System Design Decision Guide - When to Use What (HLD Focus)

Table of Contents

Introduction

> **"System design is about making the RIGHT decisions for YOUR specific requirements."**

This guide focuses on **decision-making** in HLD:

- ✅ When to use each approach
- ✅ What questions to ask
- ✅ Trade-offs for each choice
- ✅ Real-world examples
- ❌ No code (focus on architecture)

Load Balancer Decisions

Layer 4 vs Layer 7 Load Balancer

When to Use Layer 4 (Transport Layer)

Use When:

```
✅ Need maximum performance (low latency critical)
✅ Simple routing (all requests go to any server)
✅ Non-HTTP traffic (database connections, WebSocket, custom protocols)
✅ Very high throughput (millions of requests/second)
✅ Cost-sensitive (cheaper hardware)
```

Real Examples:

- **Database connection pooling**: Route PostgreSQL connections
- **Gaming servers**: Route UDP game traffic
- **Video streaming**: Route RTMP streams
- **IoT**: Route MQTT messages from millions of devices

Decision Factors:

```
Traffic Type: Simple/Uniform → Layer 4
Traffic Type: HTTP with routing → Layer 7

Throughput: > 1M QPS → Layer 4
Throughput: < 1M QPS → Layer 7

Protocol: HTTP only → Layer 7
Protocol: Mixed (HTTP + WebSocket + DB) → Layer 4
```

When to Use Layer 7 (Application Layer)

Use When:

```
✅ Need content-based routing (/api/users → User Service)
✅ Microservices architecture (route by URL path)
✅ Need SSL termination (decrypt once at LB)
✅ Want to inspect/modify HTTP headers
✅ A/B testing or canary deployments
✅ Need to aggregate responses from multiple services
```

Real Examples:

- **Uber**: Routes `/riders/*` to rider services, `/drivers/*` to driver services
- **Netflix**: Routes by device type (mobile vs TV vs web)
- **Amazon**: Routes `/products` to product service, `/cart` to cart service
- **Twitter**: Routes authenticated vs unauthenticated traffic differently

Decision Factors:

```
Architecture: Monolith → Layer 4
Architecture: Microservices → Layer 7

Routing Needs: IP/Port only → Layer 4
Routing Needs: URL/Headers → Layer 7

Security: Basic → Layer 4
Security: WAF, DDoS protection → Layer 7
```

Load Balancing Algorithm Choice

Round Robin

Use When:

```
✅ All requests take similar time
✅ All servers have same capacity
✅ Simple setup needed
✅ Starting point (simplest)
```

Avoid When:

```
❌ Mixed workload (uploads + API calls)
❌ Servers have different capacity
❌ Long-running requests
```

Example: Static website serving, API with uniform endpoints

Least Connections

Use When:

```
✅ Mixed request types (fast + slow)
✅ Long-lived connections (WebSocket, HTTP/2)
✅ File uploads/downloads
✅ Varying request durations
```

Avoid When:

```
❌ All requests are similar (Round Robin simpler)
❌ Very high QPS (tracking connections has overhead)
```

Example: Netflix (streaming + API), file sharing services, chat applications

---

IP Hash / Sticky Sessions

Use When:

```
✅ Server maintains session state in memory
✅ WebSocket connections
✅ Multi-step workflows
✅ Can't use external session store (legacy app)
```

Avoid When:

```
❌ Can use Redis for sessions (better!)
❌ Need even distribution
❌ Server failures cause data loss
```

Better Alternative: Always prefer Redis for sessions over sticky sessions

---

Interview Decision Framework:

```
Question: "What load balancing algorithm would you use?"

Answer Template:
"I'd use [ALGORITHM] because:

1. Traffic pattern: [uniform/mixed/long-lived]
2. Architecture: [monolith/microservices]
3. State management: [stateless/stateful]
4. Performance requirements: [throughput/latency]

Specifically:
- Layer 7 for microservices routing
- Least Connections for mixed workload
- Avoid sticky sessions by using Redis

Trade-off: [mention downside of chosen approach]"
```

Database Decisions

SQL vs NoSQL

When to Use SQL (PostgreSQL, MySQL)

Use When:

```
✅ Data is structured and relationships matter
✅ Need ACID transactions (payments, banking, inventory)
✅ Complex queries with JOINs required
```

```
✅ Schema is stable and well-defined
✅ Data integrity critical (foreign keys, constraints)
✅ Strong consistency required
✅ Team familiar with SQL
✅ Data size < 10 TB (can fit in single instance)
```

Real Examples:

- **Uber**: User accounts, payment transactions (need ACID)
- **Airbnb**: Booking system, payments (need transactions)
- **Instagram**: User profiles, relationships (structured data)
- **Stripe**: All financial data (strict consistency)

When It Breaks Down:

```
⚠ Write throughput > 10K QPS → Consider Cassandra
⚠ Data size > 10 TB → Need sharding (complex)
⚠ Schema changes frequently → Consider MongoDB
⚠ Need geographic distribution → Consider Cassandra
```

When to Use Cassandra (Wide-Column NoSQL)

Use When:

```
✅ Write-heavy workload (> 10K write QPS)
✅ Time-series data (logs, events, metrics)
✅ Need linear scalability (add nodes = add capacity)
✅ Geographic distribution required
✅ High availability critical (no single point of failure)
✅ Data size > 100 TB
✅ OK with eventual consistency
```

Real Examples:

- **Twitter**: 500M tweets/day (6K write QPS) → Cassandra
- **Netflix**: Viewing history (billions of events) → Cassandra
- **Instagram**: Activity feed (write-heavy) → Cassandra
- **Apple**: iMessage logs (billions of messages) → Cassandra

When to Avoid:

```
❌ Need ACID transactions → Use PostgreSQL
❌ Complex queries with aggregations → Use PostgreSQL
```

```
❌ Data size < 1 TB → PostgreSQL simpler
❌ Need strong consistency → Use PostgreSQL
```

When to Use MongoDB (Document Store)

Use When:

```
✅ Schema is flexible/changing
✅ Nested documents (product catalog with varying fields)
✅ Rapid development (schema can evolve)
✅ Mobile app backend
✅ Content management system
✅ Need some transactions (v4.0+)
```

Real Examples:

- **Uber**: Trip data (varying fields per trip type)
- **eBay**: Product catalog (each product different fields)
- **Forbes**: Content management system
- **Codecademy**: User progress tracking

When to Avoid:

```
❌ Need multi-document transactions → PostgreSQL better
❌ Complex JOINs required → PostgreSQL
❌ Write-heavy (> 50K QPS) → Cassandra better
```

When to Use Redis (Key-Value Store)

Use When:

```
✅ Need caching layer (< 1ms latency)
✅ Session storage with TTL
✅ Real-time leaderboards
✅ Rate limiting counters
✅ Pub/Sub messaging
✅ Sorted timelines (Twitter/Instagram feed)
```

Real Examples:

- **Twitter**: Timeline cache, rate limiting
- **Instagram**: Feed cache, session storage

- **Pinterest**: Board cache, follower lists
- **StackOverflow**: View counts, vote caching

When to Avoid:

```
❌ Primary data store (not durable enough)
❌ Data > 512 GB per instance (need clustering)
❌ Complex queries → Use Elasticsearch
❌ Need ACID transactions → Use PostgreSQL
```

Database Decision Tree

```
START: What are your requirements?

Data Structure?
├─ Structured + Relationships → SQL
│  ├─ Need transactions? YES → PostgreSQL/MySQL
│  ├─ Write QPS < 10K? YES → PostgreSQL/MySQL
│  └─ Write QPS > 10K? → Consider Cassandra
│
├─ Flexible Schema → MongoDB
│  ├─ Need transactions? → MongoDB 4.0+
│  └─ Write-heavy (> 50K QPS)? → Cassandra
│
├─ Time-Series + Write-Heavy → Cassandra
│  ├─ Logs, Events, Metrics → Cassandra
│  └─ Need strong consistency? → PostgreSQL with partitioning
│
└─ Simple Key-Value → Redis (cache) or DynamoDB (primary)

Caching Layer?
└─ ALWAYS add Redis for read-heavy workloads
```

Caching Decisions

When to Add Caching

Add Cache When:

```
✅ Read-to-write ratio > 10:1
✅ Database CPU > 50%
✅ Query latency > 100ms
✅ Same data queried repeatedly
✅ Data doesn't change frequently
```

Don't Add Cache When:

```
❌ Write-heavy workload
❌ Data changes constantly
❌ Each query unique (no cache hits)
❌ Database handling load fine
❌ Premature optimization
```

Real Decision Example:

```
Scenario: Social media feed

Metrics:
- 100K read QPS, 1K write QPS (100:1 ratio)
- Same users refresh feed multiple times
- Database CPU at 80%

Decision: ✅ ADD CACHE (Redis)
- Will handle 80K QPS (80% hit ratio)
- Reduce database to 20K QPS
- Database CPU drops to 16%

Cost: $750/month cache vs $12K/month more databases
```

Cache-Aside vs Write-Through vs Write-Behind

Use Cache-Aside When:

```
✅ Read-heavy (100:1 ratio)
✅ Data updated infrequently
✅ OK with slightly stale data (seconds)
✅ Most common pattern
```

Example: User profiles, product catalog, blog posts

Use Write-Through When:

```
✅ Need strong consistency (cache = database)
✅ Can't serve stale data
✅ Banking, inventory, booking systems
✅ Critical data that must be accurate
```

Example: Bank balance, seat reservation, stock count

Use Write-Behind When:

```
✅ Write-heavy workload
✅ OK to lose small amount of data
✅ Non-critical updates (likes, views, counters)
✅ Need sub-millisecond write latency
```

Example: Instagram likes, YouTube views, page analytics

Interview Decision:

```
Question: "What caching strategy would you use for Twitter?"

Answer:
"I'd use different strategies for different data:

User Profiles (Cache-Aside):
- Read-heavy, updates rare
- TTL: 1 hour
- 90% hit ratio expected

Tweet Likes (Write-Behind):
- Write-heavy (50K likes/second)
- Batch to database every 5 seconds
- Acceptable if count slightly off

Account Balance (Write-Through):
- Critical data, must be accurate
- Can't show wrong balance
- Worth the latency trade-off
"
```

Replication Decisions

Master-Slave vs Master-Master

Use Master-Slave When:

```
✅ Read-heavy workload (90%+ reads)
✅ Writes can go to single master
```

```
✅ Write QPS < 10K
✅ Acceptable to have replication lag
✅ Simpler to manage
```

Scaling Pattern:

```
1 Master + 1 Slave:   Handle 20K read QPS
1 Master + 5 Slaves:  Handle 100K read QPS
1 Master + 10 Slaves: Handle 200K read QPS

Write capacity: Still limited to master!
```

Real Examples:

- **Instagram**: 1 master + 12 slaves per region
- **GitHub**: MySQL with read replicas
- **Reddit**: PostgreSQL master-slave

When It Breaks:

```
⚠ Write QPS > 10K → Master becomes bottleneck → Need sharding
⚠ Master fails → Downtime during failover (30-60 seconds)
⚠ Replication lag > 1 second → Users see stale data
```

Use Master-Master When:

```
✅ Need write scalability (> 10K write QPS)
✅ High availability critical (zero downtime)
✅ Multi-region writes required
✅ Can handle conflict resolution
```

Scaling Pattern:

```
2 Masters:  Handle 20K write QPS total
4 Masters:  Handle 40K write QPS total

But: Complex conflict resolution needed!
```

Real Examples:

- **Global services**: Need writes in multiple regions

- **CockroachDB**: Distributed SQL with multi-master
- **Cassandra**: All nodes are masters (no single master)

When to Avoid:

```
❌ Simple read-heavy workload → Master-slave simpler
❌ Can't handle conflicts → Master-slave safer
❌ Small scale (< 10K write QPS) → Master-slave sufficient
```

Interview Decision:

```
Question: "Master-slave or master-master replication?"

Answer Template:
"I'd choose master-slave because:

Workload Analysis:
- Read-heavy (200:1 ratio)
- Write QPS: 2K (single master can handle)
- Read QPS: 400K (need replicas)

Decision:
- 1 Master for writes
- 10 Slaves for reads (40K QPS each)
- Simpler than master-master
- Replication lag < 500ms acceptable for this use case

Would consider master-master if:
- Write QPS > 10K
- Need zero-downtime writes
- Multi-region writes required
"
```

Sharding Decisions

When to Shard

Shard When:

```
✅ Single database can't handle load:
   - Write QPS > 10K
   - Data size > 10 TB
   - Query latency degrading

✅ Can't add more read replicas:
```

```
   - Already have 10+ replicas
   - Replication lag increasing

✅ Clear sharding key exists:
   - user_id, region, tenant_id

✅ Worth the complexity:
   - Scale justifies effort
```

Don't Shard When:

```
❌ Can add read replicas instead
❌ Can add cache instead
❌ Data < 1 TB
❌ Premature optimization
❌ No clear sharding key
```

Sharding Strategy Selection

Use Hash-Based Sharding When:

```
✅ Need even distribution
✅ No hot partitions expected
✅ Access pattern: Lookup by ID
✅ Don't need range queries

Example: Twitter shards by tweet_id
- Even distribution across 64 shards
- No celebrity hot partition
- Trade-off: Can't efficiently query "all tweets by user"
```

Use Range-Based Sharding When:

```
✅ Need range queries
✅ Time-series data
✅ Natural ranges exist

Example: Logs sharded by timestamp
- Q1 2024 → Shard 1
- Q2 2024 → Shard 2
- Efficient for "get logs from March"

Avoid: User shards by ID range (hot partitions!)
```

Use Geographic Sharding When:

```
✅ Users primarily in specific regions
✅ Data sovereignty requirements (GDPR)
✅ Low latency critical
✅ Regions have independent data

Example: Uber trips sharded by region
- US trips → US shard (50ms latency)
- EU trips → EU shard (50ms latency)
- vs 150ms cross-continent

Trade-off: Uneven distribution (US has most traffic)
```

Interview Decision:

```
Question: "How would you shard Twitter?"

Answer:
"I'd use hash-based sharding by tweet_id because:

Reasoning:
1. Write-heavy (500M tweets/day = 6K QPS)
2. Need even distribution (avoid celebrity problem)
3. Primary access: Get tweet by ID
4. Can sacrifice: Query all tweets by user

Sharding Plan:
- 64 shards
- Each handles 6K/64 = 94 write QPS
- Each stores 4 TB (manageable)

Trade-offs:
- Can't efficiently get user timeline (scatter-gather)
- Solution: Pre-compute timelines in Redis cache
- Benefits: Even load, no hot shards
"
```

Consistency Decisions

Strong vs Eventual Consistency

Use Strong Consistency When:

```
✅ Financial transactions (bank balance, payments)
✅ Inventory management (can't oversell)
✅ Booking systems (seats, hotels, tickets)
✅ User authentication (password changes)
✅ Incorrect data has serious consequences
```

Systems for Strong Consistency:

- PostgreSQL, MySQL (with single master)
- MongoDB (with majority writes)
- DynamoDB (optional, costs more)

Real Examples:

- **Banking**: Account balance MUST be accurate
- **Ticketmaster**: Can't sell same seat twice
- **Airbnb**: Can't double-book property
- **Amazon Checkout**: Inventory count accurate

Use Eventual Consistency When:

```
✅ Social media feeds (stale OK for seconds)
✅ View counts, like counts (approximate OK)
✅ Product recommendations
✅ Search results
✅ Analytics dashboards
✅ High availability more important than accuracy
```

Systems for Eventual Consistency:

- Cassandra (default)
- DynamoDB (default)
- DNS
- CDN caches

Real Examples:

- **Twitter**: Feed shows tweets within 1-2 seconds
- **Instagram**: Like count can be slightly off
- **YouTube**: View count updates every few seconds
- **Facebook**: News feed eventual consistency

Interview Decision:

```
Question: "Design a banking system"

Answer:
"I'd choose strong consistency because:

Requirements:
- Account balance must be accurate
- Can't show $100 when actually $0
- Better to show error than wrong balance

System Choice:
- PostgreSQL with ACID transactions
- Master-slave replication
- Read your own writes from master

Trade-offs:
- Slower than eventual consistency
- Less available during partitions
- Acceptable because correctness > speed

vs. Social Feed:
- Eventual consistency acceptable
- Availability > consistency
- Cassandra with AP model
"
```

CDN Decisions

When to Use CDN

Use CDN When:

```
✅ Serving media files (images, videos, documents)
✅ Static assets (CSS, JS, fonts)
✅ Global user base
✅ High bandwidth costs
✅ Origin server overwhelmed
```

Benefits Calculation:

```
Without CDN:
- Latency: 150ms (cross-continent)
- Bandwidth: All traffic from origin
- Cost: $0.09/GB egress from AWS

With CDN:
```

```
- Latency: 10ms (local edge)
- Bandwidth: 95% from CDN edge
- Cost: $0.02/GB from CloudFront
- Savings: 78% bandwidth cost + 15x faster
```

Don't Use CDN When:

```
❌ All users in single region (CDN adds complexity)
❌ Dynamic content (personalized for each user)
❌ Content changes every second
❌ Small scale (< 10K users)
```

Push vs Pull CDN

Use Push CDN When:

```
✅ Content updates infrequently
✅ Small number of files
✅ Know what will be popular
✅ Want immediate global availability
```

Example: Marketing website, documentation site, product catalog

---

Use Pull CDN When:

```
✅ User-generated content (millions of files)
✅ Don't know what will be popular
✅ Content updates frequently
✅ Large catalog
```

Example: Instagram photos, YouTube videos, Twitter images

Interview Decision:

```
Question: "Design Instagram photo delivery"

Answer:
"I'd use Pull CDN (Akamai/CloudFront) because:

Requirements:
```

```
- 95M photos uploaded daily
- Don't know which will be popular
- User-generated content

Strategy:
- Store in S3 (origin)
- CDN pulls on first request
- Popular photos cached at edge
- Unpopular photos only in S3

Benefits:
- 95% cache hit ratio
- Origin bandwidth reduced 20x
- Latency: 150ms → 10ms globally

Trade-off:
- First viewer in region sees 150ms
- Acceptable for user-generated content
"
```

Kafka vs RabbitMQ vs SQS

When to Use Kafka

Use When:

```
✅ High throughput (> 100K messages/second)
✅ Need message replay (debugging, new features)
✅ Event sourcing architecture
✅ Multiple consumers need same data
✅ Order matters (per partition)
✅ Long retention needed (days/weeks)
```

Real Examples:

- **Uber**: Ride events, surge pricing calculations
- **LinkedIn**: Activity streams, news feed generation
- **Netflix**: Viewing events, recommendation engine
- **Twitter**: Tweet fanout, analytics pipeline

When to Use RabbitMQ

Use When:

```
✅ Need complex routing (topic exchanges, headers)
✅ Task queues with worker pools
```

```
✅ Lower throughput (< 50K msg/sec)
✅ Need message acknowledgments
✅ Priority queues
```

Real Examples:

- **Instagram**: Background job processing
- **Reddit**: Comment processing, moderation queue
- **Task scheduling**: Celery with RabbitMQ backend

When to Use Amazon SQS

Use When:

```
✅ Want fully managed (no ops overhead)
✅ AWS-based architecture
✅ Don't need ordering
✅ Don't need replay
✅ Simple use case
```

Real Examples:

- **Serverless applications**: Lambda triggers
- **Microservices**: Decoupling in AWS
- **Background jobs**: Image processing, email sending

Decision Matrix:

```
Requirement           Choose
High throughput     → Kafka
Need replay         → Kafka
Event sourcing      → Kafka
Simple task queue   → RabbitMQ or SQS
AWS serverless      → SQS
Complex routing     → RabbitMQ
Managed service     → SQS
```

Communication Decisions

REST vs GraphQL vs gRPC

Use REST When:

```
✅ Public API (most developers know REST)
✅ CRUD operations (standard HTTP verbs)
✅ Caching important (HTTP caching)
✅ Simple requirements
✅ Resource-oriented data
```

Example: Twitter API, GitHub API, Stripe API (all REST)

Use GraphQL When:

```
✅ Client needs flexibility (mobile apps)
✅ Multiple resources per request
✅ Avoid over-fetching/under-fetching
✅ Rapidly changing requirements
✅ BFF pattern (Backend for Frontend)
```

Example: Facebook, GitHub (alongside REST), Shopify

Use gRPC When:

```
✅ Internal microservice communication
✅ Need high performance (binary protocol)
✅ Bi-directional streaming
✅ Strong typing required
✅ Polyglot environment (multiple languages)
```

Example: Google's internal services, Uber's microservices

WebSocket vs Polling

Use WebSocket When:

```
✅ Real-time bi-directional communication
✅ Chat applications
✅ Live notifications
✅ Collaborative editing
✅ Gaming
✅ Stock tickers
```

Example: Slack messages, WhatsApp Web, Google Docs

Scaling Challenge: Each connection consumes memory

```
1M concurrent WebSocket connections:
- 1 MB per connection (buffers)
- Total: 1 TB memory needed
- Solution: Multiple WebSocket servers, sticky routing
```

Use Polling (Long or Short) When:

```
✅ WebSocket not supported (legacy systems)
✅ Firewall restrictions
✅ Simple occasional updates
✅ Don't need sub-second latency
```

Example: Legacy chat systems, simple notifications

Efficiency:

```
Short Polling: 3,600 requests/hour per user
Long Polling: 60 requests/hour per user
WebSocket: 1 connection for entire session
```

Scaling Decisions by User Count

0 - 10,000 Users

Architecture:

```
[Users] → [Single Server] → [PostgreSQL]

Keep it simple!
```

Decisions:

- ❌ No load balancer yet
- ❌ No caching (database fast enough)
- ❌ No replication (can tolerate downtime)
- ✅ Vertical scaling only
- ✅ Managed database (RDS)

Cost: ~$200/month

10,000 - 100,000 Users

Architecture:

```
[Users] → [Single Server (upgraded)] → [RDS PostgreSQL]
                                         → [Redis Cache]
```

Decisions:

- ✅ Vertical scaling (bigger server)
- ✅ Add Redis cache (80% hit ratio)
- ✅ Use managed database (RDS)
- ❌ No load balancer yet (single server OK)
- ❌ No replication yet

Cost: ~$800/month

100,000 - 1M Users

Architecture:

```
[Users] → [Load Balancer]
               ↓
        [10 App Servers]
               ↓
         [Redis Cluster]
               ↓
    [PostgreSQL Master + 3 Slaves]
```

Decisions:

- ✅ Horizontal scaling (10 servers)
- ✅ Load balancer (Layer 7)
- ✅ Redis cluster (distributed cache)
- ✅ Database replication (master-slave)
- ✅ CDN for media files
- ❌ No sharding yet

Cost: ~$5,000/month

1M - 10M Users

Architecture:

```
[Users] → [CDN] → [Load Balancer]
                        ↓
                  [100 App Servers]
                        ↓
               [Redis Cluster (10 nodes)]
                        ↓
        [PostgreSQL Sharded (4 shards)]
        [Cassandra (for write-heavy data)]
                        ↓
                   [Kafka]
                        ↓
            [Background Workers]
```

Decisions:

- ✅ Database sharding (PostgreSQL can't handle writes)
- ✅ Add Cassandra for write-heavy data
- ✅ Kafka for async processing
- ✅ Microservices architecture
- ✅ Multi-region deployment starts

Cost: ~$50,000/month

10M - 100M Users

Architecture:

```
Multi-Region:

Region 1 (50% traffic):
  [CDN] → [LB] → [500 Servers] → [Cache] → [DB Shards]

Region 2 (30% traffic):
  [CDN] → [LB] → [300 Servers] → [Cache] → [DB Shards]

Region 3 (20% traffic):
  [CDN] → [LB] → [200 Servers] → [Cache] → [DB Shards]
```

Decisions:

- ✅ Multi-region for global users
- ✅ Extensive sharding (32+ shards)
- ✅ Multiple NoSQL databases (Cassandra, DynamoDB)
- ✅ Kafka clusters for event streaming
- ✅ Separate analytics pipeline

Cost: ~$500,000/month

Decision Trees

Decision Tree: Choose Database

```
START: What's your data?

Structured + Relationships?
├─ YES → Need Transactions?
│   ├─ YES → PostgreSQL/MySQL
│   └─ NO → MongoDB (if schema flexible)
│
└─ NO → What's your workload?
    ├─ Write-Heavy (> 10K QPS) → Cassandra
    ├─ Read-Heavy → MongoDB or PostgreSQL + Cache
    ├─ Key-Value only → Redis or DynamoDB
    └─ Time-Series → Cassandra or InfluxDB

Special Needs:
├─ Full-Text Search? → Add Elasticsearch
├─ Graph Relationships? → Add Neo4j
├─ File Storage? → S3 or equivalent
└─ Analytics? → Redshift or BigQuery
```

Decision Tree: Choose Caching Strategy

```
START: Do you need cache?

Read-to-Write Ratio?
├─ > 10:1 → YES, add cache
│   └─ What data?
│       ├─ User profiles → Cache-Aside, TTL 1hr
│       ├─ Social feed → Cache-Aside, TTL 5min
│       ├─ Product catalog → Cache-Aside, TTL 1day
│       └─ Session data → Write-Through, TTL 30min
│
├─ < 10:1 → Maybe not needed
│   └─ Database handling load?
│       ├─ YES → Skip cache
│       └─ NO → Add cache-aside
│
└─ Write-Heavy → Write-Behind
    └─ Examples: View counts, like counts
```

Decision Tree: Choose Load Balancer Type

```
START: What's your architecture?

Microservices?
├─ YES → Layer 7
│   └─ Route by URL path
│
└─ NO → Monolith
    └─ What protocol?
        ├─ HTTP only → Layer 7 (for SSL termination)
        └─ Mixed protocols → Layer 4

Throughput?
├─ > 1M QPS → Layer 4 (faster)
└─ < 1M QPS → Layer 7 (more features)

Budget?
├─ High → Managed LB (AWS ALB, Google CLB)
└─ Low → Self-managed (NGINX, HAProxy)
```

Quick Decision Reference

For Every System Design Question

1. Choose Database:

```
User Data, Transactions → PostgreSQL
Posts, Feed (write-heavy) → Cassandra
Search → Elasticsearch
Cache → Redis
Files → S3
Analytics → Redshift
```

2. Add Caching:

```
Read-heavy (> 10:1) → Always add Redis
TTLs:
- User profiles: 1 hour
- Feeds: 5 minutes
- Static data: 24 hours
```

3. Choose Replication:

```
Read-heavy → Master-Slave (add replicas)
Write-heavy → Shard or Cassandra
```

```
Both heavy → Cassandra (masterless)
```

4. Add CDN:

```
Media files → Always use CDN
Global users → CDN mandatory
Cost > $1000/mo bandwidth → CDN pays for itself
```

5. Message Queue:

```
Need async → Always add queue (Kafka/RabbitMQ)
High throughput → Kafka
Simple tasks → RabbitMQ or SQS
```

Complete Interview Examples

Example 1: "Design Twitter" - Decision Walkthrough

Step 1: Clarify Requirements

```
Scale: 400M DAU, 500M tweets/day
Read-heavy: 200:1 ratio
Latency: < 200ms for feed
Availability: 99.99%
```

Step 2: Database Decisions

User Data → PostgreSQL

```
Why:
- Structured (user profiles, relationships)
- Need ACID for account operations
- Moderate write QPS (user updates)

Scaling:
- Master-slave replication
- 10 read replicas
- Shard by user_id when > 100M users
```

Tweets → Cassandra

```
Why:
- Write-heavy (6K QPS)
- Time-series (sorted by time)
- No ACID needed
- Need linear scalability

Scaling:
- 64 shards (Cassandra nodes)
- Each handles 94 write QPS
- Geographic distribution
```

Step 3: Caching Decision

Timeline Cache → Redis Sorted Sets

```
Why:
- Read QPS: 463K (database can't handle)
- Perfect for sorted timeline
- 80% hit ratio expected

Configuration:
- Redis Cluster (20 nodes)
- 2 TB total cache
- TTL: 2 days
```

Step 4: CDN Decision

Media → CloudFront + S3

```
Why:
- 40M images per day
- Global users
- Reduce origin bandwidth

Benefits:
- 95% cache hit ratio
- Latency: 150ms → 10ms
- Bandwidth cost: 20x cheaper
```

Step 5: Async Processing

Feed Fanout → Kafka

```
Why:
- Don't block tweet posting
- Multiple consumers (notifications, search, analytics)
```

```
- Handle traffic spikes
- Can replay events

Architecture:
- Producer: Tweet Service
- Consumers: Feed Fanout, Notifications, Search Indexer
```

Step 6: Load Balancer

Layer 7 ALB

```
Why:
- Microservices (route by path)
- SSL termination
- Geographic routing
- Health checks
```

Final Architecture Summary:

```
[CDN] → [Layer 7 LB] → [Microservices]
                             ↓
                       [Redis Cache]
                             ↓
                  [PostgreSQL + Cassandra]
                             ↓
                         [Kafka]
                             ↓
                    [Background Workers]
```

Example 2: "Design Uber" - Decision Walkthrough

Step 1: Key Decisions

Real-Time Locations → Redis Geo

```
Why:
- Need < 50ms latency
- Geo-spatial queries (find drivers within 5km)
- High write volume (location updates every 3 seconds)
- Small dataset (active drivers fit in memory)

Alternative considered:
- PostgreSQL with PostGIS → Too slow for real-time
- Cassandra → No geo-spatial queries
- Redis Geo → Perfect fit
```

```
```

Trip History → Cassandra

```
Why:
- Time-series data
- Write-heavy (50M trips/day)
- Long-term storage (years)
- Geographic distribution

Alternative considered:
- PostgreSQL → Can't handle write volume
- MongoDB → Cassandra better for time-series
```

Payments → PostgreSQL

```
Why:
- ACID transactions mandatory (can't lose money!)
- Strong consistency required
- Financial data (audit trail)

NO alternatives acceptable for payments!
Must use SQL with ACID
```

Surge Pricing → Kafka + Redis

```
Why Kafka:
- Real-time event streaming
- Multiple consumers (riders, drivers, analytics)
- High throughput

Why Redis:
- Fast aggregations (count requests per region)
- Real-time counters
- Sub-millisecond latency
```

Example 3: "Design Instagram" - Decision Walkthrough

Photo Storage → S3 + CloudFront CDN

```
Why S3:
- Durable (11 9's)
- Infinite scalability
- Cost-effective ($0.023/GB)
- 95M photos/day = 19 TB/day
```

```
Why CDN:
- Global users
- 95% cache hit ratio
- Latency: 150ms → 10ms
- Bandwidth: 20x cheaper

Alternative considered:
- Store on app servers → Not scalable
- EBS volumes → Expensive, not durable
- S3 + CDN → Clear winner
```

Feed Generation → Hybrid Fan-out

```
Decision: Hybrid (Push + Pull)

Normal users (< 10K followers):
- Fan-out on write (push to followers)
- Pre-compute in Redis
- Fast reads

Celebrities (> 10K followers):
- Fan-out on read (pull on demand)
- Avoid 10M writes per tweet
- Merge with pre-computed feed

Why hybrid:
- Solves celebrity problem
- Fast for everyone
- Optimal resource usage
```

Photo Metadata → PostgreSQL + Cassandra

```
PostgreSQL for:
- User profiles
- Relationships

Cassandra for:
- Photo metadata (write-heavy)
- Activity feed
- Comments/likes

Why both:
- Use right tool for each job
- PostgreSQL: Relationships need SQL
- Cassandra: Feed needs write scalability
```

Common Interview Questions

Q: "Your database is slow, what do you do?"

Decision Process:

Step 1: Identify Bottleneck

```
Check metrics:
- CPU > 80% → Add cache or replica
- Disk I/O maxed → Add SSD or cache
- Memory low → Increase RAM or cache
- Connections maxed → Connection pooling
```

Step 2: Quick Wins

```
1. Add indexes (10-100x speedup)
2. Add Redis cache (5-10x fewer DB queries)
3. Query optimization (avoid N+1)
4. Connection pooling
```

Step 3: Architectural Changes

```
Read-heavy:
→ Add read replicas (scale reads)

Write-heavy:
→ Add cache (write-behind for non-critical)
→ Consider Cassandra

Both heavy:
→ Shard database
→ Or move to Cassandra
```

Q: "How do you handle 1M concurrent WebSocket connections?"

Decision:

```
Problem: 1M connections × 1 MB = 1 TB memory

Solution: Distribute across servers

Architecture:
[Users] → [LB with Sticky Sessions]
```

```
              ↓
         [100 WS Servers]
         (10K connections each)
              ↓
          [Redis Pub/Sub]
         (message broadcasting)

Why sticky:
- WebSocket is stateful
- Must route to same server
- Use IP hash or session cookie

Scaling:
- Add servers → 10K connections each
- 100 servers → 1M connections
```

Q: "When would you use Cassandra over PostgreSQL?"

Decision Framework:

```
Choose Cassandra when:

1. Write throughput > 10K QPS
   Example: Twitter (6K QPS average, 60K peak)

2. Time-series data
   Example: Netflix viewing history, IoT sensor data

3. Geographic distribution
   Example: Global app needs low latency everywhere

4. High availability critical
   Example: Can't tolerate single point of failure

5. Data size > 100 TB
   Example: Instagram activity feed

Choose PostgreSQL when:
1. Need ACID transactions
2. Complex JOINs required
3. Data < 10 TB
4. Write QPS < 10K
5. Strong consistency required
```

Q: "How do you prevent database from being overwhelmed?"

Multi-Layer Defense:

Layer 1: Rate Limiting

```
Prevent abuse at API Gateway
- 1000 requests/hour per user
- Protects all downstream systems
```

Layer 2: Caching

```
Redis cache intercepts 80% of reads
- Database only sees 20%
- 5x load reduction
```

Layer 3: Connection Pooling

```
Limit concurrent DB connections
- Pool of 50 connections
- Prevents connection exhaustion
- Graceful degradation under load
```

Layer 4: Read Replicas

```
Distribute reads across slaves
- 10 replicas → 10x read capacity
- Master only handles writes
```

Layer 5: Sharding (Last Resort)

```
When all else fails, shard
- 64 shards → 64x capacity
- But adds complexity
```

Trade-Off Analysis

Every Decision Has Trade-Offs

Caching Trade-offs:

```
Benefits:
✅ 10-100x faster reads
```

```
✅ Reduce database load 80%+
✅ Better user experience

Costs:
❌ Stale data possible
❌ Cache invalidation complexity
❌ Additional infrastructure
❌ More memory needed
```

Sharding Trade-offs:

```
Benefits:
✅ Scale writes linearly
✅ Scale storage linearly
✅ No single database limit

Costs:
❌ Can't JOIN across shards
❌ Complex queries difficult
❌ Rebalancing expensive
❌ Application complexity
```

CDN Trade-offs:

```
Benefits:
✅ 10-15x faster (global)
✅ 20x cheaper bandwidth
✅ Reduce origin load 95%

Costs:
❌ Cache invalidation delays
❌ Additional cost
❌ Complexity
❌ First request slow (miss)
```

Microservices Trade-offs:

```
Benefits:
✅ Independent scaling
✅ Team autonomy
✅ Technology flexibility
✅ Fault isolation

Costs:
❌ Network latency between services
❌ Distributed debugging hard
```

```
❌  Operational complexity
❌  Data consistency challenges
```

The Universal Decision Framework

For ANY Component Decision

```
1. REQUIREMENTS
   What do we need?
   - Scale (QPS, storage)
   - Latency targets
   - Consistency requirements
   - Availability requirements

2. OPTIONS
   What are the choices?
   - List 2-3 alternatives
   - Know what each offers

3. TRADE-OFFS
   What do we sacrifice?
   - Cost
   - Complexity
   - Performance
   - Consistency

4. DECISION
   What do we choose?
   - Pick based on requirements
   - Justify with trade-offs
   - Mention alternatives considered

5. SCALE PLAN
   How does this scale?
   - What happens at 10x growth?
   - When do we need to change?
```

Key Principles

Start Simple, Scale When Needed

Phase 1: MVP (< 100K users)

```
Keep it simple:
- Single server
- Managed database (RDS)
```

```
- No caching
- No load balancer

Why: Fast to build, cheap, good enough
```

Phase 2: Growth (100K - 1M users)

```
Add essentials:
- Load balancer
- Horizontal scaling
- Redis cache
- Database replicas
- CDN for media

Why: Needed for scale, proven patterns
```

Phase 3: Scale (1M - 10M users)

```
Add advanced:
- Database sharding
- Kafka for events
- Microservices
- Multi-region

Why: Single region/database can't handle load
```

Phase 4: Hyper-Scale (> 10M users)

```
Optimize everything:
- Multiple NoSQL databases
- Extensive caching
- Global CDN
- Custom infrastructure

Why: At this scale, custom solutions pay off
```

Default Technology Choices

For 90% of interviews, default to:

```
Load Balancer: Layer 7 (ALB, NGINX)
Database: PostgreSQL (users), Cassandra (events)
Cache: Redis
```

```
CDN: CloudFront / Akamai
Message Queue: Kafka
Search: Elasticsearch
File Storage: S3
Analytics: Redshift

These are safe, proven choices!
```

Only deviate when you can justify:

- "I'd use X instead of Y because [specific reason]"
- Always mention the trade-off

Interview Scoring

What Interviewers Look For

In Decision-Making:

Strong Candidate ✅ :

```
- States assumptions clearly
- Considers multiple options
- Explains trade-offs
- Justifies with requirements
- Mentions when to change approach
- Uses real company examples
```

Weak Candidate ❌ :

```
- "I'd use MongoDB" (no justification)
- Doesn't mention alternatives
- No trade-off discussion
- Over-engineers (Kafka for 100 users)
- Under-engineers (PostgreSQL for 1B writes/day)
```

Final Checklist

Before Finishing Any Design

Ask Yourself:

✅ **Scalability**: How does this scale to 10x? 100x?

- "At 10x traffic, we'd add sharding"

- "At 100x, we'd move to Cassandra"

✅ **Availability**: How do we handle failures?

- "Replication across 3 zones"
- "Automatic failover in 30 seconds"

✅ **Consistency**: Strong or eventual?

- "Strong for payments (PostgreSQL)"
- "Eventual for feed (Cassandra)"

✅ **Trade-offs**: What did we sacrifice?

- "Chose availability over consistency for feed"
- "Sharding sacrifices JOINs for write scalability"

✅ **Bottlenecks**: What will break first?

- "Database will bottleneck at 10K write QPS"
- "Solution: Shard or move to Cassandra"

✅ **Cost**: Is this cost-effective?

- "CDN saves $10K/month in bandwidth"
- "Cache reduces database costs 80%"

Summary: The Golden Rules

1. **Start with SQL** (PostgreSQL) unless you have specific reason not to
2. **Add Redis cache** for any read-heavy workload (> 10:1)
3. **Use CDN** for any media files and global users
4. **Layer 7 LB** for microservices, Layer 4 for performance
5. **Master-Slave** for read-heavy, shard for write-heavy
6. **Strong consistency** for money, eventual for social
7. **Kafka** for high throughput async, RabbitMQ for tasks
8. **Cassandra** for write-heavy time-series, PostgreSQL otherwise
9. **Always mention trade-offs** - nothing is free
10. **Scale incrementally** - don't over-engineer early

Document Version: 1.0
Last Updated: January 8, 2025
For: System Design HLD Interview Preparation
Status: Complete - Pure Decision-Making Guide ✅

Remember: It's not about knowing every technology - it's about making the RIGHT choice for the SPECIFIC requirements!