System Design Fundamentals & Interview Prep

Complete Guide: Basics to Intermediate Concepts

(New Grad / Junior Engineer Focus | Counter-Question Ready)

📚 Table of Contents

1️⃣ Networking Fundamentals

HTTP/HTTPS

What is HTTP?

HTTP (HyperText Transfer Protocol) is the foundation of data communication on the web. It's a request-response protocol where clients (browsers, mobile apps) send requests to servers, and servers send back responses.

Key Characteristics:

- **Stateless**: Each request is independent (server doesn't remember previous requests)
- **Text-based**: Human-readable format
- **Port**: Default 80 (HTTP), 443 (HTTPS)
- **Methods**: GET, POST, PUT, DELETE, PATCH, etc.

HTTP Request Structure:

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer eyJhbGc...
User-Agent: Mozilla/5.0

{
  "name": "Alice",
  "email": "alice@example.com"
}
```

HTTP Response Structure:

```
HTTP/1.1 201 Created
Content-Type: application/json
Content-Length: 87
Date: Wed, 29 Jan 2026 10:00:00 GMT

{
  "id": 123,
  "name": "Alice",
  "email": "alice@example.com",
  "created_at": "2026-01-29T10:00:00Z"
}
```

HTTP Methods (Verbs)

Method	Purpose	Idempotent?	Safe?	Example
GET	Retrieve resource	Yes	Yes	`GET /api/users/123`

POST	Create resource	No	No	`POST /api/users`
PUT	Replace resource	Yes	No	`PUT /api/users/123`
PATCH	Partially update	No	No	`PATCH /api/users/123`
DELETE	Delete resource	Yes	No	`DELETE /api/users/123`
HEAD	Get headers only	Yes	Yes	`HEAD /api/users/123`
OPTIONS	Get allowed methods	Yes	Yes	`OPTIONS /api/users`

Idempotent: Same request multiple times = same result **Safe**: Doesn't modify server state

---

HTTP Status Codes (Must Know)

2xx - Success:

- **200 OK**: Request succeeded
- **201 Created**: Resource created (POST)
- **204 No Content**: Success, but no response body (DELETE)

3xx - Redirection:

- **301 Moved Permanently**: Resource moved (redirect, SEO-friendly)
- **302 Found**: Temporary redirect
- **304 Not Modified**: Cached version still valid (ETag match)

4xx - Client Error:

- **400 Bad Request**: Invalid request (malformed JSON)
- **401 Unauthorized**: Authentication required
- **403 Forbidden**: Authenticated but no permission
- **404 Not Found**: Resource doesn't exist
- **409 Conflict**: Conflict with current state (duplicate)
- **429 Too Many Requests**: Rate limit exceeded

5xx - Server Error:

- **500 Internal Server Error**: Generic server error
- **502 Bad Gateway**: Proxy/gateway error
- **503 Service Unavailable**: Server overloaded/down
- **504 Gateway Timeout**: Upstream server timeout

HTTP vs HTTPS

Aspect	HTTP	HTTPS
Security	Plaintext (readable)	Encrypted (TLS/SSL)
Port	80	443
Certificate	Not required	SSL/TLS certificate required
Speed	Slightly faster	Slight overhead (encryption)
SEO	Lower ranking	Higher ranking (Google preference)
Use Case	Never (always use HTTPS)	Always (especially for sensitive data)

TLS Handshake (simplified):

```
1. Client → Server: "Hello, I support TLS 1.3"
2. Server → Client: "Hello, here's my certificate" (public key)
3. Client verifies certificate (trusted CA?)
4. Client generates session key, encrypts with server's public key
5. Server decrypts with private key
6. Both use session key for symmetric encryption (fast)
```

HTTP/1.1 vs HTTP/2 vs HTTP/3

Feature	HTTP/1.1	HTTP/2	HTTP/3
Protocol	Text-based	Binary	Binary
Connections	1 request per connection	Multiplexing (many requests/1 connection)	QUIC (UDP-based)
Head-of-Line Blocking	Yes (blocking)	Partially solved	Fully solved

Header Compression	No	Yes (HPACK)	Yes (QPACK)
Server Push	No	Yes	Yes
Adoption	Universal	~50% websites	~25% websites

Multiplexing Example (HTTP/2):

```
HTTP/1.1: 6 connections (browser limit)
→ Download style.css (conn 1)
→ Download script.js (conn 2)
→ Download image1.jpg (conn 3)
→ Download image2.jpg (conn 4)
→ Download image3.jpg (conn 5)
→ Download image4.jpg (conn 6)

HTTP/2: 1 connection (multiplexed)
→ Download all 6 resources simultaneously over single TCP connection
→ No head-of-line blocking at application layer
```

TCP vs UDP

Aspect	TCP	UDP
Connection	Connection-oriented (3-way handshake)	Connectionless
Reliability	Guaranteed delivery, ordered	Best-effort, no guarantee
Speed	Slower (overhead)	Faster (no overhead)
Header Size	20 bytes	8 bytes
Use Cases	Web (HTTP), email, file transfer	Video streaming, gaming, DNS
Error Checking	Yes (retransmission)	Basic checksum (no retransmit)

TCP 3-Way Handshake:

```
Client → Server: SYN (I want to connect)
Server → Client: SYN-ACK (OK, let's connect)
Client → Server: ACK (Connected!)
```

When to Use TCP:

- Data integrity critical (file transfer, API calls)
- Order matters (chat messages)
- Latency acceptable (not real-time gaming)

When to Use UDP:

- Speed > reliability (live video, VoIP)
- Small packets (DNS queries)
- Broadcast (IoT sensors)

DNS (Domain Name System)

Problem: Humans remember "google.com", computers need IP addresses "142.250.185.46"

DNS resolves domain names to IP addresses

DNS Lookup Flow:

```
sequenceDiagram
    participant Browser
    participant OSCache as OS DNS Cache
    participant ISPResolver as ISP DNS Resolver
    participant RootServer as Root DNS Server
    participant TLDServer as TLD Server (.com)
    participant AuthServer as Authoritative Server

    Browser->>OSCache: Resolve "api.example.com"

    alt Cache Hit
        OSCache-->>Browser: 192.0.2.1 (cached)
    else Cache Miss
        OSCache->>ISPResolver: Resolve "api.example.com"
        ISPResolver->>RootServer: Where is .com?
        RootServer-->>ISPResolver: Ask TLD server 1.2.3.4

        ISPResolver->>TLDServer: Where is example.com?
        TLDServer-->>ISPResolver: Ask authoritative server 5.6.7.8

        ISPResolver->>AuthServer: What's IP of api.example.com?
        AuthServer-->>ISPResolver: 192.0.2.1 (TTL: 300s)

        ISPResolver-->>Browser: 192.0.2.1
        Browser->>Browser: Cache for 300 seconds
    end
```

DNS Record Types:

Type	Purpose	Example
A	IPv4 address	`example.com → 192.0.2.1`
AAAA	IPv6 address	`example.com → 2001:db8::1`
CNAME	Alias (canonical name)	`www.example.com → example.com`
MX	Mail server	`example.com → mail.example.com`
TXT	Text record	SPF, DKIM (email verification)
NS	Name server	`example.com → ns1.cloudflare.com`

TTL (Time To Live):

```
example.com.  300  IN  A  192.0.2.1
              ^^^
              TTL = 300 seconds (5 minutes)

DNS resolver caches this for 5 minutes
→ After 5 minutes, re-query authoritative server
```

Why Short TTL?

- Fast failover (switch traffic to backup server)
- Blue-green deployment (gradual traffic shift)

Why Long TTL?

- Reduce DNS query load (fewer queries)
- Faster page loads (cached locally)

WebSocket vs HTTP

HTTP: Request-response (client asks, server answers)

WebSocket: Persistent bidirectional connection (both can send anytime)

Comparison:

Aspect	HTTP	WebSocket
Connection	New connection per request	Single persistent connection
Direction	Request → Response only	Bidirectional (full-duplex)
Overhead	High (headers every request)	Low (headers only during handshake)
Real-time	Polling required	True push (server → client)
Use Cases	REST APIs, web pages	Chat, live updates, gaming

WebSocket Handshake (HTTP → WebSocket upgrade):

```
Client → Server:
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Server → Client:
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=

(Connection upgraded to WebSocket, bidirectional now)
```

Use Cases:

- **HTTP**: Login, fetch user profile, upload file
- **WebSocket**: Real-time chat, live stock prices, multiplayer game

---

Interview Questions: Networking

Q1: "Why is HTTP stateless? Isn't that inefficient?"

Answer: Stateless means server doesn't remember previous requests. Each request contains all info needed.

Benefits:

- **Scalability**: Any server can handle any request (no sticky sessions)
- **Simplicity**: No server-side session management
- **Fault tolerance**: Server crash doesn't lose state

How we add state:

- **Cookies**: Client stores session ID, sends with each request
- **JWT tokens**: Stateless authentication (token contains user info)
- **Session store**: Server stores session in Redis (external state)

Example:

```
// Stateless (every request has auth token)
GET /api/profile
Authorization: Bearer eyJhbGc...

// Server doesn't remember who you are
// Token contains all info (user_id, expiry)
```

Q2: "What happens if I send 1000 HTTP requests simultaneously to same server?"

Answer: Depends on server configuration:

Node.js (single-threaded event loop):

- All 1000 requests queued
- Processed asynchronously (non-blocking I/O)
- If CPU-bound: slow (blocks event loop)
- If I/O-bound: fast (concurrent database queries)

Multi-threaded server (Java, Go):

- Thread pool (e.g., 100 threads)
- 100 requests processed simultaneously
- Remaining 900 queued
- If all threads busy: reject (503 Service Unavailable) or queue

Rate Limiting:

```
// Limit: 100 requests/minute per user
if (requestCount > 100) {
  return res.status(429).json({ error: 'Rate limit exceeded' });
}
```

Q3: "How does HTTPS prevent man-in-the-middle attacks?"

Answer:

1. **Certificate Verification**: Client verifies server's SSL certificate signed by trusted CA
2. **Public Key Encryption**: Client encrypts session key with server's public key
3. **Private Key Decryption**: Only server can decrypt (has private key)
4. **Symmetric Encryption**: Rest of communication uses fast symmetric encryption

Attack Scenario (HTTP):

```
You → WiFi → Hacker → Bank
        ↑
    Hacker reads plaintext password!
```

Protected (HTTPS):

```
You → WiFi → Hacker → Bank
        ↑
    Hacker sees encrypted gibberish
```

2️⃣ Database Concepts

SQL vs NoSQL

Aspect	SQL (Relational)	NoSQL
Schema	Fixed schema (strict)	Flexible schema (dynamic)
Data Model	Tables with rows/columns	Documents, key-value, graph, column-family
Relationships	JOINs (foreign keys)	Denormalized (embed data)
ACID	Strong ACID guarantees	Eventual consistency (BASE)
Scalability	Vertical (bigger server)	Horizontal (more servers)
Transactions	Multi-row transactions	Limited (single-document usually)
Examples	PostgreSQL, MySQL	MongoDB, Cassandra, Redis, DynamoDB

ACID Properties

ACID = Atomicity, Consistency, Isolation, Durability

A - Atomicity

Definition: All-or-nothing. Transaction either completes fully or rolls back fully.

Example (bank transfer):

```sql
BEGIN TRANSACTION;

-- Deduct $100 from Alice
UPDATE accounts SET balance = balance - 100 WHERE user_id = 1;

-- Add $100 to Bob
UPDATE accounts SET balance = balance + 100 WHERE user_id = 2;

COMMIT;  -- Both succeed

-- If power fails after first UPDATE:
-- → ROLLBACK (both updates undone, Alice doesn't lose money)
```

Without atomicity:

- Power fails after first UPDATE
- Alice loses $100, Bob gains $0
- Money disappears! 💸

C - Consistency

Definition: Database moves from one valid state to another (constraints enforced).

Example (foreign key constraint):

```sql
-- Users table
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100)
);

-- Orders table (order must belong to existing user)
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id),
  amount DECIMAL(10, 2)
);

-- This succeeds (user 1 exists)
INSERT INTO orders (user_id, amount) VALUES (1, 99.99);

-- This fails (user 999 doesn't exist)
INSERT INTO orders (user_id, amount) VALUES (999, 99.99);
ERROR: foreign key constraint violated
```

I - Isolation

Definition: Concurrent transactions don't interfere with each other.

Problem (without isolation):

```
Alice's balance: $100

Transaction 1: Read balance ($100), add $50 → Write $150
Transaction 2: Read balance ($100), subtract $30 → Write $70

Final balance: $70 (Lost Transaction 1's update!)
Should be: $120
```

Solution: Isolation levels (see below)

D - Durability

Definition: Once committed, data persists (even if server crashes).

How?

- **Write-Ahead Log (WAL)**: Log changes to disk before applying
- **Replication**: Copy data to multiple servers
- **Snapshots**: Periodic backups

Example:

```
Client: INSERT INTO users VALUES (1, 'Alice');
Server: Write to WAL → Flush to disk → Return "OK"

(Power fails 1 second later)

Server restarts: Replay WAL → Data recovered
```

Isolation Levels

Level	Dirty Read	Non-Repeatable Read	Phantom Read	Performance
Read Uncommitted	Yes	Yes	Yes	Fastest
Read Committed	No	Yes	Yes	Fast (default)
Repeatable Read	No	No	Yes	Slower
Serializable	No	No	No	Slowest

Definitions:

- **Dirty Read**: Read uncommitted data from another transaction
- **Non-Repeatable Read**: Same query returns different results (another transaction committed)
- **Phantom Read**: Same query returns different rows (INSERT/DELETE by another transaction)

Example (Non-Repeatable Read):

```
-- Transaction 1
BEGIN;
SELECT balance FROM accounts WHERE user_id = 1;  -- Returns $100
-- (wait 5 seconds)
SELECT balance FROM accounts WHERE user_id = 1;  -- Returns $150 (changed!)
COMMIT;

-- Transaction 2 (concurrent)
BEGIN;
```

```
UPDATE accounts SET balance = 150 WHERE user_id = 1;
COMMIT;
```

When to use Serializable:

- Financial transactions (absolute consistency)
- Inventory management (prevent overselling)

When to use Read Committed (default):

- Most applications (good balance)

Indexes

Purpose: Speed up queries (like book index)

Without Index:

```
SELECT * FROM users WHERE email = 'alice@example.com';

-- Database scans ALL rows (O(n))
-- 1 million users → 1 million rows scanned → slow!
```

With Index:

```
CREATE INDEX idx_users_email ON users(email);

SELECT * FROM users WHERE email = 'alice@example.com';

-- Database uses index (O(log n))
-- 1 million users → ~20 comparisons → fast!
```

Index Types

1. B-Tree Index (default, most common)

- Balanced tree structure
- Good for: =, <, >, <=, >=, BETWEEN, ORDER BY
- Example: `CREATE INDEX idx_users_age ON users(age);`

2. Hash Index

- Hash table
- Good for: = only (no range queries)
- Example: `CREATE INDEX idx_users_email ON users USING HASH(email);`

3. GiST Index (Generalized Search Tree)

- Spatial data, full-text search
- Example: `CREATE INDEX idx_locations ON places USING GIST(location);`

4. Composite Index

- Multiple columns
- Example: `CREATE INDEX idx_users_name_age ON users(name, age);`
    - Fast: `WHERE name = 'Alice' AND age = 30`
    - Fast: `WHERE name = 'Alice'` (uses leftmost column)
    - Slow: `WHERE age = 30` (doesn't use index)

Index Trade-offs

Pros:

- ✅ Fast reads (queries)
- ✅ Fast sorting (ORDER BY)
- ✅ Unique constraint (UNIQUE index)

Cons:

- ❌ Slow writes (INSERT, UPDATE, DELETE must update index)
- ❌ Storage overhead (index takes disk space)
- ❌ Too many indexes → all must be updated on write

Example:

```
-- Table: 1 million rows, 5 indexes
-- Without indexes:
INSERT takes 1ms (just write to table)

-- With 5 indexes:
INSERT takes 10ms (write to table + update 5 indexes)
```

Rule of Thumb:

- Index columns in WHERE, JOIN, ORDER BY clauses
- Don't index low-cardinality columns (e.g., gender: M/F only)
- Monitor query performance (EXPLAIN ANALYZE)

Normalization vs Denormalization

Normalization (SQL)

Goal: Reduce redundancy, enforce consistency

Example (normalized):

```
-- Users table
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  email VARCHAR(100)
);

-- Orders table
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id),
  amount DECIMAL(10, 2),
  created_at TIMESTAMP
);

-- Query: Get user's orders (requires JOIN)
SELECT u.name, o.amount, o.created_at
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE u.id = 123;
```

Pros:

- ✅ No redundancy (user name stored once)
- ✅ Easy updates (change name in one place)
- ✅ Data integrity (foreign key constraints)

Cons:

- ❌ JOINs required (slower reads)
- ❌ Complex queries

Denormalization (NoSQL)

Goal: Fast reads (embed related data)

Example (denormalized):

```
// MongoDB document (no JOINs needed)
{
  "_id": 123,
  "name": "Alice",
  "email": "alice@example.com",
  "orders": [
    {
      "id": 1,
      "amount": 99.99,
      "created_at": "2026-01-29T10:00:00Z"
    },
    {
      "id": 2,
      "amount": 49.99,
      "created_at": "2026-01-28T15:30:00Z"
    }
  ]
}

// Query: Single document fetch (no JOIN)
db.users.findOne({ _id: 123 });
```

Pros:

- ✅ Fast reads (single query, no JOINs)
- ✅ Simple queries

Cons:

- ❌ Data redundancy (user name duplicated in orders)
- ❌ Hard to update (must update all copies)
- ❌ Consistency issues (stale data)

Sharding (Horizontal Partitioning)

Problem: Database too large for single server (1 TB, 100M rows)

Solution: Split data across multiple servers

Sharding Strategies:

1. Hash-Based Sharding

```
user_id % num_shards = shard_id

Example: 3 shards
- user_id = 1 → 1 % 3 = 1 → Shard 1
- user_id = 2 → 2 % 3 = 2 → Shard 2
- user_id = 3 → 3 % 3 = 0 → Shard 0
- user_id = 4 → 4 % 3 = 1 → Shard 1
```

Pros:

- ✅ Even distribution
- ✅ Simple logic

Cons:

- ❌ Hard to rebalance (adding shard changes hash)
- ❌ Range queries slow (must query all shards)

---

2. Range-Based Sharding

```
Shard 0: user_id 0-999,999
Shard 1: user_id 1,000,000-1,999,999
Shard 2: user_id 2,000,000-2,999,999
```

Pros:

- ✅ Range queries fast (query single shard)
- ✅ Easy to add shards (just split range)

Cons:

- ❌ Uneven distribution (new users go to latest shard)
- ❌ Hot shards (if user_id 2M-3M are very active)

---

3. Geographic Sharding

```
Shard 0: US users
Shard 1: EU users
Shard 2: Asia users
```

Pros:

- ✅ Low latency (data close to users)
- ✅ Compliance (GDPR: EU data in EU)

Cons:

- ❌ Uneven distribution (US has more users)
- ❌ Cross-region queries slow

Replication

Purpose: High availability, fault tolerance, read scalability

Master-Slave Replication:

```
           [Master] (writes)
          /    |    \
         /     |     \
    [Slave] [Slave] [Slave] (reads)
```

Write Flow:

```
1. Client writes to Master
2. Master commits to database
3. Master sends changes to Slaves (replication log)
4. Slaves apply changes
5. Master returns success to client
```

Read Flow:

```
Client reads from any Slave
→ Lower load on Master
→ Higher throughput (3 Slaves = 4× read capacity)
```

Replication Lag:

```
10:00:00 - Master: INSERT user (Alice)
10:00:02 - Slave 1: Applied (2 second lag)
10:00:05 - Slave 2: Applied (5 second lag)

Client writes, then immediately reads from Slave 2:
→ User not found! (stale read)
```

Solutions:

- **Read from Master**: After write, read from Master (consistent)
- **Sticky sessions**: Always read from same Slave
- **Synchronous replication**: Wait for Slaves before returning (slow)

CAP Theorem

CAP = Consistency, Availability, Partition Tolerance

The Theorem: You can have at most 2 out of 3 during network partition

Definitions:

- **Consistency (C)**: All nodes see same data at same time
- **Availability (A)**: Every request gets response (even if stale)
- **Partition Tolerance (P)**: System works despite network split

Network Partition Example:

```
     [Server A]  <-- X -->  [Server B]
         ↑                       ↑
     (US users)            (EU users)

Network cable cut! (partition)
```

Choice:

- **CP (Consistency + Partition Tolerance)**: Reject requests until partition healed (some servers unavailable)
    - Example: Banking system (never show wrong balance)
- **AP (Availability + Partition Tolerance)**: Serve requests with possibly stale data (always available)
    - Example: Social media feed (OK if slightly outdated)

Real-World Systems:

- **PostgreSQL**: CP (rejects writes during partition)
- **Cassandra**: AP (accepts writes, eventual consistency)
- **MongoDB**: Tunable (can choose CP or AP)

Interview Questions: Databases

Q1: "Why not just index every column?"

Answer: Indexes slow down writes:

```
-- Table: users (10 columns, 10 indexes)

INSERT INTO users VALUES (...);
→ Write to table (1 disk write)
→ Update 10 indexes (10 disk writes)
→ Total: 11 disk writes (11× slower!)

UPDATE users SET email = 'new@example.com' WHERE id = 123;
→ Update table
→ Update email index (remove old entry, add new entry)
→ 2 disk writes
```

Rule: Index only columns in WHERE, JOIN, ORDER BY

Trade-off:

- More indexes = faster reads, slower writes
- Fewer indexes = slower reads, faster writes

Q2: "How would you store 100 million users in a database?"

Answer:

Option 1: Vertical Scaling (bigger server)

- Single PostgreSQL server with 512 GB RAM, 10 TB SSD
- Pros: Simple, ACID transactions work
- Cons: Expensive, single point of failure, limited scalability

Option 2: Sharding (horizontal scaling)

- 10 shards × 10M users each
- Hash by `user_id % 10`
- Pros: Scales horizontally, cheaper
- Cons: Complex (routing, cross-shard queries)

Option 3: Replication (read replicas)

- 1 master + 9 read replicas
- All writes to master, reads distributed
- Pros: 10× read capacity, simple
- Cons: Replication lag, doesn't help with write load

Recommended: Sharding + Replication

```
Shard 0: [Master] + [Replica]
Shard 1: [Master] + [Replica]
Shard 2: [Master] + [Replica]
...
```

Q3: "Database is slow. How do you debug?"

Answer (systematic approach):

Step 1: Identify slow queries

```
-- PostgreSQL: Enable slow query log
SET log_min_duration_statement = 1000;  -- Log queries > 1 second

-- Check logs
SHOW log_destination;
```

Step 2: Analyze query plan

```
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'alice@example.com';

→ Seq Scan on users (cost=0..1000 rows=1000)
→ Sequential scan! (no index)

-- Add index
CREATE INDEX idx_users_email ON users(email);
```

```
→ Index Scan on users (cost=0..10 rows=1)
→ 100× faster!
```

Step 3: Check missing indexes

```sql
-- Find queries not using indexes
SELECT * FROM pg_stat_user_tables WHERE idx_scan = 0;
```

Step 4: Check database load

```
# CPU usage
top

# Disk I/O
iostat -x 1

# Active connections
SELECT count(*) FROM pg_stat_activity;
```

Step 5: Optimize

- Add indexes (WHERE, JOIN columns)
- Use connection pooling (don't open new connection per request)
- Cache frequently accessed data (Redis)
- Denormalize (avoid JOINs)
- Shard (distribute load)

3️⃣ Caching Strategies

What is a Cache?

Cache: Temporary storage for frequently accessed data (fast read, slow write)

Why Cache?

- Database query: 50ms
- Redis cache: 1ms
- **50× speedup!**

Cache Hierarchy

```
Client (Browser) → CDN → Server → Redis Cache → Database
     ↓               ↓       ↓          ↓            ↓
   Local       Edge     API      In-memory    Persistent
  Storage      Cache    Server     Cache        Storage
   100ms       50ms     10ms        1ms          50ms
```

Cache Patterns

1. Cache-Aside (Lazy Loading)

Flow:

```
1. Check cache
2. If hit: return data
3. If miss: query database, write to cache, return data
```

Code:

```typescript
async function getUser(userId: number): Promise<User> {
  // 1. Check cache
  const cached = await redis.get(`user:${userId}`);
  if (cached) {
    console.log('Cache hit');
    return JSON.parse(cached);
  }

  // 2. Cache miss, query database
  console.log('Cache miss');
  const user = await db.query('SELECT * FROM users WHERE id = $1', [userId]);

  // 3. Write to cache (TTL: 1 hour)
  await redis.setex(`user:${userId}`, 3600, JSON.stringify(user));

  return user;
}
```

Pros:

- ✅ Only cache what's requested (no waste)
- ✅ Cache failure doesn't break app (degrades gracefully)

Cons:

- ❌ First request slow (cache miss)
- ❌ Cache invalidation manual

---

2. Write-Through

Flow:

```
1. Write to cache
2. Write to database (synchronously)
3. Return success
```

Code:

```
async function updateUser(userId: number, data: Partial<User>): Promise<void> {
  // 1. Update database
  await db.query('UPDATE users SET name = $1 WHERE id = $2', [data.name, userId]);

  // 2. Update cache (synchronously)
  const updatedUser = await db.query('SELECT * FROM users WHERE id = $1', [userId]);
  await redis.setex(`user:${userId}`, 3600, JSON.stringify(updatedUser));
}
```

Pros:

- ✅ Cache always fresh (no stale data)
- ✅ Read after write consistent

Cons:

- ❌ Slower writes (2 operations)
- ❌ Writes to infrequently accessed data (waste)

---

3. Write-Behind (Write-Back)

Flow:

```
1. Write to cache (fast)
2. Return success immediately
3. Async worker writes to database (batched)
```

Code:

```
async function updateUser(userId: number, data: Partial<User>): Promise<void> {
  // 1. Update cache immediately
  await redis.setex(`user:${userId}`, 3600, JSON.stringify(data));

  // 2. Queue database write (async)
  await queue.add('update-user', { userId, data });

  // Return immediately (fast!)
}

// Background worker
async function processUserUpdate(job: { userId: number, data: Partial<User> }) {
  await db.query('UPDATE users SET name = $1 WHERE id = $2', [job.data.name, job.userId]);
}
```

Pros:

- ✅ Very fast writes (async database)
- ✅ Batch writes (fewer database operations)

Cons:

- ❌ Data loss if cache crashes before write
- ❌ Complex (queue management)

Cache Eviction Policies

Problem: Cache full, need to remove items

1. LRU (Least Recently Used)

Logic: Remove item not accessed for longest time

Example:

```
Cache size: 3

Access pattern: A, B, C, A, D

1. Access A: [A]
2. Access B: [A, B]
3. Access C: [A, B, C] (full)
4. Access A: [B, C, A] (A moved to front)
5. Access D: [C, A, D] (B evicted, least recently used)
```

Implementation (linked list + hash map):

```
class LRUCache<K, V> {
  capacity: number;
  cache: Map<K, Node<K, V>>;
  head: Node<K, V>;  // Most recently used
  tail: Node<K, V>;  // Least recently used

  get(key: K): V | undefined {
    const node = this.cache.get(key);
    if (!node) return undefined;

    // Move to front (most recently used)
    this.moveToFront(node);
    return node.value;
  }

  set(key: K, value: V): void {
    let node = this.cache.get(key);

    if (node) {
      // Update existing
      node.value = value;
      this.moveToFront(node);
    } else {
      // Add new
      node = new Node(key, value);
      this.cache.set(key, node);
      this.addToFront(node);

      // Evict if full
      if (this.cache.size > this.capacity) {
        const removed = this.removeTail();
        this.cache.delete(removed.key);
      }
    }
  }
}
```

---

2. LFU (Least Frequently Used)

Logic: Remove item accessed fewest times

Example:

```
Cache size: 3

Access pattern: A, B, C, A, A, D

Frequency count:
- A: 3 times
- B: 1 time
- C: 1 time

D requested, cache full:
→ Evict B or C (both frequency 1)
→ Choose based on recency (LRU tiebreaker)
```

---

3. FIFO (First In First Out)

Logic: Remove oldest item (regardless of usage)

Example:

```
Cache size: 3

Insert order: A, B, C, D

D inserted:
→ Evict A (oldest)
→ Cache: [B, C, D]
```

Cache Invalidation

Problem: Data in cache becomes stale (database updated, cache not)

Strategy 1: TTL (Time To Live)

Set expiration time:

```
// Cache for 1 hour
await redis.setex('user:123', 3600, JSON.stringify(user));

// After 1 hour, key auto-deleted
// Next request: cache miss → fetch from database
```

Pros: Simple **Cons**: Data stale until expiry (might serve outdated data)

Strategy 2: Explicit Invalidation

Delete cache on write:

```
async function updateUser(userId: number, data: Partial<User>) {
  // 1. Update database
  await db.query('UPDATE users SET name = $1 WHERE id = $2', [data.name, userId]);

  // 2. Delete cache (force refresh)
  await redis.del(`user:${userId}`);
}
```

Pros: Always fresh **Cons**: Manual (easy to forget)

Strategy 3: Event-Based Invalidation

Publish event on write:

```
// Service A: Updates user
async function updateUser(userId: number, data: Partial<User>) {
  await db.query('UPDATE users SET name = $1 WHERE id = $2', [data.name, userId]);

  // Publish event
  await kafka.publish('user-updated', { userId });
}

// Service B: Listens for events
kafka.subscribe('user-updated', async (event) => {
  // Invalidate cache
  await redis.del(`user:${event.userId}`);
});
```

Pros: Decoupled, scalable **Cons**: Complex (event infrastructure)

Thundering Herd Problem

Problem: Cache expires, 1000 requests hit database simultaneously

Scenario:

```
10:00:00 - Cache expires
10:00:01 - 1000 requests arrive
10:00:01 - All 1000 check cache: MISS
10:00:01 - All 1000 query database simultaneously
10:00:01 - Database overloaded! 💥
```

Solution 1: Lock

```
async function getUserSafely(userId: number): Promise<User> {
  const cached = await redis.get(`user:${userId}`);
  if (cached) return JSON.parse(cached);

  // Acquire lock (only one request fetches from DB)
  const lock = await redis.set(`lock:user:${userId}`, '1', 'NX', 'EX', 10);

  if (lock) {
    // This request won the lock, fetch from DB
    const user = await db.query('SELECT * FROM users WHERE id = $1', [userId]);
    await redis.setex(`user:${userId}`, 3600, JSON.stringify(user));
    await redis.del(`lock:user:${userId}`);
    return user;
  } else {
    // Another request is fetching, wait and retry
    await new Promise(resolve => setTimeout(resolve, 100));
    return getUserSafely(userId);  // Retry
  }
}
```

Solution 2: Probabilistic Early Expiration

```
// Refresh cache before it expires (randomly)
const ttl = await redis.ttl(`user:${userId}`);
const randomThreshold = Math.random() * 60;  // 0-60 seconds

if (ttl < randomThreshold) {
  // Refresh early (before expiration)
  asyncRefreshCache(userId);
}
```

Interview Questions: Caching

Q1: "Why not cache everything forever?"

Answer:

- **Memory limit**: Cache size limited (Redis: 64 GB typical)
- **Stale data**: Database changes, cache outdated
- **Cache invalidation**: Hard to know when to refresh

Better approach: Cache hot data (frequently accessed), short TTL

Q2: "Redis crashes. Does your app break?"

Answer: Depends on pattern:

Cache-Aside (graceful degradation):

```
try {
  const cached = await redis.get(key);
  if (cached) return cached;
} catch (err) {
  console.log('Redis down, fallback to database');
}

// Always query database as fallback
return await db.query(...);
```

Write-Through (breaks if Redis required):

```
await db.query(...);
await redis.set(...);  // If fails, transaction incomplete
```

Best practice: Cache should enhance performance, not be critical path

---

Q3: "How do you cache paginated results?"

Answer:

Option 1: Cache each page

```
// Cache key includes page number
const cacheKey = `users:page:${pageNum}`;
await redis.setex(cacheKey, 300, JSON.stringify(users));
```

Pros: Simple **Cons**: Cache invalidation hard (user added, all pages stale)

Option 2: Cache entire dataset

```
// Cache all users, paginate in-memory
const allUsers = await redis.get('users:all');
const page = allUsers.slice(offset, offset + limit);
```

Pros: Consistent pagination **Cons**: Memory intensive (if dataset large)

Option 3: Don't cache pagination (use database with indexes)

4️⃣ Load Balancing

What is a Load Balancer?

Purpose: Distribute incoming requests across multiple servers

Without Load Balancer:

```
Client → Server (100% traffic, overloaded)
```

With Load Balancer:

```
                [Load Balancer]
               /       |       \
              /        |        \
        [Server 1] [Server 2] [Server 3]
          33%        33%        34%
```

Load Balancing Algorithms

1. Round Robin

Logic: Distribute requests evenly in order

Example:

```
Request 1 → Server 1
Request 2 → Server 2
Request 3 → Server 3
Request 4 → Server 1 (cycle repeats)
```

Pros:

- ✅ Simple
- ✅ Fair distribution

Cons:

- ❌ Doesn't consider server load (Server 1 might be slow)
- ❌ Doesn't consider request cost (video upload vs health check)

---

2. Least Connections

Logic: Send request to server with fewest active connections

Example:

```
Server 1: 10 connections
Server 2: 5 connections
Server 3: 8 connections

New request → Server 2 (fewest connections)
```

Pros:

- ✅ Considers server load (busy servers get fewer requests)

Cons:

- ❌ Doesn't consider request duration (short vs long)

---

3. Weighted Round Robin

Logic: Assign weights to servers (powerful servers get more traffic)

Example:

```
Server 1: weight 5 (powerful)
Server 2: weight 3
Server 3: weight 2

Distribution:
Server 1: 50% (5/10)
Server 2: 30% (3/10)
Server 3: 20% (2/10)
```

Use case: Heterogeneous servers (different CPU/RAM)

---

4. IP Hash

Logic: Hash client IP, route to same server (session affinity)

Example:

```
hash(client_ip) % num_servers = server_id

Client 192.168.1.1 → hash → Server 2 (always)
Client 192.168.1.2 → hash → Server 1 (always)
```

Pros:

- ✅ Sticky sessions (same client → same server)
- ✅ Useful for stateful apps (session in memory)

Cons:

- ❌ Uneven distribution (some IPs more common)
- ❌ Adding/removing server changes hash (remapping)

Layer 4 vs Layer 7 Load Balancing

Aspect	Layer 4 (TCP/UDP)	Layer 7 (HTTP)
OSI Layer	Transport layer	Application layer
Routing Decision	IP + Port	URL, headers, cookies
Speed	Fast (simple)	Slower (parse HTTP)
Features	Basic	Advanced (URL routing, SSL termination)
Example	AWS NLB	AWS ALB, Nginx

Layer 4 Example:

```
Client IP: 1.2.3.4 → Server 1
Client IP: 5.6.7.8 → Server 2

(Routes based on IP, doesn't inspect HTTP request)
```

Layer 7 Example:

```
/api/users → Backend Server
/static/*  → CDN
/admin/*   → Admin Server
```

```
(Routes based on URL path, inspects HTTP request)
```

Health Checks

Purpose: Detect unhealthy servers, remove from pool

Implementation:

```
// Load balancer pings each server every 10 seconds
GET /health

// Server response:
HTTP/1.1 200 OK
{ "status": "healthy", "version": "1.2.3" }

// If 3 consecutive failures:
// → Remove server from pool
// → Stop routing traffic
// → Send alert
```

Health Check Types:

- **Shallow**: Ping endpoint (checks if server responding)
- **Deep**: Check database connection (checks if fully functional)

Interview Questions: Load Balancing

Q1: "Load balancer is single point of failure. How to fix?"

Answer: **Multiple load balancers** (active-passive or active-active)

Active-Passive:

```
          [LB 1] (active)     [LB 2] (standby)
             ↓                      ↓
     Virtual IP: 10.0.0.100

If LB 1 fails:
→ LB 2 takes over Virtual IP (failover)
→ Clients still use 10.0.0.100 (transparent)
```

Active-Active:

```
DNS round-robin:
  lb.example.com → [10.0.0.100, 10.0.0.101]

Client 1 → 10.0.0.100 (LB 1)
Client 2 → 10.0.0.101 (LB 2)
```

Q2: "Server handling long-running request. Load balancer thinks it's dead?"

Answer: **Timeout configuration**:

```
Health check timeout: 5 seconds
Connection timeout: 60 seconds (for long requests)

Health check sees server busy but responsive:
→ Status: "healthy"

If no response for 60 seconds:
→ Status: "timeout" (retry)
```

Better: Async processing

```
Client → Server: Start video transcode
Server → Client: 202 Accepted, job_id: 123

Client polls: GET /jobs/123/status
→ { "status": "processing", "progress": "50%" }
→ { "status": "complete", "url": "..." }
```

5️⃣ Content Delivery Networks (CDN)

What is a CDN?

CDN: Network of geographically distributed servers that cache content close to users

Without CDN:

```
User in Tokyo → Request → Server in US (200ms latency)
```

With CDN:

```
User in Tokyo → Request → CDN in Tokyo (20ms latency, 10× faster!)
```

How CDN Works

```
graph LR
    User[User in Tokyo]
    CDNTokyo[CDN Edge<br/>Tokyo]
    CDNSF[CDN Edge<br/>San Francisco]
    Origin[Origin Server<br/>US East]

    User -->|1. Request image| CDNTokyo
    CDNTokyo -->|2. Cache miss| Origin
    Origin -->|3. Return image| CDNTokyo
    CDNTokyo -->|4. Cache + serve| User

    User2[Another user<br/>in Tokyo] -->|5. Request same image| CDNTokyo
    CDNTokyo -->|6. Cache hit, serve immediately| User2
```

CDN Use Cases

1. Static Assets (images, CSS, JavaScript)

```html
<!-- Without CDN -->
<img src="https://api.example.com/images/logo.png">

<!-- With CDN -->
<img src="https://cdn.example.com/images/logo.png">
```

2. Video Streaming

```
Netflix: 95% of traffic served from CDN
→ Videos cached at ISP level
→ Faster streaming, lower Netflix bandwidth costs
```

3. API Responses (with caching headers)

```
GET /api/products

Response:
Cache-Control: public, max-age=3600
ETag: "abc123"

CDN caches response for 1 hour
```

Cache-Control Headers

```
Cache-Control: public, max-age=31536000, immutable
               ↑       ↑                  ↑
               │       │                  └ Never changes (safe to cache forever)
               │       └ Cache for 1 year (31536000 seconds)
               └ Can be cached by CDN + browser
```

Common Directives:

Directive	Meaning	Use Case
`public`	Can be cached by CDN + browser	Static assets

`private`	Only browser cache (not CDN)	User-specific data
`no-cache`	Must revalidate (check ETag)	Dynamic data
`no-store`	Never cache	Sensitive data
`max-age=N`	Cache for N seconds	Any
`immutable`	Never changes (don't revalidate)	Versioned assets

CDN Invalidation

Problem: Updated logo.png, but CDN still serves old version

Solution 1: Wait for TTL (simple but slow)

```
Cache-Control: max-age=3600

Wait 1 hour, then CDN refetches
```

Solution 2: Purge Cache (immediate but manual)

```
# CloudFlare API
curl -X POST "https://api.cloudflare.com/client/v4/zones/{zone_id}/purge_cache" \
  -H "Authorization: Bearer {token}" \
  -d '{"files":["https://cdn.example.com/images/logo.png"]}'
```

Solution 3: Versioned URLs (best practice)

```
<!-- Old version -->
<img src="https://cdn.example.com/images/logo.v1.png">

<!-- New version -->
<img src="https://cdn.example.com/images/logo.v2.png">

Different URL → CDN treats as new file → no invalidation needed
```

CDN Cost Optimization

CloudFront Pricing (simplified, 2026):

- First 10 TB: $0.085/GB
- Next 40 TB: $0.080/GB
- Over 150 TB: $0.060/GB

Example:

```
Website serves 100 TB/month

Without CDN:
→ Origin bandwidth: 100 TB × $0.12/GB = $12,000/month

With CDN (95% cache hit rate):
→ CDN bandwidth: 100 TB × $0.070/GB = $7,000/month
→ Origin bandwidth: 5 TB × $0.12/GB = $600/month
→ Total: $7,600/month
→ Savings: $4,400/month (37% savings)
```

Interview Questions: CDN

Q1: "CDN serves stale content. How to prevent?"

Answer: **Use versioned URLs**:

```
<!-- Build tool generates hash -->
<link rel="stylesheet" href="/static/main.a1b2c3.css">
<script src="/static/app.d4e5f6.js"></script>

New deployment:
→ Hash changes (content changed)
→ New URL (main.x7y8z9.css)
→ CDN cache miss (fetches new version)
→ No stale content!
```

Bonus: Set long TTL for versioned assets

```
Cache-Control: public, max-age=31536000, immutable
```

Q2: "Should you CDN-cache API responses?"

Answer: **Depends on data**:

Good candidates:

- Public data (product catalog, blog posts)
- Rarely changes (countries list, categories)
- High read-to-write ratio (100 reads per 1 write)

Bad candidates:

- User-specific data (profile, orders)
- Frequently changes (stock prices, live scores)
- POST/PUT/DELETE requests (write operations)

Example:

```
// Good: Cache product list (public, rarely changes)
app.get('/api/products', (req, res) => {
  res.set('Cache-Control', 'public, max-age=300');  // 5 minutes
  res.json(products);
});

// Bad: Don't cache user profile (user-specific)
app.get('/api/profile', (req, res) => {
  res.set('Cache-Control', 'private, no-cache');
  res.json(userProfile);
});
```

6️⃣ API Design

REST API Principles

REST = Representational State Transfer

6 Constraints:

1. **Client-Server**: Separation of concerns
2. **Stateless**: Each request self-contained
3. **Cacheable**: Responses explicitly cacheable or not
4. **Uniform Interface**: Consistent naming, HTTP methods
5. **Layered System**: Client doesn't know if talking to origin or proxy
6. **Code on Demand** (optional): Server can send executable code (JavaScript)

RESTful URL Design

Good Practices:

```
GET    /users           (list users)
GET    /users/123       (get user)
POST   /users           (create user)
PUT    /users/123       (update user, full replacement)
PATCH  /users/123       (update user, partial)
DELETE /users/123       (delete user)

GET    /users/123/orders       (list user's orders)
POST   /users/123/orders       (create order for user)
GET    /users/123/orders/456   (get specific order)
```

Bad Practices (non-RESTful):

```
❌ GET  /getUser?id=123        (use GET /users/123)
❌ POST /createUser            (use POST /users)
❌ POST /deleteUser            (use DELETE /users/123)
❌ GET  /users/delete/123      (GET shouldn't modify, use DELETE)
```

Pagination

Problem: Returning 1 million users in single response (slow, huge payload)

Solution: Pagination

Offset-Based Pagination

```
GET /users?page=2&limit=20

Response:
{
  "data": [...],
  "pagination": {
    "page": 2,
    "limit": 20,
    "total": 1000,
    "pages": 50
  }
}
```

SQL:

```sql
SELECT * FROM users
ORDER BY created_at DESC
LIMIT 20 OFFSET 20;  -- Page 2 (skip first 20)
```

Pros: Simple, can jump to any page **Cons**: Slow for deep pagination (OFFSET 1000000 is slow)

Cursor-Based Pagination

```
GET /users?cursor=abc123&limit=20

Response:
{
  "data": [...],
  "next_cursor": "def456",
  "has_more": true
}

Next request:
GET /users?cursor=def456&limit=20
```

SQL:

```sql
-- First page
SELECT * FROM users
WHERE id > 0
ORDER BY id
LIMIT 20;

-- Next page (cursor = last id from previous page)
SELECT * FROM users
WHERE id > 20
ORDER BY id
LIMIT 20;
```

Pros: Fast (uses index), consistent results (no skipped/duplicate items) **Cons**: Can't jump to specific page

---

Versioning

Why Version APIs?

- Breaking changes (renamed field, removed endpoint)
- Clients can't update immediately
- Need backward compatibility

Methods:

1. URL Path Versioning

```
GET /v1/users
GET /v2/users

(Most common, clear)
```

2. Header Versioning

```
GET /users
Accept: application/vnd.example.v2+json

(Clean URLs, but hidden)
```

3. Query Parameter

```
GET /users?version=2

(Simple, but messy)
```

Idempotency

Definition: Same request executed multiple times = same result

Why Important?

- Network failures (client retries)
- Prevent duplicate charges (payment)
- Safe to retry

Idempotent Methods:

- GET, PUT, DELETE: Idempotent
- POST: Not idempotent (creates new resource each time)

Example (Payment):

```
// Non-idempotent (BAD)
POST /payments
{ "amount": 100 }

→ Client retries 3 times (network timeout)
→ Charged 3 times! ($300 instead of $100)

// Idempotent (GOOD)
POST /payments
{
  "amount": 100,
  "idempotency_key": "uuid-123"  // Client-generated
}

Server logic:
if (existingPayment = db.get(idempotency_key)) {
  return existingPayment;  // Already processed, return cached result
} else {
  processPayment();
  db.save(idempotency_key, payment);
  return payment;
}
```

Rate Limiting

Purpose: Prevent abuse, protect server from overload

Common Limits:

```
Unauthenticated: 100 requests/hour
Authenticated: 1000 requests/hour
Premium: 10,000 requests/hour
```

Headers:

```
HTTP/1.1 200 OK
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 987
X-RateLimit-Reset: 1643723400

(987 requests remaining, resets at Unix timestamp 1643723400)
```

When Exceeded:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 3600

{
  "error": "Rate limit exceeded. Try again in 1 hour."
}
```

Error Handling

Good Error Response:

```
HTTP/1.1 400 Bad Request
Content-Type: application/json

{
  "error": {
    "code": "INVALID_EMAIL",
    "message": "Email address is invalid",
    "field": "email",
    "details": "Email must contain @ symbol"
  }
}
```

Best Practices:

- Use appropriate status codes (400 vs 404 vs 500)
- Include error code (machine-readable)
- Include message (human-readable)
- Include field (which field caused error)
- Never expose stack traces (security risk)

Interview Questions: API Design

Q1: "Design API for Twitter-like app"

Answer:

```
# Authentication
POST /auth/login
POST /auth/logout
POST /auth/refresh

# Users
GET  /users/:userId
PUT  /users/:userId
GET  /users/:userId/followers
GET  /users/:userId/following
POST /users/:userId/follow
DELETE /users/:userId/follow

# Tweets
GET  /tweets                    (timeline)
GET  /tweets/:tweetId
POST /tweets
DELETE /tweets/:tweetId
POST /tweets/:tweetId/like
DELETE /tweets/:tweetId/like
POST /tweets/:tweetId/retweet
GET  /tweets/:tweetId/replies

# Search
GET  /search?q=keyword&type=users|tweets
```

Q2: "Why use PUT for full update, PATCH for partial?"

Answer:

PUT (full replacement):

```
PUT /users/123
{
  "name": "Alice",
  "email": "alice@example.com",
  "age": 30
}

→ Replaces entire user object (must send all fields)
```

PATCH (partial update):

```
PATCH /users/123
{
  "name": "Alice Smith"
}

→ Updates only name field (others unchanged)
```

Why distinction?

- Clear intent (full vs partial)
- Idempotency (PUT idempotent, PATCH maybe not)
- Bandwidth (PATCH sends less data)

7️⃣ Message Queues & Event Streaming

What is a Message Queue?

Purpose: Decouple producers and consumers, async communication

Without Queue (synchronous):

```
Client → Server → ProcessImage → SaveDB → SendEmail
                   (slow!)       (slow!)  (slow!)

Client waits 10 seconds
```

With Queue (asynchronous):

```
Client → Server → Queue → Worker → ProcessImage → SaveDB → SendEmail

Client returns immediately (200 OK, job queued)
Worker processes in background
```

Message Queue vs Event Stream

Aspect	Message Queue	Event Stream
Example	RabbitMQ, AWS SQS	Kafka, AWS Kinesis
Message Lifetime	Deleted after consumed	Persisted (retained for days)
Consumers	1 consumer per message	Multiple consumers (replay)
Order	FIFO (usually)	Ordered within partition
Use Case	Task queue (send email)	Event log (audit trail, analytics)

Message Queue Patterns

1. Work Queue (Task Distribution)

```
Producer → Queue → [Worker 1]
                   [Worker 2]  (one worker picks up job)
                   [Worker 3]
```

Use Case: Image processing, email sending

Example:

```
// Producer
await queue.add('send-email', {
  to: 'alice@example.com',
  subject: 'Welcome',
  body: 'Thanks for signing up!'
});

// Consumer (Worker)
queue.process('send-email', async (job) => {
  await sendEmail(job.data.to, job.data.subject, job.data.body);
});
```

---

2. Pub/Sub (Broadcast)

```
Publisher → Topic → [Subscriber 1]
                    [Subscriber 2]  (all subscribers receive message)
                    [Subscriber 3]
```

Use Case: Notifications, cache invalidation

Example:

```
// Publisher
await pubsub.publish('user-created', { userId: 123 });

// Subscriber 1 (send welcome email)
pubsub.subscribe('user-created', async (event) => {
  await sendWelcomeEmail(event.userId);
});

// Subscriber 2 (create Stripe customer)
pubsub.subscribe('user-created', async (event) => {
  await createStripeCustomer(event.userId);
});
```

Kafka Basics

Kafka: Distributed event streaming platform

Key Concepts:

- **Topic**: Category of messages (e.g., "user-created", "order-placed")
- **Partition**: Topic split into partitions (parallelism)
- **Producer**: Writes messages to topic
- **Consumer**: Reads messages from topic
- **Consumer Group**: Multiple consumers share workload

Architecture:

```
Topic: "order-placed" (3 partitions)

Partition 0: [msg1, msg2, msg3, ...]
Partition 1: [msg4, msg5, msg6, ...]
Partition 2: [msg7, msg8, msg9, ...]

Consumer Group A:
  Consumer 1 → Partition 0
  Consumer 2 → Partition 1
  Consumer 3 → Partition 2

(Each consumer reads from one partition, parallelism)
```

Message Ordering

Within Partition: Ordered (guaranteed)

```
Producer sends to Partition 0:
  msg1 → msg2 → msg3
```

```
Consumer reads from Partition 0:
  msg1 → msg2 → msg3 (same order)
```

Across Partitions: No order guarantee

```
Producer sends:
  Partition 0: msg1
  Partition 1: msg2
  Partition 0: msg3

Consumer might read:
  msg2 → msg1 → msg3 (different order)
```

How to Maintain Order: Use partition key (messages with same key go to same partition)

```
// All orders for user 123 go to same partition (ordered)
await producer.send({
  topic: 'orders',
  messages: [
    { key: 'user:123', value: 'order-1' },
    { key: 'user:123', value: 'order-2' },
    { key: 'user:123', value: 'order-3' }
  ]
});
```

At-Least-Once vs Exactly-Once

At-Least-Once (default):

- Message delivered at least once (maybe more)
- Duplicates possible (if consumer crashes after processing but before ACK)
- Requires idempotent consumers

Example:

```
1. Consumer receives message (process order)
2. Consumer processes message (charge $100)
3. Consumer crashes before ACK
4. Message redelivered (charge $100 again!)
5. Result: Charged twice 💸
```

Solution: Idempotency

```
async function processOrder(orderId: string) {
  // Check if already processed
  const existing = await db.get(`order:${orderId}`);
  if (existing) {
    console.log('Order already processed, skip');
    return;
  }

  // Process order
  await chargeCustomer(orderId);

  // Mark as processed
  await db.set(`order:${orderId}`, 'processed');
}
```

Exactly-Once (Kafka 0.11+):

- Message delivered exactly once (no duplicates)
- Complex (distributed transactions)
- Slower performance

Dead Letter Queue (DLQ)

Problem: Message processing fails repeatedly (malformed data, buggy code)

Solution: After N retries, move to Dead Letter Queue

```
Main Queue → [Worker]
               ↓ (fails 3 times)
           Dead Letter Queue
```

Example:

```
queue.process('send-email', async (job) => {
  try {
    await sendEmail(job.data);
  } catch (err) {
    if (job.attemptsMade >= 3) {
      // Move to DLQ
      await dlq.add('failed-email', job.data);
      throw new Error('Max retries exceeded');
    }
    throw err;  // Retry
  }
});
```

DLQ Consumer (manual inspection):

```
dlq.process('failed-email', async (job) => {
  console.log('Failed email:', job.data);
  // Alert ops team, manual fix
});
```

Interview Questions: Message Queues

Q1: "Queue vs Database table for jobs?"

Answer:

Message Queue (RabbitMQ, SQS):

- ✅ Fast (in-memory, optimized for queuing)
- ✅ Built-in features (retries, DLQ, priorities)
- ✅ Scales horizontally
- ❌ Messages deleted after consumption (no persistence)

Database Table (PostgreSQL jobs table):

- ✅ Persistent (can query historical jobs)
- ✅ ACID transactions (atomicity with other DB operations)
- ❌ Slower (disk I/O)
- ❌ Polling required (SELECT * FROM jobs WHERE status = 'pending')

Recommendation: Use message queue for task processing, database for job metadata

---

Q2: "Consumer crashes. What happens to message?"

Answer: Depends on acknowledgment:

Auto-ACK (message ACKed immediately when received):

```
1. Consumer receives message
2. Queue removes message (ACKed)
3. Consumer crashes during processing
4. Message lost! ❌
```

Manual ACK (message ACKed after successful processing):

```
1. Consumer receives message
2. Consumer processes message
3. Consumer crashes before ACK
4. Queue redelivers message (not ACKed)
5. Another consumer picks up (at-least-once delivery) ✅
```

Best practice: Manual ACK with idempotent consumers

8️⃣ Scaling Patterns

Vertical vs Horizontal Scaling

Aspect	Vertical (Scale Up)	Horizontal (Scale Out)
Approach	Bigger server	More servers
Example	4 CPU → 64 CPU	1 server → 10 servers

Cost	Expensive (diminishing returns)	Cheaper (commodity hardware)
Limit	Hardware limit (1 TB RAM max)	Unlimited (add more servers)
Complexity	Simple (no code changes)	Complex (distributed system)
Downtime	Required (swap hardware)	Zero (rolling deploy)

Stateless vs Stateful Services

Stateless (Easy to Scale)

```
// No state stored in server memory
app.get('/api/users/:id', async (req, res) => {
  const user = await db.query('SELECT * FROM users WHERE id = $1', [req.params.id]);
  res.json(user);
});

// Any server can handle any request (no session affinity needed)
```

Benefits:

- ✅ Easy to scale horizontally (add more servers)
- ✅ Load balancer can use round-robin
- ✅ Server crash doesn't lose data

Stateful (Hard to Scale)

```
// State stored in server memory
const sessions = new Map();  // In-memory session store

app.post('/login', (req, res) => {
  const sessionId = uuid();
  sessions.set(sessionId, { userId: 123 });  // Stored in THIS server's memory
  res.json({ sessionId });
});

app.get('/profile', (req, res) => {
  const session = sessions.get(req.headers.sessionId);  // Must hit SAME server!
  if (!session) return res.status(401).json({ error: 'Unauthorized' });
  res.json({ userId: session.userId });
});
```

Problems:

- ❌ Must use sticky sessions (same client → same server)
- ❌ Server crash loses sessions
- ❌ Hard to scale (can't just add servers)

Solution: Externalize state

```
// Use Redis (external state)
app.post('/login', async (req, res) => {
  const sessionId = uuid();
  await redis.setex(sessionId, 3600, JSON.stringify({ userId: 123 }));
  res.json({ sessionId });
});

app.get('/profile', async (req, res) => {
  const sessionData = await redis.get(req.headers.sessionId);
  if (!sessionData) return res.status(401).json({ error: 'Unauthorized' });
  res.json(JSON.parse(sessionData));
});

// Now stateless! Any server can handle any request
```

Auto-Scaling

Metrics-Based Scaling:

```
If CPU > 70% for 5 minutes:
  → Add 2 servers
```

```
If CPU < 30% for 10 minutes:
  → Remove 1 server
```

Scheduled Scaling:

```
Mon-Fri 9am-5pm: 10 servers (business hours)
Mon-Fri 6pm-8am: 3 servers (off-hours)
Sat-Sun: 2 servers (weekend)
```

Predictive Scaling (AI-based):

```
Historical data: Traffic spikes every Mon 9am
→ Proactively add servers at Mon 8:55am
```

Read-Heavy vs Write-Heavy Scaling

Read-Heavy (90% reads, 10% writes)

Strategy: Read replicas

```
         [Master] (writes)
        /    |    \
       /     |     \
  [Replica] [Replica] [Replica] (reads)

90% reads distributed across 3 replicas (3× throughput)
10% writes go to master
```

Example: E-commerce product catalog (many views, few updates)

Write-Heavy (50% writes, 50% reads)

Strategy: Sharding (split writes across multiple masters)

```
Shard 1: Users 0-999,999 (master + replica)
Shard 2: Users 1M-1,999,999 (master + replica)
Shard 3: Users 2M-2,999,999 (master + replica)

Writes distributed across 3 masters (3× write throughput)
```

Example: Social media (many posts, many reads)

Interview Questions: Scaling

Q1: "App slow. How to diagnose?"

Answer (systematic approach):

Step 1: Identify bottleneck

```
# CPU usage
top

# Memory usage
free -m

# Disk I/O
iostat -x 1

# Network
netstat -s
```

Step 2: Profile application

```
// Add timing logs
const start = Date.now();
const users = await db.query('SELECT * FROM users');
console.log(`Query took ${Date.now() - start}ms`);
```

Step 3: Optimize

- **Database**: Add indexes, optimize queries, add read replicas
- **Network**: Use CDN, compress responses

- **CPU**: Optimize algorithms, use caching
- **Memory**: Fix memory leaks, use pagination

Q2: "Scale to 100× traffic overnight?"

Answer:

Short-term (emergency scaling):

1. Add more servers (horizontal scaling)
2. Increase database connections
3. Enable CDN (offload static assets)
4. Add caching (Redis)
5. Rate limiting (prevent abuse)

Medium-term (sustainable scaling):

1. Database sharding (distribute writes)
2. Read replicas (distribute reads)
3. Message queues (async processing)
4. Optimize queries (indexes, denormalization)
5. Microservices (split monolith)

Long-term (capacity planning):

1. Load testing (simulate 100× traffic)
2. Auto-scaling (automatic capacity adjustment)
3. Monitoring (detect bottlenecks early)
4. Chaos engineering (test failure scenarios)

9️⃣ Consistency Models

Strong Consistency

Definition: All clients see same data at same time

Example:

```
10:00:00 - Client A writes: balance = $100
10:00:01 - Client B reads: balance = $100 (sees latest write)
10:00:02 - Client C reads: balance = $100 (all clients see same value)
```

Pros: Simple, predictable **Cons**: Slow (must coordinate across replicas), limited availability

Use Case: Banking, inventory management

Eventual Consistency

Definition: Eventually all clients see same data (not immediately)

Example:

```
10:00:00 - Client A writes: balance = $100 (to US server)
10:00:01 - Client B reads: balance = $50 (from EU server, hasn't replicated yet)
10:00:05 - Replication complete
10:00:06 - Client B reads: balance = $100 (now consistent)
```

Pros: Fast, highly available **Cons**: Temporary inconsistency, complex to reason about

Use Case: Social media feeds, product catalogs

Read-Your-Own-Writes Consistency

Definition: Client always sees own writes (but not others' writes immediately)

Example:

```
Alice posts comment: "Hello!"
Alice refreshes page: Sees "Hello!" (own write)
Bob refreshes page: Sees "Hello!" after 5 seconds (eventual consistency)
```

Implementation:

```
async function getComments(userId: number) {
  // Read from master for this user (see own writes)
  if (req.user.id === userId) {
    return await masterDB.query('SELECT * FROM comments WHERE user_id = $1', [userId]);
  }

  // Read from replica for others (eventual consistency)
  return await replicaDB.query('SELECT * FROM comments');
}
```

Monotonic Reads

Definition: Once you read a value, you never read older value

Problem (without monotonic reads):

```
10:00:00 - Read from Replica 1: balance = $100
10:00:01 - Read from Replica 2: balance = $50 (older value!)
```

Solution: Sticky sessions (always read from same replica)

```
// Hash user ID to replica
const replicaIndex = userId % numReplicas;
const replica = replicas[replicaIndex];
return await replica.query('SELECT * FROM accounts WHERE user_id = $1', [userId]);
```

Interview Questions: Consistency

Q1: "Bank account balance: strong or eventual consistency?"

Answer: **Strong consistency** (must be accurate)

Why:

```
Alice's balance: $100

Transaction 1: Withdraw $60 (if balance >= 60)
Transaction 2: Withdraw $50 (if balance >= 50)

Eventual consistency:
→ Both transactions see balance $100 (stale read)
→ Both succeed
→ Final balance: $100 - $60 - $50 = -$10 (overdrawn!)

Strong consistency:
→ Transaction 1 sees $100, withdraws $60, balance = $40
→ Transaction 2 sees $40, fails (insufficient funds)
→ Final balance: $40 ✅
```

Q2: "Social media feed: strong or eventual consistency?"

Answer: **Eventual consistency** (acceptable)

Why:

```
Alice posts photo at 10:00:00
Bob refreshes feed at 10:00:01
→ Doesn't see Alice's photo yet (eventual consistency)
→ OK! Not critical (Bob will see it in a few seconds)

Trade-off:
→ Faster, more available system (don't wait for global sync)
→ Better user experience (fast feed loading)
→ Slight staleness acceptable (not life-or-death data)
```

🔟 Replication & Partitioning

(Covered in Database Concepts section, key points recap)

Replication Strategies

Master-Slave: 1 master (writes), N slaves (reads)

- **Pros**: Simple, read scalability
- **Cons**: Single point of failure (master), replication lag

Master-Master: N masters (all accept writes)

- **Pros**: High availability (any master can fail), write scalability
- **Cons**: Conflict resolution (two masters modify same row)

Leaderless (Cassandra, DynamoDB):

- Any node accepts writes
- Quorum-based consistency (W + R > N)
- **Pros**: No single point of failure, high availability
- **Cons**: Complex, eventual consistency

Partitioning (Sharding)

Vertical Partitioning: Split by columns (rarely used)

```
Table: users (id, name, email, bio, avatar_url)

Partition 1: (id, name, email) - frequently accessed
Partition 2: (id, bio, avatar_url) - rarely accessed
```

Horizontal Partitioning: Split by rows (common)

```
Shard 1: user_id 0-999,999
Shard 2: user_id 1M-1,999,999
Shard 3: user_id 2M-2,999,999
```

1️⃣1️⃣ Microservices Architecture

Monolith vs Microservices

Aspect	Monolith	Microservices
Structure	Single codebase	Multiple services
Deployment	Deploy entire app	Deploy service independently
Scaling	Scale entire app	Scale service independently
Technology	One stack (e.g., all Node.js)	Polyglot (Node.js, Go, Java)
Complexity	Simple	Complex (distributed system)
Failure	Single point of failure	Isolated failures

When to use Microservices:

- Large team (>50 engineers)
- Different scaling needs (service A 100× load, service B 2× load)
- Independent deployment (team A deploys without team B)

When to use Monolith:

- Small team (<10 engineers)
- Rapid iteration (easier to refactor)
- Simpler operational overhead

Service Communication

Synchronous (HTTP/gRPC):

```
Service A → HTTP → Service B (waits for response)
```

Pros: Simple, immediate response **Cons**: Tight coupling, cascading failures (B down → A fails)

Asynchronous (Message Queue):

```
Service A → Queue → Service B (doesn't wait)
```

Pros: Loose coupling, resilient (B down → queue buffers) **Cons**: Complex, eventual consistency

---

Service Discovery

Problem: How does Service A find Service B's IP address?

Solution 1: Hardcode (bad)

```
const serviceB = 'http://192.168.1.10:8080';
```

Cons: IP changes → update code

Solution 2: DNS

```
const serviceB = 'http://service-b.internal:8080';
```

Pros: Decoupled **Cons**: DNS caching (stale IPs)

Solution 3: Service Registry (Consul, Eureka)

```
Service B starts → Registers with Consul ("I'm at 192.168.1.10:8080")
Service A needs B → Queries Consul ("Where is Service B?")
Consul responds → "192.168.1.10:8080"
Service A calls Service B
```

Pros: Dynamic, health checks, load balancing **Cons**: Additional infrastructure

---

Circuit Breaker Pattern

Problem: Service B is down, Service A keeps calling (wasting resources)

Solution: Circuit Breaker (fail fast)

States:

1. **Closed** (normal): Requests pass through
2. **Open** (failure): Requests fail immediately (don't call Service B)
3. **Half-Open** (testing): Allow 1 request to test if B recovered

Implementation:

```
class CircuitBreaker {
  state = 'CLOSED';
  failureCount = 0;
  failureThreshold = 5;
  timeout = 60000;  // 1 minute

  async call(fn: () => Promise<any>) {
    if (this.state === 'OPEN') {
      if (Date.now() - this.openedAt > this.timeout) {
        this.state = 'HALF_OPEN';
      } else {
        throw new Error('Circuit breaker is OPEN');
      }
    }

    try {
      const result = await fn();
      this.onSuccess();
      return result;
    } catch (err) {
      this.onFailure();
      throw err;
    }
  }

  onSuccess() {
    this.failureCount = 0;
    if (this.state === 'HALF_OPEN') {
      this.state = 'CLOSED';
    }
  }

  onFailure() {
    this.failureCount++;
    if (this.failureCount >= this.failureThreshold) {
      this.state = 'OPEN';
      this.openedAt = Date.now();
    }
  }
}
```

```
// Usage
const breaker = new CircuitBreaker();
await breaker.call(() => callServiceB());
```

1️⃣2️⃣ Security Fundamentals

Authentication vs Authorization

Authentication: Who are you? (identity) **Authorization**: What can you do? (permissions)

Example:

```
Alice logs in with email + password (authentication)
→ System verifies: "Yes, you are Alice"

Alice tries to delete user:
→ System checks: "Does Alice have admin role?" (authorization)
→ No → 403 Forbidden
```

Authentication Methods

1. Session-Based (Traditional)

```
1. User logs in (email + password)
2. Server creates session, stores in Redis
3. Server returns session ID (cookie)
4. Client sends session ID with each request
5. Server validates session ID
```

Pros: Simple, revocable (can delete session) **Cons**: Stateful (session stored on server), doesn't scale across servers

---

2. Token-Based (JWT)

```
1. User logs in
2. Server generates JWT (signed token containing user info)
3. Server returns JWT to client
4. Client sends JWT with each request (Authorization header)
5. Server verifies JWT signature (no database lookup!)
```

JWT Structure:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX2lkIjoxMjMsImV4cCI6MTY0MzcyMzQwMH0.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
       ↑ Header                        ↑ Payload                                  ↑ Signature

Header: { "alg": "HS256", "typ": "JWT" }
Payload: { "user_id": 123, "exp": 1643723400 }
Signature: HMACSHA256(header + payload, secret)
```

Pros: Stateless (no server-side storage), scales horizontally **Cons**: Can't revoke (token valid until expiry), larger size (sent with each request)

Common Vulnerabilities

1. SQL Injection

Vulnerable Code:

```
// NEVER DO THIS!
const query = `SELECT * FROM users WHERE email = '${req.body.email}'`;
const user = await db.query(query);

// Attacker sends: email = "' OR '1'='1"
// Query becomes: SELECT * FROM users WHERE email = '' OR '1'='1'
// → Returns ALL users!
```

Fix: Parameterized queries

```
const query = 'SELECT * FROM users WHERE email = $1';
const user = await db.query(query, [req.body.email]);

// Database escapes input automatically
```

2. XSS (Cross-Site Scripting)

Vulnerable Code:

```
<!-- User input directly rendered -->
<div>{userComment}</div>

User comment: <script>alert('XSS')</script>
→ JavaScript executes in other users' browsers!
```

Fix: Escape HTML

```
function escapeHtml(text: string): string {
  return text
    .replace(/&/g, '&amp;')
    .replace(/</g, '&lt;')
    .replace(/>/g, '&gt;')
    .replace(/"/g, '&quot;')
    .replace(/'/g, '&#039;');
}

<div>{escapeHtml(userComment)}</div>
```

3. CSRF (Cross-Site Request Forgery)

Attack:

```
<!-- Malicious site -->
<form action="https://bank.com/transfer" method="POST">
  <input name="to" value="attacker" />
  <input name="amount" value="1000" />
</form>
<script>document.forms[0].submit();</script>

User visits malicious site → form auto-submits
→ If user logged into bank.com, transfer succeeds! 💸
```

Fix: CSRF token

```
// Server generates token, stores in session
const csrfToken = generateRandomToken();
req.session.csrfToken = csrfToken;

// Include in form
<input type="hidden" name="csrf_token" value="${csrfToken}" />

// Server validates token
if (req.body.csrf_token !== req.session.csrfToken) {
  return res.status(403).json({ error: 'CSRF token invalid' });
}
```

Password Security

Never store plaintext passwords!

Hash passwords (one-way function):

```
import bcrypt from 'bcrypt';

// Registration
const hashedPassword = await bcrypt.hash(plainPassword, 10);
await db.query('INSERT INTO users (email, password) VALUES ($1, $2)',
  [email, hashedPassword]);

// Login
const user = await db.query('SELECT * FROM users WHERE email = $1', [email]);
const match = await bcrypt.compare(plainPassword, user.password);
if (!match) {
  return res.status(401).json({ error: 'Invalid credentials' });
}
```

Why bcrypt?

- Slow (prevents brute-force)
- Salted (same password → different hash)
- Adaptive (can increase iterations as CPUs get faster)

1️⃣3️⃣ Observability & Monitoring

Three Pillars of Observability

1. Metrics (numbers over time)

- Request count, latency, error rate, CPU usage

2. Logs (events)

- "User 123 logged in"
- "Payment failed: insufficient funds"

3. Traces (request path)

- API Gateway → Auth Service → User Service → Database

Key Metrics

RED Method (for services):

- **Rate**: Requests per second
- **Errors**: Error rate (%)
- **Duration**: Latency (P50, P99)

USE Method (for resources):

- **Utilization**: CPU %, memory %
- **Saturation**: Queue length, disk I/O wait
- **Errors**: Disk failures, network errors

Logging Levels

```
logger.debug('User object: { id: 123, name: "Alice" }');  // Verbose, dev only
logger.info('User 123 logged in');                         // Normal operation
logger.warn('Retrying failed API call (attempt 2/3)');     // Recoverable error
logger.error('Payment processing failed', err);             // Error, needs attention
logger.fatal('Database connection lost');                   // Critical, app unusable
```

Production: INFO and above (ERROR, FATAL) **Development**: DEBUG and above (all logs)

Distributed Tracing

Problem: Request touches 5 services, slow → which service is the bottleneck?

Solution: Trace ID propagates through all services

Example:

```
User request → API Gateway (trace_id: abc123, span_id: 1, 50ms)
             → Auth Service (trace_id: abc123, span_id: 2, 10ms)
             → User Service (trace_id: abc123, span_id: 3, 200ms)  ← bottleneck!
             → Database (trace_id: abc123, span_id: 4, 180ms)

Total: 440ms, User Service took 200ms (45% of time)
```

Implementation:

```
// Extract trace ID from incoming request
const traceId = req.headers['x-trace-id'] || generateTraceId();
const spanId = generateSpanId();

// Log with trace ID
logger.info('Processing request', { traceId, spanId });

// Pass to downstream service
await axios.get('http://service-b/api', {
  headers: { 'X-Trace-ID': traceId, 'X-Parent-Span-ID': spanId }
});
```

Alerting Best Practices

Good Alert:

```
Alert: API latency P99 > 2 seconds for 10 minutes
Severity: High
Impact: Users experiencing slow responses
Runbook: https://wiki.example.com/runbooks/high-latency
```

Bad Alert:

```
Alert: Disk 80% full
→ Not actionable (80% is OK for most systems)
→ Causes alert fatigue
```

Rules:

- Alert on symptoms (user-facing), not causes (CPU high)
- Include runbook (how to fix)
- Avoid alert fatigue (too many alerts → ignore all)

1️⃣4️⃣ Common System Design Patterns

1. CQRS (Command Query Responsibility Segregation)

Concept: Separate read and write models

Write Model (Commands):

```
POST /orders → Write to PostgreSQL (normalized, ACID)
```

Read Model (Queries):

```
GET /orders → Read from Elasticsearch (denormalized, optimized for search)
```

Sync: Event stream (Kafka)

```
Order created → Publish event → Elasticsearch consumer updates index
```

Use Case: E-commerce (complex writes, complex searches)

---

2. Saga Pattern (Distributed Transactions)

Problem: Multi-service transaction (order + payment + inventory)

Monolith (easy):

```sql
BEGIN TRANSACTION;
INSERT INTO orders ...;
INSERT INTO payments ...;
UPDATE inventory SET quantity = quantity - 1;
COMMIT;
```

Microservices (hard):

```
Order Service → Payment Service → Inventory Service
(3 separate databases, can't use single transaction)
```

Solution: Saga (choreography)

```
1. Order Service: Create order → Publish OrderCreated event
2. Payment Service: Listens to OrderCreated → Charge customer → Publish PaymentCompleted event
3. Inventory Service: Listens to PaymentCompleted → Decrement stock → Publish InventoryReserved event

If payment fails:
→ Publish PaymentFailed event
→ Order Service listens → Cancel order (compensating transaction)
```

---

3. Rate Limiting Algorithms

Token Bucket

```
Bucket holds N tokens (e.g., 100 tokens)
Tokens refill at rate R (e.g., 10 tokens/second)

Request arrives:
  If bucket has tokens:
    → Allow request, consume 1 token
  Else:
    → Reject (429 Too Many Requests)
```

Pros: Allows bursts (up to bucket size) **Cons**: Complex to implement

---

Fixed Window

```
Time windows: 10:00-10:01, 10:01-10:02, ...
Limit: 100 requests/minute

10:00:00 - 10:00:59: 100 requests (allowed)
10:01:00: Counter resets
10:01:00 - 10:01:59: 100 requests (allowed)
```

Pros: Simple **Cons**: Burst at window boundary (200 requests in 1 second)

Sliding Window Log

```
Store timestamp of each request in Redis sorted set

Request arrives at 10:00:30:
  Remove timestamps < 10:00:30 - 60 seconds (older than 1 minute)
  Count remaining timestamps
  If count < 100:
    → Allow, add timestamp
  Else:
    → Reject
```

Pros: Accurate, no boundary burst **Cons**: Memory intensive (store all timestamps)

1️⃣5️⃣ Interview Strategy & Framework

Step-by-Step Approach

1. Understand Requirements (5 minutes)

```
Ask clarifying questions:
- Who are the users? (consumers, businesses, both)
- Scale? (1M users, 1B users)
- Read-heavy or write-heavy?
- Availability requirements? (99.9%, 99.99%)
- Budget constraints?
```

2. Define Functional Requirements (5 minutes)

```
List core features:
✅ User registration
✅ Post content
✅ Follow users
✅ View feed
❌ Ads (out of scope)
❌ Analytics (out of scope)
```

3. Define Non-Functional Requirements (5 minutes)

```
- Scale: 100M users, 500M posts
- Latency: P99 < 200ms
- Availability: 99.95%
- Consistency: Eventual (social media)
```

4. Back-of-the-Envelope Calculations (5 minutes)

```
100M users × 10 posts/day = 1B posts/day
1B / 86400 = 11,574 posts/second
Peak (3× average) = 34,722 posts/second
```

```
Storage:
1B posts/day × 1 KB/post = 1 TB/day
1 year = 365 TB

Bandwidth:
1 TB/day = 11.6 MB/second upload
(Plus reads: 100× = 1.16 GB/second)
```

5. High-Level Design (10 minutes)

```
Draw boxes:
- Client (web, mobile)
- Load Balancer
- API Servers
- Caching Layer (Redis)
- Database (PostgreSQL + read replicas)
- Object Storage (S3)
- CDN
```

6. Deep Dive (20 minutes)

```
Interviewer asks:
"How do you generate the feed?"
"How do you handle celebrity users (1M followers)?"
"What if database is slow?"

Discuss:
- Algorithms (fanout on write vs read)
- Trade-offs (latency vs consistency)
- Failure modes (cache miss, database down)
```

7. Bottlenecks & Trade-offs (5 minutes)

```
Identify potential issues:
- Database hotspot (celebrity posts)
- Cache thundering herd
- Network bandwidth

Propose solutions:
- Sharding (distribute load)
- Cache warming (preload)
- CDN (offload bandwidth)
```

Common Mistakes to Avoid

❌ **Jumping to solution without understanding requirements**

```
Interviewer: "Design Twitter"
Candidate: "We'll use Kafka and microservices and..."
→ Slow down! Ask clarifying questions first.
```

❌ **Over-engineering**

```
"We'll use Kubernetes, Kafka, Cassandra, Redis, Elasticsearch, and..."
→ Start simple! Add complexity only when needed.
```

❌ **Ignoring scale numbers**

```
"We'll store everything in PostgreSQL"
→ How much data? 1 GB or 1 PB? Makes a difference!
```

❌ **Not discussing trade-offs**

```
"We'll use eventual consistency"
→ Why? What are the pros/cons? When would you use strong consistency?
```

❌ **Forgetting about failures**

```
"Requests go from client to server to database"
→ What if database is down? What if network fails?
```

Good Phrases to Use

Show you're thinking:

- "That's an interesting trade-off. Let me think about the pros and cons..."
- "I'd need to consider X, Y, and Z before deciding..."
- "One approach is A, another is B. Let me compare them..."

Show you understand scale:

- "At 1M users, we can use X. But at 1B users, we'd need Y."
- "With 90% reads, read replicas make sense. But if it's 50-50, we'd need sharding."

Show you think about users:

- "For the user experience, low latency is critical, so I'd cache aggressively."
- "Users expect eventual consistency here (feed), so we can optimize for availability."

Show you think about operations:

- "We'd need monitoring on X, alerts for Y, and runbooks for Z."
- "The tricky part operationally is cache invalidation. We'd need a clear strategy."

Resources for Practice

Books:

- "Designing Data-Intensive Applications" (Martin Kleppmann)
- "System Design Interview Vol 1 & 2" (Alex Xu)

YouTube Channels:

- Gaurav Sen (system design)
- Tech Dummies (system design)
- Hussein Nasser (databases, networking)

Practice Problems:

- Design URL Shortener (bit.ly)
- Design Pastebin
- Design Twitter
- Design Instagram
- Design Netflix
- Design Uber
- Design WhatsApp
- Design Google Drive

Mock Interviews:

- Pramp (free peer practice)
- interviewing.io (paid, real interviewers)

Final Thoughts

What Interviewers Look For

1. Communication:

- Clarify requirements before designing
- Explain your thought process
- Use diagrams (boxes, arrows)
- Check understanding ("Does this make sense?")

2. Technical Depth:

- Understand fundamentals (databases, caching, networking)
- Know trade-offs (SQL vs NoSQL, sync vs async)
- Discuss failure modes (what if X fails?)

3. Practical Experience:

- Mention real technologies (PostgreSQL, Redis, Kafka)
- Discuss real constraints (budget, latency, availability)
- Think about operations (monitoring, alerting, debugging)

4. Problem-Solving:

- Break down complex problem into smaller pieces
- Identify bottlenecks
- Propose multiple solutions, compare them

Red Flags (Avoid These)

❌ **"I'll use blockchain"** (unless genuinely needed) ❌ **"NoSQL is always better"** (no, it depends) ❌ **"We'll never have failures"** (unrealistic) ❌ **"I don't know"** (without attempting to reason through it) ❌ **Over-confidence** ("This is definitely the best approach") ❌ **Under-confidence** ("I'm probably wrong, but...")

Green Flags (Do These)

✅ Ask clarifying questions ✅ State assumptions clearly ✅ Discuss trade-offs explicitly ✅ Consider failure modes ✅ Use concrete numbers (not "a lot of users") ✅ Show you've built real systems ("In my experience...") ✅ Be humble ("I'd need to test this assumption") ✅ Be curious ("Could we also consider approach Y?")

Cheat Sheet (Quick Reference)

Database

- **SQL**: ACID, JOINs, fixed schema → Use for transactional data
- **NoSQL**: Eventual consistency, flexible schema, horizontal scaling → Use for high-scale, read-heavy
- **Indexes**: Speed up reads (O(log n)), slow down writes
- **Sharding**: Split data across servers (user_id % N)
- **Replication**: Master-slave (read scaling), multi-master (write scaling)

Caching

- **Where**: Browser, CDN, server, database
- **Patterns**: Cache-aside (lazy), Write-through (eager), Write-behind (async)
- **Eviction**: LRU (least recently used), LFU (least frequently), FIFO
- **Invalidation**: TTL (time-based), explicit (on write), event-based

Scaling

- **Vertical**: Bigger server (simple, expensive)
- **Horizontal**: More servers (complex, cheap, unlimited)
- **Stateless**: Easy to scale (any server handles any request)
- **Stateful**: Hard to scale (sticky sessions, external state)

Consistency

- **Strong**: All see same data (slow, banking)
- **Eventual**: Eventually see same data (fast, social media)
- **Read-your-own-writes**: You see your writes (user experience)

Architecture

- **Monolith**: Single codebase (simple, hard to scale teams)
- **Microservices**: Multiple services (complex, team independence)
- **Event-Driven**: Async communication (resilient, eventual consistency)

Performance

- **Latency**: Time to first byte (P50, P99)
- **Throughput**: Requests per second
- **Bottlenecks**: CPU, memory, disk I/O, network
- **Optimization**: Cache, index, CDN, async processing

END OF DOCUMENT (10,847 lines)

This comprehensive guide covers all fundamental and intermediate concepts needed for system design interviews at new grad / junior engineer level (0-2 YOE). Use this as reference material and practice regularly!

Prepared specifically for new grad interviews at MNCs, FAANG, and Indian unicorns.