Rate Limiter System Design - High-Level Design Focus

Table of Contents

Problem Statement & Requirements

Problem Definition

Design a distributed rate limiting system that can:

- Limit API requests per user/IP/API key
- Handle 10,000+ requests/second
- Work across multiple servers and regions
- Maintain <10ms latency overhead
- Support multiple rate limit rules and tiers

Functional Requirements

1. **Limit enforcement**: Block requests exceeding configured limits
2. **Multiple dimensions**: Rate limit by user ID, IP, API key, endpoint
3. **Flexible windows**: Support second, minute, hour, day windows
4. **Dynamic rules**: Update limits without service restart
5. **Clear feedback**: Return meaningful error messages with retry information

Non-Functional Requirements

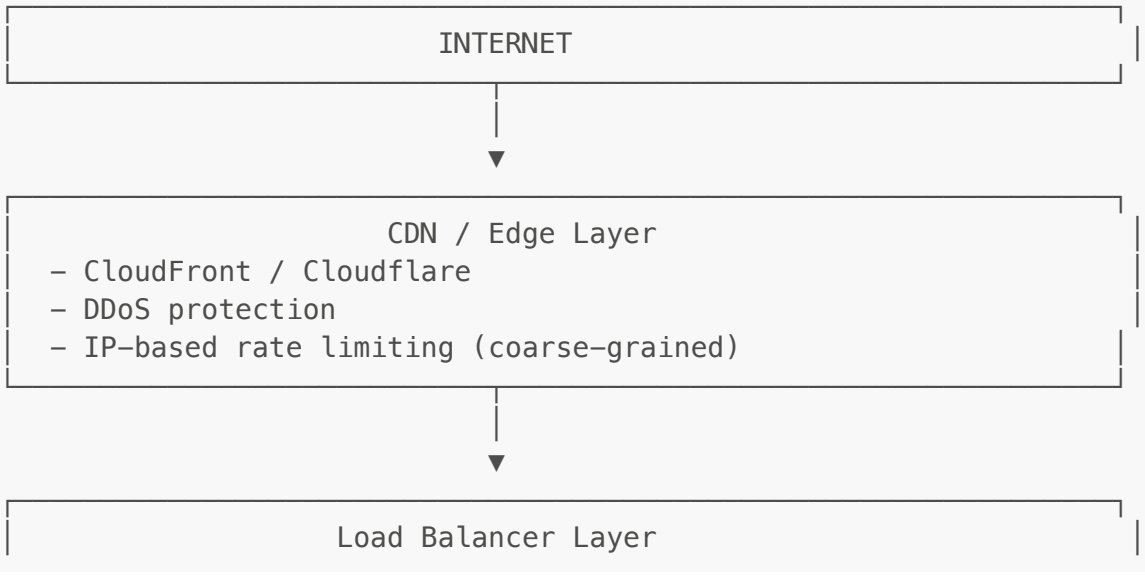
1. **Performance**: P99 latency < 10ms, support 10K+ RPS
2. **Availability**: 99.99% uptime
3. **Scalability**: Horizontal scaling, handle 10x growth
4. **Consistency**: Accept <5% over-limit (eventual consistency)
5. **Durability**: Rules survive restarts

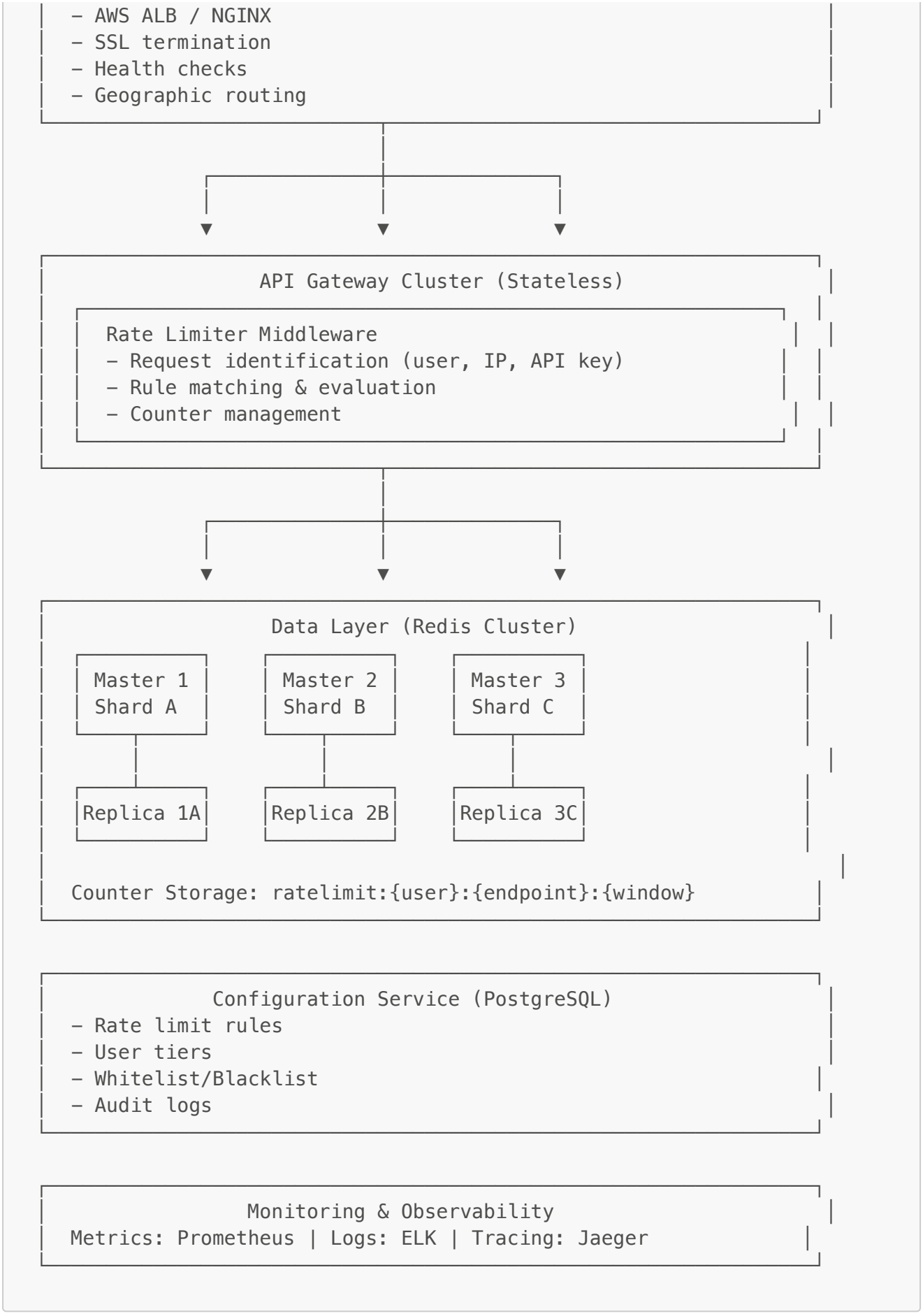
Capacity Estimation

```
Assumptions:
- 100M total users
- 10M DAU
- 50 requests/user/day
- Peak = 3x average

Daily requests = 10M × 50 = 500M
Average RPS = 500M / 86400 ≈ 5,800
Peak RPS = 17,400

Storage (Redis):
- Per user counter: 8 bytes
- 4 time windows: 32 bytes/user
- 10M DAU: 320MB
- With overhead: ~640MB

Servers needed:
- Each server: 1,000 RPS
- Peak: 17,400 / 1,000 = 18 servers
- With redundancy & growth: 70-100 servers

Cost (AWS):
- API Gateway: ~$5,000/month
- Redis Cluster: ~$1,000/month
- Load Balancers: ~$100/month
Total: ~$6,000/month
```

High-Level Architecture

System Overview





Key Architectural Decisions

1. Stateless API Gateway

- **Why**: Enables horizontal scaling, no session affinity needed
- **How**: All state in Redis, gateways are compute-only
- **Trade-off**: Network latency to Redis vs local cache

2. Redis as Central Store

- **Why**: Fast (sub-millisecond), atomic operations, proven at scale
- **Alternatives considered**:
  - In-memory (doesn't scale across servers)
  - Cassandra (higher latency ~10ms)
  - DynamoDB (higher latency, eventually consistent)
- **Trade-off**: Single point of failure → mitigated with clustering

3. Separate Configuration Store

- **Why**: Rules change infrequently, don't need sub-ms access
- **How**: PostgreSQL with caching layer in API gateways
- **Trade-off**: Eventual consistency for rule updates (acceptable 5min delay)

4. Multi-Tier Caching

```
L1: In-memory rules cache (5 min TTL)
    ↓ (cache miss)
L2: Redis rules cache (15 min TTL)
    ↓ (cache miss)
L3: PostgreSQL (source of truth)
```

Core Components

1. Request Identifier Service

Purpose: Extract and normalize user identifiers

```
Input: HTTP Request
Output: Composite Key

Components:
┌─────────────────────────────────────┐
│  User ID: user_12345                │
│  IP Address: 192.168.1.1            │
│  API Key: sk_live_abc123            │
│  Endpoint: /api/payments            │
│  Method: POST                       │
└─────────────────────────────────────┘
```

```
          ↓
Generate Key: "ratelimit:user_12345:/api/payments:1640995200"
```

Design Considerations:

- Hash user IDs for privacy
- Handle missing identifiers (fallback to IP)
- Normalize endpoints (strip query params)
- Consider X-Forwarded-For for true client IP

2. Rules Engine

Purpose: Match requests to rate limit rules with precedence

```
Rule Priority System:
┌──────────────────────────────────────────────────┐
│ Priority 1: User-specific + Endpoint-specific    │
│   Example: Premium user calling /api/payments    │
│   Limit: 10,000 req/hour                         │
└──────────────────────────────────────────────────┘
         ↓ (no match)
┌──────────────────────────────────────────────────┐
│ Priority 2: Endpoint-specific                    │
│   Example: Any user calling /api/payments        │
│   Limit: 1,000 req/hour                          │
└──────────────────────────────────────────────────┘
         ↓ (no match)
┌──────────────────────────────────────────────────┐
│ Priority 3: User tier-specific                   │
│   Example: Free tier user, any endpoint          │
│   Limit: 100 req/hour                            │
└──────────────────────────────────────────────────┘
         ↓ (no match)
┌──────────────────────────────────────────────────┐
│ Priority 4: Global default                       │
│   Limit: 1,000 req/hour                          │
└──────────────────────────────────────────────────┘
```

Key Features:

- Rule caching (5-minute TTL)
- Hot reload without restart
- A/B testing support
- Gradual rollout capability

3. Counter Management Service

Purpose: Track request counts atomically

Architecture:

```
API Gateway
    ↓
[Check Counter] → Redis GET current_window, previous_window
    ↓
[Calculate] → Sliding window estimation
    ↓
[Decision] → Allow / Deny
    ↓
[Update] → Redis INCR current_window (if allowed)
```

Atomicity Guarantees:

- Use Redis Lua scripts for atomic check-and-increment
- Prevents race conditions across servers
- Handles concurrent requests correctly

4. Response Builder

Purpose: Generate consistent API responses

```
Allowed Response:
├── Status: 200 OK
├── Headers:
│   ├── X-RateLimit-Limit: 1000
│   ├── X-RateLimit-Remaining: 742
│   └── X-RateLimit-Reset: 1640995200
└── Body: [Original API Response]

Denied Response:
├── Status: 429 Too Many Requests
├── Headers:
│   ├── X-RateLimit-Limit: 1000
│   ├── X-RateLimit-Remaining: 0
│   ├── X-RateLimit-Reset: 1640995200
│   └── Retry-After: 60
└── Body:
    {
      "error": "RATE_LIMIT_EXCEEDED",
      "message": "Try again in 60 seconds",
      "limit": 1000,
      "retry_after": 60
    }
```

Algorithm Selection

Algorithm Comparison Matrix

Criteria	Fixed Window	Sliding Log	Sliding Counter	Token Bucket
Accuracy	⭐⭐	⭐⭐⭐⭐⭐	⭐⭐⭐⭐	⭐⭐⭐
Memory	⭐⭐⭐⭐⭐	⭐⭐	⭐⭐⭐⭐	⭐⭐⭐⭐
Complexity	⭐⭐⭐⭐⭐	⭐⭐	⭐⭐⭐	⭐⭐⭐
Burst Handling	❌	✅	✅	✅✅
Boundary Issues	❌	✅	✅	✅
Distributed	✅	⚠️	✅	⚠️

Recommended: Sliding Window Counter

Why this algorithm?

1. **Good accuracy** (~95-99%, avoids fixed window boundary issue)
2. **Memory efficient** (only 2 counters per user per endpoint)
3. **Distributed-friendly** (works well with Redis)
4. **Handles bursts** (smooth transition between windows)

How it works conceptually:

```
Previous Window: [═══════] 80 requests
Current Window:  [══      ] 30 requests
                     ↑ Now (50% through window)

Estimated count = 30 + (80 × 50%) = 30 + 40 = 70 requests

If limit is 100: ALLOW (70 < 100)
If limit is 60:  DENY (70 > 60)
```

Architecture:

```
Redis Keys:
- ratelimit:user_123:/api/data:1640995200  → 30
- ratelimit:user_123:/api/data:1640995140  → 80

Calculation:
1. Get both counters (2 Redis GETs)
2. Calculate weighted sum
3. Compare to limit
4. If allowed: INCR current counter
5. Set TTL on counter (2× window size)
```

Alternative: Token Bucket (for burst scenarios)

When to use:

- Mobile apps that sync periodically
- Batch operations
- APIs where legitimate bursts are common

Conceptual model:

```
Bucket Capacity: 100 tokens
Refill Rate: 10 tokens/second

Time 0s:  [100 tokens] ← Full
          ↓ 50 requests
Time 1s:  [50 tokens] ← Half full
          ↓ (refilled 10)
Time 2s:  [60 tokens]
          ↓ 0 requests
Time 3s:  [70 tokens] ← Gradually refilling
```

Data Flow & Request Processing

Detailed Request Flow

```
┌─────────────────────────────────────────────────────────────┐
│ STEP 1: Request Arrives                                     │
└─────────────────────────────────────────────────────────────┘

         Client Request
         ↓
    POST /api/payments
    Headers:
      X-API-Key: sk_live_abc123
      X-User-ID: user_12345
    From IP: 192.168.1.1

┌─────────────────────────────────────────────────────────────┐
│ STEP 2: Load Balancer Routing                               │
└─────────────────────────────────────────────────────────────┘

         ↓
    [SSL Termination]
    [Health Check]
    [Route to API Gateway]
         ↓
    Selected: API Gateway Instance #7

┌─────────────────────────────────────────────────────────────┐
│ STEP 3: Rate Limiter Middleware Intercepts                  │
```

```
└──────────────────────────────────────────────────────────────┘
        ↓
    Extract Identifiers:
      user_id: user_12345
      api_key: sk_live_abc123
      ip_address: 192.168.1.1
      endpoint: /api/payments
        ↓
    Check Local Rules Cache:
      Cache Key: "rule:user_12345:/api/payments"
      Cache TTL: 5 minutes
      Status: HIT ✓
        ↓
    Applicable Rule:
      limit: 1000 requests
      window: 3600 seconds (1 hour)
      algorithm: sliding_window_counter

┌──────────────────────────────────────────────────────────────┐
│ STEP 4: Query Redis for Counters                             │
└──────────────────────────────────────────────────────────────┘
        ↓
    Current Time: 1640995200
    Current Window: 1640995200
    Previous Window: 1640991600
        ↓
    Redis Pipeline (batch operation):
      GET ratelimit:user_12345:/api/payments:1640995200
      GET ratelimit:user_12345:/api/payments:1640991600
        ↓
    Results:
      current_count: 45
      previous_count: 820

┌──────────────────────────────────────────────────────────────┐
│ STEP 5: Calculate Weighted Count                             │
└──────────────────────────────────────────────────────────────┘
        ↓
    Time elapsed in current window: 1800s (50%)
    Previous window weight: 50%
        ↓
    Estimated count = 45 + (820 × 0.5) = 45 + 410 = 455
        ↓
    Compare: 455 < 1000 ✓
    Decision: ALLOW

┌──────────────────────────────────────────────────────────────┐
│ STEP 6: Update Counter                                       │
└──────────────────────────────────────────────────────────────┘
        ↓
    Redis Commands:
      INCR ratelimit:user_12345:/api/payments:1640995200
      EXPIRE ratelimit:user_12345:/api/payments:1640995200 7200
```

```
          ↓
    New count: 46

┌─────────────────────────────────────────────────────────────┐
│ STEP 7: Add Response Headers                                │
└─────────────────────────────────────────────────────────────┘

          ↓
    X-RateLimit-Limit: 1000
    X-RateLimit-Remaining: 545
    X-RateLimit-Reset: 1640998800

┌─────────────────────────────────────────────────────────────┐
│ STEP 8: Forward to Backend Service                          │
└─────────────────────────────────────────────────────────────┘

          ↓
    [Process Payment]
    [Return Response]
          ↓
    Response: 200 OK {"payment_id": "12345"}

┌─────────────────────────────────────────────────────────────┐
│ STEP 9: Return to Client                                    │
└─────────────────────────────────────────────────────────────┘

          ↓
    Response with rate limit headers
```

Performance Optimization

Total Latency Breakdown:

```
Rate Limiter Overhead:
├── Rule lookup (cache): 0.1ms
├── Redis GET (2 keys): 0.5ms
├── Calculation: 0.1ms
├── Redis INCR: 0.3ms
└── Total: ~1ms ✓

Target: <10ms (10% of typical API response time)
Actual: ~1ms (well within target)
```

Distributed System Design

Challenge 1: Maintaining Consistency

Problem: Multiple servers incrementing same counter

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

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

Result: Lost update! Should be 101
```

Solution: Atomic Operations

```
Instead of Read-Then-Write:
  ❌ count = GET(key)
  ❌ count += 1
  ❌ SET(key, count)

Use Atomic Increment:
  ✅ INCR(key)  ← Atomic operation in Redis
```

Advanced: Lua Scripts for Complex Logic

```lua
-- Atomic check-and-increment
local current = redis.call('INCR', key)
if current == 1 then
    redis.call('EXPIRE', key, window)
end

if current > limit then
    redis.call('DECR', key)  -- Rollback
    return 0  -- Rejected
else
    return 1  -- Allowed
end
```

Challenge 2: Clock Synchronization

Problem: Server clocks drift

```
Server A: 10:00:00.000
Server B: 10:00:00.500

Window boundary at 10:00:00:
- Different servers see different windows
- Inconsistent rate limiting
```

Solutions:

Option 1: Centralized Time Source

```
All servers → Query Redis TIME command
- Redis provides synchronized timestamp
- Network latency: +0.5ms
- Trade-off: Slight overhead for consistency
```

Option 2: NTP Synchronization

```
All servers sync with NTP
- Accuracy: ±50ms
- Good enough for rate limiting (not for distributed transactions)
- Zero runtime overhead
```

Option 3: Accept Slight Inaccuracy

```
- Most rate limiting can tolerate ±1 second drift
- Simpler architecture
- Focus on approximate fairness
```

Challenge 3: Redis Failure Scenarios

Scenario A: Master Failure

```
Normal:
  API Gateways → Redis Master → Redis Replica

Failure:
  Master crashes
  ↓
  Sentinel detects failure (<5s)
  ↓
  Promotes Replica to Master (<5s)
  ↓
  Updates DNS / Endpoint
  ↓
  API Gateways reconnect

Impact: 5-10s of potential data loss
Mitigation: Fail-open during failover
```

Scenario B: Network Partition

```
Datacenter split:
  API Gateways (DC1) ←x→ Redis (DC2)

Options:
1. Fail Open (Recommended)
   - Allow all requests
   - Log failures
   - Resume when partition heals

2. Fail Closed (High security)
   - Deny all requests
   - Potential service outage

3. Local Fallback
   - Use in-memory counters
   - Less accurate but available
   - Sync when partition heals
```

Challenge 4: Hot Key Problem

Problem: Celebrity user causes hot key

```
Taylor Swift's API key: "ts_key"
Rate limit key: "ratelimit:ts_key:*"

All requests hit same Redis shard
→ Shard overloaded
→ Latency spikes for everyone
```

Solutions:

Option 1: Key Splitting

```
Instead of:
  ratelimit:ts_key:endpoint:window

Use:
  ratelimit:ts_key:endpoint:window:shard_0
  ratelimit:ts_key:endpoint:window:shard_1
  ratelimit:ts_key:endpoint:window:shard_2

Sum across shards for total count
Distribute load across Redis shards
```

Option 2: Dedicated Shard

```
Detect high-traffic users
Move to dedicated Redis instance
Isolate from normal traffic
```

Option 3: Client-Side Rate Limiting

```
For known high-volume clients:
- Implement client-side token bucket
- Reduce server load
- Still enforce server-side as backstop
```

Scalability & Performance

Scaling Dimensions

1. Horizontal Scaling (API Gateways)

Auto-Scaling Strategy:

```
Metrics to monitor:
├── CPU utilization: Scale at 70%
├── Request latency: Scale at P95 > 100ms
├── Request rate: Scale at 80% capacity
└── Memory usage: Scale at 80%

Scaling behavior:
├── Scale up: +50% instances in 60s
├── Scale down: -10% instances in 5 minutes
└── Min: 10 instances, Max: 100 instances
```

Load Balancing:

```
Algorithm: Least connections + IP hash (for debugging)

Health checks:
├── HTTP GET /health every 10s
├── 2 consecutive failures → Remove from pool
├── 2 consecutive successes → Add back to pool
└── Drain connections before removal (30s grace period)
```

2. Vertical Scaling (Redis)

Redis Sizing:

```
Small (0-1K RPS):
- 2 GB RAM, 2 vCPU
- Single master + replica
- Cost: $100/month

Medium (1K-10K RPS):
- 16 GB RAM, 4 vCPU
- 3 masters + 3 replicas (sharded)
- Cost: $500/month

Large (10K-100K RPS):
- 64 GB RAM per node, 8 vCPU
- 10 masters + 10 replicas
- Cost: $2,000/month
```

3. Data Partitioning (Redis Cluster)

Sharding Strategy:

```
Consistent Hashing:
  hash(user_id) % num_shards → shard_id

Example with 3 shards:
  user_123 → hash → 42 → 42 % 3 = 0 → Shard 0
  user_456 → hash → 87 → 87 % 3 = 0 → Shard 0
  user_789 → hash → 14 → 14 % 3 = 2 → Shard 2

Benefits:
✓ Evenly distributed load
✓ Easy to add shards
✓ Minimal data movement
```

Redis Cluster Mode:

```
Hash Slots: 16,384 slots
Distribution:
├── Master 1: slots 0-5461
├── Master 2: slots 5462-10922
└── Master 3: slots 10923-16383

Auto-sharding:
  Redis automatically routes keys to correct shard
  Client libraries handle redirection
```

Performance Optimization Techniques

1. Connection Pooling

```
Problem: Creating new Redis connections is expensive (~10ms)

Solution: Connection Pool
├── Min connections: 10
├── Max connections: 50
├── Idle timeout: 60s
├── Connection reuse: ✓
└── Latency reduction: 10ms → 0.5ms
```

2. Pipelining

```
Without Pipelining:
  GET key1  → 0.5ms
  GET key2  → 0.5ms
  GET key3  → 0.5ms
  Total: 1.5ms

With Pipelining:
  PIPELINE {
    GET key1
    GET key2
    GET key3
  }
  Total: 0.6ms (60% faster)
```

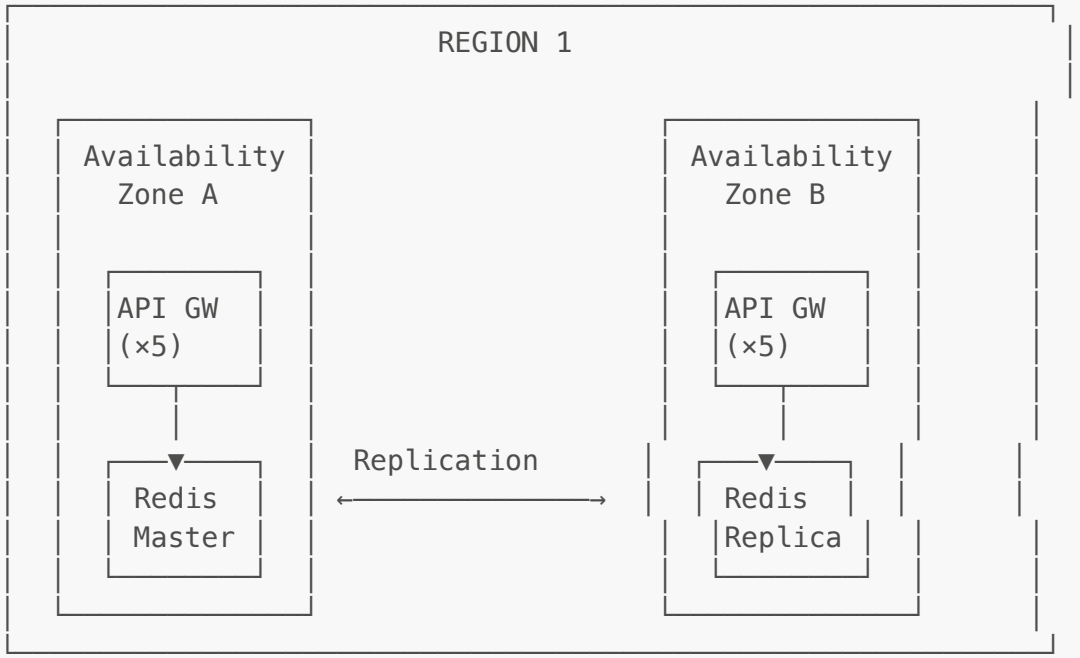
3. Caching Strategy

```
L1 Cache (In-Memory):
├── Rules cache: 10,000 rules
├── TTL: 5 minutes
├── Hit rate: 95%
├── Latency: 0.1ms
└── Eviction: LRU

L2 Cache (Redis):
├── Rules cache: 100,000 rules
├── TTL: 15 minutes
├── Hit rate: 99%
└── Latency: 0.5ms
```

```
L3 (PostgreSQL):
└── Source of truth
```

Fault Tolerance & Reliability

High Availability Architecture




```
Recovery Objectives:
├── RPO (Recovery Point Objective): <1 second
├── RTO (Recovery Time Objective): <30 seconds
└── Data Loss: Accept <100 requests during failover
```

Failure Modes & Handling

1. Redis Unavailable

```
Detection:
├── Health check every 1s
├── 3 consecutive failures → Trigger failover
└── Alert ops team

Response Options:

A. Fail Open (Recommended):
   ✓ Allow all requests
   ✓ Log failures for audit
   ✓ Resume normal operation when Redis recovers
   Risk: No rate limiting for ~30s
```

```
B. Fail Closed:
   ✗ Deny all requests
   ✗ Service outage
   Benefit: Maintain strict limits

C. Degraded Mode:
   ✓ Use local in-memory counters
   ✓ Less accurate but functional
   ✓ Sync when Redis recovers
```

2. Configuration Service Down

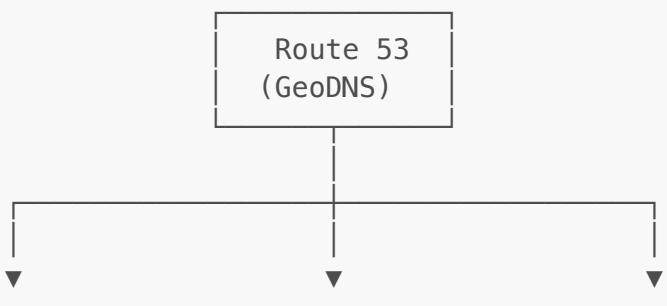
```
Impact: Cannot load new rules or user tiers

Mitigation:
├── Rules cached in API gateways (5 min TTL)
├── Stale data acceptable for rate limiting
├── Continue with cached rules until service recovers
└── No immediate impact on rate limiting operations
```

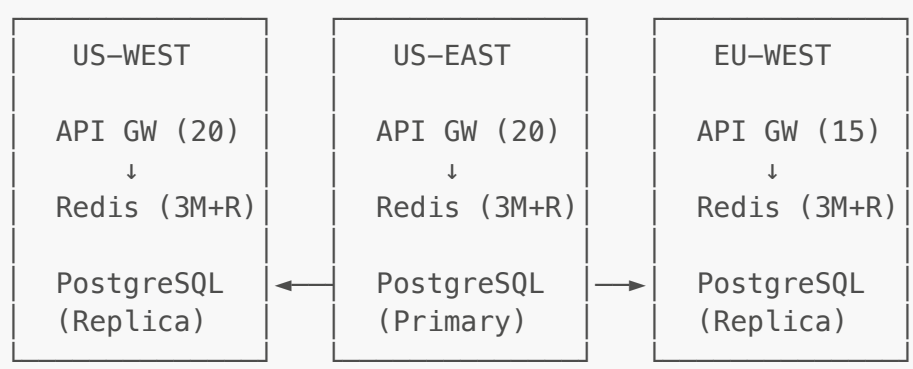
3. Network Partition

```
Symptom: API gateways can't reach Redis

Decision Tree:
├── If latency > 50ms:
│   └── Use cached counters (degraded mode)
├── If timeout (>1s):
│   └── Fail open (allow requests)
└── Log all decisions for audit
```

Multi-Region Architecture

Global Deployment Strategy






```
Traffic Distribution:
├── US-WEST: 40% (California, Pacific)
├── US-EAST: 35% (East Coast, South America)
└── EU-WEST: 25% (Europe, Middle East, Africa)
```

Consistency Model Options

Option 1: Regional Independence (Recommended)

```
Strategy: Each region operates independently

Rate Limit Distribution:
├── Global limit: 1000 req/hour
├── US-WEST quota: 400 req/hour
├── US-EAST quota: 350 req/hour
└── EU-WEST quota: 250 req/hour

Pros:
✓ Zero cross-region latency
✓ Regions isolated from each other's failures
✓ Simple to implement

Cons:
✗ User can potentially use quota in multiple regions
✗ Over-limit by ~20% if user distributes requests

Acceptable because:
- Users typically stay in one region
- 20% over-limit acceptable per requirements
- Regional failures don't cascade
```

Option 2: Global Synchronization

```
Strategy: Central Redis for all regions

Architecture:
```

```
  All regions → Central Redis (US-EAST)

Pros:
✓ Strict global limits
✓ Perfect accuracy

Cons:
✗ Cross-region latency (+50-200ms)
✗ Single point of failure
✗ Expensive (cross-region data transfer)

Use when:
- Strict accuracy required (financial APIs)
- Traffic is low (<1K RPS globally)
- Latency less critical
```

Option 3: Async Synchronization

```
Strategy: Sync counters every 1-5 seconds

Flow:
1. User hits US-WEST: 45 requests
2. Background job syncs to other regions
3. US-EAST Redis updated: 45 requests
4. User hits US-EAST: Sees 45 from counter

Pros:
✓ Better accuracy than regional independence
✓ Lower latency than global sync

Cons:
✗ Complex implementation
✗ Eventual consistency issues
✗ Race conditions during sync window
```

Region Failover Strategy

```
Normal Operation:
  User (California) → US-WEST → Local Redis

US-WEST Failure:
  User (California) → Route53 detects failure
                    → Redirect to US-EAST
                    → User quota starts fresh

Failover Implications:
├── User quota reset in new region
├── Can potentially double usage
```

```
├── Acceptable trade-off for availability
└── Monitor and alert on region failures

Recovery:
├── US-WEST comes back online
├── Gradual traffic shift (10% per minute)
├── Monitor for issues
└── Full traffic in ~10 minutes
```

Design Trade-offs

1. Accuracy vs Performance

Approach	Accuracy	Latency	Memory	Choice
Sliding Window Log	99.9%	2-5ms	High	High-value APIs
Sliding Window Counter	95-99%	1-2ms	Medium	**Default**
Fixed Window	90-95%	0.5ms	Low	MVP/Low-traffic
Token Bucket	95%	1ms	Low	Burst scenarios

Decision: Sliding Window Counter

- Sweet spot between accuracy and performance
- Handles 99% of use cases
- Can upgrade to Sliding Window Log for critical endpoints

2. Consistency vs Availability

```
CAP Theorem Applied to Rate Limiting:

Partition Tolerance (Required):
- Must work when networks fail
- Can't sacrifice this

Choose One:
┌─────────────────────────────────────┐
│ Consistency (Strict Limits)         │
│ - Reject at exactly limit           │
│ - May deny valid requests on failure│
│ - Lower availability                │
│                                     │
│ Trade-off: 99.9% availability       │
└─────────────────────────────────────┘

        vs

┌─────────────────────────────────────┐
│ Availability (Always Available)     │
│ - May allow 5-10% over-limit        │
```

```
| - Always serve requests                  |
| - Higher availability                    |
|                                          |
| Trade-off: 99.99% availability           |
└──────────────────────────────────────────┘

**Decision**: Favor Availability
- Rate limiting is not security-critical
- 5% over-limit acceptable
- Better user experience
```

3. Centralized vs Distributed State

```
Centralized (Redis):
Pros:
✓ Single source of truth
✓ Accurate counts
✓ Simple logic

Cons:
✗ Network latency
✗ Single point of failure (mitigated with clustering)

Distributed (In-Memory):
Pros:
✓ Zero network latency
✓ No external dependencies

Cons:
✗ Inconsistent across servers
✗ Lost on restart
✗ Hard to implement correctly

**Decision**: Centralized with caching
- Use Redis for counters (accuracy)
- Cache rules in-memory (performance)
- Best of both worlds
```

4. Pull vs Push for Rule Updates

```
Pull (Polling):
API Gateways poll PostgreSQL every 5 minutes

Pros:
✓ Simple implementation
✓ Predictable load

Cons:
```

```
✗ 5-minute delay for updates
✗ Unnecessary queries if no changes

Push (Event-Driven):
PostgreSQL → Kafka → API Gateways

Pros:
✓ Immediate updates
✓ No unnecessary queries

Cons:
✗ More complex
✗ Another service to manage (Kafka)

**Decision**: Pull for now, Push later
- 5-minute delay acceptable
- Simpler to implement and operate
- Can add push notifications later if needed
```

Real-World Case Studies

Case Study 1: Stripe API Rate Limiting

Scale: 100,000+ RPS globally

Architecture:

```
├── Algorithm: Token Bucket + Sliding Window
├── Multi-tier limits:
│   ├── Per API key: 100 reads/sec, 25 writes/sec
│   ├── Per endpoint: Different limits
│   └── Test vs Production: Separate limits
├── Storage: Redis Cluster (15 nodes)
└── Regions: 5 global regions
```

Key Design Decisions:

1. **Separate read/write limits**: Writes more expensive, stricter limits
2. **Burst allowance**: Token bucket allows legitimate bursts
3. **Graceful degradation**: Fail open on Redis issues
4. **Rich error messages**: Include retry-after, remaining quota

Lessons Learned:

- Communicate limits clearly in documentation
- Provide SDK libraries that respect limits
- Monitor P95/P99 of limit usage (identify power users)
- Allow temporary limit increases for events

Case Study 2: GitHub API

Scale: 50,000+ RPS

Architecture:

```
├── Algorithm: Fixed Window (1 hour)
├── Limits:
│   ├── Unauthenticated: 60 req/hour per IP
│   ├── Authenticated: 5,000 req/hour per user
│   └── GraphQL: Point-based system
├── Secondary limit: 100 req/minute (burst protection)
└── Storage: Redis + PostgreSQL
```

Key Design Decisions:

1. **GraphQL complexity scoring**: Each query costs points based on complexity
2. **Conditional requests**: 304 Not Modified doesn't count
3. **GitHub Apps**: Higher limits (15,000 req/hour)
4. **Secondary rate limit**: Prevents rapid bursts

Lessons Learned:

- Simple hourly window sufficient for most cases
- Secondary burst limit catches abuse
- Different auth methods need different limits
- Conditional requests improve UX without cost

Case Study 3: Twitter API

Scale: 200,000+ RPS

Architecture:

```
├── Algorithm: Fixed Window (15 minutes)
├── Multi-dimensional limits:
│   ├── User context: Per user
│   ├── App context: Per app
│   └── Endpoint-specific: Different for each endpoint
├── Storage: Distributed cache + Redis
└── Regions: 4 global regions
```

Key Design Decisions:

1. **15-minute windows**: Balance between accuracy and UX
2. **Per-endpoint limits**: Tweet posting stricter than reading
3. **Multiple rate limit types**: User and app limits combined

4. **Premium tiers**: Different limits for different access levels

Lessons Learned:

- Clear documentation of limits crucial
- Rate limits as product differentiator (premium tiers)
- Window resets at fixed intervals (predictable for users)
- Must handle high-profile events (millions of users simultaneously)

Case Study 4: AWS API Gateway

Scale: 1,000,000+ RPS

Architecture:

```
├── Algorithm: Token Bucket
├── Hierarchical limits:
│   ├── Account-level: 10,000 req/sec
│   ├── API-level: Configurable
│   ├── Stage-level: Per environment
│   └── Method-level: Per endpoint
├── Burst: 5,000 requests
└── Storage: Distributed in-memory + DynamoDB
```

Key Design Decisions:

1. **Token bucket for bursts**: AWS workloads often bursty
2. **Hierarchical limits**: Multiple layers of protection
3. **Per-stage configuration**: Dev/staging/prod different limits
4. **Usage plans**: Package limits with API keys

Lessons Learned:

- Burst capacity critical for serverless workloads
- Multiple limit tiers provide flexibility
- Integration with billing system (pay for higher limits)
- CloudWatch metrics for monitoring and alerting

Summary & Key Takeaways

Architecture Principles

1. **Stateless Application Layer**
    - API gateways should be stateless
    - Enables horizontal scaling
    - Simplifies deployment and operations

2. **Centralized State Management**
    - Use Redis for counters (fast, atomic)
    - Use PostgreSQL for rules (durable, queryable)
    - Cache aggressively for performance
3. **Favor Availability Over Strict Consistency**
    - Accept 5% over-limit
    - Fail open on errors
    - Better UX than false rejections
4. **Design for Failure**
    - Redis cluster with replication
    - Multiple availability zones
    - Graceful degradation strategies

Scaling Strategy

```
Growth Path:

Phase 1 (0-1K RPS): Single region, minimal setup
├── 3 API gateways
├── 1 Redis master + replica
└── Cost: $500/month

Phase 2 (1K-10K RPS): Regional scaling
├── 10-20 API gateways
├── Redis cluster (3 masters + replicas)
└── Cost: $5,000/month

Phase 3 (10K-100K RPS): Multi-region
├── 3 regions × 20 gateways each
├── Regional Redis clusters
├── Cross-region config replication
└── Cost: $30,000/month

Phase 4 (100K+ RPS): Global scale
├── 5+ regions
├── CDN integration
├── Custom optimizations
└── Cost: $100,000+/month
```

Decision Matrix

Choose Sliding Window Counter when:

- Building a new rate limiter
- Need balance of accuracy and performance

- Can tolerate ~5% over-limit
- This is the recommended default

Choose Sliding Window Log when:

- Need maximum accuracy
- Financial transactions or high-value operations
- Can afford higher memory usage

Choose Token Bucket when:

- Legitimate burst traffic expected
- Mobile apps, batch operations
- Variable request sizes

Choose Fixed Window when:

- MVP or prototype stage
- Very low traffic (<100 RPS)
- Simplicity more important than accuracy

Interview Success Tips

1. **Start with requirements** - Don't jump to solution
2. **Draw architecture diagram** - Visual communication is key
3. **Discuss trade-offs explicitly** - Show you understand nuances
4. **Consider failure scenarios** - What happens when things break?
5. **Estimate capacity** - Show you can do back-of-envelope math
6. **Reference real systems** - "Similar to how Stripe does it..."
7. **Think about operations** - Monitoring, alerting, debugging
8. **Iterate on design** - Start simple, add complexity as needed

Common Pitfalls to Avoid

❌ Over-engineering for day 1
❌ Ignoring the CAP theorem
❌ Forgetting about clock synchronization
❌ Not planning for failure scenarios
❌ Underestimating Redis performance
❌ Not considering hot key problems
❌ Forgetting about operational concerns
❌ Focusing too much on code details vs architecture

Further Reading

Essential Resources

Rate Limiting Concepts:

- "Rate Limiting Strategies" - Redis documentation
- "Distributed Rate Limiting" - Martin Kleppmann
- "API Rate Limiting Best Practices" - Kong API Gateway

System Design:

- "Designing Data-Intensive Applications" - Martin Kleppmann
- "System Design Interview" - Alex Xu
- "Building Microservices" - Sam Newman

Real-World Implementations:

- Stripe API Rate Limiting - stripe.com/docs/rate-limits
- GitHub REST API - docs.github.com/en/rest/overview/resources-in-the-rest-api
- AWS API Gateway Throttling - AWS documentation

Redis & Distributed Systems:

- "Redis in Action" - Josiah Carlson
- "Designing Distributed Systems" - Brendan Burns
- "Site Reliability Engineering" - Google SRE Book

Document Version: 1.0 (HLD Focus)
Last Updated: 2024
Focus: High-Level Design, Architecture, Scalability

<task_progress>

- ☑ Create HLD-focused rate limiter design document
- ☑ Cover problem statement and requirements
- ☑ Design high-level architecture
- ☑ Explain core components
- ☑ Compare algorithms at conceptual level
- ☑ Detail distributed system challenges
- ☑ Cover scalability and performance
- ☑ Discuss fault tolerance
- ☑ Add multi-region architecture
- ☑ Explain design trade-offs
- ☑ Include real-world case studies
- ☑ Add summary and key takeaways
  </task_progress>