

API Rate Limit Adoption – A pattern collection

Souhaila Serbout
Software Institute, USI
Lugano, Switzerland
souhaila.serbout@usi.ch

Cesare Pautasso
Software Institute, USI
Lugano, Switzerland
c.pautasso@ieee.org

Amine El Malki
University of Vienna, Faculty of Computer Science,
Software Architecture Research Group and Doctoral
School Computer Science
Vienna, Austria

Uwe Zdun
University of Vienna, Faculty of Computer Science,
Software Architecture Research Group
Vienna, Austria

ABSTRACT

The *API Rate Limit* pattern controls the rate at which clients make API requests by counting the number of requests in a specified time interval and reacting against abusive clients, in order to protect the limited resources of the API from exhaustion and denial of service attacks. This practice helps service providers to prevent abuse and ensure fair resource allocation, maintain system stability, monitor and control service availability, protect against DDoS attacks

In this research paper, we have identified patterns covering the *API Rate Limit* pattern adoption starting from its documentation to its implementation.

Our objective is to elucidate the trade-offs associated with different identified patterns and offer guidance to developers in making informed decisions when choosing the most suitable Rate Limit method, scope, and granularity for their service. By providing a comprehensive overview of how to adopt the Rate Limit pattern, this paper aims to enhance the understanding of how APIs can be designed to facilitate high scalability, security, reliability, and service availability. Furthermore, we present each pattern along with known uses observed in real-world APIs and technologies.

CCS CONCEPTS

• **Software and its engineering** → **Patterns**; *Designing software*.

KEYWORDS

Application Programming Interfaces, Rate Limit, Pattern Language

ACM Reference Format:

Souhaila Serbout, Amine El Malki, Cesare Pautasso, and Uwe Zdun. 2023. API Rate Limit Adoption – A pattern collection. In *28th European Conference on Pattern Languages of Programs (EuroPLoP 2023)*, July 5–9, 2023, Irsee, Germany. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3628034.3628039>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroPLoP 2023, July 5–9, 2023, Irsee, Germany

© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0040-8/23/07...\$15.00

<https://doi.org/10.1145/3628034.3628039>

1 INTRODUCTION

Many Web API design patterns and best practices have been proposed [7, 16, 17, 45, 47, 49] to improve API quality properties [5, 8] related to performance and reliability, among others [43]. This paper focuses on the Rate Limit pattern. Its primary goal is to avoid excessive usage of API resources by specific API clients. This avoids overwhelming the API backend and prevents its subsequent non-availability and reduced performance [11]. It can also improve other quality properties, such as security-related properties [6, 18, 42].

While API Rate Limit is a prevalent aspect of Web API management [9, 36], not all systems implement it similarly. Some may not even come equipped with this feature. The absence of support for required rate-limiting techniques in some cloud-based API management systems can create challenges for API providers in terms of implementation and enforcement of rate limits [42]. In such scenarios, API providers may resort to manual implementation or utilize specialized tools to guarantee that their APIs have proper rate limits to deter excessive usage, defend against denial-of-service attacks, and mitigate similar security risks.

Furthermore, a lack of shared understanding or standardization exists when it comes to comprehending the various options for configuring API Rate Limits [11]. This absence of common ground arises from the fact that API providers often implement different approaches to rate limiting, resulting in variations of configuration choices and terminology. Consequently, developers encounter difficulties in fully understanding the intricacies of *API Rate Limit* settings across different APIs. Additionally, the absence of a standard also leads to inconsistencies not only in selecting the appropriate configuration and implementation mechanism but also in the documentation of these aspects. This presents a challenge for developers in understanding the Rate Limit policies of different APIs and their corresponding impact on their applications, as they are often not documented in a consistent format, as indicated by the analysis conducted in the context of this research. As a result, because of the lack of proper understanding of the details of the Rate Limit policy, developers may inadvertently surpass the defined thresholds, causing their client applications to fail due to measures implemented by API providers to prevent abusive behaviors. These inconsistencies in documenting rate limits also make it difficult to conduct comprehensive systematic studies on the various rate-limiting configurations adopted in real-world systems.

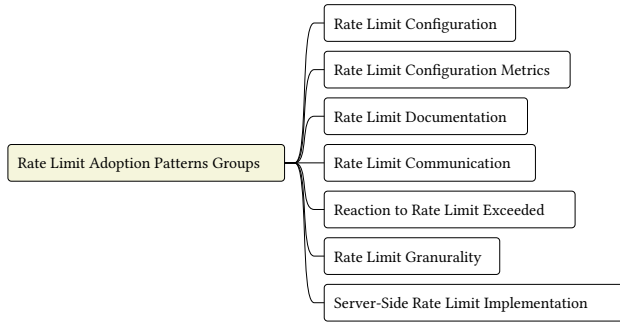


Figure 1: Rate Limit Patterns groups Abstract Overview

This paper introduces an analysis of the prevailing patterns used by developers to describe Rate Limit configurations and strategies, as well as their known uses in real-world APIs. Furthermore, we have conducted a comprehensive examination of the scope and the level of granularity at which Rate Limit constraints can be enforced. Additionally, we have identified multiple implementation-related patterns and provided a compilation of notable technological examples as known-uses.

The identified patterns in the collection pertain to the configuration and values of Rate Limit policies, the scope and granularity level of the enforced policies, the measures employed to counteract abusive clients, and different server-side implementation patterns (Figure 1). We structure the API rate limit adoption pattern collection by grouping together patterns with a common purpose.

The rest of this paper is organized as follows. In Section 2, we define the general Rate Limit pattern. The following sections cover different key aspects of the Rate Limit pattern: Rate Limit configuration (Section 3), configuration metrics (Section 4), documentation (Section 5), communication to the users (Section 6), granularity (Section 7), providers reaction to block or mitigate abusive behaviors (Section 8), and server-side implementation (Section 9). We outline the methodology we followed to define patterns in Section 10. We review related work on Web API patterns and rate limits for Web APIs in Section 11 before drawing some conclusions in Section 12.

2 THE API RATE LIMIT PATTERN

In this section, we provide a shortened version of the Rate Limit pattern, which is the main background of our work. For the complete pattern text, please refer to the *Patterns for API Design* book [49, p. 411]. The adoption patterns documented in the rest of this paper use by default the Context, Problem, and Forces sections reported here.

Rate Limit Pattern

Context: An API contract has been established with clients. An API Description specifying message exchange patterns and protocols has been defined. The API may also be offered without any contractual relationship.

Problem: How can the API provider prevent excessive usage by clients that may harm the provider’s operations or other clients?






Forces: There are several forces to consider when implementing a Rate Limit. These include the *economic aspects*, such as the cost

of implementing and maintaining prevention measures and the potentially negative reactions from clients [13]. *Performance* is a factor, as the service provider may want or be required to maintain high-quality service for all clients. *Reliability* is important, as actions must be taken to prevent API abuse from harming other clients. The *impact and severity of the risks of API abuse* must also be analyzed and weighed against the costs of prevention measures. Additionally, *client awareness* is important, as responsible clients need to be aware of their usage allowances to avoid being locked out of the API. Furthermore, API Rate Limiting plays a critical role in ensuring the *security* of an API system. It helps to protect against various types of attacks, including denial-of-service (DoS) attacks, which occur when an attacker sends a high volume of requests to an API in a short period, causing the API system to become overwhelmed and unavailable.

Solution: Implement a Rate Limit to prevent API clients from excessive usage.

Set the limit as a certain number of requests allowed per period. If the limit is exceeded, further requests can be declined, processed later, or served with best-effort guarantees using reduced resources. Customize the scope and period of the Rate Limit. Use tracking mechanisms such as tokens or monitoring tools to enforce the Rate Limit.

Related Patterns: The Rate Limit pattern may be included in a *Service Level Agreement* [49], and the details of the Rate Limit may be tied to the client’s subscription level as described in the *Pricing Plan* pattern [49]. In this case, the Rate Limit is used to enforce different billing levels of the Pricing Plan. Clients subject to a Rate Limit may be identified by their *API Key* [49].

In Figure 2, we summarize the  Rate Limit adoption patterns we identified in the scope of this paper, grouping them  depending on their purposes. In the patterns map, we employ logical operators to emphasize the cases where certain patterns can be effectively co-adopted , instances where their combination is incompatible , and situations where it is advisable to adopt all patterns as a standard practice .

3 RATE LIMIT CONFIGURATION

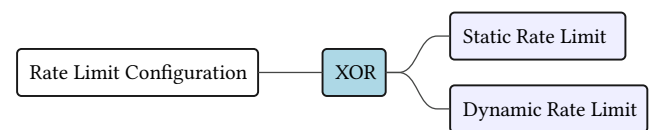


Figure 3: Rate Limit Configuration Patterns

An *API Rate Limit* is set by defining the maximum number of requests an API can receive from a specific client during a defined time frame. The time window can be measured in minutes, hours, days, or months. The Rate Limit configuration values can be static: the same value independent of the API usage, or dynamic: the value adapts to the client’s behaviors and current system capacities (Figure 3).

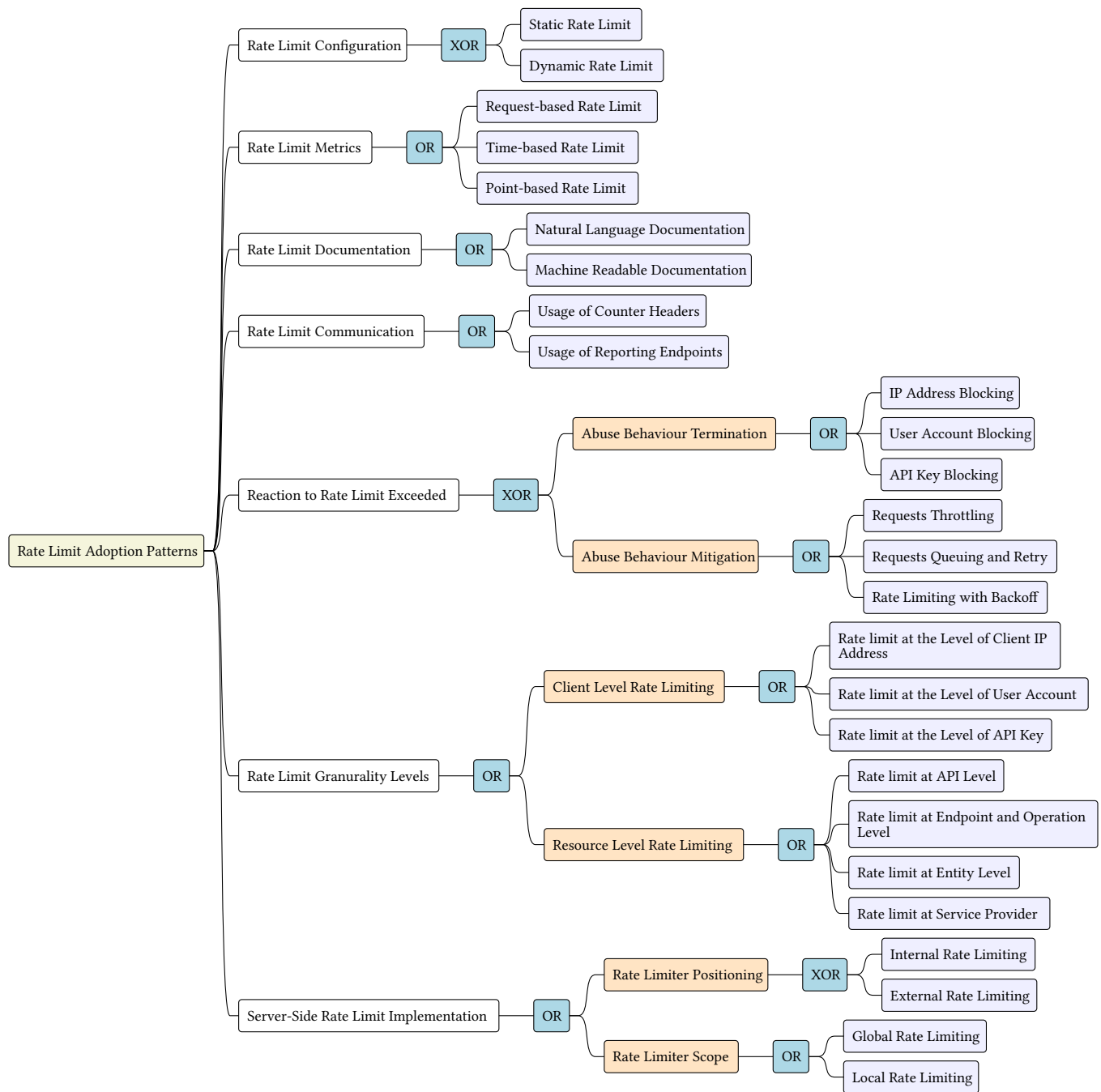


Figure 2: Rate Limit Adoption Pattern Collection Map

P 1: Static Rate Limit

Context: API providers need to set a predefined configuration of rate limits to prevent abusive consumption.

Problem: How can the API provider prevent excessive usage by clients that may harm the provider's operations or other clients, without the need for complex differentiation based on consumer behavior or characteristics?

Solution: Static rate limits are set by the API provider and remain constant regardless of the number of requests made.

Solution details: Predefined limits will be set in advance and will not change based on the current load or traffic on the network. Defining a static Rate Limit can be challenging, as it depends on various factors such as the service capacity, the expected demand, the request size and complexity, and the service level objectives (SLOs) [33]. One possible method to define a static Rate Limit is to use Little's Law [27] from queuing theory, which states that the average number of requests in a system (L) is equal to the average arrival rate (λ) multiplied by the average response time (W). Therefore, $L = \lambda W$. By rearranging this equation, we can obtain $\lambda = \frac{L}{W}$, which means that the arrival rate should not exceed the ratio of the system capacity to the response time. This can be used as a guideline to infer a static Rate Limit value for a service. However, this method assumes that the arrival rate and the response time are known, constant, and independent, which may not be true in reality. Therefore, it is advisable to monitor the service performance and adjust the Rate Limit accordingly if needed [4, 14].

Consequences:

- + *Simplicity:* Static rate limits are easy to implement and understand, as the limit is a fixed value.
- + *Economic Aspects:* A static Rate Limit is cheap to implement and maintain.
- + *Performance:* A static Rate Limit implementation has a minimum performance overhead on the server side.
- + *Client Awareness:* Clients can easily be informed and understand static rate limits.
- + *Predictability:* The limit will not change, so clients relying on the Rate Limit know exactly what to expect.
- *Economic Aspects:* A static Rate Limit can not be customized in a fine-grained manner, e.g., per customer, or adjusted to specific situations. This can have economic consequences as some clients can perceive the Rate Limit as too restrictive.
- *Inflexibility:* Static rate limits may not be able to adapt to changing traffic conditions and may cause issues if the limit is set too low or too high.
- *Client Performance and Reliability:* The client can be negatively influenced by too restrictive rate limits. As static limits cannot be adapted to specific client needs, they can have a negative impact on the performance and reliability of some clients.
- *Unfairness:* The Rate Limit may be too restrictive for some clients and too lenient for others.

P 2: Dynamic Rate Limit

Context: The demand for resources, their capacity, and the behavior of consumers can vary dynamically, requiring adaptive Rate Limit strategies.

Problem: How can an API provider or system dynamically adjust and configure rate limits for incoming requests to effectively manage resource allocation, prevent overload, and adapt to changing traffic patterns?

Solution: Adjust the Rate Limit dynamically based on conditions such as traffic or system load.

Solution details: Unlike static rate limits, which are predetermined and remain constant, dynamic rate limits are designed to be adjustable in real-time. This allows the system to respond dynamically to changes in demand, ensuring that the Rate Limit prevents resource overuse or abuse.

This approach is based on monitoring the API latency, which is the time taken to process a request [8]. A sliding window of time is used to calculate the average latency of the service, which is then compared to a target latency representing the desired level of performance.

Suppose the average latency exceeds the target latency. In that case, the Rate Limit is reduced by a specific factor, while if the average latency is below the target latency, the Rate Limit is increased by a specific factor. This allows the Rate Limit to adapt to the changing conditions of the service and maintain a reasonable quality of service [31]. This technique is inspired by feedback control theory [15] and has been applied in various domains such as web servers, cloud computing, and network traffic management.

Consequences:

- + *Adaptability:* Dynamic rate limits can adjust to changing conditions and prevent system overload.
- + *Economic Aspects:* A dynamic Rate Limit can be customized in a fine-grained manner, e.g., per customer, or adjusted to specific situations. This can have economic benefits, as the rate limits can be adjusted according to economic circumstances.
- + *Fairness:* Dynamic rate limits can ensure that resources are shared fairly among users and systems based on their current usage.
- + *Client Performance and Reliability:* Due to adaptability, the dynamic Rate Limit has the potential to help achieve good client performance and reliability.
- *Uncertainty:* Clients relying on the Rate Limit may not know what to expect, as the limit may change based on conditions outside their control.
- *Complexity:* Implementing dynamic rate limits can be more complex and require more resources, as the system must continuously monitor its performance, adjust the limits accordingly, and manage the transition between different rate limit configurations.
- *Economic Aspects:* Due to the higher complexity, a dynamic Rate Limit can require more effort to be implemented and maintained.
- *Performance:* A dynamic Rate Limit implementation has a higher performance overhead on the server side than static patterns.

4 RATE LIMIT CONFIGURATION METRICS

The configuration of the Rate Limit value is set based on a specific metric. Based on our analysis, we identified three configuration

metrics-related patterns: Request-based, Time-based, and Point-based (Figure 4).

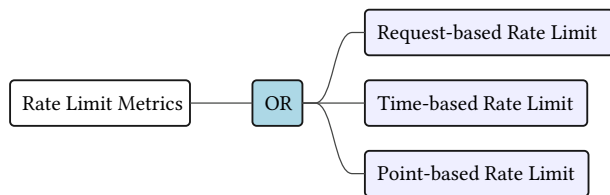


Figure 4: Rate Limit Metrics Patterns

P 3: Request Based Rate Limit

Problem: How to effectively control the frequency of API requests made by a client within a specific timeframe?

Solution: Use a request-based rate-limiting strategy to measure and limit the frequency of API requests made by specific clients.

Solution details: A Request-based Rate Limit can be implemented by using a rate limiter library or tool (such as `express-rate-limit`¹ or `koa-ratelimit`²), that allows to divide time into smaller windows (e.g. second, minute, or hour) and restrict the number of requests that can be made within those windows based on capacity and projected traffic. The rate limiter would keep track of the number of requests made by each API consumer and restrict the requests once the limit is reached. The implementation should consider adopting a specific solution for when a client reaches the limit (see Section 8).

Consequences:

- + *Predictability:* Clients can more easily predict their API usage and adjust their behavior accordingly.
- + *Implementation Simplicity:* A request-based system can be more efficient than other types of rate limiting, as it does not require tracking the time taken by each request.
- *Fairness:* each client requests may impact the service differently.

P 4: Time Based Rate Limit

Context: API or service provider is able to estimate the time it takes to process a particular type of request or operation.

Problem: How to prevent clients from sending too many highly-resource-consuming concurrent requests within a given short time window?

Solution: Estimate the processing times for different types of requests or operations, and set a time limit that a client is allowed to consume within a specific time frame. e.g. For every five minutes a client can only send a total number of requests that will need one minute of processing time.

Solution details: Fix a time window duration window and restrict the number of concurrent requests that can be made within those windows based on capacity, projected traffic, and the estimated time taken by each of the requests sent during the same window.

¹<https://github.com/express-rate-limit/express-rate-limit>

²<https://github.com/koajs/ratelimit>

In the extreme case, only one request can be processed from each client at a time window. Additional requests sent by the same client are rejected as long as the server is busy with the previous one. If the client request hangs or simply lasts beyond the time window duration, it can be aborted and an error is returned to the client.

The Frappe framework can be adopted to implement the time-based rate limit pattern. The framework implements fixed window rate-limiting based on time consumed by requests. The Rate Limit can be enabled by setting in the configuration file: `site_config.json` the values of the attributes `limit` and `window` in seconds. Where `limit` is the maximum that requests sent during a time window `window`. e.g. In the following configuration example, the sum of the time taken by all the HTTP requests coming from a specific client to 600s within each 3600s time window.

```

{
  "rate_limit": {
    "limit": 600,
    "window": 3600
  }
}
  
```

Consequences:

- + *Scalability:* By limiting the number of requests that can be made within a specific time window, the API can handle a larger number of requests and reduce the load on the backend servers.
- + *Predictable resource usage:* Since each request takes a fixed maximum amount of time, the server can more easily predict and control the maximum resource usage.
- + *Performance:* By limiting the time taken by each request, the server can ensure that requests do not monopolize resources and cause slowdowns for other clients.
- *Usability:* Certain types of requests may require more time to complete and may not be possible under a time-based rate limit.
- *Unpredictability/User Satisfaction:* Clients may be unable to predict the duration of their requests and may be disappointed if their requests are forced to stop when they take longer than the maximum amount of time allowed.
- *Implementation complexity:* Implementing time-based rate limiting can be more complex than other rate-limiting approaches since it requires tracking (on the server) and estimating (on the client) the time taken by each request.

Known uses: Among the Shopify APIs, only the Storefront API accepts requests that take up to 60 seconds per IP address.

P 5: Point Based Rate Limit

Problem: How to efficiently manage resource allocation and prevent API overload when clients access multiple resources with a single request?

Solution: Assign to each request a specific point value based on its complexity and resources required, and restrict the total number of points that can be used within a certain period.

Solution details: GraphQL APIs often employ points-based rate limit. Since in GraphQL, a query can operate on several resources, when deciding the Rate Limit value, the complexity of all the queries

that the API can handle should be taken into account. This differs from the other APIs, where every request invokes an endpoint that targets one specific resource.

Known uses: A widely known example is the GitHub GraphQL API³. The API dynamically computes a rate *limit score* based on query complexity. The limit score of all the queries made in an hour should not exceed 5000 points/token ($token_limit = 5000points/h$):

$$github_query_score = \min(token_limit, \max(1, \frac{query_cost}{complexity_weight}))$$

Where the $token_limit$ is the specific Rate Limit per used token (default value: 5000 points/h).

The $query_cost$ in the case of the GitHub API is computed based on the relative computational cost of resolving each field in the schema, which is, in other words, the number of calls needed to fulfill the query. Note that an individual query cannot exceed 500k nodes. Also, the minimum cost of a query is equal to 1, in the case of queries with depth equal to 1. By default, all fields in the GitHub GraphQL API have the same complexity weight, which is 1.

Clients track their Rate Limit status by querying fields on the `rateLimit` object. They can also compute query scores before determining whether they have sufficient points left to run a query.

Consequences:

- + *Customizability*: The point values can be customized to reflect the relative importance of different API operations or queries or the availability of different server resources.
- + *Fairness*: A points-based system can be more fair and flexible than other types of rate limiting, as it allows clients to make more requests for simpler operations and fewer requests for more complex ones.
- *Implementation complexity*: Implementing a points-based system can be more complex than other types of rate limiting, as it requires tracking the point values of each API operation or query.
- *Cost Estimation*: Service providers should provide their clients with a solution to accurately compute the server's query execution cost to enable them to adapt to Rate Limit restrictions. While a dynamic cost computation can be more accurate, it can induce additional runtime costs. A static approach would not cause additional runtime overhead but may only provide clients with an estimate or bound on the expected costs.

5 RATE LIMIT DOCUMENTATION PATTERNS

Rate Limit documentation patterns aim to provide guidelines for documenting API Rate Limit policies and guidelines making the documentation more accessible and easier to understand for developers. We identified two documentation patterns:

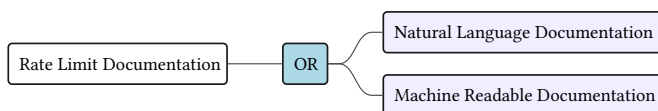


Figure 5: Rate Limit Documentation Patterns

³<https://docs.github.com/en/graphql/overview/resource-limitations>

P 6: Natural Language Documentation

Context: API provider has decided to introduce a Rate Limit, chosen a static or dynamic configuration and selected a suitable metric to define the limit.

Problem: How to communicate to API client developers the Rate Limit configuration?

Forces:

- *Trust*: Setting and meeting expectations with API clients is key to build their trust towards the API and its provider.
- *Developer Understandability*: Properly documented rate limits enable developers to understand and work within the imposed limits, optimizing their API usage and minimizing disruptions.
- *Efficient API Integration*: By providing Rate Limit documentation in natural language, API users with different levels of knowledge can grasp the usage policy and guidelines, facilitating efficient integration and optimization of API usage within the defined limits.
- *Machine Readability*: Client applications should parse the rate limit descriptions to track the correct usage metrics, be aware of their consumption and warn users when approaching the limit.
- *Standardization*: Defining and clearly communicating the chosen rate limit strategy would benefit from commonly agreed terminology, conventions and methods.

Solution: Use natural language to describe the Rate Limit strategy.

Solution details: The information regarding rate limits can be incorporated into the natural language description of the API, such as its web page. Given the potential for misinterpretation in natural language, the presentation of this information must be clear, concise, and easily understandable. This should include a clear outline of the specific details of the rate limit, including the number of permissible requests per unit of time (such as per minute or day), the response that will be returned when the limit is exceeded (such as HTTP 429 "Too Many Requests"), which API elements are limited, and the period after which the limit will reset. It is also important to inform clients of any potential consequences that may arise if the limit is reached.

Consequences:

- + *Trust*: Documenting explicitly the rate limit strategy improves the transparency of the API and increases trust among API client developers and users.
- + *Flexibility*: The ability for providers to use natural language for describing their rate-limiting strategy in detail offers flexibility in tailoring the approach to the needs of their specific API.
- +/- *Developer Understandability*: Natural language presents a high level of human readability and understandability, as it presents the information in a clear and accessible manner. However, it may require more time to grasp the information as the developer needs to go through the entire text unless it is structured in a manner that facilitates quick comprehension.
- *Machine Readability*: NL is not ideal for machine readability, as the information may not be presented in a structured

or standardized format that automated systems can easily process. AI/NLP techniques might be used to extract rate limit information from textual API description.

- **Standardization:** The lack of standardization in the use of natural language to describe rate limits across different API providers can lead to confusion and difficulties for developers, as each provider may use different terminology, conventions, or methods for communicating their Rate Limit strategy, making it challenging to compare and understand the restrictions imposed by different APIs.

Known uses: The default Rate Limit for eBay APIs is presented in a well-structured table, which enhances the ease of information comprehension. The Rate Limit metric used by eBay is consistent across all APIs, measured in terms of the number of calls per hour⁴. This provides clear and straightforward information for developers to understand and adhere to the rate limit. In contrast, GitHub APIs use purely natural language to describe their rate-limiting strategy, making it more challenging for developers to determine the Rate Limit value. The information is dispersed throughout the API documentation and requires a thorough reading of multiple paragraphs to grasp the rate-limiting approach fully.

This is also observed for Meta APIs, where rate-limit strategies are described entirely in natural language, highlighting the diversity of rate-limiting strategies used. The lack of a comprehensive metamodel for describing rate-limiting strategies across different APIs highlights the need for a unified approach to this aspect of API development.

P 7: Machine Readable Documentation

Context: API provider has decided to introduce a Rate Limit, chosen a static or dynamic configuration and selected a suitable metric to define the limit.

Problem: How to provide automated access to Rate Limit information to API clients?

Solution: Use a well-structured, machine-readable language to fully detail the Rate Limit strategy.

Solution details: For static configurations, Rate Limit values can be conveniently included in machine-readable API descriptions. Providing Rate Limit information in machine-readable documentation allows developers to use tools that can read the API rate limit strategy, assuming the metadata is represented following agreed-upon conventions or standards. It also provides developers with a systematic approach to compare the limitations of various APIs that serve similar purposes and plan their integration accordingly. This enables them to make informed decisions and select the API that best meets their specific requirements. This increased level of transparency and comparability can significantly aid developers in their API integration and usage decisions, leading to a more positive experience.

Consequences:

- + **Machine readability:** The ability to automatically parse the Rate Limit strategy out of an API description requires that an agreed-upon metadata representation is followed.

-/+ **Human readability:** It might be difficult to read in case Rate Limit metadata is encoded with complex structured languages such as XML. Still, machine-readable Rate Limit descriptions can also serve to automatically generate human-readable descriptions in natural language.

Known uses: Analyzing 248,566 OpenAPI descriptions revealed that only 4,179 contained keywords related to rate limits. This low number indicates a weak adoption of structured expression formats for statically documenting and communicating information about rate limits. It highlights the tendency of developers to focus primarily on functional aspects of APIs, neglecting to provide detailed information about the limitations imposed on usage.

Even API gateway cloud providers, such as AWS and Azure, offer the ability to import API endpoint details, including resources, methods, responses, and descriptions, as well as the mapping between API operations and backend functions, through the use of OpenAPI Specification (OAS) descriptions [2, 3]. However, they lack the capability to import and export Rate Limit and usage plan configurations in a machine-readable format. These settings can only be manually configured through forms found in the dedicated web-based user interface, limiting the level of automation and programmatic control that can be exercised over these critical aspects of API management.

6 API RATE LIMIT COMMUNICATION

Developers integrating their client applications with APIs adopting a rate limit require accurate and up-to-date information about their API usage level to track and optimize their API usage and avoid Rate Limit violations. Without a communication mechanism to retrieve Rate Limit details, developers may struggle to obtain the necessary information, leading to inefficient integration, excessive costs, and potential disruptions.

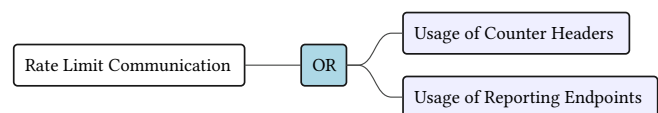


Figure 6: Rate Limit Documentation Patterns

We identified two patterns related the how API providers can communicate the current up-to-date Rate Limit state to their clients: Usage of counter headers, and usage of an endpoint to report Rate Limit state.

P 8: Usage of Counter Headers

Context: Clients invoke an API with static or dynamic rate limit configuration.

Problem: How to provide clients with their usage tracking information together with up-to-date information about dynamic API limits?

Forces:

- **Transparency:** Real-time feedback on Rate Limit consumption allows developers to accurately track their API usage, enabling them to make informed decisions and adjustments to stay within the defined limits.

⁴<https://developer.ebay.com/develop/apis/api-call-limits>

- *Performance Overhead*: Network bandwidth may be limited and expensive, thus retrieving such information with additional calls may not be an option.

Solution: Include Rate Limit information in the response header of every Web API call.

Solution details: The headers such as `X-Rate-Limit-Limit` and `X-Rate-Limit-Remaining` can be used to notify the clients about the dynamic Rate Limit value. The `X-Rate-Limit-Limit` header reports the total allowed number of requests in the current time window, and the `X-Rate-Limit-Remaining` header shows the remaining number of requests that can be made before reaching the limit. Customized headers can also be used to transmit the same or other metrics.

Known uses: Analyzing the response headers schemas included in the OpenAPI descriptions, we detected a total of 316 APIs that dynamically convey information about the API, endpoint, or provider limits through dedicated response headers.

The GitHub REST API embeds the following usage counters in the responses headers of all query operations:

```
x-ratelimit-limit: 60
x-ratelimit-remaining: 56
x-ratelimit-used: 4
x-ratelimit-reset: 1372700873
```

In the case of Shopify, all their REST APIs use a specific header field to report how many requests the client has made over the total number of allowed requests per minute. If the limit is exceeded, a `Retry-After` header is sent with the number of seconds to wait until retrying the query.

```
X-Shopify-Shop-API-Call-Limit: 32/40
```

Consequences:

- + *Performance*: Clients can receive immediate feedback on their usage of the API and can adjust their requests accordingly, reducing the number of unnecessary requests and improving overall API performance.
- + *Reliability*: By receiving up-to-date information on their usage level, clients can avoid surprises and detect whether they are still compliant with the rate limit without getting completely blocked from accessing the API and can plan their usage accordingly, ensuring reliable access to API resources.
- *Interoperability*: Client developers may find it difficult to understand the meaning of the Rate Limit headers unless they are properly documented. Additionally, some clients may not anticipate how to deal with Rate Limit headers in response payloads, resulting in unexpected errors.
- *Maintainability*: Rate Limit headers can add complexity to API documentation and implementation, requiring additional maintenance effort to ensure accurate metering and consistent usage.

Related Patterns:

- Rate Limit documentation patterns: Complement this pattern by providing comprehensive documentation that covers the

meaning of all the Rate Limit-related headers appended to the response messages.

P 9: Usage of Reporting Endpoints

Context: Clients are not about to invoke an API with a dynamic rate limiting configuration, but nevertheless they would want to discover if their previous usage lies within the limits. Service providers offering access with dynamic rate limits need to inform clients about changes.

Problem: How can API providers ensure that client developers have easy access to accurate and up-to-date Rate Limit details?

Solution: Add an endpoint that the clients can use to explicitly retrieve *API Rate Limit* settings and API usage counters.

Solution details: Dynamic Rate Limit values can be retrieved through one of the API's endpoints, such as a dedicated endpoint for checking the current Rate Limit status. This endpoint can return information such as the current rate limit value, the time window for the rate limit, and the remaining number of requests. This information can be returned in the response body in a structured format, such as JSON or XML, and can be accessed by the client through a GET request. Notifying the Rate Limit value provides a programmatic way for clients to check the Rate Limit status and can be helpful for automation and monitoring. This way, client applications can check the Rate Limit status before making requests to the API and take appropriate actions like waiting, caching, or prioritizing requests.

Note that these endpoints for retrieving API Rate Limits can be rate-limited themselves.

Consequences:

- + *Convenience*: Developers do not have to invoke a regular API endpoint with the goal of retrieving rate limit settings and usage reporting information, but can use dedicated, side-effects free endpoints.
- + *Transparency*: By providing transparent and up-to-date Rate Limit information, API providers can enhance the overall developer experience and foster a collaborative relationship with developers of client applications.
- + *Performance*: As regular messages exchanged with the API do not include additional usage counter headers, they are smaller and bandwidth consumption is reduced.
- *Security*: If the Rate Limit reporting endpoint is not secured properly, it can become a target for abuse, leading to security vulnerabilities.

Known uses:

- GitHub API: The GitHub API allows developers to interact with GitHub's platform and services. It has a Rate Limit endpoint at https://api.github.com/rate_limit that returns the current Rate Limit status for the authenticated user or the IP address.
- Spotify API: The Spotify API allows developers to access and control Spotify's music streaming service. It has a Rate Limit endpoint at <https://api.spotify.com/v1/rate-limit-status> that returns the current Rate Limit status for the authenticated user.

- **OpenWeather API:** The OpenWeather API provides weather data and forecasts for various locations. It has a Rate Limit endpoint at https://api.openweathermap.org/data/3.0/limit-status?appid=CLIENT_API_KEY that returns the current Rate Limit status for the specified app ID.

7 RATE LIMIT GRANULARITY

When establishing a Rate Limit for an API, various levels of granularity can be utilized to regulate the frequency at which requests are made. The degree of granularity selected establishes the precision of the Rate Limit application and aids in ensuring that the API is utilized efficiently and responsibly. This section categorizes the patterns related to granularity into client-level or resource-level patterns. The distinction is based on whether the Rate Limit restriction is enforced to restrict a specific client – identified by their IP address, by user authentication, or API keys – or a specific API resource – specific endpoints or operations, or entire APIs, or all APIs offered by a given provider.

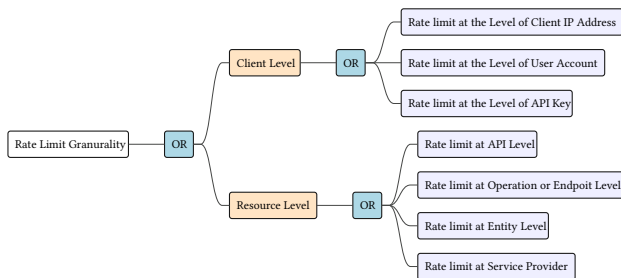


Figure 7: Rate Limit Granularity Patterns

7.1 Client Level Granularity Patterns

P 10: Rate Limit at the Level of User Account

Context: The source of client requests can be distinguished by authenticating the sender user account.

Problem: How to control the usage of the API by individual users, especially if some users are making significantly more requests than others?

Solution: Set customized quotas for each user or group of users.

This level of API Rate Limiting is appropriate when an application requires different rate limits for different users. It can be used when user accounts are attached to different usage plans.

Consequences:

- + *Flexibility:* When a Rate Limit is defined on the user accounts level, the API can also provide customized usage quotas or limits based on each user’s specific needs or usage patterns.
- *User authentication:* Users may pool authentication credentials or register additional accounts to work around the rate limit enforced on a per-user basis.

P 11: Rate Limit at the Level of Client IP Address

Context: API provider or service needs to control and restrict the rate at which incoming requests are made from individual IP addresses

Problem: How to prevent abusive usage from single IP addresses?

Solution: Limit the number of requests that can be made from a single IP address in a given period.

Solution details: This pattern can be implemented for instance using the basic configuration of NGINX. Inside the Nginx configuration, define a `limit zone`, which is typically defined in the HTTP block. A limit zone specifies the key for rate limiting, the maximum burst, and the rate limit. e.g.:

```
http {
    limit_req_zone $binary_remote_addr zone=rate_limit
}
```

- `$binary_remote_addr` is the key for rate limiting based on the client’s IP address.
- `zone=rate_limit_zone` is the name of the zone. 10m: The memory allocated for the zone.
- `rate=10r/s` is the rate limit value, in this example, allows 10 requests per second.

Consequences:

- + *Authentication-free:* This strategy does not require any client or user authentication.
- *Unfairness:* Legitimate distinct users sharing the same IP address might be affected by unintended limitations.
- *Effectiveness:* Adopting a rate-limiting strategy based only on the IP addresses is not always effective because an abusive user can still invoke the API from multiple IP addresses.

P 12: Rate Limit at the Level of API Key

Context: Every client can obtain a specific unique key using a token generator offered by the API provider.

Problem: How to control the usage of the API by a specific client?

Solution: Identify clients based on their unique API Keys.

Solution details: Client developers need to register their client applications with the API provider to obtain an API key. Providers need to generate unique keys and store them until they expire. The API key should be sent with every request so that the provider can use it to identify clients, track their usage of the API and enforce the corresponding rate limits.

Consequences:

- + *Precision:* providers can distinguish traffic originating from different client applications even if these share the same source IP address.
- *Precision:* Providers are unable to distinguish requests from different users of the same client application.
- *Security:* API keys may be leaked into access logs or code repositories and are susceptible to theft or unauthorized use if not properly secured.

Known uses:

This is the most common level of API Rate Limiting, where all requests from an application identified by its API key are subject to the same rate limit. This level is appropriate when an application does not require user-specific rate limits or when it is difficult

to identify individual users (such as with anonymous or public applications).

In the case of Github API, this rate-limiting approach is combined with rate-limiting based on the IP Address.

7.2 Resource Level Granularity Patterns

P 13: Rate Limit at the Level of Service Provider

Context: A provider manages multiple services consumed by clients. One client needs to combine multiple services offered by the same provider.

Problem: As a provider, how can all my services adhere to a consistent set of usage guidelines?

Solution: All the APIs of a given provider use the same rate-limiting configuration.

Consequences:

- + *Simplicity:* Simplified billing or uniform pricing models encourage users to try and use one or more services offered by the same provider.
- + *Usability:* Consistent user experience across all APIs and services of the provider makes it easier to transfer knowledge learned by using one API to other APIs offered by the same provider.
- *Fairness:* Users might need to access some services more frequently than others.

Known uses:

NASA Open APIs. According to the NASA Open APIs documentation on rate limiting, all NASA APIs have the same default Rate Limit of 1000 requests per hour per IP address⁵. However, they recommend obtaining a developer API key to developers who will be intensively using the APIs to support a mobile application or will be making more than a few dozen requests per hour. This key will allow for higher rate limits that are specific to the API key. Therefore, when using a developer API key, the rate limits will be different from the default Rate Limit for all APIs.

P 14: Rate Limit at the API Level

Context: Clients invoke all API features with uniform probability, so there are no predictable hotspots. There is a uniform costs of providing all API features.

Problem: How to control the usage of an entire API, independently of which features are being used?

Solution: Track usage of API features globally and set limits on the entire API, giving the same weight to each request, no matter which API feature it invokes.

Consequences:

- + *Maintainability:* Using the same Rate Limit simplifies the management and configuration of rate-limiting rules, making it easier to maintain and update the API's rate-limiting system as a whole.
- + *API monetization strategy:* Having the same Rate Limit on all the API operations makes it easier to set a pricing plan that is not confusing for clients and easier to track.

- *Scalability:* Different endpoints or functionalities within an API may have varying resource requirements. Applying a uniform Rate Limit may hinder the ability to scale certain critical endpoints independently, especially if they become hotspots, potentially leading to performance bottlenecks and inefficient resource allocation.

Known uses: Both the Search API and Files API of Stripe⁶ allow up to 20 operations per second. It should be noted that this Rate Limit applies to both reading and writing operations without distinguishing between the two.

P 15: Rate Limit at the Endpoint or Operation Level

Context: Some API endpoints may be more susceptible to abuse than others. For example, certain endpoints are particularly resource-intensive, such as those that involve complex calculations or database queries involving large amounts of data.

Problem: How to control the usage of specific features of the API, especially if some features are being used significantly more than others?

Solution: Track usage of specific API endpoints or operations and set limits according to their specific costs.

Consequences:

- + *Precision:* Endpoint-level rate limiting can be used to ensure that critical endpoints (or operations) are not overloaded with requests, which can negatively impact the performance and availability of the entire API.
- *Understandability:* Clients need to be clearly informed about which limits are applied to which endpoints (or operations).

Known uses: Google Analytics Reporting and Configuration web APIs have different default limits depending on whether the endpoint is a writing or reading endpoint. Google also allows users to request additional quotas per each project, for each of the read and the write requests separately⁷.

P 16: Rate Limit at Entity Level

Context: Some specific entities managed via the API may be more susceptible to abuse than others.

Problem: How to control the usage of one or more operations accessing a specific entity?

Solution: Track the endpoints accessing the same entity together and set limits according to their specific costs.

Consequences:

- + *Efficiency:* Entity-based rate limiting ensures that a specific entity is not overwhelmed by requests and helps to optimize its usage.
- *Scalability:* If the API is experiencing high traffic, it may be challenging to scale the Rate Limit effectively, as different entities may have varying usage patterns and requirements.

Known uses:

- In Ebay's Post-Order API, the endpoints accessing each entity have their own Rate Limit which is set to 5000 API calls per

⁵<https://api.nasa.gov/>

⁶<https://stripe.com/docs/rate-limits>

⁷<https://developers.google.com/analytics/>

day separately for each of Cancellation, Case Management, Inquiry, and Return entities⁸.

- Ebay’s Fulfillment API combines both limits per operation and per entity. For instance, the `getPaymentDispute` and `getPaymentDisputeSummaries` methods have a Rate Limit that is separately set to 250,000 API calls per day, while all the methods accessing the Order entity have a shared Rate Limit set to 100,000 API calls per day⁹.

8 PROVIDER REACTION TO RATE LIMIT EXCEEDING

In some cases, even after being temporarily blocked due to exceeding the API Rate Limit, certain clients may persist in attempting to make requests above the set limit. This can cause strain on the API and negatively impact its performance. It may be necessary to implement additional measures to prevent such clients from bypassing the rate limit, such as IP address blacklisting or more sophisticated anti-bot or denial of service prevention mechanisms [18]. We identified a set of Rate Limit adoption patterns related to the providers’ reaction to some clients’ abusive behaviors. We classified them into two categories, depending on whether the provider’s goal is to prevent abusive clients from consuming the API or to mitigate their behavior (Figure 8).

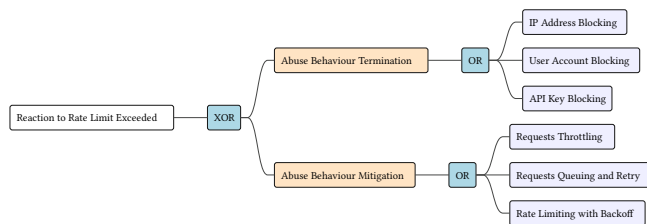


Figure 8: Providers Reaction to Rate Limit Exceeding Patterns

8.1 Abusive Behavior Termination

Clients may exceed their usage limit. As a consequence, further requests should be blocked. The specific pattern to terminate abusive behavior that can be used depends on how clients are identified and whether they are expected to be automated or not.

Client applications driven by human users exceeding the Rate Limit could imply that the application is being scraped: automatically accessed with request traffic growing beyond what human users can be expected to generate. In this case, a CAPTCHA challenge [44] could be displayed to users if they exceed the rate limit. This has the benefit of slowing down the user who has to solve the challenge. In case the bot fails to complete the challenge, the client IP address, its user account or API Key can be blocked.

In case where the API adopts a strategy that permanently and definitively terminates access from all clients whose abusive behaviors have been detected, it is also necessary to explicitly warn these clients that they will be blocked if they exceed the API rate limit. This information can also be statically part of *Rate Limit Documentation* artifact. Once the block is in place, such clients will no

longer be able to access the API, nor its usage reporting endpoints (see *Rate Limit Communication patterns*) and thus should also be notified through a separate communication channel that a block has been put in place.

P 17: IP Address Blocking

Context: Clients are identified by their IP Address.

Problem: How to effectively terminate abusive behaviors from clients identified by their IP address?

Solution: If a client exceeds a predefined limit on the number of requests they can make, further requests originating from that client IP address should be blocked.

Consequences:

- + *Simplicity:* Simple to implement with network appliances such as firewalls or packet filters.
- *Unfairness:* Legitimate users may be mistakenly blocked if the IP address is flagged for abusive behavior.

Known uses: When exceeding the limits, GitHub blocks IP addresses of non-authenticated clients. The sent response has a 403 code with a `x-xss-protection` header that informs the clients that they are being blocked.

```

'x-ratelimit-limit': '60',
'x-ratelimit-remaining': '0',
'x-ratelimit-reset': '1689004335',
'x-ratelimit-resource': 'core',
'x-ratelimit-used': '60',
'x-xss-protection': '1; mode=block'

```

The response also includes a message to inform clients that the Rate Limit value is higher for authenticated requests: “RequestError [HttpError]: API Rate Limit exceeded for <IP-ADDRESS>. (But here’s the good news: Authenticated requests get a higher rate limit. Check out the documentation for more details.)”

Related Patterns: This pattern is often used in conjunction with the “Rate limit at the level of Client IP address” pattern.

P 18: User Account Blocking

Context: Client applications require user authentication, making it possible to identify who is using them.

Problem: How can an application or service precisely terminate behaviors of specific clients associated with user accounts?

Solution: Prevent requests from any client that are made on behalf of a certain user account if they surpass the rate limit.

Consequences:

- + *Precision:* only requests from specific users are blocked.
- *Account impersonation:* Abusive users can still overload the system by using multiple accounts, effectively impersonating different users to evade the rate-limiting measures.

Related Patterns: This pattern is often used in conjunction with the “Rate limit at the level of the User Account” pattern.

P 19: API Key Revocation

⁸<https://developer.ebay.com/develop/apis>

⁹<https://developer.ebay.com/develop/apis/api-call-limits>

Context: API keys are used for authentication and authorization of clients accessing the API.

Problem: How effectively and fairly terminate behaviors of clients authenticated using an API key?

Solution: Block requests from a particular API Key if they exceed the rate limit.

Consequences:

- + *Precision:* only requests from malicious client developers are blocked.
- *Unfairness:* Legitimate users may be mistakenly blocked if the API Key of their client application is flagged for abusive behavior caused by other users.

Related Patterns: This pattern is often used in conjunction with the “Rate limit at the level of the API key” pattern.

8.2 Abusive Behavior Mitigation

P 20: Request Throttling

Context: An API is exposed to various clients, and there is a need to regulate the rate of incoming requests to maintain quality of service, prevent abuse, and allocate resources efficiently.

Problem: How to control the rate of incoming requests to a web API to prevent abuse?

Solution: Instead of completely blocking requests from abusive clients, the solution is to throttle the bandwidth allocated to transmit requests or responses to specific clients. Throttling limits the rate at which data can flow between the client and the server, effectively slowing down the client’s access.

Consequences:

- + *Fairness:* Prevent abusive behavior while still allowing legitimate clients to access the API.
- *Efficiency:* Not effective against abusive users who might distribute requests across multiple clients to bypass the throttling rate.

P 21: Request Queuing

Context: Client requests cannot be discarded as this will affect the compliance of the system with its intended functional requirements.

Problem: How to handle requests when the Rate Limit is exceeded, ensuring that clients do not lose their requests and maintaining fairness in processing?

Solution: When the Rate Limit is exceeded, queue the requests and process them in order at a later time, when the Rate Limit is no longer exceeded.

Consequences:

- + *Reliability:* Ensures that clients do not lose their requests when they are rate-limited, and they do not need to resend them when they are allowed to.
- *Timeliness:* The queued requests’ responses might differ from those that would have been sent when the requests were made.

- *Capacity:* The queued requests take up space on the server, which may run out of storage capacity in case of abusive clients.

Known uses:

- Many content delivery networks (CDNs) use request queuing to manage incoming traffic and ensure fairness.
- Cloud-based services and APIs often employ request queuing to handle rate-limited requests without losing them.
- E-Commerce platforms may use request queueing during peak traffic to manage order processing.

P 22: Rate Limit with a Rest Time

Context: Clients may temporarily exceed the limit and should not be permanently banned

Problem: How to prevent excessive requests from specific clients without definitively terminating their access to the service?

Solution: Block requests from a specific client until a predefined time has passed since the Rate Limit was exceeded, providing clients with information about when they can resume sending requests.

Consequences:

- + *Transparency and usability:* Temporarily blocked clients are aware of when they can start sending their requests again.
- *Bursting:* Abusive users can still cause a spike in traffic if they schedule all their requests to occur immediately after the Rate Limit reset time.

9 SERVER-SIDE RATE LIMIT IMPLEMENTATION

We have identified distinct patterns describing various possible Server-Side rate limit implementations, which we classified depending on the rate limiter component, its positioning, and scope. Solutions depend on the type of system architecture in which the API is located. By identifying the relevant pattern, developers can effectively enforce rate limiting at the level of the system’s interfaces.

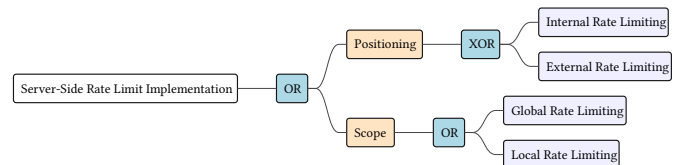


Figure 9: Rate Limit Server-Side Implementation Patterns

9.1 Rate Limiter Positioning

P 23: Internal Rate Limiter

Context: A service provider has access to their own infrastructure and systems, allowing them to exert control over the rate of incoming requests.

Problem: How to implement a controllable rate limiter that does not rely on external services?

Solution: Rate Limit is completely implemented as part of the server-side code. The rate limiter intercepts and filters clients' requests as they reach the backend service implementing the API.

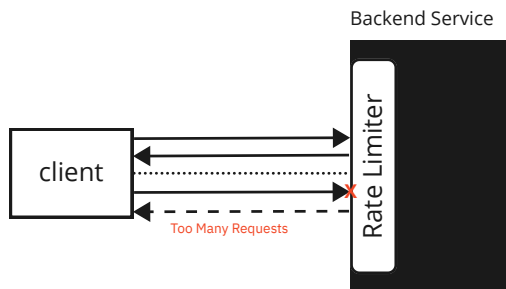


Figure 10: Internal Rate Limiter Pattern

Solution details: In the case of monolithic service architectures [22], a rate limiter can be implemented as part of the backend service.

In the case of microservice architectures [37], the Rate Limit policy can also be implemented internally within the back-end architecture (Figure 10). The exact placement of the rate limiter also depends on the chosen infrastructure to implement the microservices architecture:

- *Service Mesh based architecture.* A service mesh is a dedicated infrastructure layer that facilitates service-to-service communication within a microservice architecture [26]. It is typically implemented using a *sidecar* proxy (Figure 11) that is deployed alongside each service in the application and communicates with other *sidecar* proxies deployed alongside other services in the application to manage the flow of traffic between services [21].

In a service mesh architecture, the Rate Limit policy is placed in the control plane to ensure centralized and consistent enforcement of the Rate Limit. It also provides dynamic configuration capabilities, enabling easy management, updates, and adjustments without requiring changes to individual services. The goal is to employ the control plane to also gather Rate Limit and traffic-related data and have a centralized view of the impact of adopting Rate Limit on the architecture.

In this kind of architecture, different decisions can be adopted to implement rate limits internally within the back-end:

- The sidecar proxy can be configured to enforce rate limits on the inter-services communications based on different criteria. This solution helps to lift the Rate Limit control from the application to the networking layer (Figure 12); however, it still does not help to reduce the complexity of managing traffic.
- One of the services in a service mesh architecture can be used as a rate limiter service (Figure 13). This service can then be configured to enforce rate limits on requests to other microservices within the service mesh. This way, the Rate Limit strategy can be centrally managed and enforced for all the microservices within the mesh.

- *API Gateway based architecture.* In this architecture, an API gateway [37] sits between the client and the microservices. The API gateway acts as a reverse proxy, routing requests to the appropriate microservice. It can be used to host a rate limiter component, which rate limits incoming requests to each of the microservices based on a defined strategy [1, 11].

Consequences:

- + *Customization:* Since it is part of the provider's code, the rate limiter component can be tailored to match the specific rate limits required by the application. It can also be customized to handle different types of request differently.
- + *Integration:* This can be integrated more seamlessly with the application codebase, making it easier to update as the application evolves.
- *Increased complexity:* It requires additional development effort and maintenance overhead for the application and can add complexity to the application codebase.

Known uses:

- **API Gateway:** An API gateway can enforce a Rate Limit across multiple APIs or endpoints. An API gateway between the client and the API can intercept and modify incoming requests. Many API gateways, such as *Kong* or *Tyk*¹⁰, have built-in rate-limiting functionality.
- **Envoy¹¹ Proxy:** The proxy is used by well-known service mesh technologies like Istio¹² and Kuma¹³. It can be used as a *Front Proxy* or an *Edge Proxy* for more granular rate-limiting.

P 24: External Rate Limiter

Context: The solution to rate limiting is readily available from an external system.

Problem: How to accelerate and ease the implementation of a rate-limiting solution?

Solution: Rate Limit is implemented using third-party services or library.

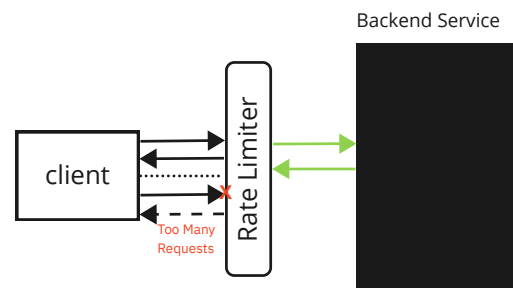


Figure 14: External Rate Limiter Pattern

¹⁰<https://tyk.io/deployment-api-gateway/>

¹¹<https://www.envoyproxy.io>

¹²<https://istio.io>

¹³<https://kuma.io>

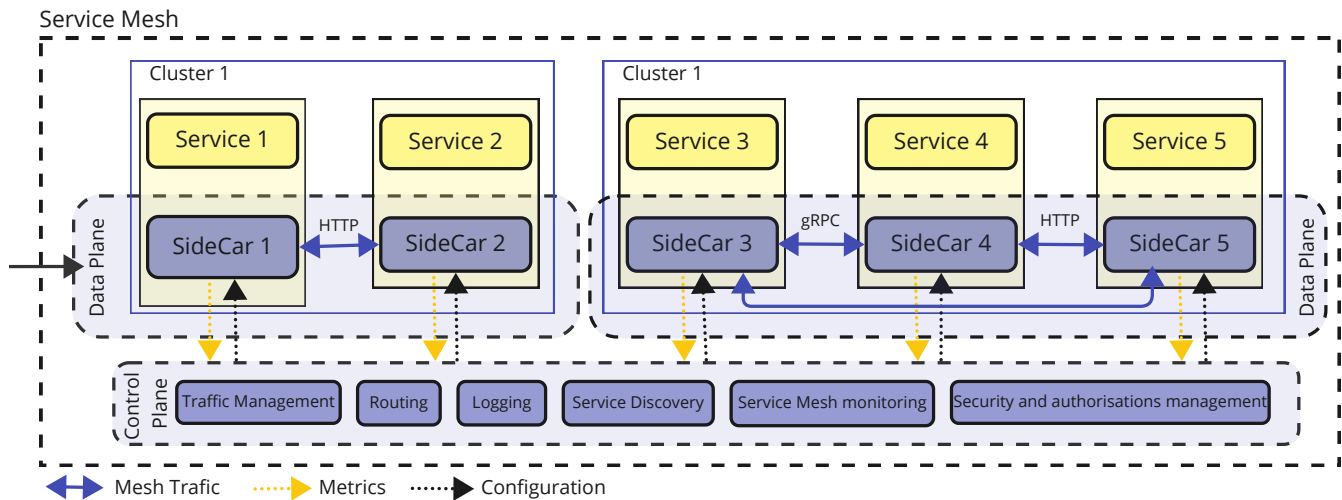


Figure 11: Service mesh microservices architecture

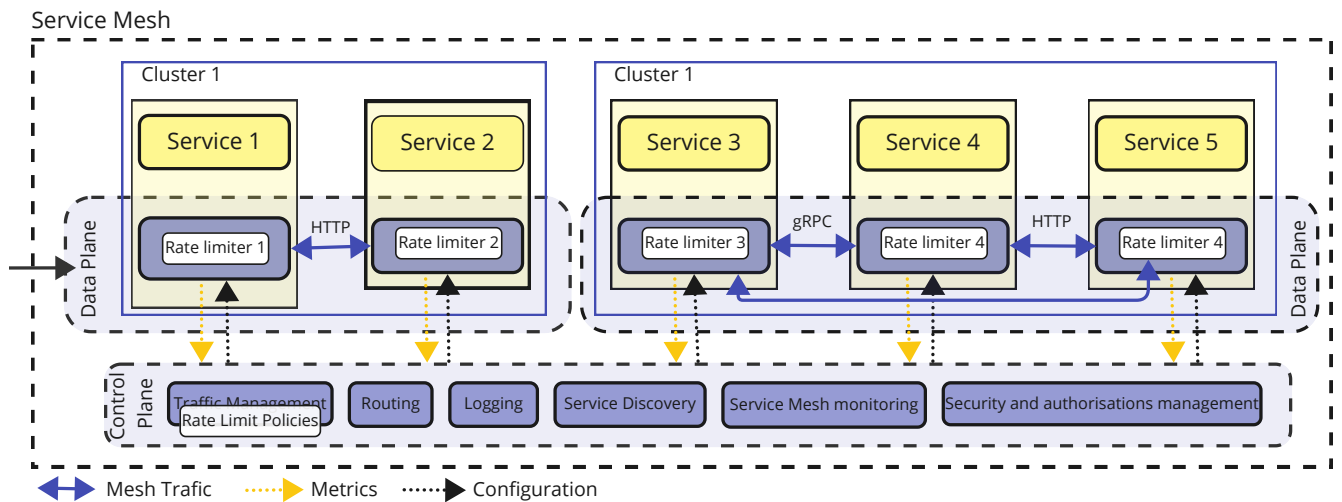


Figure 12: Internal rate limiter in each sidecar proxy

Solution details: Select a suitable rate-limiting service or tool, configure rate-limiting rules, integrate with the API, test, monitor, and fine-tune as needed.

Consequences:

- + *Speed of implementation:* Reusing an existing rate limiting component can save the time compared to implementing the rate limiter component from scratch.
- + *Compliance:* It can help ensure compliance with regulations and industry standards related to rate limiting, such as the Payment Card Industry Data Security Standard [32].
- *Security:* It may introduce new security risks, such as the exposure of sensitive data to external service providers or the risk of service providers being compromised.
- *Performance:* It may introduce additional latency and overhead in request processing.

- *Reliability:* It makes the system dependent on external factors such as network connectivity and availability of the rate limiter.

Known uses:

- **Cloud provider's rate-limiting service:** Many cloud providers, such as AWS, Google Cloud, and Azure, offer rate-limiting services that can be integrated with an application or API. These services typically provide flexible rate-limiting rules and can scale to handle high traffic volumes.
- **Third-party rate-limiting services:** There exist services such as Cloudflare, Akamai and Fastly, that can be integrated with an application or API. These services often offer advanced features such as DDoS protection and real-time monitoring.

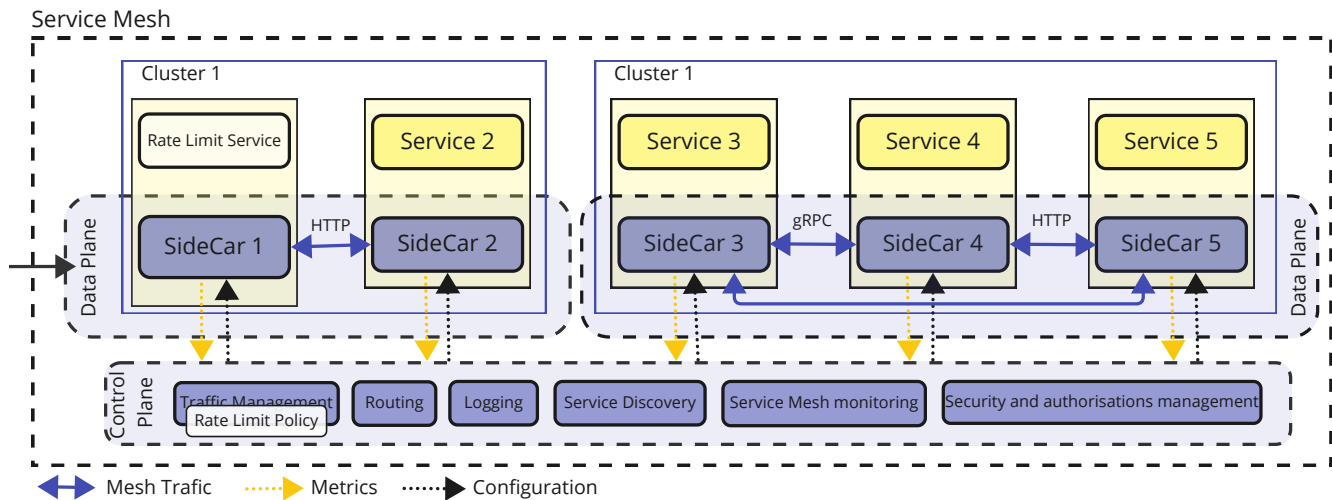


Figure 13: Internal rate limiter as a microservice in the service mesh

- **Open-source rate-limiting components:** Many open-source rate-limiting components are available for various programming languages. For example, Redis, Guava, and Bucket4j can be integrated with an application or API and can provide basic rate-limiting functionality.

Related Patterns: A hybrid approach can increase the reliability of the system by combining an *External Rate Limiter* and an *Internal Rate Limiter*. The goal is to append additional functionalities to the internal rate limiter by selecting a suitable external rate-limiting service or tool that provides the additional rate-limiting functionalities to integrate with the internal rate limiter codebase. This approach can provide a backup mechanism in case the external rate limiter fails or experiences performance issues and reduce single points of failure in the rate-limiting process. However, developers and maintainers can face challenging integration problems that occur between the *Internal Rate Limiter* and the *External Rate Limiter*.

9.2 Rate Limiter Scope

P 25: Global Rate Limiter

Context: The system architecture is decomposed into multiple services.

Problem: How to enforce a shared and uniform Rate Limit value for all services of a system?

Solution: Rate Limit is implemented using a *Front Proxy* to handle the overall amount of incoming traffic to a system.

Solution details: This approach entails identifying the operational thresholds of the system that necessitate the implementation of a global rate limiter. The Rate Limit *Front Proxy* can be enforced at either the application layer by restricting the number of requests or at the network layer by configuring the network equipment to regulate the flow of traffic passing through.

Consequences:

- + **Consistency:** A uniform rate limit is enforced for all services of the architecture.
- **Flexibility:** It can be difficult to adjust or fine-tune the rate limit to specific resource consumption rates.

Known uses:

- Envoy's service proxy¹⁴ is a well-known solution often used to implement global rate limiting in Service Mesh architectures. It uses a gRPC rate limiting service to provide rate limiting for the entire mesh. If the Rate Limit is exceeded, Envoy can respond with a 429 status code or can be configured to throttle the requests until the Rate Limit is reset.
- Nginx ingress also allows configuring global rate limits¹⁵. This can be done by configuring the `limit_req_zone` directive in the Nginx configuration file¹⁶. This directive sets up a shared memory zone that can be used to track request rates, with a maximum number of requests per second, and configure how to handle requests that exceed the limit.
- Redis¹⁷ is an in-memory data structure store that can be used to implement global rate limiting. It is supported by several API Gateway pattern implementations (e.g., krakend¹⁸, Spring Cloud Gateway¹⁹, Kong²⁰) to offer a global Rate Limit solution.

P 26: Local Rate Limiter

Context: The system architecture is decomposed into multiple services.

Problem: How to define and enforce different Rate Limit values for each service?

¹⁴<https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit>

¹⁵http://nginx.org/en/docs/http/nginx_http_limit_req_module.html#limit_req_zone

¹⁶<https://www.nginx.com/blog/rate-limiting-nginx/>

¹⁷<https://redis.com/>

¹⁸<https://www.krakend.io/>

¹⁹<https://cloud.spring.io/spring-cloud-gateway>

²⁰<https://konghq.com/>

Solution: Rate Limit is implemented as a part of the service mesh using *Edge Proxies* ensuring each service can independently filter its incoming requests.

Consequences:

- + *Flexibility:* Local rate limiting gives more control over the rate-limiting logic, allowing for more fine-grained control and customization of the rate-limiting rules for each service.
- *Complexity:* Implementing local rate limiters requires additional code and configuration, which can increase the system's complexity and potentially introduce bugs or performance issues.

Known uses:

- Envoy also supports local non-distributed rate limits where the Rate Limit can be configured to be applied on specific endpoints.
- GitHub API uses different rate limits for the different services it allows access to. For instance, the authentication services have a different Rate Limit than the search on, where the authentication requests are prioritized over unauthenticated requests when enforcing rate limits.

Related Patterns: Combining both *Global* and *Local* rate limiters can provide better control and flexibility over resource utilization in a system. Global rate limiters can ensure that the overall load on the system is kept within acceptable levels, preventing system overload and potential downtime. Meanwhile, local rate limiters can provide more fine-grained control over specific services, allowing for more efficient use of resources and improved performance.

10 PATTERN DEFINITION APPROACH

This section describes our approach to extracting Rate Limit adoption patterns. First, we provide details about the static patterns. Secondly, we give an overview of the experimentation used to demonstrate these patterns' impact on performance and reliability.

10.1 Static Perspective

10.1.1 Representing Rate Limit pattern in OpenAPI. In this work, we have extended our analytics tools used in [24, 25, 39–41] to observe the adoption of Rate Limit patterns in real-world APIs by systematically detecting the ones using a Rate Limit through an analysis of 168402 static API descriptions written in OpenAPI, a widely adopted standard language used by developers to specify various information about their APIs, including request and response payloads, available endpoints, and acceptable media types.

The current version of the OpenAPI language specification does not include predefined constructs to describe *API Rate Limit* values, even though OpenAPI descriptions can still include details such as the maximum number of requests an API can handle per unit of time and the time interval in which these requests can be made. There are various methods to include Rate Limit information in the OpenAPI documentation. Still, the easiest ones to detect are when using the *x-** extension mechanism, which includes extensions such as *x-rate-limit*, as shown in the example in Listing 1.

```
paths:
  /items:
    get:
```

```
description: Returns a list of items
responses:
  200:
    description: Successful response
    x-rate-limit:
      limit: 1000
      interval: hour
```

Listing 1: Example of an OpenAPI extension to document Rate Limit enforced on a specific endpoint

Note that the keys attached to the *x-* prefix are not previously known. Thus we defined various detectors based on our observations of samples of API descriptions containing responses featuring the 429 (Too Many Requests) HTTP status code and keywords matching the regular expression:

```
/rate limit|rateLimit|rate-limit|ratelimiting|throttling/gi
```

In OpenAPI, the response header section conveys various information to the client after each request, including Rate Limit information. Although the Rate Limit values are not typically described statically in the headers schema, the presence of specific headers can indicate the existence of Rate Limit constraints for specific endpoints. For example, the header may include fields such as *X-Rate-Limit-Limit* to describe the maximum number of requests allowed per unit of time and the time interval in which these requests can be made, and *X-Rate-Limit-Remaining* to indicate the number of remaining possible requests to make. Thus, analyzing the header section of the OpenAPI description can provide valuable insights into the API's Rate Limit practices and help identify which endpoints are subject to rate-limiting constraints, e.g.: Listing 2.

```
responses:
  200:
    description: OK
    headers:
      X-Rate-Limit-Limit:
        description: The maximum number of
          requests per minute
        type: integer
      X-Rate-Limit-Remaining:
        description: The number of remaining
          requests in the current minute
        type: integer
  429:
    description: Too Many Requests
    content:
      application/json:
        schema:
          type: object
          properties:
            message:
              type: string
            retryAfter:
              type: integer
```

Listing 2: Communicating Rate Limit information through headers in OpenAPI

The OpenAPI specification allows developers to attach descriptive information to each component of their API through the use

of description fields. These fields are intended to be written in natural language and provide valuable information about the API, including its rate-limiting strategy. However, the lack of formatting conventions for writing these descriptions poses a challenge for systematic analysis. The information within these description fields can be difficult to extract and analyze, as it often lacks structure and consistency.

Rate Limit can also be used as a security scheme in OpenAPI by defining a security definition in the securityDefinitions section in the case of Swagger 2.x, and in the components/securitySchemes in the case of the OpenAPI 3.x specification, and then including a reference to the security definition in the security section of an operation.

When Rate Limit is used as a security scheme in Swagger 2.x as shown in Listing 3

```
paths:
  /items:
    get:
      description: Returns a list of items
      responses:
        200:
          description: Successful response

securityDefinitions:
  rateLimit:
    type: apiKey
    in: header
    name: X-Rate-Limit
    description: Maximum number of requests
    allowed in a given time frame
```

Listing 3: Defining Rate Limit security mechanism in Swagger 2.x

When Rate Limit is used as a security scheme in OpenAPI 3.x, as shown in Listing 4.

```
paths:
  /items:
    get:
      description: Returns a list of items
      responses:
        200:
          description: Successful response

components:
  securitySchemes:
    RateLimit:
      type: apiKey
      name: X-Rate-Limit
      in: header
```

Listing 4: Defining Rate Limit security mechanism in OpenAPI 3.x

Based on these Rate Limit information locations in OpenAPI descriptions, we designed detectors to analyze the use of this pattern from various perspectives. These detectors were used to search a

large collection of APIs to identify instances of the Rate Limit pattern.

Once the APIs that employed rate limiting were identified, we conducted a comprehensive analysis of their specifications to understand the strategies used by developers. This included evaluating the specific parameters and configurations for rate limiting, examining the response codes and messages returned, and analyzing the methods and paths subject to rate limiting.

10.1.2 API Case Studies. In addition to the machine-readable documentation analysis in this study, we performed a manual analysis of well-known APIs providers, such as: eBay (36), Shopify (4), New York Times (10), GitHub (2), LinkedIn (1), Twilio (1), Stripe (1), Trello (1), Flickr (1).

The goal behind analyzing both APIs from the same provider and distinct providers is to see how the rate-limiting strategy is defined across different providers or within the same provider's APIs. The APIs we selected belong to different domains.

10.2 Runtime perspective: Experimentation Overview

In addition to these API studies based on static analysis, we have studied metrics and indicators based on runtime monitoring of APIs. We mainly investigated API patterns [49] that impact properties observable at runtime. One study [30] focuses on the impact of the *API Rate Limit* pattern on the reliability properties of API clients through an analytical model that considers specific workload configurations and rate limits and predicts success and failure rates. We used the observability and monitoring tools, Grafana²¹ and Prometheus²², which are already integrated with Istio to calculate those success and failure rates. In another study [29], we studied the performance impact of the API Request Bundling pattern by using a regression model and multivariate regression analysis on a microservice-based open-source business application with realistic workload scenarios. The regression model predicts the total round trip time of a request based on server-side parameters like the type of the method and the number of calls, using and not using *Request Bundle*. In those studies and others [10], we have experimented with different perspectives when implementing Rate Limit and related patterns. As a result, we derived different variations of the use of those patterns with regard to their positioning and scope.

In the Rate Limit empirical study, we wanted to evaluate its impact on the reliability of microservice-based applications from an *API Client* perspective. For that purpose, we developed an analytical model based on client workload parameters to predict the success and failure rates. We developed workload benchmark scenarios based on the typical interactions extracted in a previous study [34]. We set up the experiment simulating 20 different configurations in two environments: private cloud and Google cloud. We repeated the experiment more than 50 times to validate a proposed analytical model that measures the impact of Rate Limit on the reliability of APIs. Many of the patterns extracted in the previous section were used to evaluate the impact of Rate Limit on the performance and reliability of such an infrastructure by building up a robust

²¹<https://grafana.com>

²²<https://prometheus.io>

prediction model [30]. We have also defined new runtime adoption patterns described in Section 9.

11 RELATED WORK

11.1 Web API Patterns

Many studies have provided extensive descriptions of API patterns [35, 43, 46, 48] defined based on different approaches and targeting several Web API characteristics including API structures, versioning practices, life cycle, and evolution.

Web API Structural Patterns. In [40], Serbout et al. followed a specification-based mining approach to find recurring primitive structures within real-world web APIs. They extracted recurring structural API tree fragments from an extensive collection of OpenAPI specifications and represented each using a defined tree visualization. From a population of thousands of fragments, they selected those that frequently occur, have a relatively small size, and are centered around resource collection. The authors presented a selection of variants for each primitive, which can be composed to build larger API structures.

Web API Evolution Patterns. Researchers have utilized diverse approaches to explore and identify change patterns in Web APIs. In one study [19], a use-case-based method was applied where code analysis and usage logs of two consecutive versions of DHIS2 API were manually examined to understand the effects of changes on clients. Another work [20] employed process mining techniques on API usage logs of the DHIS2 API and identified 38 changes classified into six categories, with the most frequent being the addition of a new parameter. API components deprecation is a practice that can be adopted when following the Two-in-production pattern, which is defined as part of the evolution patterns language by Lübke et al. in [28], as part of a broader effort to document Microservice API Patterns (MAP).

API Management Patterns. In [9], the authors defined an API management pattern language composed of 21 patterns focusing on collaboration. The study identified eight stakeholders in API management and documented 35 pattern candidates and 21 patterns for API management. The paper presents two representative patterns and six general findings related to collaboration between the API provider team and the pattern language. The pattern language is based on 12 cases and semi-structured interviews with 15 API management practitioners from mostly European organizations over half a year.

API Quality Patterns. Also, as part of MAP, Stocker et al. propose in [43] five interface quality patterns that focus on observable aspects of quality-attributed-driven interface design, including efficiency, security, and manageability. These patterns include an API Key, Wish List, Rate Limit, Rate Plan, and Service Level Agreement. The patterns aim to help API designers and product owners strengthen desired quality attributes and communicate quality properties to stakeholders. In this paper, we narrowed our scope to only focus on Rate Limit adoption patterns.

11.2 Web API Rate Limiting

Rate-limiting strategies are commonly used to control access to data and protect backend resources, but setting an appropriate Rate Limit can be challenging. In [12], Firmani et al. propose a

statistical model and a technique based on uniform sampling to select an appropriate Rate Limit for Web APIs and validates it through a case study involving a large bus company. The proposed approach aims to enable organizations to choose a Rate Limit that prevents unauthorized access while still allowing the creation of valuable services, especially for public administrations and private companies providing services whose quality is regulated by formal business agreements on service levels. On another side, in [11], El Malki et al. suggest that Rate Limiting can increase the reliability properties of APIs given a specific workload situation but finding the right balance between improving the success rate and keeping the failure rate at a certain minimum level is challenging. El Malki et al. propose an analytical model to accurately predict the impact of different configurations and workloads on the reliability properties of APIs and a solid method for adaptively fine-tuning rate limits. The model was empirically validated using 50 different configurations in a private cloud and an additional 50 configurations in Google Cloud, and it showed reasonably close prediction errors to reality.

In contrast to prior works, which propose statistical models or analytical approaches to set rate limits for Web APIs, we focus on the Rate Limit pattern and present a comprehensive description of all its configuration possibilities. Our analysis of this API pattern includes both runtime and static analysis perspectives. This level of investigation has not been previously undertaken in the literature. While previous works have validated the effectiveness of rate-limiting strategies in preventing unauthorized access and improving the reliability properties of APIs, our work provides a more detailed examination of the pattern and its various implementation options.

11.3 API Analytics

Adopting a static analysis approach in our previous research on large datasets of OpenAPI²³ descriptions has led to the development of a comprehensive set of methods for API analytics. These methods have enabled us to gain valuable insights into the structural patterns of APIs, the evolution of APIs over time, and specific features and practices such as compatibility and the adoption of deprecation. Our research has provided a thorough understanding of the API landscape and allowed us to create more effective API analysis, design, and visualization tools.

In [40], the authors found that most APIs are small, with a moderate correlation between their functional structures and data models. Another study [23, 24] showed that APIs tend to grow in size and changes to widely used APIs can have a significant impact. To aid API designers, we classified commonly occurring fragments in API structures into pattern primitives and variants [39]. The schema compatibility of a large collection of public web APIs was determined [41], revealing a relatively high number of compatible APIs but a low number of compatible endpoints within the same API. Our analysis of the evolution of API operations [25] showed low utilization of deprecation. Observing versioning practices during Web APIs evolution, the authors of [38] detected, using a systematic approach, the adoption of 55 different version identifier formats, where the commonly adopted one was semantic versioning, which

²³<https://www.openapis.org/>

was used constantly during the histories of 62% of the studied 7114 Web APIs.

12 CONCLUSION

In this paper, we present a collection of patterns related to the adoption of *API Rate Limit* pattern. We grouped them into seven sub-collections depending on the problems they target: how to set the rate limit value, how to meter API usage, how to define its scope, how to implement it on the server side, and how to react to clients exceeding their limit. The findings of our static analysis – performed over a large set of real-world web APIs – indicate that while rate limiting is a well-known method for mitigating resource exhaustion, there is a lack of standardized and widely adopted machine-readable formats for describing rate limits. Moreover, there is no consensus on configuring API hosting platforms and communicating rate limits to API clients.

The patterns we have identified are related to the documentation and communication of API Rate Limits, and the metrics used to statically or dynamically define their values. Other patterns are related to the level of granularity at which a Rate Limit strategy can be applied (i.e., how clients are identified and resources scoped), and implementation-related patterns about the placement of a rate limiter (internal vs. external, local vs. global). We finally distinguish how to mitigate or stop abusive behavior as a reaction to Rate Limit violations: e.g., by blacklisting or throttling clients, temporarily or permanently.

Acknowledgments. We would like to thank our shepherd Dilum Bandara, and the EuroPLoP workshop participants, for their invaluable feedback on our paper. This work was supported by the API-ACE project, funded by SNF project 184692 and FWF (Austrian Science Fund) project I 4268.

REFERENCES

- [1] Custom Rate Limiting for Microservices. <https://dzone.com/articles/rate-limiting-for-microservices>.
- [2] Export a REST API from API Gateway. <https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-export-api.html>.
- [3] Advanced request throttling with Azure API Management. <https://learn.microsoft.com/en-us/azure/api-management/api-management-sample-flexible-throttling>.
- [4] Rate Limiting with NGINX and NGINX Plus. <https://www.nginx.com/blog/rate-limiting-nginx/>.
- [5] Esi Adeborna and Kenneth K Fletcher. An empirical study of web api quality formulation. In *Services Computing–SCC 2020: 17th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, September 18–20, 2020, Proceedings 17*, pages 145–153. Springer, 2020.
- [6] Suhail Ahmad and Ajaz Hussain Mir. Protection of centralized sdn control plane from high-rate packet-in messages. *International Journal of Information Security*, April 2023. doi: 10.1007/s10207-023-00685-z.
- [7] Alexander Bakhtin, Abdullah Al Maruf, Tomas Cerny, and Davide Taibi. Survey on tools and techniques detecting microservice api patterns. In *2022 IEEE International Conference on Services Computing (SCC)*, pages 31–38. IEEE, 2022.
- [8] David Bernbach and Erik Wittern. Benchmarking web api quality – revisited. *Journal of Web Engineering*, 19(5-6):603–646, Oct. 2020. doi: 10.13052/jwe1540-9589.19563. URL <https://journals.riverpublishers.com/index.php/JWE/article/view/5719>.
- [9] Gloria Bondel, Andre Landgraf, and Florian Matthes. Api management patterns for public, partner, and group web api initiatives with a focus on collaboration. In *26th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–17, 2021.
- [10] Amine El Malki and Uwe Zdun. Combining api rate limiting, request bundle and load balancing patterns in microservice architectures: Performance and reliability analysis. In *Submitted for publication*, 2022.
- [11] Amine El Malki, Uwe Zdun, and Cesare Pautasso. Impact of api rate limit on reliability of microservices-based architectures. In *16th International Conference on Service-Oriented System Engineering (SOSE 2022)*, pages 19–28, San Francisco, USA, August 2022. IEEE.
- [12] Donatella Firmani, Francesco Leotta, and Massimo Mecella. On computing throttling rate limits in web apis through statistical inference. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 418–425. IEEE, 2019.
- [13] Antonio Gamez-Diaz, Pablo Fernandez, Antonio Ruiz-Cortés, Pedro J Molina, Nikhil Kolekar, Prithpal Bhogill, Madhurranjan Mohaan, and Francisco Méndez. The role of limitations and slas in the api industry. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1006–1014, 2019.
- [14] Google. Rate limiting strategies and techniques, 08 2019. URL <https://cloud.google.com/architecture/rate-limiting-strategies-techniques>.
- [15] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback control of computing systems*. John Wiley & Sons, 2004.
- [16] J. Higginbotham. *Principles of Web API Design: Delivering Value with APIs and Microservices*. Addison-Wesley Signature Series. Pearson Education (US), 2021. ISBN 9780137355631.
- [17] James Higginbotham. Cloud native cloud native api management, 2020. URL <https://www.enable-u.nl/wp-content/uploads/2021/03/White-Paper-Cloud-Native.pdf>.
- [18] Alefiya Hussain, John Heidemann, and Christos Papadopoulos. A framework for classifying denial of service attacks. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM ’03, page 99–110. ACM, 2003. ISBN 1581137354. doi: 10.1145/863955.863968.
- [19] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Classification of changes in api evolution. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 243–249. IEEE, 2019.
- [20] Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. Web api evolution patterns: A usage-driven approach. *Journal of Systems and Software*, 198:111609, 2023. doi: 10.1016/j.jss.2023.111609.
- [21] Arne Koschel, Marvin Bertram, Richard Bischof, Kevin Schulze, Marc Schaaf, and Irina Astrova. A look at service meshes. In *Proc. 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, pages 1–8. IEEE, 2021.
- [22] Alexander Krause, Christian Zirkelbach, Wilhelm Hasselbring, Stephan Lenga, and Dan Kröger. Microservice decomposition via static and dynamic analysis of the monolith. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 9–16. IEEE, 2020.
- [23] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. Towards large-scale empirical assessment of web apis evolution. In *21st International Conference on Web Engineering (ICWE2021)*, pages 124–138, Biarritz, France, May 2021. Springer.
- [24] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. A large-scale empirical assessment of web api size evolution. *Journal of Web Engineering*, 21(6):1937–1980, November 2022.
- [25] Fabio Di Lauro, Souhaila Serbout, and Cesare Pautasso. To deprecate or to simply drop operations? an empirical study on the evolution of a large openapi collection. In *16th European Conference on Software Architecture (ECSA)*, pages 38–46, Prague, Czech Republic, September 2022.
- [26] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225. IEEE, 2019.
- [27] John DC Little and Stephen C Graves. Little’s law. *Building intuition: insights from basic operations management models and principles*, pages 81–100, 2008.
- [28] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs (EuroPLoP)*, pages 1–24, 2019.
- [29] Amine El Malki and Uwe Zdun. Evaluation of api request bundling and its impact on performance of microservice architectures. In *IEEE International Conference on Services Computing (SCC 2021)*, September 2021. doi: <https://doi.org/10.5281/zenodo.5087467>. URL <http://eprints.cs.univie.ac.at/6898/>.
- [30] Amine El Malki, Uwe Zdun, and Cesare Pautasso. Impact of api rate limit on reliability of microservices-based architectures. In *16th IEEE International Conference on Service-Oriented System Engineering (SOSE2022)*, 2022. URL <http://eprints.cs.univie.ac.at/7399/>.
- [31] Haithem Mezni. Web service adaptation: A decade’s overview. *Computer Science Review*, 48:100535, 2023.
- [32] Edward A Morse and Vasant Raval. Pci dss: Payment card industry data security standards in context. *Computer Law & Security Review*, 24(6):540–554, 2008.
- [33] Stefan Nastic, Andrea Morichetta, Thomas Pusztai, Schahram Dustdar, Xiaoning Ding, Deepak Vij, and Ying Xiong. Sloc: Service level objectives for next generation cloud computing. *IEEE Internet Computing*, 24(3):39–50, 2020.
- [34] Evangelos Ntontos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *18th International Conference on Service Oriented Computing (ICSOC 2020)*, page 580–596, December 2020.

- [35] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, 2017.
- [36] Marek Polák and Irena Holubová. REST API management and evolution using MDA. In *Proceedings of the Eighth International C* Conference on Computer Science & Software Engineering*, pages 102–109, 2015.
- [37] Chris Richardson. Microservice architecture patterns and best practices. URL: <http://microservices.io/index.html> [accessed: 2018-03-17], 2016.
- [38] Souhaila Serbout and Cesare Pautasso. An empirical study of web api versioning practices. In *23rd International Conference on Web Engineering (ICWE2023)*, Alicante, Spain, June 2023. Springer.
- [39] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. From openapi fragments to api pattern primitives and design smells. In *European Conference on Pattern Languages of Programs (EuroPLOP’21)*, Virtual Kloster Irsee, Germany, July 2021. ACM.
- [40] Souhaila Serbout, Fabio Di Lauro, and Cesare Pautasso. Web apis structures and data models analysis. In *Companion Proc. 19th International Conference on Software Architecture (ICSA)*, pages 84–91, 2022.
- [41] Souhaila Serbout, Cesare Pautasso, and Uwe Zdun. How composable is the web? an empirical study on openapi data model compatibility. In *IEEE World Congress on Services (ICWS Symposium on Services for Machine Learning)*, Barcelona, Spain, July 2022. IEEE.
- [42] Salah Sharieh and Alexander Ferworn. Securing apis and chaos engineering. In *2021 IEEE Conference on Communications and Network Security (CNS)*, pages 290–294. IEEE, 2021.
- [43] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, pages 1–16, 2018.
- [44] Luis Von Ahn, Manuel Blum, Nicholas J Hopper, and John Langford. CAPTCHA: Using hard AI problems for security. In *Eurocrypt*, volume 2656, pages 294–311. Springer, 2003.
- [45] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *Proc. 16th International Conference on Service-Oriented Computing (ICSOC)*, pages 73–89, Cham, 2018. Springer. ISBN 978-3-030-03596-9.
- [46] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *International Conference on Service-Oriented Computing (ICSOC)*, pages 73–89. Springer, 2018.
- [47] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. Interface representation patterns: crafting and consuming message-based remote apis. In *Proceedings of the 22nd european conference on pattern languages of programs (EuroPLOP)*, pages 1–36, 2017.
- [48] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. Microservice api patterns: Rate limit, 2020. URL <https://www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/RateLimit>.
- [49] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Uwe Zdun, and Cesare Pautasso. *Patterns for API Design – Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley Professional, Vaughn Vernon Signature Series, 2022.