System Design Templates

Essential Concepts for Staff-Level Interviews

1. Building Blocks Overview

Load Balancers

Purpose: Distribute traffic across servers
Types:

- **Layer 4 (Transport):** TCP/UDP routing, fast
- **Layer 7 (Application):** HTTP routing, content-aware

Algorithms:

- Round Robin - Equal distribution
- Least Connections - Favor idle servers
- IP Hash - Session persistence
- Weighted - Capacity-based routing

Examples: NGINX, HAProxy, AWS ELB/ALB

Caching

Purpose: Reduce latency, database load
Types:

- **Client-Side:** Browser cache, mobile app
- **CDN:** Static content distribution
- **Application:** Redis, Memcached
- **Database:** Query cache, materialized views

Eviction Policies:

- LRU - Least Recently Used (most common)
- LFU - Least Frequently Used
- FIFO - First In First Out
- TTL - Time To Live expiration

Cache Patterns:

- **Cache-Aside:** App reads cache, loads on miss
  Trade-off: Stale data risk vs reduced DB load
- **Write-Through:** Write to cache + DB synchronously
  Trade-off: Strong consistency vs write latency
- **Write-Back:** Write to cache, async to DB
  Trade-off: Fast writes vs data loss risk on crash
- **Read-Through:** Cache loads from DB automatically

Cache Invalidation (Critical Staff Question):

- **TTL (Time-To-Live):** Expire after fixed time
  Simple, but stale data until expiration
- **Write-Through:** Update cache on every write
  Automatically consistent, adds write latency
- **Event-Based:** Pub-sub on DB updates invalidates cache
  Complex setup, strong consistency, real-time
- **Version Tags:** Increment version on update
  Cache key includes version (user:123:v5)

Trade-off: Consistency (event/write-through) vs simplicity (TTL) vs latency

Databases

RDBMS (SQL):

- Strong consistency, ACID transactions
- Structured schema, relations
- Use: Financial, user data, complex queries
- Examples: PostgreSQL, MySQL, Oracle

NoSQL Types:

- **Key-Value:** Redis, DynamoDB - Simple lookups
- **Document:** MongoDB, Couchbase - JSON data
- **Column-Family:** Cassandra, HBase - Time-series
- **Graph:** Neo4j, Amazon Neptune - Relationships

SQL vs NoSQL Decision:

- SQL: Structured data, complex joins, transactions
- NoSQL: Scale horizontally, flexible schema, eventual consistency OK

Indexing Internals:

- **B-Tree:** Traditional RDBMS (MySQL, PostgreSQL)
  Balanced tree, good for reads, range queries
  Updates in place (read-modify-write)
- **LSM-Tree:** Write-optimized NoSQL (Cassandra, RocksDB, HBase)
  Append-only writes (sequential), periodic compaction
  Fast writes, slower reads (multiple levels to check)

Trade-off: Read performance (B-tree) vs write throughput (LSM-tree)

Message Queues

Purpose: Asynchronous communication, decoupling
Patterns:

- **Point-to-Point:** Queue, single consumer
- **Pub-Sub:** Topic, multiple subscribers

Key Features:

- Delivery Guarantees: At-most-once, at-least-once, exactly-once
- Ordering: FIFO, partitioned ordering
- Durability: Persistent vs in-memory

Examples: Kafka (high throughput), RabbitMQ (reliable), SQS (managed)

CDN (Content Delivery Network)

Purpose: Serve static content from edge locations
Benefits:

- Reduced latency (geographic proximity)
- Reduced bandwidth costs
- DDoS protection

Content Types: Images, videos, JS/CSS, APIs (with caching)
Examples: CloudFront, Cloudflare, Akamai

2. Observability Pillars

Metrics (The "What")

Purpose: Aggregated numerical data over time
Key Frameworks:

- **RED (for services):** Rate, Errors, Duration
  Rate: Requests per second
  Errors: Failed requests percentage
  Duration: Response time (p50, p95, p99)
- **USE (for resources):** Utilization, Saturation, Errors
  Utilization: % time resource busy (CPU, memory)
  Saturation: Queue depth, backlog
  Errors: Error count or rate

Tools: Prometheus, Graphite, CloudWatch
Why? Metrics provide high-level health indicators and enable alerting on SLO violations.

Logging (The "Why")

Purpose: Detailed, immutable event records
Best Practices:

- **Structured Logs (JSON):** Machine-readable, queryable
  Example: {"level": "error", "service": "api", "user_id": 123}
- **Centralized Collection:** Aggregate from all services
- **Log Levels:** DEBUG, INFO, WARN, ERROR, FATAL
- **Sampling:** Log 100% errors, sample INFO logs at scale

Tools: ELK Stack (Elasticsearch, Logstash, Kibana), Splunk, Datadog
Why? Logs answer "why did this specific request fail?" - debugging root cause.

Tracing (The "Where")

Purpose: Track a single request through multiple services
Key Concepts:

- **Trace ID:** Unique ID follows request across all services
- **Span:** Represents a single operation/service call
  Parent-child relationships form call tree
- **Sampling:** Trace 1-10% of requests (storage cost)

Tools: Jaeger, Zipkin, OpenTelemetry, AWS X-Ray
Why? Tracing identifies bottlenecks in distributed systems - "which service is slow?"

SLIs/SLOs/SLAs

SLI (Service Level Indicator):

- Quantitative measure of service health
- Examples: Latency (p99 ¡ 200ms), Availability (99.9%)

SLO (Service Level Objective):

- Target value for SLI
- Example: 99.9% of requests succeed (allows 0.1% error budget)

SLA (Service Level Agreement):

- Contract with consequences (refunds, credits)
- Example: 99.5% uptime guaranteed or customer gets credit

Error Budget:

- If SLO is 99.9%, error budget is 0.1% = 43 min downtime/month
- Balance reliability vs feature velocity

3. CAP Theorem & Consistency

CAP Theorem

Pick 2 of 3:

- **C (Consistency):** All nodes see same data
- **A (Availability):** Every request gets response
- **P (Partition Tolerance):** System works despite network splits

Trade-offs:

- **CP Systems:** MongoDB, HBase, Redis (single master)
  Sacrifice availability during partitions
- **AP Systems:** Cassandra, DynamoDB, Riak
  Sacrifice consistency for availability
- **CA:** Traditional RDBMS (single node, no partitions)

PACELC Extension

If Partition: Choose Availability or Consistency
Else (normal): Choose Latency or Consistency

- PA/EL: Cassandra (A + Low Latency)
- PC/EC: HBase (C + Strong Consistency)
- PA/EC: DynamoDB (tunable)

Consistency Models

Strong Consistency:

- Read returns latest write
- Linearizability (total ordering)
- Examples: Spanner, ZooKeeper

Eventual Consistency:

- Replicas converge eventually
- Higher availability, lower latency
- Examples: DynamoDB, Cassandra (tunable)

Causal Consistency:

- Preserves cause-effect ordering
- Weaker than strong, stronger than eventual

Read-Your-Writes:

- User sees their own updates
- Session-based consistency

4. Scaling Patterns

Vertical vs Horizontal Scaling

Vertical (Scale-Up):

- Add CPU/RAM to single machine
- Pros: Simple, no code changes
- Cons: Hardware limits, single point of failure
- Use: Initial growth, monoliths

Horizontal (Scale-Out):

- Add more machines
- Pros: No limits, fault tolerance
- Cons: Complex, consistency challenges
- Use: Web scale, distributed systems

Replication

Master-Slave (Primary-Secondary):

- Writes to master, reads from replicas
- Pros: Read scalability, simple
- Cons: Single write point, replication lag

Multi-Master:

- Multiple write nodes
- Pros: Write scalability, no single point of failure
- Cons: Conflict resolution complexity

Synchronous vs Asynchronous:

- Sync: Consistent, slower writes
- Async: Fast writes, eventual consistency

Trade-off: Strong consistency (sync) vs write performance (async)

Partitioning / Sharding

Purpose: Split data across multiple databases
Strategies:

- **Hash-Based:** hash(key) % N
  Pros: Even distribution
  Cons: Resharding difficult, no range queries
- **Range-Based:** Key ranges (A-M, N-Z)
  Pros: Range queries, easy to add shards
  Cons: Hot spots if unbalanced
- **Geography-Based:** By region/location
  Pros: Data locality, latency
  Cons: Uneven distribution
- **Consistent Hashing:** Virtual nodes, minimal resharding
  Why? Minimizes data re-shuffling when nodes added/removed (only K/n keys move)

Challenges:

- Joins across shards (avoid or denormalize)
- Distributed transactions (use sagas)
- Rebalancing during growth

Hot Shard Problem (Critical Staff Question):

- **Problem:** One shard overloaded (e.g., celebrity with 100M followers)
- **Detection:** Monitor per-shard QPS, identify outliers (10x avg)
- **Solutions:**
  1. Further partition hot entity (split celebrity into multiple shards)
  2. Dedicated cache for hot entities (Redis cluster for top 1000 users)
  3. Read replicas for hot shard (scale reads independently)
  4. Separate infrastructure (Twitter: Justin Bieber gets dedicated servers)
- **Prevention:** Hybrid sharding (hash most users, special-case celebrities)

Trade-off: Complexity (hybrid sharding) vs performance (dedicated cache)

5. Microservices Patterns

Service Communication

Synchronous:

- REST APIs - HTTP, stateless, simple
- gRPC - Binary, fast, contracts (protobuf)

Asynchronous:

- Message Queues - Decoupled, reliable
- Event Streaming - Kafka, real-time

Data Management

Database per Service:

- Each service owns its data
- Pros: Loose coupling, independent scaling
- Cons: Distributed queries, consistency

Saga Pattern (Distributed Transactions):

- **Choreography:** Events trigger next steps
- **Orchestration:** Central coordinator
- Compensating transactions for rollback

API Gateway

Purpose: Single entry point for clients
Responsibilities:

- Routing, load balancing
- Authentication, rate limiting
- Request aggregation (BFF pattern)
- Protocol translation

Service Discovery

Client-Side: Netflix Eureka, Consul
Server-Side: Kubernetes services, AWS ELB
DNS-Based: Route53, CoreDNS

6. Capacity Estimation

Key Metrics

QPS (Queries Per Second):

- Daily Users × Avg Requests/User / 86400
- Peak = Average × 3-5

Storage:

- Items × Size per Item × Replication Factor
- Factor in growth (5 years typical)

Bandwidth:

- QPS × Avg Response Size

- Separate read vs write bandwidth

Memory (Caching):

- 20% of daily requests × Response Size
- 80/20 rule: 20% data = 80% traffic

Server Count Estimation

Formula: Servers = QPS / (QPS per Server)

Assumptions:

- 1 server handles 1000-10000 QPS (web)
- 1 server handles 100-1000 QPS (DB queries)
- Add 30% headroom for safety

Example: URL Shortener

Assumptions:

- 100M new URLs per month
- 10:1 read-write ratio
- URL record = 500 bytes

Calculations:

- Write QPS = 100M / (30 × 86400) ≈ 40
- Read QPS = 40 × 10 = 400
- Storage (5 years) = 100M × 12 × 5 × 500B ≈ 3TB
- Bandwidth = 400 × 500B ≈ 200KB/s
- Cache (20% of daily) = 0.2 × 400 × 86400 × 500B ≈ 3.5GB

7. Common System Designs

URL Shortener (TinyURL)

Requirements:

- Generate short URL from long URL
- Redirect short to long (low latency)
- Custom aliases, expiration

Key Design:

- **Short Code:** Base62 encoding (7 chars = 62^7 ≈ 3.5T URLs)
- **ID Generation:** Auto-increment, UUID, or distributed ID (Snowflake)
- **Database:** Key-value store (Redis/DynamoDB)
  Schema: shortCode → {longURL, createdAt, expiresAt}
- **Cache:** LRU cache for hot URLs (80/20 rule)
- **Write Flow:** Generate ID → Encode → Store in DB
- **Read Flow:** Check cache → DB lookup → 301/302 redirect

Scale:

- Horizontal DB scaling with sharding (hash-based)
- CDN for redirects (cache 301s)
- Rate limiting per user (prevent abuse)

Chat System (WhatsApp/Slack)

Requirements:

- 1-on-1 and group chat
- Online status, message history
- Push notifications

Key Design:

- **Protocol:** WebSocket for real-time bidirectional
- **Message Flow:**
  Sender → Chat Server → Message Queue → Recipients
- **Message Storage:** Cassandra/HBase (wide-column)
  Schema: (userId, timestamp) → message
- **Online Status:** Redis with heartbeat + TTL
- **Group Chat:** Fan-out on write to all members
- **Read Receipts:** Track lastReadTimestamp per user

Scale:

- Partition users by hash to chat servers
- Message queue for reliability (Kafka)
- CDN for media (images, videos)
- Push service for offline users (FCM, APNs)

News Feed (Twitter/Instagram)

Requirements:

- Post creation, feed generation
- Follow/unfollow users
- Likes, comments (engagement)

Key Design:

- **Feed Generation:**
  Fan-out on write: Pre-compute feeds (better for reads)
  Fan-out on read: Compute on demand (better for celebrities)
  Hybrid: Fan-out for regular users, on-read for celebrities
  Trade-off: Fast reads + high storage/write cost (write) vs slow reads + low cost (read)
- **Storage:**
  Posts: Cassandra/DynamoDB (postId → content)
  Follow Graph: Redis/Graph DB (userId → followers/following)
  Feed Cache: Redis (userId → sorted list of postIds)
- **Ranking:** ML model scoring (recency, engagement, relevance)
- **Timeline:** Sorted by timestamp or personalized ranking

Scale:

- Async feed generation workers (Kafka)
- Cache feeds in Redis (LRU eviction)
- CDN for media assets
- Read replicas for graph queries

Ride Sharing (Uber/Lyft)

Requirements:

- Match riders to nearby drivers
- Real-time location tracking
- Fare calculation, ETA

Key Design:

- **Geospatial Indexing:**
  QuadTree or Geohash for location search
  Query: Find drivers within N km of rider
- **Matching:**
  Rider requests ride → Query nearby drivers
  Score by distance, rating, ETA
  Send to top K drivers (first accept wins)
- **Location Updates:** WebSocket or long polling (every 5s)
- **Storage:**
  Redis for active driver/rider locations (TTL)
  PostgreSQL/DynamoDB for trips, users, payments
- **Dispatch:** Message queue for ride requests

Scale:

- Shard by geography (city-based)
- In-memory geo index per region
- Event streaming (Kafka) for location updates
- Separate services: matching, routing, pricing

Video Streaming (YouTube/Netflix)

Requirements:

- Upload, transcode, stream videos
- Adaptive bitrate (quality based on bandwidth)
- Recommendations, watch history

Key Design:

- **Upload:** Direct to S3/GCS, async processing
- **Transcoding:** Worker fleet converts to multiple formats
  H.264/H.265, resolutions (1080p, 720p, 480p, 360p)
  Split into chunks (HLS, DASH) for streaming
- **Streaming:**
  CDN serves video chunks (CloudFront, Akamai)
  ABR protocol adjusts quality based on bandwidth
- **Metadata:** SQL/NoSQL for video info, comments
- **Recommendations:** ML pipeline (Spark, offline batch)

Scale:

- Multi-CDN for global reach
- Distributed transcoding (AWS MediaConvert)
- Pre-warming cache for popular videos
- Separate read/write paths (CQRS)

8. Additional Patterns

Rate Limiting

Algorithms:

- **Token Bucket:** Refill tokens at rate, consume on request
- **Leaky Bucket:** Fixed rate output, buffer overflow
- **Fixed Window:** Count per time window (can burst)
- **Sliding Window:** Weighted by timestamp (smooth)

Implementation: Redis with TTL, increment counters

Circuit Breaker

Purpose: Prevent cascading failures
States:

- Closed: Normal, requests pass through
- Open: Failures exceeded, reject requests
- Half-Open: Test if service recovered

Key Trade-off: Sacrifice single service availability to protect overall system stability

Idempotency

Purpose: Ensure multiple identical requests have the same effect as one
Why? Prevents duplicate processing in async systems (e.g., double charging a credit card, duplicate orders)
Implementation:

- **Unique Transaction ID:** Check if transaction_id exists before processing
- **State-based checks:** if status != 'processed'
- **Database constraints:** Unique index on transaction_id

Example (Payment):

- 1. Check if transaction_id in completed_transactions table
- 2. If exists, return success (already processed)
- 3. If not, process payment and insert transaction_id atomically

Use Cases: Payment processing, order creation, message queue consumers, API retries

Leader Election

Purpose: Designate a single node for coordination tasks
Why? Ensure exactly one node handles critical operations (writes, job scheduling, lock management)
Implementation:

- **Consensus Algorithms:** Paxos, Raft (distributed agreement)
- **Lease-based:** Lock service with TTL (ZooKeeper, etcd)
  Leader holds lease, must renew periodically
  If leader fails, lease expires, new election triggered

Use Cases:

- Database master selection (MySQL, PostgreSQL)
- Distributed lock manager (ZooKeeper coordination)
- Job scheduler coordination (only one node runs cron jobs)
- Kafka partition leader (handles reads/writes for partition)

CQRS (Command Query Responsibility Segregation)

Pattern: Separate read/write models
Benefits:

- Optimize read and write independently
- Scale reads with replicas/caches
- Complex queries without impacting writes

Event Sourcing

Pattern: Store state changes as events
Benefits:

- Full audit log
- Replay events to rebuild state
- Time travel (historical queries)

Combine with: CQRS for materialized views

Bloom Filter

Purpose: Space-efficient set membership test
Properties:

- No false negatives
- Small false positive rate (tunable)

Use Cases: Cache hit prediction, duplicate detection

Failure Modes & Recovery

Split-Brain Problem:

- Network partition creates 2 masters (both accept writes)
- **Prevention:** Quorum consensus - majority vote required
- Example: 5-node cluster needs 3 to agree on leader
- Used by: ZooKeeper, etcd, Consul

Cascading Failures:

- One service failure triggers downstream failures
- **Prevention:** Circuit breaker, bulkheads (isolate), rate limiting, timeouts
- Example: Dependency failure → retry storm → entire system down

Data Loss Scenarios:

- Async replication lag during primary crash
- **Trade-off:** Sync replication (slow writes, no loss) vs async (fast, risk loss)
- **Mitigation:** Set replication factor ≥ 3, cross-region backups

Quorum Reads/Writes:

- **Write Quorum (W):** Min nodes that must acknowledge write
- **Read Quorum (R):** Min nodes that must respond to read
- **Consistency Rule:** R + W ¿ N guarantees read sees latest write
  N = total replicas, typical: N=3, W=2, R=2 (strong consistency)
  DynamoDB: R=1, W=1 (fast, eventual) or R=2, W=2 (consistent)

Disaster Recovery:

- **RPO (Recovery Point Objective):** Max data loss tolerable (e.g., 1 hour)
- **RTO (Recovery Time Objective):** Max downtime tolerable (e.g., 15 min)
- **Strategies:** Multi-region replication, automated failover, backup/restore

9. Interview Strategy

Design Interview Framework (RADEO)

1. Requirements (5-10 min):

- Functional: Core features (what system does)
- Non-functional: Scale, latency, availability
- Constraints: Read/write ratio, data size

2. API Design (5 min):

- Define key endpoints (REST/RPC)
- Request/response schemas
- Example: POST /shorten, GET /:shortCode

3. Data Model (5-10 min):

- Database choice (SQL vs NoSQL)
- Schema design (tables/collections)
- Relationships, indexes

4. High-Level Design (10-15 min):

- Components: Client, LB, App, DB, Cache
- Data flow diagrams
- Key algorithms (hashing, ranking)

5. Deep Dives (15-20 min):

- Bottlenecks: Identify and address
- Trade-offs: Discuss alternatives
- Scale: Sharding, caching, replication

6. Operations (5 min):

- Monitoring: Metrics, alerts
- Failure modes: What can go wrong?
- Security: Auth, encryption, rate limiting

Key Trade-offs to Discuss

- Consistency vs Availability (CAP)
- Latency vs Consistency (PACELC)
- SQL vs NoSQL (structure vs scale)
- Sync vs Async replication
- Fan-out on write vs read (feeds)
- Horizontal vs Vertical scaling
- Microservices vs Monolith
- Push vs Pull (notifications)

Numbers to Remember

Time Units:

- 1 million seconds ≈ 11.5 days
- 1 billion seconds ≈ 31.7 years
- 1 day = 86,400 seconds

Latency Hierarchy (critical for Staff):

- L1 cache reference: 0.5 ns
- L2 cache reference: 7 ns
- RAM (main memory): 100 ns
- SSD random read: 150 μs (150,000 ns)
- HDD seek: 10 ms (10,000,000 ns)
- Network within datacenter: 0.5 ms
- Network cross-region (US-Europe): 150 ms

Key Insight: Memory is 1000x faster than SSD, SSD is 100x faster than HDD.
Network latency dominates in distributed systems.

Capacity & Throughput:

- QPS for 1M DAU $\approx$ 10-100 (depends on activity)
- 1 char = 1 byte, 1 int = 4 bytes, 1 long = 8 bytes
- 1 MB = 1000 KB, 1 GB = 1000 MB, 1 TB = 1000 GB, 1 PB = 1000 TB
- CDN bandwidth: 1-10 Gbps per edge server
- Database: 1000-10000 QPS per server
- Redis: 100K+ ops/sec per instance

Common Mistakes to Avoid

- Jumping to solution without clarifying requirements
- Over-engineering (keep it simple initially)
- Ignoring scale (assume billions of users)
- Not discussing trade-offs
- Forgetting about monitoring/operations
- Not asking clarifying questions
- Ignoring edge cases (failures, security)