200+ Backend Interview Problems

Curated by

Mission_Compile (Bhavesh Vaswani)

Insta ID: mission_compile

📖 A brief about the document:

This is not just a list of questions—it is a practical guide to help you learn, prepare, and master backend engineering interview concepts with real-world scenarios.

- ✅ Contains **200+ backend interview problems** covering system design, databases, caching, message queues, APIs, microservices, security, networking, data processing, Kubernetes, cloud architecture, and more.
- ✅ Covers **50+ essential topics** including horizontal/vertical scaling, load balancing, CAP theorem, distributed systems, authentication, GraphQL, Redis persistence, SQL injection prevention, HTTP/2 vs HTTP/3, distributed locks, ETL pipelines, load testing, blue-green deployments, chaos engineering, and deployment strategies.
- ✅ Detailed **problem-solution format** with step-by-step explanations, real-world examples, and color-coded keywords for quick reference.
- ✅ Organized by topics with **clear visual separation**, making it easy to navigate and revise specific concepts.
- ✅ Includes practical scenarios like **when to use Redis vs Kafka**, how to implement circuit breakers, database sharding strategies, and API security best practices.

- ✅ Contains **trade-off discussions** to help you make informed architectural decisions during interviews.
- ✅ Real-world examples showing **how companies like Netflix, Amazon, and Uber** solve similar problems at scale.

Perfect for preparing backend engineering interviews, revising system design concepts, or learning distributed systems from scratch! 🚀

📋 Table of Contents

Total: 50+ Topics | 200+ Problems

Problems & Solutions

📚 Horizontal vs Vertical Scaling

💡 Problem 1: When to Scale Horizontally vs Vertically

Question: INTERVIEWER: Your application is running on a single server that's hitting CPU limits. Should you upgrade the server or add more servers?

Solution:

Scaling isn't just about adding resources — it's about choosing the right growth strategy. Here's when to scale horizontally vs vertically 👇

1️⃣ Vertical Scaling (Scale Up)

👉 Add more CPU, RAM, or storage to your existing server.

Example: Upgrade from 4-core to 16-core server when database queries need more processing power.

AWS instance resize | adding RAM | upgrading CPU

2️⃣ Horizontal Scaling (Scale Out)

👉 Add more servers to distribute load across multiple machines.

Example: 10 servers handling 10K req/s each instead of 1 server struggling with 100K req/s.

Auto-scaling groups | Kubernetes pods | load balancers

3️⃣ When to Choose Vertical Scaling

👉 Use when your application can't be distributed (legacy monoliths, single-threaded apps).

Example: Old ERP system that requires all data in one place with high RAM.

Monolithic databases | single-instance applications

4️⃣ When to Choose Horizontal Scaling

👉 Use for stateless applications that can run independently on multiple machines.

Example: REST API servers that don't share state — each handles requests independently.

Microservices | stateless APIs | containerized apps

5️⃣ Vertical Scaling Limits

👉 Physical hardware limits — can't infinitely upgrade a single machine.

Example: You hit cloud provider's largest instance type (96 cores, 384GB RAM) — now what?

Hardware constraints | cost inefficiency at scale

6️⃣ Horizontal Scaling Benefits

👉 No theoretical limit — keep adding servers as traffic grows.

Example: Black Friday? Spin up 100 more servers, then scale down after.

Cloud elasticity | high availability | fault tolerance

7️⃣ Database Considerations

👉 Databases often need vertical scaling first (faster disk I/O), then horizontal (sharding).

Example: Upgrade to SSD and more RAM before sharding your database.

Database-specific scaling patterns

8️⃣ Cost Trade-offs

👉 Vertical scaling is expensive at high end; horizontal scaling is more cost-effective long-term.

Example: 1 massive server costs more than 10 medium servers with same total capacity.

TCO analysis | cloud pricing models

💡 Problem 2: Scaling a Monolith to Handle 10x Traffic

> **Question:** INTERVIEWER: Your monolithic app handles 10K users smoothly but crashes with 100K users. How do you scale it?

> **Solution:**

Scaling monoliths requires both vertical and horizontal strategies working together.

1️⃣ Add Load Balancer

👉 Distribute traffic across multiple instances of your monolith.

Example: Like having multiple checkout counters instead of one — customers get served faster.

Nginx | HAProxy | AWS ALB

2️⃣ Stateless Application Design

👉 Remove server-side session state so any instance can handle any request.

Example: Like ATMs that work anywhere — your account isn't tied to one specific machine.

JWT tokens | Redis sessions | database-backed sessions

3️⃣ Vertical Scaling as Quick Fix

👉 Temporarily upgrade server resources while you work on horizontal scaling.

Example: Like upgrading from a small restaurant to a bigger one before opening franchises.

Instance type upgrade | more RAM/CPU

4️⃣ Database Connection Pooling

👉 Reuse database connections efficiently as you add more app servers.

Example: Like sharing company cars instead of everyone driving separately — fewer resources wasted.

PgBouncer | HikariCP | connection pool configuration

5️⃣ Cache Aggressively

👉 Reduce database load by caching frequent queries.

Example: Like keeping frequently asked phone numbers on speed dial.

Redis | Memcached | CDN for static assets

6️⃣ Offload Heavy Tasks

👉 Move background jobs to separate workers using message queues.

Example: Like having a kitchen prepare food while waiters serve — specialization improves speed.

Kafka | RabbitMQ | AWS SQS with worker processes

7️⃣ Auto-Scaling Groups

👉 Automatically add/remove servers based on traffic patterns.

Example: Like hiring temporary staff during holiday rush and letting them go after.

AWS Auto Scaling | Kubernetes HPA

💡 Problem 3: Database That Can't Scale Horizontally

Question: INTERVIEWER: Your legacy database doesn't support sharding. How do you scale it to handle 100x more traffic?

Solution:

When horizontal scaling isn't possible, you optimize what you have and scale reads separately.

1️⃣ Vertical Scaling

👉 Upgrade to the largest instance your database supports.

Example: Like moving from a sedan to a bus when you can't buy multiple cars.

2️⃣ Read Replicas

👉 Create read-only copies to offload SELECT queries from the primary.

Example: Like having multiple library copies so more people can read simultaneously.

PostgreSQL streaming replication | MySQL replication

3️⃣ Aggressive Caching

👉 Cache query results to avoid hitting the database entirely.

Example: Like keeping frequently used files on your desktop instead of searching every time.

Redis | application-level cache

4️⃣ Query Optimization

👉 Add indexes, optimize slow queries, remove N+1 problems.

Example: Like organizing your closet so you find clothes faster instead of buying more closets.

5️⃣ Archive Old Data

👉 Move historical data to separate storage to keep active database small.

Example: Like storing old emails in archive folders — keeps inbox fast.

Data archival | cold storage

6️⃣ Connection Pooling

👉 Maximize concurrent connections with efficient pooling.

Example: Like a taxi service reusing cars efficiently instead of buying one per customer.

PgBouncer | RDS Proxy

7️⃣ Partition Tables

👉 Split large tables into smaller partitions within same database.

Example: Like organizing files by year so you search smaller folders.

PostgreSQL partitioning | range partitioning

📚 Load Balancing Strategies

💡 Problem 1: API Crashes Under 1M Requests Per Second

Question: INTERVIEWER: How do you design an API that survives 1 million requests per second?

Solution:

1 Million RPS isn't about code, it's about architecture.

Here are the 8 key layers to scale your API architecture.

1️⃣ Load Balancer

👉 Distributes traffic across multiple servers so no single machine melts.

Example: 1M req/s split across 200 servers = only 5k req/s each.

Nginx | HAProxy | AWS ALB/NLB

2️⃣ Horizontal Scaling

👉 Add more servers when traffic spikes instead of upgrading one big server.

Example: Black Friday? Spin up 50 more API nodes in seconds.

Auto-scaling groups | Kubernetes | ECS

3️⃣ Caching Layer

👉 Serve frequent reads from Redis/Memcached to avoid DB overload.

Example: User profile cached → avoids 10M database hits/day.

Redis | Memcached | CDN

4️⃣ CDN for Static Content

👉 Images and static assets load from edge servers near the user.

Example: A user in Delhi gets images from a Delhi CDN node.

CloudFront | Cloudflare | Akamai

5️⃣ Async Processing (Queues)

👉 Push heavy tasks to Kafka/SQS so API responds instantly.

Example: Payment API returns fast → receipt email sent in background.

Kafka | RabbitMQ | AWS SQS

6️⃣ Database Sharding

👉 Split huge datasets across multiple DB shards to scale reads/writes.

Example: Users A–M on shard 1, N–Z on shard 2.

Vitess | Citus | application-level sharding

7️⃣ Rate Limiting

👉 Block or throttle abusive clients to protect server capacity.

Example: "100 requests/sec limit" prevents bots from killing the API.

Redis rate limiting | API Gateway throttling

8️⃣ Lightweight Payloads

👉 Reduce JSON response size to cut latency and bandwidth.

Example: Return only "id, name, price" instead of 20 unnecessary fields.

Payload optimization | GraphQL for selective queries

💡 Problem 2: Choosing the Right Load Balancing Algorithm

Question: INTERVIEWER: You have 10 backend servers. How does the load balancer decide which server gets the next request?

Solution:

Load balancing algorithms determine how traffic is distributed across servers.

1️⃣ Round Robin

👉 Distributes requests sequentially to each server in rotation.

Example: Request 1 → Server A, Request 2 → Server B, Request 3 → Server C, repeat.

Simple | works well when all servers have equal capacity

2️⃣ Weighted Round Robin

👉 Servers with higher capacity get more requests based on assigned weights.

Example: Server A (weight 3) gets 3 requests for every 1 request Server B (weight 1) gets.

Use when servers have different CPU/RAM specs

3️⃣ Least Connections

👉 Routes requests to the server handling the fewest active connections.

Example: Server A has 50 connections, Server B has 30 → new request goes to Server B.

Best for long-lived connections like WebSockets

4️⃣ IP Hash

👉 Uses client's IP address to consistently route to same server.

Example: User from IP 192.168.1.1 always goes to Server C (maintains session affinity).

Sticky sessions | stateful applications

5️⃣ Least Response Time

👉 Routes to server with fastest response time and fewest active connections.

Example: Server A responds in 50ms, Server B in 100ms → prefer Server A.

Performance-optimized routing

6️⃣ Random

👉 Randomly selects a server for each request.

Example: Each request has equal probability of hitting any server.

Simple | good for stateless microservices

💡 Problem 3: Layer 4 vs Layer 7 Load Balancing

Question: INTERVIEWER: Should you use a Layer 4 or Layer 7 load balancer for your application?

Solution:

Layer 4 and Layer 7 load balancers operate at different network layers with different capabilities.

1️⃣ Layer 4 (Transport Layer)

👉 Routes traffic based on IP address and TCP/UDP port.

Example: Like a mail sorter who only looks at zip codes — fast but doesn't read letter contents.

AWS NLB | HAProxy in TCP mode

2️⃣ Layer 7 (Application Layer)

👉 Routes based on HTTP headers, URL paths, cookies, or content.

Example: Like a smart receptionist who reads your request and directs you to the right department.

AWS ALB | Nginx | HAProxy in HTTP mode

3️⃣ When to Use Layer 4

👉 Need maximum speed and lowest latency (no content inspection overhead).

Example: High-throughput TCP services, gaming servers, IoT data streams.

Lower latency | higher throughput | simpler

4️⃣ When to Use Layer 7

👉 Need content-based routing, SSL termination, or path-based routing.

Example: Route /api/ to API servers, /static/* to CDN, /admin/* to admin servers.*

More features | flexible routing | SSL offloading

5️⃣ SSL/TLS Handling

👉 Layer 4 passes encrypted traffic through (end-to-end encryption); Layer 7 terminates SSL at load balancer, inspects content, re-encrypts to backend.

Example: Layer 4 preserves end-to-end encryption; Layer 7 trades security for content inspection.

Security vs performance trade-off

6️⃣ Health Checks

👉 Layer 4 uses simple TCP connection test (is port open?); Layer 7 uses HTTP health check (does /health return 200 OK?).

Example: Layer 7 provides smarter health monitoring than Layer 4.

Layer 7 detects application-level failures

📚 Load Balancing Strategies

💡 Problem 4: Health Checks and Failover Strategy

Question: INTERVIEWER: One of your backend servers crashes. How does the load balancer detect this and stop sending traffic to it?

Solution:

Health checks ensure load balancers only route traffic to healthy servers.

1️⃣ Active Health Checks

👉 Load balancer periodically pings each server to verify it's healthy.

Example: Every 10 seconds, send GET /health → expect 200 OK response.

Proactive monitoring

2️⃣ Passive Health Checks

👉 Monitor real traffic and mark server unhealthy after consecutive failures.

Example: If 3 consecutive requests to Server B fail → remove it from pool.

React to actual traffic patterns

3️⃣ Health Check Intervals

👉 Configure how often to check server health (balance between quick detection and server load).

Example: Check every 5 seconds with 2-second timeout, mark unhealthy after 3 failures.

Tuning for responsiveness vs overhead

4️⃣ Graceful Degradation

👉 Continue serving traffic with remaining healthy servers when some fail.

Example: 10 servers → 2 crash → remaining 8 handle traffic until auto-scaling adds replacements.

5️⃣ Circuit Breaker Integration

👉 Temporarily stop routing to consistently failing server to let it recover.

Example: Like giving an injured player a break instead of forcing them to play and worsen injury.

Prevents cascading failures

6️⃣ Connection Draining

👉 Allow in-flight requests to complete before removing server from pool.

Example: Server marked unhealthy → wait 30 seconds for existing requests to finish before fully removing.

Prevents request interruption during deployments

📚 High Availability & Fault Tolerance

💡 Problem 1: Designing a System with 99.99% Uptime

Question: INTERVIEWER: Your e-commerce platform must have 99.99% uptime (only 52 minutes downtime per year). How do you architect for this?

Solution:

High availability requires eliminating single points of failure across all layers.

1️⃣ Multi-Region Deployment

👉 Deploy application across multiple AWS/Azure regions for geographic redundancy.

Example: Primary in US-East, failover in EU-West — if entire US region fails, EU takes over.

Active-active or active-passive setup

2️⃣ Load Balancer Redundancy

👉 Use multiple load balancers with health checks and DNS failover.

Example: Two load balancers behind Route 53 — if one fails, DNS routes to backup.

Eliminates load balancer as single point of failure

3️⃣ Database Replication

👉 Maintain synchronized database replicas with automatic failover.

Example: Primary DB in us-east-1, read replicas in us-west-2 — if primary fails, promote replica.

RDS Multi-AZ | PostgreSQL streaming replication

4️⃣ Stateless Application Servers

👉 Design servers with no local state so any instance failure doesn't lose data.

Example: Session data in Redis, not server memory — users stay logged in even if server crashes.

Horizontal scaling   easy recovery

5️⃣ Health Monitoring & Auto-Recovery

👉 Continuously monitor services and automatically restart or replace failed instances.

Example: Kubernetes detects crashed pod → launches replacement in 10 seconds.

Self-healing infrastructure

6️⃣ Circuit Breakers

👉 Prevent cascading failures by stopping requests to failing dependencies.

Example: Payment service down → circuit opens, show "payment temporarily unavailable" instead of crashing entire checkout.

Resilience patterns

7️⃣ Backup & Disaster Recovery

👉 Regular automated backups with tested restore procedures.

Example: Daily DB snapshots to S3, tested monthly disaster recovery drills.

RTO and RPO planning

8️⃣ Zero-Downtime Deployments

👉 Use blue-green or rolling deployments to update without downtime.

Example: Deploy new version to "green" environment, switch traffic when ready, keep "blue" as instant rollback.

Continuous deployment without service interruption

💡 Problem 2: Handling Partial System Failures

Question: INTERVIEWER: Your recommendation service is down, but users should still be able to browse and buy products. How do you handle this?

Solution:

Graceful degradation ensures core functionality works even when some services fail.

1️⃣ Service Isolation

👉 Each microservice fails independently without bringing down others.

Example: Like a restaurant — if dessert kitchen fails, you can still order main courses.

Microservices architecture | bulkheads

2️⃣ Fallback Responses

👉 Return cached or default data when service is unavailable.

Example: Recommendation service down → show "Popular items" instead of personalized recommendations.

Graceful degradation

3️⃣ Timeouts & Retries

👉 Set aggressive timeouts to fail fast and retry with exponential backoff.

Example: Wait max 500ms for recommendation service → if timeout, show default recommendations.

Prevent waiting indefinitely for failed services

4️⃣ Circuit Breaker Pattern

👉 Stop calling failing service temporarily to let it recover.

Example: After 5 consecutive failures, stop calling recommendation service for 60 seconds — prevents overload.

Hystrix Resilience4j

5️⃣ Feature Flags

👉 Dynamically disable non-critical features during incidents.

Example: Turn off "related products" section if service is struggling — keeps checkout working.

LaunchDarkly feature toggles

6️⃣ Async Background Jobs

👉 Move non-critical operations to background queues that can be retried later.

Example: Order confirmation email fails → queue it for retry, but complete the purchase.

Kafka SQS delayed job processing

💡 Problem 3: Implementing Bulkhead Pattern

Question: INTERVIEWER: How do you prevent one slow service from exhausting all resources?

Solution:

Bulkhead pattern isolates resources to contain failures.

1️⃣ Resource Isolation

👉 Separate thread pools/connection pools for different services.

Example: Payment service gets 10 threads, Search gets 20 threads → isolated.

Thread pools resource isolation

2️⃣ Preventing Cascading Failures

👉 Slow payment service doesn't block search functionality.

Example: Payment threads exhausted → search still works with its own threads.

Failure containment | independent pools

3️⃣ Connection Pool Bulkheads

👉 Separate database connection pools per service/feature.

Example: Analytics queries have separate pool → don't block transactional queries.

Connection pooling | query isolation

4️⃣ Semaphore Pattern

👉 Limit concurrent requests to external service.

Example: Max 5 concurrent calls to third-party API → prevents overwhelming it.

Semaphores | concurrency limits

5️⃣ Queue-Based Bulkheads

👉 Use separate queues for different workloads.

Example: High-priority orders queue separate from bulk reports queue.

Queue separation | priority handling

6️⃣ Kubernetes Resource Limits

👉 Set CPU/memory limits per pod to prevent resource hogging.

Example: Each microservice pod limited to 2 CPU cores, 4GB RAM.

Resource quotas | Kubernetes limits

💡 Problem 3: Single Point of Failure Elimination

Question: INTERVIEWER: Your system has a single Redis instance for caching. What happens if it crashes?

Solution:

Identify and eliminate every single point of failure in your architecture.

1️⃣ Redis Clustering

👉 Distribute cache across multiple Redis nodes with automatic failover.

Example: 3-node Redis cluster — if one node fails, others continue serving requests.

Redis Sentinel | Redis Cluster

2️⃣ Cache-Aside Pattern with Fallback

👉 If cache is down, application falls back to database directly.

Example: Try Redis → if timeout, query database directly → slower but system stays up.

Degraded performance | but no complete failure

3️⃣ Message Queue Redundancy

👉 Use clustered message brokers with replication.

Example: 3 Kafka brokers with replication factor 3 — losing one broker doesn't lose messages.

Kafka clusters | RabbitMQ mirrored queues

4️⃣ Database High Availability

👉 Primary database with automatic failover to replica.

Example: PostgreSQL primary with 2 streaming replicas — if primary fails, promote replica in seconds.

AWS RDS Multi-AZ | Patroni for PostgreSQL

5️⃣ Network Redundancy

👉 Multiple network paths and availability zones.

Example: Deploy across 3 availability zones — if one AZ loses connectivity, traffic routes to others.

Multi-AZ deployment | cross-region replication

6️⃣ Monitoring & Alerting

👉 Detect failures instantly and alert team for manual intervention if needed.

Example: Redis down → PagerDuty alert fires within 30 seconds → team investigates.

Prometheus | Datadog | CloudWatch

📚 CAP Theorem

💡 Problem 1: Choosing Between Consistency and Availability

Question: INTERVIEWER: You're building a social media 'likes' counter. Should you prioritize consistency or availability?

Solution:

CAP theorem states you can only have 2 of 3: Consistency, Availability, Partition Tolerance.

1️⃣ CP Systems (Consistency + Partition Tolerance)

👉 Always return correct data, even if it means refusing requests during network partitions.

Example: Banking system — better to show "temporarily unavailable" than show wrong account balance.

MongoDB with majority writes | HBase | ZooKeeper

2️⃣ AP Systems (Availability + Partition Tolerance)

👉 Always respond to requests, even if data might be slightly stale or inconsistent.

Example: Social media likes — showing 99 likes instead of 100 for a few seconds is acceptable.

Cassandra | DynamoDB with eventual consistency | Riak

3️⃣ CA Systems (Consistency + Availability)

👉 Traditional RDBMS that assume no network partitions (single datacenter).

Example: PostgreSQL in single AZ — consistent and available, but entire DB goes down if network partitions.

PostgreSQL | MySQL without replication

4️⃣ Eventual Consistency Trade-off

👉 Accept temporary inconsistency in exchange for high availability.

Example: Like news spreading through social media — everyone eventually hears it, but not instantly.

DNS | CDN caching | distributed counters

5️⃣ Strong Consistency Cost

👉 Requires coordination between nodes, increasing latency.

Example: Like getting everyone in a meeting to agree before announcing decision — slower but accurate.

Distributed transactions | Paxos/Raft consensus

6️⃣ Real-World Example: Amazon Shopping Cart

👉 Availability prioritized — even during network issues, you can add items to cart.

Example: Occasionally you might add same item twice, but that's better than checkout being down.

AP system design

💡 Problem 2: Network Partition Handling

Question: INTERVIEWER: Your distributed database splits into two partitions due to network failure. How do you handle reads and writes?

Solution:

Network partitions are inevitable in distributed systems — must choose how to handle them.

1️⃣ Quorum-Based Writes

👉 Require majority of nodes to acknowledge write before confirming success.

Example: 5-node cluster → write succeeds only if 3+ nodes confirm (majority quorum).

Cassandra | DynamoDB with quorum consistency

2️⃣ Split-Brain Prevention

👉 Ensure only one partition can accept writes during network split.

Example: Partition with majority of nodes stays active, minority partition refuses writes.

Avoids conflicting updates

3️⃣ Read Repair

👉 Detect and fix inconsistencies when reading data from multiple replicas.

Example: Query 3 nodes → 2 return version A, 1 returns old version B → update B to A.

Cassandra read repair | anti-entropy

4️⃣ Version Vectors / Vector Clocks

👉 Track causality of updates to resolve conflicts during partition recovery.

Example: Like Git merge — system knows which updates happened before/after others.

DynamoDB | Riak

5️⃣ Last-Write-Wins (LWW)

👉 Simplest conflict resolution — newest timestamp wins.

Example: Two users edit same document during partition → most recent edit overwrites older one.

Risk: may lose legitimate updates

6️⃣ Application-Level Conflict Resolution

👉 Let application decide how to merge conflicting updates.

Example: Shopping cart — if partition causes duplicate items, merge by adding quantities.

Custom business logic for conflicts

💡 Problem 3: HTTP/2 vs HTTP/3 Protocol Differences

Question: INTERVIEWER: Your app uses HTTP/1.1 — what do you gain by upgrading to HTTP/2 or HTTP/3?

Solution:

HTTP/2 and HTTP/3 solve head-of-line blocking and improve performance over slow networks.

1️⃣ HTTP/1.1 Limitations

👉 One request per TCP connection, or head-of-line blocking in pipelining.

Example: 6 parallel connections limit → page with 100 assets loads slowly.

HTTP/1.1 connection limits

2️⃣ HTTP/2 Multiplexing

👉 Multiple requests/responses over single TCP connection.

Example: 100 assets → all loaded over 1 connection → no connection limit bottleneck.

Multiplexing single connection

3️⃣ HTTP/2 Server Push

👉 Server proactively sends resources before client requests.

Example: Server pushes CSS, JS immediately after HTML → faster page load.

Server push preload

4️⃣ HTTP/2 Header Compression (HPACK)

👉 Compress repeated headers to reduce overhead.

Example: Cookie header repeated in 100 requests → HPACK compresses → saves bandwidth.

HPACK compression

5️⃣ HTTP/3 QUIC Protocol

👉 Built on UDP instead of TCP → faster connection establishment.

Example: TCP requires 3-way handshake (1-2 RTTs); QUIC: 0-1 RTT → faster.

QUIC UDP 0-RTT

6️⃣ HTTP/3 No Head-of-Line Blocking

👉 TCP packet loss blocks all streams; QUIC isolates streams.

Example: 1 lost packet in HTTP/2 blocks all requests; HTTP/3 only blocks affected stream.

No HOL blocking independent streams

7️⃣ When to Upgrade

👉 HTTP/2 for multiplexing; HTTP/3 for mobile/lossy networks.

Example: E-commerce site with many assets → HTTP/2; Global app → HTTP/3 for better mobile performance.

Migration use cases

💡 Problem 3: HTTP/2 vs HTTP/3 Protocol Differences

Question: INTERVIEWER: Your app uses HTTP/1.1 — what do you gain by upgrading to HTTP/2 or HTTP/3?

Solution:

HTTP/2 and HTTP/3 solve head-of-line blocking and improve performance over slow networks.

1️⃣ HTTP/1.1 Limitations

👉 One request per TCP connection, or head-of-line blocking in pipelining.

Example: 6 parallel connections limit → page with 100 assets loads slowly.

HTTP/1.1 | connection limits

2️⃣ HTTP/2 Multiplexing

👉 Multiple requests/responses over single TCP connection.

Example: 100 assets → all loaded over 1 connection → no connection limit bottleneck.

Multiplexing | single connection

3️⃣ HTTP/2 Server Push

👉 Server proactively sends resources before client requests.

Example: Server pushes CSS, JS immediately after HTML → faster page load.

Server push | preload

4️⃣ HTTP/2 Header Compression (HPACK)

👉 Compress repeated headers to reduce overhead.

Example: Cookie header repeated in 100 requests → HPACK compresses → saves bandwidth.

HPACK compression

5️⃣ HTTP/3 QUIC Protocol

👉 Built on UDP instead of TCP → faster connection establishment.

Example: TCP requires 3-way handshake (1-2 RTTs); QUIC: 0-1 RTT → faster.

QUIC UDP 0-RTT

6️⃣ HTTP/3 No Head-of-Line Blocking

👉 TCP packet loss blocks all streams; QUIC isolates streams.

Example: 1 lost packet in HTTP/2 blocks all requests; HTTP/3 only blocks affected stream.

No HOL blocking independent streams

7️⃣ When to Upgrade

👉 HTTP/2 for multiplexing; HTTP/3 for mobile/lossy networks.

Example: E-commerce site with many assets → HTTP/2; Global app → HTTP/3 for better mobile performance.

Migration use cases

📚 Idempotency

💡 Problem 1: Preventing Duplicate Payment Processing

Question: INTERVIEWER: User double-clicks the 'Pay Now' button. How do you prevent charging them twice?

Solution:

Idempotency ensures repeating the same operation produces the same result without side effects.

1️⃣ Idempotency Key

👉 Assign a unique ID to each request so backend recognizes duplicates.

Example: User clicks "Pay Now" twice → both requests use same payment_id → processed only once.

`Stripe idempotency keys` `UUID per transaction`

2️⃣ Database Unique Constraint

👉 Database rejects duplicate entries based on unique key.

Example: orders table has UNIQUE constraint on order_id → second insert with same order_id fails gracefully.

`PostgreSQL UNIQUE` `INSERT ... ON CONFLICT`

3️⃣ Distributed Lock

👉 Acquire lock before processing to prevent concurrent duplicates.

Example: Redis SET NX with order_id as key → only first request gets lock and processes payment.

`Redis SETNX` `DynamoDB conditional writes`

4️⃣ Check-Before-Process

👉 Query if request was already processed before executing.

Example: Before processing payment, check if transaction_id already exists in database.

Simple but race conditions possible without locks

5️⃣ Frontend Debouncing

👉 Disable button immediately after first click to prevent duplicate submissions.

Example: Like elevator button — pressing multiple times doesn't call multiple elevators.

Client-side prevention | but not sufficient alone

6️⃣ API Response Caching

👉 Cache API responses by idempotency key for safe retries.

Example: First request processed → cache response for 24 hours → retries return cached response.

Stripe caches idempotent responses for 24 hours

💡 Problem 2: Idempotent REST API Design

Question: INTERVIEWER: Which HTTP methods should be idempotent, and how do you implement idempotency in POST requests?

Solution:

Idempotency is critical for reliable API design, especially for retries and network failures.

1️⃣ GET Requests (Idempotent by Design)

👉 Multiple GET requests return same data without modifying state.

Example: Calling GET /users/123 ten times returns same user data every time.

Read operations are naturally idempotent

2️⃣ PUT Requests (Idempotent by Design)

👉 Updating a resource with same data multiple times produces same result.

Example: PUT /users/123 {"name": "John"} — calling twice still results in name = "John".

Full resource replacement

3️⃣ DELETE Requests (Idempotent)

👉 Deleting same resource multiple times has same effect (resource is gone).

Example: DELETE /users/123 — first call deletes user, subsequent calls return 404 but system state unchanged.

Deleting already-deleted resource is safe

4️⃣ POST Requests (NOT Idempotent by Default)

👉 POST creates new resources, so repeating creates duplicates.

Example: POST /orders creates new order each time — duplicates are problem.

Need explicit idempotency handling

5️⃣ Idempotency-Key Header

👉 Client sends unique idempotency key with POST requests.

Example: POST /orders with Idempotency-Key: abc123 → server stores key and result → retries return cached result.

Stripe-style idempotency

6️⃣ UUID-Based Resource Creation

👉 Client generates UUID and uses PUT instead of POST.

Example: PUT /orders/550e8400-e29b-41d4-a716-446655440000 → creates or updates order with that ID.

Client-generated IDs

7️⃣ Response Status Codes

👉 Return appropriate status codes for idempotent operations.

Example: First POST returns 201 Created, duplicate returns 200 OK with original resource.

Clear communication of idempotency

💡 Problem 3: Message Queue Idempotency

Question: INTERVIEWER: A message in your Kafka queue gets processed twice due to retry. How do you handle this?

Solution:

Message queue idempotency prevents duplicate processing of messages.

1️⃣ Message Deduplication

👉 Store message IDs in database or cache to detect duplicates.

Example: Like a bouncer with a guest list — checks if person already entered before letting them in.

Redis set with message IDs | database unique constraint

2️⃣ At-Least-Once with Idempotent Processing

👉 Accept that messages may be delivered multiple times, design processing to be idempotent.

Example: Email notification job → check if email already sent before sending again.

Kafka at-least-once delivery with idempotent consumers

3️⃣ Exactly-Once Semantics

👉 Use messaging system's built-in exactly-once delivery guarantees.

Example: Kafka transactions ensure message is processed exactly once across consumer group.

Kafka exactly-once semantics | SQS FIFO with deduplication

4️⃣ Database Transaction with Message Ack

👉 Process message and acknowledge it within same database transaction.

Example: Debit account AND mark message as processed in single transaction → atomic operation.

Transactional outbox pattern

5️⃣ Versioning / Timestamps

👉 Include version or timestamp in messages to detect stale duplicates.

Example: Two messages to update user email → process only the one with newer timestamp.

Conflict resolution

6️⃣ Bloom Filters for Fast Duplicate Detection

👉 Use probabilistic data structure to quickly check if message was seen before.

Example: Like a quick memory check — "have I seen this message ID before?"

Memory-efficient deduplication for high-throughput systems

📚 Circuit Breaker Pattern

💡 Problem 1: Preventing Cascading Failures

Question: INTERVIEWER: Your payment service is down and causing timeouts across your entire application. How do you prevent this from bringing down everything?

Solution:

Circuit breakers protect your system from cascading failures when dependencies fail.

1️⃣ Circuit Breaker States

👉 Circuit breakers have three states: Closed (normal), Open (failing), and Half-Open (testing recovery).

Example: Like an electrical circuit breaker that trips when overloaded, then tests before fully reconnecting.

Hystrix | Resilience4j | custom implementation

2️⃣ Closed State (Normal Operation)

👉 All requests pass through normally while monitoring for failures.

Example: Payment service working fine → all requests go through → error rate monitored.

Success threshold: < 50% error rate

3️⃣ Open State (Fail Fast)

👉 Immediately reject requests without calling the failing service.

Example: Payment service down → circuit opens → return cached response or error immediately.

Prevents wasting time on doomed requests

4️⃣ Half-Open State (Testing Recovery)

👉 Allow limited requests through to test if service has recovered.

Example: After 30 seconds, try 5 requests → if successful, close circuit; if failed, reopen.

Gradual recovery testing

5️⃣ Fallback Mechanisms

👉 Provide alternative responses when circuit is open.

Example: Payment service down → show "Pay with alternative method" or use cached data.

Graceful degradation

6️⃣ Failure Thresholds

👉 Define error rate percentage or count that triggers circuit to open.

Example: Open circuit after 50% error rate over 10 requests, or 5 consecutive failures.

Prevents false positives from single failures

7️⃣ Timeout Configuration

👉 Set aggressive timeouts to detect failures quickly.

Example: 3-second timeout → if service doesn't respond, count as failure.

Fast failure detection

8️⃣ Monitoring & Alerts

👉 Track circuit state changes and alert when circuits open.

Example: Circuit opened → alert team → investigate root cause before customers complain.

Prometheus metrics | Datadog monitoring

💡 Problem 2: Circuit Breaker for External API Calls

Question: INTERVIEWER: Your app calls a third-party weather API that's flaky. How do you ensure it doesn't slow down your entire application?

Solution:

Circuit breakers isolate external dependencies to prevent them from impacting your system.

1️⃣ Per-Dependency Circuit Breakers

👉 Use separate circuit breakers for each external dependency.

Example: Weather API circuit independent of Maps API circuit → one fails, others keep working.

Isolated failure domains

2️⃣ Request Timeout

👉 Set short timeouts for external API calls to fail fast.

Example: Weather API timeout = 2 seconds → don't wait 30 seconds for slow response.

Prevent indefinite blocking

3️⃣ Retry with Exponential Backoff

👉 Retry failed requests with increasing delays between attempts.

Example: Retry after 1s, then 2s, then 4s → gives failing service time to recover.

Avoid overwhelming failing service

4️⃣ Cached Fallback Data

👉 Return stale but cached data when external API is down.

Example: Weather API down → show yesterday's weather data instead of error.

Better user experience than complete failure

5️⃣ Bulkhead Pattern

👉 Isolate thread pools for different external dependencies.

Example: 10 threads for weather API, 10 for maps API → one exhausted pool doesn't affect other.

Resource isolation

6️⃣ Health Check Endpoint

👉 Periodically check external API health to preemptively open circuit.

Example: Poll /health every 30s → if unhealthy, open circuit before user requests fail.

Proactive monitoring

📚 Rate Limiting & Throttling

💡 Problem 1: Protecting API from Abuse

Question: INTERVIEWER: Your public API is getting hammered by bots making 10,000 requests per second. How do you protect it?

Solution:

Rate limiting prevents abuse and ensures fair usage of your API resources.

1️⃣ Fixed Window Rate Limiting

👉 Count requests in fixed time windows (e.g., per minute).

Example: Allow 100 requests per minute → reset counter at start of each minute.

Simple | but burst traffic issues at window boundaries

2️⃣ Sliding Window Rate Limiting

👉 Use rolling time window for smoother rate limiting.

Example: Count requests in last 60 seconds (sliding window) → prevents burst at window boundaries.

More accurate | prevents window boundary exploitation

3️⃣ Token Bucket Algorithm

👉 Tokens refill at steady rate, requests consume tokens.

Example: Bucket holds 100 tokens, refills 10/second → allows bursts up to 100, maintains average rate.

Allows bursts while maintaining average rate

4️⃣ Leaky Bucket Algorithm

👉 Process requests at fixed rate, queue excess requests.

Example: Like a funnel → water (requests) pours in fast, but drains out at steady rate.

Smooths traffic spikes | strict rate enforcement

5️⃣ Rate Limit Tiers

👉 Different rate limits for different user tiers.

Example: Free tier: 100 req/hour, Pro: 1000 req/hour, Enterprise: unlimited.

Monetization | fair resource allocation

6️⃣ Distributed Rate Limiting with Redis

👉 Use centralized Redis to track rate limits across multiple servers.

Example: All API servers check same Redis counter → consistent rate limiting.

INCR command with EXPIRE | synchronized across cluster

7️⃣ Rate Limit Headers

👉 Return rate limit info in response headers.

Example: X-RateLimit-Remaining: 45, X-RateLimit-Reset: 1609459200.

Client knows when to retry

8️⃣ IP-Based vs User-Based Limiting

👉 Rate limit by IP address for anonymous users, by user ID for authenticated.

Example: Anonymous users: 10 req/min per IP, Logged in: 100 req/min per user.

Flexible limiting strategies

💡 Problem 2: Handling Rate Limit Violations

Question: INTERVIEWER: A user exceeds their rate limit. What should your API return and how should it behave?

Solution:

Proper rate limit handling improves user experience and protects your API.

1️⃣ HTTP 429 Status Code

👉 Return 429 Too Many Requests when rate limit exceeded.

Example: User exceeds limit → return 429 with clear error message.

Standard HTTP status for rate limiting

2️⃣ Retry-After Header

👉 Tell client when they can retry.

Example: Retry-After: 60 → client should wait 60 seconds before retrying.

Helps clients implement proper backoff

3️⃣ Error Response Body

👉 Include helpful error message explaining rate limit.

Example: {"error": "Rate limit exceeded. Try again in 30 seconds."}

Actionable error information

4️⃣ Soft Throttling vs Hard Blocking

👉 Soft throttle by slowing down, hard block by rejecting.

Example: Soft: add 100ms delay per request over limit; Hard: reject immediately.

Tiered service quality

5️⃣ Burst Allowance

👉 Allow short bursts above sustained rate.

Example: Average 10 req/sec allowed, but burst of 50 requests tolerated briefly.

Token bucket allows this flexibility

6️⃣ Warning Thresholds

👉 Warn users before they hit rate limit.

Example: At 80% of limit, return X-RateLimit-Warning header.

Proactive communication

💡 Problem 3: Advanced Rate Limiting Algorithms

Question: INTERVIEWER: Your API has different rate limits for free vs paid users — how do you implement that?

Solution:

Advanced rate limiting balances fairness, resource protection, and business requirements.

1️⃣ Token Bucket Algorithm

👉 Bucket fills with tokens at fixed rate, each request consumes token.

Example: 100 tokens/minute bucket → burst of 100 requests allowed, then 100/min sustained.

Token bucket burst handling

2️⃣ Leaky Bucket Algorithm

👉 Requests queued, processed at fixed rate.

Example: Queue holds 50 requests → process 10/sec → smooths traffic spikes.

Leaky bucket traffic smoothing

3️⃣ Fixed Window Counter

👉 Count requests per fixed time window.

Example: 1000 requests per hour → resets at top of hour → potential burst at boundary.

Fixed window | simple counter

4️⃣ Sliding Window Log

👉 Track timestamp of each request, count in sliding window.

Example: Track last 100 request times → allow request if count in last minute < 100.

Sliding window | accurate limiting

5️⃣ User-Tier Based Limits

👉 Different limits per user tier.

Example: Free: 100/hr, Pro: 1K/hr, Enterprise: 10K/hr → stored in Redis.

Tiered limits | business logic

6️⃣ Distributed Rate Limiting

👉 Coordinate limits across multiple API servers.

Example: Redis tracks global count → all API servers check/increment → consistent limits.

Distributed | Redis coordination

7️⃣ Rate Limit Headers

👉 Inform clients of their limit status.

Example: X-RateLimit-Limit: 1000, X-RateLimit-Remaining: 847 → client can self-throttle.

HTTP headers | client awareness

💡 Problem 3: Advanced Rate Limiting Algorithms

Question: INTERVIEWER: Your API has different rate limits for free vs paid users — how do you implement that?

Solution:

Advanced rate limiting balances fairness, resource protection, and business requirements.

1️⃣ Token Bucket Algorithm

👉 Bucket fills with tokens at fixed rate, each request consumes token.

Example: 100 tokens/minute bucket → burst of 100 requests allowed, then 100/min sustained.

`Token bucket` `burst handling`

2️⃣ Leaky Bucket Algorithm

👉 Requests queued, processed at fixed rate.

Example: Queue holds 50 requests → process 10/sec → smooths traffic spikes.

`Leaky bucket` `traffic smoothing`

3️⃣ Fixed Window Counter

👉 Count requests per fixed time window.

Example: 1000 requests per hour → resets at top of hour → potential burst at boundary.

`Fixed window` `simple counter`

4️⃣ Sliding Window Log

👉 Track timestamp of each request, count in sliding window.

Example: Track last 100 request times → allow request if count in last minute < 100.

`Sliding window` `accurate limiting`

5️⃣ User-Tier Based Limits

👉 Different limits per user tier.

Example: Free: 100/hr, Pro: 1K/hr, Enterprise: 10K/hr → stored in Redis.

Tiered limits | business logic

6️⃣ Distributed Rate Limiting

👉 Coordinate limits across multiple API servers.

Example: Redis tracks global count → all API servers check/increment → consistent limits.

Distributed | Redis coordination

7️⃣ Rate Limit Headers

👉 Inform clients of their limit status.

Example: X-RateLimit-Limit: 1000, X-RateLimit-Remaining: 847 → client can self-throttle.

HTTP headers | client awareness

💡 Problem 3: DDoS Protection Strategy

Question: INTERVIEWER: Your API is under DDoS attack with millions of requests from different IPs. How do you defend it?

Solution:

DDoS protection requires multiple layers of defense beyond simple rate limiting.

1️⃣ CDN & Edge Protection

👉 Use CDN to absorb and filter DDoS traffic at edge.

Example: Cloudflare blocks malicious traffic before it reaches your servers.

Cloudflare | AWS Shield | Akamai

2️⃣ IP Reputation & Blocking

👉 Block known malicious IPs automatically.

Example: IP from known botnet → blocked at firewall level.

WAF rules IP blocklists

3️⃣ CAPTCHA for Suspicious Traffic

👉 Challenge suspicious requests with CAPTCHA.

Example: 100 requests in 10 seconds from one IP → require CAPTCHA verification.

reCAPTCHA hCaptcha

4️⃣ Connection Limiting

👉 Limit concurrent connections per IP.

Example: Max 10 concurrent connections per IP → prevents connection exhaustion.

nginx limit_conn AWS WAF

5️⃣ Request Size Limits

👉 Reject oversized requests to prevent payload attacks.

Example: Reject requests larger than 10MB.

Prevent payload-based attacks

6️⃣ Geo-Blocking

👉 Block traffic from regions you don't serve.

Example: US-only service → block traffic from other countries during attack.

CloudFront geo-restrictions

7️⃣ Auto-Scaling with Limits

👉 Scale automatically but set max limits to control costs.

Example: Auto-scale up to 100 servers max → prevents unlimited cloud bills during DDoS.

Budget protection during attacks

8️⃣ API Key Requirements

👉 Require API keys for all requests to enable accountability.

Example: Anonymous requests → reject; API key required → can track and block abusers.

Accountability and blocking granularity

💡 Problem 4: Designing Secure VPC Architecture

Question: INTERVIEWER: How do you design a VPC that's secure, scalable, and allows for disaster recovery?

Solution:

VPC design establishes network boundaries, security zones, and connectivity patterns.

1️⃣ Public vs Private Subnets

👉 Public subnets for internet-facing resources, private for internal.

Example: ALB in public subnet → app servers in private subnet → no direct internet access.

Subnet separation | security zones

2️⃣ Multi-AZ Deployment

👉 Spread resources across availability zones.

Example: 3 AZs → app servers in each → survive AZ failure.

Multi-AZ | high availability

3️⃣ NAT Gateway for Outbound Traffic

👉 Private subnets access internet via NAT gateway.

Example: App server downloads packages → routed through NAT gateway in public subnet.

NAT Gateway outbound access

4️⃣ Security Groups (Stateful Firewall)

👉 Control inbound/outbound traffic at instance level.

Example: App server SG allows port 80/443 from ALB SG only → no direct internet access.

Security Groups firewall rules

5️⃣ Network ACLs (Stateless Firewall)

👉 Additional subnet-level firewall rules.

Example: Block known malicious IP ranges at NACL level → never reach instances.

NACLs subnet firewall

6️⃣ VPC Peering for Multi-Account

👉 Connect VPCs across accounts or regions.

Example: Production VPC peers with monitoring VPC → centralized logging.

VPC Peering multi-account

7️⃣ VPN/Direct Connect for Hybrid

👉 Secure connection between on-premise and cloud.

Example: Corporate network → VPN to AWS VPC → access internal resources securely.

VPN Direct Connect hybrid cloud

💡 Problem 4: Designing Secure VPC Architecture

Question: INTERVIEWER: How do you design a VPC that's secure, scalable, and allows for disaster recovery?

Solution:

VPC design establishes network boundaries, security zones, and connectivity patterns.

1️⃣ Public vs Private Subnets

👉 Public subnets for internet-facing resources, private for internal.

Example: ALB in public subnet → app servers in private subnet → no direct internet access.

Subnet separation | security zones

2️⃣ Multi-AZ Deployment

👉 Spread resources across availability zones.

Example: 3 AZs → app servers in each → survive AZ failure.

Multi-AZ | high availability

3️⃣ NAT Gateway for Outbound Traffic

👉 Private subnets access internet via NAT gateway.

Example: App server downloads packages → routed through NAT gateway in public subnet.

NAT Gateway | outbound access

4️⃣ Security Groups (Stateful Firewall)

👉 Control inbound/outbound traffic at instance level.

Example: App server SG allows port 80/443 from ALB SG only → no direct internet access.

Security Groups | firewall rules

5️⃣ Network ACLs (Stateless Firewall)

👉 Additional subnet-level firewall rules.

Example: Block known malicious IP ranges at NACL level → never reach instances.

NACLs | subnet firewall

6️⃣ VPC Peering for Multi-Account

👉 Connect VPCs across accounts or regions.

Example: Production VPC peers with monitoring VPC → centralized logging.

VPC Peering | multi-account

7️⃣ VPN/Direct Connect for Hybrid

👉 Secure connection between on-premise and cloud.

Example: Corporate network → VPN to AWS VPC → access internal resources securely.

VPN | Direct Connect | hybrid cloud

💡 Problem 4: Gateway vs Application-Level Rate Limiting

Question: INTERVIEWER: Your API is being hammered by bots — rate-limit at gateway or app level?

Solution:

Multi-layer rate limiting provides defense in depth against abuse.

1️⃣ Gateway-Level Rate Limiting (First Line of Defense)

👉 Implement rate limiting at API Gateway before requests hit application.

Example: AWS API Gateway blocks 10K req/sec from single IP → app servers never see the traffic.

API Gateway throttling | WAF rate limits | edge protection

2️⃣ Application-Level Rate Limiting (Business Logic)

👉 Enforce per-user or per-feature rate limits based on business rules.

Example: Free tier: 100 API calls/day, Premium: 10K calls/day → enforced at app layer.

Custom rate limiting | tiered limits | business rules

3️⃣ Distributed Rate Limiting

👉 Share rate limit state across application instances via Redis.

Example: 3 app servers share Redis counter → consistent rate limit across cluster.

Redis rate limiting | centralized counters

4️⃣ IP-Based vs Token-Based Limiting

👉 Use IP limiting at gateway, token/user limiting at application.

Example: Gateway: max 1K req/sec per IP; App: max 100 req/min per API key.

Layered defense | different granularity

5️⃣ Adaptive Rate Limiting

👉 Dynamically adjust limits based on system load.

Example: CPU >80% → reduce rate limits by 50% → protect backend from overload.

Dynamic limits | load-based throttling

6️⃣ CAPTCHA Integration

👉 Challenge suspicious traffic patterns with CAPTCHA.

Example: Same IP makes 100 requests in 10 seconds → require CAPTCHA verification.

reCAPTCHA | bot detection

💡 Problem 4: Gateway vs Application-Level Rate Limiting

Question: INTERVIEWER: Your API is being hammered by bots — rate-limit at gateway or app level?

Solution:

Multi-layer rate limiting provides defense in depth against abuse.

1️⃣ Gateway-Level Rate Limiting (First Line of Defense)

👉 Implement rate limiting at API Gateway before requests hit application.

Example: AWS API Gateway blocks 10K req/sec from single IP → app servers never see the traffic.

`API Gateway throttling` `WAF rate limits` `edge protection`

2️⃣ Application-Level Rate Limiting (Business Logic)

👉 Enforce per-user or per-feature rate limits based on business rules.

Example: Free tier: 100 API calls/day, Premium: 10K calls/day → enforced at app layer.

`Custom rate limiting` `tiered limits` `business rules`

3️⃣ Distributed Rate Limiting

👉 Share rate limit state across application instances via Redis.

Example: 3 app servers share Redis counter → consistent rate limit across cluster.

`Redis rate limiting` `centralized counters`

4️⃣ IP-Based vs Token-Based Limiting

👉 Use IP limiting at gateway, token/user limiting at application.

Example: Gateway: max 1K req/sec per IP; App: max 100 req/min per API key.

`Layered defense` `different granularity`

5️⃣ Adaptive Rate Limiting

👉 Dynamically adjust limits based on system load.

Example: CPU >80% → reduce rate limits by 50% → protect backend from overload.

Dynamic limits | load-based throttling

6️⃣ CAPTCHA Integration

👉 Challenge suspicious traffic patterns with CAPTCHA.

Example: Same IP makes 100 requests in 10 seconds → require CAPTCHA verification.

reCAPTCHA | bot detection

📚 Graceful Degradation

💡 Problem 1: Serving Core Features During Outages

Question: INTERVIEWER: Your recommendation engine is down. How do you ensure users can still shop and checkout?

Solution:

Graceful degradation prioritizes core features when non-critical services fail.

1️⃣ Feature Prioritization

👉 Identify core vs nice-to-have features.

Example: Checkout is core, recommendations are nice-to-have → disable recommendations first.

Business impact analysis

2️⃣ Static Fallback Content

👉 Show static content when dynamic service fails.

Example: Recommendation engine down → show static "Trending items" list.

Simple | reliable fallback

3️⃣ Cached Fallback

👉 Serve stale cached data when live service unavailable.

Example: Product reviews service down → show cached reviews from 1 hour ago.

Better user experience than errors

4️⃣ Feature Toggles

👉 Remotely disable features without deploying code.

Example: Service struggling → flip feature flag to disable recommendations → instant relief.

LaunchDarkly | Unleash | custom flags

5️⃣ Reduced Functionality Mode

👉 Offer limited functionality instead of complete failure.

Example: Search down → enable browse by category only.

Partial service better than none

6️⃣ User Communication

👉 Display banners explaining reduced functionality.

Example: "We're experiencing high traffic. Some features may be slower than usual."

Transparency builds trust

💡 Problem 2: Database Read Replica Failure Handling

Question: INTERVIEWER: Your read replicas are lagging 5 minutes behind primary. How do you handle user queries?

Solution:

Handle replication lag gracefully to balance consistency and availability.

1️⃣ Read-After-Write Consistency

👉 Route read requests to primary after write to ensure consistency.

Example: User updates profile → next read goes to primary, not lagging replica.

Session-based routing to primary

2️⃣ Staleness Tolerance Detection

👉 Determine which queries can tolerate stale data.

Example: Product catalog can be 5 minutes stale, but order status needs real-time data.

Business logic determines acceptable staleness

3️⃣ Fallback to Primary

👉 Query primary database when replicas are too stale.

Example: Replica lag > 10 seconds → route critical reads to primary.

Consistency over performance

4️⃣ Display Staleness Warnings

👉 Show users when data might be outdated.

Example: "This information is up to 2 minutes old" banner.

Transparency about eventual consistency

5️⃣ Session Affinity

👉 Route user's reads to same replica for consistent view.

Example: User always reads from replica-2 during their session → sees their own writes.

Consistent read view within session

6️⃣ Monitoring & Alerting

👉 Detect failures instantly and alert team for manual intervention if needed.

Example: Redis down → PagerDuty alert fires within 30 seconds → team investigates.

Proactive issue detection

📚 Distributed Tracing

💡 Problem 1: Debugging Slow Requests Across Microservices

Question: INTERVIEWER: A user reports checkout takes 8 seconds, but you have 12 microservices involved. How do you find the bottleneck?

Solution:

Distributed tracing tracks requests across multiple services to identify performance issues.

1️⃣ Trace ID Propagation

👉 Generate unique trace ID for each request and propagate it across all services.

Example: Request enters API gateway → gets trace_id=abc123 → all 12 microservices log with same trace_id.

OpenTelemetry Jaeger Zipkin

2️⃣ Span Creation

👉 Each service creates a span representing its portion of work.

Example: Checkout service span: 2.5s, Payment service span: 3.2s, Inventory span: 0.8s.

Detailed timing breakdown

3️⃣ Parent-Child Span Relationships

👉 Link spans to show service call hierarchy.

Example: API span is parent → Checkout span is child → Payment span is grandchild.

Call graph visualization

4️⃣ Span Attributes & Tags

👉 Add metadata to spans for debugging context.

Example: user_id, product_id, http_status_code, error_message attached to spans.

Rich context for debugging

5️⃣ Distributed Context Propagation

👉 Pass trace context through HTTP headers, message queues, and RPC calls.

Example: traceparent header: 00-abc123-def456-01 propagated across services.

W3C Trace Context standard

6️⃣ Sampling Strategy

👉 Trace subset of requests to balance performance and visibility.

Example: Trace 10% of normal requests, 100% of errors → reduces storage cost.

Cost vs visibility trade-off

7️⃣ Waterfall Visualization

👉 Display timeline showing which services took how long.

Example: Jaeger UI shows 8-second checkout: 2s gateway, 3s payment, 2s inventory, 1s notification.

Jaeger UI Zipkin UI

8️⃣ Error Tracking

👉 Automatically capture and link errors to traces.

Example: Payment failure → trace shows exact line of code and all related spans.

Full context for error investigation

💡 Problem 2: Tracing Asynchronous Operations

Question: INTERVIEWER: An order confirmation email is delayed by 2 hours. How do you trace async operations through your message queue?

Solution:

Tracing async operations requires propagating context through message queues.

1️⃣ Trace Context in Message Headers

👉 Include trace_id and span_id in message queue headers.

Example: Kafka message includes {trace_id: abc123, parent_span_id: def456} in headers.

Maintains trace continuity

2️⃣ Producer Span

👉 Create span when publishing message to queue.

Example: Order service publishes to Kafka → creates producer span with timestamp.

Track message production time

3️⃣ Consumer Span

👉 Create child span when consuming message.

Example: Email service consumes message 2 hours later → span shows 2-hour queue delay.

End-to-end visibility

4️⃣ Queue Latency Tracking

👉 Measure time message spent in queue.

Example: Message published at 10:00 AM, consumed at 12:00 PM → 2-hour queue latency visible in trace.

Identify queue bottlenecks

5️⃣ Dead Letter Queue Tracing

👉 Maintain trace context when messages move to DLQ.

Example: Failed message → moved to DLQ → trace shows original request + all retry attempts.

Trace failure scenarios

6️⃣ Fan-Out Pattern Tracing

👉 Track one message spawning multiple child operations.

Example: Order created → triggers 5 parallel tasks (email, inventory, shipping, analytics, notification) → all linked to same trace.

Parallel operation visibility

💡 Problem 3: Instrumenting a Slow Application

Question: INTERVIEWER: Your app is slow — but you don't know why. What do you instrument first?

Solution:

Systematic instrumentation reveals performance bottlenecks quickly.

1️⃣ Application Performance Monitoring (APM)

👉 Install APM agent to get automatic transaction tracing.

Example: New Relic/Datadog agent shows 80% of time spent in database queries.

New Relic · Datadog · Dynatrace

2️⃣ Database Query Logging

👉 Log slow queries and analyze patterns.

Example: Enable slow_query_log → discover 10 queries taking >1 second each.

PostgreSQL slow query log · MySQL slow log

3️⃣ Distributed Tracing

👉 Trace requests across microservices.

Example: Jaeger shows checkout takes 8s: 5s waiting for payment service, 2s in database.

Jaeger · Zipkin · OpenTelemetry

4️⃣ Metrics Collection

👉 Track key metrics: response time, throughput, error rate.

Example: P95 latency jumped from 200ms to 3s starting at 2 PM.

Prometheus · CloudWatch · Grafana

5️⃣ Logging with Correlation IDs

👉 Add request IDs to all logs for request flow tracking.

Example: Follow request_id through 5 microservices to find where it slows down.

Structured logging | correlation IDs

6️⃣ Real User Monitoring (RUM)

👉 Measure actual user experience in browser.

Example: RUM shows 90% of users experience 5s load time (server logs say 500ms).

Google Analytics | SpeedCurve | Browser timing API

7️⃣ Profiling

👉 Profile CPU and memory usage to find hotspots.

Example: Python profiler shows 60% CPU time in image resizing function.

Python cProfile | Java Flight Recorder | perf

💡 Problem 3: Instrumenting a Slow Application

Question: INTERVIEWER: Your app is slow — but you don't know why. What do you instrument first?

Solution:

Systematic instrumentation reveals performance bottlenecks quickly.

1️⃣ Application Performance Monitoring (APM)

👉 Install APM agent to get automatic transaction tracing.

Example: New Relic/Datadog agent shows 80% of time spent in database queries.

New Relic | Datadog | Dynatrace

2️⃣ Database Query Logging

👉 Log slow queries and analyze patterns.

Example: Enable slow_query_log → discover 10 queries taking >1 second each.

PostgreSQL slow query log | MySQL slow log

3️⃣ Distributed Tracing

👉 Trace requests across microservices.

Example: Jaeger shows checkout takes 8s: 5s waiting for payment service, 2s in database.

Jaeger | Zipkin | OpenTelemetry

4️⃣ Metrics Collection

👉 Track key metrics: response time, throughput, error rate.

Example: P95 latency jumped from 200ms to 3s starting at 2 PM.

Prometheus | CloudWatch | Grafana

5️⃣ Logging with Correlation IDs

👉 Add request IDs to all logs for request flow tracking.

Example: Follow request_id through 5 microservices to find where it slows down.

Structured logging | correlation IDs

6️⃣ Real User Monitoring (RUM)

👉 Measure actual user experience in browser.

Example: RUM shows 90% of users experience 5s load time (server logs say 500ms).

Google Analytics | SpeedCurve | Browser timing API

7️⃣ Profiling

👉 Profile CPU and memory usage to find hotspots.

Example: Python profiler shows 60% CPU time in image resizing function.

Python cProfile | Java Flight Recorder | perf

📚 Retry Logic & Exponential Backoff

💡 Problem 1: Handling Transient Failures

Question: INTERVIEWER: Your API calls to payment gateway occasionally fail due to network blips. How do you handle retries?

Solution:

Smart retry logic with exponential backoff recovers from transient failures without overwhelming services.

1️⃣ Exponential Backoff

👉 Increase delay between retries exponentially.

Example: Retry after 1s, then 2s, then 4s, then 8s → gives service time to recover.

Prevents overwhelming recovering service

2️⃣ Jitter / Randomization

👉 Add random delay to prevent thundering herd.

Example: Instead of all clients retrying at exactly 1s, retry between 0.8s-1.2s.

Prevents synchronized retry storms

3️⃣ Maximum Retry Attempts

👉 Limit retries to prevent infinite loops.

Example: Max 3 retries → after that, fail permanently and alert.

Fail fast after reasonable attempts

4️⃣ Idempotency Requirement

👉 Ensure retries don't cause duplicate side effects.

Example: Payment processed with idempotency key → retries don't create duplicate charges.

Prevent duplicate side effects

5️⃣ Retry on Specific Errors Only

👉 Only retry transient errors, not permanent failures.

Example: Retry 503 Service Unavailable, 429 Rate Limit; Don't retry 400 Bad Request, 401 Unauthorized.

Smart error classification

6️⃣ Circuit Breaker Integration

👉 Stop retrying when circuit opens.

Example: After 5 failures, circuit opens → skip retry logic, fail fast for 30 seconds.

Prevent wasted retry attempts

7️⃣ Retry Budget

👉 Limit percentage of requests that can be retries.

Example: Allow max 10% retry traffic → prevents retry storms overwhelming backend.

Protect backend from retry storms

8️⃣ Timeout Configuration

👉 Set shorter timeouts for retry attempts.

Example: First attempt: 5s timeout; Retries: 2s timeout → faster failure detection.

Balance responsiveness and retry attempts

💡 Problem 2: Database Connection Retry Strategy

Question: INTERVIEWER: Your database restarts and all connections drop. How do you handle reconnection in your application?

Solution:

Database connection retry requires careful handling to avoid overwhelming recovering database.

1️⃣ Connection Pool Health Checks

👉 Validate connections before using them.

Example: Before executing query, test connection with SELECT 1 → if fails, get new connection.

Prevent using dead connections

2️⃣ Exponential Backoff for Reconnection

👉 Retry database connection with increasing delays.

Example: DB connection lost → retry after 1s, 2s, 4s, 8s, 16s → max 5 attempts.

Give database time to fully restart

3️⃣ Connection Validation Query

👉 Periodically test idle connections.

Example: Every 30 seconds, run SELECT 1 on idle connections → close if invalid.

Fast health verification

4️⃣ Maximum Lifetime & Idle Timeout

👉 Close long-lived or idle connections automatically.

Example: Max connection lifetime: 1 hour; Idle timeout: 10 minutes → prevents stale connections.

Prevent accumulating stale connections

5️⃣ Graceful Degradation During Outage

👉 Continue serving traffic with reduced capacity when systems are stressed.

Example: High load → disable non-essential features like recommendations → keep checkout working.

Maintain partial functionality

6️⃣ Read Replica Failover

👉 Automatically promote read replica to primary when primary fails.

Example: Primary DB crashes → replica promoted to primary within 60 seconds → minimal downtime.

High availability for reads

Topic 11: Database Indexing & Query Optimization

💡 Problem 31: When to Add an Index

Question: INTERVIEWER: Your query is slow on a users table with 10 million rows. When should you add an index?

Solution:

Indexes speed up reads but slow down writes — know when the tradeoff is worth it.

Here's when to add an index 👇

1️⃣ Frequently Queried Columns

👉 Add index on columns used in WHERE, JOIN, and ORDER BY clauses.

Example: Users table with frequent `WHERE email = ?` queries → index on email column.

B-tree index | PostgreSQL CREATE INDEX

2️⃣ High Selectivity Columns

👉 Index columns with many unique values (high cardinality).

Example: Email or user_id (unique) gets better index performance than gender (low cardinality).

Selectivity = unique_values / total_rows

3️⃣ Composite Indexes for Multi-Column Queries

👉 Create index on multiple columns when queries filter on those columns together.

Example: `WHERE city = ? AND age > ?` → composite index on (city, age).

Leftmost prefix rule | PostgreSQL multi-column indexes

4️⃣ Covering Indexes for SELECT Optimization

👉 Include all columns from SELECT clause in index to avoid table lookup.

Example: `SELECT name, email WHERE user_id = ?` → index on (user_id, name, email).

Index-only scan | INCLUDE clause in PostgreSQL

5️⃣ Avoid Over-Indexing Write-Heavy Tables

👉 Each index adds overhead to INSERT/UPDATE/DELETE operations.

Example: Table with 10 inserts/sec and 1 read/min → indexes hurt more than help.

Write amplification | index maintenance cost

6️⃣ Use Partial Indexes for Subset Queries

👉 Index only rows matching a condition to save space and improve performance.

Example: `WHERE status = 'active'` → partial index on only active rows.

PostgreSQL partial indexes | filtered indexes in SQL Server

💡 Problem 32: Optimizing Slow Queries

Question: INTERVIEWER: A query takes 5 seconds on production. How do you optimize it?

Solution:

Query optimization is like debugging — you need to see what's happening under the hood. Here's how to optimize slow queries 👇

1️⃣ Use EXPLAIN to Analyze Query Plan

👉 Check how database executes the query — look for sequential scans, missing indexes.

Example: `EXPLAIN ANALYZE SELECT * FROM orders WHERE user_id = 123` → shows seq scan → add index.

EXPLAIN | EXPLAIN ANALYZE | execution plan

2️⃣ Rewrite N+1 Queries with JOINs

👉 Replace multiple single-row queries with one query using JOIN.

Example: 100 queries to fetch user for each order → 1 query with `JOIN users ON orders.user_id = users.id`.

Eager loading | LEFT JOIN | INNER JOIN

3️⃣ Add Indexes on Filter/Join Columns

👉 Index columns used in WHERE, JOIN, and ORDER BY clauses.

Example: Slow `WHERE created_at > ?` → add index on created_at.

B-tree index | composite index

4️⃣ Limit Result Set with LIMIT and Pagination

👉 Fetch only necessary rows instead of full table scan.

Example: `SELECT * FROM logs` (1M rows) → `SELECT * FROM logs LIMIT 100 OFFSET 0`.

Cursor-based pagination | keyset pagination

5️⃣ Avoid SELECT * — Fetch Only Needed Columns

👉 Reduce data transfer by selecting specific columns.

Example: `SELECT *` transfers 20 columns → `SELECT id, name, email` transfers 3 columns.

Projection | covering index

6️⃣ Denormalize for Read-Heavy Queries

👉 Duplicate data to avoid expensive JOINs in read-heavy workloads.

Example: Store user name in orders table instead of JOIN with users table on every query.

Denormalization | materialized views

7️⃣ Use Database Query Cache

👉 Cache identical query results for repeated queries.

Example: Same dashboard query runs 100 times/min → cache result for 1 minute.

Query result cache | Redis

📚 Event-Driven Architecture

💡 Problem 33: Choosing the Right Index Type

Question: INTERVIEWER: You have a query searching for text patterns. What index type should you use?

Solution:

Different queries need different index types — B-tree isn't always the answer.

Here's how to choose the right index type 👇

1️⃣ B-Tree Index for Equality and Range Queries

👉 Use for `=`, `<`, `>`, `BETWEEN`, `IN`, `ORDER BY` queries.

Example: `WHERE age > 30 AND age < 50` → B-tree index on age.

PostgreSQL default | MySQL InnoDB

2️⃣ Hash Index for Exact Match Queries

👉 Use for equality comparisons only — faster than B-tree but no range support.

Example: `WHERE user_id = 123` → hash index on user_id.

PostgreSQL HASH index | not range-searchable

3️⃣ Full-Text Index for Text Search

👉 Use for pattern matching and text search queries.

Example: `WHERE description LIKE '%backend%'` → full-text index on description.

PostgreSQL GIN/GiST | MySQL FULLTEXT | Elasticsearch

4️⃣ GiST/GIN Index for JSON and Array Queries

👉 Use for querying inside JSON fields or array contains operations.

Example: `WHERE tags @> ARRAY['python', 'backend']` → GIN index on tags.

PostgreSQL GIN for JSONB | array containment

5️⃣ Spatial Index for Geolocation Queries

👉 Use for location-based queries (nearby places, distance calculations).

Example: `WHERE ST_DWithin(location, point, 10km)` → spatial index on location.

PostGIS R-tree | MongoDB geospatial index

6️⃣ Bitmap Index for Low-Cardinality Columns

👉 Use for columns with few distinct values in data warehouse scenarios.

Example: `WHERE gender = 'M'` → bitmap index on gender (only 2-3 values).

Oracle Bitmap Index | not in PostgreSQL/MySQL

💡 Problem 34: Index Maintenance and Monitoring

Question: INTERVIEWER: Your database performance degrades over time even though you have indexes. Why?

Solution:

Indexes aren't fire-and-forget — they need maintenance and monitoring.

Here's how to maintain indexes 👇

1️⃣ Monitor Index Usage

👉 Track which indexes are actually used vs unused indexes wasting space.

Example: Query `pg_stat_user_indexes` → find indexes with zero scans → drop them.

PostgreSQL pg_stat_user_indexes | MySQL INFORMATION_SCHEMA

2️⃣ Rebuild Fragmented Indexes

👉 Over time, indexes become fragmented and slower — rebuild to defragment.

Example: Like a fragmented hard drive slowing down → defrag improves performance.

PostgreSQL REINDEX | MySQL OPTIMIZE TABLE

3️⃣ Update Table Statistics

👉 Database query planner uses statistics to choose optimal plan — keep them fresh.

Example: After bulk insert of 1M rows → run ANALYZE to update statistics.

PostgreSQL ANALYZE | MySQL ANALYZE TABLE

4️⃣ Vacuum Dead Tuples in PostgreSQL

👉 DELETE/UPDATE leaves dead rows that bloat table and indexes — vacuum to reclaim space.

Example: Like emptying trash bin → reclaim disk space and improve performance.

PostgreSQL VACUUM | autovacuum daemon

5️⃣ Check Index Bloat

👉 Indexes grow larger than necessary due to dead tuples and fragmentation.

Example: 500MB index should be 100MB → indicates bloat → rebuild index.

pg_stat_user_indexes | pgstattuple extension

6️⃣ Monitor Query Plan Changes

👉 Database might stop using your index if statistics are outdated or query changes.

Example: EXPLAIN shows seq scan instead of index scan → update stats or fix query.

EXPLAIN ANALYZE | query plan monitoring

Topic 12: Database Sharding

💡 Problem 35: When to Shard Your Database

Question: INTERVIEWER: Your database has 500GB of data and queries are slowing down. Should you shard?

Solution:

Sharding is powerful but complex — make sure you actually need it first.

Here's when to shard your database 👇

1️⃣ Data Size Exceeds Single Server Capacity

👉 When your data grows beyond what one database server can handle.

Example: 5TB dataset → single PostgreSQL instance can't scale → shard across 10 servers (500GB each).

Horizontal partitioning | distributed database

2️⃣ Write Traffic Exceeds Single Server Throughput

👉 When write operations bottleneck on a single database instance.

Example: 100K writes/sec → single database maxes at 10K writes/sec → shard to distribute writes.

Write scaling | distributed writes

3️⃣ Read Replicas Can't Keep Up

👉 When adding more read replicas doesn't help because primary write bottleneck.

Example: Primary replication lag at 5 seconds even with 10 replicas → shard to reduce primary load.

Replication lag | write bottleneck

4️⃣ Natural Data Boundaries Exist

👉 When your data has logical separation (multi-tenant, geographic regions).

Example: E-commerce platform → shard by region (US shard, EU shard, Asia shard).

`Multi-tenancy` `geographic sharding`

5️⃣ Before Sharding: Try These First

👉 Exhaust simpler options before adding sharding complexity.

Example: Vertical scaling, read replicas, caching, query optimization, table partitioning.

`Avoid premature sharding` `operational complexity`

💡 Problem 36: Zero-Downtime Database Migration

Question: INTERVIEWER: You need to add a NOT NULL column to a 500M-row table — how do you avoid downtime?

Solution:

Large database migrations require careful planning to avoid locking and downtime.

1️⃣ Add Column as Nullable First

👉 Add column without NOT NULL constraint initially.

Example: ALTER TABLE users ADD COLUMN status VARCHAR(20) → instant, no row updates.

`Nullable first` `fast migration`

2️⃣ Backfill in Batches

👉 Update existing rows in small batches.

Example: UPDATE users SET status='active' WHERE id BETWEEN 1 AND 100000 → repeat.

`Batch updates` `gradual backfill`

3️⃣ Add Default Value for New Rows

👉 Application sets value for all new inserts.

Example: App code adds status='active' for new users during backfill.

Application defaults | dual writes

4️⃣ Add NOT NULL Constraint After Backfill

👉 Once all rows populated, add constraint.

Example: ALTER TABLE users MODIFY COLUMN status VARCHAR(20) NOT NULL → fast validation.

Constraint addition | validation

5️⃣ Online Schema Change Tools

👉 Use tools that avoid locking.

Example: gh-ost (GitHub), pt-online-schema-change (Percona) → non-blocking migrations.

gh-ost | pt-online-schema-change

6️⃣ Blue-Green Database Migration

👉 Migrate to new database, switch over when ready.

Example: Replicate to new DB → apply schema → catch up → switch traffic.

Blue-green DB | replication

7️⃣ Rollback Plan

👉 Always have a way to undo the migration.

Example: Keep old column temporarily → if issues, revert code → remove new column later.

Rollback | safety

💡 Problem 36: Zero-Downtime Database Migration

Question: INTERVIEWER: You need to add a NOT NULL column to a 500M-row table — how do you avoid downtime?

Solution:

Large database migrations require careful planning to avoid locking and downtime.

1️⃣ Add Column as Nullable First

👉 Add column without NOT NULL constraint initially.

Example: ALTER TABLE users ADD COLUMN status VARCHAR(20) → instant, no row updates.

Nullable first | fast migration

2️⃣ Backfill in Batches

👉 Update existing rows in small batches.

Example: UPDATE users SET status='active' WHERE id BETWEEN 1 AND 100000 → repeat.

Batch updates | gradual backfill

3️⃣ Add Default Value for New Rows

👉 Application sets value for all new inserts.

Example: App code adds status='active' for new users during backfill.

Application defaults | dual writes

4️⃣ Add NOT NULL Constraint After Backfill

👉 Once all rows populated, add constraint.

Example: ALTER TABLE users MODIFY COLUMN status VARCHAR(20) NOT NULL → fast validation.

Constraint addition | validation

5️⃣ Online Schema Change Tools

👉 Use tools that avoid locking.

Example: gh-ost (GitHub), pt-online-schema-change (Percona) → non-blocking migrations.

gh-ost pt-online-schema-change

6️⃣ Blue-Green Database Migration

👉 Migrate to new database, switch over when ready.

Example: Replicate to new DB → apply schema → catch up → switch traffic.

Blue-green DB replication

7️⃣ Rollback Plan

👉 Always have a way to undo the migration.

Example: Keep old column temporarily → if issues, revert code → remove new column later.

Rollback safety

📚 Microservices Architecture

💡 Problem 36: Sharding Strategies

Question: INTERVIEWER: You decided to shard. How do you choose which shard to send data to?

Solution:

The sharding key determines data distribution — choosing wrong can cause hotspots.

Here are common sharding strategies 👇

1️⃣ Hash-Based Sharding

👉 Hash the sharding key (user_id, order_id) to determine shard.

Example: hash(user_id) % 10 → user 12345 always goes to shard 5.

Consistent hashing | even distribution | Redis Cluster

2️⃣ Range-Based Sharding

👉 Split data by value ranges of the sharding key.

Example: Users A-M → Shard 1, Users N-Z → Shard 2.

Sorted data | range queries | MongoDB range sharding

3️⃣ Geographic Sharding

👉 Shard by location to reduce latency and comply with data residency laws.

Example: US users → US shard, EU users → EU shard (GDPR compliance).

Multi-region | data sovereignty | latency optimization

4️⃣ Directory-Based Sharding

👉 Maintain lookup table mapping each key to its shard.

Example: user_id 123 → Shard 2, user_id 456 → Shard 5.

Flexible | lookup overhead | shard rebalancing

5️⃣ Entity Group Sharding

👉 Co-locate related data on same shard for transaction support.

Example: User and their orders on same shard → local transactions instead of distributed.

Data locality | transaction support | Google Spanner

6️⃣ Composite Sharding Key

👉 Use multiple columns for sharding to improve distribution.

Example: Shard by (tenant_id, created_date) → evenly distribute multi-tenant data.

Multi-tenancy | time-series data

💡 Problem 37: Handling Shard Hotspots and Rebalancing

Question: INTERVIEWER: One of your shards is getting 80% of the traffic. How do you fix it?

Solution:

Uneven shard distribution causes hotspots — you need to rebalance or rethink sharding strategy.

Here's how to handle shard hotspots 👇

1️⃣ Identify Hotspot Cause

👉 Monitor query patterns to find why one shard is overloaded.

Example: Celebrity user on Shard 3 → 1M followers querying their data → hotspot.

Query analysis | shard metrics | uneven distribution

2️⃣ Use Consistent Hashing to Minimize Rebalancing

👉 When adding/removing shards, only rebalance affected keys instead of all data.

Example: Adding Shard 11 with consistent hashing → only ~9% of data moves, not 100%.

Consistent hashing ring | minimal data movement

3️⃣ Reshard Hot Keys to Separate Shard

👉 Move high-traffic keys to dedicated shard to isolate load.

Example: Celebrity user causing hotspot → move to dedicated shard.

Resharding | data migration | hotspot isolation

4️⃣ Use Finer-Grained Sharding

👉 Increase number of shards to distribute load more evenly.

Example: 4 shards with uneven load → 16 shards with better distribution.

More shards | rebalancing complexity

5️⃣ Add Secondary Index Shard for Read-Heavy Keys

👉 Cache or replicate hot data separately for read queries.

Example: Popular product data → replicate to all shards or cache layer.

Read replicas | caching | denormalization

6️⃣ Implement Virtual Shards

👉 Create more logical shards than physical servers, then map multiple virtual shards to one physical shard.

Example: 1000 virtual shards mapped to 10 physical shards → easier rebalancing.

Virtual shards | flexible mapping | Vitess

Topic 13: Read Replicas & Replication

💡 Problem 38: Setting Up Read Replicas

Question: INTERVIEWER: Your application has heavy read traffic. How do you scale reads without affecting writes?

Solution:

Read replicas let you scale reads horizontally without overloading the primary database.

Here's how to set up read replicas 👇

1️⃣ Asynchronous Replication to Replicas

👉 Primary writes data, then asynchronously replicates to read replicas.

Example: Like photocopying documents — original written first, copies made after.

PostgreSQL streaming replication | MySQL replication

2️⃣ Route Reads to Replicas, Writes to Primary

👉 Application logic or database proxy routes read queries to replicas.

Example: 90% read traffic → 10 replicas handle reads, primary handles writes only.

Read/write splitting | ProxySQL | AWS RDS read replicas

3️⃣ Monitor Replication Lag

👉 Track delay between primary write and replica availability.

Example: Primary writes at 10:00:00, replica shows data at 10:00:05 → 5-second lag.

Replication lag monitoring | pg_stat_replication

4️⃣ Use Multiple Replicas for Geographic Distribution

👉 Place replicas in different regions for lower latency reads.

Example: US primary → EU replica → EU users read from EU replica (lower latency).

Multi-region | geographic distribution

5️⃣ Handle Eventual Consistency

👉 Accept that replicas may show stale data due to replication lag.

Example: User updates profile → immediately reads from replica → sees old data (read-after-write inconsistency).

Eventual consistency | read-your-writes problem

💡 Problem 39: Handling Replication Lag

Question: INTERVIEWER: A user updates their profile but doesn't see the change. What's happening?

Solution:

Replication lag causes read-after-write inconsistency — you need strategies to handle it.

Here's how to handle replication lag 👇

1️⃣ Read Your Own Writes from Primary

👉 Route reads to primary for data the user just modified.

Example: User updates profile → next profile read goes to primary, not replica.

`Session affinity` `sticky routing`

2️⃣ Use Synchronous Replication for Critical Data

👉 Primary waits for at least one replica to acknowledge write before returning success.

Example: Financial transaction → wait for replica confirmation → guarantees data available on replica.

`Synchronous replication` `trade latency for consistency`

3️⃣ Check Replication Position Before Read

👉 Query replica only if it has caught up to a specific replication position.

Example: Write returns position 1000 → read waits until replica reaches position 1000.

`LSN tracking` `replication position`

4️⃣ Use Monotonic Reads with Session Token

👉 Ensure a user always reads from replica at least as fresh as their last read.

Example: Session token tracks last seen timestamp → only query replicas ahead of that timestamp.

`Monotonic reads` `session consistency`

5️⃣ Add Cache Layer for Immediate Reads

👉 Cache recent writes so reads return cached data instead of stale replica data.

Example: Profile update → write to database and Redis → read from Redis (no lag).

Write-through cache | Redis

6️⃣ Display Staleness to User

👉 Show users that data might be slightly out of date.

Example: "Data as of 5 seconds ago" message on dashboard.

User expectations | eventual consistency UX

📚 API Gateway

💡 Problem 40: Promoting Replica to Primary (Failover)

Question: INTERVIEWER: Your primary database crashes. How do you promote a replica to become the new primary?

Solution:

Database failover is critical for high availability — automate it or risk extended downtime.

Here's how to promote a replica to primary 👇

1️⃣ Detect Primary Failure

👉 Use health checks and heartbeat monitoring to detect when primary is down.

Example: Primary doesn't respond to health check for 10 seconds → trigger failover.

Health checks | Patroni | AWS RDS automated failover

2️⃣ Stop Writes to Failed Primary

👉 Ensure no new writes go to the failed primary to prevent data conflicts.

Example: Update DNS or load balancer to stop routing traffic to old primary.

Fencing STONITH

3️⃣ Select Replica with Most Recent Data

👉 Choose replica with lowest replication lag as new primary.

Example: Replica A at position 1000, Replica B at position 995 → promote Replica A.

LSN comparison replication position

4️⃣ Promote Replica to Primary

👉 Reconfigure selected replica to accept writes and become new primary.

Example: `pg_promote()` in PostgreSQL → replica starts accepting writes.

Promote command configuration change

5️⃣ Reconfigure Other Replicas

👉 Point remaining replicas to replicate from the new primary.

Example: Replica B and C now replicate from Replica A (promoted to primary).

Replication topology change

6️⃣ Update Application Connection Strings

👉 Change application configuration or DNS to point to new primary.

Example: Update database endpoint from old-primary.com to new-primary.com.

DNS update connection failover service discovery

7️⃣ Use Automated Failover Tools

👉 Tools like Patroni, Orchestrator, or cloud-managed services automate entire failover process.

Example: AWS RDS Multi-AZ automatically fails over within 1-2 minutes.

Patroni etcd AWS RDS Multi-AZ

Topic 14: Connection Pooling

💡 Problem 41: Why Use Connection Pooling

Question: INTERVIEWER: Your API creates a new database connection for every request. Why is this slow?

Solution:

Creating database connections is expensive — connection pooling reuses connections.

Here's why you need connection pooling 👇

1️⃣ Connection Creation is Expensive

👉 Each new connection requires TCP handshake, authentication, and resource allocation.

Example: Creating connection takes 50ms → 1000 req/sec = 50,000ms wasted on connections.

TCP overhead authentication cost

2️⃣ Connection Pool Reuses Connections

👉 Pre-create connections and reuse them across requests.

Example: Like rental cars — instead of buying a car for each trip, rent from a shared pool.

Connection reuse PgBouncer HikariCP

3️⃣ Limit Max Connections to Database

👉 Database has maximum connection limit — pool prevents exhausting it.

Example: PostgreSQL max_connections = 100 → connection pool limits app to 100 connections.

Max connections resource limits

4️⃣ Reduce Database Connection Overhead

👉 Fewer connections means less memory and CPU usage on database server.

Example: 1000 idle connections consume GBs of memory → 20 pooled connections consume MBs.

Memory usage connection overhead

5️⃣ Improve Application Response Time

👉 Reusing connections removes connection setup latency from request path.

Example: 50ms connection setup → with pooling, 0ms connection setup.

Latency reduction connection reuse

💡 Problem 42: Configuring Connection Pool Size

Question: INTERVIEWER: How many connections should you have in your connection pool?

Solution:

Too few connections cause queuing, too many overwhelm the database — find the sweet spot.

Here's how to configure connection pool size 👇

1️⃣ Start with Formula: connections = ((core_count * 2) + effective_spindle_count)

👉 For SSDs, effective_spindle_count ≈ 1. For 4-core DB server → pool size ≈ 9.

Example: Like a restaurant — too few servers (connections) = long wait, too many = chaos.

HikariCP recommendation empirical formula

2️⃣ Consider Database Max Connections

👉 Total connections across all app instances must stay below database limit.

Example: PostgreSQL max_connections = 100 → 10 app servers → 10 connections per server.

Max connections constraint | distributed apps

3️⃣ Monitor Connection Usage

👉 Track active vs idle connections to see if pool is too small or too large.

Example: Pool size 20, always 19-20 active → increase pool size.

Connection metrics | pool saturation

4️⃣ Use Connection Pool Per Service

👉 Each microservice or app instance gets its own connection pool.

Example: 5 app servers with 20 connections each = 100 total connections to database.

Distributed pools | connection accounting

5️⃣ Set Connection Timeouts

👉 Configure how long app waits for available connection before timing out.

Example: Connection timeout = 10 seconds → if pool exhausted, request fails after 10s.

Connection timeout | pool exhaustion handling

6️⃣ Use PgBouncer for Transaction-Level Pooling

👉 External connection pooler that multiplexes many app connections to fewer database connections.

Example: 1000 app connections → PgBouncer → 20 database connections.

PgBouncer | transaction pooling mode | connection multiplexing

📚 Service Mesh

💡 Problem 43: Connection Pooling Issues

Question: INTERVIEWER: Your connection pool shows all connections idle, but requests are timing out. What's wrong?

Solution:

Connection pools have gotchas — idle connections might be dead, or pool might be misconfigured.

Here's how to debug connection pooling issues 👇

1️⃣ Dead Connections in Pool

👉 Database closed idle connections but pool thinks they're still valid.

Example: Database closes connections after 5 minutes idle → app tries to use dead connection → query fails.

`Connection validation` `testOnBorrow`

2️⃣ Enable Connection Validation

👉 Test connections before using them from pool.

Example: Run `SELECT 1` before giving connection to app → if fails, discard and create new one.

`Connection health check` `HikariCP validationQuery`

3️⃣ Set Max Connection Lifetime

👉 Close and recreate connections periodically to avoid stale connections.

Example: Max lifetime 30 minutes → connection used for 30 min → close and create new one.

`Connection refresh` `maxLifetime`

4️⃣ Connection Leak Detection

👉 Detect when app doesn't return connections to pool after use.

Example: App queries database but forgets to close connection → pool exhausted → timeouts.

Leak detection | connection leak timeout

5️⃣ Firewall Killing Idle Connections

👉 Network firewall drops TCP connections after idle timeout.

Example: AWS NAT gateway drops connections idle >350 seconds → configure keepalive < 350s.

TCP keepalive | firewall idle timeout

6️⃣ Pool Size Too Small for Workload

👉 More concurrent requests than available connections.

Example: 100 concurrent requests, pool size 10 → 90 requests wait for connection → timeouts.

Pool saturation | increase pool size

Topic 15: ACID Transactions

💡 Problem 44: Explaining ACID Properties

Question: INTERVIEWER: What does ACID mean in databases?

Solution:

ACID guarantees ensure database transactions are reliable and consistent.

Here's what ACID means 👇

1️⃣ Atomicity

👉 Transaction either fully completes or fully fails — no partial updates.

Example: Bank transfer — debit $100 from A and credit $100 to B → both succeed or both fail.

All-or-nothing | rollback on failure

2️⃣ Consistency

👉 Transaction takes database from one valid state to another valid state.

Example: Constraint "balance >= 0" → transaction that would make balance negative is rejected.

Database constraints | invariants

3️⃣ Isolation

👉 Concurrent transactions don't interfere with each other.

Example: Two users buying last item — both see item available, but only one succeeds.

Concurrency control | locks | MVCC

4️⃣ Durability

👉 Committed transaction changes are permanent, even after crash.

Example: Transaction commits at 10:00:00, server crashes at 10:00:01 → data still saved.

Write-ahead log | fsync | persistent storage

📚 Circuit Breaker Pattern

💡 Problem 45: Transaction Isolation Levels

Question: INTERVIEWER: Two transactions read and update the same row. What isolation level do you use?

Solution:

Isolation levels trade off consistency for performance — choose based on your requirements.

Here are the transaction isolation levels 👇

1️⃣ Read Uncommitted (Lowest Isolation)

👉 Transaction can read uncommitted changes from other transactions.

Example: Transaction A updates row, Transaction B reads it before A commits → dirty read.

Dirty reads possible | rarely used

2️⃣ Read Committed (Default in PostgreSQL/Oracle)

👉 Transaction only reads committed data from other transactions.

Example: Transaction A updates row, Transaction B can only read value after A commits.

No dirty reads | prevents reading uncommitted data

3️⃣ Repeatable Read

👉 Transaction sees consistent snapshot of data throughout — no non-repeatable reads.

Example: Transaction A reads row twice → sees same value both times even if another transaction updates it.

No non-repeatable reads | snapshot isolation

4️⃣ Serializable (Highest Isolation)

👉 Transactions execute as if they ran serially, one after another.

Example: Like a single-threaded program — no concurrency anomalies possible.

Full isolation | performance cost | deadlock risk

5️⃣ Trade-off: Isolation vs Performance

👉 Higher isolation = more locks/overhead = lower throughput.

Example: Serializable prevents all anomalies but can be 10x slower than Read Committed.

Lock contention | MVCC | choose based on requirements

💡 Problem 46: Handling Distributed Transactions

Question: INTERVIEWER: You need to update two different databases atomically. How do you ensure both succeed or both fail?

Solution:

Distributed transactions are hard — you need coordination protocols or eventual consistency.

Here's how to handle distributed transactions 👇

1️⃣ Two-Phase Commit (2PC)

👉 Coordinator asks all participants to prepare, then commits if all agree.

Example: Coordinator asks DB1 and DB2 "ready to commit?" → both say yes → coordinator says "commit now".

XA transactions | blocking protocol | coordinator failure risk

---

2️⃣ Saga Pattern with Compensating Transactions

👉 Break distributed transaction into local transactions with compensation logic for failures.

Example: Order service creates order → payment fails → order service cancels order (compensating action).

Choreography or orchestration | eventual consistency

3️⃣ Use Message Queue for Reliable Delivery

👉 Write to database and publish message atomically using outbox pattern.

Example: Insert order in DB + insert event in outbox table → background job publishes event.

Outbox pattern | transactional outbox | Kafka

4️⃣ Avoid Distributed Transactions with Denormalization

👉 Duplicate data across services to avoid cross-service transactions.

Example: Store user email in orders table instead of JOIN with users service.

Denormalization | data duplication | eventual consistency

5️⃣ Use Idempotency for Retry Safety

👉 Design operations to be safely retried without side effects.

Example: Payment service receives same payment request twice → charges only once.

Idempotency keys | retry safety

6️⃣ Google Spanner / CockroachDB for True Distributed ACID

👉 Use databases that provide ACID guarantees across multiple nodes.

Example: Like a magical database that coordinates transactions globally — but complex and expensive.

TrueTime | distributed consensus | Spanner | CockroachDB

💡 Problem 47: Optimistic vs Pessimistic Locking

Question: INTERVIEWER: Two users try to book the last hotel room at the same time — how do you handle it?

Solution:

Locking strategies balance data consistency with system performance.

1️⃣ Pessimistic Locking

👉 Lock row immediately, preventing concurrent access.

*Example: SELECT * FROM rooms WHERE id=123 FOR UPDATE → blocks other transactions.*

Row locking | FOR UPDATE

2️⃣ Pessimistic Lock Downsides

👉 Blocks other users, reduces concurrency, risk of deadlocks.

Example: 100 users trying to book → 99 wait → slow user experience.

Low concurrency | deadlocks

3️⃣ Optimistic Locking

👉 Don't lock, detect conflicts at commit time.

Example: Add version column → UPDATE WHERE version=old_version → fails if changed.

Version column | conflict detection

4️⃣ Optimistic Lock Failure Handling

👉 Retry with updated data when conflict detected.

Example: Version mismatch → reload data → retry update → or show error to user.

Retry logic | user notification

5️⃣ When to Use Pessimistic

👉 High contention, critical consistency needs.

Example: Financial transactions, inventory with very limited stock.

High contention | strict consistency

6️⃣ When to Use Optimistic

👉 Low contention, performance-critical reads.

Example: Social media likes, view counts → rare conflicts → optimistic faster.

Low contention | performance

7️⃣ Hybrid Approach

👉 Use optimistic by default, pessimistic for critical operations.

Example: Product catalog updates → optimistic; Final checkout → pessimistic.

Hybrid strategy | balanced approach

💡 Problem 47: Optimistic vs Pessimistic Locking

Question: INTERVIEWER: Two users try to book the last hotel room at the same time — how do you handle it?

Solution:

Locking strategies balance data consistency with system performance.

1️⃣ Pessimistic Locking

👉 Lock row immediately, preventing concurrent access.

*Example: SELECT * FROM rooms WHERE id=123 FOR UPDATE → blocks other transactions.*

Row locking | FOR UPDATE

2️⃣ Pessimistic Lock Downsides

👉 Blocks other users, reduces concurrency, risk of deadlocks.

Example: 100 users trying to book → 99 wait → slow user experience.

Low concurrency | deadlocks

3️⃣ Optimistic Locking

👉 Don't lock, detect conflicts at commit time.

Example: Add version column → UPDATE WHERE version=old_version → fails if changed.

Version column | conflict detection

4️⃣ Optimistic Lock Failure Handling

👉 Retry with updated data when conflict detected.

Example: Version mismatch → reload data → retry update → or show error to user.

Retry logic | user notification

5️⃣ When to Use Pessimistic

👉 High contention, critical consistency needs.

Example: Financial transactions, inventory with very limited stock.

High contention | strict consistency

6️⃣ When to Use Optimistic

👉 Low contention, performance-critical reads.

Example: Social media likes, view counts → rare conflicts → optimistic faster.

Low contention | performance

7️⃣ Hybrid Approach

👉 Use optimistic by default, pessimistic for critical operations.

Example: Product catalog updates → optimistic; Final checkout → pessimistic.

Hybrid strategy | balanced approach

Topic 16: NoSQL Data Modeling

📚 Rate Limiting

💡 Problem 47: When to Use NoSQL vs SQL

Question: INTERVIEWER: Your app needs to store user profiles with flexible attributes. Should you use SQL or NoSQL?

Solution:

SQL and NoSQL excel at different use cases — choose based on data structure and access patterns.

Here's when to use NoSQL vs SQL 👇

1️⃣ Use SQL for Structured Data with Relationships

👉 Data has well-defined schema and relationships between entities.

Example: E-commerce orders, users, products with foreign keys and JOINs.

PostgreSQL | MySQL | ACID transactions | referential integrity

---

2️⃣ Use NoSQL for Flexible Schema

👉 Each document/row can have different fields without schema changes.

Example: User profiles where some users have 5 fields, others have 20 different fields.

MongoDB DynamoDB schema flexibility

3️⃣ Use NoSQL for High Write Throughput

👉 Distributed NoSQL databases scale writes horizontally better than SQL.

Example: Logging system with 1M writes/sec → Cassandra distributes across 100 nodes.

Cassandra DynamoDB eventual consistency

4️⃣ Use SQL for Complex Queries and Aggregations

👉 Need JOINs, GROUP BY, complex filtering, and ad-hoc queries.

Example: Analytics dashboard querying across multiple tables with aggregations.

SQL JOINs GROUP BY HAVING complex WHERE clauses

5️⃣ Use NoSQL for Key-Value or Document Access Patterns

👉 Primarily access data by primary key without complex joins.

Example: User session storage — fetch session by session_id.

Redis DynamoDB MongoDB

6️⃣ Use SQL for Strong Consistency Requirements

👉 Need ACID guarantees and immediate consistency.

Example: Banking system where balance must be accurate immediately.

PostgreSQL MySQL ACID

7️⃣ Use NoSQL for Horizontal Scalability

👉 Data grows beyond single server capacity and needs distributed architecture.

Example: Social media platform with billions of posts across thousands of servers.

Cassandra MongoDB sharding DynamoDB

💡 Problem 48: Modeling Data in Document Databases (MongoDB)

Question: INTERVIEWER: How do you model a blog with users, posts, and comments in MongoDB?

Solution:

Document databases let you embed or reference related data — choose based on access patterns.

Here's how to model data in document databases 👇

1️⃣ Embed Related Data in Same Document

👉 Store related data together when accessed together.

Example: Blog post with comments → embed comments array inside post document.

Atomic updates | single query retrieval

2️⃣ Reference with Document IDs

👉 Store reference to another document's ID when data is large or accessed separately.

Example: Post has author_id → separate users collection → JOIN in application code.

Normalization | avoid document size limits

3️⃣ Embed for One-to-Few Relationships

👉 Embed when child documents are small and limited in number.

Example: User has 3 addresses → embed addresses array in user document.

Simplicity | atomic updates

4️⃣ Reference for One-to-Many Relationships

👉 Use references when child documents are numerous or accessed independently.

Example: Author has 1000 blog posts → posts collection with author_id field.

Avoid unbounded arrays | separate access patterns

5️⃣ Denormalize for Read Performance

👉 Duplicate data to avoid lookups when querying.

Example: Store author name in post document even though it's in users collection.

Faster reads | data duplication | eventual consistency

6️⃣ Use Two-Way Referencing for Many-to-Many

👉 Store references in both documents for efficient bidirectional queries.

Example: Students and courses → student has course_ids, course has student_ids.

Query from either direction | data duplication

💡 Problem 49: Modeling Data in Wide-Column Stores (Cassandra)

Question: INTERVIEWER: How do you design a time-series data model in Cassandra for sensor readings?

Solution:

Wide-column stores require modeling based on query patterns — denormalize and duplicate data.

Here's how to model data in wide-column stores 👇

1️⃣ Design Tables Based on Queries, Not Entities

👉 Create separate table for each query pattern, duplicating data across tables.

Example: Query by sensor_id → table with sensor_id as partition key. Query by location → different table with location as partition key.

Query-driven design | data duplication

2️⃣ Choose Partition Key for Even Distribution

👉 Partition key determines which node stores data — avoid hotspots.

Example: sensor_id as partition key distributes 1000 sensors across cluster. Using country as partition key creates hotspots.

Even distribution | avoid hotspots

3️⃣ Use Clustering Key for Sorting Within Partition

👉 Clustering key determines sort order within partition.

Example: Partition key = sensor_id, clustering key = timestamp → all readings for sensor sorted by time.

Range queries | efficient time-series queries

4️⃣ Keep Partition Size Manageable

👉 Avoid unbounded partitions that grow infinitely.

Example: Partition by (sensor_id, date) instead of just sensor_id → creates new partition each day.

Partition size limits | time bucketing

5️⃣ Denormalize for Read Efficiency

👉 Duplicate data to serve different query patterns without JOINs.

Example: Sensor readings table AND sensor readings by location table — same data, different partition keys.

No JOINs | write amplification

6️⃣ Use Materialized Views for Multiple Query Patterns

👉 Cassandra automatically maintains additional tables for different queries.

Example: Base table partitioned by sensor_id → materialized view partitioned by location.

Automatic denormalization | Cassandra feature

Topic 17: Eventually Consistent Systems

📚 Authentication & Authorization

💡 Problem 50: Understanding Eventual Consistency

Question: INTERVIEWER: Your distributed system shows different data on different servers. Is this a bug?

Solution:

Eventual consistency is a trade-off — accept temporary inconsistency for availability and performance.

Here's how eventual consistency works 👇

1️⃣ What is Eventual Consistency?

👉 Replicas may temporarily show different data, but will eventually converge.

Example: Update post on Server A → Server B shows old data for 2 seconds → eventually both show same data.

CAP theorem | AP systems

2️⃣ Why Accept Eventual Consistency?

👉 Enables high availability and low latency by not waiting for all replicas to update.

Example: Amazon cart allows adding items even if other data centers are unreachable.

Availability over consistency | partition tolerance

3️⃣ Read-After-Write Inconsistency

👉 User writes data but immediately reads old data from replica.

Example: User posts comment → refresh page → doesn't see comment (replica lag).

Replication lag | user experience issue

4️⃣ Conflict Resolution with Last-Write-Wins

👉 When conflicting updates occur, use timestamp to decide winner.

Example: User edits profile on two devices → both writes succeed → later timestamp wins.

LWW | vector clocks | Cassandra

5️⃣ Use Cases for Eventual Consistency

👉 Non-critical data where temporary inconsistency is acceptable.

Example: Social media likes counter, view counts, caching, DNS.

Best-effort consistency | high availability

6️⃣ When NOT to Use Eventual Consistency

👉 Financial transactions, inventory management, or any data requiring immediate accuracy.

Example: Bank balance must be accurate immediately — can't accept stale data.

Strong consistency required | ACID transactions

💡 Problem 51: Handling Conflicts in Eventually Consistent Systems

Question: INTERVIEWER: Two users edit the same document at the same time in different data centers. How do you resolve the conflict?

Solution:

Conflicts are inevitable in eventually consistent systems — you need resolution strategies.

Here's how to handle conflicts 👇

1️⃣ Last-Write-Wins (LWW)

👉 Use timestamp to determine which update wins.

Example: User A updates at 10:00:00, User B updates at 10:00:05 → B's update wins.

Simple but data loss | Cassandra | DynamoDB

2️⃣ Version Vectors / Vector Clocks

👉 Track causality of updates to detect concurrent modifications.

Example: [A:1, B:0] and [A:0, B:1] → both concurrent → conflict detected.

Riak | Cassandra | causality tracking

3️⃣ Merge Conflicts Automatically with CRDTs

👉 Use Conflict-free Replicated Data Types that merge updates deterministically.

Example: Two users add items to shopping cart → both additions preserved in merged cart.

CRDTs | Riak | Redis

4️⃣ Application-Level Conflict Resolution

👉 Store all conflicting versions and let application decide which to keep.

Example: Google Docs tracks all changes and merges text edits operationally.

Custom merge logic | operational transformation

5️⃣ Prevent Conflicts with Distributed Locks

👉 Use locking to ensure only one writer at a time.

Example: User acquires lock on document → edits → releases lock → other user can edit.

Redlock | ZooKeeper | trade availability for consistency

6️⃣ Use Strong Consistency for Critical Operations

👉 Switch to synchronous replication for operations that can't have conflicts.

Example: Decrement inventory with strong consistency → prevent overselling.

Quorum writes synchronous replication

💡 Problem 52: Implementing Distributed Locks

Question: INTERVIEWER: Two servers try to process the same job simultaneously — how do you prevent that?

Solution:

Distributed locks coordinate access to shared resources across multiple processes.

1️⃣ Redis Distributed Lock (Redlock)

👉 Use Redis SET NX EX for simple distributed locking.

Example: SET lock:job123 server1 NX EX 30 → only one server acquires lock.

Redlock Redis locks

2️⃣ Lock Expiration

👉 Auto-release locks after timeout to prevent deadlocks.

Example: Lock expires after 30 seconds → if server crashes, lock auto-released.

TTL auto-release

3️⃣ Lock Renewal (Heartbeat)

👉 Extend lock if job still running.

Example: Job takes 60s → renew lock every 20s → prevents premature expiration.

Lock renewal heartbeat

4️⃣ Zookeeper Locks

👉 Use Zookeeper ephemeral nodes for locks.

Example: Create /locks/job123 → if client dies, node auto-deleted → lock released.

Zookeeper ephemeral nodes

5️⃣ Database-Based Locks

👉 Use database row with unique constraint for lock.

Example: INSERT INTO locks (resource, owner) → unique constraint prevents duplicate.

Database locks row locking

6️⃣ Handling Split-Brain

👉 Prevent two nodes thinking they both hold lock.

Example: Use fencing tokens → each lock gets incrementing ID → reject stale locks.

Fencing tokens split-brain

7️⃣ When to Avoid Locks

👉 Consider lock-free algorithms or idempotency instead.

Example: Process job idempotently → duplicate processing is safe → no lock needed.

Lock-free idempotency

💡 Problem 52: Implementing Distributed Locks

Question: INTERVIEWER: Two servers try to process the same job simultaneously — how do you prevent that?

Solution:

Distributed locks coordinate access to shared resources across multiple processes.

1️⃣ Redis Distributed Lock (Redlock)

👉 Use Redis SET NX EX for simple distributed locking.

Example: SET lock:job123 server1 NX EX 30 → only one server acquires lock.

Redlock Redis locks

2️⃣ Lock Expiration

👉 Auto-release locks after timeout to prevent deadlocks.

Example: Lock expires after 30 seconds → if server crashes, lock auto-released.

TTL auto-release

3️⃣ Lock Renewal (Heartbeat)

👉 Extend lock if job still running.

Example: Job takes 60s → renew lock every 20s → prevents premature expiration.

Lock renewal heartbeat

4️⃣ Zookeeper Locks

👉 Use Zookeeper ephemeral nodes for locks.

Example: Create /locks/job123 → if client dies, node auto-deleted → lock released.

Zookeeper ephemeral nodes

5️⃣ Database-Based Locks

👉 Use database row with unique constraint for lock.

Example: INSERT INTO locks (resource, owner) → unique constraint prevents duplicate.

Database locks row locking

6️⃣ Handling Split-Brain

👉 Prevent two nodes thinking they both hold lock.

Example: Use fencing tokens → each lock gets incrementing ID → reject stale locks.

Fencing tokens split-brain

7️⃣ When to Avoid Locks

👉 Consider lock-free algorithms or idempotency instead.

Example: Process job idempotently → duplicate processing is safe → no lock needed.

Lock-free idempotency

Topic 18: N+1 Query Problem

💡 Problem 52: Identifying N+1 Query Problem

Question: INTERVIEWER: Your API endpoint is slow and making 1000 database queries. What's wrong?

Solution:

N+1 problem happens when you query in a loop — one query for list + N queries for details.

Here's how to identify N+1 queries 👇

1️⃣ What is N+1 Query Problem?

👉 Fetch N items in one query, then loop through items making N additional queries.

Example: Fetch 100 users → loop through users → fetch posts for each user → 101 queries total.

Loop queries performance killer

2️⃣ Signs of N+1 Problem

👉 Number of queries grows with number of records, slow response times.

Example: API takes 10ms with 1 user, 1000ms with 100 users → linear growth = N+1 issue.

Query count scales with data | database CPU spike

3️⃣ Enable Query Logging to Detect

👉 Log all SQL queries to see if queries are running in a loop.

Example: Enable slow query log → see same query pattern repeated 100 times with different IDs.

PostgreSQL log_statement | MySQL slow query log

4️⃣ Use Database Query Profiler

👉 Monitor query patterns in production to identify repeated queries.

Example: APM tool shows `SELECT * FROM posts WHERE user_id = ?` executed 100 times.

New Relic | Datadog | query monitoring

5️⃣ Check ORM Query Behavior

👉 ORMs can hide N+1 queries behind lazy loading.

Example: Django ORM `users = User.objects.all()` then `user.posts.all()` in template → N+1 queries.

Lazy loading | ORM gotcha

💡 Problem 3: Lazy vs Eager Loading Trade-offs

Question: INTERVIEWER: When should you use lazy loading vs eager loading for related data?

Solution:

Loading strategies balance memory usage with query efficiency.

1️⃣ Lazy Loading

👉 Load related data only when accessed.

Example: Fetch user → access user.posts → separate query to fetch posts.

On-demand loading | deferred queries

2️⃣ Eager Loading

👉 Load related data upfront with JOIN.

*Example: SELECT * FROM users JOIN posts → all data in one query.*

Upfront loading | JOIN queries

3️⃣ Lazy Loading Benefits

👉 Don't load unnecessary data, saves memory and query time.

Example: User profile page doesn't need posts → lazy loading avoids fetching them.

Memory efficient | selective loading

4️⃣ Eager Loading Benefits

👉 Avoid N+1 queries when you know you'll need related data.

Example: User list with post counts → eager load to avoid 100 separate queries.

Prevents N+1 | fewer queries

5️⃣ N+1 Query Problem

👉 Lazy loading can cause N+1 queries if not careful.

Example: Loop through 100 users → access user.posts each time → 101 queries total.

Performance pitfall | query explosion

6️⃣ Selective Eager Loading

👉 Eager load only specific relations you need.

Example: User.findAll({ include: ['posts'] }) → loads posts, not comments.

Selective includes | controlled loading

7️⃣ Pagination Considerations

👉 Eager loading with pagination can be complex.

Example: Paginate users with eager-loaded posts → need careful query design.

Pagination complexity | query optimization

💡 Problem 53: Solving N+1 Query Problem

Question: INTERVIEWER: You identified N+1 queries fetching users and their posts. How do you fix it?

Solution:

Fix N+1 by fetching all related data in fewer queries using JOINs or batch loading.

Here's how to solve N+1 queries 👇

1️⃣ Use JOIN to Fetch Related Data

👉 Fetch parent and child data in single query with JOIN.

Example: `SELECT users.*, posts.* FROM users LEFT JOIN posts ON users.id = posts.user_id` → 1 query instead of N+1.

SQL JOIN | eager loading

2️⃣ Use ORM Eager Loading

👉 Tell ORM to prefetch related data upfront.

Example: Django `User.objects.prefetch_related('posts')` → 2 queries (users + posts) instead of N+1.

prefetch_related | select_related in Django/Rails

3️⃣ Batch Load with IN Clause

👉 Fetch all related records in one query using IN clause.

Example: Fetch 100 users → collect user IDs → `SELECT * FROM posts WHERE user_id IN (1,2,3...100)` → 2 queries total.

Batch loading | dataloader pattern

4️⃣ Use DataLoader Library

👉 Automatically batch and cache database requests within single request.

Example: GraphQL dataloader batches all user queries → fetches in single query with IN clause.

Facebook DataLoader | automatic batching

5️⃣ Cache Related Data

👉 Cache frequently accessed related data to avoid queries entirely.

Example: Cache user's posts in Redis → fetch from cache instead of database.

Redis | Memcached | query caching

6️⃣ Denormalize Data

👉 Duplicate data to avoid JOINs entirely.

Example: Store author name in posts table → no need to query users table.

Denormalization | data duplication

📚 SQL vs NoSQL Trade-offs

Topic 19: SQL vs NoSQL Trade-offs

📚 OAuth 2.0 & OpenID Connect

💡 Problem 54: Migrating from SQL to NoSQL

Question: INTERVIEWER: Your PostgreSQL database is hitting scaling limits. Should you migrate to NoSQL?

Solution:

Migration is costly and risky — make sure NoSQL actually solves your problem.

Here's how to decide on migrating to NoSQL 👇

1️⃣ Identify Why SQL is Hitting Limits

👉 Is it query complexity, write throughput, data size, or scaling cost?

Example: If slow queries → optimize indexes first. If write throughput → maybe NoSQL.

Diagnose root cause | exhaust SQL optimizations first

2️⃣ Consider Hybrid Approach

👉 Keep SQL for transactional data, add NoSQL for specific use cases.

Example: PostgreSQL for orders/payments + Redis for caching + Cassandra for logs.

Polyglot persistence | best tool for each job

3️⃣ Understand What You're Giving Up

👉 NoSQL means losing JOINs, transactions, foreign keys, and mature tooling.

Example: Multi-table reports requiring JOINs become application-side joins (slow and complex).

Trade-offs | application complexity

4️⃣ Evaluate Schema Flexibility Needs

👉 Do you actually need flexible schema or is it premature optimization?

Example: User profiles with varying fields → MongoDB. Fixed schema → stick with SQL.

JSONB in PostgreSQL can provide flexibility without full migration

---

5️⃣ Consider NewSQL Databases

👉 Get SQL semantics with horizontal scalability.

Example: CockroachDB, Google Spanner, TiDB → SQL interface with distributed architecture.

NewSQL | best of both worlds | higher cost

6️⃣ Plan Incremental Migration

👉 Don't do big-bang migration — move one service/table at a time.

Example: Migrate analytics data to Cassandra first, keep transactional data in SQL.

Risk mitigation | incremental approach | dual-write pattern

💡 Problem 55: Handling Transactions Without ACID

Question: INTERVIEWER: You moved to DynamoDB but now need to update multiple items atomically. How?

Solution:

NoSQL databases lack multi-item transactions — you need workarounds.

Here's how to handle transactions in NoSQL 👇

1️⃣ Use Database-Specific Transaction Features

👉 Some NoSQL databases support limited transactions.

Example: DynamoDB transactions support up to 25 items, MongoDB supports multi-document transactions.

Native transaction support | check database capabilities

2️⃣ Implement Two-Phase Commit in Application

👉 Manually coordinate writes across multiple items with prepare/commit phases.

Example: Mark items as "pending" → verify all ready → commit all → mark as "complete".

Application-level 2PC | complex error handling

3️⃣ Use Saga Pattern with Compensating Actions

👉 Break transaction into steps with rollback logic for each step.

Example: Reserve inventory → charge payment → if payment fails → unreserve inventory.

Compensating transactions | eventual consistency

4️⃣ Design Data Model to Avoid Multi-Item Transactions

👉 Embed related data in same document/item for atomic updates.

Example: Store order items inside order document → update entire document atomically.

Denormalization | data locality

5️⃣ Use Conditional Writes for Optimistic Locking

👉 Update only if item hasn't changed since you read it.

Example: Read item with version=5 → update with condition "version=5" → fails if someone else updated.

Optimistic concurrency | DynamoDB conditional writes | MongoDB update operators

---

6️⃣ Accept Eventual Consistency

👉 Design system to tolerate temporary inconsistencies.

Example: Shopping cart shows approximate inventory — occasional oversells handled by customer service.

Eventual consistency | business logic accommodation

📚 Session Management

💡 Problem 56: Querying Patterns in NoSQL

Question: INTERVIEWER: In SQL you'd write complex queries with JOINs and WHERE clauses. How do you do this in NoSQL?

Solution:

NoSQL databases optimize for specific query patterns — you need to design schema around queries.

Here's how to handle querying in NoSQL 👇

1️⃣ Denormalize Data for Query Patterns

👉 Duplicate data so each query hits one table/collection.

Example: Store user name in posts collection even though it's in users → no JOIN needed.

Data duplication | write amplification | faster reads

2️⃣ Create Multiple Tables for Different Access Patterns

👉 In wide-column stores, create separate table per query.

Example: Posts by author table + posts by date table → same data, different partition keys.

Cassandra | DynamoDB | query-driven design

3️⃣ Use Secondary Indexes Cautiously

👉 NoSQL secondary indexes are slower than SQL indexes — use sparingly.

Example: DynamoDB GSI for querying by email → slower than primary key access.

Global secondary index | additional cost | eventual consistency

4️⃣ Perform JOINs in Application Code

👉 Fetch related data in multiple queries and join in application.

Example: Fetch posts → collect author IDs → fetch authors with IN query → merge in code.

Application-side joins | N+1 problem | dataloader pattern

5️⃣ Use Aggregation Pipelines

👉 Document databases support aggregation frameworks for complex queries.

Example: MongoDB aggregation → $match, $group, $sort → similar to SQL GROUP BY.

MongoDB aggregation | limited compared to SQL

6️⃣ Pre-Compute Analytics Data

👉 Run background jobs to aggregate data for dashboards.

Example: Count posts per user overnight → store in separate collection → dashboard reads pre-computed data.

Materialized views | batch processing | Spark/Flink

Topic 20: Consistent Hashing

💡 Problem 57: How Consistent Hashing Works

Question: INTERVIEWER: You have 10 cache servers. When you add an 11th server, all cache keys get invalidated. Why?

Solution:

Simple modulo hashing causes massive cache invalidation when nodes change — consistent hashing solves this.

Here's how consistent hashing works 👇

1️⃣ Problem with Simple Modulo Hashing

👉 Hash(key) % N maps keys to servers, but changing N remaps all keys.

Example: Hash("user123") % 10 = server 3. Add 11th server → Hash("user123") % 11 = server 8 → cache miss.

Massive reshuffling | cache invalidation storm

2️⃣ Consistent Hashing Ring

👉 Map servers and keys onto circular hash space (0 to 2^32).

Example: Like a clock — servers at positions 3, 7, 11 o'clock → keys map to next clockwise server.

Hash ring | minimal disruption

3️⃣ Key Assignment to Servers

👉 Hash key → place on ring → clockwise to next server.

Example: Hash("user123") = 5 → server at position 7 handles it.

Deterministic assignment | consistent across clients

4️⃣ Adding Server Only Affects Neighbors

👉 New server takes some keys from next clockwise server only.

Example: Add server at position 5 → takes keys between 3 and 5 from server at 7 → other 9 servers unaffected.

Minimal data movement | ~1/N keys move

5️⃣ Removing Server Redistributes to Next Server

👉 When server fails, its keys move to next clockwise server.

Example: Server at 7 fails → its keys go to server at 11.

Automatic failover | ~1/N keys move

6️⃣ Virtual Nodes for Even Distribution

👉 Each physical server gets multiple positions on ring for balanced load.

Example: 10 servers × 150 virtual nodes each → 1500 points on ring → more even distribution.

Virtual nodes | avoid hotspots | DynamoDB uses 128 virtual nodes

---

💡 Problem 58: Implementing Consistent Hashing

Question: INTERVIEWER: How would you implement consistent hashing for a distributed cache?

Solution:

Implementation requires hash function, ring data structure, and virtual nodes for balance.

Here's how to implement consistent hashing 👇

1️⃣ Choose Hash Function

👉 Use uniform distribution hash like MD5, SHA-1, or MurmurHash.

Example: MurmurHash("cache-1-vnode-5") → 32-bit integer → position on ring.

Uniform distribution | fast computation

2️⃣ Represent Ring with Sorted Data Structure

👉 Store virtual node positions in sorted list or tree.

Example: TreeMap in Java, bisect in Python → O(log N) lookup for next server.

Binary search | efficient lookups

3️⃣ Create Virtual Nodes for Each Server

👉 Hash (server + vnodeNumber) to create multiple ring positions per server.

Example: For cache-1 with 150 vnodes → Hash("cache-1-vnode-0") through Hash("cache-1-vnode-149").

Load balancing | 100-200 vnodes per server typical

4️⃣ Lookup Function: Find Next Server

👉 Hash key → binary search for next virtual node on ring → return physical server.

Example: Hash("user123") = 1234567 → binary search ring → find first vnode >= 1234567 → return server.

O(log N) complexity | ceiling search

5️⃣ Handle Ring Wrap-Around

👉 If key hash > largest ring position, wrap to beginning.

Example: Hash("user456") = max+100 → wrap to first vnode on ring.

Circular ring | modulo arithmetic

6️⃣ Add/Remove Servers Dynamically

👉 Insert/delete virtual nodes from ring → only affected keys need remapping.

Example: Remove cache-5 → delete its 150 vnodes from ring → clients automatically use next server.

Membership changes | minimal disruption

📚 CORS (Cross-Origin Resource Sharing)

💡 Problem 59: Consistent Hashing Use Cases

Question: INTERVIEWER: When would you use consistent hashing in your system?

Solution:

Consistent hashing excels when you need dynamic cluster membership with minimal disruption.

Here are use cases for consistent hashing 👇

1️⃣ Distributed Caching

👉 Cache servers can scale up/down without invalidating entire cache.

Example: Memcached cluster with 20 servers → add 5 more → only 20% of cache invalidated instead of 100%.

Redis Cluster | Memcached | CDN caching

2️⃣ Load Balancing Across Servers

👉 Distribute requests consistently to same server for session affinity.

Example: User sessions stick to same backend server → consistent hashing on session_id.

Stateful services | session persistence

3️⃣ Distributed Databases

👉 Partition data across nodes with minimal rebalancing when nodes change.

Example: Cassandra, DynamoDB, Riak use consistent hashing for data distribution.

Data sharding | node membership changes

4️⃣ CDN Request Routing

👉 Route requests to same edge server for cache locality.

Example: Video chunks for movie_id=123 always go to same CDN edge → better cache hit rate.

Akamai Cloudflare edge caching

5️⃣ Service Discovery

👉 Route service requests consistently to same instance for caching/state.

Example: API requests for user_id always routed to same service instance.

Microservices consul etcd

6️⃣ When NOT to Use Consistent Hashing

👉 Avoid if you need range queries or perfect load balance with few nodes.

Example: Range query "users WHERE age > 30" doesn't work with consistent hashing.

Range queries small clusters prefer range partitioning

Topic 21: Caching Strategies (Cache Invalidation)

💡 Problem 60: Cache Invalidation Strategies

Question: INTERVIEWER: Your cached data is getting stale. How do you keep cache in sync with database?

Solution:

Cache invalidation is hard — choose strategy based on consistency requirements and access patterns.

Here are cache invalidation strategies 👇

1️⃣ Time-Based Expiration (TTL)

👉 Set expiration time on cached entries — simple but may serve stale data.

Example: Cache user profile for 5 minutes → stale for up to 5 min after update.

`TTL` `Redis EXPIRE` `simple but eventual consistency`

2️⃣ Write-Through Cache

👉 Write to cache and database simultaneously — cache always fresh.

Example: Update user profile → write to database AND cache → reads always see latest data.

`Synchronous writes` `slower writes` `strong consistency`

3️⃣ Write-Behind (Write-Back) Cache

👉 Write to cache immediately, asynchronously write to database later.

Example: Update user profile → write to cache → background job writes to database.

`Fast writes` `risk of data loss` `eventual consistency`

4️⃣ Cache Invalidation on Write

👉 Delete cache entry when database is updated — next read repopulates cache.

Example: Update user profile → delete from cache → next read fetches from database and caches.

Cache-aside pattern | read-after-write fetches from database

5️⃣ Event-Based Invalidation

👉 Publish event when data changes — cache subscribers invalidate their entries.

Example: User service updates profile → publishes event → cache service receives event → deletes cache entry.

Pub/sub | Kafka | RabbitMQ | decoupled architecture

6️⃣ Version-Based Invalidation

👉 Include version number in cache key — new version creates new cache entry.

Example: Cache key "user:123:v5" → update user → cache key becomes "user:123:v6" → old cache ignored.

Versioned keys | no explicit invalidation | cache bloat risk

📚 WebSockets

💡 Problem 61: Cache Warming Strategies

Question: INTERVIEWER: After deploying new cache servers, cache hit rate is 0% and database is overloaded. How do you fix this?

Solution:

Cold cache causes thundering herd on database — pre-populate cache before taking traffic.

Here's how to warm up cache 👇

1️⃣ Pre-Populate Critical Data

👉 Before taking traffic, load frequently accessed data into cache.

Example: Load top 1000 products before launching sale → cache ready for traffic spike.

Batch pre-loading | startup script

2️⃣ Gradual Traffic Ramp-Up

👉 Send small percentage of traffic to new cache server, gradually increase.

Example: 1% traffic for 10 min → 10% for 10 min → 100% → cache warms gradually.

Traffic shaping | canary deployment

3️⃣ Copy from Existing Cache

👉 If adding cache servers, copy hot keys from existing servers.

Example: Redis SCAN existing server → copy top 10K hot keys to new server.

Cache replication | hot key migration

4️⃣ Background Refresh for Predictable Access

👉 Periodically refresh cache entries before they expire.

Example: Refresh homepage data every 4 minutes (TTL 5 min) → always warm.

Proactive refresh | scheduled jobs

5️⃣ Probabilistic Early Expiration

👉 Refresh cache entries probabilistically before expiration to smooth load.

Example: TTL 60s → when TTL < 10s, 20% chance to refresh → avoids synchronized expiration.

Stampede prevention | jittered expiration

6️⃣ Read-Through with Lazy Loading

👉 Accept initial cache misses and let organic traffic warm cache.

Example: First request hits database → populates cache → subsequent requests hit cache.

Lazy loading | simpler but initial spike

💡 Problem 62: Multi-Layer Caching

Question: INTERVIEWER: How would you design caching for a high-traffic website with multiple layers?

Solution:

Multi-layer caching reduces latency and load at each tier — place caches close to access points.

Here's how to design multi-layer caching 👇

1️⃣ Browser/Client Cache

👉 Cache static assets and API responses in browser.

Example: Cache CSS/JS with Cache-Control: max-age=31536000 → served from browser cache.

HTTP caching | ETags | localStorage

2️⃣ CDN Edge Cache

👉 Cache content at CDN edge locations close to users.

Example: Product images cached at CDN edge → 10ms latency instead of 100ms.

Cloudflare | Akamai | CloudFront

3️⃣ Application-Level Cache (In-Memory)

👉 Cache frequently accessed data in application memory.

Example: Cache user session in application RAM → no network call needed.

Caffeine | Guava Cache | local cache

4️⃣ Distributed Cache (Redis/Memcached)

👉 Shared cache across multiple application servers.

Example: User profile cached in Redis → all app servers share same cache.

Redis Memcached shared state

5️⃣ Database Query Cache

👉 Database caches query results internally.

Example: PostgreSQL caches frequently executed queries in shared buffers.

PostgreSQL shared_buffers MySQL query cache

6️⃣ Cache Hierarchy Strategy

👉 Check caches from closest to farthest — L1, L2, L3 → database.

Example: Check local cache → check Redis → check database → populate caches on way back.

Cascade caching return-and-populate pattern

7️⃣ Cache Consistency Across Layers

👉 Invalidate all cache layers when data changes to avoid stale data.

Example: Update user profile → invalidate local cache, Redis, CDN.

Multi-layer invalidation cache coherence

Topic 22: Cache-Aside Pattern

📚 GraphQL

💡 Problem 63: Implementing Cache-Aside Pattern

Question: INTERVIEWER: How do you implement caching in your application to reduce database load?

Solution:

Cache-aside (lazy loading) is the most common pattern — application manages cache explicitly.

Here's how cache-aside pattern works 👇

1️⃣ Read Path: Check Cache First

👉 Application checks cache before querying database.

Example: Fetch user → check Redis → if found, return → if miss, query database.

Cache lookup | Redis GET

2️⃣ Cache Miss: Fetch from Database

👉 On cache miss, query database for data.

Example: User not in cache → query `SELECT * FROM users WHERE id = 123`.

Database query | fallback to source of truth

3️⃣ Populate Cache After Fetch

👉 Store database result in cache for future requests.

Example: Fetched user from database → store in Redis with `SET user:123 {data} EX 3600`.

Cache population | set TTL

4️⃣ Write Path: Update Database First

👉 On write, update database then invalidate cache.

Example: Update user → write to database → delete cache key `DEL user:123`.

Write-through invalidation | next read repopulates cache

5️⃣ Handle Race Conditions

👉 Multiple threads may fetch same data simultaneously on cache miss.

Example: 100 requests for user:123 → all miss cache → all query database → use lock or accept duplicate queries.

Thundering herd | use setnx lock

6️⃣ Cache-Aside Benefits and Trade-offs

👉 Simple to implement, cache only what's accessed, but cache misses impact latency.

Example: Cache hit = 1ms, cache miss = 100ms database query.

Lazy loading | eventual consistency | read-heavy workloads

💡 Problem 64: Cache-Aside vs Write-Through vs Write-Behind

Question: INTERVIEWER: What's the difference between cache-aside, write-through, and write-behind caching?

Solution:

Different patterns trade off consistency, latency, and complexity.

Here are the caching pattern trade-offs 👇

1️⃣ Cache-Aside (Lazy Loading)

👉 Application explicitly manages cache — read from cache, on miss fetch from database and populate.

Example: Most common pattern — application checks Redis, then database.

Application-managed | eventual consistency | simple

---

2 Write-Through Cache

👉 Writes go through cache to database — cache always consistent with database.

Example: Update user → cache writes to database synchronously → returns success.

Synchronous writes | slower writes | strong consistency

---

3 Write-Behind (Write-Back) Cache

👉 Writes go to cache immediately, asynchronously written to database.

Example: Update user → write to cache → return → background job flushes to database.

Fast writes | data loss risk | eventual consistency

---

4 Read-Through Cache

👉 Cache sits in front of database — cache automatically fetches from database on miss.

Example: Application queries cache → cache fetches from database if miss → returns data.

Cache-managed | transparent to application

---

5 When to Use Each Pattern

👉 Cache-aside for most use cases, write-through for critical consistency, write-behind for high write throughput.

Example: User profiles → cache-aside. Financial transactions → write-through. Analytics events → write-behind.

Choose based on requirements

Topic 23: Distributed Caching (Redis)

💡 Problem 65: Redis vs Memcached

INTERVIEWER: When would you use Redis vs Memcached?

Question:

Solution:

Both are distributed caches, but Redis offers richer data structures and persistence.

Here's when to use Redis vs Memcached 👇

1️⃣ Use Memcached for Simple Key-Value Caching

👉 Memcached is simpler, faster for basic get/set operations.

Example: Cache API responses, HTML fragments → Memcached sufficient.

Simple caching | multi-threaded | less memory overhead

2️⃣ Use Redis for Rich Data Structures

👉 Redis supports lists, sets, sorted sets, hashes beyond simple strings.

Example: Leaderboard with sorted set, shopping cart with hash, job queue with list.

Data structures | ZADD | HSET | LPUSH

3️⃣ Use Redis for Persistence

👉 Redis can persist cache to disk for recovery after restart.

Example: Session data → Redis with AOF persistence → survives server restart.

RDB snapshots | AOF | durability

4️⃣ Use Redis for Pub/Sub

👉 Redis supports publish/subscribe messaging pattern.

Example: Real-time notifications → publish to Redis channel → subscribers receive instantly.

PUBLISH SUBSCRIBE message broker

5️⃣ Use Redis for Transactions and Lua Scripts

👉 Redis supports atomic operations with MULTI/EXEC and Lua scripting.

Example: Decrement inventory atomically with Lua script → prevent race conditions.

MULTI/EXEC EVAL atomicity

6️⃣ Use Memcached for Multi-Threaded Performance

👉 Memcached uses multi-threading, better CPU utilization on multi-core systems.

Example: 16-core server → Memcached uses all cores, Redis single-threaded (one core per instance).

Multi-threading horizontal scaling

📚 gRPC

💡 Problem 66: Redis Clustering and High Availability

INTERVIEWER: Your single Redis instance is a single point of failure. How do you make Redis highly available?

Question:

Solution:

Redis supports replication, Sentinel for failover, and clustering for horizontal scaling.

Here's how to make Redis highly available 👇

1️⃣ Redis Replication (Master-Slave)

👉 Replicate data from master to one or more replicas for read scaling and failover.

Example: 1 master for writes, 3 replicas for reads → reads scale horizontally.

`REPLICAOF` `asynchronous replication`

2️⃣ Redis Sentinel for Automatic Failover

👉 Sentinel monitors master, automatically promotes replica if master fails.

Example: Master crashes → Sentinel detects → promotes replica to master → updates clients.

`Sentinel quorum` `automatic failover` `monitoring`

3️⃣ Redis Cluster for Horizontal Scaling

👉 Shard data across multiple master nodes using hash slots.

Example: 3 master nodes → 16384 hash slots divided evenly → keys distributed across masters.

`Redis Cluster` `16384 hash slots` `no single point of failure`

4️⃣ Read Replicas for Read Scaling

👉 Route read queries to replicas to reduce load on master.

Example: 90% read traffic → replicas handle reads → master handles writes only.

`Read/write splitting` `replica lag`

5️⃣ Persistence for Data Durability

👉 Enable RDB snapshots or AOF to recover data after crash.

Example: Server crashes → restart → load RDB snapshot → data restored.

`RDB` `AOF` `fsync policy`

6️⃣ Connection Pooling and Client-Side Failover

👉 Use connection pool with multiple Redis endpoints for client-side failover.

Example: Client connects to master → master fails → client switches to replica.

Jedis redis-py client libraries

💡 Problem 67: Redis Persistence (RDB vs AOF)

INTERVIEWER: How does Redis persist data to disk, and what are the trade-offs?

Question:

Solution:

Redis offers two persistence mechanisms with different durability and performance trade-offs.

Here's how Redis persistence works 👇

1️⃣ RDB (Redis Database Snapshot)

👉 Periodically save full snapshot of data to disk.

Example: Save snapshot every 5 minutes → server crashes → restore from last snapshot → lose up to 5 min of data.

BGSAVE point-in-time snapshots compact files

2️⃣ AOF (Append-Only File)

👉 Log every write operation to disk — can replay to recover exact state.

Example: Every write appended to AOF → server crashes → replay AOF → restore all data.

Append-only log better durability larger files

3️⃣ RDB Performance

👉 RDB is faster and generates smaller files, but higher data loss risk.

Example: 1GB RDB snapshot takes 10 seconds to load → faster recovery than AOF.

Faster restarts | background fork | data loss risk

4️⃣ AOF Durability

👉 AOF provides better durability with configurable fsync policy.

Example: fsync every second → lose max 1 second of data on crash.

appendfsync everysec | appendfsync always | appendfsync no

5️⃣ Use Both RDB and AOF

👉 Enable both for best durability and fast recovery.

Example: RDB every 5 min + AOF fsync every second → recover from RDB → replay recent AOF for latest state.

Hybrid approach | Redis loads RDB then AOF

6️⃣ AOF Rewrite for Compaction

👉 AOF grows large over time — rewrite to compact.

Example: 1000 SET operations on same key → rewrite as single SET → smaller file.

BGREWRITEAOF | automatic rewrite

💡 Problem 68: Choosing Redis Persistence Strategy

> INTERVIEWER: Your Redis has 10GB of session data — how do you ensure it survives a restart?

Question:

Solution:

Redis persistence balances durability, performance, and recovery time.

1️⃣ RDB (Point-in-time Snapshots)

👉 Periodic snapshots saved to disk.

Example: Save snapshot every 5 minutes → fast restart, but lose up to 5 minutes of data on crash.

RDB snapshots | backup

2️⃣ AOF (Append-Only File)

👉 Log every write operation to file.

Example: Every SET, INCR command logged → replay on restart → minimal data loss but slower.

AOF | durability

3️⃣ AOF fsync Policies

👉 Control how often AOF is flushed to disk.

Example: fsync always (slow, durable), everysec (balanced), or no (fast, risky).

fsync policies | trade-offs

4️⃣ RDB + AOF Hybrid

👉 Use both for best durability and fast restarts.

Example: RDB snapshot every hour + AOF for recent writes → fast recovery with minimal loss.

Hybrid persistence | best of both

5️⃣ AOF Rewrite

👉 Compact AOF file by removing redundant operations.

Example: SET key val1, SET key val2, SET key val3 → rewrites to single SET key val3.

AOF compaction | disk space

6️⃣ No Persistence (Cache-Only)

👉 For ephemeral data that can be regenerated.

Example: Temporary rate-limit counters → no persistence needed → faster.

Cache-only ephemeral data

7️⃣ Backup Strategy

👉 Regular backups of RDB files to external storage.

Example: Cron job copies RDB to S3 every 6 hours → disaster recovery.

Backup S3 disaster recovery

💡 Problem 68: Choosing Redis Persistence Strategy

> INTERVIEWER: Your Redis has 10GB of session data — how do you ensure it survives a restart?

Question:

Solution:

Redis persistence balances durability, performance, and recovery time.

1️⃣ RDB (Point-in-time Snapshots)

👉 Periodic snapshots saved to disk.

Example: Save snapshot every 5 minutes → fast restart, but lose up to 5 minutes of data on crash.

RDB snapshots backup

2️⃣ AOF (Append-Only File)

👉 Log every write operation to file.

Example: Every SET, INCR command logged → replay on restart → minimal data loss but slower.

AOF durability

3️⃣ AOF fsync Policies

👉 Control how often AOF is flushed to disk.

Example: fsync always (slow, durable), everysec (balanced), or no (fast, risky).

`fsync policies` `trade-offs`

4️⃣ RDB + AOF Hybrid

👉 Use both for best durability and fast restarts.

Example: RDB snapshot every hour + AOF for recent writes → fast recovery with minimal loss.

`Hybrid persistence` `best of both`

5️⃣ AOF Rewrite

👉 Compact AOF file by removing redundant operations.

Example: SET key val1, SET key val2, SET key val3 → rewrites to single SET key val3.

`AOF compaction` `disk space`

6️⃣ No Persistence (Cache-Only)

👉 For ephemeral data that can be regenerated.

Example: Temporary rate-limit counters → no persistence needed → faster.

`Cache-only` `ephemeral data`

7️⃣ Backup Strategy

👉 Regular backups of RDB files to external storage.

Example: Cron job copies RDB to S3 every 6 hours → disaster recovery.

`Backup` `S3` `disaster recovery`

Topic 24: Redis Data Structures

📚 REST API Best Practices

💡 Problem 68: Using Redis Strings and Hashes

INTERVIEWER: How would you store user session data in Redis?

Question:

Solution:

Redis strings and hashes are perfect for key-value storage with expiration.

Here's how to use Redis strings and hashes 👇

1️⃣ Simple String Storage

👉 Store serialized data (JSON, MessagePack) as string.

Example: `SET session:abc123 "{user_id:5,name:Alice}" EX 3600` → expires in 1 hour.

SET GET EXPIRE serialization

2️⃣ Hash for Structured Data

👉 Store individual fields in hash for partial access.

Example: `HSET user:123 name Alice age 30 email alice@example.com` → access fields individually.

HSET HGET HGETALL field-level access

3️⃣ Atomic Increment/Decrement

👉 Use INCR/DECR for counters with atomic operations.

Example: `INCR page:views:123` → increment view counter atomically → prevent race conditions.

INCR DECR INCRBY atomic operations

4️⃣ Set Expiration on Keys

👉 Auto-delete keys after TTL for session management and cache invalidation.

Example: Session expires after 30 minutes → `EXPIRE session:abc123 1800`.

EXPIRE TTL automatic cleanup

5️⃣ Conditional Operations with NX/XX

👉 SETNX sets only if key doesn't exist → useful for distributed locks.

Example: `SET lock:resource:123 1 NX EX 10` → acquire lock if not held.

SETNX distributed lock Redlock

6️⃣ Batch Operations with MGET/MSET

👉 Fetch/set multiple keys in single round trip.

Example: `MGET user:1 user:2 user:3` → 1 network call instead of 3.

MGET MSET pipelining performance

💡 Problem 69: Using Redis Lists and Sets

INTERVIEWER: How would you implement a job queue and a tagging system in Redis?

Question:

Solution:

Redis lists are perfect for queues, sets for unique collections and membership tests.

Here's how to use Redis lists and sets 👇

1️⃣ List as FIFO Queue

👉 LPUSH to enqueue, RPOP to dequeue — simple job queue.

Example: `LPUSH jobs:pending {job_id:1,task:send_email}` *→* `RPOP jobs:pending` *→ process job.*

LPUSH RPOP FIFO queue

2️⃣ Blocking Pop for Worker Pools

👉 BRPOP blocks until item available — efficient worker pattern.

Example: Worker calls `BRPOP jobs:pending 30` *→ blocks up to 30 seconds → processes job when available.*

BRPOP blocking operation worker pool

3️⃣ List as Activity Feed

👉 Store recent activity, trim to keep limited history.

Example: `LPUSH feed:user:123 {action:like,post:456}` *→* `LTRIM feed:user:123 0 99` *→ keep last 100 items.*

LPUSH LTRIM LRANGE recent activity

4️⃣ Set for Unique Collections

👉 Store unique items, automatic deduplication.

Example: `SADD tags:post:456 python backend redis` *→ no duplicates.*

SADD SMEMBERS unique values

5️⃣ Set Operations (Union, Intersection, Difference)

👉 Combine sets to find common or unique elements.

Example: `SINTER tags:post:1 tags:post:2` *→ find common tags between posts.*

SINTER SUNION SDIFF set algebra

6️⃣ Set Membership Testing

👉 Check if element exists in set — O(1) operation.

Example: `SISMEMBER followers:user:123 user:456` → check if user:456 follows user:123.

SISMEMBER | fast membership test

💡 Problem 70: Using Redis Sorted Sets

> INTERVIEWER: How would you implement a real-time leaderboard that ranks users by score?

Question:

Solution:

Redis sorted sets maintain sorted order automatically — perfect for leaderboards and priority queues.

Here's how to use Redis sorted sets 👇

1️⃣ Add Elements with Scores

👉 ZADD stores members with scores, automatically sorted.

Example: `ZADD leaderboard 1500 user:123 2000 user:456` → users ranked by score.

ZADD | score-based sorting

2️⃣ Get Top N Elements

👉 ZREVRANGE returns highest-scoring elements.

Example: `ZREVRANGE leaderboard 0 9 WITHSCORES` → top 10 users with scores.

ZREVRANGE | leaderboard queries

3️⃣ Get User Rank

👉 ZREVRANK returns user's position in leaderboard.

Example: `ZREVRANK leaderboard user:123` → returns rank 42 → user is 42nd place.

ZREVRANK | position lookup | O(log N)

4️⃣ Increment Score Atomically

👉 ZINCRBY atomically increases user's score.

Example: User earns 50 points → `ZINCRBY leaderboard 50 user:123` → atomic update.

ZINCRBY | atomic increment | race-free

5️⃣ Range Queries by Score

👉 ZRANGEBYSCORE returns elements within score range.

Example: `ZRANGEBYSCORE leaderboard 1000 2000` → users with scores between 1000 and 2000.

ZRANGEBYSCORE | score filtering

6️⃣ Time-Based Leaderboards with Timestamps

👉 Use timestamp as score for time-ordered data.

Example: `ZADD trending:posts <unix_timestamp> post:456` → get recent posts with ZREVRANGEBYSCORE.

Timestamp as score | time-series data

Topic 25: Cache Stampede Problem

💡 Problem 71: Understanding Cache Stampede

INTERVIEWER: A popular cache entry expires, and suddenly 10,000 requests hit your database at once. What happened?

Question:

Solution:

Cache stampede (thundering herd) happens when many requests try to regenerate expired cache simultaneously.

Here's how cache stampede happens 👇

1️⃣ What is Cache Stampede?

👉 Multiple requests simultaneously detect cache miss and query database.

Example: Popular product page cached → cache expires → 10K concurrent requests → all hit database → database overload.

Thundering herd | synchronized cache miss

2️⃣ Why It's Problematic

👉 Database handles 1K req/sec normally → suddenly receives 10K concurrent queries → crashes or slows down.

Example: Homepage cache expires → millions of users hit database → database CPU at 100%.

Database overload | cascading failure

3️⃣ When Does It Occur?

👉 Popular cached items with synchronized expiration times.

Example: All cache entries created at deploy time with TTL 3600 → all expire at same time → stampede.

Synchronized expiration | hot keys

4️⃣ Effect on User Experience

👉 Users experience slow responses during stampede.

Example: Page loads in 50ms from cache → during stampede takes 5 seconds from database.

Latency spike | timeout errors

5️⃣ Risk of Cascading Failure

👉 Database overload causes timeouts → retries increase load → more timeouts → death spiral.

Example: Database slow → app retries → more queries → database crashes → entire system down.

Cascading failure | retry storm

💡 Problem 3: Where to Add Caching for Slow Homepage

> INTERVIEWER: Your homepage takes 2 seconds to load — where exactly do you add caching?

Question:

Solution:

Strategic caching placement at multiple layers dramatically reduces page load time.

1️⃣ Browser Caching

👉 Cache static assets (CSS, JS, images) in user's browser.

Example: Set Cache-Control: max-age=31536000 for images → users download once, use for a year.

Browser cache headers | CDN integration

2️⃣ CDN Caching

👉 Serve static content from edge servers close to users.

Example: Homepage HTML cached at CDN → 50ms latency instead of 500ms from origin server.

CloudFront Cloudflare Akamai

3️⃣ Application-Level Caching

👉 Cache database query results in Redis/Memcached.

Example: Product listings, user profiles cached → avoid 10 DB queries on every page load.

Redis Memcached in-memory cache

4️⃣ Database Query Caching

👉 Cache expensive aggregation queries at database level.

Example: SELECT COUNT(*) from million-row table → cache result for 5 minutes.

MySQL query cache materialized views

5️⃣ Full Page Caching

👉 Cache entire rendered HTML for anonymous users.

Example: Homepage identical for all logged-out users → generate once, serve millions.

Varnish Nginx proxy cache static site generation

6️⃣ Fragment/Partial Caching

👉 Cache reusable page components separately.

Example: Product recommendations widget cached independently from rest of page.

Edge Side Includes React Server Components

7️⃣ Cache Warming

👉 Pre-populate cache before traffic surge.

Example: Before Black Friday, pre-cache top 1000 products → no cold start delays.

Scheduled jobs background workers

💡 Problem 4: Reducing API Latency from 450ms to <100ms

INTERVIEWER: Your search API must respond in <100ms — but DB takes 450ms. What's your move?

Question:

Solution:

Sub-100ms API responses require eliminating database calls from the critical path.

1️⃣ Add Search Engine Layer

👉 Use Elasticsearch/Algolia for fast full-text search instead of DB.

Example: Elasticsearch returns search results in 10-30ms vs 450ms from PostgreSQL.

Elasticsearch | Algolia | Typesense

2️⃣ Aggressive Caching

👉 Cache search results for popular queries.

Example: "iPhone 15" searched 10K times/hour → cache results for 60 seconds.

Redis cache | CDN caching | stale-while-revalidate

3️⃣ Read Replicas for Search

👉 Route search queries to optimized read replicas.

Example: Dedicated read replica with search-optimized indexes → 200ms → 80ms.

PostgreSQL replicas | read-only nodes

4️⃣ Database Query Optimization

👉 Add proper indexes and optimize search queries.

Example: Add GIN index for full-text search → 450ms → 150ms (still not enough).

Index optimization | query planning

5️⃣ Denormalization

👉 Pre-compute search-friendly data structures.

Example: Denormalized search_index table with all searchable fields → simpler, faster queries.

Materialized views | denormalized tables

6️⃣ Asynchronous Search

👉 Return instant results from cache, update in background if stale.

Example: Return cached results in 20ms, trigger background refresh if >5 minutes old.

Background jobs | eventual consistency

7️⃣ Search-Specific Database

👉 Use database optimized for search workloads.

Example: Migrate from PostgreSQL to Elasticsearch for product search only.

Specialized databases | polyglot persistence

💡 Problem 3: Where to Add Caching for Slow Homepage

> INTERVIEWER: Your homepage takes 2 seconds to load — where exactly do you add caching?

Question:

Solution:

Strategic caching placement at multiple layers dramatically reduces page load time.

1️⃣ Browser Caching

👉 Cache static assets (CSS, JS, images) in user's browser.

Example: Set Cache-Control: max-age=31536000 for images → users download once, use for a year.

Browser cache headers | CDN integration

2️⃣ CDN Caching

👉 Serve static content from edge servers close to users.

Example: Homepage HTML cached at CDN → 50ms latency instead of 500ms from origin server.

CloudFront | Cloudflare | Akamai

3️⃣ Application-Level Caching

👉 Cache database query results in Redis/Memcached.

Example: Product listings, user profiles cached → avoid 10 DB queries on every page load.

Redis | Memcached | in-memory cache

4️⃣ Database Query Caching

👉 Cache expensive aggregation queries at database level.

Example: SELECT COUNT(*) from million-row table → cache result for 5 minutes.

MySQL query cache | materialized views

5️⃣ Full Page Caching

👉 Cache entire rendered HTML for anonymous users.

Example: Homepage identical for all logged-out users → generate once, serve millions.

Varnish | Nginx proxy cache | static site generation

6️⃣ Fragment/Partial Caching

👉 Cache reusable page components separately.

Example: Product recommendations widget cached independently from rest of page.

Edge Side Includes | React Server Components

7️⃣ Cache Warming

👉 Pre-populate cache before traffic surge.

Example: Before Black Friday, pre-cache top 1000 products → no cold start delays.

Scheduled jobs | background workers

💡 Problem 4: Reducing API Latency from 450ms to <100ms

> INTERVIEWER: Your search API must respond in <100ms — but DB takes 450ms. What's your move?

Question:

Solution:

Sub-100ms API responses require eliminating database calls from the critical path.

1️⃣ Add Search Engine Layer

👉 Use Elasticsearch/Algolia for fast full-text search instead of DB.

Example: Elasticsearch returns search results in 10-30ms vs 450ms from PostgreSQL.

Elasticsearch | Algolia | Typesense

2️⃣ Aggressive Caching

👉 Cache search results for popular queries.

Example: "iPhone 15" searched 10K times/hour → cache results for 60 seconds.

Redis cache | CDN caching | stale-while-revalidate

3️⃣ Read Replicas for Search

👉 Route search queries to optimized read replicas.

Example: Dedicated read replica with search-optimized indexes → 200ms → 80ms.

PostgreSQL replicas | read-only nodes

4️⃣ Database Query Optimization

👉 Add proper indexes and optimize search queries.

Example: Add GIN index for full-text search → 450ms → 150ms (still not enough).

Index optimization | query planning

5️⃣ Denormalization

👉 Pre-compute search-friendly data structures.

Example: Denormalized search_index table with all searchable fields → simpler, faster queries.

Materialized views | denormalized tables

6️⃣ Asynchronous Search

👉 Return instant results from cache, update in background if stale.

Example: Return cached results in 20ms, trigger background refresh if >5 minutes old.

Background jobs | eventual consistency

7️⃣ Search-Specific Database

👉 Use database optimized for search workloads.

Example: Migrate from PostgreSQL to Elasticsearch for product search only.

Specialized databases | polyglot persistence

📚 Webhooks

💡 Problem 72: Preventing Cache Stampede

INTERVIEWER: How do you prevent cache stampede from bringing down your database?

Question:

Solution:

Use locking, probabilistic expiration, or early cache refresh to prevent thundering herd.

Here's how to prevent cache stampede 👇

1️⃣ Lock-Based Cache Regeneration

👉 First request acquires lock, regenerates cache → others wait for lock to be released.

Example: Request 1 gets lock → queries database → populates cache → releases lock → other requests read from cache.

Distributed lock | Redis SETNX | Redlock

2️⃣ Probabilistic Early Expiration

👉 Randomly refresh cache before expiration based on probability.

Example: TTL 60s → when TTL < 10s, 10% chance to refresh → avoids synchronized expiration.

XFetch algorithm | jittered expiration

3️⃣ Serve Stale While Revalidating

👉 Return expired cache immediately, refresh asynchronously in background.

Example: Cache expired → return stale data → background job refreshes cache → next request gets fresh data.

Stale-while-revalidate eventual consistency

4️⃣ Background Refresh Before Expiration

👉 Proactively refresh cache before it expires.

Example: Cache expires in 60s → background job refreshes at 55s → cache never expires.

Proactive refresh scheduled jobs cron

5️⃣ Add Jitter to TTLs

👉 Randomize expiration times to avoid synchronized cache misses.

Example: Base TTL 3600s → add random 0-300s → TTL between 3600-3900s → expirations spread out.

TTL jitter desynchronization

6️⃣ Use Semaphore/Rate Limiting

👉 Limit concurrent database queries during cache miss.

Example: Max 10 concurrent regenerations → others wait or return stale data.

Semaphore rate limiting backpressure

Topic 26: Kafka Architecture

💡 Problem 73: Understanding Kafka Topics, Partitions, and Consumer Groups

INTERVIEWER: Explain how Kafka distributes messages across multiple consumers.

Question:

Solution:

Kafka partitions topics for parallelism — consumer groups enable load balancing across consumers.

Here's how Kafka topics and partitions work 👇

1️⃣ Topics as Message Categories

👉 Topic is a logical grouping of related messages.

Example: `user-events` topic for user actions, `payment-events` topic for payments.

Logical separation | message routing

2️⃣ Partitions for Parallelism

👉 Topic divided into partitions — each partition is ordered log of messages.

Example: `user-events` topic with 10 partitions → messages distributed across 10 partitions.

Horizontal scaling | parallelism | ordering per partition

3️⃣ Partition Assignment by Key

👉 Messages with same key go to same partition for ordering guarantee.

Example: All events for user_id=123 go to same partition → ordered per user.

Hash(key) % num_partitions | ordering guarantee

4️⃣ Consumer Groups for Load Balancing

👉 Consumers in same group split partitions among themselves.

Example: 10 partitions, 5 consumers → each consumer reads from 2 partitions.

Load balancing | horizontal scaling

5️⃣ One Partition → One Consumer Per Group

👉 Each partition consumed by exactly one consumer in a group at a time.

Example: 10 partitions, 15 consumers → 10 consumers active, 5 idle (can't have more consumers than partitions).

Partition-to-consumer mapping | scaling limit

6️⃣ Multiple Consumer Groups for Different Use Cases

👉 Each consumer group independently consumes all messages.

Example: Group A for analytics, Group B for notifications → both process all messages.

Fan-out pattern | independent processing

📚 Idempotency

💡 Problem 74: Kafka Offsets and Message Delivery Guarantees

> INTERVIEWER: How does Kafka ensure messages aren't lost or processed twice?

Question:

Solution:

Kafka uses offsets for tracking and offers three delivery semantics.

Here's how Kafka offsets and delivery guarantees work 👇

1️⃣ Offset as Message Position

👉 Offset is sequential ID of message within partition.

Example: Partition has messages at offsets 0, 1, 2, 3... → consumer tracks "I've read up to offset 100".

Sequential position | per-partition offset

2️⃣ Consumer Commits Offset

👉 Consumer periodically saves its current offset to Kafka.

Example: Consumer reads offset 100 → commits offset 100 → crashes → restarts from offset 100.

Offset commit | resume from last commit

3️⃣ At-Most-Once Delivery

👉 Commit offset before processing message — message may be lost on crash.

Example: Read offset 100 → commit 100 → crash before processing → message lost.

Commit-then-process | data loss risk | lowest latency

4️⃣ At-Least-Once Delivery

👉 Process message then commit offset — message may be reprocessed on crash.

Example: Read offset 100 → process → crash before commit → restart → process offset 100 again.

Process-then-commit | duplicates possible | most common

5️⃣ Exactly-Once Semantics

👉 Kafka transactions ensure message processed and offset committed atomically.

Example: Process message and commit offset in single transaction → no loss or duplication.

Kafka transactions | idempotent producer | highest consistency

---

6️⃣ Manual vs Automatic Offset Commit

👉 Auto-commit commits offsets periodically, manual commit gives fine-grained control.

Example: Auto-commit every 5 seconds → simple but may duplicate/lose messages. Manual commit after processing → more control.

enable.auto.commit | commitSync | commitAsync

💡 Problem 75: Kafka Replication and High Availability

INTERVIEWER: How does Kafka ensure messages aren't lost when a broker crashes?

Question:

Solution:

Kafka replicates partitions across brokers for fault tolerance.

Here's how Kafka replication works 👇

1️⃣ Replication Factor

👉 Each partition replicated to N brokers.

Example: Replication factor 3 → partition stored on 3 different brokers.

Fault tolerance data redundancy

2️⃣ Leader and Follower Replicas

👉 One replica is leader (handles reads/writes), others are followers (replicate).

Example: Broker 1 is leader for partition 0 → Brokers 2 and 3 are followers.

Leader election read/write routing

3️⃣ In-Sync Replicas (ISR)

👉 Replicas caught up with leader are "in-sync" — eligible for leadership.

Example: Leader has offset 1000 → followers at 999 and 1000 are ISR → follower at 900 is not ISR.

Replica lag ISR set

4️⃣ Producer Acknowledgments (acks)

👉 Control how many replicas must acknowledge write before success.

Example: acks=1 → leader ack only (fast, data loss risk). acks=all → wait for all ISR (slow, durable).

Durability vs latency | acks config

5️⃣ Leader Election on Failure

👉 When leader fails, Kafka elects new leader from ISR.

Example: Leader broker crashes → Kafka controller picks ISR follower → promotes to leader.

Automatic failover | ZooKeeper/KRaft coordination

6️⃣ Min In-Sync Replicas

👉 Minimum number of ISRs required for write to succeed.

Example: Replication factor 3, min.insync.replicas=2 → write succeeds if 2+ replicas available.

Availability vs durability | min.insync.replicas

Topic 27: RabbitMQ Patterns

📚 Pagination

💡 Problem 76: RabbitMQ Exchanges and Routing

INTERVIEWER: How does RabbitMQ route messages from producers to queues?

Question:

Solution:

RabbitMQ uses exchanges with routing rules to deliver messages to queues.

Here's how RabbitMQ exchanges work 👇

1️⃣ Producer → Exchange → Queue → Consumer

👉 Producers send to exchanges, exchanges route to queues based on routing key.

Example: Producer sends to "orders" exchange → exchange routes to specific queue.

Decoupling | flexible routing

2️⃣ Direct Exchange (Routing Key Match)

👉 Route message to queue whose binding key exactly matches routing key.

Example: Message with routing key "error" → routed to queue bound with "error".

Exact match | unicast

3️⃣ Fanout Exchange (Broadcast)

👉 Broadcast message to all bound queues, ignore routing key.

Example: Log message → fanout exchange → delivered to all logging queues.

Broadcast | pub/sub pattern

4️⃣ Topic Exchange (Pattern Matching)

👉 Route based on wildcard pattern matching.

Example: Routing key "user.signup.email" → matches queue bound to "user..email" or "user.signup.*".*

Wildcard routing | * = one word | # = zero or more words

5️⃣ Headers Exchange (Header Matching)

👉 Route based on message header attributes instead of routing key.

Example: Message with headers {type: email, priority: high} → routed to matching queue.

Attribute-based routing complex matching

6️⃣ Default Exchange (Empty String)

👉 Automatically routes to queue with name matching routing key.

Example: Send to default exchange with routing key "my-queue" → delivered to queue named "my-queue".

Implicit routing simple use cases

💡 Problem 77: RabbitMQ Message Acknowledgments and Delivery Guarantees

> INTERVIEWER: How does RabbitMQ ensure messages aren't lost if a consumer crashes?

Question:

Solution:

RabbitMQ uses acknowledgments and persistence for message durability.

Here's how RabbitMQ delivery guarantees work 👇

1️⃣ Manual Acknowledgment (Ack)

👉 Consumer explicitly acks message after successful processing.

Example: Receive message → process → send ack → RabbitMQ deletes message.

Manual ack reliable delivery

2️⃣ Auto-Acknowledgment

👉 RabbitMQ marks message delivered as soon as sent to consumer.

Example: Send message → auto-ack immediately → consumer crashes → message lost.

Auto-ack fire-and-forget data loss risk

3️⃣ Negative Acknowledgment (Nack/Reject)

👉 Consumer rejects message → RabbitMQ requeues or discards it.

Example: Processing fails → send nack with requeue=true → message goes back to queue.

Error handling | retry mechanism

4️⃣ Publisher Confirms

👉 RabbitMQ confirms to producer that message was received and persisted.

Example: Producer publishes → waits for confirmation → ensures message not lost.

Producer reliability | at-least-once guarantee

5️⃣ Message Persistence

👉 Mark messages and queues as durable → survives broker restart.

Example: Durable queue + persistent message → RabbitMQ writes to disk → survives crash.

Durability | disk storage | performance cost

6️⃣ Prefetch Count for Flow Control

👉 Limit unacknowledged messages per consumer to prevent overload.

Example: Prefetch=10 → consumer gets max 10 messages before acking → prevents overwhelming slow consumer.

Flow control | backpressure | QoS

📚 Search & Indexing

💡 Problem 78: RabbitMQ Work Queues and Competing Consumers

> INTERVIEWER: You have 10,000 jobs to process. How do you distribute them across multiple workers using RabbitMQ?

Question:

Solution:

Work queues distribute tasks across competing consumers for parallel processing.

Here's how RabbitMQ work queues work 👇

1️⃣ Multiple Consumers on Same Queue

👉 Multiple consumers read from same queue — RabbitMQ round-robins messages.

Example: Queue with 1000 jobs, 10 workers → each worker gets ~100 jobs.

Competing consumers | load balancing

2️⃣ Round-Robin Dispatching

👉 RabbitMQ distributes messages evenly across consumers.

Example: Message 1 → Consumer A, Message 2 → Consumer B, Message 3 → Consumer C.

Fair distribution | default behavior

3️⃣ Prefetch for Fair Dispatch

👉 Set prefetch count so workers get messages based on processing speed.

Example: Fast worker finishes and gets new message → slow worker still processing → avoids slow worker bottleneck.

Fair dispatch | prefetch count | QoS

4️⃣ Message Acknowledgment for Reliability

👉 If worker crashes, unacked messages redelivered to other workers.

Example: Worker A crashes while processing → RabbitMQ redelivers to Worker B.

Fault tolerance | no message loss

5️⃣ Priority Queues

👉 Assign priorities to messages → higher priority processed first.

Example: Urgent email → priority 10, regular email → priority 1 → urgent processed first.

Priority-based processing | x-max-priority

6️⃣ Dead Letter Exchange for Failed Jobs

👉 Failed jobs routed to dead letter queue after max retries.

Example: Job fails 3 times → moved to DLX → manual investigation or retry later.

Error handling | DLX | x-dead-letter-exchange

Topic 28: Event-Driven Architecture

💡 Problem 79: Designing Event-Driven Systems

> INTERVIEWER: How would you design a system where services communicate via events instead of direct API calls?

Question:

Solution:

Event-driven architecture decouples services — producers emit events, consumers react asynchronously.

Here's how to design event-driven systems 👇

1️⃣ Events as State Changes

👉 Event represents something that happened in the past.

Example: "OrderCreated", "PaymentProcessed", "UserSignedUp" → immutable facts.

Event naming | past tense | immutable

2️⃣ Event Bus/Broker as Central Backbone

👉 Services publish events to central broker, other services subscribe.

Example: Order service publishes OrderCreated → inventory, notification, analytics services subscribe.

Kafka | RabbitMQ | AWS EventBridge

3️⃣ Loose Coupling Between Services

👉 Producers don't know about consumers — add new consumers without changing producers.

Example: Add fraud detection service → subscribes to PaymentProcessed → no changes to payment service.

Decoupling | independent deployment

4️⃣ Choreography vs Orchestration

👉 Choreography: services react to events independently. Orchestration: central controller directs workflow.

Example: Choreography → OrderCreated → inventory decrements, email sends, analytics logs. Orchestration → Order orchestrator calls each service.

Distributed coordination | saga pattern

5️⃣ Event Sourcing for Audit Trail

👉 Store all events as source of truth, rebuild state by replaying events.

Example: Account balance = replay all deposit/withdrawal events.

Event sourcing | append-only log | full audit trail

6️⃣ CQRS (Command Query Responsibility Segregation)

👉 Separate read and write models — events update read models asynchronously.

Example: Write to event store → events update materialized views for queries.

CQRS | read model | write model separation

7️⃣ Handle Eventual Consistency

👉 Events processed asynchronously → temporary inconsistency between services.

Example: Order created → inventory updated 2 seconds later → eventual consistency.

Eventual consistency | compensating actions

💡 Problem 80: Event Schema Design and Versioning

> INTERVIEWER: Your event schema needs to change. How do you handle event schema evolution without breaking consumers?

Question:

Solution:

Backward and forward compatibility are critical — use versioning and schema registries.

Here's how to handle event schema evolution 👇

1️⃣ Use Schema Registry

👉 Centralized registry stores and validates event schemas.

Example: Confluent Schema Registry with Avro → enforces schema compatibility.

Schema registry | Avro | Protobuf | JSON Schema

2️⃣ Add Fields Only (Backward Compatibility)

👉 Add optional fields → old consumers ignore new fields → no breaking changes.

Example: Add "email_verified" field to UserCreated → old consumers still work.

Backward compatibility | optional fields | default values

3️⃣ Never Remove Fields (Forward Compatibility)

👉 Mark fields as deprecated instead of removing → old events still valid.

Example: Deprecate "phone" field but keep it → new consumers handle its absence.

Forward compatibility | deprecation

4️⃣ Version Events Explicitly

👉 Include version in event name or metadata.

Example: UserCreatedV1, UserCreatedV2 → consumers handle specific versions.

Explicit versioning | parallel versions

5️⃣ Schema Evolution Rules

👉 Follow strict compatibility rules to prevent breaking changes.

Example: Only add fields with defaults, never change field types, never reorder fields.

Compatibility modes | schema registry enforcement

6️⃣ Dual Publishing During Migration

👉 Temporarily publish both old and new event versions during transition.

Example: Publish UserCreatedV1 and UserCreatedV2 → consumers migrate gradually.

Gradual migration | dual publishing

📚 Logging & Monitoring

💡 Problem 81: Event Ordering and Causality

INTERVIEWER: Two events for the same user arrive out of order. How do you handle this?

Question:

Solution:

Ordering is only guaranteed within partition — handle out-of-order events explicitly.

Here's how to handle event ordering 👇

1️⃣ Partition by Entity ID for Ordering

👉 All events for same entity go to same partition → ordering guaranteed.

Example: All events for user_id=123 in same Kafka partition → processed in order.

Kafka partitioning | ordering per key

2️⃣ Include Timestamp and Sequence Number

👉 Add timestamp and sequence number to detect out-of-order events.

Example: Event 1 at time T with sequence 5, Event 2 at time T-1 with sequence 4 → detect out-of-order.

Timestamp | sequence number | causality

3️⃣ Reorder Buffer for Late Events

👉 Buffer events temporarily to reorder before processing.

Example: Event seq=4 arrives → buffer → Event seq=3 arrives → process in order 3, 4.

Reorder buffer | windowing | complexity

4️⃣ Idempotent Processing

👉 Design consumers to handle duplicate and out-of-order events safely.

Example: "SetEmail" event is idempotent → processing twice or out-of-order doesn't cause issues.

Idempotency | state-based vs delta-based

5️⃣ Use Vector Clocks for Causality

👉 Track causal relationships between events across services.

Example: Vector clock detects that Event B happened after Event A → preserve causality.

Vector clocks | Lamport timestamps | distributed systems

6️⃣ Accept Out-of-Order for Non-Critical Events

👉 Some events don't require strict ordering — trade-off consistency for simplicity.

Example: Page view events → order doesn't matter → process as they arrive.

Eventual consistency | acceptable for analytics

💡 Problem 82: Implementing CQRS Pattern

> INTERVIEWER: Your app has complex reads but simple writes — how does CQRS help?

Question:

Solution:

CQRS separates read and write models for independent optimization and scaling.

1️⃣ Separate Read and Write Models

👉 Different data structures for queries vs commands.

Example: Write to normalized SQL; Read from denormalized Elasticsearch.

Model separation | optimization

2️⃣ Command Side (Write)

👉 Focus on business logic validation and consistency.

Example: CreateOrder command → validate inventory → save to SQL → publish event.

Commands write model

3️⃣ Query Side (Read)

👉 Optimized for fast, complex queries.

Example: Product search → query Elasticsearch with filters, facets, full-text search.

Queries read model

4️⃣ Eventual Consistency

👉 Read model updated asynchronously after write.

Example: Order created → event published → read model updated 100ms later.

Eventual consistency async updates

5️⃣ Event Sourcing Integration

👉 CQRS often paired with event sourcing.

Example: Store all OrderCreated, OrderShipped events → rebuild read model from events.

Event sourcing event replay

6️⃣ Multiple Read Models

👉 Different read models for different use cases.

Example: SQL for reporting, Elasticsearch for search, Redis for real-time dashboard.

Multiple models specialized views

7️⃣ When NOT to Use CQRS

👉 Adds complexity — only worth it for complex domains.

Example: Simple CRUD app → CQRS overkill → stick with single model.

Complexity trade-offs

💡 Problem 82: Implementing CQRS Pattern

INTERVIEWER: Your app has complex reads but simple writes — how does CQRS help?

Question:

Solution:

CQRS separates read and write models for independent optimization and scaling.

1️⃣ Separate Read and Write Models

👉 Different data structures for queries vs commands.

Example: Write to normalized SQL; Read from denormalized Elasticsearch.

Model separation optimization

2️⃣ Command Side (Write)

👉 Focus on business logic validation and consistency.

Example: CreateOrder command → validate inventory → save to SQL → publish event.

Commands write model

3️⃣ Query Side (Read)

👉 Optimized for fast, complex queries.

Example: Product search → query Elasticsearch with filters, facets, full-text search.

Queries read model

4️⃣ Eventual Consistency

👉 Read model updated asynchronously after write.

Example: Order created → event published → read model updated 100ms later.

Eventual consistency | async updates

5️⃣ Event Sourcing Integration

👉 CQRS often paired with event sourcing.

Example: Store all OrderCreated, OrderShipped events → rebuild read model from events.

Event sourcing | event replay

6️⃣ Multiple Read Models

👉 Different read models for different use cases.

Example: SQL for reporting, Elasticsearch for search, Redis for real-time dashboard.

Multiple models | specialized views

7️⃣ When NOT to Use CQRS

👉 Adds complexity — only worth it for complex domains.

Example: Simple CRUD app → CQRS overkill → stick with single model.

Complexity | trade-offs

Topic 29: Dead Letter Queues

💡 Problem 82: Implementing Dead Letter Queues

INTERVIEWER: Some messages repeatedly fail processing. How do you prevent them from blocking the queue?

Question:

Solution:

Dead letter queues capture failed messages for investigation and retry.

Here's how to implement dead letter queues 👇

1️⃣ What is a Dead Letter Queue?

👉 Separate queue for messages that failed processing after max retries.

Example: Message fails 3 times → moved to DLQ → manual investigation.

Error handling | poison messages | retry exhaustion

2️⃣ Move Messages After Max Retries

👉 Configure max retry count → exceeded → send to DLQ.

Example: RabbitMQ message rejected 5 times → moved to DLX → sent to DLQ.

x-max-retries | retry policy

3️⃣ Include Failure Metadata

👉 Add error details, retry count, timestamps to DLQ message.

Example: DLQ message includes original message + error reason + retry history.

Debugging info | failure context

4️⃣ Monitor DLQ Size

👉 Alert when DLQ grows beyond threshold → indicates systemic issues.

Example: DLQ has 10K messages → alarm → investigate root cause.

Monitoring | alerting | CloudWatch

5️⃣ Replay from DLQ After Fix

👉 Fix bug → replay DLQ messages to reprocess.

Example: Bug fixed → batch replay DLQ messages to main queue → reprocess successfully.

Replay mechanism | manual intervention

6️⃣ TTL on DLQ Messages

👉 Auto-expire old DLQ messages to prevent unbounded growth.

Example: DLQ messages older than 30 days → automatically deleted.

TTL | message expiration | cleanup

💡 Problem 83: Handling Poison Messages

> INTERVIEWER: A malformed message keeps getting retried and fails every time. How do you handle it?

Question:

Solution:

Poison messages can deadlock consumers — detect and route to DLQ quickly.

Here's how to handle poison messages 👇

1️⃣ Detect Poison Messages Early

👉 Validate message format before processing — fail fast on malformed messages.

Example: JSON parsing fails → immediately send to DLQ without retries.

Validation | fail fast | schema validation

2️⃣ Set Max Retries to Prevent Infinite Loops

👉 Limit retries to prevent poison message from blocking queue forever.

Example: Max 3 retries → after 3 failures → send to DLQ.

Retry budget | bounded retries

3️⃣ Use Exponential Backoff for Retries

👉 Increase retry delay exponentially to avoid overwhelming system.

Example: Retry after 1s → 2s → 4s → 8s → max retries → DLQ.

Exponential backoff | jitter

4️⃣ Separate Queue for Known Issues

👉 Route specific types of failures to dedicated queues for investigation.

Example: Validation errors → validation-errors queue, transient errors → retry queue.

Error classification | multiple DLQs

5️⃣ Add Circuit Breaker

👉 Stop processing after consecutive failures to prevent cascading failures.

Example: 10 consecutive failures → pause consumer → alert → investigate.

Circuit breaker | backpressure

6️⃣ Manual Inspection and Fix

👉 Have process for manually inspecting and fixing poison messages.

Example: DevOps reviews DLQ → fixes message format → replays to main queue.

Manual intervention | runbooks

📚 Kafka vs RabbitMQ

Topic 30: Kafka vs RabbitMQ

📚 Distributed Tracing

💡 Problem 84: When to Use Kafka vs RabbitMQ

INTERVIEWER: Should you use Kafka or RabbitMQ for your messaging needs?

Question:

Solution:

Kafka excels at high-throughput event streaming, RabbitMQ at flexible routing and task queues.

Here's when to use Kafka vs RabbitMQ 👇

1️⃣ Use Kafka for High-Throughput Event Streams

👉 Kafka handles millions of messages/sec with horizontal scaling.

Example: Log aggregation, clickstream analytics, activity tracking.

High throughput | append-only log | partitioned

2️⃣ Use RabbitMQ for Complex Routing

👉 RabbitMQ exchanges support topic, fanout, headers routing.

Example: Route notifications based on user preferences, delivery method, priority.

Flexible routing | exchanges | routing keys

3️⃣ Use Kafka for Event Replay and Reprocessing

👉 Kafka stores messages with configurable retention — consumers can rewind.

Example: Deploy new analytics service → replay last 7 days of events.

Message retention | replay | offset management

4️⃣ Use RabbitMQ for Task Queues and Work Distribution

👉 RabbitMQ competing consumers pattern ideal for distributing jobs.

Example: Process 10K background jobs across worker pool.

Work queues | competing consumers | acks

5️⃣ Use Kafka for Event Sourcing and CQRS

👉 Kafka's append-only log perfect for storing event history.

Example: Store all account transactions as events → rebuild state by replaying.

Event sourcing | immutable log | audit trail

6️⃣ Use RabbitMQ for Request/Reply Patterns

👉 RabbitMQ supports RPC pattern with reply queues.

Example: Send request to worker → worker responds to reply queue → synchronous feel.

RPC | request/reply | correlation ID

7️⃣ Use Kafka for Multi-Consumer Scenarios

👉 Multiple consumer groups independently consume all messages.

Example: Same event stream consumed by analytics, notifications, audit systems.

Fan-out | consumer groups | independent processing

💡 Problem 85: Kafka vs RabbitMQ Performance and Scalability

> INTERVIEWER: Which is faster and more scalable — Kafka or RabbitMQ?

Question:

Solution:

Kafka optimizes for throughput and horizontal scaling, RabbitMQ for low latency and routing flexibility.

Here's how Kafka and RabbitMQ compare 👇

1️⃣ Kafka Throughput

👉 Kafka handles millions of messages/sec with batch writes and zero-copy.

Example: Kafka can sustain 100K-1M msgs/sec per broker.

`High throughput` `batch processing` `sequential disk I/O`

2️⃣ RabbitMQ Latency

👉 RabbitMQ has lower per-message latency than Kafka.

Example: RabbitMQ <1ms latency, Kafka 5-10ms due to batching.

`Lower latency` `real-time delivery`

3️⃣ Kafka Horizontal Scaling

👉 Add partitions and brokers to scale linearly.

Example: 10 partitions → add 10 more → double throughput.

`Linear scaling` `partition-based`

4️⃣ RabbitMQ Vertical Scaling

👉 RabbitMQ scales primarily by adding nodes to cluster or using federation.

Example: Add more RabbitMQ nodes → shared queues → limited horizontal scaling.

`Cluster` `federation` `mirrored queues`

5️⃣ Kafka Message Retention

👉 Kafka stores messages for configured time regardless of consumption.

Example: Retain for 7 days → 10TB storage → high disk usage.

Storage cost | retention policy

6️⃣ RabbitMQ Memory Usage

👉 RabbitMQ stores messages in memory when possible → faster but memory-constrained.

Example: 1M queued messages → high RAM usage → potential memory pressure.

Memory-based | disk fallback | flow control

💡 Problem 86: Building Reliable ETL Pipelines

> INTERVIEWER: You need to sync 100M user records from MySQL to Elasticsearch nightly — how do you build that?

Question:

Solution:

ETL pipelines extract, transform, and load data reliably at scale.

1️⃣ Batch vs Streaming

👉 Batch for scheduled bulk loads, streaming for real-time sync.

Example: Nightly reports → batch; Real-time search → streaming (CDC).

Batch ETL | streaming ETL

2️⃣ Change Data Capture (CDC)

👉 Capture database changes in real-time.

Example: Debezium reads MySQL binlog → publishes changes to Kafka → Elasticsearch consumes.

CDC Debezium binlog

3️⃣ Idempotent Processing

👉 Handle duplicate records gracefully.

Example: Use primary key for upsert → re-processing same record doesn't break.

Idempotency upsert

4️⃣ Checkpointing

👉 Track progress to resume from failure.

Example: Process 10M records → checkpoint every 100K → crash at 5.2M → resume from 5.2M.

Checkpointing resumable

5️⃣ Data Validation

👉 Verify data quality before loading.

Example: Check for null emails, invalid dates → reject bad records → alert team.

Validation data quality

6️⃣ Error Handling & Dead Letter Queue

👉 Route failed records to DLQ for manual review.

Example: Record transformation fails → sent to DLQ → team fixes manually.

DLQ error handling

7️⃣ Monitoring & Alerting

👉 Track pipeline health, lag, and errors.

Example: Kafka lag > 1 hour → PagerDuty alert → investigate bottleneck.

Monitoring pipeline lag

💡 Problem 86: Building Reliable ETL Pipelines

INTERVIEWER: You need to sync 100M user records from MySQL to Elasticsearch nightly — how do you build that?

Question:

Solution:

ETL pipelines extract, transform, and load data reliably at scale.

1️⃣ Batch vs Streaming

👉 Batch for scheduled bulk loads, streaming for real-time sync.

Example: Nightly reports → batch; Real-time search → streaming (CDC).

Batch ETL | streaming ETL

2️⃣ Change Data Capture (CDC)

👉 Capture database changes in real-time.

Example: Debezium reads MySQL binlog → publishes changes to Kafka → Elasticsearch consumes.

CDC | Debezium | binlog

3️⃣ Idempotent Processing

👉 Handle duplicate records gracefully.

Example: Use primary key for upsert → re-processing same record doesn't break.

Idempotency | upsert

4️⃣ Checkpointing

👉 Track progress to resume from failure.

Example: Process 10M records → checkpoint every 100K → crash at 5.2M → resume from 5.2M.

Checkpointing resumable

5️⃣ Data Validation

👉 Verify data quality before loading.

Example: Check for null emails, invalid dates → reject bad records → alert team.

Validation data quality

6️⃣ Error Handling & Dead Letter Queue

👉 Route failed records to DLQ for manual review.

Example: Record transformation fails → sent to DLQ → team fixes manually.

DLQ error handling

7️⃣ Monitoring & Alerting

👉 Track pipeline health, lag, and errors.

Example: Kafka lag > 1 hour → PagerDuty alert → investigate bottleneck.

Monitoring pipeline lag

Topic 31: Message Deduplication

📚 Alerting & Incident Management

💡 Problem 86: Detecting Duplicate Messages

INTERVIEWER: Your message queue delivers the same message twice. How do you detect duplicates?

Question:

Solution:

Use message IDs and deduplication windows to identify duplicate messages.

Here's how to detect duplicate messages 👇

1️⃣ Unique Message ID

👉 Assign unique ID to each message — check if already processed.

Example: Generate UUID for message → store in processed set → reject if seen before.

Message ID | UUID | idempotency key

2️⃣ Content-Based Hashing

👉 Hash message content → use hash as dedup key.

Example: Hash(message payload) → check if hash exists in cache → skip if duplicate.

Content hash | MD5 | SHA-256

3️⃣ Sliding Window Deduplication

👉 Track message IDs in time window to detect recent duplicates.

Example: Keep last 5 minutes of message IDs in Redis → check new messages against set.

Time window | Redis SET with TTL | bounded memory

4️⃣ Database Unique Constraint

👉 Use database unique index to enforce deduplication.

Example: INSERT message_id → unique constraint violation → duplicate detected.

Unique index database enforcement

5️⃣ Sequence Numbers

👉 Producer assigns increasing sequence numbers — detect gaps and duplicates.

Example: Receive seq 1, 2, 3, 2 → detect duplicate seq 2.

Sequence number gap detection

6️⃣ Bloom Filter for Probabilistic Dedup

👉 Use Bloom filter for memory-efficient duplicate detection.

Example: Check Bloom filter → "definitely new" or "probably seen" → handle accordingly.

Bloom filter false positives memory efficient

💡 Problem 87: Implementing Idempotent Message Processing

INTERVIEWER: How do you ensure processing the same message twice doesn't cause problems?

Question:

Solution:

Design idempotent operations that produce same result regardless of how many times they're executed.

Here's how to implement idempotent processing 👇

1️⃣ Idempotency Key from Message ID

👉 Use message ID as idempotency key — store in database to prevent re-execution.

Example: Process payment with idempotency key → check if already processed → skip if yes.

Idempotency key dedupe table

2️⃣ Set-Based Operations (Naturally Idempotent)

👉 Operations like SET are inherently idempotent.

Example: SET user.email = "alice@example.com" → executing twice has same result.

`Set operations` `state-based`

3️⃣ Conditional Updates with Version/Timestamp

👉 Update only if record hasn't changed since read.

Example: UPDATE balance WHERE version=5 → fails if someone else updated → prevents duplicate updates.

`Optimistic locking` `compare-and-swap`

4️⃣ Database Transactions with Dedup Check

👉 Within transaction, check if message processed, then process and mark done.

Example: BEGIN TRANSACTION → check processed → insert record → mark processed → COMMIT.

`Atomic dedup` `transactional processing`

5️⃣ External System Idempotency

👉 Use external system's idempotency features.

Example: Stripe payment API with idempotency key → duplicate requests return same result.

`Third-party idempotency` `API support`

6️⃣ Avoid Non-Idempotent Operations

👉 Replace delta operations with set operations.

Example: Instead of "balance += 10", use "balance = old_balance + 10" with version check.

`State-based vs delta-based` `design for idempotency`

📚 Performance Optimization

💡 Problem 88: Handling Exactly-Once Delivery

INTERVIEWER: Can you achieve true exactly-once message delivery in distributed systems?

Question:

Solution:

Exactly-once is hard — combine at-least-once delivery with idempotent processing.

Here's how to achieve exactly-once semantics 👇

1️⃣ At-Least-Once Delivery + Idempotency

👉 Accept duplicates from message system, handle with idempotent processing.

Example: Kafka at-least-once + idempotency keys → effective exactly-once.

Practical approach | most common

2️⃣ Kafka Exactly-Once Semantics

👉 Kafka transactions ensure message produced and consumed exactly once.

Example: Kafka producer → transaction → consume from input topic + produce to output topic → commit → all-or-nothing.

Kafka transactions | enable.idempotence=true | transactional.id

---

3️⃣ Database Transaction with Message Commit

👉 Process message and commit offset in same database transaction.

Example: PostgreSQL transaction → process message → insert result → save offset → COMMIT.

Transactional inbox/outbox | atomicity

4️⃣ Two-Phase Commit Across Systems

👉 Coordinate commit across message system and database.

Example: 2PC → prepare database + message ack → commit both → distributed transaction.

Complex | blocking | rarely used in practice

5️⃣ Idempotent Producer

👉 Kafka producer deduplicates messages on broker side.

Example: Network retry sends duplicate → broker detects via sequence number → ignores duplicate.

Producer idempotence | Kafka feature

6️⃣ Accept "Effectively Once"

👉 In practice, at-least-once + idempotency is "exactly-once enough".

Example: Payment processed at most once due to idempotency key → effectively exactly-once.

Pragmatic approach | good enough for most use cases

Topic 32: REST API Best Practices

💡 Problem 89: Designing RESTful API Endpoints

> INTERVIEWER: How do you design clean, intuitive REST API endpoints?

Question:

Solution:

RESTful APIs use HTTP methods and resource-based URLs for predictable, intuitive interfaces.

Here's how to design RESTful API endpoints 👇

1️⃣ Use Nouns for Resources, Not Verbs

👉 Endpoints represent resources — use HTTP methods for actions.

Example: `/users` instead of `/getUsers`, `/users/123` instead of `/getUserById/123`.

Resource-based URLs | RESTful convention

2️⃣ Use HTTP Methods Correctly

👉 GET for reading, POST for creating, PUT/PATCH for updating, DELETE for deleting.

Example: GET `/users/123`, POST `/users`, PUT `/users/123`, DELETE `/users/123`.

HTTP verb semantics | idempotent methods

3️⃣ Use Hierarchical URLs for Relationships

👉 Nest resources to show relationships.

Example: `/users/123/orders` for user's orders, `/users/123/orders/456` for specific order.

Nested resources | relationship clarity

4️⃣ Use Plural Nouns for Collections

👉 Collection endpoints use plural form.

Example: `/users` for user collection, `/orders` for order collection.

Plural convention | consistency

5️⃣ Use Query Parameters for Filtering and Pagination

👉 Keep URLs clean, use query params for optional filters.

Example: `/users?status=active&page=2&limit=20` instead of `/users/active/page/2/limit/20`.

Query parameters | optional filters

6️⃣ Version Your API

👉 Include version in URL or header to allow breaking changes.

Example: `/v1/users` or `/v2/users` with different response formats.

API versioning | backward compatibility

💡 Problem 90: REST API Status Codes and Error Handling

> INTERVIEWER: What HTTP status codes should you return for different scenarios?

Question:

Solution:

HTTP status codes communicate what happened — use them correctly for clear API semantics.

Here's how to use HTTP status codes 👇

1️⃣ 2xx Success Codes

👉 200 OK for successful GET/PUT, 201 Created for POST, 204 No Content for DELETE.

Example: POST `/users` → 201 with created user in response body.

Success codes | semantic meaning

2️⃣ 4xx Client Error Codes

👉 400 Bad Request for invalid input, 401 Unauthorized for missing auth, 403 Forbidden for insufficient permissions, 404 Not Found for missing resource.

Example: POST `/users` with invalid email → 400 Bad Request with error details.

Client errors | validation failures

3️⃣ 5xx Server Error Codes

👉 500 Internal Server Error for unexpected errors, 503 Service Unavailable for temporary unavailability.

Example: Database connection fails → 500 with generic error message (don't expose internals).

Server errors | don't leak implementation details

4️⃣ Return Error Details in Response Body

👉 Include error code, message, and field-level details for debugging.

Example: `{"error": "VALIDATION_ERROR", "message": "Email is invalid", "fields": {"email": "Invalid format"}}`.

Error structure | debugging info

5️⃣ Use 422 Unprocessable Entity for Validation Errors

👉 Distinguish between malformed request (400) and validation failure (422).

Example: Valid JSON but email already exists → 422 Unprocessable Entity.

Semantic validation errors | GitHub convention

6️⃣ Use 429 Too Many Requests for Rate Limiting

👉 Return 429 with Retry-After header when rate limit exceeded.

Example: Client makes 1000 req/min → 429 with `Retry-After: 60`.

Rate limiting | backpressure

💡 Problem 91: REST API Pagination, Filtering, and Sorting

> INTERVIEWER: Your API returns 10,000 users. How do you handle large result sets?

Question:

Solution:

Pagination, filtering, and sorting make large datasets manageable for clients.

Here's how to handle large result sets 👇

1️⃣ Offset-Based Pagination

👉 Use `page` and `limit` or `offset` and `limit` query parameters.

Example: `/users?page=2&limit=20` returns users 21-40.

Simple | stateless | deep pagination slow

2️⃣ Cursor-Based Pagination

👉 Use cursor (encoded ID) to fetch next page — better for real-time data.

Example: `/users?cursor=abc123&limit=20` → response includes `next_cursor`.

Consistent results | handles insertions/deletions

3️⃣ Return Pagination Metadata

👉 Include total count, page info, and next/prev links in response.

Example: `{"data": [...], "total": 1000, "page": 2, "limit": 20, "next": "/users?page=3&limit=20"}`.

Discoverability | client convenience

4️⃣ Filtering with Query Parameters

👉 Allow filtering by resource fields.

Example: `/users?status=active&role=admin&created_after=2024-01-01`.

Flexible filtering | query parameters

5️⃣ Sorting with Query Parameters

👉 Use `sort` or `order_by` parameter with direction.

Example: `/users?sort=created_at:desc` or `/users?order_by=name&direction=asc`.

Sorting | ascending/descending

6️⃣ Limit Maximum Page Size

👉 Cap `limit` parameter to prevent expensive queries.

Example: Client requests `limit=10000` → cap at 100 → return max 100 results.

Resource protection   DoS prevention

📚 Database Indexing

💡 Problem 92: Optimizing API Response Time from 2 Seconds to 200ms

INTERVIEWER: Our API endpoint takes 2 seconds to respond. How would you improve it to 200ms?

Question:

Solution:

API performance optimization requires identifying bottlenecks and applying targeted improvements. Here's how to optimize API response time 👇

1️⃣ Identify the Bottleneck First

👉 Profile the request to find where time is spent — database, external APIs, or computation.

Example: Use APM tools (New Relic, Datadog) or simple logging to measure each operation's duration.

Measure first   optimize later   avoid premature optimization

2️⃣ Database Query Optimization

👉 Add indexes on frequently queried columns, optimize N+1 queries with eager loading, use EXPLAIN to analyze query plans.

Example: Query takes 1.8s → add index on user_id → query takes 50ms.

Indexing | query optimization | database performance

3️⃣ Implement Caching

👉 Cache frequently accessed data in Redis/Memcached to avoid repeated database queries.

Example: Product catalog queried 1000x/sec → cache in Redis with 5-min TTL → 99% of requests served from cache (5ms instead of 500ms).

Redis caching | TTL strategy | cache-aside pattern

4️⃣ Use Database Connection Pooling

👉 Reuse existing database connections instead of creating new ones for each request.

Example: Connection creation takes 100ms → connection pool provides instant connection → save 100ms per request.

Connection pooling | reduce latency | resource reuse

5️⃣ Optimize External API Calls

👉 Run external API calls in parallel, implement timeouts, use circuit breakers, or cache responses.

Example: 3 sequential API calls (500ms each = 1.5s total) → run in parallel → 500ms total.

Parallel execution | async/await | Promise.all

6️⃣ Reduce Payload Size

👉 Return only necessary fields, compress responses with gzip/brotli, paginate large result sets.

Example: 5MB JSON response → remove unused fields → 500KB → enable gzip → 50KB → network transfer time reduced.

Payload optimization | compression | selective fields

7️⃣ Asynchronous Processing

👉 Move heavy operations to background jobs and return response immediately.

Example: Order placement with email notification → accept order (200ms) → send email asynchronously via queue.

Background jobs | async processing | immediate response

8️⃣ Use Read Replicas

👉 Route read-heavy operations to read replicas to reduce load on primary database.

Example: Dashboard query hits primary DB → move to read replica → reduce primary load → faster writes.

Read replicas | database scaling | load distribution

9️⃣ Implement CDN for Static Assets

👉 Serve images, CSS, JS from CDN edge locations closer to users.

Example: Loading 10 images from origin server (200ms each) → CDN edge cache (10ms each) → save 1.9s.

CDN | edge caching | geographic distribution

👉 Remove unnecessary computations, optimize algorithms, use efficient data structures.

Example: $O(n^2)$ loop processing 10k records → optimize to $O(n)$ → reduce from 1s to 10ms.

Algorithm optimization | computational efficiency

1️⃣1️⃣ Monitor and Measure Continuously

👉 Set up performance monitoring, track P50/P95/P99 latencies, set up alerts.

Example: Track API response time → set alert for P95 > 300ms → proactively fix degradations.

APM | observability | performance metrics

Topic 33: API Versioning

💡 Problem 93: API Versioning Strategies

INTERVIEWER: You need to make breaking changes to your API. How do you version it?

Question:

Solution:

API versioning allows backward-incompatible changes without breaking existing clients.

Here are API versioning strategies 👇

1️⃣ URL Path Versioning

👉 Include version in URL path.

Example: `/v1/users` vs `/v2/users` — clear and visible.

Most common | easy to route | URL pollution

2️⃣ Header Versioning

👉 Specify version in custom header or Accept header.

Example: `X-API-Version: 2` or `Accept: application/vnd.myapi.v2+json`.

Clean URLs | less discoverable | harder to test

3️⃣ Query Parameter Versioning

👉 Pass version as query parameter.

Example: `/users?version=2` — simple but not RESTful.

Easy to implement | non-standard | easily forgotten

4️⃣ Maintain Multiple Versions Simultaneously

👉 Support old versions for transition period, deprecate gradually.

Example: Support v1 and v2 for 6 months → notify clients → sunset v1.

Backward compatibility | maintenance burden

5️⃣ Deprecation Headers

👉 Warn clients about deprecated endpoints with headers.

Example: `Deprecation: true`, `Sunset: 2024-12-31`, `Link: <https://api.example.com/v2/users>; rel="successor"`.

RFC 8594 | graceful migration

6️⃣ Semantic Versioning for Major Changes

👉 Only increment major version for breaking changes.

Example: v1 → v2 when response structure changes, v1.1 for new optional fields.

Semantic versioning | clear expectations

💡 Problem 94: Managing API Breaking Changes

> INTERVIEWER: What constitutes a breaking change in an API?

Question:

Solution:

Breaking changes break existing clients — distinguish from backward-compatible changes.

Here's what counts as a breaking change 👇

1️⃣ Breaking Changes Requiring New Version

👉 Removing fields, changing field types, changing URL structure, changing required parameters.

Example: Remove `phone` field from response → breaks clients expecting it.

Major version bump | client impact

2️⃣ Non-Breaking Changes

👉 Adding optional fields, adding new endpoints, adding optional query parameters.

Example: Add new `email_verified` field → old clients ignore it.

Backward compatible | minor version

3️⃣ Additive Changes Are Safe

👉 Clients should ignore unknown fields — adding fields is safe.

Example: Response adds `created_at` field → clients using `name` and `email` still work.

Robustness principle | tolerant readers

4️⃣ Deprecate Before Removing

👉 Mark field/endpoint as deprecated, give clients time to migrate.

Example: Mark `phone` deprecated → wait 6 months → remove in v2.

Gradual deprecation | migration window

5️⃣ Support Old and New Formats Temporarily

👉 Accept both formats during transition, return new format.

Example: Accept `user_id` and `userId` during migration → eventually only `userId`.

Dual support | transition period

📚 Authentication (OAuth 2.0, JWT)

Topic 34: Authentication (OAuth 2.0, JWT)

📚 N+1 Query Problem

💡 Problem 95: Understanding OAuth 2.0 Flow

INTERVIEWER: Explain how OAuth 2.0 works for third-party authentication.

Question:

Solution:

OAuth 2.0 delegates authentication to authorization server — users grant limited access without sharing passwords.

Here's how OAuth 2.0 works 👇

1️⃣ Authorization Code Flow (Most Secure)

👉 User redirected to auth server → grants permissions → auth code returned → exchange code for access token.

Example: "Login with Google" → redirect to Google → user approves → code returned → backend exchanges for token.

Server-side apps | PKCE extension for mobile

2️⃣ Access Token for API Requests

👉 Client includes access token in Authorization header for API calls.

Example: `Authorization: Bearer eyJhbGc...` → API validates token → grants access.

Bearer token short-lived

3️⃣ Refresh Token for Token Renewal

👉 Long-lived refresh token exchanges for new access token without re-authentication.

Example: Access token expires in 1 hour → use refresh token to get new access token.

Refresh token long-lived secure storage

4️⃣ Scopes for Permission Granularity

👉 Limit access to specific resources/actions.

Example: Request `scope=read:user,write:repos` → token only allows reading user info and writing repos.

Least privilege scope-based access

5️⃣ Client Credentials Flow for Service-to-Service

👉 Backend service authenticates with client ID and secret to get token.

Example: Microservice A calls microservice B → uses client credentials → no user involved.

Machine-to-machine service accounts

6️⃣ Implicit Flow (Deprecated)

👉 Access token returned directly in redirect URL — less secure, avoid for new apps.

Example: Single-page app → token in URL fragment → token exposure risk.

Legacy use Authorization Code with PKCE instead

💡 Problem 96: JWT (JSON Web Tokens) Structure and Validation

> INTERVIEWER: How do you implement stateless authentication with JWTs?

Question:

Solution:

JWTs encode claims in signed tokens — servers validate without database lookups.

Here's how JWTs work 👇

1️⃣ JWT Structure: Header.Payload.Signature

👉 Header has algorithm, payload has claims, signature proves authenticity.

Example: `eyJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjoxMjN9.signature` → base64-encoded JSON.

Three parts | base64url encoded

2️⃣ Claims in Payload

👉 Standard claims: `sub` (subject/user ID), `exp` (expiration), `iat` (issued at), `iss` (issuer).

Example: `{"sub": "user123", "exp": 1735689600, "role": "admin"}`.

JWT claims | application-specific data

3️⃣ Signature Verification

👉 Server verifies signature using secret key (HMAC) or public key (RSA).

Example: Verify signature with secret → if valid, trust claims → if invalid, reject token.

HMAC-SHA256 | RSA | signature validation

4️⃣ Stateless Authentication

👉 No database lookup needed — all info in token.

Example: JWT contains user_id and role → server reads from token → validates signature → grants access.

Stateless | horizontal scaling | no session storage

5️⃣ Short Expiration Times

👉 Set short expiration to limit damage if token stolen.

Example: Access token expires in 15 minutes, refresh token in 7 days.

exp claim | security vs usability

6️⃣ Can't Revoke JWTs Easily

👉 JWTs valid until expiration — use short expiration and refresh tokens for revocation.

Example: User logs out → can't invalidate JWT → short expiration limits exposure.

Revocation challenge | blacklist or short TTL

📚 Connection Pooling

💡 Problem 97: JWT Security Best Practices

> INTERVIEWER: What are the security risks of using JWTs and how do you mitigate them?

Question:

Solution:

JWTs have specific security considerations — follow best practices to avoid vulnerabilities.

Here are JWT security best practices 👇

1️⃣ Always Verify Signature

👉 Never trust JWT without signature verification.

Example: Attacker modifies payload → changes `role: user` to `role: admin` → signature invalid → reject.

Signature verification | cryptographic integrity

2️⃣ Use Strong Signing Algorithm

👉 Use RS256 (RSA) or HS256 (HMAC-SHA256), avoid `none` algorithm.

Example: Attacker sets `alg: none` → no signature → server must reject `none` algorithm.

Algorithm whitelist | reject `none`

3️⃣ Validate Claims (exp, iat, iss, aud)

👉 Check expiration, issuer, audience to prevent misuse.

Example: Token expired 1 hour ago → check `exp` claim → reject expired token.

Claim validation | expiration check

4️⃣ Store Secrets Securely

👉 Use environment variables or secret management for signing keys.

Example: Store JWT secret in AWS Secrets Manager, not hardcoded in code.

Secret management | KMS | Vault

5️⃣ Use HTTPS Only

👉 JWTs sent over HTTP can be intercepted.

Example: Man-in-the-middle attack intercepts JWT → attacker impersonates user.

TLS/SSL | encrypted transport

6️⃣ Don't Store Sensitive Data in JWT

👉 Payload is base64-encoded, not encrypted — anyone can read it.

Example: Don't put SSN, credit card in JWT → only store user_id, role.

JWT is signed | not encrypted | public data only

💡 Problem 98: Session-Based vs Token-Based Authentication

INTERVIEWER: Should you use session-based or token-based authentication?

Question:

Solution:

Sessions store state on server, tokens are stateless — each has trade-offs.

Here's when to use sessions vs tokens 👇

1️⃣ Session-Based Authentication

👉 Server stores session data, client gets session ID cookie.

Example: User logs in → server creates session in Redis → returns session ID cookie → client sends cookie on each request.

Server-side state | easy revocation

2️⃣ Token-Based Authentication (JWT)

👉 Client stores token, server validates signature — stateless.

Example: User logs in → server issues JWT → client stores in localStorage → sends in Authorization header.

Stateless | scales horizontally | no session storage

3️⃣ Use Sessions for Traditional Web Apps

👉 Server-rendered apps with cookies.

Example: Server-side rendered pages with Express sessions.

HttpOnly cookies | CSRF protection needed

4️⃣ Use Tokens for APIs and SPAs

👉 Mobile apps, single-page apps, microservices.

Example: React app calls API → sends JWT in Authorization header.

CORS-friendly mobile-friendly stateless

5️⃣ Session Revocation is Easier

👉 Delete session from storage → user logged out immediately.

Example: User logs out → delete Redis session → subsequent requests denied.

Immediate revocation centralized control

6️⃣ Tokens Scale Better

👉 No shared session storage needed across servers.

Example: 10 API servers → no session synchronization → validate JWT locally.

Horizontal scaling no sticky sessions

Topic 35: Authorization (RBAC)

📚 Batch Processing

💡 Problem 99: Implementing Role-Based Access Control (RBAC)

> INTERVIEWER: How do you implement authorization where different users have different permissions?

Question:

Solution:

RBAC assigns roles to users, roles have permissions — check permissions before granting access.

Here's how to implement RBAC 👇

1️⃣ Define Roles

👉 Create roles representing job functions.

Example: Roles: `admin`, `editor`, `viewer` → different permission levels.

Role definition | job-based

2️⃣ Assign Permissions to Roles

👉 Roles have specific permissions to perform actions.

Example: `admin` can `create:users`, `delete:users` → `editor` can `edit:posts` → `viewer` can `read:posts`.

Permission mapping | role-permission relationship

3️⃣ Assign Roles to Users

👉 Users get one or more roles.

Example: User Alice has role `admin` → Bob has role `editor`.

User-role assignment | database relationship

4️⃣ Check Permissions in Code

👉 Before action, verify user has required permission.

Example: User requests DELETE `/users/123` → check if user has `delete:users` permission → allow or deny.

Permission check | authorization middleware

5️⃣ Store Roles in JWT or Session

👉 Include user's roles in authentication token.

Example: JWT payload `{"user_id": 123, "roles": ["admin", "editor"]}`.

Stateless authorization | claims-based

6️⃣ Database Schema for RBAC

👉 Tables: users, roles, permissions, user_roles, role_permissions.

Example: User → user_roles → Role → role_permissions → Permission.

Many-to-many relationships | normalized schema

💡 Problem 100: Attribute-Based Access Control (ABAC) and Fine-Grained Authorization

> INTERVIEWER: RBAC is too coarse-grained. How do you implement fine-grained authorization?

Question:

Solution:

ABAC uses attributes (user, resource, environment) for dynamic access decisions.

Here's how to implement fine-grained authorization 👇

1️⃣ ABAC with Policies

👉 Define policies based on user attributes, resource attributes, and context.

Example: User can edit post if `user.id == post.author_id OR user.role == admin`.

Policy-based | dynamic rules

2️⃣ Resource-Level Permissions

👉 Check ownership or relationship to resource.

Example: User can DELETE `/posts/123` only if they own the post.

Ownership check | resource-specific

3️⃣ Use Authorization Library

👉 Libraries like Casbin, OPA (Open Policy Agent) for policy enforcement.

Example: Define policies in Rego (OPA) → query OPA for access decision.

Policy engine | centralized authorization

4️⃣ Row-Level Security in Database

👉 Database enforces access control at row level.

Example: PostgreSQL RLS → `ALTER TABLE posts ENABLE ROW LEVEL SECURITY` → users only see their own posts.

Database-level security | PostgreSQL RLS

5️⃣ Context-Based Authorization

👉 Consider time, location, device for access decisions.

Example: Admin can delete users only from office IP range.

Environmental attributes | contextual policies

6️⃣ Combine RBAC and ABAC

👉 Use roles for general permissions, attributes for fine-grained control.

Example: User has `editor` role → can edit posts → but only their own posts (attribute check).

Hybrid approach | layered authorization

Topic 36: API Security

📚 Stream Processing

💡 Problem 101: Protecting APIs from Common Attacks

INTERVIEWER: What are the most common API security vulnerabilities and how do you prevent them?

Question:

Solution:

APIs face threats like injection, broken auth, excessive data exposure — follow OWASP API Security Top 10.

Here's how to protect APIs 👇

1️⃣ SQL Injection Prevention

👉 Use parameterized queries, never concatenate user input into SQL.

Example: `SELECT * FROM users WHERE id = ?` with parameter, not `SELECT * FROM users WHERE id = ${userId}`.

Prepared statements | ORM | input validation

2️⃣ Broken Authentication and Authorization

👉 Validate JWT signature, check expiration, enforce permissions.

Example: Every API request → verify JWT → check user has required permission.

Token validation | RBAC | principle of least privilege

3️⃣ Excessive Data Exposure

👉 Return only necessary fields, don't expose internal IDs or sensitive data.

Example: `/users/me` returns only `name`, `email` → not `password_hash`, `ssn`.

Data minimization | API response filtering

4️⃣ Rate Limiting

👉 Limit requests per user/IP to prevent abuse.

Example: Max 100 requests per minute per API key → return 429 if exceeded.

Rate limiting | DDoS protection | API gateway

5️⃣ Input Validation

👉 Validate all inputs against expected format, type, range.

Example: Email field → validate regex → reject if invalid.

Schema validation | sanitization | joi/yup libraries

6️⃣ HTTPS Only

👉 Enforce TLS for all API traffic to prevent eavesdropping.

Example: Redirect HTTP to HTTPS, use HSTS header.

TLS/SSL | encrypted transport | HSTS

7️⃣ CORS Configuration

👉 Configure CORS to allow only trusted origins.

Example: `Access-Control-Allow-Origin: https://trusted-app.com` → not `*`.

CORS | origin whitelist | preflight requests

💡 Problem 102: API Rate Limiting and Throttling

INTERVIEWER: How do you prevent API abuse and ensure fair usage?

Question:

Solution:

Rate limiting controls request frequency per client — protects infrastructure and ensures fair access.

Here's how to implement API rate limiting 👇

1️⃣ Fixed Window Rate Limiting

👉 Allow N requests per time window.

Example: 100 requests per minute → counter resets every minute at :00.

Simple | burst at window boundary

2️⃣ Sliding Window Rate Limiting

👉 Track requests in rolling time window.

Example: 100 requests per 60-second window → smooth enforcement.

Accurate | more complex | Redis sorted set

3️⃣ Token Bucket Algorithm

👉 Bucket refills at fixed rate, consume token per request.

Example: Bucket capacity 100, refill 10/sec → allows bursts up to 100, then 10 req/sec.

Burst handling | smooth rate

4️⃣ Return Rate Limit Headers

👉 Tell clients their rate limit status.

Example: `X-RateLimit-Limit: 100`, `X-RateLimit-Remaining: 42`, `X-RateLimit-Reset: 1735689600`.

Transparency | client awareness

5️⃣ Different Limits for Different Tiers

👉 Free tier gets 100 req/min, paid tier gets 10K req/min.

Example: Check API key → determine tier → apply corresponding limit.

Tiered pricing API key-based

6️⃣ Return 429 with Retry-After

👉 Return 429 Too Many Requests with retry delay.

Example: `429 Too Many Requests`, `Retry-After: 60` → client waits 60 seconds.

Backpressure client retry guidance

📚 Data Serialization

💡 Problem 103: API Keys and Secret Management

INTERVIEWER: How do you manage API keys and secrets securely?

Question:

Solution:

API keys authenticate clients — store securely, rotate regularly, limit scope.

Here's how to manage API keys securely 👇

1️⃣ Hash API Keys Before Storing

👉 Store hashed version in database like passwords.

Example: Generate API key → show user once → store SHA-256 hash → validate by hashing incoming key.

Hash storage | prevent database leak exposure

2️⃣ Use Secret Management Services

👉 Store secrets in dedicated secret managers, not environment variables or code.

Example: AWS Secrets Manager, HashiCorp Vault, Azure Key Vault.

Centralized secrets | rotation | access control

3️⃣ Rotate API Keys Regularly

👉 Force rotation or allow users to rotate keys.

Example: API keys expire after 90 days → user must generate new key.

Key rotation | limit exposure window

4️⃣ Scope API Keys to Specific Permissions

👉 Each key has limited permissions, not full access.

Example: Read-only API key for analytics, write key for data ingestion.

Least privilege | scoped keys

5️⃣ Rate Limit by API Key

👉 Track usage per API key to enforce quotas.

Example: API key `abc123` → 100K requests/day → deny after limit.

Per-key rate limiting | quota enforcement

6️⃣ Monitor API Key Usage

👉 Log and alert on suspicious activity.

Example: API key suddenly used from different country → alert security team.

Anomaly detection | audit logs

💡 Problem 104: Managing Secrets Securely

INTERVIEWER: Where do you store database passwords and API keys in production?

Question:

Solution:

Secrets must be encrypted, rotated, and never hardcoded or committed to version control.

1️⃣ Never Commit Secrets to Git

👉 Use .env files or secret managers, not hardcoded values.

Example: GitHub secret scanning detects committed AWS keys → auto-revoked → service breaks.

Git secrets | security breach

2️⃣ Secret Management Services

👉 Use dedicated services for secret storage.

Example: AWS Secrets Manager, HashiCorp Vault, Azure Key Vault → encrypted storage.

Secrets Manager | Vault | Key Vault

3️⃣ Encryption at Rest & In Transit

👉 Secrets encrypted in storage and during transmission.

Example: Secrets Manager encrypts with KMS → accessed via TLS → end-to-end encryption.

Encryption | KMS

4️⃣ Secret Rotation

👉 Automatically rotate secrets periodically.

Example: Database password rotated every 90 days → Secrets Manager updates app automatically.

Rotation automated updates

5️⃣ Least Privilege Access

👉 Only authorized services can access specific secrets.

Example: Payment service can access Stripe key, not database password.

IAM least privilege

6️⃣ Audit Logging

👉 Track all secret access for compliance.

Example: CloudTrail logs every Secrets Manager access → audit who accessed what when.

Audit logs compliance

7️⃣ Environment-Specific Secrets

👉 Use different secrets for dev, staging, production.

Example: Dev uses dev database credentials → limits blast radius of leaked secrets.

Environment separation blast radius

💡 Problem 104: Managing Secrets Securely

> INTERVIEWER: Where do you store database passwords and API keys in production?

Question:

Solution:

Secrets must be encrypted, rotated, and never hardcoded or committed to version control.

1️⃣ Never Commit Secrets to Git

👉 Use .env files or secret managers, not hardcoded values.

Example: GitHub secret scanning detects committed AWS keys → auto-revoked → service breaks.

Git secrets | security breach

2️⃣ Secret Management Services

👉 Use dedicated services for secret storage.

Example: AWS Secrets Manager, HashiCorp Vault, Azure Key Vault → encrypted storage.

Secrets Manager | Vault | Key Vault

3️⃣ Encryption at Rest & In Transit

👉 Secrets encrypted in storage and during transmission.

Example: Secrets Manager encrypts with KMS → accessed via TLS → end-to-end encryption.

Encryption | KMS

4️⃣ Secret Rotation

👉 Automatically rotate secrets periodically.

Example: Database password rotated every 90 days → Secrets Manager updates app automatically.

Rotation | automated updates

5️⃣ Least Privilege Access

👉 Only authorized services can access specific secrets.

Example: Payment service can access Stripe key, not database password.

IAM | least privilege

6️⃣ Audit Logging

👉 Track all secret access for compliance.

Example: CloudTrail logs every Secrets Manager access → audit who accessed what when.

Audit logs compliance

7️⃣ Environment-Specific Secrets

👉 Use different secrets for dev, staging, production.

Example: Dev uses dev database credentials → limits blast radius of leaked secrets.

Environment separation blast radius

💡 Problem 104: Preventing API Injection and Data Leakage

INTERVIEWER: How do you prevent attackers from exploiting your API to access unauthorized data?

Question:

Solution:

Injection attacks and data leakage exploit poor input validation and authorization — validate, sanitize, and enforce access control.

Here's how to prevent API injection and leakage 👇

1️⃣ Parameterized Queries for SQL Injection

👉 Never concatenate user input into queries.

Example: `db.query('SELECT * FROM users WHERE id = ?', [userId])` → not `'SELECT * FROM users WHERE id = ' + userId`.

Prepared statements ORM

2️⃣ Validate Input Against Schema

👉 Reject requests with unexpected fields or types.

Example: Expect `{name: string, age: number}` → reject `{name: string, age: "drop table"}`.

JSON schema validation | joi | yup

3️⃣ NoSQL Injection Prevention

👉 Validate types and sanitize operators in NoSQL queries.

Example: MongoDB `{username: req.body.username}` → attacker sends `{username: {$ne: null}}` → bypass auth.

Type checking | sanitize operators

4️⃣ Check Resource Ownership

👉 Verify user owns resource before allowing access.

Example: User requests `/orders/123` → check `orders.user_id == current_user.id` → deny if mismatch.

Authorization check | ownership validation

5️⃣ Filter Response Data

👉 Don't return entire database objects — select specific fields.

Example: Return `{id, name, email}` → not entire user object with `password_hash`, `internal_notes`.

Data minimization | response filtering

6️⃣ Prevent Mass Assignment

👉 Don't blindly assign request body to database model.

Example: User sends `{name: "Alice", role: "admin"}` → only update allowed fields → ignore `role`.

Whitelist allowed fields | mass assignment protection

💡 Problem 105: Preventing SQL Injection Attacks

INTERVIEWER: A user enters in login field — how do you protect your database?

Question:`' OR '1'='1`

Solution:

SQL injection is a top security risk that can expose or destroy your entire database.

1️⃣ Parameterized Queries (Prepared Statements)

👉 Use placeholders instead of string concatenation.

*Example: SELECT * FROM users WHERE email = ? → database treats input as data, not code.*

Prepared statements | safe queries

2️⃣ ORM/Query Builders

👉 Use libraries that automatically parameterize queries.

Example: User.find({email: input}) in Sequelize → auto-escapes input.

ORM | Sequelize | TypeORM

3️⃣ Input Validation

👉 Validate and sanitize all user inputs.

Example: Email field → regex validation → reject if contains SQL keywords.

Input validation | sanitization

4️⃣ Least Privilege Database User

👉 App database user should have minimal permissions.

Example: App user can only SELECT/INSERT users table, not DROP or ALTER.

Least privilege | database permissions

5️⃣ Stored Procedures

👉 Pre-defined database procedures with fixed logic.

Example: CALL get_user(email) → prevents dynamic SQL injection.

Stored procedures encapsulation

6️⃣ WAF (Web Application Firewall)

👉 Filter malicious SQL patterns at network level.

Example: AWS WAF blocks requests with SQL keywords → prevents attacks before reaching app.

WAF CloudFlare ModSecurity

7️⃣ Database Activity Monitoring

👉 Alert on suspicious database queries.

Example: Query with 'DROP TABLE' → instant PagerDuty alert → block IP.

Monitoring anomaly detection

💡 Problem 105: Preventing SQL Injection Attacks

> INTERVIEWER: A user enters in login field — how do you protect your database?

Question:`' OR '1'='1`

Solution:

SQL injection is a top security risk that can expose or destroy your entire database.

1️⃣ Parameterized Queries (Prepared Statements)

👉 Use placeholders instead of string concatenation.

*Example: SELECT * FROM users WHERE email = ? → database treats input as data, not code.*

Prepared statements | safe queries

2️⃣ ORM/Query Builders

👉 Use libraries that automatically parameterize queries.

Example: User.find({email: input}) in Sequelize → auto-escapes input.

ORM | Sequelize | TypeORM

3️⃣ Input Validation

👉 Validate and sanitize all user inputs.

Example: Email field → regex validation → reject if contains SQL keywords.

Input validation | sanitization

4️⃣ Least Privilege Database User

👉 App database user should have minimal permissions.

Example: App user can only SELECT/INSERT users table, not DROP or ALTER.

Least privilege | database permissions

5️⃣ Stored Procedures

👉 Pre-defined database procedures with fixed logic.

Example: CALL get_user(email) → prevents dynamic SQL injection.

Stored procedures | encapsulation

6️⃣ WAF (Web Application Firewall)

👉 Filter malicious SQL patterns at network level.

Example: AWS WAF blocks requests with SQL keywords → prevents attacks before reaching app.

WAF | CloudFlare | ModSecurity

7️⃣ Database Activity Monitoring

👉 Alert on suspicious database queries.

Example: Query with 'DROP TABLE' → instant PagerDuty alert → block IP.

Monitoring | anomaly detection

Topic 37: Microservices Communication

💡 Problem 105: Synchronous vs Asynchronous Communication

INTERVIEWER: When should microservices communicate synchronously vs asynchronously?

Question:

Solution:

Synchronous communication couples services tightly, asynchronous decouples — choose based on requirements.

Here's when to use sync vs async communication 👇

1️⃣ Synchronous Communication (REST/gRPC)

👉 Direct request/response between services — immediate response needed.

Example: User service calls payment service → waits for payment confirmation → returns order status.

REST API | gRPC | tight coupling | latency dependency

2️⃣ Asynchronous Communication (Message Queues/Events)

👉 Services publish events, others consume asynchronously — fire-and-forget.

Example: Order created → publish OrderCreated event → inventory, notification, analytics services react independently.

Kafka RabbitMQ loose coupling eventual consistency

3️⃣ Use Sync for Critical Path Operations

👉 When user waits for immediate response and consistency required.

Example: Login API → validate credentials synchronously → return success/failure.

Real-time response user-facing operations

4️⃣ Use Async for Non-Critical Background Tasks

👉 When operation can complete later without blocking user.

Example: User uploads photo → return success immediately → resize image asynchronously in background.

Background jobs worker queues improved UX

5️⃣ Sync Creates Cascade Failures

👉 If downstream service fails, entire request chain fails.

Example: Service A calls B calls C → C is down → entire request fails.

Cascading failures circuit breaker needed

6️⃣ Async Provides Better Fault Tolerance

👉 Services continue operating even if consumers are down.

Example: Order service publishes events → notification service down → events queued → processed when service recovers.

Resilience eventual consistency retry mechanisms

💡 Problem 4: Cross-Service Joins Without Shared Database

INTERVIEWER: Each microservice uses its own DB — so how do you handle cross-service joins?

Question:

Solution:

Distributed data requires denormalization, API composition, or event-driven synchronization.

1️⃣ API Composition Pattern

👉 Application layer queries multiple services and combines results.

Example: Order details = call Order service + call User service + call Product service → merge in app.

API Gateway composition | BFF pattern

2️⃣ Denormalization / Data Duplication

👉 Store frequently joined data in same service database.

Example: Order service stores {order_id, user_name, user_email} even though Users service owns user data.

Controlled duplication | eventual consistency

3️⃣ CQRS with Read Models

👉 Build specialized read databases optimized for specific queries.

Example: OrderHistory service subscribes to events → builds denormalized view combining orders + users + products.

CQRS | read models | materialized views

4️⃣ Event-Driven Synchronization

👉 Services publish events when data changes, others subscribe and update local copies.

Example: User service publishes UserUpdated event → Order service updates cached user data.

Event sourcing | domain events | Kafka

5️⃣ Backend for Frontend (BFF)

👉 Create service that aggregates data from multiple services for specific UI needs.

Example: Mobile BFF queries 5 services → returns single optimized response to mobile app.

BFF pattern | aggregation layer

6️⃣ GraphQL Federation

👉 Use GraphQL to query across multiple services transparently.

Example: Single GraphQL query fetches order (Order service) + user (User service) + products (Product service).

Apollo Federation | GraphQL stitching

7️⃣ Avoid Cross-Service Joins

👉 Redesign bounded contexts to minimize need for joins.

Example: Instead of joining Orders + Users, make Orders service store all needed user data at order time.

Domain-driven design | bounded contexts

💡 Problem 4: Cross-Service Joins Without Shared Database

> INTERVIEWER: Each microservice uses its own DB — so how do you handle cross-service joins?

Question:

Solution:

Distributed data requires denormalization, API composition, or event-driven synchronization.

1️⃣ API Composition Pattern

👉 Application layer queries multiple services and combines results.

Example: Order details = call Order service + call User service + call Product service → merge in app.

API Gateway composition | BFF pattern

2️⃣ Denormalization / Data Duplication

👉 Store frequently joined data in same service database.

Example: Order service stores {order_id, user_name, user_email} even though Users service owns user data.

Controlled duplication | eventual consistency

3️⃣ CQRS with Read Models

👉 Build specialized read databases optimized for specific queries.

Example: OrderHistory service subscribes to events → builds denormalized view combining orders + users + products.

CQRS | read models | materialized views

4️⃣ Event-Driven Synchronization

👉 Services publish events when data changes, others subscribe and update local copies.

Example: User service publishes UserUpdated event → Order service updates cached user data.

Event sourcing | domain events | Kafka

5️⃣ Backend for Frontend (BFF)

👉 Create service that aggregates data from multiple services for specific UI needs.

Example: Mobile BFF queries 5 services → returns single optimized response to mobile app.

BFF pattern | aggregation layer

6️⃣ GraphQL Federation

👉 Use GraphQL to query across multiple services transparently.

Example: Single GraphQL query fetches order (Order service) + user (User service) + products (Product service).

Apollo Federation GraphQL stitching

7️⃣ Avoid Cross-Service Joins

👉 Redesign bounded contexts to minimize need for joins.

Example: Instead of joining Orders + Users, make Orders service store all needed user data at order time.

Domain-driven design bounded contexts

📚 Consensus Algorithms

💡 Problem 106: Service Discovery and Load Balancing

INTERVIEWER: How do microservices find and connect to each other?

Question:

Solution:

Service discovery enables dynamic routing — services register and discover each other at runtime.

Here's how service discovery works 👇

1️⃣ Client-Side Service Discovery

👉 Client queries service registry to find available instances.

Example: Service A queries Consul → gets list of Service B instances → picks one → makes request.

Eureka | Consul | client-side load balancing

2️⃣ Server-Side Service Discovery

👉 Client makes request to load balancer, which queries registry and routes.

Example: Service A calls load balancer → load balancer queries registry → routes to healthy Service B instance.

AWS ALB | Nginx | simpler clients

3️⃣ Service Registry (Consul, etcd, Eureka)

👉 Central database of service instances and their health status.

Example: Service B instances register with Consul → Service A discovers from Consul.

Service registry | health checks | heartbeats

4️⃣ DNS-Based Discovery

👉 Use DNS to resolve service names to IP addresses.

Example: Service A queries `service-b.internal` → DNS returns IP addresses of healthy instances.

Kubernetes DNS | AWS Cloud Map | simple but limited

5️⃣ Client-Side Load Balancing

👉 Client library chooses which instance to call.

Example: Service A has list of Service B instances → uses round-robin to pick one.

Ribbon | client libraries | flexible algorithms

6️⃣ Health Checks for Instance Removal

👉 Registry removes unhealthy instances from available pool.

Example: Service B instance fails health check → removed from registry → traffic stops routing to it.

Health endpoints | TTL | automatic deregistration

💡 Problem 107: API Gateway Pattern

INTERVIEWER: Should clients call microservices directly or through an API gateway?

Question:

Solution:

API gateway provides single entry point — handles routing, authentication, rate limiting for all services.

Here's how API gateway pattern works 👇

1️⃣ Single Entry Point for Clients

👉 Gateway exposes unified API, routes requests to appropriate microservices.

Example: Mobile app calls `/api/orders` → gateway routes to order service.

Kong | AWS API Gateway | Apigee | single endpoint

2️⃣ Authentication and Authorization

👉 Gateway validates auth tokens before routing to services.

Example: Client sends JWT → gateway validates → extracts user_id → passes to service.

Centralized auth | services trust gateway

3️⃣ Rate Limiting and Throttling

👉 Gateway enforces rate limits per client or API key.

Example: Free tier limited to 100 req/min → gateway enforces → returns 429 if exceeded.

Centralized rate limiting | tiered access

4️⃣ Request/Response Transformation

👉 Gateway transforms external API format to internal service format.

Example: External API uses camelCase → gateway converts to snake_case for services.

Protocol translation | format adaptation

5️⃣ Aggregation of Multiple Services

👉 Gateway calls multiple services and combines responses.

Example: User dashboard → gateway calls user service, order service, notification service → combines into single response.

BFF pattern | reduce client round trips

6️⃣ Gateway as Single Point of Failure

👉 If gateway goes down, entire API unavailable — need high availability.

Example: Deploy multiple gateway instances behind load balancer.

HA | redundancy | health checks

Topic 38: Saga Pattern

📚 Service Discovery

💡 Problem 108: Understanding Saga Pattern for Distributed Transactions

INTERVIEWER: How do you handle transactions across multiple microservices?

Question:

Solution:

Sagas break distributed transactions into local transactions with compensation — eventual consistency instead of ACID.

Here's how saga pattern works 👇

1️⃣ What is a Saga?

👉 Sequence of local transactions, each with compensating transaction for rollback.

Example: Order → Reserve Inventory → Charge Payment → if payment fails → Unreserve Inventory.

Distributed transaction | eventual consistency

2️⃣ Choreography-Based Saga

👉 Each service publishes events, other services react independently.

Example: Order service publishes OrderCreated → Inventory service reserves → publishes InventoryReserved → Payment service charges.

Event-driven | decentralized | complex to trace

3️⃣ Orchestration-Based Saga

👉 Central orchestrator coordinates saga steps.

Example: Saga orchestrator tells Order service → then Inventory → then Payment → handles failures.

Centralized control | easier to understand | single point of failure

---

4️⃣ Compensating Transactions

👉 Undo previous steps if later step fails.

Example: Payment fails → run compensation → unreserve inventory → cancel order.

Rollback logic | compensation handlers

5️⃣ Saga State Machine

👉 Track saga progress through states.

Example: States: OrderCreated → InventoryReserved → PaymentProcessed → Completed or OrderCreated → InventoryReserved → PaymentFailed → Compensating → Cancelled.

State tracking | recovery

6️⃣ Idempotency for Retry Safety

👉 Each saga step must be idempotent for safe retries.

Example: Reserve inventory request sent twice → only reserves once.

Idempotency keys | safe retries

💡 Problem 109: Saga Failure Handling and Compensation

INTERVIEWER: What happens if a step in the middle of a saga fails?

Question:

Solution:

Sagas must compensate completed steps when failure occurs — undo in reverse order.

Here's how to handle saga failures 👇

1️⃣ Backward Recovery (Compensation)

👉 Execute compensating transactions in reverse order.

Example: Order → Reserve Inventory → Charge Payment fails → Unreserve Inventory → Cancel Order.

Rollback | compensating actions

2️⃣ Forward Recovery (Retry)

👉 Retry failed step until success.

Example: Payment service temporarily down → retry payment → eventually succeeds.

`Retry with exponential backoff` `eventual success`

3️⃣ Semantic Lock for Reservations

👉 Reserve resources without committing until saga completes.

Example: Inventory reserved but not decremented → if saga fails → release reservation.

`Pessimistic locking` `reservation pattern`

4️⃣ Saga Log for Recovery

👉 Store saga state for crash recovery.

Example: Orchestrator crashes mid-saga → restart → read log → resume from last step.

`Persistence` `recovery` `database log`

5️⃣ Timeout Handling

👉 If step doesn't complete in time, trigger compensation.

Example: Payment service doesn't respond after 30 seconds → assume failure → compensate.

`Timeout policies` `failure detection`

6️⃣ Manual Intervention for Non-Compensable Steps

👉 Some actions can't be undone — require manual handling.

Example: Email sent to user → can't unsend → send apology email or mark as error.

`Human intervention` `support queue`

📚 Container Orchestration

💡 Problem 110: Saga vs Two-Phase Commit (2PC)

INTERVIEWER: Why not use two-phase commit for distributed transactions?

Question:

Solution:

2PC provides strong consistency but blocks on failures — sagas trade consistency for availability.

Here's saga vs 2PC trade-offs 👇

1️⃣ Two-Phase Commit (2PC)

👉 Coordinator asks all participants to prepare → all agree → coordinator commits all.

Example: Coordinator: "Prepare?" → All: "Yes" → Coordinator: "Commit now" → All commit.

XA transactions | strong consistency | blocking

2️⃣ 2PC Blocks on Coordinator Failure

👉 If coordinator crashes after prepare, participants locked indefinitely.

Example: Coordinator crashes → participants in prepared state → can't commit or abort → resources locked.

Blocking protocol | single point of failure

3️⃣ Sagas Are Non-Blocking

👉 Each step commits immediately, compensation undoes if needed.

Example: Payment commits → inventory commits → if later step fails → run compensations.

Non-blocking | eventual consistency

4️⃣ 2PC Requires All Services Up

👉 If one service down, entire transaction blocks.

Example: Payment service down → 2PC can't complete → blocks forever or times out.

High availability requirement | coupled services

5️⃣ Sagas Provide Better Availability

👉 Services continue operating, eventual consistency achieved.

Example: Payment service down → saga retries → eventually succeeds → saga completes.

Resilience | retry mechanisms

6️⃣ When to Use Each

👉 Use 2PC for tightly coupled systems requiring strong consistency (rare in microservices). Use sagas for loosely coupled services with eventual consistency.

Example: Banking core system → 2PC. E-commerce order flow → Saga.

Trade-offs | choose based on requirements

Topic 39: API Gateway Pattern

💡 Problem 111: Implementing Backend for Frontend (BFF)

> INTERVIEWER: Your mobile and web apps need different API responses. How do you handle this?

Question:

Solution:

BFF creates separate backend API for each frontend — optimized for specific client needs.

Here's how BFF pattern works 👇

1️⃣ Separate API Gateway per Client Type

👉 Mobile BFF, web BFF, desktop BFF — each optimized for that client.

Example: Mobile BFF returns minimal data for bandwidth, web BFF returns full data.

Client-specific API | optimized responses

2️⃣ Aggregate Multiple Service Calls

👉 BFF calls multiple microservices, combines responses.

Example: Dashboard request → BFF calls user service, order service, notification service → returns combined JSON.

Reduce client round trips | N+1 problem avoided

3️⃣ Transform Data for Client Format

👉 BFF adapts internal service format to client-friendly format.

Example: Services return snake_case → BFF converts to camelCase for JavaScript clients.

Data transformation | protocol adaptation

4️⃣ Client-Specific Business Logic

👉 BFF implements logic specific to client needs.

Example: Mobile BFF returns paginated results with limit=10, web BFF returns limit=50.

Client-optimized logic | experience customization

5️⃣ BFF Owned by Frontend Team

👉 Frontend team controls BFF, can iterate independently.

Example: Mobile team updates mobile BFF without affecting web team.

Team autonomy independent deployment

6️⃣ Avoid Duplication Across BFFs

👉 Extract shared logic to libraries or shared services.

Example: Authentication logic shared across all BFFs via library.

Code reuse shared utilities

📚 Blue-Green Deployment

💡 Problem 112: GraphQL as API Gateway Alternative

> INTERVIEWER: Should you use GraphQL instead of REST API gateway?

Question:

Solution:

GraphQL lets clients request exactly what they need — reduces over-fetching and under-fetching.

Here's when to use GraphQL 👇

1️⃣ Client Specifies Required Fields

👉 Client queries only needed fields, avoids over-fetching.

Example: Query `{ user { name, email } }` → returns only name and email, not full user object.

Flexible queries bandwidth optimization

2️⃣ Fetch Related Data in Single Request

👉 GraphQL resolves nested relationships without N+1 requests.

Example: Query `{ user { orders { items } } }` → single request gets user, orders, and items.

Nested queries | eliminates multiple round trips

3️⃣ Schema-Driven API

👉 GraphQL schema defines available types and fields.

Example: Type definitions → clients discover API capabilities → strongly typed.

Type safety | introspection | schema validation

4️⃣ Use GraphQL Gateway Over Microservices

👉 GraphQL server aggregates data from multiple microservices.

Example: GraphQL resolvers call order service, user service, inventory service → combine data.

Federation | Apollo Gateway | schema stitching

5️⃣ Trade-offs: Complexity and Caching

👉 GraphQL adds complexity, harder to cache than REST.

Example: REST `/users/123` cached by URL → GraphQL POST with query body harder to cache.

HTTP caching | CDN | complexity trade-off

6️⃣ When to Use REST vs GraphQL

👉 Use REST for simple CRUD, public APIs, caching. Use GraphQL for complex data fetching, mobile apps.

Example: Public API with rate limiting → REST. Mobile app with varied data needs → GraphQL.

Choose based on requirements | not hype

Topic 40: Serverless Architecture

💡 Problem 113: When to Use Serverless (AWS Lambda, Cloud Functions)

INTERVIEWER: Should you use serverless functions or traditional servers?

Question:

Solution:

Serverless is great for event-driven, variable workloads — but has cold start and execution limits. Here's when to use serverless 👇

1️⃣ Event-Driven Workloads

👉 Functions triggered by events like file uploads, HTTP requests, queue messages.

Example: S3 upload triggers Lambda to resize image.

AWS Lambda | Google Cloud Functions | event triggers

2️⃣ Variable or Unpredictable Traffic

👉 Auto-scales from zero to thousands of concurrent executions.

Example: Black Friday sale → traffic spikes 100x → Lambda scales automatically.

Auto-scaling | pay per execution

3️⃣ Short-Lived Tasks

👉 Functions run for seconds to minutes, not long-running processes.

Example: API endpoint, data transformation, webhook handler.

Max 15 minutes on Lambda | short executions

4️⃣ Avoid Serverless for Persistent Connections

👉 WebSockets, database connection pools don't work well.

Example: WebSocket server needs persistent connection → use container instead.

Connection limits | cold starts

5️⃣ Cold Start Latency

👉 First request after idle takes longer to start function.

Example: Function idle for 5 minutes → next request waits 1-2 seconds for cold start.

Cold start | provisioned concurrency option

6️⃣ Cost Effective for Low Volume

👉 Pay only for execution time, no idle server costs.

Example: Function runs 1M times/month at 100ms each → cheaper than running server 24/7.

Pay-per-use | cost optimization

📚 Bloom Filters

💡 Problem 114: Serverless Best Practices

INTERVIEWER: How do you optimize serverless functions for performance and cost?

Question:

Solution:

Minimize cold starts, optimize bundle size, use async patterns — serverless has unique constraints.

Here are serverless best practices 👇

1️⃣ Minimize Cold Start Time

👉 Keep deployment package small, minimize dependencies.

Example: Lambda with 50MB package → slow cold start. 5MB package → fast cold start.

Bundle optimization | tree shaking | reduce dependencies

2️⃣ Reuse Connections Outside Handler

👉 Initialize database connections, HTTP clients outside handler function.

Example: Database pool created once → reused across invocations in same container.

Connection reuse | warm containers

3️⃣ Use Provisioned Concurrency for Critical Paths

👉 Keep functions warm to eliminate cold starts.

Example: API endpoint with strict latency SLA → provision 10 warm instances.

Provisioned concurrency | cost vs performance

4️⃣ Avoid Heavy Synchronous Chains

👉 Don't chain many Lambda functions synchronously.

Example: Lambda 1 → Lambda 2 → Lambda 3 → Lambda 4 → slow and expensive. Use async events instead.

Step Functions | async patterns

5️⃣ Right-Size Memory Allocation

👉 More memory = more CPU → sometimes faster and cheaper.

Example: Function with 128MB takes 10 seconds, 1024MB takes 2 seconds → 1024MB might cost less.

Memory optimization | CPU allocation

6️⃣ Use Environment Variables for Config

👉 Avoid hardcoded values, use env vars for config.

Example: Database URL in environment variable → change without redeploying code.

Configuration management | flexibility

💡 Problem 115: Serverless Data Processing Patterns

INTERVIEWER: How do you process large amounts of data with serverless?

Question:

Solution:

Serverless excels at parallel processing — trigger multiple functions to process batches concurrently.

Here are serverless data processing patterns 👇

1️⃣ Fan-Out Pattern for Parallel Processing

👉 Split work into chunks, process each chunk in separate function.

Example: Process 1M records → split into 100 batches of 10K → 100 Lambdas process in parallel.

Parallel processing | SQS fanout | high throughput

2️⃣ S3 Event Triggers for File Processing

👉 File uploaded to S3 triggers Lambda to process.

Example: CSV uploaded → Lambda parses and inserts into database.

Event-driven | S3 notifications | automatic trigger

3️⃣ Stream Processing with Kinesis/Kafka

👉 Lambda consumes events from stream, processes in batches.

Example: Clickstream data in Kinesis → Lambda aggregates and writes to database.

Real-time processing | stream batching

4️⃣ Step Functions for Orchestration

👉 Coordinate multiple Lambda functions in workflow.

Example: ETL pipeline → Step 1: Extract → Step 2: Transform → Step 3: Load → Step Functions orchestrates.

Workflow | state machine | error handling

5️⃣ Use SQS for Rate Limiting and Retry

👉 Queue messages, Lambda pulls at controlled rate.

Example: 1M messages in SQS → Lambda processes 100 concurrently → rate-limited processing.

Queue-based | backpressure | retry on failure

\n

6️⃣ Avoid Timeouts with Partial Processing

👉 Process batch partially, requeue remaining work.

Example: Lambda has 10 minutes to process 100K records → processes 50K → requeues rest.

Chunking | checkpoint pattern | progress tracking

💡 Problem 116: Handling Large File Uploads at Scale

> INTERVIEWER: Your users upload 50GB of videos every hour — where do you store them?

Question:

Solution:

Scalable file storage requires object storage, not local disks or databases.

1️⃣ Object Storage (S3, GCS, Azure Blob)

👉 Store files in cloud object storage designed for massive scale.

Example: 50GB/hour = 1.2TB/day → S3 handles this easily with 99.999999999% durability.

AWS S3 | Google Cloud Storage | Azure Blob Storage

2️⃣ Direct Upload to S3 (Pre-signed URLs)

👉 Let users upload directly to S3, bypassing your servers.

Example: Generate pre-signed URL → client uploads 2GB video directly to S3 → no server bandwidth used.

S3 pre-signed URLs | direct upload pattern

3️⃣ Multipart Upload

👉 Split large files into chunks for parallel, resumable uploads.

Example: 5GB video split into 100MB chunks → upload in parallel → auto-retry failed chunks.

S3 multipart upload | resumable uploads

4️⃣ CDN Integration

👉 Serve uploaded files through CDN for fast global access.

Example: Video uploaded to S3 us-east-1 → served via CloudFront edge locations worldwide.

CloudFront | Cloudflare R2

5️⃣ Lifecycle Policies

👉 Automatically move old files to cheaper storage tiers.

Example: Videos >90 days old → moved to Glacier ($0.004/GB vs S3's $0.023/GB).

S3 Lifecycle | storage classes | archival

6️⃣ Metadata in Database

👉 Store file metadata (URL, size, owner) in database, not files themselves.

Example: uploads table with {file_id, s3_key, user_id, size, created_at}.

`Metadata separation` `database for search`

7️⃣ Virus Scanning

👉 Scan uploaded files asynchronously before making public.

Example: Upload triggers Lambda → ClamAV scan → if clean, move to public bucket.

`ClamAV` `async processing` `quarantine bucket`

💡 Problem 117: Choosing S3 Storage Classes

> INTERVIEWER: You store 500TB of user uploads — how do you minimize S3 costs?

Question:

Solution:

S3 storage classes balance access speed, durability, and cost for different use cases.

1️⃣ S3 Standard

👉 Frequently accessed data, highest cost.

Example: $0.023/GB/month → use for recent uploads accessed daily.

`S3 Standard` `frequent access`

2️⃣ S3 Intelligent-Tiering

👉 Automatically moves objects between tiers based on access patterns.

Example: File not accessed for 30 days → auto-moved to cheaper tier → saves cost.

Intelligent-Tiering | automatic optimization

3️⃣ S3 Standard-IA (Infrequent Access)

👉 Lower storage cost, but charges for retrieval.

Example: $0.0125/GB/month + $0.01/GB retrieval → for backups accessed monthly.

Standard-IA | infrequent access

4️⃣ S3 One Zone-IA

👉 Cheaper than Standard-IA, single AZ (lower durability).

Example: $0.01/GB/month → for reproducible data like thumbnails.

One Zone-IA | single AZ

5️⃣ S3 Glacier Instant Retrieval

👉 Archive storage with millisecond retrieval.

Example: $0.004/GB/month → for archives accessed quarterly.

Glacier Instant | archive

6️⃣ S3 Glacier Flexible/Deep Archive

👉 Lowest cost, slow retrieval (hours).

Example: $0.00099/GB/month (Deep Archive) → for compliance archives, 7-year retention.

Glacier Deep Archive | long-term archive

7️⃣ Lifecycle Policies

👉 Automatically transition objects between classes.

Example: Standard (30 days) → Standard-IA (90 days) → Glacier (1 year) → Deep Archive.

Lifecycle rules | automated transitions

💡 Problem 117: Choosing S3 Storage Classes

INTERVIEWER: You store 500TB of user uploads — how do you minimize S3 costs?

Question:

Solution:

S3 storage classes balance access speed, durability, and cost for different use cases.

1️⃣ S3 Standard

👉 Frequently accessed data, highest cost.

Example: $0.023/GB/month → use for recent uploads accessed daily.

S3 Standard | frequent access

2️⃣ S3 Intelligent-Tiering

👉 Automatically moves objects between tiers based on access patterns.

Example: File not accessed for 30 days → auto-moved to cheaper tier → saves cost.

Intelligent-Tiering | automatic optimization

3️⃣ S3 Standard-IA (Infrequent Access)

👉 Lower storage cost, but charges for retrieval.

Example: $0.0125/GB/month + $0.01/GB retrieval → for backups accessed monthly.

Standard-IA | infrequent access

4️⃣ S3 One Zone-IA

👉 Cheaper than Standard-IA, single AZ (lower durability).

Example: $0.01/GB/month → for reproducible data like thumbnails.

One Zone-IA single AZ

5️⃣ S3 Glacier Instant Retrieval

👉 Archive storage with millisecond retrieval.

Example: $0.004/GB/month → for archives accessed quarterly.

Glacier Instant archive

6️⃣ S3 Glacier Flexible/Deep Archive

👉 Lowest cost, slow retrieval (hours).

Example: $0.00099/GB/month (Deep Archive) → for compliance archives, 7-year retention.

Glacier Deep Archive long-term archive

\n

7️⃣ Lifecycle Policies

👉 Automatically transition objects between classes.

Example: Standard (30 days) → Standard-IA (90 days) → Glacier (1 year) → Deep Archive.

Lifecycle rules automated transitions

💡 Problem 117: Building Sub-50ms Real-Time Chat

> INTERVIEWER: Your chat needs messages in <50ms — how would you build that?

Question:

Solution:

Real-time communication requires WebSockets and in-memory message passing.

1️⃣ WebSocket Connections

👉 Use WebSockets for bidirectional, low-latency communication.

Example: HTTP polling: 500ms latency; WebSocket: 10-30ms latency.

WebSockets | Socket.io | SignalR

2️⃣ In-Memory Message Broker

👉 Use Redis Pub/Sub for real-time message distribution.

Example: User sends message → published to Redis → all chat server instances get it in <5ms.

Redis Pub/Sub | in-memory messaging

3️⃣ Connection Management

👉 Keep persistent WebSocket connections open per user.

Example: 10K concurrent users = 10K open WebSocket connections across server pool.

Connection pooling | stateful connections

4️⃣ Horizontal Scaling with Sticky Sessions

👉 Route user's WebSocket connection to same server instance.

Example: Load balancer uses user_id hash → consistent server routing.

Sticky sessions | consistent hashing

5️⃣ Message Persistence (Asynchronous)

👉 Store messages in database asynchronously, don't block delivery.

Example: Message delivered via WebSocket in 20ms → saved to DB in background (200ms).

Async writes | eventual persistence

6️⃣ Presence System

👉 Track online/offline status with Redis or similar.

Example: User connects → SET user:123:online with 60s TTL → heartbeat every 30s.

Redis TTL | heartbeat mechanism

7️⃣ Geographic Distribution

👉 Deploy chat servers in multiple regions for lower latency.

Example: US users connect to us-east servers, EU users to eu-west → <50ms regional latency.

Multi-region | edge servers

💡 Problem 118: Scaling WebSocket Connections

> INTERVIEWER: You have 100K concurrent WebSocket connections — how do you scale them?

Question:

Solution:

WebSocket connections are stateful and long-lived, requiring different scaling patterns than HTTP.

1️⃣ Stateful Connection Challenge

👉 Each user connected to specific server instance.

Example: User on server1 → can't handle message from user on server2 without coordination.

Stateful | sticky connections

2️⃣ Redis Pub/Sub for Message Broadcasting

👉 Publish messages to all server instances via Redis.

Example: User on server1 sends message → published to Redis → all servers receive → broadcast to connected users.

Redis Pub/Sub | message broadcasting

3️⃣ Sticky Sessions (Load Balancer)

👉 Route user to same server for connection lifetime.

Example: ALB uses source IP hash → user always routed to same server.

`Sticky sessions` `session affinity`

4️⃣ Connection Limits Per Server

👉 Each server has max connection capacity.

Example: Node.js server handles ~10K connections → 100K users = 10 servers minimum.

`Connection limits` `capacity planning`

5️⃣ Horizontal Scaling

👉 Add more servers as connection count grows.

Example: Auto-scale based on connection count metric → add server when > 8K connections/server.

`Auto-scaling` `horizontal scaling`

6️⃣ Dedicated WebSocket Servers

👉 Separate WebSocket servers from HTTP API servers.

Example: HTTP API handles REST, WebSocket servers handle only real-time → optimized separately.

`Service separation` `optimization`

7️⃣ Connection State Management

👉 Store connection metadata in shared storage.

Example: Track which user connected to which server in Redis → route messages correctly.

`State management` `connection tracking`

💡 Problem 118: GraphQL vs REST Trade-offs

INTERVIEWER: When should you choose GraphQL over REST, and what are the trade-offs?

Question:

Solution:

GraphQL solves specific API problems but introduces new complexity.

1️⃣ Over-fetching Problem in REST

👉 REST endpoints return fixed data structures, often with unnecessary fields.

Example: GET /users returns 20 fields, but mobile app only needs name and email → wasted bandwidth.

REST limitations | data over-fetching

2️⃣ Under-fetching Problem in REST

👉 Need multiple REST calls to get related data.

Example: Get user → separate call for posts → separate call for comments → 3 round trips.

N+1 problem | multiple requests

3️⃣ GraphQL Single Endpoint

👉 One endpoint with flexible queries for exact data needs.

Example: Single query fetches user + posts + comments in one request with only needed fields.

Flexible queries | single endpoint

4️⃣ GraphQL Complexity

👉 Requires schema definition, resolvers, and query complexity analysis.

Example: Must prevent malicious deep queries that could DoS your server.

Query complexity depth limiting

5️⃣ Caching Challenges

👉 GraphQL makes HTTP caching harder due to POST requests.

Example: REST GET /users/123 → easily cached; GraphQL POST with query → needs custom caching.

Caching complexity Apollo cache

6️⃣ When to Use REST

👉 Simple CRUD operations, public APIs, heavy caching needs.

Example: Public weather API with standard endpoints → REST is simpler.

REST advantages simple APIs

7️⃣ When to Use GraphQL

👉 Complex data relationships, multiple clients with different needs, real-time updates.

Example: Social media app with web, mobile, smart TV → each needs different data subsets.

GraphQL advantages flexible clients

💡 Problem 118: Scaling WebSocket Connections

> INTERVIEWER: You have 100K concurrent WebSocket connections — how do you scale them?

Question:

Solution:

WebSocket connections are stateful and long-lived, requiring different scaling patterns than HTTP.

1️⃣ Stateful Connection Challenge

👉 Each user connected to specific server instance.

Example: User on server1 → can't handle message from user on server2 without coordination.

Stateful | sticky connections

2️⃣ Redis Pub/Sub for Message Broadcasting

👉 Publish messages to all server instances via Redis.

Example: User on server1 sends message → published to Redis → all servers receive → broadcast to connected users.

Redis Pub/Sub | message broadcasting

3️⃣ Sticky Sessions (Load Balancer)

👉 Route user to same server for connection lifetime.

Example: ALB uses source IP hash → user always routed to same server.

Sticky sessions | session affinity

4️⃣ Connection Limits Per Server

👉 Each server has max connection capacity.

Example: Node.js server handles ~10K connections → 100K users = 10 servers minimum.

Connection limits | capacity planning

5️⃣ Horizontal Scaling

👉 Add more servers as connection count grows.

Example: Auto-scale based on connection count metric → add server when > 8K connections/server.

Auto-scaling | horizontal scaling

6️⃣ Dedicated WebSocket Servers

👉 Separate WebSocket servers from HTTP API servers.

Example: HTTP API handles REST, WebSocket servers handle only real-time → optimized separately.

Service separation | optimization

7️⃣ Connection State Management

👉 Store connection metadata in shared storage.

Example: Track which user connected to which server in Redis → route messages correctly.

State management | connection tracking

💡 Problem 118: GraphQL vs REST Trade-offs

> INTERVIEWER: When should you choose GraphQL over REST, and what are the trade-offs?

Question:

Solution:

GraphQL solves specific API problems but introduces new complexity.

1️⃣ Over-fetching Problem in REST

👉 REST endpoints return fixed data structures, often with unnecessary fields.

Example: GET /users returns 20 fields, but mobile app only needs name and email → wasted bandwidth.

REST limitations | data over-fetching

2️⃣ Under-fetching Problem in REST

👉 Need multiple REST calls to get related data.

Example: Get user → separate call for posts → separate call for comments → 3 round trips.

N+1 problem | multiple requests

3️⃣ GraphQL Single Endpoint

👉 One endpoint with flexible queries for exact data needs.

Example: Single query fetches user + posts + comments in one request with only needed fields.

Flexible queries | single endpoint

4️⃣ GraphQL Complexity

👉 Requires schema definition, resolvers, and query complexity analysis.

Example: Must prevent malicious deep queries that could DoS your server.

Query complexity | depth limiting

5️⃣ Caching Challenges

👉 GraphQL makes HTTP caching harder due to POST requests.

Example: REST GET /users/123 → easily cached; GraphQL POST with query → needs custom caching.

Caching complexity | Apollo cache

6️⃣ When to Use REST

👉 Simple CRUD operations, public APIs, heavy caching needs.

Example: Public weather API with standard endpoints → REST is simpler.

REST advantages | simple APIs

\n\n\n

7️⃣ When to Use GraphQL

👉 Complex data relationships, multiple clients with different needs, real-time updates.

Example: Social media app with web, mobile, smart TV → each needs different data subsets.

GraphQL advantages | flexible clients

💡 Problem 116: Handling Large File Uploads at Scale

INTERVIEWER: Your users upload 50GB of videos every hour — where do you store them?

Question:

Solution:

Scalable file storage requires object storage, not local disks or databases.

1️⃣ Object Storage (S3, GCS, Azure Blob)

👉 Store files in cloud object storage designed for massive scale.

Example: 50GB/hour = 1.2TB/day → S3 handles this easily with 99.999999999% durability.

AWS S3 | Google Cloud Storage | Azure Blob Storage

2️⃣ Direct Upload to S3 (Pre-signed URLs)

👉 Let users upload directly to S3, bypassing your servers.

Example: Generate pre-signed URL → client uploads 2GB video directly to S3 → no server bandwidth used.

S3 pre-signed URLs | direct upload pattern

3️⃣ Multipart Upload

👉 Split large files into chunks for parallel, resumable uploads.

Example: 5GB video split into 100MB chunks → upload in parallel → auto-retry failed chunks.

S3 multipart upload | resumable uploads

4️⃣ CDN Integration

👉 Serve uploaded files through CDN for fast global access.

Example: Video uploaded to S3 us-east-1 → served via CloudFront edge locations worldwide.

CloudFront | Cloudflare R2

5️⃣ Lifecycle Policies

👉 Automatically move old files to cheaper storage tiers.

Example: Videos >90 days old → moved to Glacier ($0.004/GB vs S3's $0.023/GB).

S3 Lifecycle | storage classes | archival

6️⃣ Metadata in Database

👉 Store file metadata (URL, size, owner) in database, not files themselves.

Example: uploads table with {file_id, s3_key, user_id, size, created_at}.

Metadata separation | database for search

\n

7️⃣ Virus Scanning

👉 Scan uploaded files asynchronously before making public.

Example: Upload triggers Lambda → ClamAV scan → if clean, move to public bucket.

ClamAV | async processing | quarantine bucket

💡 Problem 117: Building Sub-50ms Real-Time Chat

> INTERVIEWER: Your chat needs messages in <50ms — how would you build that?

Question:

Solution:

Real-time communication requires WebSockets and in-memory message passing.

1️⃣ WebSocket Connections

👉 Use WebSockets for bidirectional, low-latency communication.

Example: HTTP polling: 500ms latency; WebSocket: 10-30ms latency.

WebSockets | Socket.io | SignalR

2️⃣ In-Memory Message Broker

👉 Use Redis Pub/Sub for real-time message distribution.

Example: User sends message → published to Redis → all chat server instances get it in <5ms.

Redis Pub/Sub | in-memory messaging

3️⃣ Connection Management

👉 Keep persistent WebSocket connections open per user.

Example: 10K concurrent users = 10K open WebSocket connections across server pool.

Connection pooling | stateful connections

4️⃣ Horizontal Scaling with Sticky Sessions

👉 Route user's WebSocket connection to same server instance.

Example: Load balancer uses user_id hash → consistent server routing.

Sticky sessions | consistent hashing

5️⃣ Message Persistence (Asynchronous)

👉 Store messages in database asynchronously, don't block delivery.

Example: Message delivered via WebSocket in 20ms → saved to DB in background (200ms).

Async writes | eventual persistence

6️⃣ Presence System

👉 Track online/offline status with Redis or similar.

Example: User connects → SET user:123:online with 60s TTL → heartbeat every 30s.

Redis TTL heartbeat mechanism

\n\n

7️⃣ Geographic Distribution

👉 Deploy chat servers in multiple regions for lower latency.

Example: US users connect to us-east servers, EU users to eu-west → <50ms regional latency.

Multi-region edge servers

Topic 41: Auto Scaling Strategies

💡 Problem 116: Application Auto Scaling

INTERVIEWER: How do you automatically scale your application based on load?

Question:

Solution:

Auto scaling adds/removes instances based on metrics — scale out for traffic, scale in to save cost.

Here's how auto scaling works 👇

1️⃣ Metric-Based Scaling

👉 Scale based on CPU, memory, request count, or custom metrics.

Example: CPU > 70% for 5 minutes → add 2 instances. CPU < 30% → remove 1 instance.

CloudWatch metrics | threshold-based

2️⃣ Target Tracking Scaling

👉 Maintain metric at target value.

Example: Target 1000 requests per instance → traffic increases → auto scaling adds instances to maintain target.

AWS target tracking | automatic calculation

3️⃣ Scheduled Scaling

👉 Scale based on known patterns.

Example: Scale up every weekday at 8 AM, scale down at 6 PM.

Predictable traffic | cron-based

4️⃣ Step Scaling for Rapid Changes

👉 Add/remove multiple instances based on alarm severity.

Example: CPU 70-80% → add 1 instance, 80-90% → add 3 instances, >90% → add 5 instances.

Aggressive scaling | step policies

5️⃣ Cooldown Period

👉 Wait after scaling action before next scaling.

Example: Scale up → wait 5 minutes → check if more scaling needed → prevents flapping.

Stabilization | prevent thrashing

\n

6️⃣ Scale Out Faster Than Scale In

👉 Add capacity quickly, remove slowly.

Example: Scale out immediately on high load, wait 10 minutes before scaling in.

Aggressive scale out | conservative scale in

💡 Problem 117: Handling Overnight Viral Traffic

> INTERVIEWER: Your app goes viral overnight — what's the first thing that breaks?

Question:

Solution:

Sudden traffic spikes expose bottlenecks in least scalable components.

1️⃣ Database Connection Pool Exhaustion

👉 Fixed-size connection pools can't scale with traffic.

Example: 10K concurrent users → 10K app server connections → but DB pool only 100 connections → timeouts.

Connection pooling | PgBouncer | read replicas

2️⃣ Rate Limit External APIs

👉 Third-party API rate limits hit unexpectedly.

Example: 10x traffic → 10x payment API calls → hit Stripe rate limit → checkout fails.

Circuit breakers | queue external calls | upgrade API tiers

3️⃣ Memory Exhaustion

👉 Application servers run out of memory with increased load.

Example: Each user session = 10MB → 10K users = 100GB RAM → servers crash.

Stateless design | external session storage | horizontal scaling

4️⃣ CDN Origin Overload

👉 CDN cache misses overwhelm origin servers.

Example: Viral post links to uncached page → 100K users → all hit origin → origin crashes.

Cache warming | higher CDN cache TTL | origin rate limiting

5️⃣ Database Write Bottleneck

👉 Single primary database can't handle write traffic.

Example: 10K sign-ups/minute → single DB primary maxed out → sign-up fails.

Write sharding | queue writes | batch inserts

6️⃣ Logging/Monitoring Overload

👉 Excessive logging consumes resources.

Example: 100x traffic → 100x logs → disk full → app crashes trying to write logs.

Log sampling | async logging | log levels

\n\n

7️⃣ Auto-Scaling Lag

👉 Auto-scaling reacts too slowly to sudden spike.

Example: Traffic spikes in 30 seconds → auto-scaling takes 5 minutes to launch instances → app down in between.

Pre-warming | faster scaling | over-provisioning buffer

💡 Problem 117: Handling Overnight Viral Traffic

INTERVIEWER: Your app goes viral overnight — what's the first thing that breaks?

Question:

Solution:

Sudden traffic spikes expose bottlenecks in least scalable components.

1️⃣ Database Connection Pool Exhaustion

👉 Fixed-size connection pools can't scale with traffic.

Example: 10K concurrent users → 10K app server connections → but DB pool only 100 connections → timeouts.

Connection pooling | PgBouncer | read replicas

2️⃣ Rate Limit External APIs

👉 Third-party API rate limits hit unexpectedly.

Example: 10x traffic → 10x payment API calls → hit Stripe rate limit → checkout fails.

Circuit breakers | queue external calls | upgrade API tiers

3️⃣ Memory Exhaustion

👉 Application servers run out of memory with increased load.

Example: Each user session = 10MB → 10K users = 100GB RAM → servers crash.

Stateless design | external session storage | horizontal scaling

---

4️⃣ CDN Origin Overload

👉 CDN cache misses overwhelm origin servers.

Example: Viral post links to uncached page → 100K users → all hit origin → origin crashes.

Cache warming | higher CDN cache TTL | origin rate limiting

5️⃣ Database Write Bottleneck

👉 Single primary database can't handle write traffic.

Example: 10K sign-ups/minute → single DB primary maxed out → sign-up fails.

Write sharding | queue writes | batch inserts

6️⃣ Logging/Monitoring Overload

👉 Excessive logging consumes resources.

Example: 100x traffic → 100x logs → disk full → app crashes trying to write logs.

Log sampling | async logging | log levels

\n

7️⃣ Auto-Scaling Lag

👉 Auto-scaling reacts too slowly to sudden spike.

Example: Traffic spikes in 30 seconds → auto-scaling takes 5 minutes to launch instances → app down in between.

Pre-warming | faster scaling | over-provisioning buffer

💡 Problem 117: Kubernetes Horizontal Pod Autoscaler (HPA)

> INTERVIEWER: How does Kubernetes auto-scale pods based on load?

Question:

Solution:

HPA adjusts pod replicas based on metrics — CPU, memory, or custom metrics from application.

Here's how Kubernetes HPA works 👇

1️⃣ CPU-Based Scaling

👉 Scale pods when average CPU usage crosses threshold.

Example: Target 50% CPU → actual 80% → HPA adds pods to bring average down to 50%.

Resource requests | metrics server

2️⃣ Memory-Based Scaling

👉 Scale based on memory utilization.

Example: Memory > 70% → add pods.

Memory metrics | resource requests required

3️⃣ Custom Metrics from Application

👉 Scale based on application-specific metrics.

Example: Queue length > 100 → scale up workers → queue length < 20 → scale down.

Custom metrics API | Prometheus adapter

4️⃣ Min and Max Replicas

👉 Set bounds to prevent under/over-provisioning.

Example: Min 2 replicas (high availability), max 50 replicas (cost control).

Replica limits | safety bounds

5️⃣ Stabilization Window

👉 Wait before scaling down to prevent flapping.

Example: Load drops → wait 5 minutes → if still low → scale down.

Scale-down delay | stability

6️⃣ Vertical Pod Autoscaler (VPA)

👉 Adjust resource requests/limits instead of replica count.

Example: Pod using 2GB RAM but requested 4GB → VPA adjusts to 2.5GB.

Resource optimization | right-sizing

💡 Problem 118: Advanced HPA Scaling Strategies

INTERVIEWER: HPA scales based on CPU — but your app is memory-bound. How do you fix that?

Question:

Solution:

Advanced HPA configurations handle complex scaling scenarios beyond simple CPU metrics.

1️⃣ Memory-Based Scaling

👉 Scale based on memory utilization instead of CPU.

Example: scale when memory > 80% → prevents OOM kills.

Memory metrics | resource limits

2️⃣ Custom Metrics (Prometheus)

👉 Scale on application-specific metrics.

Example: Scale based on queue depth, active connections, request latency from Prometheus.

Custom metrics | Prometheus adapter

3️⃣ Multiple Metrics

👉 Combine multiple metrics for scaling decisions.

Example: Scale if CPU > 70% OR memory > 80% OR queue > 1000 messages.

Multiple metrics | OR conditions

4️⃣ Scale-Up vs Scale-Down Policies

👉 Aggressive scale-up, conservative scale-down.

Example: Scale up immediately on spike, scale down slowly over 5 minutes.

Scale policies | stabilization window

5️⃣ Min/Max Replicas

👉 Set bounds to prevent over-scaling or under-provisioning.

Example: Min 3 replicas (HA), max 50 replicas (cost control).

Replica limits | cost control

6️⃣ Handling Cold Starts

👉 Pre-warm pods to handle sudden spikes.

Example: Keep 5 idle pods ready → instant capacity for traffic spike.

Pre-warming | cold start

7️⃣ VPA (Vertical Pod Autoscaler) Alternative

👉 Adjust pod resources instead of replica count.

Example: App needs more memory → VPA increases memory limit → restart pod.

VPA | resource adjustment

💡 Problem 118: Implementing Blue-Green Deployments

> INTERVIEWER: How do you deploy new code with zero downtime and instant rollback?

Question:

Solution:

Blue-green deployments enable zero-downtime releases with instant rollback capability.

1️⃣ Two Identical Environments

👉 Maintain two production environments: blue (current) and green (new).

Example: Blue environment serves traffic; Deploy new version to green environment.

Blue-green | parallel environments

2️⃣ Switch Traffic with Load Balancer

👉 Flip traffic from blue to green instantly.

Example: Update ALB target group from blue to green → instant switch.

Load balancer | traffic switch

3️⃣ Instant Rollback

👉 If issues detected, switch back to blue immediately.

Example: Green has bug → switch traffic back to blue in seconds.

Rollback | safety

4️⃣ Testing Green Before Switch

👉 Verify green environment before sending production traffic.

Example: Run smoke tests against green → if pass, switch traffic.

Smoke tests | verification

5️⃣ Database Migration Challenge

👉 Ensure schema compatible with both versions.

Example: Add column in backward-compatible way → deploy code → run migration.

Schema compatibility | migrations

6️⃣ Cost Consideration

👉 Running two environments doubles infrastructure cost during deploy.

Example: 20 servers × 2 environments = 40 servers during deployment window.

Cost resource doubling

7️⃣ Canary as Alternative

👉 Gradually shift traffic instead of instant switch.

Example: Route 5% → 25% → 50% → 100% to new version → lower risk.

Canary deployment gradual rollout

💡 Problem 118: Advanced HPA Scaling Strategies

> INTERVIEWER: HPA scales based on CPU — but your app is memory-bound. How do you fix that?

Question:

Solution:

Advanced HPA configurations handle complex scaling scenarios beyond simple CPU metrics.

1️⃣ Memory-Based Scaling

👉 Scale based on memory utilization instead of CPU.

Example: scale when memory > 80% → prevents OOM kills.

Memory metrics resource limits

2️⃣ Custom Metrics (Prometheus)

👉 Scale on application-specific metrics.

Example: Scale based on queue depth, active connections, request latency from Prometheus.

Custom metrics Prometheus adapter

3️⃣ Multiple Metrics

👉 Combine multiple metrics for scaling decisions.

Example: Scale if CPU > 70% OR memory > 80% OR queue > 1000 messages.

Multiple metrics | OR conditions

4️⃣ Scale-Up vs Scale-Down Policies

👉 Aggressive scale-up, conservative scale-down.

Example: Scale up immediately on spike, scale down slowly over 5 minutes.

Scale policies | stabilization window

5️⃣ Min/Max Replicas

👉 Set bounds to prevent over-scaling or under-provisioning.

Example: Min 3 replicas (HA), max 50 replicas (cost control).

Replica limits | cost control

6️⃣ Handling Cold Starts

👉 Pre-warm pods to handle sudden spikes.

Example: Keep 5 idle pods ready → instant capacity for traffic spike.

Pre-warming | cold start

7️⃣ VPA (Vertical Pod Autoscaler) Alternative

👉 Adjust pod resources instead of replica count.

Example: App needs more memory → VPA increases memory limit → restart pod.

VPA | resource adjustment

💡 Problem 118: Implementing Blue-Green Deployments

INTERVIEWER: How do you deploy new code with zero downtime and instant rollback?

Question:

Solution:

Blue-green deployments enable zero-downtime releases with instant rollback capability.

1️⃣ Two Identical Environments

👉 Maintain two production environments: blue (current) and green (new).

Example: Blue environment serves traffic; Deploy new version to green environment.

Blue-green | parallel environments

2️⃣ Switch Traffic with Load Balancer

👉 Flip traffic from blue to green instantly.

Example: Update ALB target group from blue to green → instant switch.

Load balancer | traffic switch

3️⃣ Instant Rollback

👉 If issues detected, switch back to blue immediately.

Example: Green has bug → switch traffic back to blue in seconds.

Rollback | safety

4️⃣ Testing Green Before Switch

👉 Verify green environment before sending production traffic.

Example: Run smoke tests against green → if pass, switch traffic.

Smoke tests | verification

5️⃣ Database Migration Challenge

👉 Ensure schema compatible with both versions.

Example: Add column in backward-compatible way → deploy code → run migration.

Schema compatibility migrations

6️⃣ Cost Consideration

👉 Running two environments doubles infrastructure cost during deploy.

Example: 20 servers × 2 environments = 40 servers during deployment window.

Cost resource doubling

7️⃣ Canary as Alternative

👉 Gradually shift traffic instead of instant switch.

Example: Route 5% → 25% → 50% → 100% to new version → lower risk.

Canary deployment gradual rollout

📚 CDN & Edge Caching

Topic 42: CDN & Edge Caching

💡 Problem 118: Using CDN for Static and Dynamic Content

> INTERVIEWER: How do you reduce latency for users across the globe?

Question:

Solution:

CDN caches content at edge locations near users — reduces latency and origin server load.

Here's how CDN works 👇

1️⃣ Edge Locations Cache Content

👉 CDN servers in many geographic locations cache content.

Example: User in Tokyo requests image → served from Tokyo edge location → 10ms latency instead of 200ms from US origin.

CloudFront | Cloudflare | Akamai | edge caching

2️⃣ Static Content Caching

👉 Images, CSS, JS, fonts served from CDN with long cache TTL.

Example: `Cache-Control: max-age=31536000` → cached for 1 year.

Static assets | immutable content | cache forever

3️⃣ Dynamic Content Caching

👉 Cache API responses with shorter TTL.

Example: Product page cached for 5 minutes → most users hit cache → origin gets 1 request per 5 minutes.

API caching | Cache-Control headers

4️⃣ Cache Invalidation

👉 Purge or invalidate CDN cache when content updates.

Example: Deploy new website version → invalidate CDN cache → users get new version immediately.

Cache purge | versioned URLs | cache busting

5️⃣ Origin Shield for Origin Protection

👉 Additional caching layer between edge and origin to reduce origin load.

Example: 100 edge locations → origin shield → origin. Edge requests consolidated at shield.

`CloudFront origin shield` `cache hierarchy`

6️⃣ Gzip/Brotli Compression

👉 CDN compresses responses before sending to clients.

Example: 1MB HTML file → compressed to 200KB → 5x bandwidth savings.

`Compression` `Accept-Encoding` `faster delivery`

💡 Problem 119: CDN Cache Control and Strategies

> INTERVIEWER: How do you control what gets cached and for how long in a CDN?

Question:

Solution:

Use Cache-Control headers and cache keys to control CDN caching behavior.

Here's how to control CDN caching 👇

1️⃣ Cache-Control Header

👉 Tell CDN how long to cache content.

Example: `Cache-Control: public, max-age=3600` → cache for 1 hour.

`max-age` `s-maxage` `public/private`

2️⃣ Vary Header for Multiple Versions

👉 Cache different versions based on request headers.

Example: `Vary: Accept-Encoding` → cache gzip and non-gzip versions separately.

Vary header | multiple cache entries

3️⃣ Query String in Cache Key

👉 Include or exclude query parameters from cache key.

Example: `/api/products?id=123` → cache per ID. `/api/products?utm_source=email` → ignore utm params.

Cache key configuration | query string handling

4️⃣ Cookie-Based Caching

👉 Include cookies in cache key for personalized content.

Example: Logged-in users see personalized page → cache keyed by session cookie.

Personalization | cache fragmentation

5️⃣ Stale-While-Revalidate

👉 Serve stale content while fetching fresh content in background.

Example: `Cache-Control: max-age=60, stale-while-revalidate=300` → serve stale for 5 min while revalidating.

Improved availability | eventual freshness

6️⃣ Bypass Cache for Dynamic Content

👉 Set `Cache-Control: no-cache` for content that shouldn't be cached.

Example: User account page → no-cache → always fetched from origin.

No-cache | dynamic content | user-specific data

💡 Problem 120: Response Compression Strategies

> INTERVIEWER: Your API response is 2MB — how do you reduce bandwidth costs?

Question:

Solution:

Response compression reduces bandwidth and improves load times, especially on slow networks.

1️⃣ Gzip Compression

👉 Standard HTTP compression algorithm.

Example: 2MB JSON → gzip → 200KB → 90% size reduction.

Gzip deflate

2️⃣ Brotli Compression

👉 Modern algorithm with better compression than gzip.

Example: Same 2MB JSON → Brotli → 150KB → better than gzip's 200KB.

Brotli better compression

3️⃣ Content-Encoding Header

👉 Server indicates compression used via HTTP header.

Example: Content-Encoding: br → browser auto-decompresses Brotli.

HTTP headers Content-Encoding

4️⃣ Compression Level Trade-offs

👉 Higher compression = smaller size but more CPU.

Example: Brotli level 4 (fast) vs level 11 (max compression, slow).

CPU usage compression levels

5️⃣ Pre-compression for Static Assets

👉 Compress files at build time, serve pre-compressed.

Example: Webpack generates .js.gz and .js.br files → nginx serves pre-compressed.

Pre-compression | static assets

6️⃣ Don't Compress Images/Video

👉 Already-compressed formats see minimal benefit.

Example: JPEG, PNG, MP4 already compressed → gzipping adds overhead, no gain.

Skip compression | binary files

7️⃣ Selective Compression

👉 Compress only responses above size threshold.

Example: Only compress responses > 1KB → small responses not worth CPU overhead.

Threshold | selective compression

💡 Problem 120: Response Compression Strategies

> INTERVIEWER: Your API response is 2MB — how do you reduce bandwidth costs?

Question:

Solution:

Response compression reduces bandwidth and improves load times, especially on slow networks.

1️⃣ Gzip Compression

👉 Standard HTTP compression algorithm.

Example: 2MB JSON → gzip → 200KB → 90% size reduction.

Gzip deflate

---

2 Brotli Compression

👉 Modern algorithm with better compression than gzip.

Example: Same 2MB JSON → Brotli → 150KB → better than gzip's 200KB.

Brotli better compression

---

3 Content-Encoding Header

👉 Server indicates compression used via HTTP header.

Example: Content-Encoding: br → browser auto-decompresses Brotli.

HTTP headers Content-Encoding

---

4 Compression Level Trade-offs

👉 Higher compression = smaller size but more CPU.

Example: Brotli level 4 (fast) vs level 11 (max compression, slow).

CPU usage compression levels

---

5 Pre-compression for Static Assets

👉 Compress files at build time, serve pre-compressed.

Example: Webpack generates .js.gz and .js.br files → nginx serves pre-compressed.

Pre-compression static assets

---

6 Don't Compress Images/Video

👉 Already-compressed formats see minimal benefit.

Example: JPEG, PNG, MP4 already compressed → gzipping adds overhead, no gain.

Skip compression binary files

\n

7️⃣ Selective Compression

👉 Compress only responses above size threshold.

Example: Only compress responses > 1KB → small responses not worth CPU overhead.

Threshold | selective compression

💡 Problem 120: Reducing Global Latency from 3s to <200ms

> INTERVIEWER: US loads in 150ms, India in 3 seconds — how do you fix global latency?

Question:

Solution:

Global performance requires geo-distributed infrastructure and edge caching.

1️⃣ Multi-Region Deployment

👉 Deploy application servers in multiple geographic regions.

Example: Servers in us-east, eu-west, ap-southeast → users routed to nearest region.

AWS regions | GCP multi-region | geo-routing

2️⃣ CDN with Edge Caching

👉 Cache static and dynamic content at edge locations worldwide.

Example: India users hit Mumbai edge location (10ms) instead of US origin (300ms).

CloudFront | Cloudflare | Akamai

3️⃣ GeoDNS Routing

👉 DNS routes users to geographically closest servers.

Example: Route53 routes Indian users to ap-south servers, US users to us-east servers.

Route53 geolocation | traffic policies

4️⃣ Database Read Replicas Per Region

👉 Place read replicas in each region for faster data access.

Example: India replica serves Indian users → 20ms DB latency vs 300ms to US primary.

Cross-region replication | Aurora Global Database

5️⃣ Asset Optimization

👉 Compress images, minify JS/CSS, use modern formats.

Example: 5MB images → WebP compression → 500KB → 10x faster download over slow connections.

Image optimization | WebP | Brotli compression

6️⃣ Connection Optimization

👉 Use HTTP/2, connection pooling, and keep-alive.

Example: HTTP/1.1 = 6 parallel connections; HTTP/2 = multiplexed → faster page loads.

HTTP/2 | HTTP/3 | connection pooling

7️⃣ Pre-loading Critical Resources

👉 Use resource hints to load critical assets early.

Example: for fonts, critical CSS → loaded before parser reaches them.

Resource hints | critical rendering path

💡 Problem 121: Load Testing Before Production

> INTERVIEWER: How do you know your system can handle Black Friday traffic?

Question:

Solution:

Load testing validates system capacity and identifies bottlenecks before they cause outages.

1️⃣ Load Testing Tools

👉 Simulate realistic user traffic patterns.

Example: k6, JMeter, Gatling, Locust → simulate 10K concurrent users.

k6 JMeter Gatling Locust

2️⃣ Realistic Traffic Patterns

👉 Model actual user behavior, not just constant load.

Example: Ramp up over 10 mins → hold 10K users for 30 mins → ramp down.

Traffic patterns ramp-up

3️⃣ Soak Testing

👉 Run sustained load to find memory leaks and resource exhaustion.

Example: Run 5K users for 24 hours → detect slow memory leak in session storage.

Soak testing memory leaks

4️⃣ Spike Testing

👉 Sudden traffic bursts to test auto-scaling.

Example: 1K users → spike to 20K in 1 minute → verify auto-scaling responds in time.

Spike testing auto-scaling

5️⃣ Identify Bottlenecks

👉 Monitor all components during load test.

Example: Database CPU at 90%, app servers at 40% → database is bottleneck.

Bottleneck analysis | monitoring

6️⃣ Test in Production-Like Environment

👉 Staging should match production capacity.

Example: Prod has 10 servers → staging has 10 servers → realistic results.

Staging | production parity

7️⃣ Continuous Load Testing

👉 Automate load tests in CI/CD pipeline.

Example: Every deploy → run 5-minute load test → fail build if p95 latency > 500ms.

CI/CD | automated testing

💡 Problem 122: Practicing Chaos Engineering

> INTERVIEWER: How do you know your system will survive a production failure before it happens?

Question:

Solution:

Chaos engineering proactively tests system resilience by intentionally injecting failures.

1️⃣ Start with Hypotheses

👉 Define what you expect to happen during failure.

Example: 'If one AZ goes down, app should continue serving with <1% error rate.'

Hypothesis expected behavior

2️⃣ Chaos Monkey (Random Instance Termination)

👉 Randomly kill instances to test auto-healing.

Example: Chaos Monkey terminates 1 instance every hour → verify auto-scaling replaces it.

Chaos Monkey Netflix instance failure

3️⃣ Network Latency Injection

👉 Add artificial latency to test timeout handling.

Example: Inject 5s delay to database calls → verify circuit breaker opens.

Latency injection timeout testing

4️⃣ Dependency Failure Testing

👉 Simulate external service outages.

Example: Block payment API → verify graceful degradation → queue payments for later.

Dependency failure graceful degradation

5️⃣ Resource Exhaustion

👉 Consume CPU/memory to test resource limits.

Example: Fill disk to 95% → verify alerting triggers and app handles gracefully.

Resource limits capacity testing

6️⃣ Chaos in Production (Carefully)

👉 Run controlled chaos experiments in production.

Example: Start with off-peak hours, single AZ → gradually increase scope.

Production chaos controlled experiments

7️⃣ Continuous Chaos

👉 Automate chaos experiments as part of CI/CD.

Example: Every deploy → run chaos suite → fail build if resilience tests fail.

Automated chaos | CI/CD integration

💡 Problem 121: Load Testing Before Production

> INTERVIEWER: How do you know your system can handle Black Friday traffic?

Question:

Solution:

Load testing validates system capacity and identifies bottlenecks before they cause outages.

1️⃣ Load Testing Tools

👉 Simulate realistic user traffic patterns.

Example: k6, JMeter, Gatling, Locust → simulate 10K concurrent users.

k6 | JMeter | Gatling | Locust

2️⃣ Realistic Traffic Patterns

👉 Model actual user behavior, not just constant load.

Example: Ramp up over 10 mins → hold 10K users for 30 mins → ramp down.

Traffic patterns | ramp-up

3️⃣ Soak Testing

👉 Run sustained load to find memory leaks and resource exhaustion.

Example: Run 5K users for 24 hours → detect slow memory leak in session storage.

Soak testing | memory leaks

4️⃣ Spike Testing

👉 Sudden traffic bursts to test auto-scaling.

Example: 1K users → spike to 20K in 1 minute → verify auto-scaling responds in time.

Spike testing | auto-scaling

5️⃣ Identify Bottlenecks

👉 Monitor all components during load test.

Example: Database CPU at 90%, app servers at 40% → database is bottleneck.

Bottleneck analysis | monitoring

6️⃣ Test in Production-Like Environment

👉 Staging should match production capacity.

Example: Prod has 10 servers → staging has 10 servers → realistic results.

Staging | production parity

7️⃣ Continuous Load Testing

👉 Automate load tests in CI/CD pipeline.

Example: Every deploy → run 5-minute load test → fail build if p95 latency > 500ms.

CI/CD | automated testing

💡 Problem 122: Practicing Chaos Engineering

> INTERVIEWER: How do you know your system will survive a production failure before it happens?

Question:

Solution:

Chaos engineering proactively tests system resilience by intentionally injecting failures.

1️⃣ Start with Hypotheses

👉 Define what you expect to happen during failure.

Example: 'If one AZ goes down, app should continue serving with <1% error rate.'

Hypothesis | expected behavior

2️⃣ Chaos Monkey (Random Instance Termination)

👉 Randomly kill instances to test auto-healing.

Example: Chaos Monkey terminates 1 instance every hour → verify auto-scaling replaces it.

Chaos Monkey | Netflix | instance failure

3️⃣ Network Latency Injection

👉 Add artificial latency to test timeout handling.

Example: Inject 5s delay to database calls → verify circuit breaker opens.

Latency injection | timeout testing

4️⃣ Dependency Failure Testing

👉 Simulate external service outages.

Example: Block payment API → verify graceful degradation → queue payments for later.

Dependency failure | graceful degradation

5️⃣ Resource Exhaustion

👉 Consume CPU/memory to test resource limits.

Example: Fill disk to 95% → verify alerting triggers and app handles gracefully.

Resource limits capacity testing

6️⃣ Chaos in Production (Carefully)

👉 Run controlled chaos experiments in production.

Example: Start with off-peak hours, single AZ → gradually increase scope.

Production chaos controlled experiments

\n\n

7️⃣ Continuous Chaos

👉 Automate chaos experiments as part of CI/CD.

Example: Every deploy → run chaos suite → fail build if resilience tests fail.

Automated chaos CI/CD integration

💡 Problem 120: Reducing Global Latency from 3s to <200ms

> INTERVIEWER: US loads in 150ms, India in 3 seconds — how do you fix global latency?

Question:

Solution:

Global performance requires geo-distributed infrastructure and edge caching.

1️⃣ Multi-Region Deployment

👉 Deploy application servers in multiple geographic regions.

Example: Servers in us-east, eu-west, ap-southeast → users routed to nearest region.

AWS regions GCP multi-region geo-routing

2️⃣ CDN with Edge Caching

👉 Cache static and dynamic content at edge locations worldwide.

Example: India users hit Mumbai edge location (10ms) instead of US origin (300ms).

CloudFront | Cloudflare | Akamai

3️⃣ GeoDNS Routing

👉 DNS routes users to geographically closest servers.

Example: Route53 routes Indian users to ap-south servers, US users to us-east servers.

Route53 geolocation | traffic policies

4️⃣ Database Read Replicas Per Region

👉 Place read replicas in each region for faster data access.

Example: India replica serves Indian users → 20ms DB latency vs 300ms to US primary.

Cross-region replication | Aurora Global Database

5️⃣ Asset Optimization

👉 Compress images, minify JS/CSS, use modern formats.

Example: 5MB images → WebP compression → 500KB → 10x faster download over slow connections.

Image optimization | WebP | Brotli compression

6️⃣ Connection Optimization

👉 Use HTTP/2, connection pooling, and keep-alive.

Example: HTTP/1.1 = 6 parallel connections; HTTP/2 = multiplexed → faster page loads.

HTTP/2 | HTTP/3 | connection pooling

\n

7️⃣ Pre-loading Critical Resources

👉 Use resource hints to load critical assets early.

Example: for fonts, critical CSS → loaded before parser reaches them.

Resource hints | critical rendering path

💡 Problem 120: Edge Computing and Lambda@Edge

> INTERVIEWER: How can you run code at CDN edge locations?

Question:

Solution:

Edge computing runs code close to users — enables personalization, A/B testing, authentication at edge.

Here's how edge computing works 👇

1️⃣ Lambda@Edge for Request/Response Manipulation

👉 Run code on CloudFront edge to modify requests/responses.

Example: Viewer request → Lambda@Edge adds authentication check → forwards to origin or returns 403.

CloudFront Functions | Lambda@Edge | edge logic

2️⃣ A/B Testing at Edge

👉 Route users to different origins based on cookies or randomization.

Example: 50% of users see version A, 50% see version B → routed at edge.

A/B testing | traffic splitting | edge decision

3️⃣ URL Rewriting and Redirects

👉 Modify URLs or redirect at edge without hitting origin.

Example: `/old-url` → edge function redirects to `/new-url`.

URL rewrite | redirect | edge routing

4️⃣ Bot Detection and Security

👉 Block malicious requests at edge before reaching origin.

Example: Edge function checks user-agent → blocks known bots → protects origin.

WAF | bot detection | DDoS protection

5️⃣ Personalization with Edge Functions

👉 Customize content based on location, device, or headers.

Example: User in France → edge function returns French content → no origin request.

Personalization | localization | edge logic

6️⃣ Edge Compute Limitations

👉 Limited execution time, memory, and no persistent storage.

Example: Lambda@Edge max 5 seconds, 128-256MB → lightweight logic only.

Constraints | stateless | cold starts

📚 Asynchronous Processing

Topic 43: Asynchronous Processing

💡 Problem 121: Background Job Processing

INTERVIEWER: How do you handle long-running tasks without blocking the user?

Question:

Solution:

Move long tasks to background workers — return immediately to user, process asynchronously.

Here's how background job processing works 👇

1️⃣ Job Queue Pattern

👉 Enqueue job, worker picks up and processes asynchronously.

Example: User uploads video → return success immediately → background worker transcodes video.

RabbitMQ | AWS SQS | Redis Queue

2️⃣ Worker Pool for Parallel Processing

👉 Multiple workers process jobs concurrently.

Example: 100 jobs in queue, 10 workers → each worker processes 10 jobs in parallel.

Horizontal scaling | worker instances

3️⃣ Job Retries with Exponential Backoff

👉 Retry failed jobs with increasing delay.

Example: Job fails → retry after 1 second → fails again → retry after 2 seconds → 4 seconds → 8 seconds.

Retry mechanism | exponential backoff | max retries

4️⃣ Job Status Tracking

👉 Store job status so user can check progress.

Example: Job ID returned to user → user polls `/jobs/123/status` → returns "processing", "completed", or "failed".

`Status API` `polling` `webhooks`

5️⃣ Priority Queues

👉 High-priority jobs processed before low-priority.

Example: Paid users → high priority queue, free users → low priority queue.

`Priority levels` `SLA-based processing`

6️⃣ Dead Letter Queue for Failed Jobs

👉 Jobs that fail max retries moved to DLQ for investigation.

Example: Job fails 5 times → moved to DLQ → manual inspection.

`DLQ` `error handling` `manual intervention`

💡 Problem 122: Webhook and Callback Patterns

> INTERVIEWER: How do you notify clients when async jobs complete?

Question:

Solution:

Use webhooks to push results to clients — avoid polling overhead.

Here's how webhook patterns work 👇

1️⃣ Webhook Registration

👉 Client provides callback URL when creating job.

Example: POST `/jobs` with `{"callback_url": "https://client.com/webhook"}` → job completes → POST to callback URL.

Callback URLs | push notifications

2️⃣ Webhook Retries

👉 Retry webhook delivery if client is unavailable.

Example: POST to webhook fails → retry after 1 min → 5 min → 15 min → mark as failed.

Retry policy | exponential backoff

3️⃣ Webhook Signatures for Security

👉 Sign webhook payload so client can verify authenticity.

Example: HMAC-SHA256(payload + secret) → include in header → client verifies signature.

Signature verification | security

4️⃣ Polling as Fallback

👉 Allow clients to poll job status if webhook delivery fails.

Example: Client polls `/jobs/123/status` every 5 seconds until completion.

Polling | fallback mechanism

5️⃣ Webhook Ordering

👉 Include sequence number or timestamp for ordering.

Example: Multiple webhooks for same job → sequence numbers ensure correct order.

Event ordering | sequence numbers

6️⃣ Idempotent Webhook Processing

👉 Client must handle duplicate webhook deliveries.

Example: Webhook delivered twice → client processes idempotently → no duplicate side effects.

Idempotency duplicate handling

📚 Bloom Filters

Topic 44: Bloom Filters

💡 Problem 123: Understanding Bloom Filters

INTERVIEWER: How do you check if an element might be in a set without storing the entire set?

Question:

Solution:

Bloom filters are space-efficient probabilistic data structures — can have false positives, never false negatives.

Here's how Bloom filters work 👇

1️⃣ What is a Bloom Filter?

👉 Bit array with multiple hash functions — space-efficient set membership test.

Example: Store 1M URLs in 1MB Bloom filter instead of 100MB hash set.

Probabilistic space-efficient false positives possible

2️⃣ False Positives, No False Negatives

👉 Bloom filter might say "maybe in set" but never wrong about "not in set".

Example: Check if URL visited → "definitely not visited" (accurate) or "probably visited" (might be false positive).

False positive rate | no false negatives

3️⃣ How It Works

👉 Hash element with K hash functions → set K bits in bit array.

Example: Hash "hello" with 3 functions → set bits 5, 23, 47 → to check membership, verify all 3 bits are set.

Multiple hash functions | bit array

4️⃣ Trade-off: Size vs Accuracy

👉 Larger bit array = lower false positive rate, more hash functions = lower rate.

Example: 1% false positive rate needs 9.6 bits per element, 0.1% needs 14.4 bits.

Size calculation | accuracy tuning

5️⃣ Can't Delete Elements

👉 Clearing a bit might affect other elements.

Example: Deleting one element clears bits → might cause false negatives for other elements.

Immutable | counting Bloom filter variant

6️⃣ Use Cases

👉 Cache filters, spell checkers, malicious URL detection, database query optimization.

Example: Check Bloom filter before expensive database query → skip query if definitely not in DB.

Optimization | pre-filter | cache efficiency

🎉 Congratulations! You've completed all backend problems from ebook! 🎉