised from entries
Daily reconcile vs bank/provider
```

Principal answer: Never update a balance in place without an entry.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

125. **Design an inventory management system for an e-commerce platform with multiple warehouses.**

data

 Spoken answer · ~61 s

Each SKU at each warehouse has on-hand and reserved quantities. At checkout, a reservation is made atomically, for example a conditional update that decrements available stock only if enough remains, or a row in a reservations table with a unique constraint, with an expiry for unpaid carts. When the order is paid, the reservation converts to an allocation and then to a pick and shipment; cancellations or expiry release stock. Warehouse events such as receipts, picks, damages and returns update on-hand counts, generally through events from the warehouse management system, and periodic cycle counts reconcile differences. The product page shows availability from a cached, slightly stale view, with safety stock buffers to reduce oversell risk, while the reservation path remains strongly consistent. Allocation logic selects the best warehouse by distance and stock. Hot SKUs during sales may need dedicated counters or sharding to avoid lock contention.

Likely follow-ups

- How do you avoid overselling?
- Why show availability from a cache?

Track available and reserved stock per SKU and location, reserve atomically at checkout, and sync with warehouses eventually.

```
Order -> reserve(SKU, qty, warehouse) atomic
Stock = on_hand - reserved
Events: received/picked/returned -> update
Reconcile with WMS counts
```

Principal answer: Separate display availability from the reservation.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

 Spoken answer · ~63 s

The decision runs inline with the payment, so the hot path must be fast and reliable. When a transaction arrives, the service fetches precomputed features from a low-latency feature store: counts of recent transactions per card, device and IP, typical amounts, account age and risk signals. These features are updated in near real time by a stream processor. A rules engine applies hard policies, such as blocklists and velocity limits, and a machine learning model, usually gradient boosted trees for speed, produces a risk score. The combination yields allow, deny or send to review or step-up authentication. The system must have a safe fallback if the model service is slow, such as conservative rules. Labels arrive late through chargebacks and manual reviews, so retraining uses delayed feedback and monitors drift. Every decision is logged with features for audit and debugging, and thresholds are tuned against the cost of false positives.

Likely follow-ups

- Why precompute features?
- How do delayed labels affect training?

Combine fast rules and precomputed features with a model, score synchronously, and learn from delayed labels.

```
Payment -> feature store (velocity, device, history)
        -> rules + ML score (<50ms) -> allow/review/deny
Events -> stream -> retrain; chargebacks = labels
```

Principal answer: Precompute features; keep the hot path simple.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

127. Design a data lake and analytics platform for a company with many data sources.

data

 Spoken answer · ~58 s

Data from databases, applications, events and third parties is ingested through change data capture, batch loads and streams into a raw zone in object storage, unchanged and immutable. Transformation jobs clean, validate and conform the data into curated zones using columnar formats like Parquet with a table format such as Iceberg or Delta, which provide schema evolution, transactions and time travel. A catalog registers datasets with schemas, owners and descriptions; lineage tracks how data flows; data quality checks catch problems early. Query engines such as Trino or Spark and a warehouse serve analysts, while orchestration tools schedule pipelines. Governance includes fine-grained access control, encryption, masking of personal data and retention policies. Without catalog, ownership and quality, a lake becomes a swamp. I partition data by date and common filters, compact small files, and manage cost with lifecycle tiers.

Likely follow-ups

- What do table formats add to a data lake?
- How do you stop a lake becoming a swamp?

Ingest raw data into object storage, organise zones, use a catalog and table formats, and serve through query engines with governance.

Sources -> CDC/batch/stream ingest -> raw zone (S3)
-> clean/curated (Parquet/Iceberg) -> warehouse/BI
Catalog + lineage + access control

Principal answer: Governance is what makes a lake usable.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

128. Design a schema registry and handle schema evolution in event streams.

data

 Spoken answer · ~62 s

In an event-driven system, producers and consumers deploy independently, so changes to message structure can break them. A schema registry stores each schema version for a subject, such as a topic, and enforces a compatibility policy when a new version is registered. Backward compatibility means new consumers can read old data; forward means old consumers can read new data; full means both. Messages carry a small schema identifier plus a compact binary payload in Avro, Protobuf or similar, and consumers fetch and cache the schema by id. Safe evolutions include adding optional fields with defaults and removing optional fields; breaking changes, like changing a type or renaming a required field, are rejected or require a new topic with a migration plan. Registry checks in CI catch problems before deployment. Operationally, the registry must be highly available and cached by clients so it is not on the critical path.

Likely follow-ups

- What is backward compatibility?
- Why put the schema id in the message?

Store versioned schemas, enforce compatibility rules at publish time, and let producers and consumers evolve independently.

Producer -> register schema (v3) -> compat check (backward)
Message = schema id + payload (Avro/Protobuf)
Consumer fetches schema by id

Principal answer: Add optional fields; never reuse field numbers.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

129. Design a sharded key-value store like DynamoDB with consistent hashing and replication.

data

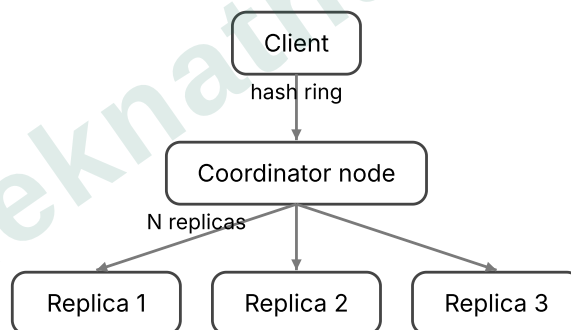
🗣 Spoken answer · ~61 s

Keys are hashed onto a ring, and each node owns several virtual positions, which balances load and lets capacity be added smoothly with minimal data movement. A request goes to a coordinator, which writes to the N nodes following the key's position, acknowledging after W succeed, and reads from R nodes, so tuning W and R trades consistency against latency and availability. When a replica is down, hinted handoff stores writes temporarily on another node to deliver later, preserving availability. Divergent replicas are reconciled through read repair and background anti-entropy using Merkle trees to find differences efficiently. Concurrent updates are resolved by last-write-wins with timestamps or by vector clocks that expose conflicts for application-level resolution. Membership and failure detection use gossip. Storage on each node uses an LSM tree with compaction. Hot keys can be mitigated with caching and key splitting, and secondary indexes add complexity.

Likely follow-ups

- What are virtual nodes for?
- How are divergent replicas repaired?

🚩 Dynamo-style KV store



Hash keys onto a ring with virtual nodes, replicate to N successors, use quorums and anti-entropy for repair.

key -> hash -> ring -> coordinator + N-1 successors
vnodes for balance; hinted handoff when node down
Merkle trees for anti-entropy; vector clocks / LWW

Principal answer: Virtual nodes smooth the distribution.

📖 Official docs

- [PostgreSQL: High availability, load balancing, replication](#)
- [Microsoft: Sharding pattern](#)
- [Amazon DynamoDB Developer Guide](#)

130. **Design a metadata and search service for a large object store, such as listing and querying billions of objects.**

data

 Spoken answer · ~58 s

Object data and object metadata have different access patterns, so I separate them. Blobs go to a durable, replicated or erasure-coded store addressed by internal identifiers. Metadata, such as bucket, key, size, checksum, timestamps, tags and the blob location, lives in a sharded, strongly consistent database sorted by key within a bucket, so listing by prefix becomes an efficient range scan. Sharding by hash spreads load but makes listing harder, so range partitioning with automatic splitting of hot ranges is common, with care to avoid hotspots from sequential key names. Strong read-after-write consistency for puts and lists requires consensus-backed metadata. Richer queries by tags or attributes are served by asynchronously updated secondary indexes or a search engine, accepting some lag. Lifecycle rules, versioning and replication are driven by metadata scans and change streams. Garbage collection removes orphaned blobs.

Likely follow-ups

- Why separate metadata from data?
- What makes prefix listing efficient?

Keep object metadata in a sharded, indexed store separate from blob data, with prefix listing and asynchronous secondary indexes.

PUT -> blob store (data) + metadata shard (key, size, etag, tags)
LIST prefix -> range scan on sorted key
Search -> async index (tags, attributes)

Principal answer: Separate the data plane from the metadata plane.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

131. **How do you design data retention, deletion and privacy compliance such as the right to be forgotten?**

data

 Spoken answer · ~58 s

I start with a data inventory: which services store personal data, in what systems, and how it flows to derived stores such as caches, search indexes, analytics warehouses, logs and backups. Fields are classified and tagged, and new services must register their data. A deletion request enters a central orchestrator that fans out to every system with idempotent delete tasks, tracks completion and retries failures, and records an audit trail. Where immediate deletion is impractical, such as immutable backups, the design uses retention windows and ensures deleted records are not restored without re-applying deletions, or uses per-user encryption keys that can be destroyed, a technique called crypto-shredding. Analytics may use pseudonymised or aggregated data. Retention policies automatically expire data that is no longer needed, and logs avoid personal data. Regular tests verify that deletion truly works end to end.

Likely follow-ups

- What is crypto-shredding?
- How do you handle backups?

Map where personal data lives, tag it, centralise deletion workflows, propagate to derived stores and backups, and audit.

Request -> deletion orchestrator
-> primary DBs, caches, search, analytics, lake, backups (expire)
Data map + PII tags + audit log

Principal answer: You can't delete what you can't find.

 Official docs

- [Prometheus: Storage \(TSDB\)](#)

 Spoken answer · ~61 s

Images are stored as layers addressed by content digest in object storage, which gives natural deduplication, and manifests that reference them with tags kept in a database. The registry API implements the push and pull protocols, authenticates users and CI systems through OIDC, and authorises by repository and role. For performance and resilience, regional pull-through caches or replicated registries serve nodes near them, avoiding cross-region traffic and reducing startup time, and a peer-to-peer distribution layer helps for very large fleets. Security features include vulnerability scanning on push and continuously, signing and attestation storage, tag immutability for releases, and quotas. Garbage collection removes unreferenced layers carefully, using locks or read-only windows to avoid races with concurrent pushes. Retention policies control storage growth. The registry sits on the critical path for deployments and autoscaling, so availability and capacity planning matter, and mirroring public base images avoids rate limit failures.

Likely follow-ups

- Why use regional caches?
- How does garbage collection work safely?

Store content-addressed layers in object storage, keep manifests and metadata in a database, cache regionally, and add scanning, signing and access control.

```
push -> registry API -> layers (blob store, by digest)
manifests + tags (DB) ; auth (OIDC/RBAC)
pull-through cache per region ; scan + sign + GC
```

Principal answer: Immutable tags and digest pinning.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

133. Design a configuration management and dynamic configuration service.

platform

 Spoken answer · ~63 s

Configuration changes can cause outages as easily as code, so I treat them with the same discipline. Configs live in version control or a versioned store, with schema validation and automated tests in CI. A config service distributes them to clients, which fetch via polling or a streaming connection and cache locally, keeping the last-known-good version so a service outage does not break running applications. Changes roll out progressively by environment, region or percentage, with automatic rollback if health metrics degrade, and feature flags handle behavioural toggles. Sensitive values come from a secrets manager, not the config store. Every change is audited with author, reason and diff, and permissions restrict who can change production values. Clients validate configs on load and refuse invalid ones, and the system exposes which version each instance is running. Latency of propagation is defined and measured, and a kill switch ensures emergency changes can be applied quickly.

Likely follow-ups

- Why keep a last-known-good copy?
- How do you roll out a config change safely?

A versioned config store with validation, staged rollout, client caching with local fallback, and audit and rollback.

Config repo -> validate -> config service (versioned)
Clients: poll/stream -> local cache -> last-known-good
Rollout by % / env; instant rollback

Principal answer: A config change is a deployment.

Official docs

- [Kubernetes: Deployments](#)

134. Design a workflow orchestration engine like Airflow or Temporal.

platform

 Spoken answer · ~61 s

A workflow is a graph or a program of steps with dependencies, retries, timeouts and sometimes long waits. The engine persists workflow state in a durable store, either as task state rows as in Airflow, or as an event history from which state is rebuilt by replaying deterministic workflow code, as in Temporal. A scheduler or matching service determines which tasks are ready and places them on queues, from which workers poll and execute, reporting results. Timers and retry policies are stored durably so they survive restarts, and heartbeats detect worker failure and reschedule tasks. Because delivery is at least once, tasks must be idempotent. Scale comes from sharding workflow state, partitioning task queues and horizontally scaling workers. Features include versioning of workflow definitions for in-flight runs, visibility APIs, backfills, and rate limiting. Concerns are scheduler throughput, hot partitions and the history size of long-running workflows.

Likely follow-ups

- Why must tasks be idempotent?
- How does replay recover state?

Persist workflow state, schedule tasks to workers, handle retries and timeouts, and guarantee progress after crashes.

Definition -> scheduler -> task queues -> workers
State store (event history) ; timers ; retries/backoff
Resume from history after failure

Principal answer: Durable execution needs deterministic replay or idempotent tasks.

Official docs

- [AWS Builders' Library: Timeouts, retries and backoff with jitter](#)

135. Design an audit logging system that is tamper-evident and queryable.

platform

 Spoken answer · ~58 s

Services emit structured audit events for security-relevant actions with who, what, when, where and outcome, using a standard schema and a common library so coverage is consistent. Events are sent through a durable pipeline so none are lost, and written to append-only storage with write-once-read-many retention, such as object lock, in an account separate from the systems being audited. To make tampering evident, each batch includes a hash of the previous batch, forming a chain, or records are periodically signed and anchored. A search index, built asynchronously, supports investigations with filters by user, resource and time, while the immutable store remains the source of truth. Access to audit data is tightly controlled and itself audited. Retention follows legal requirements, and sensitive fields are masked. Monitoring detects gaps in the event stream, since missing logs are as suspicious as altered ones.

Likely follow-ups

- How do hash chains show tampering?
- Why store audit logs separately?

Emit structured events at the source, ship to append-only storage, chain hashes or use object lock, index for search, and restrict access.

```
Services -> audit events -> Kafka -> writer
-> WORM storage (object lock) + hash chain
-> index (search) ; strict RBAC ; retention
```

Principal answer: Audit logs must not be editable by the audited.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

136. **Design a multi-region traffic management system with failover and latency-based routing.**

platform

 Spoken answer · ~61 s

Global traffic management directs each user to the best region. Options include DNS-based routing with health checks, which returns different answers by geography or latency measurements, and anycast, where the same IP is advertised from many locations and the network routes to the nearest. Health checks from multiple vantage points remove failed regions from rotation, and weights allow gradual shifts for canaries, maintenance or capacity balancing. DNS caching means failover takes as long as the TTL, so I use short TTLs for critical records and, where faster failover is needed, a global load balancer or anycast front end. Each region must have enough spare capacity to absorb a failed neighbour's traffic, otherwise failover causes a cascading overload. Stateful data needs a separate strategy: active-passive with replication, or active-active with conflict handling. I test failover regularly with game days and use automation but keep human override for safety.

Likely follow-ups

- Why does DNS failover lag?
- What capacity does each region need?

Use health-checked global DNS or anycast, route by latency or geography with weights, and drain regions gracefully.

Users -> GeoDNS/Anycast -> region A | B | C
Health checks -> remove unhealthy region
Weights for canary; stateful data replicated separately

Principal answer: Failover capacity must exist before the failure.

 Official docs

- [Route 53: Routing policies](#)
- [Google Cloud: Multi-region deployment archetypes](#)

137. Design a cost-aware autoscaling and capacity management system.

platform

 Spoken answer · ~60 s

The system scales on signals tied to load, such as requests per second, queue depth or latency, instead of only CPU, with minimum and maximum bounds and cooldowns to prevent flapping. Predictable patterns, like daily peaks or planned events, use scheduled or predictive scaling to pre-warm capacity because new instances take time to start. At the node level, an autoscaler adds and removes machines, mixing on-demand capacity for baseline and critical workloads with cheaper spot instances for fault-tolerant ones, diversified across types and zones to reduce interruption risk. To control cost, I right-size requests with usage data, bin-pack workloads, consolidate underused nodes and shut down non-production environments out of hours. Budgets, tags and showback reports give teams visibility, and alerts flag anomalies. Capacity management also includes reserved or committed use discounts for the stable baseline, and load tests to verify that scaling meets latency objectives.

Likely follow-ups

- Why scale on queue depth?
- How do you use spot instances safely?

Scale on demand signals with predictive and scheduled capacity, mix instance types, and use budgets and right-sizing feedback.

Metrics (RPS, queue) -> HPA/KEDA -> pods
Node autoscaler -> spot + on-demand mix
Forecast -> pre-scale ; budgets + right-size reports

Principal answer: Scaling fast is a cost lever, not just a reliability one.

Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

138. **Design a DNS service and explain how resolution, caching and TTLs shape failover.**

networking

📌 Spoken answer · ~61 s

DNS is a hierarchical distributed database. A client asks a recursive resolver, which, on a cache miss, walks from the root servers to the top-level domain servers to the authoritative servers for the zone, and caches the answer for its time to live. An authoritative service stores zones, replicated across multiple servers and typically served over anycast for low latency and resilience against attacks, with zone data distributed from a primary through transfers or an API. Caching at many layers makes DNS fast and scalable but means changes propagate gradually: a record with a one-hour TTL may be served stale for an hour. For planned migrations I lower the TTL in advance. For failover, short TTLs help but can increase query load, and some resolvers ignore low values. Security measures include DNSSEC, rate limiting and protection against amplification attacks. Observability tracks query rates, latency and error codes.

Likely follow-ups

- Why lower the TTL before a migration?
- What does anycast give authoritative DNS?

🚩 DNS resolution



Hierarchical authoritative servers with recursive resolvers caching answers; TTL balances load against change propagation speed.

```
Client -> stub -> recursive resolver (cache)
      -> root -> TLD -> authoritative (zone)
Answers cached for TTL; anycast authoritative servers
```

Principal answer: Lower TTL before a planned change.

📖 Official docs

- [Kubernetes: DNS for Services and Pods](#)
- [RFC 1035: DNS implementation and specification](#)
- [AWS Builders' Library: Caching challenges and strategies](#)

 Spoken answer · ~60 s

At the edge, load balancers terminate TLS with modern protocol versions and cipher suites, session resumption and OCSP stapling for performance. Certificates are issued and renewed automatically through ACME or an internal certificate authority, and distributed to edge nodes from a central store with private keys protected in a hardware security module or key management service, with strict access control. Short-lived certificates limit the damage of key compromise and force automation. For service-to-service traffic, mutual TLS with workload identities, such as SPIFFE, issued and rotated by a mesh or a certificate manager, provides authentication and encryption without developer effort. Operationally, I monitor expiry across every endpoint and store, alert early, test renewal paths, support rotation without downtime through hot reload, and keep an inventory of certificates and owners. For multi-tenant platforms with custom domains, automated issuance on demand with rate limiting is needed.

Likely follow-ups

- Why short-lived certificates?
- How do services authenticate each other?

Terminate at the edge with automated issuance and renewal, secure key storage, modern protocols, and monitoring of expiry.

```
Edge LB (TLS 1.3, OCSP stapling) <- cert store/HSM
ACME automation -> issue/renew -> distribute
mTLS internal via mesh/SPIFFE ; expiry monitoring
```

Principal answer: Automate renewal; short-lived certs reduce risk.

 Start here (official docs)

- [Azure Architecture Center: cloud design patterns](#)

140. Design an API rate limiting and quota system for a public API with tiers.

networking

 Spoken answer · ~61 s

Clients are identified by API key or OAuth token and mapped to a tier with defined limits. Short-term rate limits, for example requests per second with allowed bursts, use a token bucket or sliding window counter held in a shared low-latency store like Redis, updated atomically with scripts to avoid races. To reduce latency and load, gateways can keep local approximate counters and sync periodically, accepting slight inaccuracy. Longer-term quotas, like monthly request allowances, are metered separately through asynchronous aggregation, because they tie to billing and need durability rather than millisecond decisions. Responses use status 429 with Retry-After and rate limit headers so clients can back off. Different limits can apply per endpoint cost, with expensive operations weighted more. Abuse protection adds IP-based limits and anomaly detection. Failure behaviour must be decided: usually fail open if the limiter store is down, to avoid an outage, with alerts.

Likely follow-ups

- Why separate rate limits from quotas?
- Should the limiter fail open or closed?

Identify clients by key, enforce limits at the gateway using a token bucket in a shared fast store, return clear headers, and meter quotas separately.

```
Request -> gateway -> key lookup (tier)
        -> Redis token bucket (rate) ; counter (monthly quota)
429 + Retry-After + X-RateLimit-* headers
```

Principal answer: Separate burst control from billing quotas.

Official docs

- [RFC 6585: HTTP 429 Too Many Requests](#)
- [Microsoft: Cloud design pattern – Throttling](#)
- [Stripe Engineering: Scaling your API with rate limiters](#)

 Spoken answer · ~60 s

A traditional VPN gives broad network access once connected, so a compromised laptop can reach many systems. A zero-trust design treats the network as hostile and verifies each access. Users authenticate with the corporate identity provider using single sign-on and strong multi-factor authentication, and device posture is checked: managed, encrypted, patched, endpoint protection active. An identity-aware proxy or access broker sits in front of internal applications and evaluates policy per request, combining user, group, device state, location and risk, granting access only to specific applications or ports. Connections are encrypted and applications are not exposed to the internet directly. Access decisions and sessions are logged for audit, and sessions are re-evaluated continuously, so revoking a user or detecting a risky device cuts access promptly. For infrastructure access like SSH or databases, short-lived credentials and session recording are used. Migration is incremental, application by application.

Likely follow-ups

- What does device posture add?
- How does it differ from a VPN?

Authenticate user and device, authorise per application with least privilege through an identity-aware proxy, and log everything; avoid broad network access.

User + device posture -> IdP (MFA)
-> identity-aware proxy -> app (per-policy)
No flat network access ; continuous evaluation ; audit

Principal answer: Never trust the network; verify each request.

 Official docs

- [Cloudflare Learning: What is a VPN?](#)
- [NIST Zero Trust Architecture \(SP 800-207\)](#)