Rate Limiter System Design (Interview Deep Dive)

Table of Contents

Interview Approach

Step-by-Step Framework

In a system design interview, follow this structured approach:

```
1. CLARIFY REQUIREMENTS (5-10 min)
   ↓
2. ESTABLISH SCOPE (2-3 min)
   ↓
3. CAPACITY ESTIMATION (5 min)
   ↓
4. API DESIGN (3-5 min)
   ↓
5. HIGH-LEVEL DESIGN (10-15 min)
   ↓
6. DETAILED DESIGN (15-20 min)
   ↓
7. IDENTIFY BOTTLENECKS (5 min)
   ↓
8. SCALING & OPTIMIZATION (5-10 min)
```

Key Points to Address

✅ **DO:**

- Ask clarifying questions before jumping to solutions
- Start simple, then iterate
- Discuss trade-offs explicitly
- Think out loud
- Draw diagrams
- Consider failure scenarios
- Mention monitoring and alerting

❌ **DON'T:**

- Jump into implementation details immediately
- Assume requirements without asking
- Ignore non-functional requirements
- Forget about edge cases
- Design for infinite scale from day one

Problem Clarification

Essential Questions to Ask

1. Scope and Context

- **Q: What type of rate limiter are we building?**
  - Client-side vs. Server-side?
  - Middleware vs. Standalone service?
  - For internal services or public APIs?
- **Q: Where does this rate limiter sit in the architecture?**
  - API Gateway level?
  - Application level?
  - Load balancer level?
  - CDN level?

2. Rate Limiting Rules

- **Q: What entities should we rate limit?**
  - User ID?
  - IP address?
  - API key?
  - Combination of multiple factors?
- **Q: What rate limiting rules do we need to support?**
  - Fixed rate (100 requests per minute)?

- Burst allowance?
  - Different tiers (free, premium, enterprise)?
  - Different limits per endpoint?

3. Scale and Performance

- **Q: What's the expected scale?**
  - How many requests per second?
  - How many users?
  - Global or regional?
- **Q: What's the acceptable latency?**
  - <1ms? <10ms? <100ms?
  - Can we sacrifice some accuracy for performance?

4. Consistency and Availability

- **Q: How strict should the rate limiting be?**
  - Hard limit (never exceed)?
  - Soft limit (best effort)?
  - What if rate limiter is down?
- **Q: Should the system be distributed?**
  - Single datacenter or multi-region?
  - Consistency vs. Availability trade-off?

5. Behavior and Features

- **Q: What happens when limit is exceeded?**
  - Return 429 error?
  - Queue requests?
  - Different actions for different scenarios?
- **Q: Do we need to support dynamic rule updates?**
  - Can rules change in real-time?
  - How quickly should changes propagate?

Example Interview Dialog

Interviewer: "Design a rate limiting system."

Candidate: "Great! Let me clarify the requirements first. Are we building this for a specific company's API gateway, or is it a general-purpose rate limiter?"

Interviewer: "Let's design it for a large-scale API company like Stripe or Twitter."

Candidate: "Understood. A few more questions:

1. What scale are we targeting? Let's say requests per second?
2. Should we support multiple rate limiting strategies like per-user, per-IP, or both?
3. What's our consistency requirement - is it okay if the rate limiter allows a few extra requests occasionally?
4. Should this be distributed across multiple data centers globally?"

Interviewer: "Good questions. Let's say 10,000 requests/second, support both per-user and per-IP limiting, we can tolerate some inaccuracy (<5% over-limit), and yes, distribute globally."

Candidate: "Perfect. Based on these requirements, let me start with capacity estimation..."

Requirements Deep Dive

Functional Requirements (Detailed)

1. Core Rate Limiting

- **Enforce rate limits** based on configurable rules
  - Per user/customer ID
  - Per IP address
  - Per API key/access token
  - Per endpoint/resource
  - Combination rules (e.g., per-user-per-endpoint)

2. Multiple Time Windows

- Support various time windows:
  - Per second (1s)
  - Per minute (60s)
  - Per hour (3600s)
  - Per day (86400s)
  - Custom windows

3. Multiple Rate Limit Tiers

```
Free Tier:     100 requests/hour
Basic Tier:    1,000 requests/hour
Premium Tier:  10,000 requests/hour
Enterprise:    Custom limits
```

4. Response Handling

- Return appropriate HTTP status codes (429)
- Include rate limit headers

- Provide retry-after information
- Clear error messages

5. Rule Management

- Create/update/delete rate limit rules
- Support rule precedence
- Enable/disable rules dynamically
- Version control for rules

6. Whitelist/Blacklist

- Whitelist certain IPs/users (no limits)
- Blacklist abusive IPs/users (always reject)
- Support CIDR ranges

Non-Functional Requirements (Detailed)

1. Performance Requirements

```
Latency:
  - P50: < 1ms
  - P95: < 5ms
  - P99: < 10ms

Throughput:
  - Support 10,000+ requests/second
  - Handle traffic spikes (10x normal)
```

2. Availability Requirements

```
SLA: 99.99% uptime (4.32 minutes downtime/month)
```

3. Scalability Requirements

- Horizontal scaling (add more servers)
- Handle 10x growth without architecture changes
- Support multi-region deployment

4. Consistency Requirements

- **Eventual consistency acceptable** for distributed systems
- Accuracy: <5% over-limit allowed
- No under-limiting (don't reject valid requests)

5. Durability Requirements

- Rate limit rules should survive service restarts
- Configuration changes should be persistent
- Audit log for rule changes

6. Security Requirements

- Prevent DDoS attacks
- Protect against rate limiter bypass attempts
- Secure rule management API

Capacity Estimation

Back-of-the-Envelope Calculations

Assumptions

```
Total users: 100 million
Daily active users (DAU): 10 million
Average requests per user per day: 50
Peak to average ratio: 3x
```

Request Volume

```
Daily requests = 10M users × 50 requests = 500M requests/day

Average RPS = 500M / 86400 seconds ≈ 5,800 requests/second

Peak RPS = 5,800 × 3 = 17,400 requests/second
```

Storage Requirements

1. Counter Storage (Redis)

For fixed window approach:

```
Per user per window: 1 key = 8 bytes (counter)
Time windows: second, minute, hour, day = 4 windows
Total per user = 4 × 8 bytes = 32 bytes

For 10M DAU:
Memory = 10M × 32 bytes = 320 MB
```

Add overhead for Redis (keys, metadata): ~2x

```
Total memory ≈ 640 MB
```

2. Configuration Storage

```
Rule configuration:
- User ID: 16 bytes
- Limit value: 4 bytes
- Window size: 4 bytes
- Metadata: 20 bytes
Total per rule: ~50 bytes

For 100M users with 2 rules each:
Storage = 100M × 2 × 50 bytes = 10 GB
```

3. Request Log Storage (Sliding Window Log)

```
Per request:
- Timestamp: 8 bytes
- Request ID: 16 bytes
Total: 24 bytes

For 1 hour window, 10M users, 50 req/hour:
Storage = 10M × 50 × 24 bytes = 12 GB/hour
```

Bandwidth Requirements

```
Rate limit check request:
- User ID: 16 bytes
- Endpoint: 50 bytes
- Headers: 100 bytes
Total: ~200 bytes

Rate limit check response:
- Status: 1 byte
- Headers: 100 bytes
Total: ~100 bytes

Total per request: 300 bytes

At peak (17,400 RPS):
Bandwidth = 17,400 × 300 bytes ≈ 5.2 MB/second ≈ 42 Mbps
```

Server Requirements

Assuming each server handles 1,000 RPS:

```
Servers needed = 17,400 RPS / 1,000 = 18 servers

Add redundancy (2x): 36 servers
Add headroom for growth (2x): 72 servers
```

Redis Cluster Requirements

```
Memory per Redis instance: 16 GB
Assuming 640 MB for counters + headroom: 3-5 Redis instances

For high availability:
- Master-replica setup: 10 instances (5 masters, 5 replicas)
```

Cost Estimation (AWS Example)

```
Rate Limiter Servers:
- 72 × m5.large ($0.096/hour)
- 72 × $0.096 × 24 × 30 = $4,976/month

Redis Cluster:
- 10 × r5.large ($0.126/hour)
- 10 × $0.126 × 24 × 30 = $907/month

Load Balancers:
- 3 × ALB = ~$100/month

Total: ~$6,000/month
```

API Design

REST API Endpoints

1. Rate Limit Check (Primary API)

```
POST /v1/ratelimit/check
Content-Type: application/json

{
  "user_id": "user_12345",
```

```
  "api_key": "sk_live_abc123",
  "ip_address": "192.168.1.1",
  "endpoint": "/api/payments",
  "method": "POST"
}

Response (Allowed):
HTTP/1.1 200 OK
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 742
X-RateLimit-Reset: 1640995200

{
  "allowed": true,
  "limit": 1000,
  "remaining": 742,
  "reset_at": 1640995200
}

Response (Rate Limited):
HTTP/1.1 429 Too Many Requests
X-RateLimit-Limit: 1000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1640995200
Retry-After: 60

{
  "allowed": false,
  "error": {
    "code": "RATE_LIMIT_EXCEEDED",
    "message": "Rate limit exceeded. Try again in 60 seconds.",
    "limit": 1000,
    "retry_after": 60,
    "reset_at": 1640995200
  }
}
```

2. Rule Management APIs

Create Rule:

```
POST /v1/admin/rules
Authorization: Bearer admin_token

{
  "name": "api_payment_limit",
  "description": "Rate limit for payment API",
  "selector": {
    "user_tier": "premium",
    "endpoint": "/api/payments"
```

```
  },
  "limit": 1000,
  "window": "1h",
  "action": "reject"
}

Response:
HTTP/1.1 201 Created

{
  "rule_id": "rule_abc123",
  "created_at": "2024-01-01T00:00:00Z"
}
```

Get Rule:

```
GET /v1/admin/rules/{rule_id}
Authorization: Bearer admin_token

Response:
{
  "rule_id": "rule_abc123",
  "name": "api_payment_limit",
  "status": "active",
  "limit": 1000,
  "window": "1h",
  "created_at": "2024-01-01T00:00:00Z",
  "updated_at": "2024-01-01T00:00:00Z"
}
```

Update Rule:

```
PUT /v1/admin/rules/{rule_id}
Authorization: Bearer admin_token

{
  "limit": 2000,
  "status": "active"
}
```

Delete Rule:

```
DELETE /v1/admin/rules/{rule_id}
Authorization: Bearer admin_token
```

List Rules:

```
GET /v1/admin/rules?page=1&limit=50
Authorization: Bearer admin_token

Response:
{
  "rules": [...],
  "total": 150,
  "page": 1,
  "page_size": 50
}
```

3. Statistics and Monitoring APIs

Get User Statistics:

```
GET /v1/stats/users/{user_id}?window=1h
Authorization: Bearer admin_token

Response:
{
  "user_id": "user_12345",
  "window": "1h",
  "total_requests": 856,
  "allowed_requests": 856,
  "rejected_requests": 0,
  "current_limit": 1000,
  "remaining": 144
}
```

Get Global Statistics:

```
GET /v1/stats/global?from=2024-01-01T00:00:00Z&to=2024-01-01T23:59:59Z

Response:
{
  "total_requests": 500000000,
  "allowed_requests": 485000000,
  "rejected_requests": 15000000,
  "rejection_rate": 0.03,
  "top_limited_users": [...],
  "top_limited_endpoints": [...]
}
```

4. Health Check APIs

```
GET /health

Response:
{
  "status": "healthy",
  "version": "1.0.0",
  "redis": {
    "status": "connected",
    "latency_ms": 0.5
  },
  "uptime_seconds": 86400
}
```

gRPC API (Alternative)

```
syntax = "proto3";

service RateLimiter {
  rpc CheckRateLimit(RateLimitRequest) returns (RateLimitResponse);
  rpc GetUserStats(UserStatsRequest) returns (UserStatsResponse);
}

message RateLimitRequest {
  string user_id = 1;
  string api_key = 2;
  string ip_address = 3;
  string endpoint = 4;
  string method = 5;
}

message RateLimitResponse {
  bool allowed = 1;
  int64 limit = 2;
  int64 remaining = 3;
  int64 reset_at = 4;
  string error_message = 5;
}
```

High-Level Design

System Architecture Diagram

```
┌──────────────────────────────────────────────────────────────────┐
│                           CLIENTS                                │
│  (Web, Mobile, Third-party APIs)                                 │
└──────────────────────────────────────────────────────────────────┘
                            │
                            ▼
┌──────────────────────────────────────────────────────────────────┐
│                    CDN / Edge Locations                          │
│  - Geographic distribution                                       │
│  - Initial rate limiting (IP-based)                              │
└──────────────────────────────────────────────────────────────────┘
                            │
                            ▼
┌──────────────────────────────────────────────────────────────────┐
│                    Load Balancer (L7)                            │
│  - SSL Termination                                               │
│  - Health checks                                                 │
│  - Distribute to API Gateway                                     │
└──────────────────────────────────────────────────────────────────┘
                            │
                            ▼
┌──────────────────────────────────────────────────────────────────┐
│                    API Gateway Cluster                           │
│  ┌────────────────────────────────────────────────────────────┐  │
│  │               Rate Limiter Middleware                      │  │
│  │  - Request interception                                    │  │
│  │  - User identification                                     │  │
│  │  - Rate limit check                                        │  │
│  └────────────────────────────────────────────────────────────┘  │
│                   │                                              │
│  ┌────────────────────────────────────────────────────────────┐  │
│  │               Rules Engine                                 │  │
│  │  - Load rate limit rules                                   │  │
│  │  - Rule matching and evaluation                            │  │
│  │  - Cache rules locally                                     │  │
│  └────────────────────────────────────────────────────────────┘  │
└──────────────────────────────────────────────────────────────────┘
                  │
     ┌────────────┼────────────┐
     │            │            │
     ▼            ▼            ▼
┌──────────┐ ┌──────────┐ ┌──────────┐
│  Redis   │ │  Redis   │ │  Redis   │
│  Master 1│ │  Master 2│ │  Master 3│
│          │ │          │ │          │
│[Counters]│ │[Counters]│ │[Counters]│
└──────────┘ └──────────┘ └──────────┘
     │            │            │
     ▼            ▼            ▼
┌──────────┐ ┌──────────┐ ┌──────────┐
│  Redis   │ │  Redis   │ │  Redis   │
│  Replica 1│ │  Replica 2│ │  Replica 3│
```

```
┌─────────────────────────────────────────────────────────────────┐
│                  Configuration Service                            │
│  - Store rate limit rules                                         │
│  - Distribute rule updates                                        │
│  - Rule versioning                                                │
└─────────────────────────────────────────────────────────────────┘
                            │
                            │
                            ▼
                    ┌──────────────┐
                    │  PostgreSQL  │
                    │  (Rules DB)  │
                    └──────────────┘

┌─────────────────────────────────────────────────────────────────┐
│                  Monitoring & Logging                             │
│  - Metrics (Prometheus)                                           │
│  - Logs (ELK Stack)                                               │
│  - Alerting (PagerDuty)                                         │
│  - Dashboards (Grafana)                                         │
└─────────────────────────────────────────────────────────────────┘

                            │
                            ▼
                    ┌──────────────┐
                    │   Backend    │
                    │   Services   │
                    └──────────────┘
```

Request Flow

```
Step-by-Step Request Flow:

1. Client sends request → Load Balancer
   - SSL termination
   - Route to API Gateway

2. API Gateway receives request
   - Extract user identifier (API key, user ID, IP)
   - Extract endpoint information

3. Rate Limiter Middleware intercepts
   - Check local rule cache
   - Identify applicable rules

4. Query Redis for current count
   - Key: "ratelimit:user_123:endpoint_payments:window_1640995200"
   - INCR operation (atomic)
```

```
5. Evaluate against limit
   IF count <= limit:
     - Allow request
     - Add rate limit headers
     - Forward to backend
   ELSE:
     - Reject request
     - Return 429 status
     - Include retry-after header

6. Response flows back through gateway
   - Additional headers added
   - Logged for monitoring

7. Client receives response
```

Design Evolution

Version 1: Single Server (MVP)

```
┌──────────┐
│  Client  │
└────┬─────┘
     │
     ▼
┌──────────────┐
│  Web Server  │
│  + In-Memory │
│    Counter   │
└──────────────┘
```

Pros:

- Simple to implement
- Low latency
- No external dependencies

Cons:

- Not distributed
- Lost state on restart
- Single point of failure

Version 2: Centralized Storage

```
┌──────────┐     ┌──────────┐
│  Server  │────▶│  Server  │
```

```
└──────────┘    └──────────┘
     │               │
     └───────┬───────┘
             │
             ▼
        ┌────────┐
        │ Redis  │
        └────────┘
```

Pros:

- Distributed across servers
- Persistent state
- Consistent counts

Cons:

- Network latency
- Redis is single point of failure

Version 3: Redis Cluster (Production)

```
┌──────────┐    ┌──────────┐    ┌──────────┐
│  Server  │    │  Server  │    │  Server  │
└──────────┘    └──────────┘    └──────────┘
     │               │               │
     └───────────────┼───────────────┘
                     │
                     ▼
        ┌─────────────────────┐
        │   Redis Cluster     │
        │ (Multiple Masters)  │
        │ (Multiple Replicas) │
        └─────────────────────┘
```

Pros:

- Highly available
- Horizontally scalable
- No single point of failure

This is the recommended production setup.

Rate Limiting Algorithms

Detailed Algorithm Comparison

Algorithm	Accuracy	Memory	Latency	Burst	Best For

Algorithm	Accuracy	Memory	Latency	Burst	Best For
Token Bucket	Good	Low	Low	✅ Yes	APIs needing burst capacity
Leaky Bucket	Good	Medium	Low	❌ No	Streaming, queue-based systems
Fixed Window	Fair	Very Low	Very Low	⚠ Boundary issue	Simple use cases, low traffic
Sliding Window Log	Excellent	High	Medium	✅ Yes	High-value APIs, strict limits
Sliding Window Counter	Very Good	Low	Low	✅ Yes	**Recommended for production**

1. Token Bucket (Detailed)

Concept

Think of a bucket that:

- Holds tokens (maximum capacity)
- Gets refilled at a constant rate
- Each request takes 1 token
- Request denied if no tokens available

Visual Representation

```
Time: 0s     1s     2s     3s     4s     5s
      ┌──┐   ┌──┐   ┌──┐   ┌──┐   ┌──┐   ┌──┐
      │10│   │9 │   │8 │   │9 │   │10│   │9 │
      └──┘   └──┘   └──┘   └──┘   └──┘   └──┘
       ↓      ↓      ↓      ↑      ↓      ↓
      Req    Req    Req    Refill Req    Req
```

Implementation (Production-Grade)

```python
import time
import threading
from typing import Optional

class TokenBucket:
    def __init__(self, capacity: int, refill_rate: float):
        """
        Args:
            capacity: Maximum number of tokens
```

```
            refill_rate: Tokens added per second
        """
        self.capacity = capacity
        self.refill_rate = refill_rate
        self.tokens = float(capacity)
        self.last_refill = time.time()
        self.lock = threading.Lock()

    def allow_request(self, tokens_requested: int = 1) -> bool:
        """
        Check if request is allowed and consume tokens if yes.

        Args:
            tokens_requested: Number of tokens to consume

        Returns:
            True if request allowed, False otherwise
        """
        with self.lock:
            self._refill()

            if self.tokens >= tokens_requested:
                self.tokens -= tokens_requested
                return True
            return False

    def _refill(self) -> None:
        """Refill tokens based on elapsed time."""
        now = time.time()
        elapsed = now - self.last_refill

        # Calculate tokens to add
        tokens_to_add = elapsed * self.refill_rate

        # Add tokens without exceeding capacity
        self.tokens = min(self.capacity, self.tokens + tokens_to_add)

        self.last_refill = now

    def get_available_tokens(self) -> float:
        """Get current available tokens."""
        with self.lock:
            self._refill()
            return self.tokens

    def wait_time_for_tokens(self, tokens_needed: int) -> float:
        """
        Calculate wait time until tokens are available.

        Returns:
            Seconds to wait, 0 if tokens already available
        """
        with self.lock:
```

```
            self._refill()

            if self.tokens >= tokens_needed:
                return 0.0

            tokens_deficit = tokens_needed - self.tokens
            return tokens_deficit / self.refill_rate

# Example usage
bucket = TokenBucket(capacity=10, refill_rate=1.0)  # 10 tokens, refill
1/sec

for i in range(15):
    if bucket.allow_request():
        print(f"Request {i+1}: ALLOWED")
    else:
        wait_time = bucket.wait_time_for_tokens(1)
        print(f"Request {i+1}: DENIED (wait {wait_time:.2f}s)")
    time.sleep(0.5)
```

Redis Implementation

```
-- Token bucket implementation in Redis Lua
local key = KEYS[1]
local capacity = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])
local requested = tonumber(ARGV[3])
local now = tonumber(ARGV[4])

-- Get current state
local tokens_key = key .. ":tokens"
local timestamp_key = key .. ":timestamp"

local last_tokens = tonumber(redis.call("GET", tokens_key))
if not last_tokens then
    last_tokens = capacity
end

local last_timestamp = tonumber(redis.call("GET", timestamp_key))
if not last_timestamp then
    last_timestamp = now
end

-- Calculate refill
local time_passed = math.max(0, now - last_timestamp)
local tokens_to_add = time_passed * refill_rate
local current_tokens = math.min(capacity, last_tokens + tokens_to_add)

-- Check if we can allow the request
local allowed = current_tokens >= requested
```

```lua
local new_tokens = current_tokens

if allowed then
    new_tokens = current_tokens - requested
end

-- Update Redis
redis.call("SET", tokens_key, new_tokens)
redis.call("SET", timestamp_key, now)
redis.call("EXPIRE", tokens_key, 3600)
redis.call("EXPIRE", timestamp_key, 3600)

-- Return result
return {allowed and 1 or 0, math.floor(new_tokens)}
```

Pros and Cons

Advantages:

1. **Allows bursts**: User can consume tokens quickly if available
2. **Smooth refill**: Natural traffic flow
3. **Simple**: Easy to understand and implement
4. **Memory efficient**: Only stores token count and timestamp

Disadvantages:

1. **Not distributed-friendly**: Hard to sync across servers
2. **Potential gaming**: Users can time requests to maximize burst
3. **Parameter tuning**: Finding right capacity and rate requires testing

When to use:

- APIs that need to allow legitimate burst traffic
- Systems where occasional over-limit is acceptable
- Services with variable request sizes (can request multiple tokens)

2. Fixed Window Counter (Detailed)

Concept

```
Window 1         Window 2         Window 3
[00:00-01:00]    [01:00-02:00]    [02:00-03:00]

  Counter: 0       Counter: 0       Counter: 0
  Limit: 100       Limit: 100       Limit: 100

  +1, +1, +1...    +1, +1...        +1...

  At 01:00: Counter resets to 0
```

The Boundary Problem

```
Timeline:
  00:59:30 ──────────── 01:00:00 ──────────── 01:00:30
  Window 1                 |          Window 2
                       Boundary

Problem:
- 50 requests at 00:59:45 (Window 1, allowed)
- 50 requests at 01:00:05 (Window 2, allowed)
- Total: 100 requests in 20 seconds (5x the limit!)

Expected: 100 requests/hour
Actual:   100 requests/20 seconds
```

Implementation

```python
import time
from collections import defaultdict
from typing import Dict

class FixedWindowCounter:
    def __init__(self, limit: int, window_seconds: int):
        self.limit = limit
        self.window_seconds = window_seconds
        self.counters: Dict[str, int] = defaultdict(int)

    def allow_request(self, user_id: str) -> bool:
        """Check if request is allowed."""
        now = time.time()
        window_start = int(now / self.window_seconds) *
self.window_seconds
        key = f"{user_id}:{window_start}"

        # Cleanup old windows
        self._cleanup_old_windows()

        count = self.counters[key]
        if count < self.limit:
            self.counters[key] += 1
            return True
        return False

    def _cleanup_old_windows(self) -> None:
        """Remove counters from old windows."""
        now = time.time()
        current_window = int(now / self.window_seconds) *
```

```
self.window_seconds

        keys_to_delete = [
            k for k in self.counters.keys()
            if int(k.split(':')[1]) < current_window
        ]

        for key in keys_to_delete:
            del self.counters[key]
```

Redis Implementation

```
# Fixed window with Redis
# Key format: ratelimit:user_123:window_1640995200

MULTI
INCR ratelimit:user_123:window_1640995200
EXPIRE ratelimit:user_123:window_1640995200 60
EXEC

# Check if count exceeds limit in application code
```

Lua Script (Atomic):

```lua
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])

local current = redis.call('INCR', key)
if current == 1 then
    redis.call('EXPIRE', key, window)
end

if current > limit then
    return 0  -- Rejected
else
    return 1  -- Allowed
end
```

3. Sliding Window Log (Detailed)

Concept

Store every request timestamp and count requests in the sliding window.

```
Current time: 10:30:45
Window: 1 hour
Look back to: 09:30:45

Stored timestamps:
09:31:20 ✓ (within window)
09:45:10 ✓ (within window)
10:15:30 ✓ (within window)
10:28:00 ✓ (within window)
09:20:00 ✗ (outside window, remove)

Count = 4 requests in last hour
```

Implementation

```python
import time
from collections import defaultdict
from typing import List, Dict
import bisect

class SlidingWindowLog:
    def __init__(self, limit: int, window_seconds: int):
        self.limit = limit
        self.window_seconds = window_seconds
        # Store sorted list of timestamps per user
        self.logs: Dict[str, List[float]] = defaultdict(list)

    def allow_request(self, user_id: str) -> bool:
        """Check if request is allowed."""
        now = time.time()
        cutoff = now - self.window_seconds

        # Get user's request log
        user_log = self.logs[user_id]

        # Remove timestamps outside window using binary search
        # Find index of first timestamp >= cutoff
        idx = bisect.bisect_left(user_log, cutoff)
        user_log[:] = user_log[idx:]

        # Check if we're under limit
        if len(user_log) < self.limit:
            # Add new timestamp (maintains sorted order)
            user_log.append(now)
            return True
        return False

    def get_request_count(self, user_id: str) -> int:
        """Get current request count for user."""
```

```
        now = time.time()
        cutoff = now - self.window_seconds
        user_log = self.logs[user_id]

        # Count requests in window
        idx = bisect.bisect_left(user_log, cutoff)
        return len(user_log) - idx
```

Redis Implementation (Sorted Set)

```
# Add request timestamp
ZADD ratelimit:user_123 1640995200.5 "req_uuid_1"

# Remove old entries
ZREMRANGEBYSCORE ratelimit:user_123 0 1640991600

# Count requests in window
ZCARD ratelimit:user_123

# Set expiry
EXPIRE ratelimit:user_123 3600
```

Lua Script:

```lua
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local now = tonumber(ARGV[3])
local request_id = ARGV[4]

-- Remove old entries
local cutoff = now - window
redis.call('ZREMRANGEBYSCORE', key, 0, cutoff)

-- Count current requests
local count = redis.call('ZCARD', key)

if count < limit then
    -- Add new request
    redis.call('ZADD', key, now, request_id)
    redis.call('EXPIRE', key, window)
    return 1  -- Allowed
else
    return 0  -- Rejected
end
```

4. Sliding Window Counter (Hybrid - Recommended)

Concept

Estimate current window count using weighted average of previous and current fixed windows.

```
Previous Window    Current Window
[00:00-01:00]     [01:00-02:00]
Count: 80         Count: 30

Current time: 01:30 (halfway through current window)

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

Visual Representation

```
Previous Window (100 requests)     Current Window (60 requests)
├───────────────────────────┤├───────────────────────────┤
00:00                     01:00                       02:00
                                      ↑
                                 01:45 (now)

Elapsed in current window: 45 minutes (75%)
Weight for previous window: 25%

Estimated count = 60 + (100 × 0.25) = 60 + 25 = 85 requests
```

Implementation

```python
import time
from collections import defaultdict
from typing import Dict, Tuple

class SlidingWindowCounter:
    def __init__(self, limit: int, window_seconds: int):
        self.limit = limit
        self.window_seconds = window_seconds
        # key -> (count, window_start)
        self.counters: Dict[str, Dict[int, int]] = defaultdict(dict)

    def allow_request(self, user_id: str) -> Tuple[bool, int]:
        """
        Check if request is allowed.

        Returns:
            (allowed, remaining_quota)
        """
        now = time.time()
```

```
        current_window_start = int(now / self.window_seconds) *
self.window_seconds
        previous_window_start = current_window_start -
self.window_seconds

        # Calculate position in current window (0.0 to 1.0)
        elapsed_in_current = now - current_window_start
        current_window_position = elapsed_in_current /
self.window_seconds

        # Weight for previous window (decreases as we move through
current window)
        previous_window_weight = 1.0 - current_window_position

        # Get counts
        user_windows = self.counters[user_id]
        current_count = user_windows.get(current_window_start, 0)
        previous_count = user_windows.get(previous_window_start, 0)

        # Calculate weighted count
        estimated_count = current_count + (previous_count *
previous_window_weight)

        # Check limit
        if estimated_count < self.limit:
            user_windows[current_window_start] = current_count + 1
            remaining = int(self.limit - estimated_count - 1)

            # Cleanup old windows
            self._cleanup_old_windows(user_id, previous_window_start)

            return True, max(0, remaining)

        return False, 0

    def _cleanup_old_windows(self, user_id: str, keep_from: int) ->
None:
        """Remove windows older than previous window."""
        user_windows = self.counters[user_id]
        old_windows = [w for w in user_windows.keys() if w < keep_from]

        for window in old_windows:
            del user_windows[window]

# Example usage
limiter = SlidingWindowCounter(limit=10, window_seconds=60)

for i in range(20):
    allowed, remaining = limiter.allow_request("user_123")
    print(f"Request {i+1}: {'ALLOWED' if allowed else 'DENIED'}
(Remaining: {remaining})")
    time.sleep(2)
```

Redis Implementation with Lua

```lua
local key_prefix = KEYS[1]
local limit = tonumber(ARGV[1])
local window = tonumber(ARGV[2])
local now = tonumber(ARGV[3])

-- Calculate windows
local current_window = math.floor(now / window) * window
local previous_window = current_window - window

-- Get counts
local current_key = key_prefix .. ":" .. current_window
local previous_key = key_prefix .. ":" .. previous_window

local current_count = tonumber(redis.call('GET', current_key)) or 0
local previous_count = tonumber(redis.call('GET', previous_key)) or 0

-- Calculate weights
local elapsed_in_current = now - current_window
local current_window_position = elapsed_in_current / window
local previous_weight = 1.0 - current_window_position

-- Estimate count
local estimated_count = current_count + (previous_count *
previous_weight)

if estimated_count < limit then
    -- Increment current window
    redis.call('INCR', current_key)
    redis.call('EXPIRE', current_key, window * 2)

    local remaining = math.max(0, limit - estimated_count - 1)
    return {1, math.floor(remaining)}  -- Allowed
else
    return {0, 0}  -- Rejected
end
```

5. Leaky Bucket (Detailed)

Concept

Requests enter a queue and are processed at a fixed rate. Think of it as water leaking from a bucket at a constant rate.

```
Incoming Requests (variable rate)
         ↓
```

```
    ┌───────┐
    │ ● ● ● │   ← Queue (bucket)
    │ ● ● ● │
    │ ● ●   │
    └───┬───┘
        ↓ (leak at constant rate)
   Process 1 request/second
```

Implementation

```python
import time
import threading
from collections import deque
from typing import Optional

class LeakyBucket:
    def __init__(self, capacity: int, leak_rate: float):
        """
        Args:
            capacity: Maximum queue size
            leak_rate: Requests processed per second
        """
        self.capacity = capacity
        self.leak_rate = leak_rate
        self.queue = deque()
        self.last_leak = time.time()
        self.lock = threading.Lock()

    def allow_request(self, request_id: str) -> bool:
        """
        Try to add request to queue.

        Returns:
            True if request accepted, False if queue full
        """
        with self.lock:
            self._leak()

            if len(self.queue) < self.capacity:
                self.queue.append((request_id, time.time()))
                return True
            return False

    def _leak(self) -> None:
        """Process (remove) requests at constant rate."""
        now = time.time()
        elapsed = now - self.last_leak

        # Calculate how many requests to process
        requests_to_process = int(elapsed * self.leak_rate)
```

```
        # Remove processed requests
        for _ in range(min(requests_to_process, len(self.queue))):
            self.queue.popleft()

        if requests_to_process > 0:
            self.last_leak = now

    def get_queue_size(self) -> int:
        """Get current queue size."""
        with self.lock:
            self._leak()
            return len(self.queue)
```

When to use Leaky Bucket:

- When you need to protect downstream services with strict rate limits
- For job processing systems
- When smoothing bursty traffic is more important than responsiveness

---

Detailed Component Design

1. Rate Limiter Middleware

```
from flask import Flask, request, jsonify, g
from functools import wraps
import redis
import time
from typing import Optional, Tuple

app = Flask(__name__)
redis_client = redis.Redis(host='localhost', port=6379,
decode_responses=True)

class RateLimiterMiddleware:
    def __init__(self, redis_client):
        self.redis = redis_client
        self.rules_cache = {}  # Cache rules locally
        self.cache_ttl = 300  # 5 minutes

    def get_rate_limit_rule(self, user_id: str, endpoint: str) ->
Optional[dict]:
        """
        Get applicable rate limit rule from cache or database.

        Priority:
        1. User-specific + endpoint-specific
        2. User-specific
        3. Endpoint-specific
```

```
        4. Global default
        """
        cache_key = f"rule:{user_id}:{endpoint}"

        # Check cache
        if cache_key in self.rules_cache:
            rule, cached_at = self.rules_cache[cache_key]
            if time.time() - cached_at < self.cache_ttl:
                return rule

        # Query rules in order of specificity
        rule = (
            self._get_user_endpoint_rule(user_id, endpoint) or
            self._get_user_rule(user_id) or
            self._get_endpoint_rule(endpoint) or
            self._get_default_rule()
        )

        # Cache result
        self.rules_cache[cache_key] = (rule, time.time())

        return rule

    def check_rate_limit(
        self,
        user_id: str,
        endpoint: str
    ) -> Tuple[bool, dict]:
        """
        Check if request should be allowed.

        Returns:
            (allowed, metadata) where metadata includes limit,
remaining, reset_at
        """
        rule = self.get_rate_limit_rule(user_id, endpoint)

        if not rule:
            # No rule found, allow by default
            return True, {}

        limit = rule['limit']
        window = rule['window_seconds']
        algorithm = rule.get('algorithm', 'sliding_window_counter')

        if algorithm == 'fixed_window':
            return self._check_fixed_window(user_id, endpoint, limit,
window)
        elif algorithm == 'sliding_window_counter':
            return self._check_sliding_window_counter(user_id, endpoint,
limit, window)
        elif algorithm == 'token_bucket':
            return self._check_token_bucket(user_id, endpoint, limit,
```

```python
window)
        else:
            return True, {}

    def _check_sliding_window_counter(
        self,
        user_id: str,
        endpoint: str,
        limit: int,
        window: int
    ) -> Tuple[bool, dict]:
        """Sliding window counter implementation using Redis."""
        now = time.time()
        current_window = int(now / window) * window
        previous_window = current_window - window

        key_current = f"ratelimit:{user_id}:{endpoint}:{current_window}"
        key_previous = f"ratelimit:{user_id}:{endpoint}:
{previous_window}"

        # Get counts
        pipe = self.redis.pipeline()
        pipe.get(key_current)
        pipe.get(key_previous)
        results = pipe.execute()

        current_count = int(results[0] or 0)
        previous_count = int(results[1] or 0)

        # Calculate weighted count
        elapsed_in_current = now - current_window
        current_window_position = elapsed_in_current / window
        previous_weight = 1.0 - current_window_position

        estimated_count = current_count + (previous_count *
previous_weight)

        if estimated_count < limit:
            # Increment counter
            pipe = self.redis.pipeline()
            pipe.incr(key_current)
            pipe.expire(key_current, window * 2)
            pipe.execute()

            remaining = int(limit - estimated_count - 1)
            reset_at = current_window + window

            return True, {
                'limit': limit,
                'remaining': max(0, remaining),
                'reset_at': reset_at
            }
```

```python
        reset_at = current_window + window
        retry_after = int(reset_at - now)

        return False, {
            'limit': limit,
            'remaining': 0,
            'reset_at': reset_at,
            'retry_after': retry_after
        }

    def _check_fixed_window(
        self,
        user_id: str,
        endpoint: str,
        limit: int,
        window: int
    ) -> Tuple[bool, dict]:
        """Fixed window implementation using Redis + Lua."""
        now = time.time()
        window_start = int(now / window) * window
        key = f"ratelimit:{user_id}:{endpoint}:{window_start}"

        # Lua script for atomic increment
        lua_script = """
        local key = KEYS[1]
        local limit = tonumber(ARGV[1])
        local window = tonumber(ARGV[2])

        local current = redis.call('INCR', key)
        if current == 1 then
            redis.call('EXPIRE', key, window)
        end

        if current > limit then
            return {0, current - 1, limit}
        else
            return {1, limit - current, limit}
        end
        """

        result = self.redis.eval(lua_script, 1, key, limit, window)
        allowed = bool(result[0])
        remaining = result[1]
        limit_value = result[2]

        reset_at = window_start + window
        metadata = {
            'limit': limit_value,
            'remaining': remaining,
            'reset_at': reset_at
        }

        if not allowed:
```

```
            metadata['retry_after'] = int(reset_at - now)

        return allowed, metadata

    def _get_user_endpoint_rule(self, user_id: str, endpoint: str) ->
Optional[dict]:
        """Get user-specific and endpoint-specific rule."""
        # Query database or cache
        return None  # Implement based on your storage

    def _get_user_rule(self, user_id: str) -> Optional[dict]:
        """Get user-specific rule."""
        # Query database for user tier
        return None

    def _get_endpoint_rule(self, endpoint: str) -> Optional[dict]:
        """Get endpoint-specific rule."""
        return None

    def _get_default_rule(self) -> dict:
        """Get default rate limit rule."""
        return {
            'limit': 1000,
            'window_seconds': 3600,
            'algorithm': 'sliding_window_counter'
        }

# Initialize middleware
rate_limiter = RateLimiterMiddleware(redis_client)

def rate_limit_decorator(f):
    """Decorator to apply rate limiting to routes."""
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # Extract user identifier
        user_id = request.headers.get('X-User-ID') or
request.remote_addr
        endpoint = request.endpoint or request.path

        # Check rate limit
        allowed, metadata = rate_limiter.check_rate_limit(user_id,
endpoint)

        # Store in request context
        g.rate_limit_metadata = metadata

        if not allowed:
            response = jsonify({
                'error': {
                    'code': 'RATE_LIMIT_EXCEEDED',
                    'message': f"Rate limit exceeded. Try again in
{metadata.get('retry_after', 0)} seconds.",
                    **metadata
```

```
                }
            })
            response.status_code = 429

            # Add headers
            if 'limit' in metadata:
                response.headers['X-RateLimit-Limit'] =
str(metadata['limit'])
            if 'remaining' in metadata:
                response.headers['X-RateLimit-Remaining'] =
str(metadata['remaining'])
            if 'reset_at' in metadata:
                response.headers['X-RateLimit-Reset'] =
str(int(metadata['reset_at']))
            if 'retry_after' in metadata:
                response.headers['Retry-After'] =
str(metadata['retry_after'])

            return response

        # Execute route handler
        response = f(*args, **kwargs)

        # Add rate limit headers to successful response
        if hasattr(g, 'rate_limit_metadata'):
            meta = g.rate_limit_metadata
            if hasattr(response, 'headers'):
                if 'limit' in meta:
                    response.headers['X-RateLimit-Limit'] =
str(meta['limit'])
                if 'remaining' in meta:
                    response.headers['X-RateLimit-Remaining'] =
str(meta['remaining'])
                if 'reset_at' in meta:
                    response.headers['X-RateLimit-Reset'] =
str(int(meta['reset_at']))

        return response

    return decorated_function

# Usage
@app.route('/api/data')
@rate_limit_decorator
def get_data():
    return jsonify({'data': 'some data'})

@app.route('/api/payments', methods=['POST'])
@rate_limit_decorator
def create_payment():
    return jsonify({'payment_id': '12345'})
```

2. Rules Engine

```
from dataclasses import dataclass
from typing import List, Optional, Dict
from enum import Enum

class RuleAction(Enum):
    ALLOW = "allow"
    REJECT = "reject"
    THROTTLE = "throttle"

class RuleSelector(Enum):
    USER_ID = "user_id"
    IP_ADDRESS = "ip_address"
    API_KEY = "api_key"
    ENDPOINT = "endpoint"
    USER_TIER = "user_tier"
    CUSTOM = "custom"

@dataclass
class RateLimitRule:
    rule_id: str
    name: str
    priority: int  # Lower number = higher priority
    enabled: bool

    # Selectors (matching criteria)
    selectors: Dict[RuleSelector, str]

    # Rate limit configuration
    limit: int
    window_seconds: int
    algorithm: str  # 'fixed_window', 'sliding_window_counter',
'token_bucket'

    # Actions
    action: RuleAction
    action_params: Optional[Dict] = None

    # Metadata
    created_at: float = 0
    updated_at: float = 0
    created_by: str = ""

class RulesEngine:
    def __init__(self):
        self.rules: List[RateLimitRule] = []
        self._load_rules()

    def _load_rules(self) -> None:
        """Load rules from database."""
```

```python
        # In production, load from PostgreSQL or similar
        self.rules = [
            RateLimitRule(
                rule_id="rule_1",
                name="Premium User Payment API",
                priority=1,
                enabled=True,
                selectors={
                    RuleSelector.USER_TIER: "premium",
                    RuleSelector.ENDPOINT: "/api/payments"
                },
                limit=10000,
                window_seconds=3600,
                algorithm="sliding_window_counter",
                action=RuleAction.REJECT
            ),
            RateLimitRule(
                rule_id="rule_2",
                name="Free User Global",
                priority=10,
                enabled=True,
                selectors={
                    RuleSelector.USER_TIER: "free"
                },
                limit=100,
                window_seconds=3600,
                algorithm="sliding_window_counter",
                action=RuleAction.REJECT
            ),
            RateLimitRule(
                rule_id="rule_default",
                name="Default Rate Limit",
                priority=100,
                enabled=True,
                selectors={},  # Matches all
                limit=1000,
                window_seconds=3600,
                algorithm="sliding_window_counter",
                action=RuleAction.REJECT
            )
        ]

        # Sort by priority
        self.rules.sort(key=lambda r: r.priority)

    def find_applicable_rule(
        self,
        user_id: str,
        ip_address: str,
        endpoint: str,
        user_tier: str,
        api_key: str
    ) -> Optional[RateLimitRule]:
```

```
        """
        Find the most specific rule that matches the request.

        Returns the highest priority (lowest number) matching rule.
        """
        context = {
            RuleSelector.USER_ID: user_id,
            RuleSelector.IP_ADDRESS: ip_address,
            RuleSelector.ENDPOINT: endpoint,
            RuleSelector.USER_TIER: user_tier,
            RuleSelector.API_KEY: api_key
        }

        for rule in self.rules:
            if not rule.enabled:
                continue

            if self._rule_matches(rule, context):
                return rule

        return None

    def _rule_matches(
        self,
        rule: RateLimitRule,
        context: Dict[RuleSelector, str]
    ) -> bool:
        """Check if rule matches the given context."""
        if not rule.selectors:
            # Empty selectors match everything (default rule)
            return True

        # All selectors must match
        for selector, required_value in rule.selectors.items():
            actual_value = context.get(selector, "")

            if not self._selector_matches(selector, required_value,
actual_value):
                return False

        return True

    def _selector_matches(
        self,
        selector: RuleSelector,
        required: str,
        actual: str
    ) -> bool:
        """Check if a single selector matches."""
        if selector == RuleSelector.ENDPOINT:
            # Support prefix matching for endpoints
            return actual.startswith(required)
        elif selector == RuleSelector.IP_ADDRESS:
```

```python
            # Support CIDR matching (simplified)
            return actual == required  # TODO: Implement CIDR matching
        else:
            # Exact match for other selectors
            return actual == required

    def add_rule(self, rule: RateLimitRule) -> None:
        """Add a new rule."""
        self.rules.append(rule)
        self.rules.sort(key=lambda r: r.priority)
        # Persist to database

    def update_rule(self, rule_id: str, updates: Dict) -> bool:
        """Update an existing rule."""
        for rule in self.rules:
            if rule.rule_id == rule_id:
                for key, value in updates.items():
                    if hasattr(rule, key):
                        setattr(rule, key, value)
                # Persist to database
                return True
        return False

    def delete_rule(self, rule_id: str) -> bool:
        """Delete a rule."""
        self.rules = [r for r in self.rules if r.rule_id != rule_id]
        # Persist to database
        return True
```

Database Schema

PostgreSQL Schema

```sql
-- Rate limit rules table
CREATE TABLE rate_limit_rules (
    rule_id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    description TEXT,
    priority INTEGER NOT NULL DEFAULT 100,
    enabled BOOLEAN NOT NULL DEFAULT true,

    -- Rate limit configuration
    limit_value INTEGER NOT NULL,
    window_seconds INTEGER NOT NULL,
    algorithm VARCHAR(50) NOT NULL DEFAULT 'sliding_window_counter',

    -- Action configuration
    action VARCHAR(50) NOT NULL DEFAULT 'reject',
    action_params JSONB,
```

```sql
    -- Metadata
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(255),

    -- Indexes
    CONSTRAINT chk_priority CHECK (priority >= 0),
    CONSTRAINT chk_limit CHECK (limit_value > 0),
    CONSTRAINT chk_window CHECK (window_seconds > 0)
);

CREATE INDEX idx_rules_priority ON rate_limit_rules(priority);
CREATE INDEX idx_rules_enabled ON rate_limit_rules(enabled);

-- Rule selectors table (for matching criteria)
CREATE TABLE rate_limit_rule_selectors (
    id SERIAL PRIMARY KEY,
    rule_id UUID NOT NULL REFERENCES rate_limit_rules(rule_id) ON DELETE
CASCADE,
    selector_type VARCHAR(50) NOT NULL,
    selector_value VARCHAR(255) NOT NULL,

    UNIQUE(rule_id, selector_type)
);

CREATE INDEX idx_selectors_rule_id ON
rate_limit_rule_selectors(rule_id);
CREATE INDEX idx_selectors_type ON
rate_limit_rule_selectors(selector_type);

-- User tiers table
CREATE TABLE user_tiers (
    user_id VARCHAR(255) PRIMARY KEY,
    tier VARCHAR(50) NOT NULL,
    custom_limits JSONB,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_user_tiers_tier ON user_tiers(tier);

-- Whitelist/Blacklist table
CREATE TABLE access_control_list (
    id SERIAL PRIMARY KEY,
    entry_type VARCHAR(20) NOT NULL CHECK (entry_type IN ('whitelist',
'blacklist')),
    identifier_type VARCHAR(50) NOT NULL,  -- 'user_id', 'ip_address',
'api_key'
    identifier_value VARCHAR(255) NOT NULL,
    reason TEXT,
    expires_at TIMESTAMP,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(255),
```

```sql
    UNIQUE(entry_type, identifier_type, identifier_value)
);

CREATE INDEX idx_acl_type_identifier ON access_control_list(entry_type,
identifier_type, identifier_value);

-- Audit log table
CREATE TABLE rate_limit_audit_log (
    id BIGSERIAL PRIMARY KEY,
    event_type VARCHAR(50) NOT NULL,  -- 'rule_created', 'rule_updated',
'rule_deleted'
    rule_id UUID,
    user_id VARCHAR(255),
    changes JSONB,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(255)
);

CREATE INDEX idx_audit_rule_id ON rate_limit_audit_log(rule_id);
CREATE INDEX idx_audit_created_at ON rate_limit_audit_log(created_at);

-- Example inserts
INSERT INTO rate_limit_rules (name, description, priority, limit_value,
window_seconds, algorithm)
VALUES
('Default Rate Limit', 'Global default rate limit', 100, 1000, 3600,
'sliding_window_counter'),
('Premium API Payment', 'Rate limit for premium payment API', 1, 10000,
3600, 'sliding_window_counter');

INSERT INTO rate_limit_rule_selectors (rule_id, selector_type,
selector_value)
SELECT rule_id, 'user_tier', 'premium' FROM rate_limit_rules WHERE name
= 'Premium API Payment';

INSERT INTO user_tiers (user_id, tier) VALUES
('user_123', 'free'),
('user_456', 'premium');
```

Distributed System Challenges

Challenge 1: Race Conditions

Problem:
Multiple servers incrementing the same counter simultaneously.

```
Time     Server A        Server B        Redis Counter
t0       Read: 99        -               99
```

```
t1      -               Read: 99        99
t2      Write: 100      -               100
t3      -               Write: 100      100  ← Should be 101!
```

Solution:
Use atomic operations (Redis INCR, Lua scripts).

```lua
-- Atomic increment with limit check
local key = KEYS[1]
local limit = tonumber(ARGV[1])

local current = redis.call('INCR', key)
if current > limit then
    redis.call('DECR', key)  -- Rollback
    return 0  -- Rejected
end

return 1  -- Allowed
```

Challenge 2: Clock Synchronization

Problem:
Different servers have slightly different clocks.

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

Window boundary at 10:00:00:
- Server A thinks it's in old window
- Server B thinks it's in new window
```

Solutions:

1. **Use centralized timestamps (Redis TIME command)**

```python
# Get time from Redis (synchronized)
time_response = redis.execute_command('TIME')
redis_time = float(f"{time_response[0]}.{time_response[1]}")
```

2. **Use NTP to synchronize clocks**
3. **Accept slight inaccuracy** (usually acceptable for rate limiting)

Challenge 3: Redis Availability

Problem:
What if Redis goes down?

Solutions:

Option 1: Fail Open (Recommended for most cases)

```python
def check_rate_limit(user_id):
    try:
        # Check rate limit with Redis
        return redis.check_limit(user_id)
    except RedisConnectionError:
        # Log error and allow request
        logger.error("Redis unavailable, failing open")
        return True  # Allow request
```

Option 2: Fail Closed (High security requirements)

```python
def check_rate_limit(user_id):
    try:
        return redis.check_limit(user_id)
    except RedisConnectionError:
        logger.error("Redis unavailable, failing closed")
        return False  # Reject request
```

Option 3: Local Cache Fallback

```python
class RateLimiterWithFallback:
    def __init__(self):
        self.redis = redis.Redis()
        self.local_cache = {}  # In-memory fallback

    def check_rate_limit(self, user_id):
        try:
            return self._check_redis(user_id)
        except RedisConnectionError:
            logger.warning("Redis down, using local cache")
            return self._check_local_cache(user_id)
```

Challenge 4: Data Consistency Across Regions

Problem:
Multi-region deployment with eventual consistency.

```
US-WEST: User makes 50 requests
US-EAST: User makes 50 requests

Total: 100 requests, but each region only sees 50
If limit is 60, both regions allow all requests!
```

Solutions:

Option 1: Regional Limits

```
Global limit: 100 requests/hour
Per-region limit: 60 requests/hour (with 40 buffer)

Even if one region fails, total won't exceed ~120
```

Option 2: Centralized Redis (Higher latency)

```
All regions → Single Redis Cluster in one region
Adds ~50-200ms latency
```

Option 3: Eventual Consistency with Quota Distribution

```
# Distribute quota across regions
total_limit = 100
num_regions = 3
per_region_limit = total_limit / num_regions  # 33 per region

# Accept some over-limit (100-120 requests possible)
```

Challenge 5: Redis Cluster Partitioning

Problem:
How to distribute rate limit counters across Redis cluster?

Solution: Consistent Hashing

```
import hashlib

def get_redis_shard(user_id: str, num_shards: int) -> int:
    """Determine which Redis shard to use."""
    hash_value = int(hashlib.md5(user_id.encode()).hexdigest(), 16)
    return hash_value % num_shards
```

```python
# Usage
redis_clients = [Redis(host=f"redis-{i}") for i in range(5)]
shard = get_redis_shard("user_123", len(redis_clients))
redis_clients[shard].incr("ratelimit:user_123")
```

Edge Cases and Corner Cases

1. Window Boundary Edge Cases

Edge Case: Requests exactly at window boundary

```
# Example: Fixed window at minute boundaries
# Limit: 100 req/minute

# 10:00:59.999 - Request 100 (Window 1, allowed)
# 10:01:00.000 - Request 101 (Window 2, allowed)
# 10:01:00.001 - Request 102 (Window 2, allowed)

# Result: 3 requests in ~1ms, all allowed
```

Solution:
Use sliding window counter or sliding window log to avoid this.

2. Distributed Counter Synchronization

Edge Case: Two servers increment at exact same time

```
Server A: Check count (99) → Increment to 100 → Allow
Server B: Check count (99) → Increment to 100 → Allow

Result: 101 requests allowed (limit was 100)
```

Solution:
Use Lua scripts for atomic check-and-increment.

3. Clock Skew Between Servers

Edge Case: Server clocks are out of sync

```
Server A time: 10:00:00 (Window 1)
Server B time: 10:00:05 (Window 2)

User makes 100 requests to Server A (Window 1, allowed)
User makes 100 requests to Server B (Window 2, allowed)
```

```
Result: 200 requests in 5 seconds
```

Solution:

- Use Redis TIME command for synchronized time
- Or implement NTP synchronization

4. Burst at Window Reset

Edge Case: Users time their requests for window reset

```
User waits until 09:59:55
At 09:59:55: Send 100 requests (Window 1, all allowed)
At 10:00:00: Send 100 requests (Window 2, all allowed)

Result: 200 requests in 5 seconds
```

Solution:
Implement sliding window counter to smooth out transitions.

5. Redis Key Expiration Timing

Edge Case: Key expires while checking

```
# Thread 1: Check key exists → Yes (count: 50)
# Thread 2: Check key exists → Yes (count: 50)
# [Key expires here]
# Thread 1: INCR key → Creates new key with count: 1
# Thread 2: INCR key → count: 2

# Lost the previous count of 50!
```

Solution:
Use Lua scripts to handle expiration atomically.

```lua
local current = redis.call('GET', key)
if not current then
    current = 0
end

current = tonumber(current) + 1
redis.call('SET', key, current)
redis.call('EXPIRE', key, window)

return current <= limit and 1 or 0
```

```
```

6. Very Large Bursts

Edge Case: Legitimate burst of 10,000 requests in 1 second

```
Example: Black Friday sale starts
Thousands of users hit "Buy" button simultaneously
```

Solution:

- Use Token Bucket to allow bursts
- Implement queue-based rate limiting
- Scale Redis cluster horizontally

7. User Switching IPs

Edge Case: Mobile user switches between WiFi and cellular

```
User on WiFi: IP 192.168.1.1
Makes 50 requests

Switches to cellular: IP 10.0.0.1
Makes 50 more requests

If rate limiting by IP: Sees as 2 different users
Total: 100 requests from same user
```

Solution:

- Rate limit by user ID instead of IP
- Or combine both: `min(ip_limit, user_limit)`

8. Shared IP (NAT/Proxy)

Edge Case: Corporate network with single public IP

```
100 users behind corporate NAT
All share IP 1.2.3.4

If rate limit is 100 req/hour per IP:
Each user only gets 1 request/hour!
```

Solution:

- Don't rate limit by IP alone

- Use user ID + IP combination
- Higher limits for known corporate IPs

9. Redis Failover During Request

Edge Case: Redis master fails mid-request

```
1. Request arrives
2. Check rate limit (Master Redis)
3. [Master fails, replica promoted]
4. Increment counter (New Master)
5. Counter state may be inconsistent
```

Solution:

- Use Redis Sentinel or Redis Cluster for automatic failover
- Accept brief inconsistency window
- Implement retry logic with exponential backoff

10. Time Travel (System Clock Changed)

Edge Case: Server clock set backwards

```
Time: 10:00:00 → 09:00:00 (clock set back)

Window calculations become invalid:
- Current window: 09:00:00
- Previous window: 08:00:00
- Lost all counters from 09:00-10:00!
```

Solution:

- Monitor clock skew and alert
- Use Redis TIME instead of local time
- Reject requests if clock skew detected

11. Integer Overflow

Edge Case: Counter exceeds max integer

```
# After 2^31 - 1 increments (2.1 billion)
counter = 2147483647
counter += 1  # Overflow to -2147483648 in 32-bit systems
```

Solution:

```
# Use 64-bit integers
# Set reasonable limits
# Reset counters periodically
MAX_SAFE_INTEGER = 9007199254740991  # JavaScript max safe integer

if counter > MAX_SAFE_INTEGER - 1000:
    # Reset or warn
    pass
```

12. DDoS via Rate Limiter

Edge Case: Attacker deliberately triggers rate limiter

```
Attacker makes requests using victim's API key
Victim's API key gets rate limited
Legitimate requests from victim are now blocked
```

Solution:

- Implement CAPTCHA after repeated failures
- Monitor for suspicious patterns
- Allow whitelist-based bypass for verified users
- Use multiple rate limit dimensions (IP + API key)

Scaling the System

Horizontal Scaling Strategy

Phase 1: Single Region (0-1K RPS)

```
┌─────────────┐
│   Clients   │
└──────┬──────┘
       │
       ▼
┌─────────────┐
│     LB      │
└──────┬──────┘
       │
       ├───────┬───────┐
       ▼       ▼       ▼
   ┌────┐  ┌────┐  ┌────┐
   │API │  │API │  │API │
   │GW 1│  │GW 2│  │GW 3│
   └─┬──┘  └─┬──┘  └─┬──┘
```

```
        └────────┬────────┘
                 │
                 ▼
           ┌──────────┐
           │  Redis   │
           │ (Master) │
           └────┬─────┘
                │
                ▼
           ┌──────────┐
           │  Redis   │
           │(Replica) │
           └──────────┘
```

Cost: ~$500/month
Capacity: 1,000 RPS
Latency: <5ms

Phase 2: Regional Scaling (1K-10K RPS)

```
┌─────────────┐
│   Clients   │
└──────┬──────┘
       │
       ▼
┌─────────────┐
│  CDN/WAF    │
└──────┬──────┘
       │
       ▼
┌─────────────┐
│   ALB (×3)  │
└──────┬──────┘
       │
       ├────────┬────────┬────────┐
       ▼        ▼        ▼        ▼
    ┌────┐   ┌────┐   ┌────┐   ┌────┐
    │API │   │API │   │API │...│API │
    │GW  │   │GW  │   │GW  │   │GW  │
    │×10 │   │×10 │   │×10 │   │×10 │
    └─┬──┘   └─┬──┘   └─┬──┘   └─┬──┘
      └────────┼────────┼────────┘
               ▼        ▼
         ┌──────────┬──────────┐
         │  Redis   │  Redis   │
         │ Cluster  │ Cluster  │
         │(Master×5)│(Replica) │
         └──────────┴──────────┘
```

Cost: ~$6,000/month
Capacity: 10,000 RPS
Latency: <10ms

Phase 3: Multi-Region (10K-100K RPS)

```
            ┌─────────────┐
            │   Clients   │
            └──────┬──────┘
                   │
                   ▼
            ┌─────────────┐
            │  GeoDNS     │
            └──────┬──────┘
                   │
        ┌──────────┴──────────┐
        ▼                     ▼
  ┌─────────┐          ┌─────────┐
  │ US-WEST │          │ US-EAST │
  │ Region  │          │ Region  │
  │         │          │         │
  │ API GW  │          │ API GW  │
  │ (×50)   │          │ (×50)   │
  │   ↓     │          │   ↓     │
  │ Redis   │←────────→│ Redis   │
  │Cluster  │  Sync    │Cluster  │
  └─────────┘          └─────────┘
```

Cost: ~$30,000/month
Capacity: 100,000 RPS
Latency: <20ms

Auto-Scaling Configuration

```yaml
# Kubernetes HPA for API Gateway
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: rate-limiter-api-gateway
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api-gateway
  minReplicas: 10
  maxReplicas: 100
  metrics:
  - type: Resource
```

```yaml
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
  behavior:
    scaleUp:
      stabilizationWindowSeconds: 60
      policies:
      - type: Percent
        value: 50
        periodSeconds: 60
    scaleDown:
      stabilizationWindowSeconds: 300
      policies:
      - type: Percent
        value: 10
        periodSeconds: 60
```

Database Scaling Strategy

Read Replicas

```python
class RateLimiterWithReadReplicas:
    def __init__(self):
        self.master = Redis(host='master')
        self.replicas = [
            Redis(host='replica1'),
            Redis(host='replica2'),
            Redis(host='replica3')
        ]
        self.replica_index = 0

    def increment_counter(self, key):
        """Writes go to master."""
        return self.master.incr(key)

    def get_counter(self, key):
        """Reads from replica (round-robin)."""
        replica = self.replicas[self.replica_index]
        self.replica_index = (self.replica_index + 1) %
len(self.replicas)
        return replica.get(key)
```

Redis Cluster Sharding

```python
from redis.cluster import RedisCluster

# Redis Cluster automatically shards data
redis_cluster = RedisCluster(
    startup_nodes=[
        {"host": "redis-1", "port": "6379"},
        {"host": "redis-2", "port": "6379"},
        {"host": "redis-3", "port": "6379"},
    ],
    decode_responses=True,
    skip_full_coverage_check=True
)

# Keys automatically distributed across shards
redis_cluster.incr("ratelimit:user_123")  # → Shard 1
redis_cluster.incr("ratelimit:user_456")  # → Shard 2
```

Caching Strategy

```python
from functools import lru_cache
import time

class CachedRulesEngine:
    def __init__(self):
        self.db = PostgreSQL()
        self.cache_ttl = 300  # 5 minutes

    @lru_cache(maxsize=10000)
    def get_rule_cached(self, cache_key, timestamp):
        """Cache rules with TTL."""
        return self._get_rule_from_db(cache_key)

    def get_rule(self, user_id, endpoint):
        # Create cache key
        cache_key = f"{user_id}:{endpoint}"

        # Use timestamp for cache invalidation
        cache_timestamp = int(time.time() / self.cache_ttl)

        return self.get_rule_cached(cache_key, cache_timestamp)
```

Monitoring and Operations

Key Metrics to Track

1. Request Metrics

```python
# Prometheus metrics
from prometheus_client import Counter, Histogram, Gauge

# Request counters
requests_total = Counter(
    'rate_limiter_requests_total',
    'Total requests',
    ['endpoint', 'user_tier', 'result']  # result: allowed/rejected
)

requests_allowed = Counter(
    'rate_limiter_requests_allowed_total',
    'Allowed requests',
    ['endpoint', 'user_tier']
)

requests_rejected = Counter(
    'rate_limiter_requests_rejected_total',
    'Rejected requests',
    ['endpoint', 'user_tier', 'reason']
)

# Latency histogram
rate_limit_check_duration = Histogram(
    'rate_limiter_check_duration_seconds',
    'Rate limit check duration',
    ['algorithm'],
    buckets=[0.001, 0.005, 0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1.0]
)

# Current rate limit usage
rate_limit_usage = Gauge(
    'rate_limiter_usage_percentage',
    'Current rate limit usage',
    ['user_id', 'endpoint']
)
```

2. System Health Metrics

```python
# Redis health
redis_connection_pool_size = Gauge(
    'redis_connection_pool_size',
    'Redis connection pool size'
)

redis_latency = Histogram(
    'redis_operation_duration_seconds',
```

```python
    'Redis operation duration',
    ['operation']  # GET, SET, INCR, etc.
)

# Error rates
errors_total = Counter(
    'rate_limiter_errors_total',
    'Total errors',
    ['error_type']  # redis_connection, timeout, etc.
)
```

Logging Strategy

```python
import logging
import json

class StructuredLogger:
    def __init__(self):
        self.logger = logging.getLogger('rate_limiter')

    def log_rate_limit_check(
        self,
        user_id: str,
        endpoint: str,
        allowed: bool,
        limit: int,
        current_count: int,
        latency_ms: float
    ):
        """Log rate limit check with structured data."""
        log_data = {
            'event': 'rate_limit_check',
            'user_id': user_id,
            'endpoint': endpoint,
            'allowed': allowed,
            'limit': limit,
            'current_count': current_count,
            'latency_ms': latency_ms,
            'timestamp': time.time()
        }

        if allowed:
            self.logger.info(json.dumps(log_data))
        else:
            self.logger.warning(json.dumps(log_data))

    def log_rate_limit_exceeded(
        self,
        user_id: str,
        endpoint: str,
```

```python
        limit: int,
        attempts: int,
        window: str
    ):
        """Log rate limit exceeded event."""
        log_data = {
            'event': 'rate_limit_exceeded',
            'user_id': user_id,
            'endpoint': endpoint,
            'limit': limit,
            'attempts': attempts,
            'window': window,
            'timestamp': time.time()
        }

        self.logger.warning(json.dumps(log_data))
```

Alerting Rules

```yaml
# Prometheus alerting rules
groups:
  - name: rate_limiter_alerts
    interval: 30s
    rules:
      # High rejection rate
      - alert: HighRejectionRate
        expr: |
          rate(rate_limiter_requests_rejected_total[5m]) /
          rate(rate_limiter_requests_total[5m]) > 0.10
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High rate limit rejection rate"
          description: "Rejection rate is {{ $value | humanizePercentage }} (threshold: 10%)"

      # Redis latency high
      - alert: RedisLatencyHigh
        expr: |
          histogram_quantile(0.99,
            rate(redis_operation_duration_seconds_bucket[5m])
          ) > 0.010
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Redis P99 latency high"
          description: "Redis P99 latency is {{ $value }}s (threshold: 10ms)"
```

```
      # Redis connection issues
      - alert: RedisConnectionErrors
        expr: |
          rate(rate_limiter_errors_total{error_type="redis_connection"}
[5m]) > 1
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "Redis connection errors detected"
          description: "{{ $value }} Redis connection errors per second"

      # Rate limiter latency high
      - alert: RateLimiterLatencyHigh
        expr: |
          histogram_quantile(0.99,
            rate(rate_limiter_check_duration_seconds_bucket[5m])
          ) > 0.050
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "Rate limiter P99 latency high"
          description: "Rate limiter P99 latency is {{ $value }}s
(threshold: 50ms)"
```

Grafana Dashboard

```
{
  "dashboard": {
    "title": "Rate Limiter Monitoring",
    "panels": [
      {
        "title": "Requests Per Second",
        "targets": [
          {
            "expr": "rate(rate_limiter_requests_total[1m])",
            "legendFormat": "Total RPS"
          },
          {
            "expr": "rate(rate_limiter_requests_allowed_total[1m])",
            "legendFormat": "Allowed RPS"
          },
          {
            "expr": "rate(rate_limiter_requests_rejected_total[1m])",
            "legendFormat": "Rejected RPS"
          }
        ]
      },
```

```json
        {
          "title": "Rejection Rate",
          "targets": [
            {
              "expr": "rate(rate_limiter_requests_rejected_total[5m]) /
rate(rate_limiter_requests_total[5m])",
              "legendFormat": "Rejection Rate"
            }
          ]
        },
        {
          "title": "Latency Distribution",
          "targets": [
            {
              "expr": "histogram_quantile(0.50,
rate(rate_limiter_check_duration_seconds_bucket[5m]))",
              "legendFormat": "P50"
            },
            {
              "expr": "histogram_quantile(0.95,
rate(rate_limiter_check_duration_seconds_bucket[5m]))",
              "legendFormat": "P95"
            },
            {
              "expr": "histogram_quantile(0.99,
rate(rate_limiter_check_duration_seconds_bucket[5m]))",
              "legendFormat": "P99"
            }
          ]
        },
        {
          "title": "Top Limited Users",
          "targets": [
            {
              "expr": "topk(10, sum by (user_id)
(rate(rate_limiter_requests_rejected_total[5m])))",
              "legendFormat": "{{user_id}}"
            }
          ]
        }
      ]
    }
  }
```

Real-World Examples

1. Stripe API Rate Limiting

Implementation:

- **Algorithm:** Token Bucket + Sliding Window
- **Limits:**
  - 100 reads/second (burst allowed)
  - 25 writes/second
  - Different limits per API version
- **Headers:**

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 87
X-RateLimit-Reset: 1640995200
```

Key Features:

- Separate limits for read vs. write operations
- Test mode has separate rate limits
- Webhook endpoints have different limits
- Burst capacity for legitimate traffic spikes

2. GitHub API Rate Limiting

Implementation:

- **Algorithm:** Fixed Window per hour
- **Limits:**
  - Unauthenticated: 60 requests/hour (by IP)
  - Authenticated: 5,000 requests/hour (by user)
  - GraphQL: Calculated by query complexity
- **Headers:**

```
X-RateLimit-Limit: 5000
X-RateLimit-Remaining: 4999
X-RateLimit-Reset: 1640995200
X-RateLimit-Used: 1
```

Special Features:

- Secondary rate limit for rapid bursts (max 100 req/minute)
- Different limits for GitHub Apps
- Conditional requests don't count toward limit
- GraphQL uses point system based on query complexity

3. Twitter API Rate Limiting

Implementation:

- **Algorithm:** Fixed Window (15 minutes)

- **Limits:**
  - User context: 15 requests/15 min per user
  - App context: 15 requests/15 min per app
  - Different limits per endpoint
- **Response:**

```
{
  "errors": [{
    "code": 88,
    "message": "Rate limit exceeded"
  }]
}
```

Key Features:

- Per-endpoint rate limits
- Window is 15 minutes (not hourly)
- Rate limits reset at specific intervals
- Premium tiers have higher limits

4. AWS API Gateway Throttling

Implementation:

- **Algorithm:** Token Bucket
- **Limits:**
  - Steady-state: 10,000 requests/second
  - Burst: 5,000 requests
  - Per-API key or per-client limits

Configuration Example:

```
ThrottleSettings:
  BurstLimit: 5000
  RateLimit: 10000

MethodSettings:
  - HttpMethod: GET
    ResourcePath: /users/*
    ThrottleSettings:
      BurstLimit: 1000
      RateLimit: 500
```

5. Redis Rate Limiting (redis-cell)

Implementation:
Uses Generic Cell Rate Algorithm (GCRA) - variant of leaky bucket.

```
# redis-cell module
CL.THROTTLE user123 15 30 60 1
               │         │  │  │  └─ tokens to consume
               │         │  │  └──── window in seconds
               │         │  └─────── max burst
               │         └────────── capacity
               └──────────────────── key

Response:
1) (integer) 0         # 0 = allowed, 1 = rejected
2) (integer) 16        # Total capacity
3) (integer) 15        # Remaining capacity
4) (integer) -1        # Seconds until retry (-1 if allowed)
5) (integer) 2         # Seconds until full capacity
```

Interview Q&A

Common Interview Questions & Answers

Q1: "How would you handle a DDoS attack?"

Answer:
"I'd implement multiple layers of defense:

1. **CDN/WAF Level:** Block obvious attacks at the edge
    - IP reputation filtering
    - Geographic restrictions
    - Challenge suspicious traffic with CAPTCHA
2. **Rate Limiter Level:**
    - Very strict IP-based limits (100 req/min per IP)
    - Exponential backoff for repeat offenders
    - Temporary IP blacklisting
3. **Application Level:**
    - Require authentication for sensitive endpoints
    - Implement honeypot endpoints
    - Monitor for suspicious patterns (same user-agent, sequential IPs)
4. **Infrastructure:**
    - Auto-scaling to handle legitimate spikes

    - Circuit breakers to protect backend services
    - Separate rate limit pools for different request types"

Q2: "What if a celebrity tweets your website and you get 100x traffic?"

Answer:
"This is actually a good problem! Here's my approach:

1. **Immediate Response (0-5 minutes):**

    - Auto-scaling kicks in (configured for 10x burst)
    - CloudFront/CDN absorbs static content requests
    - Rate limiter allows burst traffic (token bucket algorithm)

2. **Short-term (5-30 minutes):**

    - Monitor which endpoints are hit hardest
    - Temporarily increase rate limits for unauthenticated users