Rate Limiter Architecture - Educational Deep Dive

> **Learning Guide**: This document explains WHY we make each architectural decision, not just WHAT the decision is. Perfect for understanding distributed systems design.

Table of Contents

Why Rate Limiting Exists

The Problem We're Solving

Imagine you run a pizza delivery service:

- You have 10 delivery drivers
- Each driver can deliver 1 pizza per hour
- Your capacity: 10 pizzas/hour

What happens if 100 people order at once?

- ❌ Accept all orders → Customers wait 10 hours → Everyone angry
- ✅ Accept 10 orders, tell others "come back in 1 hour" → Clear expectations

This is rate limiting: Managing demand to match capacity.

Why Digital Systems Need It

Problem 1: Resource Exhaustion

```
Without Rate Limiting:
User A sends 10,000 requests/second
→ Server CPU: 100%
→ Database: Overloaded
→ All other users: Can't access service

With Rate Limiting:
User A limited to 100 requests/second
→ Server CPU: 30%
→ Database: Healthy
→ Everyone gets service
```

Problem 2: Abuse Prevention

```
Attacker sends 1,000,000 requests
→ Without rate limiting: $10,000 cloud bill
→ With rate limiting: Blocked after 1,000 requests
```

Problem 3: Fair Resource Allocation

```
1000 users, each needs 10 req/min = 10,000 req/min total

Without limiting:
- User A (greedy): Uses 5,000 req/min
- Other 999 users: Share remaining 5,000 req/min
- Result: Unfair

With limiting:
- Each user: Max 10 req/min
- Result: Fair for everyone
```

Architectural Decision Breakdown

Decision 1: Stateless vs Stateful API Gateway

What Does "Stateless" Mean?

Stateful (Bad for our use case):

```
Request 1 → Server A (remembers: "user_123 made 5 requests")
Request 2 → Server B (doesn't know about those 5 requests)
Problem: Inconsistent rate limiting
```

Stateless (What we choose):

```
Request 1 → Server A (checks Redis: "user_123 has 5 requests")
Request 2 → Server B (checks Redis: "user_123 has 5 requests")
Result: Both servers see the same data
```

Why Stateless?

Reason 1: Horizontal Scaling

```
With State (in server memory):
- Add Server C → Needs to sync state with A and B
- Complex synchronization
- Slow

Without State (in Redis):
- Add Server C → Just connects to Redis
- No synchronization needed
- Fast
```

Reason 2: Server Failures

```
Stateful:
Server A crashes → Lost all rate limit data
→ Users who were near limit can now spam

Stateless:
Server A crashes → Redis still has all data
→ Server B takes over → Consistent limits maintained
```

Reason 3: Deployment

```
Stateful:
Deploy new version:
1. Drain connections (wait for requests to finish)
2. Sync state to new server
3. Switch traffic
Time: 5-10 minutes

Stateless:
Deploy new version:
1. Start new server
2. Connect to Redis
3. Switch traffic
Time: 30 seconds
```

The Trade-off

What we gain: Scalability, reliability, simple operations
What we lose: Network latency to Redis (~0.5ms)
Why acceptable: 0.5ms is negligible compared to typical API response time (100-500ms)

Decision 2: Redis vs Other Storage Options

Let's compare all realistic options:

Option A: In-Memory (HashMap in application)

```
// In each API server
const counters = new Map();
counters.set("user_123", 45);
```

Why NOT this?

```
Problem 1: Not shared across servers
Server A: user_123 has 45 requests
Server B: user_123 has 0 requests (doesn't know about A)
Result: User can make 100 requests per server instead of 100 total

Problem 2: Lost on restart
Server restarts → counters = new Map() → All data gone
User who hit limit can now spam again

Problem 3: Can't scale memory independently
Need more memory? Must scale entire server (expensive)
```

When WOULD this work?

- Single server applications
- Testing/development
- As a fallback when Redis is down

Option B: PostgreSQL

```
CREATE TABLE rate_limits (
    user_id VARCHAR,
    endpoint VARCHAR,
    count INTEGER,
    window_start TIMESTAMP
);

UPDATE rate_limits SET count = count + 1
WHERE user_id = 'user_123';
```

Why NOT this?

```
Performance Benchmark:
PostgreSQL UPDATE: ~10ms
Redis INCR: ~0.5ms
Difference: 20x slower!
```

```
At 10,000 RPS:
PostgreSQL: 10ms × 10,000 = 100 seconds of DB time per second
→ Need 100 database servers

Redis: 0.5ms × 10,000 = 5 seconds of Redis time per second
→ Need 5 Redis servers

Cost difference: 20x
```

When WOULD this work?

- Low traffic (<100 RPS)
- Rate limit rules storage (changes infrequently)
- Audit logs

Option C: DynamoDB

```
await dynamodb.updateItem({
    Key: { userId: 'user_123' },
    UpdateExpression: 'ADD request_count :inc',
    ExpressionAttributeValues: { ':inc': 1 }
});
```

Why NOT this?

```
Latency:
DynamoDB: 5-10ms (cross-AZ)
Redis: 0.5ms (same AZ)
Difference: 10-20x slower

Cost at 10K RPS:
DynamoDB: 10,000 × $0.25 per million = $2.50/million requests
Redis: Fixed monthly cost ~$1000 regardless of requests

For 1 billion requests/month:
DynamoDB: $2,500
Redis: $1,000
```

When WOULD this work?

- Multi-region with DynamoDB Global Tables
- Serverless architecture (no servers to manage)
- You're already using DynamoDB heavily

Option D: Redis (Our Choice)

```
INCR ratelimit:user_123:1640995200
# Returns: 46 (new count)
# Atomic, fast (0.5ms), battle-tested
```

Why Redis?

Performance: Sub-millisecond latency

```
Operation benchmarks:
GET: 0.3ms
SET: 0.3ms
INCR: 0.3ms
EVAL (Lua script): 0.5ms

Fast enough for 10,000+ RPS per Redis node
```

Atomic Operations: No race conditions

```
Without atomicity:
Thread 1: GET count → 99
Thread 2: GET count → 99
Thread 1: SET count → 100
Thread 2: SET count → 100
Result: Lost update! Should be 101

With Redis INCR:
Thread 1: INCR → 100
Thread 2: INCR → 101
Result: Correct!
```

Proven at Scale:

- Twitter: 1M+ RPS
- GitHub: 100K+ RPS
- Stripe: 100K+ RPS
- Pinterest: 1M+ RPS

Built-in Features:

```
EXPIRE key 3600  # Auto-cleanup old counters
Lua scripts      # Complex atomic operations
Cluster mode     # Horizontal scaling
Pub/Sub          # Real-time notifications
Replication      # High availability
```

Summary Decision Matrix

Requirement	In-Memory	PostgreSQL	DynamoDB	Redis
Latency	✅ 0.1ms	❌ 10ms	⚠ 5ms	✅ 0.5ms
Distributed	❌ No	✅ Yes	✅ Yes	✅ Yes
Atomic Ops	⚠ Local	✅ Yes	✅ Yes	✅ Yes
Cost @10K RPS	✅ $0	❌ High	⚠ Medium	✅ Low
Ops Complexity	✅ Simple	⚠ Medium	✅ Simple	⚠ Medium
Persistence	❌ No	✅ Yes	✅ Yes	✅ Yes

Winner: Redis - Best balance of performance, features, and cost

---

Decision 3: Why Separate Config Store (PostgreSQL)?

You might ask: "If Redis is so good, why not store rate limit rules in Redis too?"

Great question! Let's explore:

What Are "Rate Limit Rules"?

```json
{
  "rule_id": "rule_123",
  "user_tier": "premium",
  "endpoint": "/api/payments",
  "limit": 10000,
  "window": 3600,
  "created_at": "2024-01-01",
  "created_by": "admin@company.com"
}
```

This is **configuration data** - very different from **counter data**.

Configuration vs Counter Data

Counter Data (Redis):

```
Characteristics:
- Changes constantly (every request)
- Needs to be FAST (sub-millisecond)
- Temporary (old counters can be deleted)
- Simple structure (key → number)
- No complex queries needed
```

```
Example:
ratelimit:user_123:1640995200 → 45
(Gets incremented thousands of times per second)
```

Configuration Data (PostgreSQL):

```
Characteristics:
- Changes rarely (once per week/month)
- Can be slightly slower (10ms acceptable)
- Permanent (need history, audit trail)
- Complex structure (foreign keys, joins)
- Need complex queries (search, filter, aggregate)

Example:
SELECT * FROM rules
WHERE user_tier = 'premium'
  AND endpoint LIKE '/api/%'
  AND enabled = true
ORDER BY priority;
```

Why PostgreSQL for Config?

Reason 1: Complex Queries

```
-- Find all rules that might apply to a request
SELECT * FROM rules r
JOIN rule_selectors rs ON r.rule_id = rs.rule_id
WHERE (rs.selector_type = 'user_tier' AND rs.selector_value = 'premium')
   OR (rs.selector_type = 'endpoint' AND rs.selector_value =
'/api/payments')
ORDER BY r.priority
LIMIT 10;

In Redis? Would need to:
1. Get all rules (expensive)
2. Filter in application code (slow)
3. Complex to maintain
```

Reason 2: Audit Trail

```
-- Who changed this rule and when?
SELECT * FROM audit_log
WHERE rule_id = 'rule_123'
ORDER BY created_at DESC;

-- Track history of changes
```

```
SELECT
  created_at,
  created_by,
  old_value,
  new_value
FROM rule_changes
WHERE rule_id = 'rule_123';
```

Redis is not designed for this type of querying.

Reason 3: Data Relationships

```
Rules
  ├─ Selectors (many-to-one)
  ├─ Audit Logs (many-to-one)
  └─ User Tiers (many-to-one)

These foreign key relationships are natural in SQL,
but awkward in Redis.
```

Reason 4: Backup and Recovery

```
PostgreSQL:
- Point-in-time recovery
- SQL dumps (human-readable)
- Well-understood backup strategies

Redis:
- RDB snapshots (binary)
- AOF logs (for counters, not config)
- Less ideal for long-term config storage
```

The Caching Strategy

Best of both worlds:

```
User request comes in
    ↓
Check L1 Cache (in-memory): 0.1ms
    ↓ (if miss)
Check L2 Cache (Redis): 0.5ms
    ↓ (if miss)
Query PostgreSQL: 10ms
    ↓
Cache result in L1 and L2
```

```
    ↓
Return to application
```

Cache Hit Rates:

- L1 (in-memory): 95% hit rate
- L2 (Redis): 4.5% hit rate
- L3 (PostgreSQL): 0.5% hit rate

Effective Latency:

```
Average = (0.95 × 0.1ms) + (0.045 × 0.5ms) + (0.005 × 10ms)
        = 0.095ms + 0.023ms + 0.05ms
        = 0.168ms

Almost as fast as pure Redis!
```

Decision 4: Multi-Tier Caching Strategy

Let's understand WHY we have 3 cache levels.

The Problem: Latency vs Accuracy

```
Scenario: API Gateway checks rate limit rules

Option A: Always query PostgreSQL
→ Accurate (always latest data)
→ Slow (10ms per request)
→ At 10K RPS: 100 seconds of DB time per second
→ Need 100 database connections

Option B: Cache in memory forever
→ Fast (0.1ms)
→ Inaccurate (stale data for hours)
→ Rule changes not reflected

Option C: Multi-tier caching (our choice)
→ Fast (0.1ms for 95% of requests)
→ Reasonably accurate (5min staleness acceptable)
→ Scalable (minimal DB load)
```

Understanding Each Tier

L1 Cache: In-Memory (Application)

```
// Inside each API Gateway server
const rulesCache = new Map();
const CACHE_TTL = 5 * 60 * 1000; // 5 minutes

function getCachedRule(key) {
  const cached = rulesCache.get(key);
  if (cached && Date.now() - cached.timestamp < CACHE_TTL) {
    return cached.rule; // Super fast: 0.1ms
  }
  return null; // Cache miss
}
```

Why 5 minutes TTL?

```
Too short (1 minute):
- More cache misses
- More load on L2/L3
- Minimal accuracy gain

Too long (1 hour):
- Rule changes take too long to propagate
- Users confused: "I updated the rule but it's not working"

5 minutes sweet spot:
- High hit rate (95%)
- Acceptable staleness for rate limiting
- Rule changes visible within 5 minutes
```

L2 Cache: Redis

```
async function getRuleFromRedis(key) {
  const rule = await redis.get(`rule:${key}`);
  if (rule) {
    // Cache in L1 for next time
    rulesCache.set(key, {
      rule: JSON.parse(rule),
      timestamp: Date.now()
    });
    return JSON.parse(rule);
  }
  return null;
}
```

Why Redis as L2?

- Shared across all API servers (no duplication)

- Faster than PostgreSQL (0.5ms vs 10ms)
- Reduces database load by 95%

L3 Source: PostgreSQL

```
async function getRuleFromDB(key) {
  const rule = await db.query(
    'SELECT * FROM rules WHERE key = $1',
    [key]
  );

  if (rule) {
    // Cache in L2
    await redis.setex(
      `rule:${key}`,
      900, // 15 minutes
      JSON.stringify(rule)
    );

    // Cache in L1
    rulesCache.set(key, {
      rule: rule,
      timestamp: Date.now()
    });
  }

  return rule;
}
```

Cache Invalidation Strategy

The famous computer science joke: "There are only two hard things in Computer Science: cache invalidation and naming things."

Let's tackle cache invalidation:

Option 1: TTL (Time To Live) - What we use

```
Rule updated in PostgreSQL
→ L1 cache: Stale for up to 5 minutes
→ L2 cache: Stale for up to 15 minutes
→ Eventually consistent

Pros:
✅ Simple to implement
✅ No additional infrastructure
✅ Works even if notification system fails

Cons:
```

```
❌ Delayed propagation (5-15 minutes)
❌ Temporary inconsistency between servers
```

Option 2: Active Invalidation

```
Rule updated in PostgreSQL
    ↓
Publish message to Redis Pub/Sub
    ↓
All API servers receive message
    ↓
Each server clears its L1 cache
    ↓
Next request fetches fresh data

Pros:
✅ Immediate propagation
✅ Consistent across servers

Cons:
❌ More complex
❌ Requires Pub/Sub infrastructure
❌ All servers must be subscribed
❌ What if a server is restarting?
```

Our Decision: TTL for Now

Why?

1. **Simplicity**: Less moving parts
2. **Acceptable delay**: 5-15 minutes fine for rate limiting
3. **Self-healing**: Old caches automatically expire
4. **Failure tolerant**: No dependencies on message buses

When to upgrade to Active Invalidation?

- If 5-minute delay becomes problematic
- If you already have Pub/Sub infrastructure
- If you need real-time rule updates

Algorithm Selection Explained

Let's understand WHY we choose each algorithm through real-world analogies.

Fixed Window Counter

Real-World Analogy: Gym membership

```
Gym Rules:
- You can visit 10 times per month
- Month = Calendar month (Jan 1 - Jan 31)

Scenario:
January 31, 11:00 PM: Visit 10 times ✓
February 1, 12:01 AM: Counter resets!
February 1, 12:01 AM: Visit 10 MORE times ✓

Result: 20 visits in 2 hours (instead of 10 per month)
```

This is the **boundary problem**.

In Code:

```javascript
function fixedWindowCheck(user, now) {
  const windowStart = Math.floor(now / WINDOW) * WINDOW;
  const key = `${user}:${windowStart}`;
  const count = redis.incr(key);

  if (count === 1) {
    redis.expire(key, WINDOW);
  }

  return count <= LIMIT;
}

// Problem visualization:
// Window 1: [00:00:00 - 00:59:59]
// Window 2: [01:00:00 - 01:59:59]
//
// At 00:59:30: 100 requests (allowed, Window 1)
// At 01:00:01: 100 requests (allowed, Window 2, NEW window!)
// Result: 200 requests in 31 seconds!
```

When to use Fixed Window:

- ✅ MVP / Prototype
- ✅ Low traffic (<100 RPS)
- ✅ Simplicity is critical
- ❌ **NOT for production** if accuracy matters

Sliding Window Log

Real-World Analogy: Movie theater ticket checking

```
Rule: Max 100 people in theater at once

Approach: Keep a list of everyone's entry time

Person 101 arrives at 2:30 PM
→ Check list: Remove everyone who entered before 1:30 PM
→ Count remaining: 95 people
→ Allow entry ✓

Advantages:
✓ Exact count always
✓ No boundary problem

Disadvantages:
✗ Must store 100 timestamps (memory intensive)
✗ Must scan and filter list every time (slow)
```

In Code:

```
function slidingWindowLogCheck(user, now) {
  const key = `${user}:requests`;
  const cutoff = now - WINDOW;

  // Remove old timestamps
  redis.zremrangebyscore(key, 0, cutoff);

  // Count remaining
  const count = redis.zcard(key);

  if (count < LIMIT) {
    // Add new timestamp
    redis.zadd(key, now, `${now}:${uuid()}`);
    return true;
  }
  return false;
}

// Memory usage example:
// User makes 1000 req/hour
// Store 1000 timestamps × 24 bytes each = 24 KB per user
// For 1M users = 24 GB just for timestamps!
```

When to use Sliding Window Log:

- ✅ Financial transactions (need exact limits)
- ✅ High-value APIs
- ✅ Low-medium traffic
- ❌ NOT for high traffic (memory intensive)

Sliding Window Counter (Our Choice)

Real-World Analogy: Estimating crowd size

```
Scenario: Counting people at a concert

Exact method (Sliding Window Log):
Count everyone individually
→ Accurate but slow

Estimation method (Sliding Window Counter):
- Previous hour: 8,000 people
- Current hour (30 min in): 3,000 people
- Estimate: 3,000 + (8,000 × 50%) = 7,000 people

Close enough! And much faster.
```

In Code:

```
function slidingWindowCounterCheck(user, now) {
  const currentWindow = Math.floor(now / WINDOW) * WINDOW;
  const previousWindow = currentWindow - WINDOW;

  // Get both counters
  const [currentCount, previousCount] = await redis.mget([
    `${user}:${currentWindow}`,
    `${user}:${previousWindow}`
  ]);

  // Calculate position in current window (0.0 to 1.0)
  const elapsedInCurrent = now - currentWindow;
  const currentPosition = elapsedInCurrent / WINDOW;

  // Weight previous window (decreases as we move through current)
  const previousWeight = 1.0 - currentPosition;

  // Estimate total count
  const estimated = currentCount + (previousCount * previousWeight);

  if (estimated < LIMIT) {
    redis.incr(`${user}:${currentWindow}`);
    redis.expire(`${user}:${currentWindow}`, WINDOW * 2);
    return true;
  }
  return false;
}
```

Why This Works:

```
Visualization:
Previous Window: [════════════════] 80 requests
Current Window:  [════════        ] 30 requests (50% complete)

Time 0% (window start):
Estimated = 30 + (80 × 100%) = 30 + 80 = 110

Time 25%:
Estimated = 30 + (80 × 75%) = 30 + 60 = 90

Time 50%:
Estimated = 30 + (80 × 50%) = 30 + 40 = 70

Time 75%:
Estimated = 30 + (80 × 25%) = 30 + 20 = 50

Time 100% (window end):
Estimated = 30 + (80 × 0%) = 30 + 0 = 30

The previous window's influence GRADUALLY decreases!
```

Accuracy Analysis:

```
Worst Case Error:
- User makes 100 requests at end of Window 1
- User makes 100 requests at start of Window 2
- At 1% into Window 2:
  Estimated = 100 + (100 × 99%) = 199
  Actual = 200
  Error = 0.5% ✓

Best Case:
- Even distribution
- Error ≈ 0%

Typical Case:
- Most users don't game the system
- Error ≈ 1-3%
- Well within our 5% tolerance
```

Memory Efficiency:

```
Fixed Window: 1 counter per window
Sliding Window Counter: 2 counters per user
Sliding Window Log: N timestamps (N = request count)
```

```
Example with 1000 req/hour:
Fixed Window: 8 bytes
Sliding Window Counter: 16 bytes (2 counters)
Sliding Window Log: 24,000 bytes (1000 timestamps)

1500x more memory efficient than log!
```

Why This is Our Default Choice:

Requirement	Score	Notes
Accuracy	⭐⭐⭐⭐	95-99%, good enough
Memory	⭐⭐⭐⭐⭐	Only 2 counters
Speed	⭐⭐⭐⭐⭐	2 Redis GETs + calculation
Distributed	⭐⭐⭐⭐⭐	Works perfectly
Complexity	⭐⭐⭐⭐	Moderate implementation

Perfect balance for 95% of use cases.

Token Bucket

Real-World Analogy: Subway ticket dispenser

```
Machine holds 10 tickets (capacity)
New ticket appears every 6 seconds (refill rate = 10/minute)

Scenario 1: Rush Hour Burst
- 10 people arrive at once
- Machine: "Take all 10 tickets!" ✓
- Next person: "Sorry, wait 6 seconds"

Scenario 2: Steady Flow
- 1 person per minute
- Machine: Always has tickets available

This allows BURSTS while maintaining average rate.
```

In Code:

```
function tokenBucketCheck(user, now) {
  const state = getTokenBucketState(user);

  // Refill tokens based on elapsed time
  const elapsed = now - state.lastRefill;
  const tokensToAdd = elapsed * REFILL_RATE;
  const newTokens = Math.min(CAPACITY, state.tokens + tokensToAdd);

  if (newTokens >= 1) {
    // Consume 1 token
    saveState(user, {
      tokens: newTokens - 1,
      lastRefill: now
    });
    return true;
  }

  return false;
}
```

Visual Example:

```
Capacity: 10 tokens
Refill: 1 token/second

Time 0s: [●●●●●●●●●●] 10 tokens
5 requests: [●●●●●     ] 5 tokens

Time 1s: [●●●●●●    ] 6 tokens (refilled 1)

5 more requests: [●         ] 1 token

Time 5s: [●●●●●●    ] 6 tokens (refilled 5)

Allows bursts while maintaining average rate!
```

When to use Token Bucket:

- ✅ Mobile apps that sync data periodically
- ✅ Batch jobs
- ✅ APIs where bursts are legitimate (not abuse)
- ✅ Variable request sizes (can consume multiple tokens)

Example: AWS API Gateway

```
Configuration:
- Burst: 5,000 requests
- Steady rate: 10,000 req/second

What this means:
- Can handle sudden spike of 5,000 requests
```

```
  - Then processes 10,000 per second ongoing
  - Perfect for serverless/bursty workloads
```

Distributed Systems Concepts

Concept 1: Atomic Operations

The Problem:

```
Scenario: Two servers, same user

Time T0:
Server A reads: count = 99
Server B reads: count = 99

Time T1:
Server A writes: count = 100
Server B writes: count = 100

Result: Count should be 101, but it's 100!
Lost Update Problem
```

Real-World Analogy: Bank account

```
You have $100 in your account

Withdraw at ATM 1: Read $100, withdraw $50, write $50
Withdraw at ATM 2: Read $100, withdraw $50, write $50

If not atomic:
Both ATMs read $100 simultaneously
Both write $50
You withdrew $100 but account shows $50!
Free $50!

With atomic operations (database transactions):
ATM 1: LOCK, read $100, withdraw $50, write $50, UNLOCK
ATM 2: Wait for lock, read $50, withdraw $50, write $0
Correct!
```

In Redis:

```
# Bad (non-atomic):
count = GET user:123
count = count + 1
```

```
SET user:123 count

# Good (atomic):
INCR user:123  # Single atomic operation

# Even better (complex atomicity):
EVAL "
  local count = redis.call('INCR', KEYS[1])
  if count == 1 then
    redis.call('EXPIRE', KEYS[1], ARGV[1])
  end
  if count > tonumber(ARGV[2]) then
    redis.call('DECR', KEYS[1])
    return 0
  end
  return 1
" 1 user:123 3600 100
```

Why Lua Scripts?

```
Multiple Redis commands together, atomically:

Without Lua (NOT atomic):
1. INCR user:123           → 101
2. GET user:123            → 101
3. IF 101 > 100:
4.   DECR user:123         → 100

Problem: Another server could INCR between steps 2 and 4

With Lua (Atomic):
All steps execute as one atomic block
No other operations can interleave
```

Concept 2: Consistency vs Availability (CAP Theorem)

CAP Theorem: In a distributed system, you can only have 2 of 3:

- **C**onsistency: Everyone sees the same data
- **A**vailability: System always responds
- **P**artition Tolerance: Works despite network failures

Real-World Analogy: Conference call

```
Scenario: 3 people on video call decide on lunch

Perfect World (All 3):
- Everyone hears everyone (Consistency)
- Everyone can speak (Availability)
- Works even if connection drops (Partition Tolerance)
Impossible!

Reality - Choose 2:

Option CP (Consistency + Partition Tolerance):
If connection drops, meeting paused until fixed
Everyone hears same thing, but might wait

Option AP (Availability + Partition Tolerance):
If connection drops, groups decide separately
Everyone can speak, but might differ

Option CA (Consistency + Availability):
Perfect in same room, breaks if rooms separate
Not realistic for distributed systems
```

Applied to Rate Limiting:

```
CP System (Consistency Priority):
- Strict 100 req/hour limit
- Never exceeds 100
- If Redis unreachable → BLOCK ALL REQUESTS
- 99.9% availability (some downtime)

AP System (Availability Priority):
- ~100 req/hour limit (best effort)
- Might allow 105-110 during issues
- If Redis unreachable → ALLOW ALL REQUESTS
- 99.99% availability (always responds)

Our Choice: AP (Availability)
```

Why Choose Availability?

Let's think through the business impact:

```
Scenario: Redis goes down for 30 seconds

With CP (Consistency Priority):
→ Block all API requests for 30 seconds
→ Revenue loss: $10,000 (if processing $20M/day)
→ User experience: Complete outage
→ Customer support: Flooded with complaints

With AP (Availability Priority):
→ Allow all requests for 30 seconds
→ Some users exceed limits (maybe 105-110 requests instead of 100)
→ Revenue: Maintained
→ User experience: Normal
→ Cost: Minimal (few extra backend calls)

Business Decision: Lose $10,000 or allow 5% over-limit?
Clear choice: AP (Availability)
```

When WOULD you choose CP?

```
Financial Trading System:
- Strict account limits (can't go negative)
- Regulatory requirements (must never exceed)
- Better to deny service than allow over-limit

Payment Processing:
- Daily transaction limits (legal requirements)
- Can't exceed without compliance issues
- Downtime acceptable over violations
```

For rate limiting APIs? **Almost always choose AP (Availability)**.

Concept 3: Sharding and Partitioning

Why Do We Need Sharding?

```
Problem: Single Redis Instance Limits

Single Redis:
- Max throughput: ~100K ops/second
- Max memory: 256 GB
- Single point of failure

Our needs:
- Throughput: 200K+ ops/second
- Can't fit all counters in 256 GB
- Need high availability

Solution: Split data across multiple Redis instances (sharding)
```

Real-World Analogy: Library organization

```
Bad: All books in one room
- Hard to find books
- Slow checkout (everyone waiting)
- If room floods, lose everything
```

```
Good: Books organized by subject across multiple rooms
- Fiction: Room A
- Science: Room B
- History: Room C
- Fast to find (go to right room)
- If one room floods, others safe
```

In Rate Limiting:

```
Shard by User ID:

User user_123 → hash(user_123) → 42 → 42 % 3 = 0 → Shard 0
User user_456 → hash(user_456) → 87 → 87 % 3 = 0 → Shard 0
User user_789 → hash(user_789) → 14 → 14 % 3 = 2 → Shard 2

Result:
- Users evenly distributed
- Each shard handles ~33% of traffic
- Can scale by adding more shards
```

Consistent Hashing Benefits:

```
Regular Hashing Problem:
3 shards: hash(user) % 3
Add 4th shard: hash(user) % 4
→ ALL keys move to different shards!
→ Cache invalidation storm
→ Massive performance hit

Consistent Hashing:
3 shards: hash(user) → ring position
Add 4th shard: Only ~25% of keys move
→ Minimal disruption
→ Smooth scaling
```

Concept 4: Replication for High Availability

Why Replication?

```
Single Redis Master:

Problem 1: Crash
Master crashes → All data lost → Downtime

Problem 2: Maintenance
```

```
Need to upgrade → Must stop service → Downtime

Problem 3: Read Load
1 master handling 100K reads/second → Overloaded
```

Master-Replica Architecture:

```
        ┌─────────────┐
        │   Master    │ ← Writes go here
        │  (Shard 1)  │
        └──────┬──────┘
               │ Replication
        ┌──────┴─────┐
        │            │
   ┌────▼────┐  ┌────▼────┐
   │Replica A│  │Replica B│ ← Reads can go here
   └─────────┘  └─────────┘
```

How Replication Helps:

```
Problem 1 (Crash) - Solved:
Master crashes
→ Sentinel detects (3 seconds)
→ Promotes Replica A to Master (2 seconds)
→ Downtime: 5 seconds (vs hours)

Problem 2 (Maintenance) - Solved:
Upgrade Replica A → Make it Master → Upgrade old Master
→ Zero downtime

Problem 3 (Read Load) - Solved:
Split reads:
- Writes: 20K/sec → Master
- Reads: 80K/sec → 40K/sec per Replica
- Master not overloaded!
```

Replication Lag Consideration:

```
Master writes: count = 100
→ Replica syncs after 10ms
→ Replica still shows: count = 99

If we read from replica:
- Might allow 101st request
- Slight over-limit
```

```
Trade-off:
- For counters: Usually write to master, read from master (consistency)
- For rules: Can read from replica (eventual consistency okay)
```

Scaling Patterns Explained

Vertical vs Horizontal Scaling

Vertical Scaling (Scale Up):

```
Current: 4 CPU, 16 GB RAM server
Upgrade: 8 CPU, 32 GB RAM server

Analogy: Bigger truck
- One driver, bigger truck
- Easier to coordinate
- But limited by truck size

Limits:
- Eventually hit hardware limits
- Expensive at high end
- Single point of failure
```

Horizontal Scaling (Scale Out):

```
Current: 1 server
Add: 9 more identical servers

Analogy: More trucks
- Many drivers, regular trucks
- Harder to coordinate
- But nearly unlimited capacity

Benefits:
- Can add capacity indefinitely
- Cheaper (commodity hardware)
- No single point of failure
```

For Rate Limiting:

```
We use BOTH:

Vertical (Redis):
- Start: 2 GB Redis
- Grow: 16 GB → 64 GB Redis
- Limit: Stop at 256 GB (Redis limit)

Horizontal (API Gateways):
- Start: 3 servers
- Grow: 10 → 50 → 100 servers
- Limit: Nearly unlimited

Why this combination?
- Redis: Needs to be powerful (centralized state)
- API Gateways: Can be numerous (stateless)
```

Auto-Scaling Explained

Why Auto-Scaling?

```
Traffic Pattern:
Morning (8 AM): 1,000 RPS
Afternoon (2 PM): 5,000 RPS
Evening (8 PM): 10,000 RPS
Night (2 AM): 200 RPS

Without Auto-Scaling:
- Provision for 10,000 RPS (peak)
- Servers: 10 instances running 24/7
- Utilization at 2 AM: 2% (waste)
- Cost: $7,200/month

With Auto-Scaling:
- 2 AM: 1 instance
- 8 AM: 2 instances
- 2 PM: 5 instances
- 8 PM: 10 instances
- Average: 4.5 instances
- Cost: $3,240/month (55% savings!)
```

How Auto-Scaling Works:

```
┌─────────────────────────────────────────┐
│  CloudWatch Metrics                     │
│  - CPU: 85% (threshold: 70%)           │
│  - Latency: P95 = 150ms (threshold: 100ms) │
└─────────────────────────────────────────┘
                 │
                 │ Alert!
                 ▼
┌─────────────────────────────────────────┐
│  Auto Scaling Group                     │
│  - Current: 5 instances                 │
│  - Decision: Scale up needed            │
```

```
└──────────────────────────────────────────┘
                  │
                  │
                  ▼
┌──────────────────────────────────────────┐
│  Launch 3 New Instances                  │
│  - Start servers                         │
│  - Connect to Redis                      │
│  - Register with load balancer           │
│  Time: 60 seconds                        │
└──────────────────────────────────────────┘
```

Configuration Decisions Explained:

Scale Up Threshold: 70% CPU

```
Why not 90%?
- Need buffer for sudden spikes
- Time to start new servers: 60s
- Could hit 100% before new servers ready

Why not 50%?
- Over-provisioning
- Higher costs
- Frequent scaling (thrashing)

70% sweet spot:
- Enough buffer for spikes
- Efficient resource usage
- Stable scaling behavior
```

Scale Down: Wait 5 Minutes

```
Why wait?
- Traffic might spike again
- Prevent thrashing (scale up/down/up/down)
- Terminating servers loses in-flight requests

Why 5 minutes?
- Short enough: Don't waste money
- Long enough: Avoid thrashing
- Typical: 5-15 minutes industry standard
```

Load Balancing Strategies

Algorithm: Least Connections

```
Why "Least Connections" vs "Round Robin"?

Round Robin:
Server A: 100 slow requests (10s each)
Server B: 100 fast requests (0.1s each)
Next request → Server A (it's "next")
→ Server A overloaded, Server B idle

Least Connections:
Server A: 50 active connections
Server B: 10 active connections
Next request → Server B (fewer connections)
→ Better distribution!
```

Why Not "Random"?

```
Random:
- Simple
- But can create hot spots
- Server A might get 3 requests in a row by chance
- Less optimal

Least Connections:
- Slightly more complex
- Better distribution
- Self-balancing
- Worth the complexity
```

Health Checks Explained:

```
Health Check Configuration:
- Interval: 10 seconds
- Timeout: 5 seconds
- Unhealthy threshold: 2 failures
- Healthy threshold: 2 successes

Why these numbers?

Interval = 10s (not 1s):
- 1s: Too frequent, overhead on servers
- 60s: Too slow, slow to detect failures
- 10s: Good balance

Unhealthy threshold = 2 (not 1):
- Single failure might be transient (network blip)
- 2 failures: Likely real problem
- Prevents false positives
```

```
Healthy threshold = 2 (not 1):
- Server recovering might still be unstable
- 2 successes: Likely stable
- Prevents premature routing
```

Scaling Patterns Explained

Pattern 1: Read-Write Splitting

Concept: Separate read and write traffic

```
Problem:
- 80% of operations are reads (GET)
- 20% of operations are writes (INCR)
- Single Redis handles both
- Reads slow down writes

Solution:
- Writes: Go to Master only
- Reads: Go to Replicas
- Master handles 20% of traffic (not overloaded)
- Replicas handle 80% of traffic
```

Applied to Rate Limiting:

```
Counter Operations:
- Write (INCR): Must go to Master (need consistency)
- Read (GET for calculation): Can go to Replica

Rules Operations:
- Write (UPDATE rules): PostgreSQL Primary
- Read (GET rules): PostgreSQL Replica OR cache

Result:
- Master: 2K writes/sec (manageable)
- Replicas: 8K reads/sec (distributed)
- Better performance, same accuracy
```

Pattern 2: Circuit Breaker

Real-World Analogy: Electrical circuit breaker

```
House Circuit:
- Normal: Electricity flows
- Overload: Breaker trips (prevents fire)
- Manual reset after fixing issue

Software Circuit Breaker:
- Normal: Requests to Redis succeed
- Failures: Trip after 5 consecutive failures
- Auto-reset: Try again after 30 seconds
```

Why Circuit Breaker?

```
Without Circuit Breaker:
Redis is down
→ Every request tries Redis
→ Waits for timeout (1 second)
→ 10,000 RPS × 1 second = 10,000 hanging requests
→ Servers run out of memory
→ Cascade failure

With Circuit Breaker:
Redis is down
→ First 5 requests try Redis (fail)
→ Circuit opens (stops trying)
→ Next 9,995 requests: Immediate local decision (fail-open)
→ Servers healthy
→ Try Redis again after 30s
```

Implementation:

```
States:
┌─────────┐   Success   ┌─────────┐
│ CLOSED  │────────────▶│ CLOSED  │
│(Normal) │             │(Normal) │
└─────────┘             └─────────┘
     │
     │ 5 failures
     ▼
┌─────────┐
│  OPEN   │ ← Fast fail all requests
│(Tripped)│
└─────────┘
     │
     │ After 30s
     ▼
┌──────────┐   Success   ┌─────────┐
│HALF-OPEN │────────────▶│ CLOSED  │
│(Testing) │             │         │
└──────────┘             └─────────┘
     │ Failure
     │
```

```
       ▼
┌─────────┐
│  OPEN   │
└─────────┘
```

Pattern 3: Bulkhead Pattern

Real-World Analogy: Ship compartments

```
Old ships (Titanic):
- One big hull
- Hole in one place → Entire ship floods → Sinks

Modern ships:
- Multiple compartments (bulkheads)
- Hole in one compartment → Only that floods → Ship stays afloat
```

Applied to Rate Limiting:

```
Separate Resource Pools:

Without Bulkheads:
- Single Redis handles all rate limits
- Payment API gets attacked (100K RPS)
- Redis overloaded
- ALL APIs slow down (even unrelated ones)

With Bulkheads:
- Redis Pool 1: Critical APIs (payments, auth)
- Redis Pool 2: Regular APIs
- Redis Pool 3: Public/unauthenticated

Attack on public APIs:
→ Only Pool 3 affected
→ Pools 1 & 2 unaffected
→ Critical APIs stay fast
```

Configuration:

```
┌─────────────────────────────────────┐
│  Critical APIs (High Priority)      │
│  - Payments, Authentication         │
│  - Redis Pool: 10 instances         │
│  - Reserved capacity: Never shared  │
└─────────────────────────────────────┘
```

```
┌─────────────────────────────────────┐
│  Regular APIs (Medium Priority)     │
│  - User data, Settings              │
│  - Redis Pool: 5 instances          │
│  - Shared capacity                  │
└─────────────────────────────────────┘


┌─────────────────────────────────────┐
│  Public APIs (Low Priority)         │
│  - Public data, Search              │
│  - Redis Pool: 3 instances          │
│  - Can be throttled aggressively    │
└─────────────────────────────────────┘
```

Multi-Region Architecture Explained

Why Multiple Regions?

Problem: Single region limitations

```
Single Region (US-EAST only):

User in California:
- Request travels: SF → Virginia → SF
- Latency: ~70ms each way = 140ms
- User experience: Slow

User in Europe:
- Request travels: London → Virginia → London
- Latency: ~150ms each way = 300ms
- User experience: Very slow

User in Asia:
- Latency: 400ms+
- User experience: Unacceptable
```

With Multiple Regions:

```
Regions: US-WEST, US-EAST, EU-WEST, ASIA-PACIFIC

User in California → US-WEST → 10ms ✓
User in Europe → EU-WEST → 15ms ✓
User in Asia → ASIA-PACIFIC → 20ms ✓

Everyone gets fast service!
```

Regional Independence vs Global Consistency

This is the **KEY architectural decision** for multi-region.

Option 1: Regional Independence (What We Choose)

```
Architecture:
┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│   US-WEST    │   │   US-EAST    │   │   EU-WEST    │
│              │   │              │   │              │
│ Redis (Local)│   │ Redis (Local)│   │ Redis (Local)│
│              │   │              │   │              │
│ No sync      │   │ No sync      │   │ No sync      │
└──────────────┘   └──────────────┘   └──────────────┘

Rate Limit: 100 requests/hour globally

Strategy: Distribute quota
- US-WEST: 40 requests/hour
- US-EAST: 35 requests/hour
- EU-WEST: 25 requests/hour
Total: 100 requests/hour
```

How Does This Work?

```
Scenario: User in California

Normal Usage:
- User makes 30 requests → US-WEST
- US-WEST quota: 40/hour
- All allowed ✓

Edge Case - User Travels:
- User makes 30 requests → US-WEST (30/40 used)
- User flies to Europe
- User makes 25 requests → EU-WEST (25/25 used)
- Total: 55 requests (over 40+25=65 budget)

Analysis:
- Over-limit by: 0 (still under distributed total)
- If user made 40 in US-WEST + 25 in EU-WEST = 65 total
- This is acceptable (within our 5% tolerance)
```

Why This Works:

```
Statistics:
- 95% of users stay in one region
- 4% switch once
- 1% are truly multi-region

Cost of regional independence:
- 5% might exceed quota
- Acceptable per requirements!

Benefits:
✓ Zero cross-region latency
✓ Regions isolated (failure in US doesn't affect EU)
✓ Simple to implement
✓ Easy to operate

This is the right trade-off!
```

Option 2: Global Synchronization (Not Recommended)

```
Architecture:
┌──────────────┐   ┌──────────────┐   ┌──────────────┐
│   US-WEST    │   │   US-EAST    │   │   EU-WEST    │
│   API GW     │   │   API GW     │   │   API GW     │
└──────┬───────┘   └──────┬───────┘   └──────┬───────┘
       │                  │                  │
       └──────────────────┼──────────────────┘
                          │
                          ▼
                 ┌─────────────────┐
                 │  Central Redis  │
                 │   (US-EAST)     │
                 └─────────────────┘

Every request goes to US-EAST Redis
```

Problems:

```
Latency:
- EU user → EU Gateway → US-EAST Redis
- Network latency: 80-150ms
- Too slow for rate limiting!

Availability:
- US-EAST region fails
- Global outage (all regions down)
- Single point of failure

Cost:
- Cross-region data transfer
- $0.02 per GB
- At 10K RPS: $5,000/month just for data transfer
```

When WOULD this work?

```
Use cases:
- Financial APIs (need strict limits)
- Low traffic (<1K RPS globally)
- High-value transactions
- Latency less critical

Example: Banking API
- 100 transactions/day per account
- Strict regulatory requirement
- Can afford 100ms extra latency
→ Global sync acceptable
```

Option 3: Async Synchronization (Complex)

```
Architecture:
┌──────────────┐     ┌──────────────┐     ┌──────────────┐
│   US-WEST    │     │   US-EAST    │     │   EU-WEST    │
│ Redis (Local)│ ──► │ Redis (Local)│ ◄── │ Redis (Local)│
│              │     │              │     │              │
│ Sync queue   │     │ Sync master  │     │ Sync queue   │
└──────────────┘     └──────────────┘     └──────────────┘
         ↑                  ↑                    ↑
         └─────── Sync every 5 seconds ─────────┘

Each region:
1. Tracks requests locally
2. Every 5s: Send delta to other regions
3. Other regions: Update their view

Example:
T=0s: US-WEST: 10 requests, US-EAST: 5 requests
T=5s: Sync happens
Now everyone knows: Total = 15 requests
```

Problems:

```
Race Condition:
T=0s: US-WEST sends "I have 10 requests"
T=1s: US-EAST sends "I have 5 requests"
T=2s: US-WEST receives "5 from US-EAST" → Total: 15
T=2s: US-EAST receives "10 from US-WEST" → Total: 15
T=3s: User makes 50 requests in US-WEST
T=5s: Sync sends "60 from US-WEST"
```

```
T=5s: US-EAST thinks total = 65
      But US-WEST might have received more from EU-WEST
→ Inconsistencies accumulate

Complex to implement correctly!
```

When WOULD this work?

```
Requirements:
- Need better accuracy than regional independence
- Can't afford global sync latency
- Have experienced distributed systems team
- Can handle eventual consistency bugs

Example: LinkedIn, Facebook
- Massive scale (1M+ RPS)
- Can't use global sync (latency)
- Can afford complex implementation
- Have teams to manage it
```

Our Recommendation: Regional Independence

Why?

```
Priorities (in order):
1. Availability (99.99%) ✓
2. Low latency (<10ms) ✓
3. Simplicity (easy to operate) ✓
4. Accuracy (95%+) ✓

Regional independence achieves all four!

Trade-off: 5% might exceed quota
Is this acceptable? YES, per requirements.
```

Common Mistakes & How to Avoid Them

Mistake 1: Not Using Atomic Operations

The Mistake:

```
// WRONG: Read-modify-write
const count = await redis.get(key);
if (count < limit) {
  await redis.set(key, count + 1);
```

```
    return true;
  }
  return false;
```

Why It's Wrong:

```
Server A and B both read count=99 simultaneously
Both think they can increment
Both write 100
Result: Lost update
```

The Fix:

```
// RIGHT: Use atomic operations
const count = await redis.incr(key);
return count <= limit;
```

Lesson: Always use database/cache atomic operations for counters.

Mistake 2: Ignoring Clock Skew

The Mistake:

```
// Using local server time
const now = Date.now();
const windowStart = Math.floor(now / WINDOW) * WINDOW;
```

Why It's Wrong:

```
Server A clock: 10:00:00 (Window 1)
Server B clock: 10:00:10 (Window 2)

User hits Server A: 100 requests in Window 1
User hits Server B: 100 requests in Window 2 (different window!)
Total: 200 requests in 10 seconds (should be 100/hour)
```

The Fix:

```
// Option 1: Use Redis time (synchronized)
const [seconds, microseconds] = await redis.time();
const now = parseFloat(`${seconds}.${microseconds}`);
```

```
// Option 2: Use NTP + monitor skew
// Option 3: Accept slight inaccuracy (often acceptable)
```

Lesson: In distributed systems, never assume clocks are synchronized.

Mistake 3: Not Planning for Failures

The Mistake:

```
async function checkRateLimit(user) {
  const count = await redis.incr(key);  // What if this fails?
  return count <= limit;
}
```

Why It's Wrong:

```
Redis goes down:
→ Exception thrown
→ No error handling
→ Requests fail with 500 error
→ Bad user experience
```

The Fix:

```
async function checkRateLimit(user) {
  try {
    const count = await redis.incr(key);
    return count <= limit;
  } catch (error) {
    // Log error
    logger.error('Redis error:', error);

    // Metric
    metrics.increment('rate_limiter.redis_errors');

    // Fail-open (allow request)
    return true;
  }
}
```

Decision Tree for Failures:

```
Redis Error Occurs
    ↓
┌─────────────────────────────────────┐
│ What kind of API is this?           │
└──────────┬──────────────────────────┘
           │
    ┌──────┴──────┐
    │             │
Financial/      Regular
Critical        API
    │             │
    ▼             ▼
Fail Closed   Fail Open
(Deny)        (Allow)
    │             │
    └─────┬───────┘
          ▼
   Log + Alert
```

Lesson: Always plan for component failures, have explicit failure handling strategy.

Mistake 4: Not Considering Hot Keys

The Mistake:

```javascript
// Simple sharding
const shard = hash(userId) % NUM_SHARDS;
redis[shard].incr(`ratelimit:${userId}`);
```

Why It's Wrong:

```
Taylor Swift's user ID: user_celebrity
hash(user_celebrity) % 3 = 0 → Always Shard 0

Taylor has 10M fans, each checking her profile:
→ All requests go to Shard 0
→ Shard 0: 10M req/sec
→ Shards 1 & 2: Idle
→ Shard 0 crashes!
```

The Fix:

Option 1: Detect and Isolate

```
// Monitor request rates per key
if (requestRate[key] > THRESHOLD) {
  // Move to dedicated instance
  moveToSeparateRedis(key);
}

Taylor Swift → Dedicated Redis
Other users → Normal Redis cluster
```

Option 2: Probabilistic Sharding

```
// For high-traffic keys, split across multiple shards
if (isHighTrafficKey(userId)) {
  // Use micro-sharding
  const microShard = hash(userId + timestamp) % 10;
  const key = `ratelimit:${userId}:${microShard}`;
  // Sum across all micro-shards when checking
}
```

Lesson: Monitor for hot keys, have mitigation strategy ready.

Mistake 5: Over-Engineering Day 1

The Mistake:

```
Day 1 Requirements: 100 RPS

Design:
- 50 API servers
- 20-node Redis cluster
- Multi-region deployment
- Complex sharding logic
- Custom monitoring

Result:
- 6 months to build
- $50,000/month to run
- Team of 5 to maintain
```

Why It's Wrong:

```
You don't need 50 servers for 100 RPS!
You don't know if product will succeed!
Over-engineering = wasted time and money
```

The Fix - Start Simple:

```
Day 1 (100 RPS):
- 2 API servers
- 1 Redis instance (master + replica)
- Single region
- Cost: $200/month
- Time to build: 1 week

Scale as needed:
Week 4 (1K RPS): Add 3 more API servers
Month 6 (10K RPS): Add Redis cluster
Year 2 (100K RPS): Add regions
```

Growth Strategy:

```
┌─────────────────────────────────────────┐
│ Start Simple (MVP)                      │
│ - Prove product works                   │
│ - Learn actual usage patterns           │
│ - Iterate based on real data            │
└─────────────────────────────────────────┘
        ↓ Product succeeds
┌─────────────────────────────────────────┐
│ Scale Vertically First                  │
│ - Bigger Redis instance                 │
│ - More API servers                      │
│ - Same architecture                     │
└─────────────────────────────────────────┘
        ↓ Hit single-instance limits
┌─────────────────────────────────────────┐
│ Scale Horizontally                      │
│ - Redis cluster                         │
│ - Load balancers                        │
│ - Caching layers                        │
└─────────────────────────────────────────┘
        ↓ Hit regional limits
┌─────────────────────────────────────────┐
│ Multi-Region                            │
│ - Deploy to multiple regions            │
│ - GeoDNS routing                        │
│ - Regional independence                 │
└─────────────────────────────────────────┘
```

Lesson: Start simple, scale based on actual needs, not imagined future needs.

Practical Decision-Making Framework

Step 1: Understand Your Constraints

Ask yourself:

```
1. What's my actual traffic?
   Current: ____ RPS
   Expected (1 year): ____ RPS

2. What's my budget?
   Can spend: $____ /month

3. What's my team size?
   Engineers: ____
   Experience level: Junior / Mid / Senior

4. What's my accuracy requirement?
   Strict (99.9%+)  / Moderate (95%+)  / Loose (90%+)

5. What's my availability target?
   99.9% / 99.99% / 99.999%
```

Step 2: Map to Architecture

```
Traffic < 1K RPS:
→ Simple setup
→ Single region
→ 3 servers + Redis
→ Fixed Window OR Sliding Window Counter
→ Cost: $500/month

Traffic 1K-10K RPS:
→ Regional setup
→ 10-20 servers
→ Redis cluster (3 shards)
→ Sliding Window Counter
→ Cost: $5,000/month

Traffic 10K-100K RPS:
→ Multi-region
→ 50+ servers per region
→ Redis cluster per region
→ Sliding Window Counter
→ Cost: $30,000/month

Traffic > 100K RPS:
→ Global deployment
→ Custom optimizations
```

```
  → Dedicated team
  → Sliding Window Counter + Token Bucket (hybrid)
  → Cost: $100,000+/month
```

Step 3: Choose Algorithm Based on Needs

```
┌─────────────────────────────────────────────────────────┐
│ Decision Tree for Algorithm Selection                   │
└─────────────────────────────────────────────────────────┘

Q1: Do you need to allow bursts?
├─ YES → Consider Token Bucket
└─ NO  → Continue to Q2

Q2: Is strict accuracy critical? (Financial APIs)
├─ YES → Sliding Window Log
└─ NO  → Continue to Q3

Q3: Is this your first rate limiter? (MVP)
├─ YES → Start with Fixed Window (simple)
└─ NO  → Continue to Q4

Q4: Building for production?
└─ YES → Sliding Window Counter (recommended)
```

Step 4: Validate Your Design

```
Checklist:

Performance:
□ Latency < 10ms?
□ Can handle 10x traffic spike?
□ Redis connection pooling configured?

Reliability:
□ What happens if Redis fails?
□ Multiple availability zones?
□ Circuit breaker implemented?

Operations:
□ Monitoring/alerting configured?
□ Can update rules without restart?
□ Logs for debugging?

Security:
□ DDoS protection?
□ Authentication for admin APIs?
□ Rate limits on rate limiter itself?
```

Deep Dive: Why Sliding Window Counter is the Best Default

Let me explain this algorithm in much more detail because it's the **recommended choice** for most production systems.

The Mathematical Intuition

The Core Idea: Approximate the sliding window using two fixed windows.

```
Think of it like a temperature forecast:

Yesterday's high: 80°F
Today (halfway through): 70°F so far

Estimate for full day:
- If temp rises same as yesterday: ~80°F
- Current trajectory: ~70°F

Weighted estimate = (70 × 50%) + (80 × 50%) = 75°F

This gives us a reasonable estimate without tracking every minute!
```

Applied to Rate Limiting:

```
Previous Hour: 80 requests
Current Hour (30 minutes in): 30 requests

If user continues at same rate:
- 30 requests in 30 min
- Will make ~60 requests in full hour

But previous hour was heavier (80 requests)
So weighted estimate: 30 + (80 × 50%) = 70 requests

This smooths out the transition between windows!
```

Why the Weighting Works

Mathematical Proof (simplified):

```
Let's say true sliding window has X requests

Our estimate:
E = C + (P × W)
```

```
Where:
C = Current window count
P = Previous window count
W = Weight (decreases from 1 to 0)

Worst case (all requests at boundaries):
- P = 100, C = 100
- At 1% into current window: W = 99%
- E = 100 + (100 × 0.99) = 199
- True value = 200
- Error = 0.5%

Best case (even distribution):
- Error ≈ 0%

Average case:
- Error ≈ 1-3%
```

Why Only 2 Windows?

Question: Why not use 3 or 4 windows for more accuracy?

```
With 2 windows:
Memory: 16 bytes (2 counters)
Redis calls: 2 GETs
Latency: 0.5ms
Accuracy: 95-99%

With 3 windows:
Memory: 24 bytes (3 counters)
Redis calls: 3 GETs
Latency: 0.6ms
Accuracy: 97-99.5%

With 4 windows:
Memory: 32 bytes (4 counters)
Redis calls: 4 GETs
Latency: 0.8ms
Accuracy: 98-99.7%

Diminishing returns!
- 2→3 windows: +2% accuracy for +20% latency
- 3→4 windows: +1% accuracy for +30% latency

2 windows is the sweet spot!
```

Comparison with Sliding Window Log

```
Scenario: 1,000 requests/hour, 1M users

Sliding Window Counter:
- Memory: 1M users × 16 bytes = 16 MB ✓
- Latency: 0.5ms ✓
- Accuracy: 97% ✓

Sliding Window Log:
- Memory: 1M users × 1000 timestamps × 24 bytes = 24 GB ❌
- Latency: 2-5ms (need to scan timestamps) ⚠
- Accuracy: 99.9% ✓

For 3% accuracy improvement:
- Pay 1500x more memory
- Pay 5x more latency

Worth it? Usually NO.
When worth it? Financial APIs, regulatory requirements.
```

Understanding Redis Cluster

Why Cluster? Single Redis Limitations

Limitation 1: Memory

```
Single Redis max: 256 GB

Our needs: 10M users × 1KB per user = 10 GB (fits!)

But scaling:
100M users × 1KB = 100 GB (still fits)
1B users × 1KB = 1000 GB (doesn't fit!)

Solution: Cluster splits data across nodes
```

Limitation 2: Throughput

```
Single Redis: ~100K ops/second

Our peak: 17K RPS × 3 operations = 51K ops/second (fits!)

But Black Friday:
Peak: 100K RPS × 3 = 300K ops/second (doesn't fit!)

Solution: Cluster distributes load
```

Limitation 3: Availability

```
Single Redis crashes:
→ All rate limiting stops
→ Choose fail-open or fail-closed
→ Both bad

Cluster: One node crashes
→ Other nodes still work
→ Only 1/N of users affected
→ Gradual degradation (better!)
```

How Redis Cluster Works

Hash Slots Concept:

```
Redis Cluster uses 16,384 hash slots

Imagine 16,384 buckets:
Bucket 0, Bucket 1, ..., Bucket 16,383

Each key goes into one bucket:
CRC16(key) % 16384 = slot number

Example:
CRC16("user_123") % 16384 = 5,234
→ user_123 goes in slot 5,234
```

Distribution Across Nodes:

```
3-node cluster:
Node A: Slots 0 - 5,461      (33%)
Node B: Slots 5,462 - 10,922 (33%)
Node C: Slots 10,923 - 16,383 (33%)

Key "user_123" → Slot 5,234 → Node A
Key "user_456" → Slot 10,500 → Node B
Key "user_789" → Slot 15,000 → Node C

Evenly distributed!
```

Adding a Node:

```
Before (3 nodes):
Node A: 5,461 slots
Node B: 5,461 slots
Node C: 5,462 slots

Add Node D:

After (4 nodes):
Node A: 4,096 slots (moved 1,365 to D)
Node B: 4,096 slots (moved 1,365 to D)
Node C: 4,096 slots (moved 1,366 to D)
Node D: 4,096 slots (received 4,096)

Each existing node gave up ~25% of slots
This is MUCH better than rehashing all keys!
```

Cluster vs Sentinel: When to Use Each

Redis Sentinel (Master-Replica Monitoring):

```
Use when:
- Single shard is enough (< 100K ops/sec)
- Need automatic failover
- Want simple setup

Architecture:
Master → Replicas
Sentinel monitors and promotes on failure
```

Redis Cluster (Sharding + Replication):

```
Use when:
- Need multiple shards (> 100K ops/sec)
- Need to distribute data (> 100 GB)
- Want automatic sharding

Architecture:
Multiple Masters (shards)
Each has replicas
Auto-failover built-in
```

Our Recommendation:

```
Start: Redis Sentinel (simpler)
Scale: Upgrade to Redis Cluster when needed

Migration path:
1. Single master + replica (Sentinel)
```

```
2. Add more memory (vertical scaling)
3. When hitting limits → Migrate to Cluster
4. Cluster handles future growth
```

Monitoring: Why It's Critical

The Problem Without Monitoring

```
Scenario: Rate limiter starts failing

Without monitoring:
Day 1: 1% of requests fail (unnoticed)
Day 2: 5% fail (users complain on Twitter)
Day 3: 20% fail (customers leave)
Day 4: You discover the issue
Result: Lost revenue, angry customers

With monitoring:
Minute 1: Alert fires (1% error rate)
Minute 5: Team investigates
Minute 10: Issue identified (Redis memory full)
Minute 15: Scale Redis cluster
Minute 20: Issue resolved
Result: Minimal impact
```

What to Monitor - The Four Golden Signals

1. Latency (How long requests take)

```
Why it matters:
Slow rate limiting = Slow API for everyone

What to measure:
- P50 (median): 50% of requests
- P95: 95% of requests
- P99: 99% of requests (catch outliers)

Thresholds:
- P50 < 1ms: ✅ Great
- P95 < 5ms: ✅ Good
- P99 < 10ms: ✅ Acceptable
- P99 > 50ms: ❌ Problem!

Why P99 matters more than average:
Average might be 2ms but P99 could be 100ms
→ 1% of users have terrible experience
→ Those are your most active users (making most requests)
→ They're most likely to complain/churn
```

2. Traffic (Request volume)

```
Why it matters:
Sudden changes indicate attacks or issues

What to measure:
- Total requests/second
- Allowed vs denied ratio
- Per-endpoint breakdown
- Per-user top consumers

Patterns to detect:
- Sudden 10x spike: DDoS attack?
- Gradual increase: Organic growth (good!)
- High rejection rate: Limits too strict?
- Single user 50% of traffic: Hot key problem?
```

3. Errors (Things going wrong)

```
Why it matters:
Errors invisible to users still cause problems

What to measure:
- Redis connection errors
- Timeout errors
- Lua script errors
- Configuration lookup failures

Example cascading failure:
Redis connection errors increase
→ Rate limiter falls back to local cache
→ Less accurate limiting
→ Some users exceed limits
→ Backend database overloaded
→ Site slows down
→ User experience degraded

Catching it early (Redis errors) prevents cascade!
```

4. Saturation (How full are your resources)

```
Why it matters:
Hitting resource limits causes failures
```

```
What to measure:
- Redis memory usage (80% = warning)
- CPU usage on API servers
- Network bandwidth usage
- Connection pool saturation

Example:
Redis at 90% memory
→ Close to limit
→ Start evicting keys (bad!)
→ Or OOM crash (worse!)

Alert at 80% → Scale before hitting limit
```

Alert Fatigue: The Balance

Too Many Alerts (Bad):

```
Every minute: "Latency is 11ms!" (threshold: 10ms)
Result:
- Team ignores alerts
- Real issues missed
- "Boy who cried wolf" problem
```

Too Few Alerts (Bad):

```
Only alert on complete outage
Result:
- Small issues grow into big issues
- No early warning
- Reactive instead of proactive
```

Just Right (Our approach):

```
Tiered Alerting:

Page (wake up engineer):
- Redis completely down
- Error rate > 50%
- P99 latency > 100ms for 5+ minutes

Warn (check during work hours):
- Error rate > 10%
- Memory > 80%
- P99 latency > 20ms for 5+ minutes
```

```
Info (review in weekly meeting):
- Approaching limits
- Trending issues
- Optimization opportunities
```

Real-World Scenario Walkthroughs

Scenario 1: Black Friday Sale

Situation:

- Normal traffic: 5,000 RPS
- Black Friday: 50,000 RPS (10x)
- Duration: 2 hours

How Our System Handles It:

```
T-10 minutes (Preparation):
- Auto-scaling policy already configured
- Redis cluster sized for 3x normal (15K RPS)
- Need to handle 50K RPS

T-0 (Sale starts):
- Traffic: 5K → 50K RPS instantly
- API Servers: CPU spikes to 90%

T+1 minute:
- Auto-scaler triggers (CPU > 70%)
- Starts launching 30 new servers
- Current: 10 servers at 100% CPU (uh oh!)

T+2 minutes:
- New servers starting up
- Still 10 servers (overloaded)
- Some requests timeout

T+3 minutes:
- 20 new servers online
- Total: 30 servers
- CPU drops to 60%
- System stabilizes!

T+120 minutes (Sale ends):
- Traffic returns to 5K RPS
- 30 servers now at 10% CPU
- Auto-scaler waits 5 minutes
- Then gradually scales down
- After 30 minutes: Back to 10 servers
```

Lessons:

1. **Pre-warming** is better than auto-scaling for known events

   ```
   Before Black Friday:
   - Manually scale to 40 servers
   - Avoid the 2-minute struggle
   - Cost: $100 extra, Worth it!
   ```

2. **Token Bucket helps** with sudden bursts

   ```
   Configuration:
   - Normal: 100 req/minute per user
   - Burst: 200 requests allowed
   - Accommodates initial rush
   ```

3. **Multi-tier rate limiting**

   ```
   Global limit: 50K RPS (protect overall system)
   Per-user limit: 100 req/min (fairness)
   Per-endpoint: Vary by criticality
   ```

Scenario 2: Redis Master Failure

Situation: Redis master crashes during peak traffic

```
T=0 (Crash occurs):
Master Redis crashes
All write operations start failing

T+1 second:
API servers notice errors
Circuit breaker: After 5 failures, open
Start failing open (allow all requests)

T+3 seconds:
Redis Sentinel detects master is down
Begins promoting replica to master

T+5 seconds:
Replica promoted to master
DNS/config updated

T+6 seconds:
```

```
API servers detect new master
Circuit breaker: Try new master
First request succeeds!

T+7 seconds:
Circuit breaker closes
Normal operation resumed

Total impact:
- 6 seconds of no rate limiting
- ~100K requests allowed without checking
- Acceptable! (fail-open strategy)
```

Alternative Scenario - Fail Closed:

```
T=0: Master crashes
T+1: Start rejecting ALL requests
     (Can't check rate limit, so deny)

Impact:
- 6 seconds of complete API outage
- $2,000 revenue loss
- Angry customers
- Support tickets

Much worse than allowing 100K unchecked requests!
```

Lesson: Fail-open usually better for rate limiting.

Scenario 3: Viral Tweet (100x Traffic)

Situation: Celebrity tweets your website

```
Before Tweet:
- Traffic: 1,000 RPS
- Capacity: 3,000 RPS (3x buffer)
- System healthy

T=0 (Tweet posted):
- Traffic: 1K → 100K RPS instantly!
- Way beyond capacity

T+1 minute:
Without proper rate limiting:
→ Servers overloaded
→ Database crashes
→ Site down
→ Nobody can access
```

```
With proper rate limiting:
→ Rate limiter kicks in
→ Allows 3K RPS (capacity)
→ Returns 429 to 97K requests
→ Site stays up for 3K users
→ Better than 0 users!
```

Improvement Strategy:

```
Short-term (0-5 minutes):
- Rate limiter doing its job
- Auto-scaling launches more servers
- Capacity increases: 3K → 10K RPS

Medium-term (5-30 minutes):
- More servers online
- Capacity: 10K → 50K RPS
- Most users getting through

Long-term (30+ minutes):
- Fully scaled
- Capacity: 50K → 100K RPS
- Handle the traffic!

Additional tactics:
- Increase user rate limits temporarily
- Prioritize authenticated users
- Serve cached content where possible
- Add CDN caching
```

Common Interview Questions - Detailed Answers

Q1: "Why not implement rate limiting in the client?"

What they're really asking: Can we save server resources by having clients enforce their own limits?

The Answer:

```
Client-side rate limiting:
- User can modify client code
- Malicious users bypass limits
- No actual protection

Example:
// Client code
if (requestCount < 100) {
  makeRequest();
}

Attacker:
// Modified client code
if (true) {  // Always true!
  makeRequest();
}

Result: No rate limiting at all!
```

When client-side IS useful:

```
As a UX enhancement (not security):

Example: Twitter app
- Client tracks: "You have 10 tweets left this hour"
- Prevents user from hitting rate limit
- Shows clear countdown
- But server still enforces!

Benefits:
✓ Better user experience
✓ Reduces wasted requests
✓ Educational for users

Still need server-side:
✓ Actual enforcement
✓ Can't be bypassed
✓ Protects against abuse
```

Complete Answer:
"Client-side rate limiting improves UX but can't replace server-side enforcement. Malicious users can bypass client logic, so we must enforce limits on the server. However, client-side limiting as a UX enhancement (showing remaining quota, preventing unnecessary requests) is valuable when combined with server enforcement."

Q2: "How do you handle database rate limiting?"

What they're asking: Rate limiting is about API requests, but what about protecting the database?

The Answer - Layered Protection:

```
Layer 1: API Rate Limiting (what we've discussed)
User → API Gateway → Rate Limiter
Effect: Limits requests reaching API
```

```
Layer 2: Database Connection Pooling
API → Connection Pool → Database
Pool size: 20 connections (limit concurrency)

Layer 3: Database Query Rate Limiting
Expensive queries get separate limits

Layer 4: Database Circuit Breaker
If DB slow, stop sending requests
```

Detailed Example:

```
Scenario: Analytics endpoint hits database hard

Without DB protection:
User calls /api/analytics (runs 10-second query)
→ 100 users call it
→ 100 × 10-second queries
→ Database: 100 connections, all slow queries
→ Database crashes
→ Everything fails

With layered protection:

Layer 1 (API Rate Limit):
- /api/analytics: Max 10 req/minute per user
- Limits how many can call it

Layer 2 (Connection Pool):
- Max 20 concurrent database connections
- Other requests queue

Layer 3 (Query-Specific Limit):
- Analytics queries: Max 5 concurrent
- Even if 20 connections available

Result:
- Max 5 slow queries at once
- Database stays healthy
- Other APIs unaffected
```

Complete Answer:
"Database protection requires layered rate limiting: API-level limits reduce request volume, connection pooling limits concurrency, and query-specific limits protect against expensive operations. We might also implement separate rate limit pools for endpoints that hit the database vs those that don't, using the bulkhead pattern to isolate failures."

Q3: "What if users share IP addresses (NAT)?"

The Problem:

```
Corporate Office:
- 1,000 employees
- All share IP: 1.2.3.4 (corporate NAT)

Rate limit: 100 req/hour per IP

Result:
1,000 employees share 100 req/hour
= 0.1 requests per hour per employee
= 1 request every 10 hours per employee!

Completely unusable!
```

Solutions:

Solution 1: Don't Rate Limit by IP Alone

```
Instead of:
  Key: ip:1.2.3.4

Use:
  Key: user_id:12345  (if authenticated)
  Fallback to IP only if unauthenticated

Result:
- Each employee has own limit (100 req/hour)
- IP-based rate limiting only for unauthenticated
```

Solution 2: Detect and Whitelist Known Corporate IPs

```
Identify corporate IPs:
- Look for IPs with many unique user_ids
- Manual submission by companies
- Increase limits for known corporate IPs

Example:
Normal IP: 100 req/hour
Corporate IP: 10,000 req/hour

Trade-off: Some abuse possible, but better UX
```

Solution 3: Use X-Forwarded-For Header

```
Client IP: 1.2.3.4 (NAT)
X-Forwarded-For: 192.168.1.50 (internal IP)

Consider both:
- Rate limit by internal IP (if trustworthy proxy)
- Combine: user_id + internal_ip

Caveat: Header can be spoofed, only trust if from known proxies
```

Complete Answer:
"We should avoid rate limiting by IP alone. Instead, use authenticated user IDs as the primary identifier, falling back to IP only for unauthenticated requests. For known corporate IPs with legitimate high traffic, we can maintain a whitelist with higher limits. We can also use X-Forwarded-For for internal IP tracking, but only from trusted proxies since the header can be spoofed."

Q4: "How do you test a rate limiter?"

What they're asking: How do you verify it works correctly?

Testing Strategy:

Level 1: Unit Tests

```python
def test_sliding_window_counter():
    limiter = SlidingWindowCounter(limit=10, window=60)

    # Test 1: Allow up to limit
    for i in range(10):
        assert limiter.allow("user1") == True

    # Test 2: Deny over limit
    assert limiter.allow("user1") == False

    # Test 3: Reset after window
    time.sleep(61)
    assert limiter.allow("user1") == True

    # Test 4: Boundary condition
    # Make 10 requests at end of window
    # Advance time 1 second
    # Should not allow 10 more immediately
```

Level 2: Integration Tests

```python
def test_redis_rate_limiting():
    # Test actual Redis integration
    client = RateLimiterClient(redis_url)

    # Test normal operation
    response = client.check_limit(user="test", endpoint="/api")
    assert response.allowed == True

    # Test limit enforcement
    for _ in range(100):
        client.check_limit(user="test", endpoint="/api")

    response = client.check_limit(user="test", endpoint="/api")
    assert response.allowed == False
    assert "retry_after" in response
```

Level 3: Load Tests

```
# Simulate 10K RPS for 5 minutes
wrk -t12 -c400 -d300s http://api/rate-limit-check

Measure:
- Latency distribution
- Error rate
- Redis CPU/memory
- API server CPU/memory

Success criteria:
- P99 latency < 10ms
- Error rate < 0.1%
- No memory leaks
- System remains stable
```

Level 4: Chaos Testing

```
# Test failure scenarios
def test_redis_failure():
    # Normal operation
    assert check_rate_limit("user1") == True

    # Kill Redis
    kill_redis_master()

    # Should fail-open (allow requests)
    assert check_rate_limit("user1") == True
    assert check_rate_limit("user2") == True

    # Check that errors are logged
    assert "redis connection error" in logs
    # Restore Redis
    start_redis_master()

    # Should resume normal operation
    time.sleep(5)  # Wait for reconnection
    # Now rate limiting should work again
```

Level 5: Production Testing (Gradual Rollout)

```
Week 1: Deploy to 1% of traffic
- Monitor errors
- Monitor latency
- Compare to baseline

Week 2: Increase to 10%
- Still healthy? Continue

Week 3: Increase to 50%

Week 4: Full rollout (100%)

If issues at any stage:
→ Rollback immediately
→ Fix issues
→ Start over

This is how big companies (Google, Facebook) roll out changes!
```

Key Architectural Principles Summary

Principle 1: Design for Failure

```
Assumption: Everything will fail eventually

Components that WILL fail:
- Redis
- API servers
- Network
- Load balancers
- Entire data centers

Design with failure in mind:
✓ Circuit breakers
✓ Graceful degradation
✓ Automatic failover
✓ Multiple availability zones
✓ Comprehensive monitoring
```

Principle 2: Start Simple, Scale Later

```
Avoid premature optimization

Day 1: Simple is better
- 2 servers
- 1 Redis
- Basic monitoring
- Ship fast!

Learn from real usage:
- Which endpoints need strict limits?
- What's actual traffic pattern?
- Where are bottlenecks?

Scale based on data:
- Not guesses
- Not "what if"
- Real metrics
```

Principle 3: Favor Availability Over Consistency

```
For rate limiting specifically:

User experience priority:
1. Site works (most important)
2. Site is fast
3. Rate limits are approximate

NOT:
1. Perfect rate limit accuracy
2. Site might be down
3. Users frustrated

Exception: Financial/regulated APIs
→ Then consistency might be more important
```

Principle 4: Make It Observable

```
You can't fix what you can't see

Must have:
✓ Metrics (Prometheus)
✓ Logs (structured, searchable)
✓ Distributed tracing (see request flow)
✓ Alerts (actionable)
✓ Dashboards (visualize health)

Without observability:
- Flying blind
- Slow incident response
- Can't optimize

With observability:
- See issues immediately
- Fast resolution
- Data-driven optimization
```

Final Checklist for Your Design

When presenting your rate limiter design, make sure you can answer:

Architecture Questions

- ☐ Why stateless API gateways?
- ☐ Why Redis and not alternatives?
- ☐ Why separate config store?
- ☐ How does caching work?
- ☐ What's the data flow?

Algorithm Questions

- ☐ Which algorithm did you choose?
- ☐ Why that algorithm?
- ☐ What are the trade-offs?
- ☐ How accurate is it?
- ☐ How does it handle bursts?

Distributed Systems Questions

- ☐ How do you handle race conditions?
- ☐ What about clock skew?
- ☐ How do you shard data?
- ☐ What if Redis fails?
- ☐ CAP theorem trade-offs?

Scalability Questions

- ☐ How do you scale to 10x traffic?
- ☐ Auto-scaling strategy?
- ☐ How many servers needed?
- ☐ Cost estimation?
- ☐ Bottlenecks identified?

Operations Questions

- ☐ What do you monitor?
- ☐ What alerts do you set?
- ☐ How do you debug issues?
- ☐ How do you deploy updates?
- ☐ Disaster recovery plan?

Conclusion: The Learning Journey

You've Learned:

1. **System Design Thinking**
    - Ask questions before solving
    - Consider trade-offs explicitly
    - Start simple, iterate
    - Design for failure
2. **Distributed Systems Concepts**
    - Atomic operations
    - CAP theorem
    - Consistency models
    - Failure modes
3. **Architectural Patterns**
    - Stateless services
    - Caching strategies
    - Sharding/partitioning
    - Circuit breakers
    - Bulkhead pattern
4. **Practical Skills**
    - Capacity estimation
    - Cost calculation
    - Technology selection
    - Monitoring strategy

What Makes a Good System Design?

```
Not just: "Works correctly"

But also:
✓ Scalable (can grow 10x)
✓ Reliable (handles failures gracefully)
✓ Observable (can debug issues)
✓ Cost-effective (doesn't waste money)
✓ Maintainable (team can operate it)
✓ Simple (no unnecessary complexity)
```

Your Next Steps

1. **Build a simple rate limiter**
    - Start with fixed window
    - Single server + Redis
    - Get it working
2. **Add complexity gradually**
    - Switch to sliding window counter
    - Add monitoring
    - Add auto-scaling
3. **Study real systems**
    - Read Stripe API docs
    - Read GitHub API docs
    - Understand their choices
4. **Practice explaining**
    - Draw architectures
    - Explain trade-offs
    - Use analogies
    - Think out loud

Remember:

> "Premature optimization is the root of all evil" - Donald Knuth

Start simple. Measure. Optimize based on data. This is how you build great systems.

Document Purpose: Educational guide explaining architectural decisions
Target Audience: Engineers learning distributed systems design
Key Focus: WHY behind decisions, not just WHAT or HOW

<task_progress>

- ✅ Create comprehensive educational guide
- ✅ Explain why rate limiting exists
- ✅ Break down all architectural decisions with reasoning
- ✅ Compare storage options (Redis vs PostgreSQL vs alternatives)
- ✅ Explain caching strategy in detail

- ✅ Detail all algorithm choices with analogies
- ✅ Cover distributed systems concepts (CAP, atomicity, sharding)
- ✅ Explain scaling patterns with examples
- ✅ Provide real-world scenario walkthroughs
- ✅ Answer common interview questions in depth
- ✅ Include practical decision-making framework
- ✅ Add final checklist and learning summary
  </task_progress>