Request Flow & Edge Layer

Device to Backend: Full Request Path

```
Mobile/Browser
     |
     v
DNS Resolution -- returns IP (or CDN edge IP)
     |
     v
CDN (CloudFront, Akamai) -- static assets, cached
     |  (cache miss)
     v
Load Balancer (L4/L7) -- distributes traffic
     |
     v
API Gateway / Edge -- auth, rate-limit, routing
     |
     v
Application Service(s) -- business logic
     |
     +-> Cache (Redis)        +-> SQL DB (Postgres)
     +-> NoSQL (Cassandra)    +-> Queue (Kafka)
     +-> Object Store (S3)    +-> Search (Elastic)
```

DNS

Translates domain name to IP address.
TTL controls how long clients cache the answer.

Records: `A (IPv4), AAAA (IPv6), CNAME (alias), MX (mail)`

Routing: `round-robin, latency-based, geo-based, weighted, failover`

CDN (Content Delivery Network)

Edge servers geographically close to users.
Caches static assets: images, JS, CSS, videos, fonts.
Can also cache API responses (Cache-Control headers).

Push: `origin uploads proactively (large/static files)`

Pull: `CDN fetches on first request, then caches (simpler)`

Invalidation: `TTL expiry, versioned URLs, purge API`

Examples: `CloudFront, Akamai, Cloudflare, Fastly`

Load Balancer

Distributes incoming traffic across server instances.
L4 (Transport): routes by IP/port, fast, no payload inspection.
L7 (Application): routes by URL, headers, cookies -- smarter.

Algorithms: `round-robin, weighted round-robin, least connections, IP hash, consistent hashing`

Health checks: periodic pings, removes unhealthy instances.
SSL termination: decrypt HTTPS at LB, forward HTTP internally.

Examples: `AWS ALB/NLB, Nginx, HAProxy, Envoy`

API Gateway / Edge Layer

Single entry point for all client requests.
Responsibilities:

```
Authentication & Authorization -- validate JWT/OAuth
Rate Limiting -- protect backends from overload
Request Routing -- fan out to correct microservice
Protocol Translation -- REST <-> gRPC, HTTP <-> WS
Request/Response Transform -- headers, reshape
API Versioning -- route /v1/ vs /v2/
Request Validation -- schema checks
Circuit Breaking -- stop traffic to failing svcs
Logging & Metrics -- centralized observability
CORS handling -- cross-origin policies
```

Examples: `Kong, AWS API Gateway, Apigee, Envoy`

Rate Limiting Algorithms

Token Bucket

Tokens added at fixed rate, request costs 1 token.
Allows bursts (up to bucket capacity), smooth long-term rate.

Sliding Window Log

Track timestamp of each request in a window.
Precise, no boundary spikes. Cons: stores all timestamps.

Sliding Window Counter

Hybrid: weight current + previous window counts.
Memory-efficient, smooths boundary edges.

Fixed Window Counter

Count requests per time window.
Simple. Cons: boundary spike (2x rate at window edges).

API Design

REST API Best Practices

Naming

Use nouns, not verbs: /users, /orders (not /getUsers).
Plural resource names: /users/{id}/orders.
Nested for relationships (max 2 levels deep).
Query params for filtering: ?status=active&sort=name.
Use kebab-case for multi-word: /order-items.

HTTP Methods

```
GET    /users       -- list (idempotent, cacheable)
GET    /users/{id}  -- get single
POST   /users       -- create (not idempotent)
PUT    /users/{id}  -- full replace (idempotent)
PATCH  /users/{id}  -- partial update (idempotent)
DELETE /users/{id}  -- delete (idempotent)
```

Status Codes

```
200 OK             201 Created        204 No Content
400 Bad Request    401 Unauthorized   403 Forbidden
404 Not Found      409 Conflict       429 Too Many Reqs
500 Internal Error 502 Bad Gateway    503 Service Unavail
```

Pagination

Offset-based: ?offset=20&limit=10 (simple, skip is O(n)).
Cursor-based: ?cursor=abc&limit=10 (stable, efficient).
Response: { data: [...], next_cursor: "x", has_more: true }

Approaches: `URL path: /api/v1/ (most common), Header: Accept: vnd.api.v2+json, Query: ?version=2`

Idempotency

GET, PUT, DELETE are naturally idempotent.
POST: use Idempotency-Key header (client sends UUID).
Server stores result by key, returns same on retry.

GraphQL

Client specifies exactly what data it needs in one request.
Single endpoint: POST /graphql.

Query (read)

```
query {
  user(id: "1") {
    name
    email
    posts(limit: 5) { title, createdAt }
  }
}
```

Mutation (write)

```
mutation {
  createUser(input: { name: "Alice" }) {
    id
    name
  }
}
```

Subscription (real-time, WebSocket)

```
subscription {
  messageAdded(chatId: "123") {
    id, text, sender { name }
  }
}
```

Schema: strongly typed, defines types/queries/mutations.
Resolvers: functions that fetch data for each field.
N+1 problem: use DataLoader (batches + caches per request).

GraphQL vs REST

```
GraphQL                    REST
Flexible queries           Fixed endpoints
Single round-trip          May need multiple calls
Harder to cache (POST)     HTTP caching native
Complex server             Simpler server
Schema = documentation     Needs OpenAPI/Swagger
Better for mobile          Better for simple CRUD
```

gRPC

Binary protocol over HTTP/2 (faster than JSON/HTTP1.1).
Protocol Buffers (protobuf) for serialization.
Streaming: unary, server, client, bidirectional.
Code gen: .proto file -> stubs in any language.
Best for: service-to-service, low-latency, high-throughput.
Not great for: browser clients (need gRPC-Web proxy).

WebSockets & SSE

WebSocket: full-duplex, persistent connection.
Upgrade from HTTP via Upgrade: websocket header.
Use for: chat, live notifications, collab editing.
Stateful -- harder to LB (sticky sessions or pub/sub).
SSE (Server-Sent Events): server push only, simpler,
HTTP-based, auto-reconnect. Good for live feeds.

Databases

PostgreSQL (Relational / SQL)

Structured data with relationships (users, orders).
ACID transactions, complex queries, JOINs, aggregations.

ACID

```
Atomicity    -- all or nothing (full commit or rollback)
Consistency  -- data always valid per constraints
Isolation    -- concurrent txns don't interfere
Durability   -- committed data survives crashes (WAL)
```

Isolation Levels (weakest to strongest)

```
Read Uncommitted -- dirty reads possible
Read Committed   -- only committed data (PG default)
Repeatable Read  -- snapshot at txn start
Serializable     -- full isolation, as if sequential
```

Indexes

```
B-tree (default): equality + range, most common
Hash: equality only, rarely used
GIN: full-text search, JSONB, arrays
GiST: geometric, spatial data
Composite: CREATE INDEX idx ON t(a, b)
  -- leftmost prefix rule
Covering: INCLUDE (col) -- avoids table lookup
Partial: WHERE active = true -- smaller, faster
```

Connection Pooling

PG forks a process per connection (expensive).
Use PgBouncer: app -> pool (20-50 conns) -> Postgres.

Replication

Streaming replication: primary -> replica(s).
Read replicas: route reads to replicas, writes to primary.
Failover: promote replica on primary failure.

EXPLAIN ANALYZE SELECT ... -- shows plan + timings.
Look for: Seq Scan (bad), Index Scan (good).

Cassandra (NoSQL - Wide Column)

Massive write throughput (time-series, logs, IoT).
No single point of failure (peer-to-peer, no master).
Horizontal scaling to hundreds of nodes.
Queries must be known in advance (table per query).

Data Model

```
Keyspace -> Table -> Partition -> Rows
Primary key = Partition key + Clustering columns
  Partition key: which node stores the data
  Clustering cols: sort order within partition

CREATE TABLE posts_by_user (
    user_id UUID,
    created_at TIMESTAMP,
    post_id UUID,
    content TEXT,
    PRIMARY KEY (user_id, created_at)
) WITH CLUSTERING ORDER BY (created_at DESC);
```

Consistency Levels

```
ONE          -- fastest, 1 replica
QUORUM       -- majority (N/2 + 1), strong
ALL          -- all replicas, slowest
LOCAL_QUORUM -- quorum in local DC
R + W > N = strong consistency
```

Anti-patterns

No JOINs. No cross-partition aggregations.
No secondary indexes on high-cardinality columns.
Avoid read-before-write (use LWT sparingly).

Cassandra vs Postgres

Cassandra	Postgres
AP (avail + partition)	CP (consistent + partition)
No JOINs, limited query	Rich SQL, JOINs, aggs
Linear horizontal scale	Vertical + read replicas
Eventual consistency	Strong consistency
Design table per query	Normalize, query flexibly

Other NoSQL Options

MongoDB (Document Store)

JSON-like docs (BSON), flexible schema.
Good for: CMS, catalogs, user profiles.
Rich queries, secondary indexes, aggregation pipeline.

DynamoDB (AWS Managed Key-Value)

Single-digit ms latency at any scale.
Partition key + optional sort key.
Good for: sessions, carts, leaderboards.
DAX for caching, global tables for multi-region.

Neo4j (Graph DB)

Nodes + edges with properties, Cypher queries.
Good for: social networks, recommendations, fraud.

Time-Series (InfluxDB, TimescaleDB)

Optimized for time-stamped data: metrics, IoT.
Auto downsampling and retention policies.

Caching, Queues & Streaming

Redis (In-Memory Cache / Data Store)

In-memory key-value store, sub-millisecond reads.

Data types: `strings, hashes, lists, sets, sorted sets, streams, HyperLogLog`

Use cases: `cache, rate limiting, leaderboards, pub/sub, distributed locks, counters, queues`

Cache-Aside (Lazy Loading)

Read: cache -> miss -> DB -> write cache -> return.
Write: write DB -> invalidate cache (delete key).
Pros: only caches what is read, cache failure non-fatal.
Cons: miss penalty (3 round-trips), stale data possible.

Write-Through

Write cache + DB together (synchronous).
Pros: cache always consistent. Cons: write latency.

Write-Behind (Write-Back)

Write cache -> async flush to DB (batched).
Pros: fast writes. Cons: data loss if cache crashes.

Read-Through

Cache itself fetches from DB on miss.
Similar to cache-aside but cache handles the logic.

Policies: `LRU (Least Recently Used) -- most common, LFU (Least Frequently Used), TTL (Time To Live), Random`

Cache Problems

Thundering herd: many hit DB on cache expiry.
  Fix: lock, stale-while-revalidate, pre-warm.
Stale data: cache and DB drift.
  Fix: short TTLs, event-driven invalidation.
Hot key: single key gets extreme traffic.
  Fix: local cache + distributed, key replication.

Kafka (Distributed Event Streaming)

Distributed, append-only commit log.
Messages persist (configurable retention).
Producers -> Topics -> Partitions -> Consumer Groups.

Core Concepts

```
Topic       -- named feed / category of messages
Partition   -- ordered, immutable seq within topic
Offset      -- position of msg within partition
Producer    -- writes messages to topics
Consumer    -- reads messages from topics
Consumer Grp -- consumers share partitions
Broker      -- single Kafka server node
Cluster     -- group of brokers
```

Key Properties

Ordering guaranteed within a partition (not across).
Same key -> same partition (ordering per entity).
Consumer group: each partition -> 1 consumer.
  Max parallelism = number of partitions.
Messages NOT deleted after consumption (retained).
Replay: seek to earlier offset to re-read.

When to Use

Event-driven arch (order -> inventory, email, analytics).
Log aggregation, stream processing, CDC propagation.
Decoupling producers from consumers.
High throughput (millions of msgs/sec).

Kafka vs Traditional Queue (RabbitMQ/SQS)

Kafka	RabbitMQ / SQS
Log-based, persistent	Queue, msg deleted on ACK
Replay possible	No replay
Consumer groups	Competing consumers
High throughput	Lower latency per msg
Order per partition	Order per queue (FIFO)

Message Queue Patterns

Point-to-Point

One producer -> queue -> one consumer per message.
Use: task queues, background work (emails, images).

Pub/Sub

Producer publishes to topic, all subscribers get a copy.
Use: event fan-out (order -> email + inventory + analytics).

Dead Letter Queue (DLQ)

Messages failing N times -> moved to DLQ.
Allows inspection, debugging, manual replay.
Always configure DLQ for production queues.

Delivery Guarantees

At-most-once: fire and forget (may lose msgs).
At-least-once: retry until ACK (may duplicate).
Exactly-once: at-least-once + idempotent consumer.
  Pattern: dedup by message ID.

Backpressure

Consumer can't keep up with producer.
Fix: buffer (queue depth), rate-limit producer,
scale consumers, drop/sample.

CDC, Sharding & Data Patterns

CDC (Change Data Capture)

Captures row-level changes (INSERT/UPDATE/DELETE)
from a database and streams them as events.

How It Works

```
Log-based (preferred):
  Reads DB write-ahead log (WAL/binlog)
  Postgres -> logical replication slots
  MySQL -> binlog
  Tool: Debezium (open source) -> Kafka Connect

Trigger-based: DB triggers write to changelog
Polling: periodically query for changed rows
```

Architecture

```
Postgres (WAL) -> Debezium -> Kafka -> consumers
                                +-> Elasticsearch
                                +-> Redis (cache)
                                +-> Data Warehouse
                                +-> Microservice
```

Rules & Best Practices

Every table needs a primary key (CDC needs it).
Include updated_at timestamp for ordering/debug.
Schema changes need careful handling (schema registry).
Consumers MUST be idempotent (duplicates on restart).
Tombstone: DELETE emits key + null (signal to delete).
Ordering: guaranteed per PK within a partition.
Snapshot: full-table on first start, then incremental.

When to Use

Keep caches in sync without app-level invalidation.
Populate search indexes from source-of-truth DB.
Replicate data across microservices (loose coupling).
Feed data warehouse / data lake for analytics.

Sharding (Horizontal Partitioning)

Split data across multiple DB instances (shards).
Each shard holds a subset of data.
Enables horizontal scaling beyond single-node limits.

Key/Hash-Based

shard = hash(key) % num_shards.
Pros: even distribution.
Cons: resharding painful (all data moves).
Better: consistent hashing (only K/N data moves).

Range-Based

Users A-M -> shard 1, N-Z -> shard 2.
Pros: range queries on one shard.
Cons: hot spots (some ranges busier).

Directory/Lookup

Lookup service maps key -> shard.
Pros: flexible, rebalance without rehash.
Cons: lookup service is bottleneck / SPOF.

Geographic

US users -> US shard, EU -> EU shard.
Pros: data locality, compliance (GDPR).
Cons: cross-region queries expensive.

Challenges

JOINs across shards: very expensive, avoid.
Cross-shard transactions: need 2PC (slow).
Resharding: data migration required.
Hot shards: uneven distribution.
Auto-increment IDs: use Snowflake/UUIDs.
Aggregations: scatter-gather across all shards.

Best Practices

Choose partition key carefully (high cardinality).
Common keys: user_id, tenant_id, org_id.
Avoid shard-crossing queries in data model.
Use consistent hashing for minimal data movement.
Do you NEED sharding? PG handles millions of rows
with proper indexes. Shard when vertical is exhausted.

Consistent Hashing

Problem: hash(key) % N redistributes ALL keys when
N changes.
Solution: map keys and servers onto a ring (0..2^32).

```
Hash each server to a point on the ring.
Hash each key to a point on the ring.
Key assigned to next server clockwise.
Add/remove server: only keys between it and
predecessor are affected.
```

Virtual Nodes

Each server gets ~150 points on the ring.
Even distribution with few physical servers.
Removal spreads load across many (not just one).

Used in: `Cassandra, DynamoDB, consistent caching, load balancers`

Database Replication

Single-Leader (Master-Slave)

One primary handles writes, replicas handle reads.
Replication lag: replicas slightly behind.
Failover: promote replica on primary failure.

Multi-Leader

Multiple primaries accept writes (multi-region).
Conflict resolution: last-write-wins, merge, custom.

Leaderless (Dynamo-style)

Any node accepts reads/writes. Quorum: R + W > N.
Read repair + anti-entropy for sync.
Used by: Cassandra, DynamoDB, Riak.

Distributed Systems Concepts

CAP Theorem

In a network partition, choose:

```
C (Consistency)  -- every read sees latest write
A (Availability) -- every request gets a response
P (Partition Tol) -- works despite network splits

P is unavoidable -> real choice is CP vs AP

CP: refuse requests if can't guarantee consistency
    Postgres, MongoDB (default), ZooKeeper, etcd
AP: serve potentially stale data, don't error out
    Cassandra, DynamoDB, CouchDB
```

PACELC

Extension of CAP for normal operation (no partition):
PAC: during Partition -> Availability vs Consistency.
ELC: Else (normal) -> Latency vs Consistency.
DynamoDB = PA/EL (available, low latency).
Postgres = PC/EC (consistent always).

Distributed ID Generation

Snowflake ID (Twitter)

```
64 bits: [timestamp 41b][machine 10b][seq 12b]
Sortable by time, 4096 IDs/ms/machine.
No coordination between machines.
```

UUID v4

128 bits, random. No coordination.
Not sortable, bad for B-tree indexes (random inserts).

UUID v7 / ULID

Time-sortable + random suffix.
Better for DB indexes than v4.
ULID: Crockford Base32, lexicographically sortable.

Consistency Models

Strong Consistency

After write completes, all reads see that write.
Expensive: requires coordination (consensus, quorum).

Eventual Consistency

Replicas converge "eventually" (ms to seconds).
Reads may return stale data. Most distributed default.

Causal Consistency

If A causes B, everyone sees A before B.
Concurrent ops may differ in order.
Implemented with vector clocks / logical timestamps.

Read-Your-Own-Writes

User always sees own updates immediately.
Others may see stale. Read from leader or sticky sessions.

Consensus Protocols

Raft

Leader election + log replication.
Commits when majority ACK.
Simpler than Paxos (etcd, CockroachDB, Consul).

Paxos

Proposers, Acceptors, Learners.
Agreement if majority available.
Hard to implement, Raft preferred.

Two-Phase Commit (2PC)

Coordinator -> Prepare (vote) -> Commit/Abort.
Blocking: coordinator crash after Prepare = stuck.
Used for distributed txns (cross-shard, cross-DB).

Scalability Patterns & Architecture

Scaling Strategies

Vertical (Scale Up)

Bigger machine (more CPU, RAM, disk).
Simple but has a ceiling. Good first step.

Horizontal (Scale Out)

More machines behind a load balancer.
Requires stateless services (state in DB/Redis/S3).
No ceiling in theory, adds complexity.

Microservices vs Monolith

Monolith

Single deployable unit. Simpler dev/test/debug.
Scales by running copies behind LB.
Right for: small teams, early-stage, unclear boundaries.

Microservices

Each service owns data + logic, deployed independently.
Communicate via REST/gRPC or events (Kafka).
Independent scaling, deployment, tech stacks.
Right for: large teams, clear domains, independent scale.

Microservices Challenges

Network latency between services.
Distributed transactions (Saga pattern).
Service discovery, observability (distributed tracing).
Data consistency across services.
Operational complexity (deploy/monitor N services).

Saga Pattern (Distributed Transactions)

Sequence of local transactions, each publishes event.
If step fails -> compensating transactions to undo.

Choreography

Each service listens and reacts (decoupled).
Harder to debug, no central view.

Orchestration

Central coordinator directs the saga.
Easier to understand, single point of control.

Circuit Breaker

Downstream service failing -> don't keep hammering it.

```
Closed    -> requests flow, count failures
Open      -> reject immediately (fail fast)
Half-Open -> let a few through to test recovery
```

Prevents cascade failures.
Gives downstream time to recover.

Service Mesh

Infrastructure layer for svc-to-svc communication.
Handles: mTLS, retries, circuit breaking, observability.
Sidecar proxy (Envoy) per service.
Examples: Istio, Linkerd.

Back-of-Envelope Estimation

Key Numbers

```
1 day  = 86,400s  ~ 10^5 s
1 year = 31.5M s  ~ 3x10^7 s

1 KB = 10^3    1 MB = 10^6
1 GB = 10^9    1 TB = 10^12
```

QPS Example

```
100M DAU x 10 req/day = 1B req/day
= 1B / 100K ~ 10K QPS avg
Peak ~ 2-3x avg ~ 20-30K QPS
```

Storage Example

```
1M users x 1 KB profile = 1 GB
1M users x 1 MB media   = 1 TB
1B msgs/day x 100B = 100 GB/day ~ 36 TB/yr
```

Latency Reference

```
L1 cache          ~ 1 ns
L2 cache          ~ 4 ns
RAM               ~ 100 ns
SSD random read   ~ 100 us
HDD random read   ~ 10 ms
Same DC round-trip ~ 0.5 ms
Cross-continent   ~ 100-150 ms
```

Availability & SLA

```
99%    -> 3.65 days downtime/year
99.9%  -> 8.76 hours/year
99.99% -> 52.6 minutes/year
99.999% -> 5.26 minutes/year

Series:   A -> B  = A x B (both must work)
Parallel: A || B  = 1-(1-A)(1-B) (either works)
```

Observability & Security

Observability (Three Pillars)

Metrics

Numeric time-series: counters, gauges, histograms.
USE method (resources): Utilization, Saturation, Errors.
RED method (services): Rate, Errors, Duration.
Tools: Prometheus + Grafana, Datadog, CloudWatch.

Logging

Structured logs (JSON) with correlation IDs.
Levels: DEBUG, INFO, WARN, ERROR.
Centralized: ELK (Elastic, Logstash, Kibana), Splunk.
Include: timestamp, service, trace_id, request_id.

Tracing (Distributed)

Follow request across multiple services.
Each service adds a span to the trace.
Propagate trace_id via headers (W3C Trace Context).
Tools: Jaeger, Zipkin, AWS X-Ray, OpenTelemetry.

Alerting

Alert on symptoms, not causes (error rate > CPU).
SLO-based alerts: burn rate exceeds threshold.
PagerDuty, OpsGenie for on-call routing.

Authentication & Authorization

JWT (JSON Web Token)

Stateless: token has claims, signed by server.
Header.Payload.Signature (Base64).
No server-side session storage needed.
Downside: can't revoke until expiry (short TTL + refresh).

OAuth 2.0 Flow

```
1. User -> auth provider (Google, GitHub)
2. User grants permission
3. Provider returns auth code
4. Backend exchanges code -> access + refresh token
5. Backend calls provider API with access_token
6. Issue your own JWT/session to user
```

Session-Based

Server stores session in Redis/DB.
Client holds session_id cookie.
Stateful but revocable, simpler.

RBAC: `Role-Based (admin, editor, viewer)`

ABAC: `Attribute-Based (dept=eng AND level>=5)`

ACL: `Access Control Lists (per-resource permissions)`

Security Considerations

HTTPS everywhere (TLS termination at LB/edge).
Input validation at API boundary.
Rate limiting to prevent abuse.
SQL injection: parameterized queries.
CORS: restrict allowed origins.
Secrets: Vault, AWS Secrets Manager (never in code).
Encryption at rest (DB, S3) and in transit (TLS).
Principle of least privilege for svc-to-svc auth.

Interview Framework & Common Problems

System Design Interview Framework

Step 1: Requirements (3-5 min)

Functional: what does the system DO? (core features).
Non-functional: scale, latency, availability, consistency.
Ask: DAU? Read/write ratio? Data size? Geo distribution?
Scope: what is in vs out for this interview?

Step 2: Estimation (3-5 min)

QPS (average and peak), storage (per day/year),
bandwidth, cache size needed.

Step 3: High-Level Design (10-15 min)

Draw: Client -> CDN -> LB -> API GW -> Services -> DB.
Identify core services and responsibilities.
Choose data store(s), caching, queuing.

Step 4: Detailed Design (10-15 min)

Deep dive 2-3 critical components.
Data model: tables, keys, indexes, access patterns.
API design: key endpoints, request/response.
Discuss trade-offs for each decision.

Step 5: Bottlenecks & Scale (5 min)

Single points of failure? Redundancy.
Hot spots? Distribute load.
What happens at 10x, 100x? Monitoring & alerting.

Common Design Problems

URL Shortener

Write: hash long URL -> store mapping.
Read: lookup short code -> redirect 301/302.
Read-heavy: cache-aside with Redis, shard by code.
Handle: hash collisions, analytics, expiration.

Chat System (WhatsApp/Slack)

WebSocket for real-time delivery.
Store in Cassandra (write-heavy, partition by chat_id).
Online presence: heartbeat + Redis pub/sub.
Push notifications for offline users.
Group: fan-out on write vs fan-out on read.

News Feed (Twitter/Instagram)

Fan-out on write: pre-compute feed (fast read).
Fan-out on read: assemble at read time (simple write).
Hybrid: write for normal users, read for celebrities.
Store: Redis sorted set for feed, Postgres for posts.

Notification System

Kafka for event ingestion.
Template, user preferences, dedup.
Delivery: push (APNs/FCM), email (SES), SMS, in-app.
Priority queue, rate limiting per user/channel.

Rate Limiter

Token bucket or sliding window.
Redis: INCR + EXPIRE for distributed limiting.
Key by: user_id, IP, API key.
Return 429 with Retry-After header.

Distributed File Storage (S3-like)

Metadata service: file info -> SQL DB.
Block storage: chunk files, replicate across nodes.
Consistent hashing for chunk placement.
Erasure coding for durability (vs 3x replication).

Search Autocomplete

Trie for prefix matching.
Precompute top suggestions per prefix.
Update async via MapReduce on search logs.
Cache hot prefixes in Redis, CDN for common ones.

Ride-Sharing (Uber/Lyft)

Geospatial index: QuadTree or Geohash.
Match rider to nearest available driver.
Real-time location: WebSocket or frequent polling.
ETA: Dijkstra + historical traffic.
Surge pricing: supply/demand per geo region.

Video Streaming (YouTube/Netflix)

Upload: chunk, transcode to multiple resolutions.
CDN: serve chunks from edge (most traffic).
Adaptive bitrate: client switches on bandwidth.
Metadata: Postgres. View counts: Kafka -> Cassandra.