Caching in System Design - Complete HLD Interview Guide

Table of Contents

What is Caching & Why It Matters

Core Concept

```
Without Cache: Request → Database (50-100ms)
With Cache:    Request → Cache (1ms) ✓ [10-100x faster]
```

The 80/20 Rule

```
80% of requests hit 20% of data
→ Cache that 20% for massive gains

Example: 1M products, cache 200K popular ones
Result: 80% requests served from cache (1ms vs 50ms)
```

Performance & Cost Impact

```
E-commerce Example:

Without Cache:
├── Response time: 350ms
├── DB servers: 10 servers for 10K QPS
```

```
└── Cost: $10,000/month

With Cache (95% hit rate):
├── Response time: 3ms (117x faster!)
├── DB servers: 1 server for 10K QPS
└── Cost: $1,500/month (85% savings!)
```

When NOT to Cache

```
❌ Data changes every second (real-time stock prices)
❌ Hit rate <10% (overhead > benefit)
❌ Strict consistency needed (financial transactions)
❌ Highly personalized, no reuse
❌ Write-heavy workload
```

Cache Hierarchy & Layers

Complete Stack

```
Layer 1: CLIENT-SIDE (Browser/Mobile)
         Latency: 0ms | Hit Rate: 30-50%
              ↓
Layer 2: CDN / EDGE (CloudFront, Cloudflare)
         Latency: 10-50ms | Hit Rate: 70-90%
              ↓
Layer 3: APPLICATION (In-Memory, Local)
         Latency: 0.1-1ms | Hit Rate: 40-60%
              ↓
Layer 4: DISTRIBUTED (Redis, Memcached)
         Latency: 1-5ms | Hit Rate: 80-95%
              ↓
Layer 5: DATABASE CACHE (Buffer pool, Query cache)
         Latency: 5-20ms | Hit Rate: 60-80%
              ↓
Layer 6: DISK (SSD/NVMe)
         Latency: 50-100ms
```

Layer Use Cases

```
┌──────────────────┬──────────────────────────────────┐
│ Layer            │ Best For                         │
├──────────────────┼──────────────────────────────────┤
│ Client-Side      │ Static assets, user preferences  │
│ CDN              │ Images, videos, global content   │
```

```
| Application          | Config, sessions, rate limits    |
| Distributed          | User data, product catalog       |
| Database             | Automatic (query results)        |
```

Caching Strategies

Strategy Comparison

Strategy	Read Path	Write Path	Best For
Cache-Aside (Lazy Load)	App→Cache→DB→Cache	App→DB→Invalidate	Read-heavy Most common ⭐
Read-Through	App→Cache→DB	App→DB→Invalidate	Simpler app code
Write-Through	App→Cache	App→Cache→DB (sync)	Strong consistency
Write-Back	App→Cache	App→Cache→DB (async)	Write-heavy ⚠ Data loss risk
Write-Around	App→Cache→DB	App→DB (skip cache	Write-once, read-rarely

1. Cache-Aside (Lazy Loading) - Most Common ⭐

```
Read: App checks cache → Miss → Query DB → Store in cache
Write: Update DB → Invalidate cache

When to Use:
✓ Read-heavy workload (10:1 ratio)
✓ Data changes infrequently
✓ Most general-purpose scenarios

Examples: User profiles, product catalogs, blog posts
```

2. Write-Through - Strong Consistency

```
Read: App reads from cache (always present)
Write: App writes to cache → Cache writes to DB (sync)
```

```
When to Use:
✓ Need strong consistency
✓ Read-after-write pattern
✓ Can tolerate higher write latency

Examples: Shopping cart, user preferences, session data
```

3. Write-Back - High Write Performance

```
Read: App reads from cache
Write: App writes to cache → Async DB write (queued)

When to Use:
✓ Write-heavy workload
✓ Can tolerate eventual consistency
✓ Write performance critical

⚠ Risk: Data loss if cache crashes

Examples: Analytics, logging, metrics
❌ NOT for: Financial transactions
```

Cache Eviction Policies

LRU (Least Recently Used) - Recommended ⭐

```
Algorithm: Evict least recently accessed item

┌──────────────────────────────────────────┐
│  MRU: [E] ↔ [C] ↔ [A]                    │
│  LRU: ← EVICT                            │
└──────────────────────────────────────────┘

Access item → Move to front
Cache full → Remove from back
Time Complexity: O(1)

Best For:
✓ General purpose (recommended default)
✓ Web applications
✓ 90% of use cases
```

LFU (Least Frequently Used)

```
Algorithm: Evict item with lowest access count

Freq 5: [A]
Freq 3: [B] [C]
Freq 1: [D] [E] ← EVICT

Best For:
✓ Stable access patterns
✓ Content recommendation
✗ New items disadvantaged
```

Policy Comparison

Policy	Hit Rate	Complexity	Recommended
LRU	High	O(1)	Yes ⭐
LFU	Higher	O(log N)	Stable only
FIFO	Low	O(1)	No
Random	Lowest	O(1)	No

Cache Invalidation

"There are only two hard things in Computer Science: cache invalidation and naming things" - Phil Karlton

Invalidation Strategies

1. TTL (Time-To-Live) - Simplest

```
Set expiry on cached data
Automatic eviction after TTL

TTL Selection Guide:
├── Static content: 1 year
├── Product catalog: 1 hour
├── User profiles: 5–15 minutes
├── Real-time data: 5–30 seconds
└── Session data: Session duration

Pros: Simple, automatic
Cons: Data stale until expiry
```

2. Event-Based - Most Accurate

```
On data update → Invalidate cache immediately

Workflow:
  Update DB → Delete/update cache entry

Pros: No stale data, immediate consistency
Cons: Code coordination, complexity
```

3. Message Queue - Microservices

```
Architecture:
  Service A updates → Kafka event → All services invalidate

Pros: Decoupled, scalable
Cons: Eventual consistency, more infrastructure
```

Cache Stampede ("Thundering Herd")

Problem:

```
Popular item expires → 10K concurrent requests
→ All miss cache → All hit DB → DB crashes
```

Solutions:

```
1. Lock-Based:
   First request locks → queries DB
   Others wait → read from cache
   Result: Only 1 DB query

2. Probabilistic Early Expiration:
   Refresh before actual expiry (randomly)
   Spreads load over time

3. Serve Stale:
   Return expired data immediately
   Refresh asynchronously
   Users always get fast response

Recommended: Combination of #2 and #3
```

Distributed Caching

Redis Cluster

```
┌─────────────────────────────────────┐
│ 16,384 hash slots across nodes      │
│                                     │
│ Master 1: Slots 0-5460              │
│    ↓ (replica)                      │
│ Master 2: Slots 5461-10922          │
│    ↓ (replica)                      │
│ Master 3: Slots 10923-16383         │
│    ↓ (replica)                      │
└─────────────────────────────────────┘

Routing: slot = CRC16(key) % 16384
Benefits: Horizontal scaling, fault tolerance
```

Consistent Hashing

```
Problem: Traditional hashing (hash % N)
  Add/remove server → 75% cache misses!

Solution: Consistent hashing (hash ring)
  Add/remove server → Only 25% remapped ✓
```

Replication

```
Master-Slave (Recommended):
  Writes → Master
  Reads → Slaves (3+ replicas)

Benefits:
✓ Fast writes
✓ Read scalability
✓ High availability
```

When to Recommend Caching (Interview Focus)

✅ STRONG SIGNALS - Always Mention Caching

1. Read-Heavy Workload (>10:1 ratio)

```
Signals:
- "Users read profiles 1000x more than update"
- "Products viewed millions of times daily"

What to Say:
"This 100:1 read:write ratio is perfect for caching.
With 95% hit rate, we reduce DB load by 95% and
improve latency from 50ms to 1ms."
```

2. Expensive Operations

```
Signals:
- "5-table joins required"
- "ML recommendations take 200ms"
- "Complex aggregations"

What to Say:
"These expensive operations (200ms) benefit hugely
from caching. Cache hit returns in 1ms - that's
200x faster and 99% cost savings."
```

3. Hot Data Pattern (80/20 Rule)

```
Signals:
- "Top 20% products get 80% traffic"
- "Celebrity posts have millions of views"
- "Popular searches repeated"

What to Say:
"Classic 80/20 distribution. Caching top 20% of data
with 1GB cache will serve 80% of requests at 1ms
instead of 50ms from database."
```

4. Static or Slow-Changing Data

```
Signals:
- "Product descriptions rarely change"
- "Configuration updated daily"
- "Historical/immutable data"

What to Say:
"This static data is ideal for caching with long TTL
(hours to days). We can achieve 95%+ hit rates with
minimal invalidation complexity."
```

5. Performance Critical (<100ms requirement)

```
Signals:
- "Need sub-100ms response"
- "Real-time user experience"
- "100K+ requests per second"

What to Say:
"To meet sub-100ms requirement at 100K QPS, caching
is essential. Multi-tier strategy: CDN (10ms) + Redis
(1ms) + local cache (0.1ms) = consistent performance."
```

❌ RED FLAGS - Don't Recommend Caching

1. Frequently Changing Data

```
Signals:
- "Updates every second"
- "Real-time stock prices"
- "Live sports scores"

What to Say:
"Data changes too frequently (<1s TTL). Cache hit rate
would be <10%, not worth the complexity. Direct DB
queries or streaming solution more appropriate."
```

2. Strong Consistency Required

```
Signals:
- "Financial transactions"
- "Payment processing"
- "Legal/compliance data"

What to Say:
"Strong consistency required here. Caching introduces
eventual consistency which is unacceptable. We'll
use database directly with proper transaction isolation."
```

3. Write-Heavy Pattern

```
Signals:
- "More writes than reads"
- "Data written but rarely read"

What to Say:
"Write-heavy pattern doesn't benefit from caching.
We'd spend more time invalidating cache than serving
from it. Better to optimize database writes directly."
```

Top 15 HLD Questions with Caching

1. Design Twitter/Instagram ⭐⭐⭐

Where to Cache:

```
✅ User timelines (Redis, 5 min TTL)
✅ Profile information (Redis, 15 min)
✅ Images/videos (CDN, 1 year)
✅ Trending topics (Redis, 1 min)
✅ Follower counts (Redis, 10 min)
```

What to Say:

```
"Three-tier caching strategy:

1. CDN for media (1 year TTL):
   - 90% of bandwidth
   - 200ms → 20ms latency

2. Redis for timelines (5 min TTL):
   - Expensive join operations
   - 50ms → 1ms per timeline
   - Hit rate: 85-90%

3. Invalidation:
   - New post: Invalidate follower timelines
   - Fanout-on-write: Regular users
   - Fanout-on-read: Celebrities (1M+ followers)

Result: Serve 10M users with 10x less infrastructure"
```

2. Design YouTube/Netflix ⭐⭐⭐

Where to Cache:

```
✅ Video content (CDN, immutable)
✅ Metadata (Redis, 1 hour)
✅ Recommendations (Redis, 30 min)
✅ Thumbnails (CDN, 1 day)
✅ Watch history (Redis, 5 min)
```

What to Say:

```
"Caching is fundamental here:

1. CDN for videos (90% of traffic):
   - Edge caching globally
   - Mumbai user: 20ms (local edge) vs 200ms (US origin)
   - Bandwidth savings: ~80%

2. Metadata caching (Redis):
   - Video details rarely change
   - 1 video = millions of metadata requests
   - DB: 1K QPS → Cache: 100K QPS

3. Recommendations (pre-computed):
   - Expensive ML (200ms)
   - Cache per user segment
   - Update every 30 minutes

Impact: Page load 2s → 500ms (4x faster)"
```

3. Design URL Shortener ⭐⭐⭐

Where to Cache:

```
✅ Popular URLs (Redis, no TTL)
✅ Click counts (Redis, 1 min)
```

What to Say:

```
"Perfect caching scenario:

1. Extreme Read-Heavy: 100:1 ratio
   - URL created once, clicked thousands of times
   - Viral links: millions of clicks

2. Simple key-value:
   - short_url → original_url
   - Redis sub-millisecond lookup
```

```
3. 80/20 Rule applies:
   - Top 20% URLs = 80% traffic
   - 1GB cache = 10M URLs
   - LRU eviction

4. Performance:
   - Without cache: 50ms
   - With cache: 1ms (50x faster!)
   - Hit rate: 95%+

Can serve 1M QPS with minimal infrastructure"
```

4. Design Amazon E-commerce ⭐⭐⭐

Where to Cache:

```
✅ Product catalog (Redis, 1 hour)
✅ Product images (CDN, 1 year)
✅ Search results (Redis, 30 min)
✅ Shopping cart (Redis, 2 hours)
✅ Inventory (Redis, 30 sec)
```

What to Say:

```
"Multi-layer caching crucial for e-commerce:

1. Product Catalog (1 hour TTL):
   - 1M products, top 20% = 80% traffic
   - Cache popular products in Redis
   - Hit rate: 90%+

2. Inventory/Pricing (30 sec TTL):
   - More dynamic, shorter TTL
   - Write-through for consistency
   - No stale stock issues

3. Search Results (30 min):
   - Top 1000 searches = 60% of all searches
   - Expensive Elasticsearch queries cached

4. CDN for images (1 year):
   - Product photos served from edge
   - Hit rate: 95%

Impact:
- Page load: 200ms → 50ms (75% faster)
```

```
- Cache hit rate: 90%+
- DB load reduction: 85%"
```

5. Design WhatsApp/Messenger ⭐⭐

Where to Cache:

```
✅ Recent conversations (Redis, 5 min)
✅ Online status (Redis, 5 sec)
✅ Unread counts (Redis, real-time)
✅ Profile pictures (CDN, 1 day)
```

What to Say:

```
"Real-time messaging needs aggressive caching:

1. Conversations (5 min TTL):
   - Users check same 20 conversations repeatedly
   - Cache last 50 conversations per user
   - 80% DB load reduction

2. Online Status (5 sec TTL):
   - Millions of status checks per second
   - Redis Pub/Sub for real-time updates
   - Extremely high read rate

3. Messages (24 hour cache):
   - 90% of reads are recent messages
   - Older messages: Lazy load from DB"
```

6. Design Google Search ⭐⭐

Where to Cache:

```
✅ Search results (15 min)
✅ Auto-complete (1 hour)
✅ DNS lookups (1 hour)
✅ Crawled content
```

What to Say:

```
"Search engines are cache-heavy:

1. Search Results (15 min TTL):
```

```
   - Top 1000 queries = 60% of all searches
   - Cache these aggressively
   - Personalized per region/language

2. Auto-complete (1 hour):
   - Predictable patterns
   - Sub-millisecond requirement
   - Regional differences

3. DNS Cache:
   - Crawler makes millions of requests
   - 90% hit rate
   - Significant latency reduction"
```

7. Design Uber/Ride-Sharing ⭐⭐

Where to Cache:

```
✅ Driver locations (10 sec TTL)
✅ Surge pricing (2 min)
✅ Map tiles (CDN)
✅ Route calculations (5 min)
✅ User profiles (30 min)
```

What to Say:

```
"Location-based apps need strategic caching:

1. Driver Locations (10 sec TTL):
   - Redis Sorted Set (geohash)
   - Fast nearby driver queries
   - Millions of updates/sec

2. Surge Pricing (2 min):
   - Compute-intensive
   - Cache per region
   - Balance freshness vs load

3. Map Data (CDN):
   - Static tiles cached at edge
   - Same routes requested repeatedly

Profile data: 30 min TTL, 90% DB load reduction"
```

8. Design Rate Limiter ⭐⭐⭐

Where to Cache:

```
✅ Request counters (Redis, window duration)
✅ Rate limit rules (Local, 5 min)
✅ Whitelist/blacklist (Redis, 10 min)
```

What to Say:

```
"Rate limiter IS a caching system:

1. Counter Storage (Redis):
   - Atomic INCR operations
   - Key: user_id:timestamp_window
   - TTL matches rate limit window
   - Sub-millisecond performance

2. Performance Critical:
   - Must add <10ms overhead
   - Every API request passes through
   - Cache enables this requirement

Without Redis: Rate limiting impossible at scale"
```

9. Design News Feed (Reddit/HackerNews) ⭐⭐

Where to Cache:

```
✅ Hot posts/trending (30 min)
✅ Vote counts (1 min)
✅ Comment trees (15 min)
✅ User karma (10 min)
```

What to Say:

```
"News aggregators are cache-heavy:

1. Front Page (30 min TTL):
   - Changes slowly
   - Serve 1M users from cache
   - Regenerate every 30 minutes

2. Vote Counts (1 min):
   - Expensive to calculate in real-time
   - Accept slight staleness
   - Massive performance gain

3. Comments (15 min):
```

```
   - 95% of reads on top 5% threads
   - Pre-build comment tree structure"
```

10. Design Notification System ⭐⭐

Where to Cache:

```
✅ User preferences (1 hour)
✅ Device tokens (30 min)
✅ Templates (1 day)
✅ Unread counts (real-time)
```

What to Say:

```
"Notification systems need caching for scale:

1. User Preferences (1 hour):
   - Read for every notification
   - Rarely change
   - Prevents DB hotspot

2. Device Tokens (30 min):
   - Required for push notifications
   - Cache all active users
   - Millions of tokens"
```

11. Design Dropbox/Google Drive ⭐⭐

Where to Cache:

```
✅ File metadata (15 min)
✅ Directory listings (5 min)
✅ Recently accessed files (CDN)
✅ Sharing permissions (10 min)
```

What to Say:

```
"File storage needs strategic caching:

1. Metadata (15 min TTL):
   - File info queried frequently
   - Expensive DB operations
   - High hit rate
```

```
2. File Content (CDN):
   - Popular files cached at edge
   - Reduces origin server load 90%"
```

12. Design Distributed Cache (Design Redis) ⭐⭐⭐

Key Points:

```
This IS the caching layer for other systems!

Core Features to Design:
✅ In-memory key-value store
✅ LRU eviction policy
✅ TTL support
✅ Replication (master-slave)
✅ Sharding (consistent hashing)
✅ Persistence (optional: RDB, AOF)
✅ High availability

Focus on:
- Data structures (String, Hash, List, Set, Sorted Set)
- Atomic operations
- Pub/Sub for real-time updates
```

13. Design TinyURL/Bitly ⭐⭐⭐

(See #3 - Same as URL Shortener)

14. Design Spotify/Music Streaming ⭐⭐

Where to Cache:

```
✅ Song metadata (1 hour)
✅ Playlists (15 min)
✅ Audio files (CDN, immutable)
✅ User library (5 min)
✅ Recommendations (30 min)
```

What to Say:

```
"Similar to Netflix but with different constraints:

1. Audio Content (CDN):
   - Smaller files than video
   - Cache entire songs at edge
   - Sequential playback patterns
```

```
2. Metadata (1 hour):
   - Artist info, album details
   - Millions of requests per song
   - Redis caching essential

3. Playlists (15 min):
   - User-curated data
   - Frequent access
   - Write-through updates"
```

15. Design Booking System (Airbnb/Hotel) ⭐⭐

Where to Cache:

```
✅ Property listings (1 hour)
✅ Search results (15 min)
✅ Property images (CDN, 1 year)
✅ Availability calendar (30 sec)
✅ Reviews (1 hour)
```

What to Say:

```
"Booking systems have mixed caching needs:

1. Property Listings (1 hour):
   - Details change infrequently
   - High read rate
   - Cache popular properties

2. Availability (30 sec TTL):
   - Changes frequently (bookings)
   - Short TTL required
   - Write-through on booking

3. Search Results (15 min):
   - Popular destinations/dates
   - Expensive queries
   - 70% hit rate"
```

How to Present Caching in Interviews

5-Step Framework

Step 1: Identify the Need (30 seconds)

```
"Looking at requirements:
- 10M reads/day vs 10K writes/day (1000:1 ratio)
- Classic read-heavy workload
- Excellent caching candidate"
```

Step 2: Propose Architecture (1 minute)

```
"Multi-tier caching strategy:
- L1: CDN for static assets (images, CSS)
- L2: Redis for application data (user profiles)
- L3: Local in-memory for config

This balances performance and cost optimally"
```

Step 3: Justify with Numbers (1 minute)

```
"With this strategy:
- Expected hit rate: 90%+
- Latency: 50ms → 5ms (10x faster)
- DB load: Reduced 90%
- Cost: ~70% reduction
- Can serve 10M users with same infrastructure"
```

Step 4: Address Challenges (1 minute)

```
"I'm aware of challenges:
- Cache invalidation: TTL + event-based
- Consistency: 5-min staleness acceptable
- Cache stampede: Lock-based for hot keys
- Monitoring: Hit rate, latency, memory"
```

Step 5: Scale Considerations (30 seconds)

```
"As we scale:
- Redis cluster for horizontal scaling
- Multi-region: Regional caches
- Monitoring: Alert if hit rate <80%"
```

Cache Technologies

Redis vs Memcached

Feature	Redis	Memcached
Data Types	5+ types	String only
Persistence	Yes (RDB/AOF)	No
Replication	Yes	No
Transactions	Yes	No
Pub/Sub	Yes	No
Performance	Excellent	Slightly faster
Recommend	Yes ⭐	Rarely

```
Use Redis: More features, minimal performance difference
Use Memcached: Only for pure key-value at extreme scale
```

Redis Data Structures

```
String:  Simple KV
  Use: User profiles, product details

Hash:    Structured data
  Use: User objects with fields

List:    Ordered collections
  Use: Recent activity, message queues

Set:     Unique items
  Use: Tags, followers, online users

Sorted Set: Rankings
  Use: Leaderboards, priority queues
```

Design Trade-offs

1. Cache Size vs Hit Rate

```
Diminishing Returns:

1GB cache   → 70% hit rate
2GB cache   → 85% hit rate (+15%)
4GB cache   → 92% hit rate (+7%)
8GB cache   → 95% hit rate (+3%)
16GB cache  → 97% hit rate (+2%)
```

```
Sweet spot: Usually 2-4GB
```

2. TTL Selection

```
Trade-off: Freshness vs Performance

Short TTL (1 min):
  ✓ Fresh data
  ✗ More cache misses
  ✗ Higher DB load

Long TTL (1 hour):
  ✓ High hit rate
  ✓ Low DB load
  ✗ Stale data

Recommendation: Start with longer TTL, reduce if needed
```

3. Consistency vs Availability (CAP)

```
Strong Consistency:
  - Write-through caching
  - Synchronous invalidation
  - Higher latency
  Use for: Critical data

Eventual Consistency:
  - Write-back caching
  - Asynchronous invalidation
  - Better performance
  Use for: Non-critical data
```

Common Patterns

1. Feed/Timeline Pattern

```
Architecture:
  CDN → Redis (timelines) → Database

Strategy: Cache-aside
TTL: 5-15 minutes
Invalidation: Event-based
Use Case: Twitter, Instagram, LinkedIn
```

2. E-commerce Pattern

```
Architecture:
  CDN (images) → Redis (catalog, cart) → Database

Products: 1 hour TTL
Inventory: 30 sec TTL (write-through)
Cart: Session TTL (write-through)
Use Case: Amazon, Shopify
```

3. Leaderboard Pattern

```
Architecture:
  Redis Sorted Sets

No TTL (persistent)
Atomic updates (ZINCRBY)
Fast rank queries (O(log N))
Use Case: Gaming, competitions
```

4. Session Pattern

```
Architecture:
  Redis with TTL = session duration

Write-through updates
Survives server restarts
Use Case: Web applications
```

Interview Checklist

When Discussing Caching, Cover:

```
✓ Identify cacheable data patterns
✓ Choose appropriate cache layer(s)
✓ Select caching strategy (cache-aside, write-through, etc.)
✓ Define TTL values with justification
✓ Explain eviction policy (usually LRU)
✓ Address cache invalidation approach
✓ Handle cache misses gracefully
✓ Discuss cache stampede mitigation
✓ Mention consistency trade-offs
```

```
✓ Estimate cache hit rates (with justification)
✓ Calculate performance improvements (with numbers)
✓ Consider monitoring & alerting
✓ Plan for cache failures (graceful degradation)
```

Common Mistakes to Avoid

```
❌ Suggesting caching for everything
❌ Not justifying WHY you're caching
❌ Ignoring invalidation strategy
❌ Not mentioning TTL values
❌ Forgetting about cache stampede
❌ No numbers/estimates for hit rates
❌ Not considering consistency requirements
❌ Caching personalized data
❌ Ignoring monitoring
❌ Treating cache as requirement (not optimization)
```

Summary & Golden Rules

Golden Rules for HLD Interviews

1. Cache What's Expensive

```
✓ Database queries (especially joins)
✓ External API calls
✓ Complex computations (ML models)
✓ Image/video transformations
```

2. Start Simple, Then Optimize

```
Phase 1: Cache-aside + LRU + TTL
Phase 2: Add event-based invalidation
Phase 3: Multi-tier caching
Phase 4: Distributed cache cluster
```

3. Always Justify with Numbers

```
❌ "We should cache this"
✓ "Caching reduces latency from 50ms to 1ms (50x faster)
   and enables 90% DB load reduction at 95% hit rate"
```

4. Address the Hard Parts

```
Must mention:
- Invalidation strategy
- Consistency trade-offs
- Cache stampede handling
- Monitoring approach
```

5. Tie to Business Value

```
Don't just say "faster"
Say: "3ms response improves conversion by 20%
     and reduces infrastructure cost by 70%"
```

Quick Reference Card

```
┌─────────────────────────────────────────────────────────┐
│           CACHING DECISION TREE                         │
└─────────────────────────────────────────────────────────┘

Is it read-heavy (>10:1)?
  ├─ Yes → Strong caching candidate
  └─ No → Consider other optimizations

Is data expensive to generate/fetch?
  ├─ Yes → Cache it
  └─ No → Maybe skip

Does 80/20 rule apply (hot data)?
  ├─ Yes → Cache the 20%
  └─ No → Full dataset caching risky

How frequently does it change?
  ├─ Rarely (hours/days) → Long TTL, high hit rate
  ├─ Moderately (minutes) → Medium TTL, cache-aside
  ├─ Frequently (seconds) → Short TTL, careful
  └─ Constantly (<1s) → Don't cache

What consistency is needed?
  ├─ Strong → Write-through or don't cache
  ├─ Eventual → Cache-aside with TTL
  └─ Relaxed → Any strategy works

What's the performance requirement?
  ├─ <100ms → Multi-tier caching essential
  ├─ <500ms → Single-tier sufficient
  └─ >1s → Caching nice-to-have
```

```
Decision: Cache if 3+ "Yes" answers above
```

Interview Response Templates

When Interviewer Asks: "How would you improve performance?"

```
"I would implement caching because:

1. Workload Analysis:
   - X million reads vs Y thousand writes (ratio: Z:1)
   - Read-heavy pattern perfect for caching

2. Architecture:
   - [Specific cache layers for this system]
   - [TTL values with justification]

3. Expected Impact:
   - Latency: [before]ms → [after]ms ([X]x faster)
   - Hit rate: [X]%+
   - DB load: [X]% reduction

4. Challenges Addressed:
   - Invalidation: [strategy]
   - Stampede: [solution]
   - Monitoring: [metrics]"
```

When Interviewer Asks: "What about consistency?"

```
"For this use case:

1. Consistency Requirement:
   - [Assess: Strong vs Eventual]

2. If Eventual OK:
   - Use cache-aside with TTL
   - [X] minute staleness acceptable
   - Trade: Performance > Strict consistency

3. If Strong Required:
   - Use write-through OR
   - Skip caching for critical paths
   - Maintain ACID guarantees"
```

When Interviewer Asks: "How do you handle cache failures?"

```
```

```
"Graceful degradation strategy:

1. Cache as Enhancement:
   - System works without cache (just slower)
   - Cache failure → Fallback to DB

2. Circuit Breaker:
   - Detect cache unavailability
   - Bypass cache temporarily
   - Resume when healthy

3. Monitoring:
   - Alert on cache failures
   - Track fallback rate
   - Auto-recovery"
```

Real-World Scale Examples

Facebook TAO (Cache)

```
Scale:
- Billions of objects
- Trillions of associations
- Millions of QPS

Architecture:
- Geographically distributed
- Multi-tier caching
- Write-through to MySQL

Hit Rate: 96%+
```

Netflix EVCache

```
Scale:
- 1 trillion requests/day
- Terabytes of cached data
- Millions of subscribers

Architecture:
- Memcached-based
- Regional clusters
- Multi-zone replication

Hit Rate: 98%+
```

Twitter Cache

```
Scale:
- 400M tweets/day
- 6K tweets/second
- 300K requests/second

Architecture:
- FlockDB (social graph cache)
- Manhattan (key-value)
- Multiple tiers

Hit Rate: 95%+
```

Final Tips for HLD Interviews

DO's

```
✓ Mention caching early in design
✓ Provide specific TTL values
✓ Quantify expected improvements
✓ Discuss trade-offs explicitly
✓ Address invalidation strategy
✓ Consider cache stampede
✓ Mention monitoring approach
✓ Reference real systems (Netflix, Facebook)
✓ Draw cache layers in diagram
✓ Explain "why" not just "what"
```

DON'Ts

```
❌ Cache everything indiscriminately
❌ Ignore invalidation
❌ Forget to mention TTL
❌ Overlook consistency trade-offs
❌ Skip performance calculations
❌ Ignore failure scenarios
❌ Use vague statements without numbers
❌ Get into LLD code details
❌ Forget about monitoring
❌ Assume interviewer knows caching well
```

Power Phrases

```
"I would implement caching here because..."
"This is a read-heavy workload with [X]:1 ratio..."
"Following the 80/20 principle..."
"With [X]% expected hit rate, we reduce latency from [Y]ms to [Z]ms..."
"We'll use [strategy] caching with [TTL] TTL because..."
"For invalidation, I'd use [approach] because..."
"This improves performance by [X]x while reducing costs by [Y]%..."
```

Document Version: 2.0 (HLD Focus)
Last Updated: 2024
Optimized For: System Design Interviews - HLD Focus, No Code
Covers: 15 Top HLD Questions with Caching Scenarios