Software Architecture and Design Specification

Project: API Rate Limiter System

Version: 1.0

Authors: Nigam Reddy (PES1UG23CS904), Yeswant Padavala (PES1UG23CS721), Vishal Naik (PES1UG23CS695)

Date: 14-09-2025

Status: Draft

Revision History

Version	Date	Author	Summary
1.0	14/09/2025	Team	Initial draft creation

Approvals

Role	Name	Signature / Date
QA Lead	Nigam Reddy	Nigam 14/09/2025
Dev Lead	Yeswant Padavala	Yeswant 14/09/2025
Product Owner	Vishal Naik	Vishal 14/09/2025

1. Introduction

1.1 Purpose

This document specifies the architecture and design of the API Rate Limiter System.

1.2 Scope

Covers the complete API Rate Limiter System including:

- Request counting and tracking mechanisms
- Rate limit enforcement algorithms
- Policy management interfaces
- Monitoring and logging capabilities
- High availability and clustering support
- Security and DoS protection features

1.3 Audience

- Software developers and architects
- QA engineers and testers
- Security auditors
- DevOps and infrastructure teams
- Product managers and stakeholders

1.4 Definitions

- **API**: Application Programming Interface
- **RPS**: Requests Per Second
- **DoS**: Denial of Service
- **TLS**: Transport Layer Security
- **Redis**: In-memory data structure store
- **ADR**: Architecture Decision Record

2. Document Overview

2.1 How to use this document

Provides architectural deliverables including UML diagrams, component specifications, API designs, security considerations, and deployment guidelines.

2.2 Related Documents

- API Rate Limiter SRS (Software Requirements Specification)
- Software Test Plan (STP) v1.0
- Design Specifications v1.0
- Security Requirements Document

3. Architecture

3.1 Goals & Constraints

Goals:

- **Performance**: Add minimal latency (<10ms) to API requests
- **Scalability**: Handle high-throughput traffic (1000+ RPS)
- **Reliability**: Achieve 99.9% system uptime
- **Security**: Protect against DoS attacks and unauthorized access
- **Flexibility**: Support multiple rate limiting algorithms

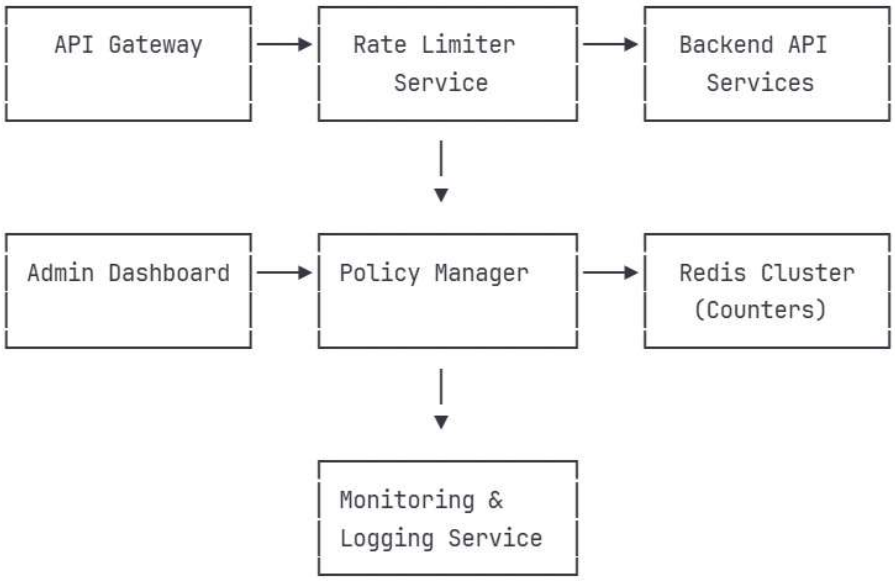
Constraints:

- Must integrate with existing API infrastructure
- Redis dependency for distributed counter management
- TLS 1.2+ encryption requirement
- Container deployment compatibility (Docker/Kubernetes)

3.2 Stakeholders & Concerns

- **API Consumers (Customers)**: Low latency, fair usage policies
- **System Administrators**: Easy policy management, monitoring capabilities
- **Security Teams**: DoS protection, credential security
- **Development Teams**: Simple integration, comprehensive logging
- **Operations Teams**: High availability, performance monitoring

3.3 Component (UML) Diagram



3.4 Component Descriptions

Rate Limiter Service

- **Purpose**: Core rate limiting logic and request validation
- **Responsibilities**: Request counting, limit enforcement, algorithm implementation
- **Interfaces**: REST API for rate checks, metrics collection

Policy Manager

- **Purpose**: Configuration and management of rate limiting policies
- **Responsibilities**: Policy CRUD operations, rule validation, policy distribution
- **Interfaces**: Admin API, configuration management

Redis Cluster

- **Purpose**: Distributed counter storage and caching
- **Responsibilities**: Request counting, sliding window data, high-availability storage
- **Interfaces**: Redis protocol, clustering support

Monitoring & Logging Service

- **Purpose**: System observability and audit trails
- **Responsibilities**: Metrics collection, alerting, log aggregation
- **Interfaces**: Prometheus metrics, structured logging

Admin Dashboard

- **Purpose**: User interface for policy management
- **Responsibilities**: Policy configuration UI, monitoring dashboards, user management
- **Interfaces**: Web UI, REST API consumption

3.5 Chosen Architecture Pattern and Rationale

Selected Pattern: Microservices with Event-Driven Architecture

Rationale:

- **Microservices**: Enables independent scaling of rate limiting components
- **Event-Driven**: Supports real-time monitoring and alerting
- **Separation of Concerns**: Clear boundaries between rate limiting, policy management, and monitoring

Alternatives Considered:

- **Monolithic**: Rejected due to scalability limitations
- **Serverless**: Rejected due to latency requirements and state management complexity

3.6 Technology Stack & Data Stores

Core Technologies

- **Runtime**: Java 11+ with Spring Boot
- **Container**: Docker with Kubernetes orchestration
- **Load Balancing**: NGINX or HAProxy
- **Service Mesh**: Istio (optional)

Data Storage

- **Primary Cache**: Redis 6.0+ cluster
- **Configuration Store**: PostgreSQL for policy persistence
- **Time-Series Data**: InfluxDB for metrics storage

Monitoring & Security

- **Metrics**: Prometheus + Grafana
- **Logging**: ELK Stack (Elasticsearch, Logstash, Kibana)

- **Security**: TLS 1.2+, OAuth 2.0 for admin access
- **Testing Tools**: Apache JMeter, OWASP ZAP

3.7 Risks & Mitigations

Risk	Impact	Probability	Mitigation
Redis cluster failure	High	Medium	Multi-region Redis deployment, automatic failover
Performance degradation	High	High	Extensive load testing, performance monitoring
Security vulnerabilities	High	Medium	Security testing, code reviews, penetration testing
Configuration errors	Medium	Medium	Validation frameworks, testing environments

3.8 Traceability to Requirements

Requirement	Component	Implementation
RL-F-001: Request counting	Rate Limiter Service	Counter algorithms with Redis backend
RL-F-004: Limit enforcement	Rate Limiter Service	HTTP 429 response with Retry-After headers
RL-F-007: Policy management	Policy Manager	CRUD operations with validation
RL-F-010: Monitoring	Monitoring Service	Prometheus metrics, structured logging
RL-F-013: High availability	Redis Cluster	Multi-node Redis with replication

3.9 Security Architecture

Threat Modeling (STRIDE Analysis)

Spoofing:

- Threat: Fake API keys or user impersonation
- Mitigation: Strong authentication, API key validation, certificate-based identity

Tampering:

- Threat: Configuration manipulation, counter tampering
- Mitigation: Encrypted communication, integrity checks, audit logging

Information Disclosure:

- Threat: Exposure of rate limiting policies, user data
- Mitigation: TLS encryption, access controls, data classification

Denial of Service:

- Threat: Rate limiter bypass, resource exhaustion
- Mitigation: Multiple rate limiting layers, resource monitoring, circuit breakers

Elevation of Privilege:

- Threat: Unauthorized policy changes, admin access
- Mitigation: Role-based access control (RBAC), principle of least privilege

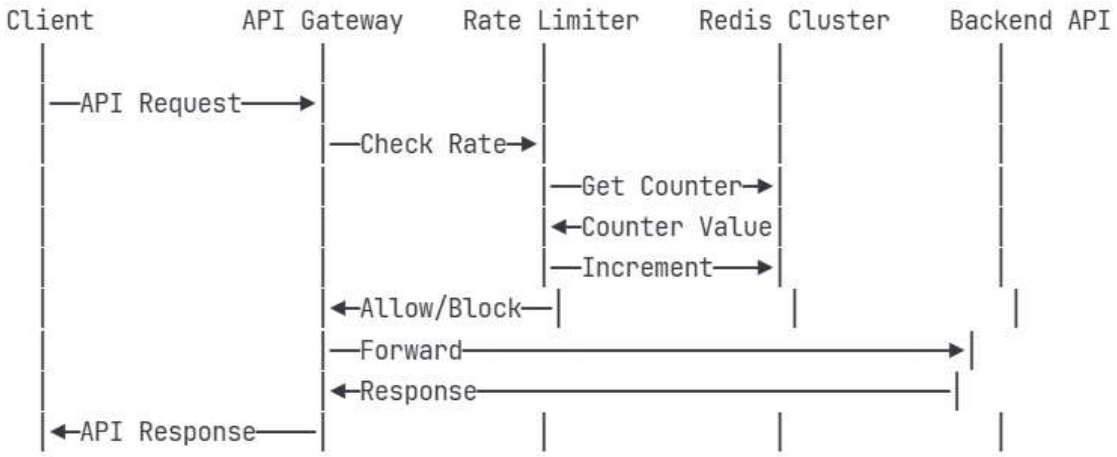
4. Design

4.1 Design Overview

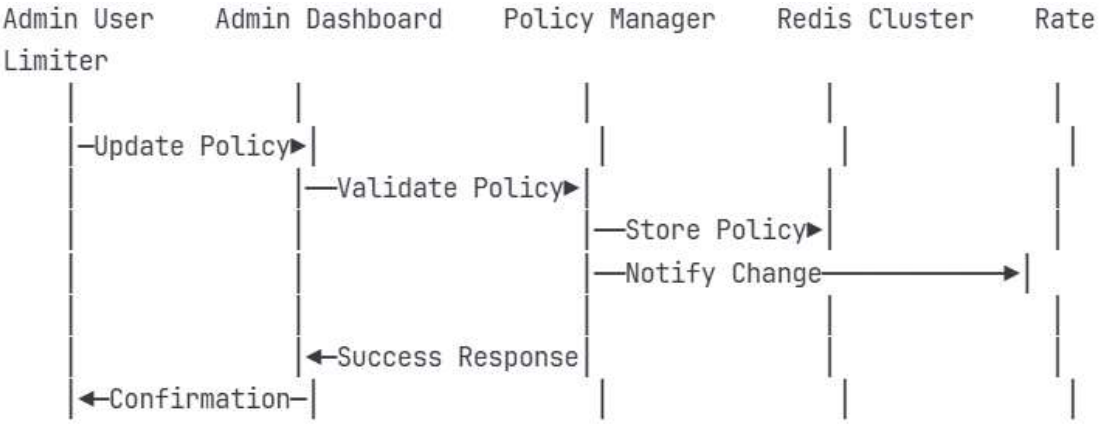
The API Rate Limiter employs a layered architecture with clear separation between presentation, business logic, and data persistence layers. This assists in performance, scalability, and maintainability through microservices patterns and distributed caching strategies.

4.2 UML Sequence Diagrams

Sequence Diagram 1: Rate Limit Check Flow



Sequence Diagram 2: Policy Update Flow



4.3 API Design

Rate Limiter Service API

Endpoint: /api/v1/ratelimit/check

Method: POST
Description: Check if request should be rate limited

Request:

json

```
{
  "clientId": "api_key_123",
  "resource": "/api/users",
  "method": "GET",
  "clientIp": "192.168.1.100"
}
```

Response (Success):

json

```
{
  "allowed": true,
  "limit": 1000,
  "remaining": 999,
  "resetTime": 1694678400,
  "retryAfter": null
}
```

Response (Rate Limited):

json

```
{
  "allowed": false,
  "limit": 1000,
  "remaining": 0,
  "resetTime": 1694678400,
  "retryAfter": 60
}
```

Errors:

- 400 Bad Request: Invalid request format
- 401 Unauthorized: Invalid API key
- 503 Service Unavailable: Rate limiter service unavailable

Endpoint: /api/v1/policies

Method: POST
Description: Create new rate limiting policy

Request:

json

```
{
  "name": "Premium User Policy",
  "clientType": "premium",
  "limits": [
    {
      "resource": "*",
      "method": "*",
      "limit": 10000,
      "window": 3600,
      "algorithm": "sliding_window"
    }
  ]
}
```

Response:

json

```
{
```

```
  "policyId": "pol_456",
  "status": "created",
  "message": "Policy created successfully"
}
```


Errors:

- 400 Bad Request: Invalid policy configuration
- 409 Conflict: Policy name already exists
- 403 Forbidden: Insufficient permissions

4.4 Error Handling, Logging & Monitoring

Error Handling Strategy

- **Graceful Degradation**: Allow requests when rate limiter is unavailable
- **Circuit Breaker**: Prevent cascade failures to backend services
- **Standardized Errors**: Consistent error response format across all APIs

Logging Framework

- **Structured Logging**: JSON format with correlation IDs
- **Log Levels**: ERROR, WARN, INFO, DEBUG
- **Sensitive Data**: No credentials or personal data in logs
- **Retention**: 30 days for application logs, 90 days for audit logs

Monitoring Metrics

- **Performance**: Request latency (p95, p99), throughput (RPS)
- **Availability**: Service uptime, error rates by endpoint
- **Business**: Rate limit violations, policy usage statistics
- **Infrastructure**: Redis cluster health, memory usage, CPU utilization

Key Alerts:

- Response time > 10ms (95th percentile)
- Error rate > 1%
- Redis cluster node failure
- Memory usage > 80%

4.5 UX Design

Admin Dashboard Features

- **Intuitive Policy Builder**: Drag-and-drop interface for creating rate limiting rules
- **Real-time Monitoring**: Live dashboards showing traffic patterns and limit violations
- **Responsive Design**: Mobile-friendly interface for on-the-go management
- **Accessibility**: WCAG 2.1 AA compliance with keyboard navigation support

API Consumer Experience

- **Clear Error Messages**: Descriptive HTTP 429 responses with retry guidance
- **Rate Limit Headers**: Standard headers (X-RateLimit-Limit, X-RateLimit-Remaining)
- **Documentation**: Comprehensive API docs with rate limiting examples
- **Developer Tools**: SDK support for popular programming languages

4.6 Open Issues & Next Steps

Current Limitations

- Single-region deployment only
- Limited algorithm support (fixed window, sliding window, token bucket)
- Manual policy deployment process

Future Enhancements

1. **Multi-region Support**: Global rate limiting with consistency guarantees
2. **ML-based Anomaly Detection**: Intelligent traffic pattern analysis
3. **GraphQL Rate Limiting**: Query complexity-based rate limiting
4. **Advanced Algorithms**: Adaptive rate limiting, user behavior-based limits
5. **Integration Improvements**: Native Kubernetes ingress controller support

Technical Debt

- Refactor legacy configuration management code
- Implement comprehensive integration test suite
- Optimize Redis memory usage patterns
- Enhance monitoring dashboard performance

5. Appendices

5.1 Glossary

- **API Rate Limiting**: Controlling the number of requests a client can make to an API
- **Fixed Window**: Rate limiting algorithm using fixed time intervals
- **Sliding Window**: Rate limiting algorithm using moving time windows
- **Token Bucket**: Rate limiting algorithm using token-based consumption model
- **Circuit Breaker**: Design pattern for preventing cascade failures
- **Redis**: In-memory data structure store used for caching and messaging

5.2 References

- IEEE 42010: Systems and Software Engineering - Architecture Description
- RFC 6585: Additional HTTP Status Codes (429 Too Many Requests)
- OWASP API Security Top 10
- Redis Clustering Best Practices
- Microservices Patterns by Chris Richardson

5.3 Tools & Technologies

- **Architecture Modeling**: PlantUML, draw.io, Lucidchart
- **API Documentation**: Swagger/OpenAPI 3.0
- **Load Testing**: Apache JMeter, Artillery.io
- **Security Testing**: OWASP ZAP, Burp Suite
- **Monitoring**: Prometheus, Grafana, ELK Stack
- **Container Orchestration**: Docker, Kubernetes, Helm