Software Architecture and Design Specification

Project: API Rate Limiter

Version: 1.1

Authors: Raunak Bagaria(PES1UG23CS475), R Jeevith(PES1UG23CS457), Rishav Ghosh(PES1UG23CS478), Roshit Sharma (PES1UG23CS489)

Date: 15-10-2025

Status: Draft

Revision History

Version	Date	Author	Change Summary
1.0	12-10-2025	Rishav Ghosh	Initial draft. Defined "Pipe and Filter" architecture, Admin API, and LLDs (ARL-3, ARL-4) based on a database config.
1.1	15-10-2025	R Jeevith	Re-architected to a "Multi-Layered Policy" model. Replaced the Admin API with a `policies.csv` System of Record. Updated all diagrams and LLDs to reflect the new atomic, multi-policy check logic.

Approvals

Role	Name	Signature/Date

Developer	Raunak	12-10-2025
QA Lead	Rishav	16-10-2025
Testing Engineer	Roshit	16-10-2025
Developer	Raunak	16-10-2025

1. Introduction

1.1 Purpose

This document specifies the architecture and design of the API Rate Limiter System, a middleware service that enforces rate limits on API endpoints to prevent abuse, ensure fair usage, and protect backend services from excessive traffic and denial-of-service attempts.

1.2 Scope

This covers what the API Rate Limiter does, like limiting client requests and protecting backend APIs, but not internal server details.

1.3 Audience

Developers, QA testers, security teams, and managers who need to understand or work on the system.

1.4 Definitions

API: Application Programming Interface

Rate Limiting: Controlled restriction of API request rates within defined time intervals

Token Bucket: Algorithm that allows bursts of traffic up to a maximum rate

Sliding Window: Rate limiting algorithm that maintains a rolling time window

Fixed Window: Rate limiting algorithm using discrete time periods

CIDR: Classless Inter-Domain Routing notation for IP address ranges

TLS:A security lock that encrypts data (e.g., API keys, requests) moving between clients, the limiter, and backend servers.

2. Document Overview

2.1 How to use this document

This document serves as the architectural and design reference for the API Rate Limiter System.

It provides:

- **Architecture Deliverables**: UML diagrams, component descriptions, chosen design patterns, and security considerations.
- **Design Specifications**: Sequence diagrams, API designs, error-handling strategies, and monitoring practices.
- **Traceability**: Links requirements from the SRS and backlog to architectural components.

Intended Audience:

- **Developers** → Use this document to implement and integrate system components.
- **Testers/QA Engineers** → Refer to it for designing test plans based on the architecture.
- **System Administrators/Operators** → Understand configuration, monitoring, and scalability aspects.
- **Stakeholders & Evaluators** → Verify whether the design meets business and technical goals.

2.2 Related Documents

- **Software Requirements Specification (SRS)** – API Rate Limiter System (defines functional and non-functional requirements)
- **Product Backlog (Jira)** – API Rate Limiter (user stories, epics, acceptance criteria)
- **Requirements Traceability Matrix (RTM)** – Links functional requirements and non-functional requirements to architecture and test cases
- **Software Test Plan (STP)** – Defines testing strategy, tools, and validation process (future document).

3. Architecture

3.1 Goals & Constraints

Goals :

- Protection: Shield backend APIs from abuse and DoS attacks
- Performance: Add minimal latency overhead (<10ms per request)
- Scalability: Handle >1000 requests/minute
- Reliability: Achieve 99.9% availability
- Flexibility: Support multiple rate limiting algorithms and client identification methods

Constraints :

- Must operate as middleware/proxy service
- TLS 1.2+ required for all external communications
- Rate limit accuracy must be within 5% deviation

3.2 Stakeholders & Concerns

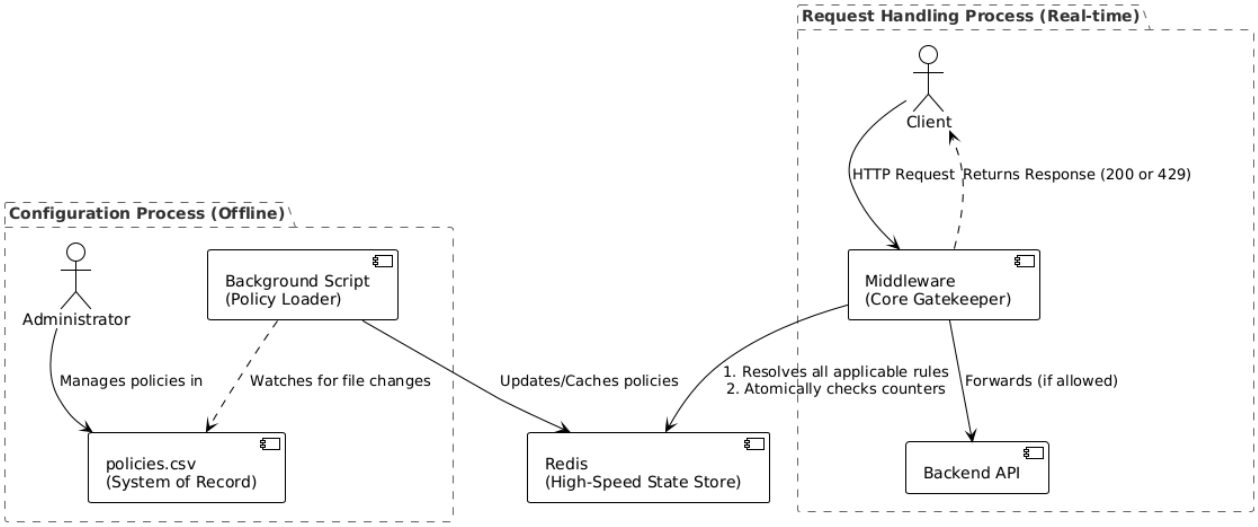
API Consumers : Predictable performance, clear feedback on rate limits, fair usage allocation

API Providers : Backend service protection, availability guarantees

Administrators/Operators : Configuration management, monitoring capabilities, alert systems, policy control

Developers : Modularity, maintainability, and extensibility for new algorithms

3.3 Component (UML) Diagram



3.4 Component Descriptions

Our architecture is built on three distinct components that work together to execute our multi-layered policy design. The core logic from our original user stories (like client identification and rate limiting) is now consolidated into the central `Middleware` component for high efficiency.

1. The Middleware (Express.js): The Core Gatekeeper This is the central nervous system of our operation. It's a lean Express.js function that intercepts every single request. This one component is where the logic from your user stories, which you previously mapped to separate components, now lives:

- It acts as the **`Client Identifier`** (fulfilling requirements F-001, F-002, F-003) by inspecting the request to extract the API Key, source IP, and URL path.
- It acts as the **`Rate Limit Engine`** (fulfilling F-004, F-005, F-006) by first looking up all matching policy IDs from Redis (e.g., a user's tier policy *and* an endpoint protection policy) and then executing a single atomic check against all their counters.
- It acts as the **`Proxy Engine`** (fulfilling F-007) by making the final "allow" or "deny" decision.
- When a request is denied, it's responsible for generating the correct response, including **custom error messages** (F-015) and the informative **response headers** (F-009) like `X-RateLimit-Remaining`.
- It also acts as the source for **`Monitoring & Analytics`** by collecting metrics on allowed/denied requests and triggering alerts (F-008, F-010), which can then be fed to a dashboard or logging system.

2. CSV File Store: The System of Record This component is our new, simplified source of truth for all policies. This file-based approach **replaces** the need for the original `Admin API` and `Configuration Manager`.

- **Role:** Its role is purely administrative. Administrators can edit this file to manage policies and manual blocks (F-011, F-014).
- **Decoupling & Logic:** The middleware *never* reads this file directly. A separate background script is responsible for watching this file. This script handles the crucial logic of **configuration validation** (F-012) and applying **rule precedence** (F-013) before loading the policies into Redis.

3. Redis: The High-Speed State Store (Using in-memory hash maps for now) This is the high-performance engine that enables the middleware to make sub-millisecond decisions. It is the concrete implementation of the `Data Store Interface` from the original design.

- **Policy Caching:** It holds the cached policies and lookup sets that are loaded from the CSV file.
- **Live Counter Management:** It stores the "token buckets" (live counters) for every single policy. We use a Redis Lua script to guarantee that checking and consuming tokens is an atomic operation, which is essential for accuracy.

3.5 Chosen Architecture Pattern and Rationale

Chosen Pattern: Multi-Layered Policy Architecture

Rationale: We have replaced the original "Pipe and Filter" architecture with a more powerful **Multi-Layered Policy Architecture**.

The core principle of this new design is to treat rate limiting not as a single check, but as a system of **cascading, multi-layered rules**. Every incoming API request is evaluated against all relevant policy layers it matches. Access is only granted if the request satisfies **every single applicable rule**.

This pattern was chosen because it allows us to effectively enforce two different types of policies simultaneously:

1. **Commercial & Identity Policies:** These are the broad service tiers, such as a "Free Tier" or "Pro Tier," which are typically tied to a user's API key .
2. **Resource & Operational Policies:** These are specific, surgical rules designed to protect resources, such as limiting requests to a single slow endpoint (e.g., `/api/v1/uploads`) or blocking a problematic IP range

This design solves a key problem: a "Pro Tier" user with a high general limit (e.g., 5000 requests/hour) will *still* be correctly blocked if they violate a separate, low-limit policy (e.g., 10 requests/hour) on a protected endpoint . This ensures that our operational safeguards and resource protection policies are always enforced, regardless of a user's commercial tier.

3.6 Technology Stack & Data Stores

Backend Service: Node.js, Express for implementing the rate-limiting engine.

In-Memory Data Store: Redis (currently using in-memory hash maps)– used for fast request counting, TTL-based token buckets, sliding windows, and distributed synchronization.

Persistent Store (System of Record): A CSV text file (`policies.csv`) is now the human-readable source of truth for all rate-limiting policies.

Persistent Database (Optional): PostgreSQL may still be used for audit logs, but it is *not* the system of record for configuration.

Frontend (Dashboard): React/Nextjs – for a dashboard to visualize rate limits, request statistics, and logs (not for configuration).

Security: TLS (1.2+) for all external communication, AES-256 encryption for sensitive configuration data at rest, and role-based access control (RBAC) for admin endpoints.

3.7 Risks & Mitigations

Risk: Configuration corruption -> Mitigation : Configuration validation, rollback mechanisms

Risk: DDoS bypassing rate limits -> Mitigation : Multiple identification methods, IP blocklists

3.8 Traceability to Requirements

- **Middleware (Core Gatekeeper):** This single component is responsible for all real-time logic. * **Client Identification:** F-001, F-002, F-003 * **Rate Limiting:** F-004, F-005, F-006 * **Proxying & Error Response:** F-007, F-015 * **Monitoring & Response Headers:** F-008, F-009, F-010
- **CSV File Store & Background Script:** This component handles all configuration logic. * **Configuration Management:** F-011, F-012, F-013, F-014

3.9 Security Architecture

Threat Modeling (STRIDE Analysis):

Spoofing Identity:

Threat: Clients impersonating others to bypass limits

Mitigation: API key validation (F-001), IP verification (F-002)

Tampering with Data:

Threat: Malicious modification of rate limit counters

Mitigation: TLS encryption, Redis AUTH, request signing

Repudiation:

Threat: Denying excessive API usage

Mitigation: Comprehensive audit logging, tamper-proof logs

Information Disclosure:

Threat: Exposing rate limiting policies or client patterns

Mitigation: TLS for all communications. Policy configuration is managed via an offline CSV file, not an exposed API, which eliminates this online attack vector.

Denial of Service:

Threat: Overwhelming the rate limiter itself

Mitigation: The configuration process is offline and decoupled from live requests. The middleware itself is a lean function, and resource monitoring (CPU, Redis connections) is in place.

Elevation of Privilege:

Threat: Gaining admin access to modify policies

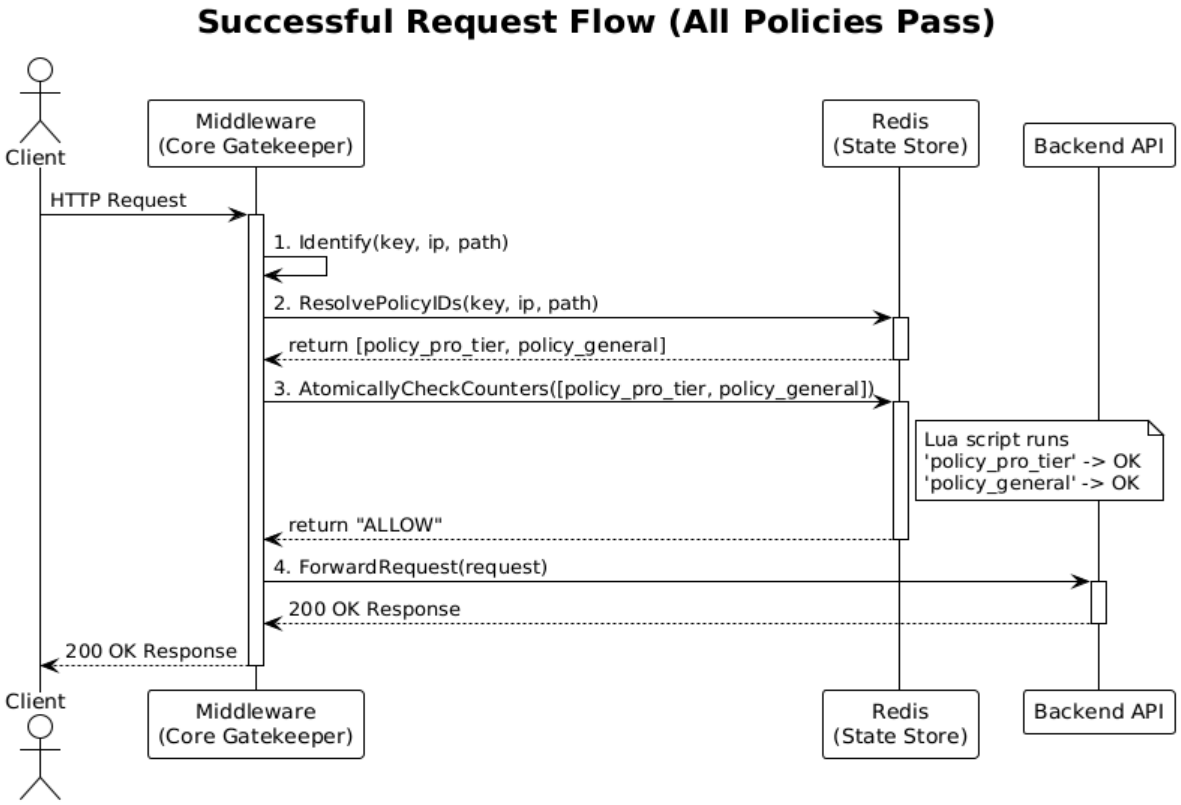
Mitigation: Access control on the file system (file permissions) and code repository where policies.csv is managed.

4. Design

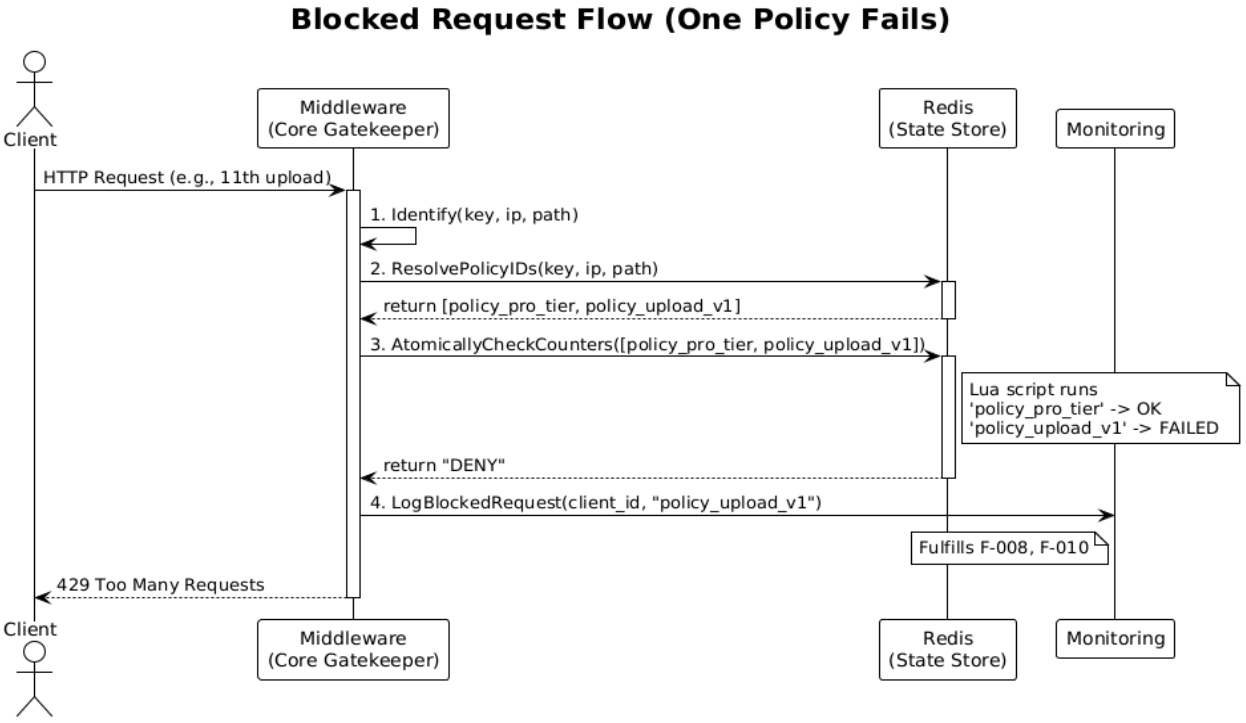
4.1 Design Overview

An API Rate Limiter acts as a middleware component positioned in front of backend APIs. Its function is to intercept incoming requests and apply a specific algorithm (like token bucket or fixed window) to limit the number of requests a client can make in a given timeframe. The design prioritizes being performant, scalable, and resilient. The system uses a data store to track request counts per client and has a built-in mechanism to either block or delay requests that exceed the set limits

4.2 UML Sequence Diagrams



Successful Request Flow



Blocked Request Flow

4.3 API Design

4.3.1 Rate Limiter Proxy Interface

Endpoint: /* (All routes proxied)

Method: ANY
Headers:

- Authorization: Bearer <token> (for API key identification)
- X-Real-IP: <client-ip> (for IP-based identification)

Success Response (Within limits):
HTTP 200 OK

X-RateLimit-Limit: 1000

X-RateLimit-Remaining: 847

X-RateLimit-Reset: 1642676400

{ "data": "Response from backend API"}

Rate Limited Response:

HTTP 429 Too Many Requests

X-RateLimit-Limit: 1000

X-RateLimit-Remaining: 0

X-RateLimit-Reset: 1642676400

Retry-After: 60

{

"error": "Rate limit exceeded",

"message": "Too many requests. Please try again later.",

"reset_time": "2022-01-20T10:00:00Z"

}

4.3.2 Configuration Data Model (CSV File)

The source of truth for all policy definitions is a CSV file (e.g., `policies.csv`). This file is managed by administrators and read by a background script; it is *not* read directly by the middleware during a request.

The CSV file structure must contain the following columns:

- `id`: A unique identifier for the policy.
- `name`: A human-readable name for the policy.
- `scope`: The type of rule (`api_key`, `endpoint`, or `ip`).
- `identifier`: The value to match (e.g., `PRO_KEY_*`, `/api/v1/uploads/*`, or `203.0.113.0/24`).
- `limit`: The number of requests allowed.
- `window_seconds`: The time window for the limit.
- `priority`: A number to resolve conflicts (lower numbers can be higher priority).

Example `policies.csv` structure:

id	name	scope	identifier	limit	window_seconds	priority
policy_free_tier	Free Tier Users	api_key	FREE_KEY_*	100	60	20
policy_pro_tier	Pro Tier Users	api_key	PRO_KEY_*	5000	3600	10
policy_upload_v1	Protect Upload Endpoint	endpoint	/api/v1/uploads/*	10	3600	5

policy_sec_ip_blk	Security Block for Office IP	ip	203.0.113.0/24	20	60	1

4.4 Error Handling, Logging & Monitoring

When a client exceeds the rate limit, the system should return a specific HTTP status code to inform the client of the error. The standard code for this is 429 Too Many Requests. The response should also include informative headers to guide the client on how to handle the situation.

Logging: The system should log every request that is rate-limited, including details like the timestamp, client identifier (e.g., API key or IP address), and the specific rate-limiting rule that was triggered. This log data is vital for post-incident analysis and for fine-tuning the rate-limiting rules. Logging sensitive information should be avoided.

Monitoring: The rate limiter's performance should be monitored with metrics such as:

- Rate-limited requests: The number of requests that are blocked or delayed over time.
- Request patterns: Tracking average requests per user and identifying unusual traffic spikes.
- Error rates: Monitoring the frequency of 429 errors.
- System load: Observing server performance metrics like CPU usage, especially during high-traffic periods

4.5 UX Design

The UX design for an API Rate Limiter focuses on making it easy for developers to work with. It's not about a visual interface, but about clear communication.

- Clear Errors: The API should return a specific 429 Too Many Requests status code when a client is rate-limited.
- Informative Headers: The response should include headers like X-RateLimit-Limit, X-RateLimit-Remaining, and X-RateLimit-Reset to give developers real-time status updates.
- Good Documentation: The API's documentation should clearly explain the rate-limiting rules, helping developers build their applications to respect the limits.

4.6 Open Issues & Next Steps

OPEN ISSUES

Security: A rate limiter acts as a security measure to defend against malicious attacks, such as brute-force attacks, DDoS attacks, and web scraping. By limiting the number of requests from a single source, it makes it harder for attackers to overwhelm the system.

Resource Management: APIs have finite resources. Rate limiting prevents a single user or bot from monopolizing those resources, ensuring the API remains available and performs well for all users.

Cost Control: For APIs that charge on a per-call basis, rate limiting is essential to prevent unintended overages and control costs.

Quality Control: It ensures a fair distribution of service and maintains a high level of performance and dependability for the API.

NEXT STEPS

Move to Dynamic and Adaptive Rate Limiting: Instead of using fixed, static limits, a key next step is to implement a dynamic rate limiter that can adjust thresholds in real-time based on factors like server load, CPU usage, and user behavior. This approach can more effectively handle traffic spikes and prevent a denial of service (DoS).

Provide Granular Control: Future rate limiters will offer more granular control by applying different limits based on user tiers (e.g., free vs. paid), specific endpoints, or the complexity of a request.

Integrate AI and Machine Learning: A future direction is to use machine learning to analyze API usage patterns and detect unusual behavior. AI can help identify and proactively adjust limits to prevent abuse before it happens.

5. Appendices

5.1 Glossary:

- **API (Application Programming Interface):** A set of endpoints that allows communication between applications.
- **Rate Limiting:** Technique to control the number of requests a client can make within a defined timeframe.
- **Token Bucket / Leaky Bucket:** Algorithms used for implementing rate limiting strategies.
- **Client:** An application, user, or service consuming backend APIs.
- **Administrator:** Role responsible for configuring, monitoring, and managing the system.

- **Redis / Memcached:** In-memory datastores used for request tracking and distributed coordination.
- **HTTP 429 (Too Many Requests):** Standard response code when a client exceeds its rate limit.

- **HTTP 403 (Forbidden):** Response code used for blocked sources (e.g., IP blocklist).
- **TLS (Transport Layer Security):** Protocol ensuring encrypted communication.

5.2 References:

Standards & Guidelines

- IEEE 42010 – Recommended practice for software architecture documentation.
- NIST SP 800-160 – Systems security engineering best practices.
- OWASP API Security Top 10 – Security risks and mitigations for APIs.

Protocols & Specifications

- RFC 6585 – Defines HTTP 429 (Too Many Requests).
- TLS 1.2/1.3 – Secure communication protocols.

Technologies & Tools

- Redis Documentation – In-memory datastore for rate limiting counters.
- Kubernetes & Docker Documentation – Deployment and scaling references.
- Prometheus & ELK Stack Documentation – Monitoring and logging tools.

5.3 Tools:

- **PlantUML / Draw.io** – UML modeling and system diagrams.
- **Swagger / OpenAPI** – API documentation and design.
- **Prometheus & ELK Stack** – Monitoring, logging, and analytics.
- **JUnit / Postman** – Automated and manual API testing.
- **Git + CI/CD Pipeline (Jenkins/GitHub Actions)** – Version control and continuous deployment.
- **Load Testing Tools (JMeter / Locust)** – Performance and stress testing.

LOW LEVEL DESIGN

A Deeper Look into Policy Strategy

The power of this design comes from how we define and combine different types of policies within the CSV file.

1. API Key Policies: The Commercial & Identity Layer

- **Purpose:** These policies are the primary tool for implementing your business logic and tiered service levels. They answer the question: "What general level of service has this user been assigned?".
- **How it Works:** You create one row in the CSV for each service tier, using a common prefix or wildcard in the `identifier` column for the API keys in that tier. (e.g., `scope=api_key`, `identifier=PRO_KEY_*`).

2. Endpoint Policies: The Resource Protection Layer

- **Purpose:** These are specialized, surgical rules designed to protect specific resources. They answer the question: "Is this particular part of our system sensitive, slow, or expensive to run?".
- **How it Works:** You apply these policies directly to URL paths by adding a row with the `scope` set to `endpoint`. This is an operational safeguard, independent of any user's tier. (e.g., `scope=endpoint`, `identifier=/api/v1/uploads/*`).

(Note: This same logic applies to IP-based policies, replacing the old ARL-4 logic by simply adding a row with `scope=ip`).

How This Works: Practical Scenarios

Let's trace two distinct scenarios to see how the system behaves under different conditions.

Scenario 1: The Pro User Accessing a Protected Resource This scenario demonstrates how the resource protection layer (Endpoint Policy) correctly overrides the commercial layer (API Key Policy).

- The Setup (from `policies.csv`):
    - `policy_pro_tier`: A Pro User has an API key (e.g., `PRO_KEY_123`) granting 5000 requests/hour.
    - `policy_upload_v1`: The `/api/v1/uploads/*` endpoint is protected by a policy allowing only 10 requests/hour.
- **The Action:** The Pro User, having made only 100 requests so far this hour, attempts to make their 11th upload.
- **System Evaluation:**
    a. The middleware intercepts the request and identifies two applicable policies from the Redis cache: `policy_pro_tier` (from the API key) and `policy_upload_v1` (from the URL path).

    b. It asks Redis to atomically check the token buckets for *both*.
    c. The check for the `policy_pro_tier` (5000/hr) **passes**.
    d. The check for the `policy_upload_v1` (10/hr) **fails** (it's the 11th request).
- **Result:** Because one of the checks failed, the entire operation is **denied**. The user receives a `429 Too Many Requests` response, correctly prioritizing the protection of the specific resource over the user's general tier.

Scenario 2: A Tiered User Under an IP-Based Security Restriction This scenario shows how the system can apply dynamic security rules that stack with commercial tiers.

- **The Setup (from `policies.csv`):**
    a. `policy_free_tier`: A Free User has a limit of 100 requests/minute.
    b. `policy_sec_ip_blk`: To mitigate a bot attack, an administrator adds a row to the CSV limiting an IP range (e.g., `203.0.113.0/24`) to 20 requests/minute.
    c. A background script detects the change and updates Redis.
- **The Action:** The Free User, working from an office within that IP range, attempts to make their 21st request in 30 seconds
- **System Evaluation:**
    a. The middleware identifies two applicable policies: `policy_free_tier` (from the API key) and `policy_sec_ip_blk` (from the source IP).
    b. It asks Redis to check both buckets.
    c. The check for the `policy_free_tier` (100/min) **passes**.
    d. The check for the `policy_sec_ip_blk` (20/min) **fails**.
- **Result:** The request is **blocked**. This demonstrates how the system can be used for dynamic security enforcement by simply editing and saving a text file.

Middleware LLD: Request Evaluation Logic

This section replaces the old `ARL-3` and `ARL-4` logic. Instead of separate services, all validation logic is consolidated into the `Middleware (Core Gatekeeper)`. The process for every request is as follows:

1. **Identification:** The middleware first inspects the request to extract all possible identifiers.
    - It reads the `Authorization` header for an API Key.
    - It reads the `X-Real-IP` (or socket IP) for the source IP address.
    - It reads the `URL path`.
2. **Rule Resolution:** The middleware performs high-speed lookups against Redis to find all policies that match the request's identifiers.
    - It queries a Redis Set/Hash for the API Key (e.g., `key:PRO_KEY_123`) to get `[policy_pro_tier]`.

    - It queries a Redis Set/Hash for the URL Path (e.g., `path:/api/v1/uploads/*`) to get `[policy_upload_v1]`.
    - It queries a Redis store for the IP to find any matching CIDR policies (e.g., `ip:203.0.113.0/24`) to get `[policy_sec_ip_blk]`.
    - It collects all unique policy IDs found. For the "Pro User" in Scenario 1, this list would be: `[policy_pro_tier, policy_upload_v1]`.
3. **Atomic Evaluation:** The middleware does *not* check each policy one by one. Instead, it bundles all applicable policy IDs (`[policy_pro_tier, policy_upload_v1]`) and executes a **single, atomic operation** against Redis.
    - This is implemented as a **Redis Lua script**.
    - The script iterates through the list of policy IDs.
    - For each ID, it checks the corresponding token bucket counter.
    - If *any* counter has reached its limit, the script immediately stops and returns "DENY" .
    - If *all* counters are within their limits, the script decrements all of them and returns "ALLOW".
4. **Decision & Action:** The middleware receives the single "ALLOW" or "DENY" signal from Redis.
    - **On "ALLOW":** It forwards the request to the Backend API.
    - **On "DENY":** It immediately terminates the request, logs the event to Monitoring , and returns a `429 Too Many Requests` error to the client.