Software Architecture and Design Specification (SAD)

Project: API Rate Limiter

Version: 1.0

Authors: QuadCore

Date: 18-09-2025

Status: Draft

Team mem: Ankita muni PES1UG23CS081
Arya S PES1UG23CS107
Manasa S PES1UG24CS811
Likitha H PES1UG24CS810

1. Introduction

1.1 Purpose

This document specifies the architecture and design of the API Rate Limiter system. The purpose is to ensure backend services are protected from abuse, denial-of-service attacks, and excessive API consumption, while enforcing fairness among clients.

1.2 Scope

Covers API request tracking, limit enforcement, error handling, logging, monitoring, and configuration. Excludes backend API business logic.

1.3 Audience

DevOps Engineers, Frontend and Backend Developers, API Consumers, Security Analysts, QA/ Test Engineers

1.4 Definitions

API: Application Programming Interface
Rate Limiting – Restricting API calls per time unit
Token Bucket/Leaky Bucket – Algorithms for rate control
HTTP 429 – Standard error code for “Too Many Requests”
Retry-After – Header to indicate cooldown time
Redis – In-memory store for counters

2. Document Overview

2.1 How to use this document

This SAD provides a complete architectural view of the API Rate Limiter, including goals, component diagrams, sequence flows, API design, error handling, and security architecture.

2.2 Related Documents

SRS v1.0 – Software Requirements Specification
STP v1.0 – Software Test Plan
RTM – Requirements Traceability Matrix

3. Architecture

3.1 Goals & Constraints

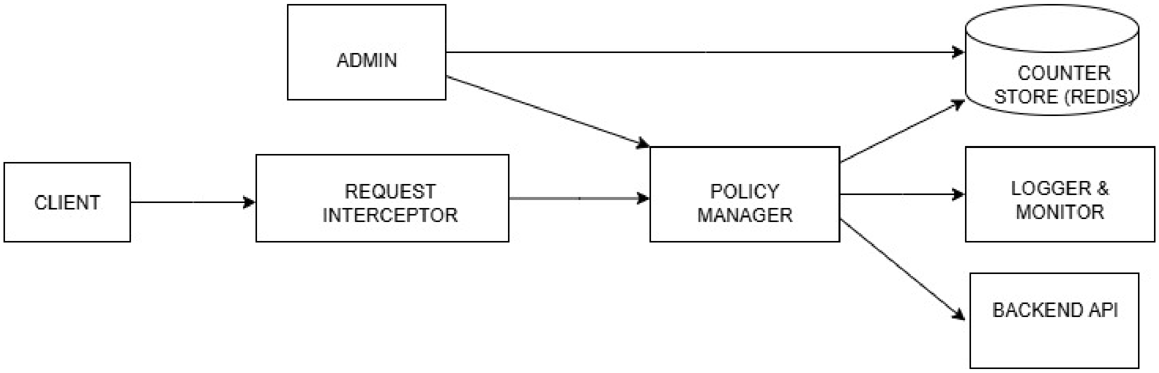
Goals:
- Enforce configurable rate limits
- Ensure <1 ms latency overhead per request (p99)
- Achieve 99.99% uptime
- Provide fair usage and prevent abuse

Constraints:
- Stateless servers with distributed counters
- Must scale horizontally
- must integrate with REST APIs
- handle 1M+ requests/day

3.2 Stakeholders & Concerns

API Clients, Admins/DevOps, Security Teams, Developers

3.3 Component (UML) Diagram



3.4 Component Descriptions

- Request Interceptor: Sits in front of APIs, checks request counts.
- Policy Manager: Implements algorithms (fixed, sliding, token bucket).
- Counter Store: Distributed backend (Redis) for consistent state.
- Admin API: REST endpoints for configuring limits and viewing stats.
- Logger/Monitor: Sends structured logs to monitoring tools.

3.5 Chosen Architecture Pattern and Rationale

A Layered + Middleware architecture is chosen. This ensures clear separation: interception, policy enforcement, storage, and monitoring. Microservices style is avoided to keep latency minimal.

3.6 Technology Stack & Data Stores

Languages/Frameworks: Node.js / Java Spring Boot / Python Flask
Data Store: Redis or Memcached for counters
Protocols: HTTP/HTTPS (REST)
Monitoring: Prometheus

3.7 Risks & Mitigations

- Redis downtime → fallback to local memory + auto-retry
- High load spikes → burst handling + horizontal scaling
- Config tampering → secured RBAC for Admin API
- Log overload → structured logging with sampling

3.8 Traceability to Requirements

RL-F-001 (Enforce rate limit) → Request Interceptor + Counter Store
RL-F-004 (Admin API) → Admin Module
RL-NF-005 (Latency ≤1ms) → System-wide optimization

3.9 Security Architecture (STRIDE)

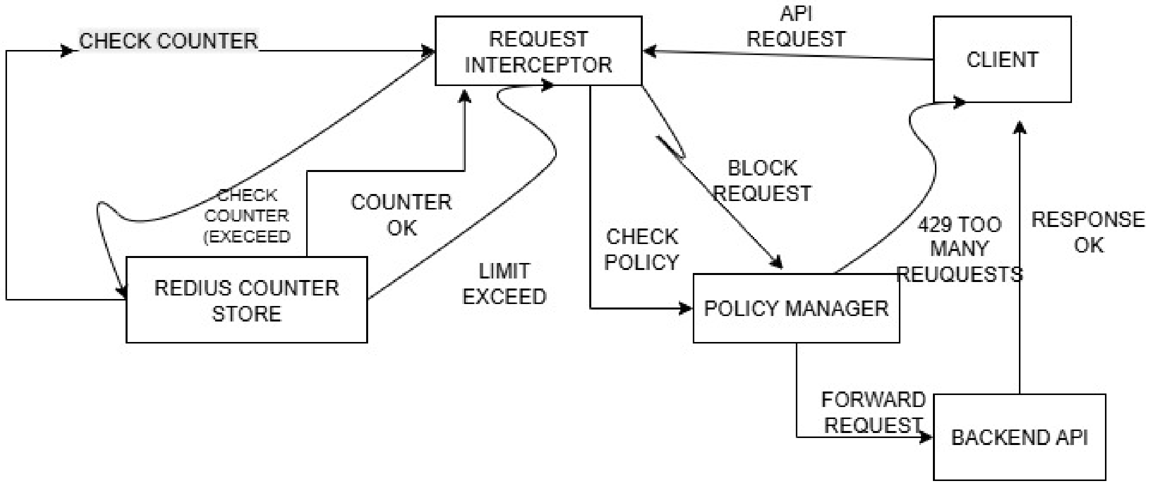
Spoofing → Authenticated clients, TLS
Tampering → Signed configs, role-based access
Repudiation → Audit logs with timestamps
Info Disclosure → No sensitive logs, TLS enforced
DoS → Rate-limiting policies, burst handling
Elevation of Privilege → RBAC for admin endpoints

4. Design

4.1 Design Overview

The system intercepts requests, checks counters in Redis, applies policies, and either forwards the request or blocks with HTTP 429.

4.2 UML Sequence Diagrams



4.3 API Design

Endpoint: /check-limit
Method: GET
Response: {status: allowed/blocked, remaining: int, reset_time: timestamp}
Errors: 429 Too Many Requests

Endpoint: /admin/config
Method: POST
Request: {endpoint: '/api/orders', limit: 100, window: '1m'}
Response: {status: success}
Errors: 401 Unauthorized, 403 Forbidden

4.4 Error Handling, Logging & Monitoring

- Standardized JSON error responses.
- No sensitive info in logs.
- Logs include: client ID, timestamp, endpoint, status.
- Metrics: % requests blocked, average latency, top clients by usage.

4.5 UX Design

Client-facing: Clear error responses with retry-after
Admin-facing: REST APIs + optional dashboard for live statistics

4.6 Open Issues & Next Steps

Future support for adaptive rate limiting using ML models
Distributed policies across multi-cloud environments
Integration with API gateways (Kong, Nginx, Envoy)

5. Appendices

5.1 Glossary

API, Rate Limiting, HTTP 429, Token Bucket, Redis, RBAC

5.2 References

IEEE 42010
OWASP API Security Top 10
NIST SP 800-160

5.3 Tools

PlantUML, draw.io (for diagrams)
Swagger/OpenAPI (for API contracts)