KU LEUVEN

FACULTY OF
ENGINEERING SCIENCE

Using the OpenAPI specification for REST API security

Stef Van Poucke

Thesis submitted for the degree of
Master of Science in Cybersecurity

Supervisors

Prof. dr. ir. Wouter Joosen
Dr. Pieter Philippaerts

Assessors

Dr. ir. Sayon Duttagupta
Dr. ir. Kristof Jannes

Assistant-supervisor

Dr. Pieter Philippaerts

Academic year 2025 – 2026

Published by Stef Van Poucke,
Faculty of Engineering Science, Kasteelpark Arenberg 1 bus 2200, B-3001 Leuven

Preface

This thesis marks the end of an intensive and valuable learning process. I would like to thank everyone who supported me during this work.

First, I would like to thank my supervisor, Dr. Pieter Philippaerts, for his guidance, feedback, and support throughout the development of this thesis. His input helped me refine the direction of the work and improve the quality of the final result.

I would also like to express my sincere gratitude to my manager, Yannick Betrains, and to Johan Vandewalle, General Manager of Antwerp Terminal Services (ATS), for their support and trust throughout my studies. Their willingness to provide me with the flexibility needed to successfully combine my professional and academic responsibilities made it possible to pursue both paths simultaneously. I am also deeply grateful for the investment in my professional development through the funding of my Master's degree in Cybersecurity.

I would also like to thank my family for their encouragement and patience. Their support gave me the motivation to continue working through the more challenging moments of this project.

Finally, I would like to thank my cats for their questionable but persistent contributions to the writing process, usually by sitting exactly where they were least helpful.

Stef Van Poucke

Contents

Abstract

Application Programming Interfaces (APIs) have become a foundational component of modern software systems. They enable integration and communication between distributed applications, cloud services, mobile clients, and third-party platforms. This widespread adoption has also increased the exposure of APIs to abuse and attacks. Security frameworks such as the OWASP API Security Top 10 identify unrestricted resource consumption as an important risk in API-driven architectures.

The OpenAPI Specification is widely used to describe, document, and generate HTTP APIs. It supports contract-first development and forms a central part of many modern API development workflows. While OpenAPI can formally describe API structure and security mechanisms such as authentication, it does not currently provide a standard way to express resource consumption constraints such as rate limiting. As a result, these controls are often implemented separately in application code, API gateways, or infrastructure configuration. This separation can lead to inconsistencies between the documented API contract, the generated implementation, and the actual runtime behavior.

This thesis proposes an OpenAPI vendor extension for formally defining rate limiting constraints at the API contract level. The proposed `x-rateLimit` extension supports common rate limiting algorithms, including fixed window, sliding window, and token bucket policies, as well as different keying strategies such as IP-based, header-based, query-based, and path-based limits. The thesis also investigates how server-side code generation can be adapted so that these policies are automatically enforced in generated API implementations.

The approach is evaluated through a proof-of-concept implementation based on OpenAPI Generator and a generated Python Flask server. In addition, a specification-driven test tool is developed to validate whether generated endpoints behave according to the rate limiting policies defined in the OpenAPI document. The results show that rate limiting policies can be modeled at the specification level and translated into working server-side enforcement. This supports a more consistent, contract-first approach to API security and reduces duplication between specification, implementation, and testing.

List of Figures and Tables

List of Figures

List of Tables

List of abbreviations and symbols

Abbreviations

API	Application Programming Interface
BOLA	Broken Object Level Authorization
CORS	Cross-Origin Resource Sharing
CSV	Comma-Separated Values
DDoS	Distributed Denial of Service
EMEA	Europe, Middle East, and Africa
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDOR	Insecure Direct Object Reference
IoT	Internet of Things
IP	Internet Protocol
ISO	International Organization for Standardization
JSON	JavaScript Object Notation
OAS	OpenAPI Specification
OAuth	Open Authorization
OWASP	Open Worldwide Application Security Project
REST	Representational State Transfer
RQ	Research Question
SSRF	Server-Side Request Forgery
UI	User Interface
URI	Uniform Resource Identifier
WAF	Web Application Firewall
XML	eXtensible Markup Language
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Context

Modern software systems increasingly rely on Web APIs (Application Programming Interfaces) as their primary communication mechanism. APIs expose structured functionality and data to external clients, allowing integrations between services, mobile applications, cloud systems and third-party platforms.

Unlike traditional web applications that primarily serve human users, APIs are designed for automated, machine-to-machine interaction. This characteristic makes them an attractive target for automated abuse such as scraping, brute-force attacks, and resource exhaustion. Recent industry reports [1] show that nearly half of all web application attacks in the Europe, Middle East, and Africa (EMEA) region target APIs. This highlights the need for dedicated API security controls.

The OpenAPI Specification (OAS) is a widely used standard for describing HTTP APIs. It enables contract-first development, automatic documentation, and server/client code generation. While OpenAPI supports the formal definition of some security aspects like authentication mechanisms, it does not natively describe constraints related to resource consumption such as rate limiting or request quotas.

Unrestricted resource consumption is identified by the Open Worldwide Application Security Project (OWASP) as API4:2023 in the OWASP API Security Top 10, a widely recognized industry standard that categorizes the most critical API security risks. This risk refers to the failure to properly restrict the number, size, or computational complexity of requests, potentially leading to denial-of-service conditions or degraded service availability. Although resource consumption issues are sometimes perceived as performance or scalability concerns, they are in fact security problems: attackers can deliberately exploit the absence of such controls to exhaust system resources and disrupt availability.

From a security perspective, this means that mechanisms such as rate limiting are essential safeguards. Without these controls, APIs remain vulnerable to automated abuse, including brute-force attacks, scraping, and distributed denial-of-service (DDoS) scenarios.

Today, rate limiting is typically implemented in server-side code or configured

separately at the API gateway level. These policies are not part of the API specification and must be managed independently. This separation introduces several challenges: rate limiting rules are often duplicated across environments, inconsistently applied, and difficult to validate against the intended API contract.

Defining rate limiting at the API specification level would establish a single source of truth for both functional and non-functional constraints. This improves consistency between documentation, implementation, and deployment configurations. It also enables better portability, as the same specification can be used across different API gateways and server frameworks without redefining policies.

Furthermore, integrating rate limiting into the OpenAPI Specification allows tooling to reason about these constraints. Code generators can automatically include enforcement logic in generated server implementations, while API gateways can derive configuration directly from the specification. This reduces manual configuration effort, minimizes misconfigurations, and supports a more declarative and contract-first approach to API security.

1.2 Goals

The goal of this thesis is to design, formalize, and evaluate an extension to the OpenAPI Specification that enables the definition of rate limiting and resource consumption constraints at the API contract level.

Currently, such constraints are implemented outside the API specification, leading to fragmentation between design, implementation, and deployment. This work aims to bridge that gap by introducing a machine-readable and semantically well-defined model that can be embedded into OpenAPI documents using vendor extensions. The proposed extension must remain compatible with the OpenAPI 3.x specification, while being expressive enough to capture commonly used rate limiting strategies.

In addition to defining the extension, this thesis investigates how such specification-level constructs can be put into practice. This includes integrating the proposed model into an existing OpenAPI code generation workflow and automatically translating rate limiting definitions into executable server-side logic. As such, the work not only focuses on specification design, but also on its practical applicability in real-world development environments.

The evaluation of this approach focuses on two main aspects. First, the expressiveness and correctness of the proposed model are assessed by mapping real-world rate limiting configurations to the extension. Second, the feasibility and usability of automated enforcement are evaluated by generating server implementations and verifying whether the specified policies are correctly enforced at runtime.

Based on these objectives, the following research questions are defined:

> *Can modern rate limiting configurations be formally modeled and incorporated into the OpenAPI Specification?* (RQ1)

Can the proposed OpenAPI Specification extension be leveraged to generate server-side implementations that enforce the specified rate limiting policies? (RQ2)

1.3 Overview

To systematically address the research questions, the remainder of this thesis is organized as follows.

Chapter 2 - Background provides the necessary theoretical and technical background. It introduces Web APIs and their role in modern systems, discusses common API security risks with a focus on the OWASP API Security Top 10 and explains the fundamentals of the OpenAPI Specification. Additionally, it reviews commonly used rate limiting algorithms and their trade-offs.

Chapter 3 - Design presents the proposed extension to the OpenAPI Specification. It explains the rationale for using vendor extensions and formally defines the structure, semantics and constraints of the extension. The chapter also discusses design choices and how different rate limiting strategies are represented.

Chapter 4 - Implementation describes how the proposed extension is integrated into the OpenAPI code generation workflow. It details the modifications required to support the extension and explains how rate limiting policies are translated into executable server-side logic. The chapter also outlines the overall architecture of the generated solution.

Chapter 5 - Evaluation assesses the proposed approach from both a functional and practical perspective. It evaluates whether the extension can represent common rate limiting configurations, whether these policies are correctly translated into generated enforcement code and whether generated implementations enforce the specified policies at runtime. Additionally, it discusses maintainability, limitations, and the trade-offs between application-level and infrastructure-level enforcement.

Finally, Chapter 6 - Conclusion summarizes the main findings of the thesis. It reflects on the research questions, highlights key contributions and outlines directions for future work.

Chapter 2

Background

This chapter provides an overview of the core concepts required to understand the problem domain of this thesis. It provides background on Web APIs, their security challenges, and the mechanisms used to describe and protect them.

Section 2.1 explains the structure and operation of Web APIs, including methods, data formats, authentication mechanisms, and error handling. Section 2.2 presents the OWASP API Security Top 10, highlighting common vulnerabilities, with particular focus on unrestricted resource consumption. Section 2.3 introduces the OpenAPI Specification as a machine-readable API description format and discusses its role in modern development workflows. Section 2.4 covers rate limiting techniques, including common algorithms and policy considerations, which form the basis for the proposed extension in this thesis. Finally, Section 2.5 reviews related work on the use of OpenAPI for security analysis, testing, and API security automation.

2.1 Web APIs

HTTP-based Application Programming Interfaces (APIs) enable communication between systems using the Hypertext Transfer Protocol (HTTP). They expose resources through Uniform Resource Identifiers (URIs) and allow clients to interact with these resources using standardized HTTP methods. Due to their simplicity, interoperability, and compatibility with existing web infrastructure, HTTP APIs are widely adopted in web applications, mobile backends, microservices architectures, and IoT systems.

2.1.1 Request and response

At their core, Web APIs follow a request–response model. A client sends an HTTP request to a specific endpoint, optionally including parameters, headers, and a request body. The server processes the request and returns a response containing a status code and, in most cases, structured data.

The following example illustrates a real-world Web API interaction using cloud storage. The example server allows a user to store and manipulate files using

a Web API. Figure 2.1 shows an example request to retrieve the current contents of the storage. Part of the URI refers to the specific storage being accessed, `stmpetanalyticsdev011`. The provided parameters indicate both the targeted resource and the requested operation, namely listing its contents.

Figure 2.2 shows the server response to the request. An HTTP status code of 200 (OK) is returned, along with a response body containing the requested data.

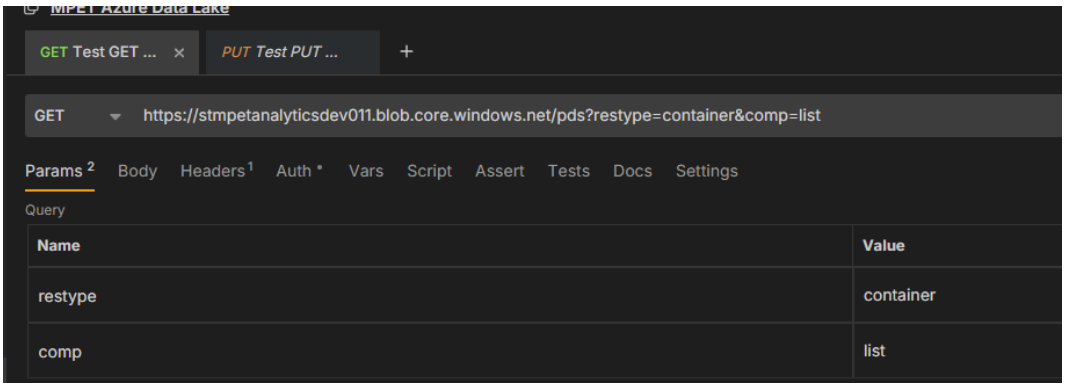

Figure 2.1: Example GET request to the Azure Storage API

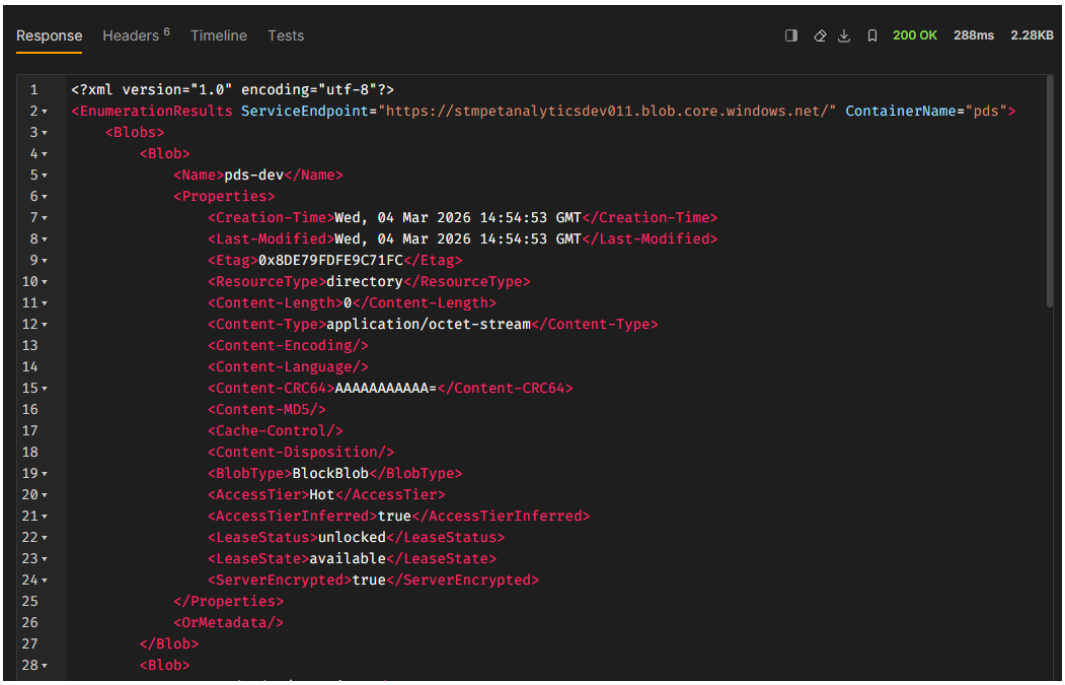

Figure 2.2: Example GET response from the Azure Storage API

2.1.2 HTTP Methods

HTTP methods define the action to be performed on a resource. The GET method retrieves a representation of a resource without modifying it, as shown in subsection 2.1.1. The POST method creates a new resource or triggers server-side operations. Other HTTP methods provide additional functionality: PUT replaces an existing resource entirely, PATCH applies partial modifications, and DELETE removes a resource from the server.

Expanding on the cloud storage example, Figures 2.3 and 2.4 show the request and response for a PUT operation. The request creates a file named `test321.json` with the content `{"hello":"world"}`. The response includes metadata in the headers and returns a 201 (Created) status code, indicating that the resource was successfully created.

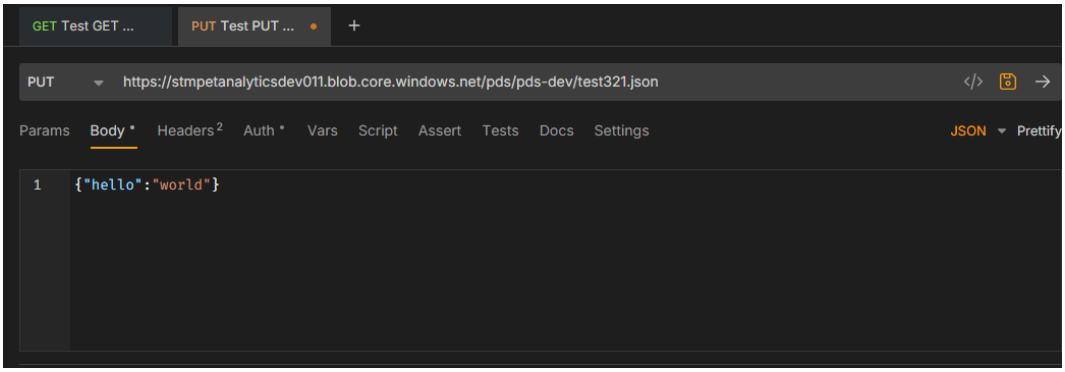

FIGURE 2.3: Example PUT request to the Azure Storage API

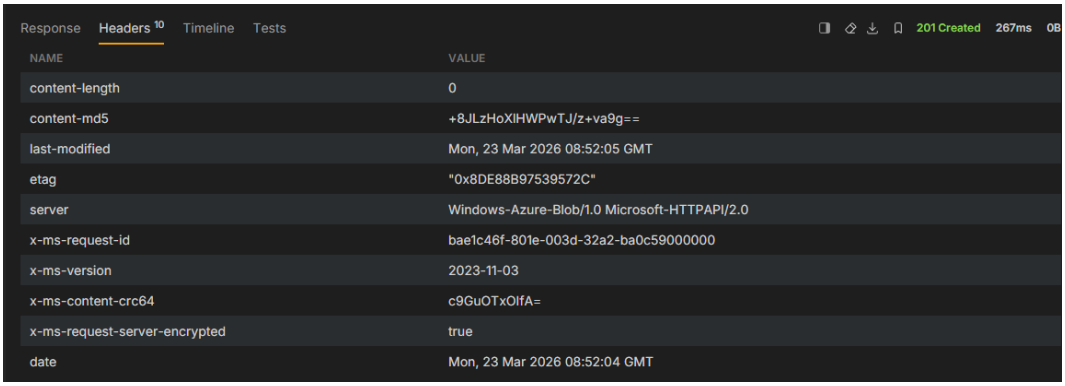

FIGURE 2.4: Example PUT response from the Azure Storage API

The created file can subsequently be removed using the DELETE method, as shown in Figures 2.5 and 2.6. In this case, the response returns a 202 (Accepted) status code, indicating that the deletion request has been accepted for processing.

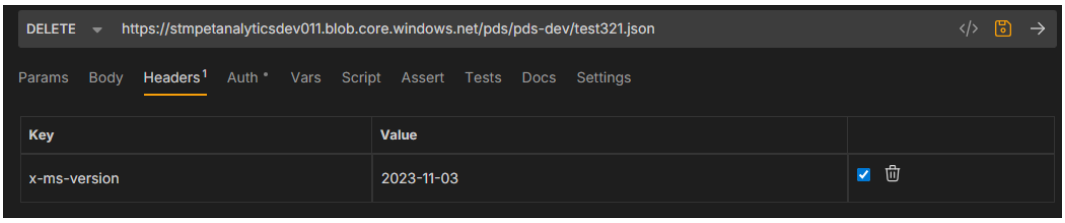

Figure 2.5: Example DELETE request to the Azure Storage API

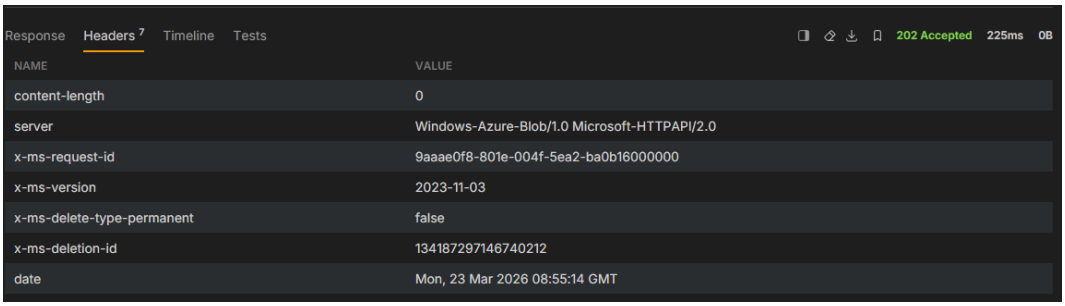

Figure 2.6: Example DELETE response from the Azure Storage API

In addition to these commonly used methods, HTTP defines several others. HEAD is similar to GET but only retrieves response headers without a message body, making it useful for checking resource availability or metadata. OPTIONS is used to determine which HTTP methods are supported by a server or endpoint and is commonly used in Cross-Origin Resource Sharing (CORS) preflight requests. TRACE is intended for diagnostic purposes by echoing the received request, although it is often disabled for security reasons. CONNECT establishes a tunnel to the server, typically used for HTTPS communication through proxies.

The standardized semantics of these methods enable consistent interface design and simplify implementation.

2.1.3 Data Formats

Web APIs commonly exchange structured data formats such as JavaScript Object Notation (JSON) and Extensible Markup Language (XML). JSON has become the dominant format due to its lightweight syntax, ease of parsing, and human readability. XML is still used in legacy systems and in domains where strict schema validation is required, as illustrated in the storage example in Figure 2.2. Other formats, such as Protocol Buffers or YAML, may be used in specific contexts, particularly where performance or compact serialization is important.

2.1.4 Status Codes and Error Handling

HTTP status codes communicate the outcome of a request and are grouped into five categories based on their semantics. Each category reflects a different stage in

request processing and helps clients understand how to handle responses.

Informational responses (1xx) indicate that the request has been received and is being processed. For example, 100 (Continue) signals that the client may proceed with sending the remainder of the request.

Successful responses (2xx) confirm that the request was handled correctly. Common examples include 200 (OK) for a standard successful response and 201 (Created) when a new resource has been created.

Redirection responses (3xx) indicate that further action is required by the client to complete the request. For instance, 301 (Moved Permanently) redirects the client to a new location, while 307 (Temporary Redirect) indicates that the resource is temporarily available at a different URI and that the original request method should be preserved.

Client error responses (4xx) indicate that the request cannot be fulfilled due to an issue on the client side. Typical examples include 400 (Bad Request), 401 (Unauthorized), 404 (Not Found), and 429 (Too Many Requests), the latter being particularly relevant for rate limiting.

Server error responses (5xx) indicate that the server failed to process a valid request. Examples include 500 (Internal Server Error) and 503 (Service Unavailable), often caused by overload or maintenance.

Consistent use of status codes combined with structured error responses improves reliability, debuggability, and overall developer experience. Web APIs provide a standardized and scalable mechanism for system integration and form a foundational component of modern distributed software architectures.

2.1.5 Authentication Mechanisms

Web APIs often expose sensitive data or operations and therefore require mechanisms to verify the identity of clients. Authentication ensures that only authorized users or systems can access protected resources and is typically enforced before any request is processed.

Several authentication approaches are commonly used in Web APIs. Simple mechanisms such as API keys include a static token in a request header or query parameter to identify the client. More secure approaches rely on token-based authentication, where a client first obtains a token from an authorization server and includes it in subsequent requests. This approach avoids transmitting credentials with every request and enables fine-grained access control.

A widely adopted standard for token-based authentication is OAuth 2.0. OAuth 2.0 separates authentication from resource access by introducing an authorization server that issues access tokens with specific scopes and lifetimes. These tokens are then included in API requests, typically in the `Authorization` header as a bearer token.

Figure 2.7 shows the OAuth 2.0 configuration used for the cloud storage example in this chapter, including client credentials and token endpoints for obtaining access tokens.

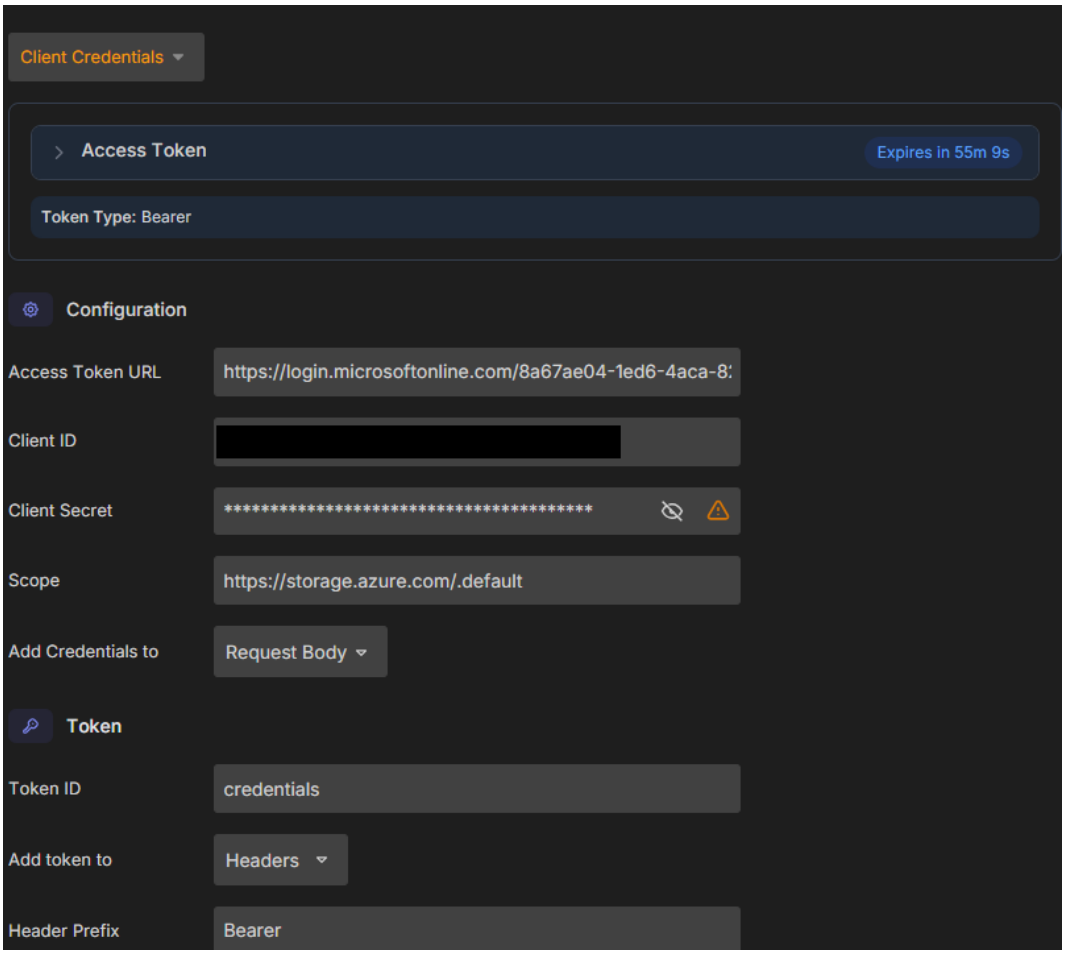

Figure 2.7: Example OAuth 2.0 configuration for API authentication

By using standardized authentication mechanisms such as OAuth 2.0, Web APIs can securely manage access while supporting scalable and distributed architectures.

2.2 OWASP API Security Top 10

The Open Worldwide Application Security Project (OWASP)[1] provides widely recognized guidance on application security risks. In response to the increasing reliance on APIs in modern software systems, OWASP introduced a dedicated *API Security Top 10*, highlighting the most critical API-specific vulnerabilities.

The 2023 edition [12] identifies the following key risks:

- **API1:2023 – Broken Object Level Authorization**
- **API2:2023 – Broken Authentication**

[1] https://owasp.org/

- **API3:2023 – Broken Object Property Level Authorization**
- **API4:2023 – Unrestricted Resource Consumption**
- **API5:2023 – Broken Function Level Authorization**
- **API6:2023 – Unrestricted Access to Sensitive Business Flows**
- **API7:2023 – Server Side Request Forgery (SSRF)**
- **API8:2023 – Security Misconfiguration**
- **API9:2023 – Improper Inventory Management**
- **API10:2023 – Unsafe Consumption of APIs**

Of particular relevance to this thesis is **API4:2023 – Unrestricted Resource Consumption**. This risk refers to insufficient limitations on request rates, payload sizes, computational complexity, or concurrent usage. If such controls are absent or improperly configured, attackers may deliberately exhaust server resources, leading to denial-of-service conditions or severe performance degradation.

MITRE categorizes software vulnerabilities using the Common Weakness Enumeration (CWE) system. CWE is a standardized and publicly available classification of software weaknesses, designed to provide a common language for describing security flaws in software and hardware systems. Rather than focusing on specific vulnerabilities or exploits, CWE describes underlying weakness types that can lead to security issues, such as improper input validation, missing authorization checks, or uncontrolled resource usage. This abstraction allows developers, security researchers, and tool vendors to consistently identify, analyze, and mitigate classes of weaknesses across different systems.

The following CWE entries are closely related to **API4:2023 – Unrestricted Resource Consumption**:

- **CWE-400**: Uncontrolled Resource Consumption
- **CWE-770**: Allocation of Resources Without Limits or Throttling
- **CWE-799**: Improper Control of Interaction Frequency

CWE-400 Was listed by MITRE as one of the top 25 most dangerous software weaknesses [11].

These weaknesses emphasize the importance of implementing effective rate limiting, quota enforcement, and computational safeguards. As APIs increasingly serve as direct entry points to backend systems, formalizing and consistently enforcing such constraints has become a critical security requirement.

2.3 The OpenAPI Specification

The OpenAPI Specification (OAS)[2] is a widely adopted, language-agnostic standard for formally describing HTTP-based APIs in a structured and machine-readable format. Originally introduced as the Swagger specification, it is now maintained by the OpenAPI Initiative under the Linux Foundation and has become the industry standard for API description across enterprises and cloud providers.

An OpenAPI document provides a comprehensive description of an API's interface. It defines:

- Available endpoints (paths) and supported HTTP methods
- Request parameters, headers, query parameters, and request bodies
- Data schemas for input validation
- Response formats and associated HTTP status codes
- Security schemes and authentication requirements
- Metadata such as versioning, server locations, and descriptive documentation

OpenAPI documents are written in YAML or JSON and function as explicit API contracts between service providers and consumers. By separating interface definition from implementation, OAS enables a clear contract-driven development model in which stakeholders can agree on functionality before code is written.

Because the specification is machine-readable, it enables a rich ecosystem of automated tooling. Common use cases include:

- Interactive documentation generation (e.g., Swagger UI)
- Automatic request and response validation
- Server stub generation for multiple programming languages
- Client code generation for multiple programming languages
- Import into API gateways and API management platforms

In contract-first development approaches, the OpenAPI document becomes the central design element of the system. Server implementations and client libraries can be generated directly from the specification, reducing inconsistencies between documentation and code. This approach improves interoperability, accelerates development, and reduces integration errors.

The OpenAPI Specification also provides extensibility mechanisms through vendor-specific extensions (commonly prefixed with `x-`). These extensions allow organizations to attach additional metadata or custom semantics without violating the core

[2] https://spec.openapis.org/oas/v3.1.0.html

specification. Such flexibility makes OAS suitable not only for documentation, but also as a configuration source for runtime infrastructure components.

Vendor-specific extensions are also the way to propose extensions to the official specification itself. This thesis will use vendor-specific extensions to extend the OpenAPI Specification with rate-limiting definitions.

2.4 Rate Limiting

Rate limiting is a defensive control that restricts how many requests a client may perform within a defined time interval. It is widely used to mitigate resource exhaustion, brute-force attacks, scraping, and denial-of-service attempts.

Several algorithmic approaches exist for implementing rate limiting, each offering different trade-offs in burst tolerance, memory consumption and algorithm complexity.

2.4.1 Fixed Window

The fixed window algorithm counts requests within discrete time intervals of fixed length (e.g., 100 requests per minute). Once the limit is reached within a window, subsequent requests are rejected until the next window begins.

This approach is computationally simple and easy to implement, often requiring only a counter and a timestamp. However, it suffers from the *boundary burst problem*: a client may send requests at the end of one window and immediately again at the beginning of the next, effectively doubling the allowed rate in a short period.

Due to its simplicity, fixed window rate limiting is commonly implemented in API gateways and cloud API management services.

2.4.2 Sliding Window

The sliding window algorithm evaluates requests over a rolling time interval rather than fixed discrete windows. Each incoming request is checked against the number of requests received in the preceding time span (e.g., the last 60 seconds).

This approach provides smoother enforcement and mitigates boundary burst effects. Implementations typically rely on timestamped logs or approximations using weighted counters.

While more accurate than fixed windows, sliding window algorithms require additional memory and computational overhead.

2.4.3 Token Bucket

The token bucket algorithm generates tokens at a fixed rate and stores them in a bucket with a maximum capacity. Each incoming request consumes one token. If no tokens are available, the request is rejected.

This model allows short bursts of traffic (up to the bucket capacity) while enforcing a long-term average rate. It is particularly suitable for APIs that need to tolerate temporary spikes without violating sustained throughput limits.

Token bucket implementations are widely used in networking and traffic shaping and are supported by several API gateways.

2.4.4 Leaky Bucket

The leaky bucket algorithm enforces a constant request processing rate. Incoming requests are queued and processed at a steady rate, effectively smoothing traffic spikes.

Unlike token bucket, which controls admission, leaky bucket controls output rate. If the queue becomes full, additional incoming requests are dropped.

Leaky bucket is often used in traffic shaping and infrastructure-level throttling, but is less common in application-layer API rate limiting frameworks.

2.4.5 Comparison of Algorithms

Each algorithm presents distinct trade-offs:

- **Fixed Window:** Simple, low overhead, but prone to burst effects.
- **Sliding Window:** Fairer enforcement, higher accuracy, increased complexity.
- **Token Bucket:** Supports controlled bursts with long-term fairness.
- **Leaky Bucket:** Smooth output rate, suitable for traffic shaping.

In distributed environments, additional challenges arise, such as synchronization across multiple nodes. These concerns often require centralized storage or gateway-level enforcement.

2.4.6 Rate Limiting Keys and Policy Scope

Another important aspect of rate limiting concerns the key used to enforce policies. Rate limits are typically not applied globally to all traffic, but are scoped to a specific identifier. Common identifiers include IP address, authenticated user or API key.

The choice of rate limiting key directly affects fairness and isolation between clients. For example, per-IP limits may unintentionally group multiple users behind a shared proxy, while per-user or per-token limits provide finer-grained control.

Rate limiting is also frequently applied in a layered manner. An API may define a global rate limit for the entire application, combined with more specific limits per endpoint or per operation. Such hierarchical policies allow coarse-grained protection at system level while maintaining fine-grained control over sensitive or resource-intensive operations.

2.5 Related Work

The OpenAPI Specification (OAS) has been used in several research efforts as a machine-readable source for API security analysis and automated testing. Existing work mainly uses OpenAPI documents to detect potential vulnerabilities, generate security tests, or reason about API design issues. This thesis builds on the same observation that an OpenAPI document can be processed automatically, but applies it to the specification and enforcement of rate limiting policies.

2.5.1 OpenAPI-based security analysis

Several approaches use OpenAPI specifications for static or semi-automatic security analysis. Barabanov et al. [4] propose an approach for detecting potential Insecure Direct Object Reference (IDOR) and Broken Object Level Authorization (BOLA) vulnerabilities by processing OpenAPI specifications. Their method maps properties of endpoints and parameters to possible attack techniques and produces potential attack vectors or security requirements. This shows how API specifications can be used during design or threat modeling, even before dynamic testing is performed.

Cheh and Chen [5] analyze OpenAPI specifications as graphs to identify security design issues. Their approach models relationships between API calls, resources, and data fields to infer which fields may be sensitive and which API calls may require additional protection. This demonstrates that OpenAPI documents can contain useful contextual information for reasoning about security risks at design time.

Mazidi et al. [9] focus on mass assignment vulnerabilities in REST APIs. Their LightMass approach mines OpenAPI specifications to identify operations and attributes that may be prone to mass assignment. By comparing attributes returned by read operations with attributes accepted by write operations, the approach identifies fields that may be unintentionally modifiable. This is relevant because it shows how specification-level analysis can be used to identify security risks before interacting with a running API.

2.5.2 OpenAPI-based security testing

Other work uses OpenAPI specifications to generate dynamic tests. Taya et al. [14] propose an automated vulnerability assessment approach that considers both API requests and responses. Their method generates test cases from API reference information and evaluates observed behavior to detect authorization-related flaws.

Atlidakis et al. [3] extend stateful REST API fuzzing with security property checkers. Their work defines rules such as use-after-free, resource-leak, resource-hierarchy, and user-namespace properties, and then generates request sequences to test whether a running API violates these rules. This line of work is relevant because it demonstrates that OpenAPI specifications can drive systematic runtime validation, not only documentation or static analysis.

Together, these approaches show that OpenAPI documents can support both static and dynamic security analysis. However, they primarily focus on detecting

vulnerabilities or testing whether an existing implementation violates expected security properties.

2.5.3 Rate limiting and API policy specification

Rate limiting has also been studied as an API design and management concern. Serbout et al. [13] present a pattern collection for API rate limit adoption. Their work identifies patterns related to rate limit configuration, metrics, documentation, communication, granularity, reactions to exceeded limits, and server-side implementation. This is closely related to the design of the proposed `x-rateLimit` extension, because these categories correspond to several decisions that must be captured in a reusable rate limiting model.

Their work also highlights that rate limit policies are not documented consistently across APIs and that machine-readable descriptions of rate limits are not yet widely standardized. This supports the motivation for representing rate limiting policies directly in the OpenAPI document rather than only in natural-language documentation or external gateway configuration.

Kalyanasundaram et al. [7] provide a broader survey of API rate limiting methods. They discuss rate limit definitions, enforcement granularity, implementation placement, distributed enforcement, algorithms, communication mechanisms, and policy actions. This survey is useful for positioning rate limiting as a broader design problem that involves more than choosing an algorithm. It also shows that implementation placement and distributed state management are important considerations when moving from specification-level policy definition to production deployment.

2.5.4 Positioning of this thesis

Existing OpenAPI-based security research shows that API specifications can be used for vulnerability discovery, design analysis, and automated test generation. Existing rate limiting research shows that rate limiting is an important API management pattern and that rate limit policies involve several recurring design choices, including configuration, scope, granularity, and implementation placement.

However, the combination of these two directions remains underexplored. Prior work does not provide a concrete OpenAPI extension for expressing rate limiting policies as part of the API contract, nor does it demonstrate how such policies can be translated into generated server-side enforcement and then validated through specification-driven testing. This thesis addresses that gap by proposing the `x-rateLimit` extension and evaluating a proof-of-concept workflow from specification, to generated enforcement, to automated validation.

Chapter 3

Design

This chapter addresses Research Question 1:

> *Can modern rate limiting configurations be formally modeled and incorporated into the OpenAPI Specification?*

The objective of this chapter is to design a formal, machine-readable model for rate limiting policies that can be embedded directly into an OpenAPI document. The proposed design must remain compatible with the OpenAPI 3.x specification, preserve backward compatibility, and allow automated processing by tooling such as code generators and API gateways.

3.1 Real-World Implementations

To determine whether modern rate limiting configurations can be formally represented within the OpenAPI Specification, it is necessary to first examine how rate limiting is implemented in contemporary API management platforms.

Several widely used API gateways and API management solutions were analyzed. The supported algorithms and keying mechanisms were derived from the official product documentation listed in Table 3.1.

These systems represent the current state of practice in API traffic management and are widely deployed in production environments.

Most implementations are based on a small number of well-known algorithms such as fixed window, sliding window, and token bucket rate limiting. Similarly, most allow for the configuration of a few default keys like IP, APIKey, UserID or Subscription. Some also allow specifying a custom key like a specific field in the header or body of the request.

However, the configuration syntax used to express these policies varies significantly between platforms. Even when implementing conceptually identical rate limiting behavior, each system defines its own policy structure and configuration format.

Table 3.1: Analyzed API Management Platforms

Company	Product	Documented Rate Limiting Mechanisms
Microsoft	Azure API Management [10]	Sliding window, token bucket
Amazon	AWS API Gateway [2]	Token bucket
Google	Apigee [6]	Sliding window, token bucket
Kong Inc.	Kong Gateway [8]	Fixed window, sliding window
Tyk Technologies	Tyk API Gateway [15]	Fixed window, sliding window, token bucket
Zuplo Inc.	Zuplo API Management [17]	Fixed window
WSO2	WSO2 API Manager [16]	Fixed window

3.1.1 Example Rate Limiting Configurations

The following examples illustrate how a rate limit of **30 requests per minute per IP address** can be expressed in several API management platforms. Although the intended behavior is identical, the configuration syntax differs significantly.

Google Apigee (SpikeArrest)

```
<SpikeArrest name="SA-Static-30pm-IP">
  <Rate>30pm</Rate>
  <Identifier ref="client.ip" />
  <UseEffectiveCount>true</UseEffectiveCount>
</SpikeArrest>
```

Azure API Management

```
<rate-limit-by-key calls="30"
                   renewal-period="60"
                   counter-key="@(context.Request.IpAddress)" />
```

Kong Gateway

```
_format_version: "3.0"
plugins:
  - name: rate-limiting-advanced
    config:
      limit:
      - 30
      window_size:
      - 60
```

Tyk API Gateway

```
apiVersion: tyk.tyk.io/v1alpha1
kind: ApiDefinition
metadata:
  name: httpbin-global-rate-limit
spec:
  name: httpbin-global-rate-limit
  use_keyless: true
  protocol: http
  active: true
  proxy:
    target_url: http://httpbin.org
    listen_path: /httpbin
    strip_listen_path: true
  global_rate_limit:
    rate: 30
    per: 60
```

Zuplo

```
{
  "name": "my-rate-limit-inbound-policy",
  "policyType": "rate-limit-inbound",
  "handler": {
    "export": "RateLimitInboundPolicy",
    "module": "$import(@zuplo/runtime)",
    "options": {
      "rateLimitBy": "ip",
      "requestsAllowed": 30,
      "timeWindowMinutes": 1
    }
  }
}
```

These differences in syntax highlight the need for standardization. A standardized, platform-independent way of expressing rate limiting policies would make it easier to migrate APIs between gateways and improve portability.

3.2 Extension Placement

Rate limiting policies may be defined at different levels within an OpenAPI document:

- Global level (applies to all operations)
- Operation level (applies to a single HTTP operation)

This is similar to the securitySchemes component that can be applied as a default to the whole API or to individual operations. When multiple rate limiting policies are defined at different levels, they are evaluated cumulatively rather than hierarchically overridden. In other words, all applicable rate limits must be satisfied for a request to be accepted.

For example, a system may define:

- A global limit of 1000 requests per minute for the entire API
- Operation-level limits of 50 requests per minute for individual HTTP methods such as `GET`, `POST`, or `PUT`.

In such a configuration, a request must satisfy all active limits simultaneously. If any limit is exceeded, the request is rejected. This layered enforcement model allows different levels of protection to coexist, combining global safeguards against large-scale abuse with stricter limits on sensitive endpoints.

The cumulative evaluation strategy mirrors the behavior of many API gateways and rate limiting systems, where multiple policies may apply concurrently to the same request.

3.3 RateLimit Object Definition

A `RateLimit` object describes the algorithm, scope, and enforcement parameters of a rate limiting policy. The object is introduced as the value of the `x-rateLimit` extension.

Table 3.2: RateLimit Object Definition

Field Name	Type	Applies To	Description
algorithm	string	Any	**REQUIRED.** The rate limiting algorithm to enforce. Valid values: fixedWindow, slidingWindow or tokenBucket.
description	string	Any	Explanation of the rate limiting policy and its intended usage.

Field Name	Type	Applies To	Description
key	Key Object	Any	Defines how requests are grouped for rate limit enforcement. If omitted, a single global counter is used for all requests. The key extracts a value from the request, such as the client IP address or a specific request parameter.
requestCount	int32	fixedWindow, slidingWindow	**REQUIRED.** Maximum number of requests allowed within the specified timeWindow. Must be greater than 0.
timeWindow	string (ISO8601)	fixedWindow, slidingWindow	**REQUIRED.** Duration of the evaluation window (e.g., PT1M, PT1H).
capacity	int32	tokenBucket	**REQUIRED.** Maximum number of requests or tokens the bucket can hold.
refillRate	Rate Object	tokenBucket	**REQUIRED.** Defines how tokens are added back into the bucket over time.

3.3.1 Key Object Definition

The `Key` object defines how requests are grouped for rate limit enforcement. The key extracts a value from the incoming request and associates all requests with the same value to a shared rate limiting counter. The design deliberately mirrors OpenAPI parameter identification (in and name) to allow rate limiting policies to reference request elements already defined in the API specification.

Table 3.3: Key Object Definition

Field Name	Type	Applies To	Description
in	string	Any	**REQUIRED.** Location from which the key value is extracted. Valid values: `IP`, `header`, `query`, `path`, `cookie`.
name	string	header, query, path, cookie	Name of the request field used as key (e.g., `api-key`, `userId`). Not required when `in = IP`.

3.3.2 Rate Object Definition

The `Rate` object defines a generic rate expression and is used by the token bucket algorithm.

Table 3.4: Rate Object Definition

Field Name	Type	Applies To	Description
tokens	int32	Any	**REQUIRED.** Number of tokens added to the token bucket per interval. Must be greater than 0.
interval	string (ISO8601)	Any	**REQUIRED.** Length of the rate interval (e.g., PT1S, PT1M).

3.4 Validation Rules

The following validation rules apply:

- The `algorithm` field is mandatory and must be one of the supported values: `fixedWindow`, `slidingWindow`, or `tokenBucket`.
- Algorithm-specific parameters must be present and mutually exclusive:
  - `requestCount` and `timeWindow` are required for `fixedWindow` and `slidingWindow`.
  - `capacity` and `refillRate` are required for `tokenBucket`.
- Numerical values such as `requestCount`, `capacity`, and `tokens` must be strictly greater than zero.
- Duration fields such as `timeWindow` and `interval` must follow the ISO8601 duration format.
- If a `key` object is defined, the `in` field is mandatory and must be one of the following values: `ip`, `header`, `query`, `path`, or `cookie`.
- When `key.in` is set to `header`, `query`, `path`, or `cookie`, the `name` field must be provided and must correspond to a valid request field defined in the API specification.
- When `key.in = ip`, the `name` field must not be present.
- The `refillRate` field must contain a valid `Rate` object where both `tokens` and `interval` are provided.

- Multiple rate limits may be defined on the same operation by specifying `x-rateLimit` as a list of `RateLimit` objects. All defined limits must be evaluated independently.

These constraints ensure that rate limiting policies can be consistently interpreted and processed by automated tools implementing the specification.

3.5 Example Usage

An example of the proposed extension at operation level:

```
paths:
  /pets:
    get:
      x-rateLimit:
        algorithm: slidingWindow
        requestCount: 15
        timeWindow: PT1M
```

An example of multiple rate limits on the same operation:

```
paths:
  /pets:
    get:
      x-rateLimit:
        - algorithm: slidingWindow
          requestCount: 60
          timeWindow: PT1M
        - algorithm: tokenBucket
          capacity: 10
          refillRate:
            tokens: 5
            interval: PT1S
```

An example of a rate limit with IP as the key:

```
paths:
  /pets:
    get:
      x-rateLimit:
        algorithm: slidingWindow
        key:
          in: ip
        requestCount: 15
        timeWindow: PT1M
```

An example of a rate limit with a header API key as the key:

```
paths:
  /pets:
    get:
      x-rateLimit:
        algorithm: slidingWindow
        key:
          in: header
          name: API-Key
        requestCount: 15
        timeWindow: PT1M
```

3.6 Limitations of the design

The proposed model focuses on the core parameters required to express commonly used rate limiting algorithms. While this allows the majority of real-world rate limiting policies to be represented, several advanced configurations used by some API gateways are intentionally not covered by the current design.

First, some platforms support more complex request accounting mechanisms such as weighted requests. In such systems, different operations may consume different numbers of tokens depending on their computational cost or resource impact. This resource impact is typically expressed through code. The proposed model assumes that each request consumes a single unit and therefore does not include a mechanism for weighted counting.

Second, some API gateways allow the definition of differentiated policies for specific clients or groups of clients. For example, premium users may be allowed a higher request quota than free-tier users. Although such behavior can typically be implemented by defining separate rate limits using different keys, the current model does not include explicit constructs for conditional or tier-based rate limiting policies.

Another limitation relates to the practical constraints of enforcement mechanisms. Algorithms such as sliding window rate limiting may require access to historical request data within a given time interval. In distributed systems, storing and retrieving this historical information efficiently can impose practical limits on the maximum time window that can be evaluated. These operational constraints are implementation-specific and are therefore not captured at the specification level.

Despite these limitations, the proposed model captures the fundamental parameters shared by modern rate limiting mechanisms and provides a structured way to represent them within the OpenAPI Specification.

Chapter 4

Implementation

This chapter describes the implementation of the proposed rate limiting extension within the OpenAPI code generation workflow. The objective of this implementation is to demonstrate that rate limiting policies defined using the `x-rateLimit` extension can be automatically translated into executable server-side logic.

Rather than implementing rate limiting manually in application code, the approach integrates the extension directly into the OpenAPI Generator[1], enabling rate limiting to be enforced as part of the generated server.

4.1 Implementation Approach

The implementation is realized as a proof of concept by extending the OpenAPI Generator. Code was added to the existing `python-flask` generator to support the `x-rateLimit` extension.

The implementation consists of three main steps:

- Parsing the `x-rateLimit` extension from the OpenAPI document,
- Transforming the extension into a generator-friendly representation,
- Injecting rate limiting logic into the generated Python server code.

This approach ensures that rate limiting policies defined during design are automatically enforced at runtime without requiring manual implementation.

4.2 Extension of OpenAPI Generator

The core of the implementation is a modification of the `PythonFlaskConnexionServerCodegen` class.

[1] https://github.com/OpenAPITools/openapi-generator

4.2.1 Supporting File Injection

First, a rate limiting module is added to the generated project by extending the list of supporting files:

```
supportingFiles.add(
    new SupportingFile("limiter.mustache",packagePath(),"limiter.py")
);
```

The additional `limiter.mustache` file contains limiter classes and functions to enforce add rate limiting functionality. This ensures that each generated server includes a reusable implementation of the rate limiting logic.

4.2.2 Processing `x-rateLimit`

The generator is extended by overriding the `fromOperation` method. This method is responsible for transforming OpenAPI operations into internal generator models.

The implementation inspects the `x-rateLimit` extension:

```
Object extObj = operation.getExtensions().get("x-rateLimit");
```

Both single objects and lists of rate limits are supported. Each rate limit is resolved and converted into a structured representation that can be used by the templates.

Resolved rate limits are stored in the operation's vendor extensions:

```
op.vendorExtensions.put("xRateLimit", rateLimits);
```

This makes the rate limiting configuration available during template rendering.

4.2.3 Reference Resolution

To support reusable rate limit definitions, references to `components/x-rateLimit` are resolved:

```
Map<String, Object> resolvedCfg =
    resolveRateLimitRef(this.openAPI, ref);
```

This allows rate limiting policies to be defined once and reused across multiple operations, improving consistency and maintainability.

4.2.4 Algorithm Mapping

Each rate limiting algorithm is mapped to a specific configuration used by the templates.

For fixed and sliding window algorithms, the configuration is converted into a format compatible with Flask-Limiter:

```
String limit = requestCount + " per " + windowFlaskLimiter;
```

Flask-Limiter expects rates written in a format such as `10 per 10 seconds` or `30 per hour`. `windowFlaskLimiter` is a string converted from an ISO 8601 duration to this Flask-Limiter-compatible format. For the token bucket algorithm, parameters such as capacity and refill rate are extracted and passed directly to the template.

4.3 Template-Based Code Generation

The generator modifications are complemented by changes to mustache templates, which inject the rate limiting logic into the generated code.

4.3.1 Controller Integration

The controller template is extended to include rate limiting decorators that are generated dynamically based on the `x-rateLimit` configuration.

The following excerpt shows the relevant part of the mustache template:

```
{{#vendorExtensions.xRateLimit}}
{{#isTokenBucket}}
@tokenbucket_limiter.limit(
    "capacity={{capacity}}&refillRequests={{refillRequests}}&
       refillInterval={{refillInterval}}"
    {{#key}},
    key_config={"in": "{{in}}"{{#name}}, "name": "{{name}}"{{/
       name}}}
    {{/key}}
)
{{/isTokenBucket}}
{{^isTokenBucket}}
@{{algorithmPrefix}}_limiter.limit(
    "{{limit}}",
    override_defaults=True
    {{#key}},
    key_config={"in": "{{in}}"{{#name}}, "name": "{{name}}"{{/
       name}}}
    {{/key}}
)
{{/isTokenBucket}}
{{/vendorExtensions.xRateLimit}}
```

This template iterates over all resolved rate limits stored in `vendorExtensions.xRateLimit`. Depending on the algorithm, it generates either a Flask-Limiter decorator or a custom token bucket decorator.

For example, the following OpenAPI definition:

```
x-rateLimit:
  algorithm: slidingWindow
  requestCount: 15
  timeWindow: PT1M
  key:
    in: header
    name: API-Key
```

is first processed by the generator into the following internal representation:

```
{
  "algorithmPrefix": "sliding",
  "limit": "15 per minute",
  "key": {
    "in": "header",
    "name": "API-Key"
  }
}
```

The mustache template then renders this into the following Python code:

```
@sliding_limiter.limit(
    "15 per minute",
    override_defaults=True,
    key_config={"in": "header", "name": "API-Key"}
)
def get_pets():
    ...
```

Similarly, a token bucket configuration such as:

```
x-rateLimit:
  algorithm: tokenBucket
  capacity: 10
  refillRate:
    tokens: 5
    interval: PT1S
```

is transformed into this Python code:

```
@tokenbucket_limiter.limit(
    "capacity=10&refillRequests=5&refillInterval=PT1S"
)
def get_pets():
    ...
```

Multiple rate limits defined on the same operation result in multiple decorators being generated and stacked. This directly reflects the cumulative evaluation model defined in Chapter 3 - Design.

4.4 Runtime Implementation

The runtime behavior is implemented in the generated `limiter.py` module.

4.4.1 Dynamic Key Resolution

The implementation supports flexible key extraction based on the `Key` object for the key types implemented in the proof of concept:

- IP address,
- headers,
- query parameters,
- path parameters.

Cookie-based keys are included in the specification-level design but are not implemented in this proof of concept. If no key is specified, a global key is used:

```
def resolve_key(key_config):
    # No key → global bucket
    if not key_config:
        return "global"

    location = key_config.get("in")

    if location == "IP":
        return request.remote_addr or "global"

    if location == "header":
        value = request.headers.get(key_config.get("name"))
        return value if value else "global"

    if location == "query":
        value = request.args.get(key_config.get("name"))
```

```
        return value if value else "global"

    if location == "path":
        value = request.view_args.get(key_config.get("name"))
        return value if value else "global"

    return "global"
```

This ensures that the grouping behavior defined in the specification is respected at runtime.

4.4.2 DynamicLimiter Wrapper

To support dynamic key configuration, a wrapper around Flask-Limiter is introduced:

```
key_func = (
    (lambda: resolve_key(key_config))
    if key_config
    else (lambda: "global")
)
```

This allows the same limiter instance to be reused with different key configurations per endpoint.

4.4.3 Token Bucket Implementation

Since Flask-Limiter does not provide a native token bucket implementation, a custom limiter is developed.

Each client is assigned a token bucket identified by:

```
(func.__name__, client_key, config_string)
```

The bucket:

- Refills tokens periodically,
- Tracks remaining capacity,
- Rejects requests when insufficient tokens are available.

This implementation ensures that all three algorithms defined in the design are supported.

4.5 Resulting Workflow

The complete workflow can be summarized as follows:

1. A rate limiting policy is defined in the OpenAPI document using `x-rateLimit`,
2. The OpenAPI Generator processes the extension and stores it in vendor extensions,
3. Mustache templates generate Python code with appropriate decorators,
4. The generated server enforces rate limiting at runtime.

This demonstrates that rate limiting can be fully integrated into the API lifecycle, from specification to execution.

4.6 Conclusion

The implementation confirms that the proposed rate limiting model can be successfully integrated into the OpenAPI Generator.

By extending both the generator and its templates, rate limiting policies defined at design time are automatically enforced in the generated server, validating the feasibility of the approach as a practical proof of concept.

Chapter 5

Evaluation

This chapter evaluates the proposed rate limiting extension and its proof-of-concept implementation. The evaluation focuses on four aspects. First, it assesses whether the proposed `x-rateLimit` model is expressive enough to represent commonly used rate limiting policies. Second, it examines whether each rate limiting policy defined in the OpenAPI document is correctly translated into the corresponding generated server-side enforcement code. Third, it validates whether the generated server-side implementation enforces the configured policies at runtime. Fourth, it considers the maintainability of the approach and discusses the trade-offs between application-level enforcement and enforcement at infrastructure level, such as through a reverse proxy, API gateway, or Web Application Firewall (WAF).

The evaluation does not attempt to measure long-term developer adoption or the reduction of rate limiting vulnerabilities in real projects. Such an evaluation would require deploying the proposed tooling with multiple development teams and observing their behavior over a longer period of time, which is outside the scope of this thesis. Instead, this chapter evaluates the more direct claim of this work: A rate limiting policy can be specified once in an OpenAPI document and then carried through the development workflow into generated server-side enforcement and specification-driven validation.

This chapter builds further on the concepts introduced in Chapter 2 - Background, the design decisions from Chapter 3 - Design, and the implementation described in Chapter 4 - Implementation.

5.1 Evaluation approach

The evaluation is performed on the proof-of-concept implementation described in Chapter 4 - Implementation. In addition to the generator modifications, a dedicated test application was developed to validate the generated rate limiting behavior. Rather than relying on ad hoc manual testing, this tool derives test inputs from the OpenAPI specification itself and executes them against a running target API.

The evaluation focuses on functional correctness and engineering practicality, not on production-scale performance. More specifically, the following evaluation

questions are considered:

- To what extent can the proposed `x-rateLimit` extension represent the rate limiting features identified in the API management platforms analyzed in Chapter 3 - Design?
- Is each `x-rateLimit` policy from the specification translated into generated enforcement logic?
- Does the generated implementation reject requests when the configured rate limit is exceeded?
- Are keyed rate limits enforced independently for different key values?
- Can stacked rate limits be tested in a systematic manner?
- Does the approach reduce duplication between specification, implementation, and testing?
- In which situations is application-level enforcement preferable to gateway- or proxy-level enforcement, and where are its limitations?

These questions correspond to four evaluation dimensions of the proposed approach. First, expressiveness can be evaluated by mapping rate limiting features from existing API management platforms to the proposed model. Second, code generation correctness can be evaluated by comparing the policies present in the OpenAPI document with the decorators and runtime configuration emitted in the generated server. Third, runtime correctness can be evaluated by sending request sequences to the generated server and observing whether successful responses and `429 Too Many Requests` responses occur in accordance with the configured policy. Fourth, maintainability can be evaluated by comparing how many artifacts must be updated when a rate limiting policy changes.

5.2 Expressiveness evaluation

The first part of the evaluation concerns the expressiveness of the proposed model. The goal is to determine whether the `x-rateLimit` extension can represent the types of policies commonly found in API management platforms. This evaluation step directly addresses RQ1, formulated in Chapter 1 - Introduction as follows:

> *Can modern rate limiting configurations be formally modeled and incorporated into the OpenAPI Specification?*

The comparison is based on the platform analysis introduced in Chapter 3 - Design. The proposed model supports the three algorithms selected in the design chapter: fixed window, sliding window, and token bucket. At the specification level,

Table 5.1: Expressiveness of the proposed `x-rateLimit` model

Policy feature	**Example**	**Supported**	**Mapping**
Fixed window limit	100 requests per minute	Yes	`algorithm`, `requestCount`, `timeWindow`
Sliding window limit	100 requests over a rolling minute	Yes	`algorithm`, `requestCount`, `timeWindow`
Token bucket limit	Capacity of 10 tokens, refill of 5 tokens per second	Yes	`algorithm`, `capacity`, `refillRate`
Global limit	One counter shared by all clients	Yes	Omit the `key` object
IP-based limit	One counter per client IP address	Yes	`key.in: ip`
Header-based limit	One counter per API key header	Yes	`key.in: header`, `key.name`
Query-based limit	One counter per query parameter value	Yes	`key.in: query`, `key.name`
Path-based limit	One counter per path parameter value	Yes	`key.in: path`, `key.name`
Cookie-based limit	One counter per cookie value	modeled, not implemented in the proof of concept	`key.in: cookie`, `key.name`
Stacked policies	Global limit combined with operation-level limit	Yes	Multiple applicable `x-rateLimit` objects
Reusable policies	Shared policy referenced by several operations	Yes	`$ref` to reusable rate limit definition
Weighted requests	Expensive operations consume more than one unit	No	Outside current model
Tier-based limits	Different quotas for free and premium users	Partially	Can be approximated through keys, but no explicit condition model
Distributed counter consistency	Shared counters across multiple server instances	No	Deployment concern outside specification-level model

it also supports several common keying strategies, including global limits, IP-based limits, and limits based on headers, query parameters, path parameters, or cookies.

The results in Table 5.1 show that the proposed extension covers the core rate limiting features that are common across API gateways and API management platforms. The main unsupported features are advanced policy mechanisms such as weighted request costs, explicit tier-based conditions, and distributed counter consistency. These limitations are acceptable for the scope of this thesis because the design focuses on a portable specification-level model rather than a full replacement for production API gateway policy engines.

5.3 Test program

To support the runtime evaluation, a separate web-based test tool was implemented. Its purpose is to automatically derive test inputs from an OpenAPI specification, execute request bursts against a target API, and summarize whether the observed behavior matches the configured policy. The backend of this tool is implemented as a Python-Flask application, while the frontend provides a browser-based interface for importing a specification, selecting endpoints, starting tests, and visualizing results.

The test tool plays an important role in the evaluation because it demonstrates that the proposed extension is not only useful as documentation, but can also be interpreted by automated tooling. The same OpenAPI document is therefore used in two different ways: first as input for server generation, and second as input for runtime validation.

5.3.1 Endpoint discovery

After an OpenAPI document is uploaded, the tool traverses the `paths` section and combines operation metadata with any applicable rate limiting definitions. A simplified excerpt is shown below:

```
def extract_oas_targets(spec: Dict[str, Any]) -> List[Dict[str, Any]]:
    results = []
    for path, path_item in (spec.get("paths") or {}).items():
        for method, operation in path_item.items():
            effective_rl = parse_rate_limit_obj(
                operation.get("x-rateLimit"), spec
            )
            results.append({
                "path": path,
                "method": method.upper(),
                "rate_limits": effective_rl,
            })
    return results
```

This makes it possible for the frontend to present the user with a list of discovered endpoints and their associated rate limiting policies. As a result, the same OpenAPI document can be used both to generate a server and to automatically prepare evaluation targets for that server.

5.3.2 Specification-driven test generation

A central property of the test tool is that it is specification-driven. After an OpenAPI document is uploaded, the backend parses the `x-rateLimit` definitions associated with paths and operations, resolves references, and extracts the relevant rate limiting parameters for each endpoint.

The extraction logic is centered around the processing of the `x-rateLimit` extension object. The following simplified excerpt illustrates this behavior:

```
def parse_rate_limit_obj(
    raw: Any,spec: Dict[str, Any]) -> List[Dict[str, Any]]:
    if raw is None:
        return []
    if isinstance(raw, dict) and "$ref" in raw:
        resolved = resolve_ref(raw["$ref"], spec)
        if resolved is None:
            return []
        return parse_rate_limit_obj(resolved, spec)

    items = raw if isinstance(raw, list) else [raw]
    parsed = []

    for item in items:
        algorithm = item.get("algorithm")
        request_count = item.get("requestCount")
        burst_capacity = item.get("capacity")
        time_window = item.get("timeWindow")

        parsed.append({
            "algorithm": algorithm,
            "requestCount": request_count,
            "burstCapacity": burst_capacity,
            "timeWindow": time_window,
        })

    return parsed
```

This means that the evaluation input is derived directly from the same specification that also drives server generation. The specification is therefore not merely a description of the API, but also the source from which validation behavior is constructed.

5.3.3 Test phases

The tool evaluates endpoints using two main request patterns:

- a burst phase, intended to trigger immediate rate limiting behavior or burst-capacity exhaustion.
- a sustained phase, intended to observe how the endpoint behaves over a longer interval and to test refill or rolling-window behavior.

For stacked rate limits, the tool plans separate test phases per layer. It first identifies the strictest applicable limit and then progressively probes the other layers. This is necessary because when multiple policies apply simultaneously, the strictest one may mask the behavior of the others. For example, if an endpoint has both a limit of 5 requests per minute and a limit of 100 requests per hour, the minute-level policy will be triggered first during a burst test. The outer policy can only be meaningfully evaluated with a test plan that takes the stricter inner policy into account.

The planning logic is implemented explicitly in the backend. In simplified form, it behaves as follows:

```
def plan_stacked_tests(
    rate_limits: List[Dict[str, Any]]) -> List[Dict[str, Any]]:
    if not rate_limits:
        return []

    layers = sorted(rate_limits, key=lambda r: (
        r.get("burstCapacity") or r.get("requestCount") or 1
    ))

    phases = []
    for layer in layers:
        cap = layer.get("burstCapacity") or layer.get("requestCount")
        phases.append({
            "layer": layer,
            "drain_steps":[{"batch_size": cap + 2,"delay_secs":0.0}],
        })
    return phases
```

Although the real implementation is more elaborate, the principle is the same: the test program computes a concrete execution plan from the specification rather than relying on hard-coded endpoint-specific test scripts. This is important because it keeps the evaluation aligned with the contract-first approach of the thesis.

5.3.4 Support for keyed limits

The test tool also supports keyed rate limits. When a rate limit is based on a request header or query parameter, the frontend allows two distinct key values to be configured. The backend can then execute the same request pattern independently for both keys. This makes it possible to verify whether limits are applied per key rather than globally across all traffic.

The following simplified code fragment illustrates how the backend injects the selected key into the outgoing request:

```
def _build_request_headers_params(
    config: Dict[str, Any], key_value: Optional[str]):
    headers = dict(config.get("headers") or {})
    params = dict(config.get("query_params") or {})

    key_in = config.get("rl_key_in") or ""
    key_name = config.get("rl_key_name") or ""

    if key_in == "header" and key_name and key_value:
        headers[key_name] = key_value
    elif key_in == "query" and key_name and key_value:
        params[key_name] = key_value

    return headers, params
```

This mechanism is particularly useful for validating limits based on API keys or similar client identifiers. It also allows the evaluation to confirm whether the grouping semantics defined in Chapter 3 - Design are correctly respected by the generated runtime implementation.

5.3.5 Assessment output

After each test run, the tool reports the number of successful requests, the number of `429 Too Many Requests` responses, and basic timing information. The results can be inspected in the frontend or exported as JSON or CSV. This makes the test runs easier to repeat, compare, and verify than manual testing.

5.4 Translation to generated enforcement code

This part of the evaluation checks whether each rate limiting policy defined in the OpenAPI document is correctly translated into the corresponding generated server-side enforcement code. This is important because the proposed extension is only useful if the generator actually uses the `x-rateLimit` information when producing the server implementation.

This was evaluated by comparing the original `x-rateLimit` definition with the generated Python code. For each policy, the generated code should contain the same algorithm, threshold, time window, keying strategy, and operation placement.

Table 5.2: Code generation checks for `x-rateLimit` policies

OpenAPI policy feature	**Expected generated code**	**Pass/Fail**
Fixed or sliding window policy	A limiter decorator with the configured request count and time window	Pass
Token bucket policy	A token bucket decorator with the configured capacity and refill rate	Pass
Header, query, path, or IP-based key	A key configuration that extracts the correct value from the request	Pass
Multiple policies on one operation	Multiple decorators on the same generated operation	Pass
Referenced policy using `$ref`	Generated code based on the resolved referenced policy	Pass

A policy is considered correctly translated when the generated enforcement code reflects the same meaning as the original OpenAPI definition. For example, a sliding window policy of 15 requests per minute using the `API-Key` header should result in a sliding-window limiter configured for 15 requests per minute and keyed by that header.

5.5 Functional evaluation of generated enforcement

This part of the evaluation validates the runtime behavior of the generated server. The objective is to determine whether endpoints protected by generated rate limiting logic accept requests within the configured limit and reject requests that exceed it.

For each tested endpoint, the test tool sends a sequence of requests and records the number of successful responses and `429 Too Many Requests` responses. Because timing effects can influence the exact request at which a limit is triggered, the evaluation allows a small tolerance margin rather than treating the first rejected request as an exact mathematical boundary.

The functional test cases listed in Table 5.3 cover the main behaviors required by the proposed model. The evaluated scenarios include the three supported rate-limiting algorithms, keyed limits, stacked limits, and referenced policies.

5.5.1 Key isolation

Key isolation verifies that requests are grouped according to the configured key. For example, when a limit is based on an `API-Key` header, requests with `API-Key: client-a` and `API-Key: client-b` should be counted separately.

Table 5.3: Functional test cases for generated enforcement

Test	Policy type	Expected behavior	Pass/Fail
T1	Fixed window	Requests beyond the configured threshold return `429` until the window resets	Pass
T2	Sliding window	Requests beyond the rolling-window threshold return `429`; later requests are accepted again after enough time has passed	Pass
T3	Token bucket	A burst is accepted up to the bucket capacity; additional requests return `429` until tokens are refilled	Pass
T4	Keyed limit	Requests with different key values are counted independently	Pass
T5	Stacked limits	A request is rejected when any applicable policy is exceeded	Pass
T6	Referenced policy	A policy referenced through `$ref` is enforced like an inline policy	Pass

The test tool checks this by first sending enough requests with one key value to reach the limit. It then sends a request with a second key value. If the second request is accepted, the counters are isolated correctly. If it is rejected immediately, the implementation is probably using a shared counter instead of a keyed counter.

This validates that rate limiting is not only applied, but applied to the correct request group.

5.5.2 Stacked limits

Stacked limits occur when more than one rate limiting policy applies to the same operation. For example, an API may define a global limit for all traffic and a stricter operation-level limit for a sensitive endpoint. In the proposed model, all applicable limits are evaluated cumulatively: a request must satisfy every active policy.

The test tool evaluates stacked limits by identifying the strictest applicable policy and triggering it first. This is necessary because the strictest policy may hide the behavior of less restrictive policies during simple burst testing. The generated code is also inspected to verify that all configured policies are present.

A stacked-limit test is considered successful when all applicable policies are generated, the endpoint rejects requests when any policy is exceeded, and the limits are applied cumulatively rather than as alternatives or overrides.

5.5.3 Black-box testing limitations and ambiguity

The evaluation tool primarily performs black-box testing. It observes externally visible behavior, such as successful responses, `429 Too Many Requests` responses,

and the request number at which limiting occurs. This is useful for validating whether enforcement exists and whether it broadly matches the configured policy, but it also introduces uncertainty when interpreting the exact internal algorithm.

In particular, distinguishing between sliding window and token bucket behavior is not always straightforward from external observations alone. Both algorithms may permit a short burst of requests and then recover gradually over time. When only request timing and status codes are observed, these patterns can appear similar, especially for small time windows or low request counts. As a result, the test tool can provide a strong indication of the implemented behavior, but not always absolute certainty about the exact internal mechanism.

This ambiguity is further influenced by timing-related effects in the execution environment. Network latency may delay requests or responses. Thread scheduling can affect the exact order and spacing of concurrent requests. In addition, the timing granularity of the operating system, Python runtime, and HTTP stack may cause small deviations between the intended test schedule and the actual arrival time of requests at the application. These factors can slightly shift the exact request at which the first `429` response appears.

For this reason, the evaluation should not interpret the first rejected request as an exact mathematical boundary in every case. Instead, the results should be understood as an approximation of the configured behavior within a realistic tolerance margin. This is sufficient for validating whether a generated implementation behaves consistently with the policy defined in the OpenAPI document, but it also shows that black-box testing alone has limits when used to infer detailed algorithmic behavior.

These limitations do not invalidate the evaluation. Rather, they clarify its scope. The objective of the test tool is to verify practical enforcement behavior, not to formally prove the internal correctness of the limiter implementation. For that reason, black-box testing is best understood as a pragmatic validation method that complements, rather than replaces, inspection of the generated code and runtime logic.

5.6 Maintainability

A major motivation for integrating rate limiting into the OpenAPI workflow is maintainability. In traditional setups, rate limiting rules are often duplicated across documentation, gateway configuration, application code, and test scripts. This creates a risk of drift: the documented policy may no longer match the enforced policy, or tests may validate outdated assumptions.

The proposed approach improves this situation by making the OpenAPI document the primary source of truth. The same `x-rateLimit` definition is used as input for code generation and for test derivation. This reduces duplication and makes policy changes easier to propagate. A modified request threshold, time window, algorithm, or key definition only needs to be changed once in the specification.

After regeneration, the server code and specification-driven tests reflect the updated policy automatically.

This maintainability benefit can be evaluated by comparing the number of places that must be updated when a rate limiting policy changes. In a manual approach, a policy change may require updates to documentation, application code, gateway configuration, and test scripts. In the proposed approach, the policy is changed in the OpenAPI document and the dependent artifacts are regenerated or re-derived.

Table 5.4: Maintainability comparison between manual and specification-driven rate limiting

Change scenario	**Manual approach**	**Specification-driven approach**
Change request threshold	Update documentation, enforcement code or gateway policy, and tests	Update `x-rateLimit` in the OpenAPI document and regenerate
Change time window	Update documentation, enforcement code or gateway policy, and tests	Update `timeWindow` and regenerate
Change keying strategy	Update documentation, request parsing logic, limiter configuration, and tests	Update the `key` object and regenerate
Add the same policy to multiple operations	Duplicate or manually reuse configuration across several locations	Reuse a referenced policy definition
Validate policy behavior	Maintain separate endpoint-specific test scripts	Derive test inputs from the same OpenAPI document

The comparison in Table 5.4 shows that the proposed approach reduces duplication between specification, implementation, and testing. The main maintenance burden shifts from repetitive manual configuration in each API to centralized tooling maintenance.

However, this benefit is not free. The generator no longer only translates OpenAPI operations into server stubs, but also interprets rate limiting metadata and emits corresponding enforcement code. In this proof of concept, this required custom parsing logic in Java, additional Mustache templates, and supporting Python runtime code in the generated project. Future updates to the upstream OpenAPI Generator may therefore require these changes to be rebased, reviewed, or adapted.

This means that the approach is most maintainable when the generator is reused across multiple APIs or projects. In that case, the tooling maintenance cost is centralized, while individual API specifications remain simpler and more consistent. If the approach were used for only a single small API, the additional generator complexity may be harder to justify compared with a manual implementation.

Overall, the maintainability result is positive. The approach reduces policy duplication and improves consistency across specification, implementation, and testing. The main maintenance burden shifts from repeated manual configuration to centralized tooling maintenance, which is generally a favorable trade-off in contract-first development.

5.7 Application-level rate limiting versus WAF or proxy enforcement

An important question is whether rate limiting should be implemented in generated application code at all, given that many production systems already enforce similar policies at the level of a reverse proxy, API gateway, or WAF.

5.7.1 Advantages of application-level enforcement

Application-level enforcement offers several benefits in the context of this thesis.

First, it provides direct alignment with the API contract. Since the generated code is derived from the OpenAPI document, the enforcement logic stays close to the documented API behavior. This is especially useful in development, prototyping and environments where no dedicated gateway is present.

Second, it allows fine-grained access to request context. The application can inspect operation-specific metadata, route parameters, headers, and other semantic information already known to the framework. This makes it straightforward to support key extraction from headers, query parameters, or path values, as required by the proposed model.

Third, it lowers the barrier to adoption. A developer who already uses OpenAPI-based server generation can obtain a working proof of concept without also having to provision external infrastructure such as Kong, Apigee, or Azure API Management.

5.7.2 Advantages of gateway, proxy, or WAF enforcement

Despite these advantages, infrastructure-level enforcement remains highly attractive in real-world deployments.

A reverse proxy, API gateway, or WAF sits in front of the application and can reject excessive traffic before requests reach the backend. This is particularly important for protecting compute resources and absorbing abusive traffic early. In contrast, application-level enforcement still requires the request to be received and partially processed by the application stack.

Infrastructure components are also often better suited for distributed deployments. They may provide centralized counters, shared storage, or highly optimized implementations for enforcing limits consistently across multiple application instances. By comparison, an application-level proof of concept may be sufficient for single-node or demonstration scenarios, but becomes harder to scale reliably in multi-instance environments.

In addition, gateways and WAFs commonly provide richer operational features such as dashboards, central policy management, tenant-specific plans, and integrated DDoS mitigation. These features fall outside the scope of the present implementation.

5.7.3 Trade-off discussion

The trade-off is therefore not best understood as a choice of one mechanism replacing the other. Rather, the proposed extension is best seen as a standardized policy description that can feed multiple enforcement targets.

For development and generated server stubs, application-level enforcement is highly valuable because it keeps specification and behavior aligned and enables immediate validation. For production-scale deployments, gateway- or proxy-level enforcement will often remain preferable for performance, resilience, and centralized operations.

From this perspective, the most important contribution of the thesis is not that rate limiting must be enforced inside application code, but that the policy should be represented once in a machine-readable contract. Application-level generation merely demonstrates feasibility. Gateway or WAF integration would be a logical next step for broader industrial adoption.

5.8 Limitations of the evaluation

The evaluation demonstrates that rate limiting policies defined in the OpenAPI specification can be translated into generated enforcement logic and that the resulting implementation behaves consistently with the configured policies. However, several limitations should be considered when interpreting the results.

As discussed in Section 5.5.3, the evaluation relies primarily on black-box testing. While this approach is effective for validating observable enforcement behavior, it cannot provide absolute certainty about the exact internal implementation of a rate limiting algorithm. Timing effects and similarities between different algorithms may also introduce some ambiguity when interpreting results.

In addition, the evaluation was performed on a proof-of-concept implementation based on the Python Flask generator of OpenAPI Generator. The results therefore demonstrate feasibility for this specific toolchain, but do not necessarily imply identical implementation effort or behavior for other generators, frameworks, or programming languages.

These limitations do not affect the main objective of the evaluation, which is to determine whether rate limiting policies can be specified in OpenAPI, translated into generated enforcement logic, and validated through specification-driven testing.

5.9 Conclusion

The evaluation shows that the proposed OpenAPI extension can be used not only to document rate limiting policies, but also to drive automated enforcement and automated validation. The expressiveness evaluation shows that the proposed model covers the core policy features found in common rate limiting systems, including fixed window, sliding window, token bucket, keyed limits, stacked limits, and reusable policies. The main unsupported features are advanced production concerns such as weighted request costs, explicit tier-based conditions, and distributed counter consistency.

The developed test program reads the specification, derives endpoint-specific test plans, executes burst and sustained request sequences, and reports whether the generated server behaves in accordance with the configured policy. This demonstrates that a policy defined once in the API contract can be carried through the workflow from specification, to code generation, to runtime enforcement, and finally to validation.

From a maintainability perspective, the approach improves consistency by reducing duplication between specification, implementation, and testing, although it introduces some maintenance effort in the generator and runtime tooling itself. Finally, the comparison with WAF- and proxy-based enforcement shows that application-level enforcement is useful as a proof of concept and in contract-first workflows, while infrastructure-level enforcement remains important for large-scale and production-grade deployments.

Taken together, these findings indicate that specification-level modeling of rate limits is practically valuable, and that generated enforcement is a viable first implementation target.

Chapter 6

Conclusion

This thesis investigated whether rate limiting policies can be formally described in the OpenAPI Specification and whether such descriptions can be used to generate server-side enforcement logic. The motivation for this work is the growing importance of APIs in modern software systems and the corresponding need to protect them against abuse, resource exhaustion, and denial-of-service scenarios. Although OpenAPI is widely used for documenting and generating APIs, it does not currently provide a standard mechanism for describing rate limiting or similar resource consumption constraints.

To address this gap, this thesis proposed an `x-rateLimit` extension for OpenAPI. The extension provides a structured way to define rate limiting policies at both global and operation level. It supports commonly used algorithms, namely fixed window, sliding window, and token bucket rate limiting. It also supports different rate limiting keys, such as IP address, headers, query parameters, path parameters, and cookies. By expressing these policies directly in the OpenAPI document, the API specification can act as a single source of truth for both API behavior and security-related traffic constraints.

The first research question was:

> Can modern rate limiting configurations be formally modeled and incorporated into the OpenAPI Specification?

The design presented in this thesis shows that this is possible for a relevant subset of modern rate limiting configurations. The analysis of existing API management platforms showed that, although their configuration formats differ, many of them rely on a small number of shared concepts: a rate limiting algorithm, a request threshold or capacity, a time interval, and a key used to group requests. These concepts were captured in the proposed `x-rateLimit` object. The resulting model is expressive enough to describe the core rate limiting policies commonly used in API gateways and application frameworks, while remaining compatible with existing OpenAPI documents through the use of vendor extensions.

The second research question was:

> Can the proposed OpenAPI Specification extension be leveraged to generate server-side implementations that enforce the specified rate limiting policies?

The proof-of-concept implementation confirms that this is feasible. By extending the Python Flask generator of OpenAPI Generator, rate limiting definitions in the OpenAPI document were parsed, transformed into generator-friendly metadata, and injected into the generated server code through Mustache templates. The generated implementation uses Flask-Limiter for fixed and sliding window policies and a custom token bucket implementation for token bucket policies. This demonstrates that rate limiting policies can be carried from specification to generated runtime enforcement without requiring manual duplication in application code.

The evaluation further showed that the same specification can also be used to derive tests. A separate specification-driven test tool was developed to parse `x-rateLimit` definitions, discover endpoints, generate request patterns, and verify whether the generated server responds with behavior consistent with the configured policy. This completes the workflow from specification through code generation and runtime enforcement to automated validation.

The main contribution of this thesis is therefore not only the proposed extension itself, but also the demonstration that OpenAPI can be used as a practical source of truth for security-relevant server-side behavior. Rate limiting is treated as part of the API contract rather than as a separate configuration concern. This reduces the risk of inconsistencies between documentation, implementation, and tests.

At the same time, this work has limitations. The proposed model focuses on the most common rate limiting algorithms and does not cover all advanced features offered by production API gateways, such as weighted requests, tier-based policies, distributed counter synchronization, or complex conditional policies. The implementation is also limited to one generator and one target runtime, namely the Python Flask generator. As a result, the proof of concept demonstrates feasibility, but does not yet prove that the approach can be adopted with equal effort across all OpenAPI tooling ecosystems.

The evaluation focuses on functional correctness and feasibility rather than performance characteristics. In production environments, rate limiting is often enforced at the level of an API gateway, reverse proxy, or Web Application Firewall (WAF). These components are better suited for rejecting abusive traffic before it reaches the application and for coordinating limits across distributed deployments. Therefore, application-level enforcement should not be seen as a replacement for infrastructure-level enforcement in all cases. Instead, it is best understood as one possible enforcement target for a standardized policy description.

Several directions for future work follow from these limitations. First, the proposed extension could be implemented for additional OpenAPI generators and programming languages to evaluate how portable the approach is. Second, API gateway integrations could be developed so that the same `x-rateLimit` definitions can generate gateway or proxy configuration. This would make the extension more useful for production deployments. Third, the model could be extended with advanced

features such as semantics for weighted requests, client tiers and dynamic ratelimiting based on system health. Finally, broader industry feedback would be valuable to determine whether rate limiting should remain a vendor extension or evolve into a more standardized OpenAPI feature.

In conclusion, this thesis shows that rate limiting can be modeled at the OpenAPI specification level and that such a model can be used to generate working server-side enforcement. The proposed approach improves consistency, supports contract-first API security, and demonstrates how OpenAPI can be extended beyond documentation and code generation toward more secure-by-design API development workflows.

Bibliography

[1] Akamai Technologies. Lurking in the shadows: Attack trends. https://www.akamai.com/resources/state-of-the-internet/lurking-in-the-shadows, 2023. State of the Internet / Security Report.

[2] Amazon Web Services. Throttle requests to your rest apis for better throughput in api gateway. https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-request-throttling.html, 2026. Accessed: 2026-05-10.

[3] V. Atlidakis, P. Godefroid, and M. Polishchuk. Checking security properties of cloud service rest apis. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 387–397. IEEE, 2020.

[4] A. V. Barabanov, D. O. Dergunov, D. N. Makrushin, and A. N. Teplov. Automatic detection of access control vulnerabilities via api specification processing. *Voprosy Kiberbezopasnosti*, (1(47)):49–65, 2022.

[5] C. Cheh and B. Chen. Analyzing openapi specifications for security design issues. In *IEEE Secure Development Conference (SecDev)*. IEEE, 2021.

[6] Google Cloud. Apigee spikearrest policy. https://docs.cloud.google.com/apigee/docs/api-platform/reference/policies/spike-arrest-policy, 2026. Accessed: 2026-05-10.

[7] T. Kalyanasundaram, K. Panchalingam, T. Jegatheesan, A. Wijayasiri, and S. Perera. A survey of api rate limiting methods. In *2026 IEEE International Research Conference on Smart Computing and Systems Engineering (SCSE)*. IEEE, 2026.

[8] Kong Inc. Gateway rate limiting. https://developer.konghq.com/gateway/rate-limiting/, 2026. Accessed: 2026-05-10.

[9] A. Mazidi, D. Corradini, and M. Ghafari. Mining rest apis for potential mass assignment vulnerabilities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, Salerno, Italy, 2024. ACM. Preprint version available on arXiv.

[10] Microsoft. Azure api management policy reference: rate-limit-by-key. https://learn.microsoft.com/en-us/azure/api-management/rate-limit-by-key-policy, 2026. Accessed: 2026-05-10.

[11] MITRE Corporation. CWE-1430: Weaknesses in the 2024 CWE Top 25 Most Dangerous Software Weaknesses. https://cwe.mitre.org/data/definitions/1430.html, 2024. Accessed: 2026-02-28.

[12] OWASP Foundation. Owasp api security top 10 – 2023. https://owasp.org/API-Security/editions/2023/en/0x11-t10/, 2023. Accessed: 2026-02-03.

[13] S. Serbout, A. E. Malki, C. Pautasso, and U. Zdun. Api rate limit adoption – a pattern collection. In *Proceedings of the 28th European Conference on Pattern Languages of Programs (EuroPLoP 2023)*, Irsee, Germany, 2023. ACM.

[14] T. Taya, M. Hanada, Y. Murakami, A. Waseda, Y. Ishida, T. Mimura, M. W. Kim, and E. Nunohiro. An automated vulnerability assessment approach for webapi that considers requests and responses. In *International Conference on Advanced Communications Technology (ICACT)*, pages 423–425. IEEE, 2022.

[15] Tyk Technologies. Rate limiting. https://tyk.io/docs/api-management/rate-limit, 2026. Accessed: 2026-05-10.

[16] WSO2. Advanced throttling policies. https://cloud.docs.wso2.com/en/latest/learn/control-api-traffic/advanced-throttling-policies/, 2026. Accessed: 2026-05-10.

[17] Zuplo. Rate limiting. https://zuplo.com/docs/concepts/rate-limiting, 2026. Accessed: 2026-05-10.

Code of conduct and transparency statement on the use of GenAI for KU Leuven-students (academic year 2025–2026)

Generative AI (GenAI) assistance tools can be used to generate various types of content, including text, images, code, video, music, or combinations thereof. Common examples of such tools include ChatGPT, Google Gemini, Microsoft Copilot, Midjourney, Claude.ai, Perplexity.ai, and DALL · E, among others.

This code of conduct is a tool that helps students to be transparent about the use of GenAI and fits within the university's principles on academic integrity.

Important guidelines and remarks

- **Sensitive or personal data:** Some GenAI tools protect your input and sensitive or personal data better than others. There is often no transparency on what the owners of the AI applications do with the data entered. Therefore, **do not enter sensitive or personal data in free GenAI tools.** More info about what sensitive and personal data are, is described in the three confidentiality levels of the KU Leuven data classification model (non-confidential, confidential, strictly confidential). For confidential data you should preferably use **M365 Copilot Chat with your KU Leuven account**. If you use another GenAI tool, you must be **absolutely certain** it does **not store or reuse the data you enter. Try to avoid entering these data. Do so only if strictly necessary, and only to the extent required. For personal data, work with anonymous or pseudonymized data.** Additionally, in case of strictly confidential data this should first be discussed with the teaching staff of the course or your thesis supervisor.
- **Copyrighted materials:** For **lawfully obtained copyrighted material** you should preferably use M**365 Copilot Chat with your KU Leuven**

account. If you use another GenAI tool, you must be **absolutely certain** it does n**ot store or reuse the data you enter.**

- GenAI assistance may not be used for data or topics covered by a Non-Disclosure Agreement (NDA). Even in the absence of an NDA, certain information may still need to be treated as confidential—for example, due to regulatory requirements or the risk of significant harm to the university if disclosed. In case of doubt, check with your teaching staff or supervisor.
- If your master's thesis is under embargo, you should first discuss with your supervisor whether the use of GenAI is permitted.
- Before using a GenAI tool, always consider whether its use is responsible, including from a sustainability perspective (e.g. when using GenAI as a search engine, language assistant,...).
- **Take a scientific and critical attitude** when interacting with GenAI assistance and interpreting its output, that may not always be correct.
- As a student you are responsible for complying with Article 84 of the Regulations on Education and Examinations: your report or thesis should reflect your own knowledge, understanding and skills. Be aware that plagiarism rules also apply to (work that is the result of) the use of GenAI assistance tools.
  Exam Regulations Article 84: *"Every conduct individual students display with which they (partially) inhibit or attempt to inhibit a correct judgement of their own knowledge, understanding and/or skills or those of other students, is considered an irregularity which may result in a suitable penalty. A special type of irregularity is plagiarism, i.e. copying the work (ideas, texts, structures, designs, images, plans, codes , ...) of others or prior personal work in an exact or slightly modified way without adequately acknowledging the sources. Every possession of prohibited resources during an examination (see article 65) is considered an irregularity."*
- In order to maintain academic integrity and avoid plagiarism, **more information about being transparent on the use of GenAI assistance and about correctly citing and referencing GenAI** can be found on this website for students (Dutch/English).
- **Additional reading: KU Leuven guidelines on responsible use of Generative AI tools, and other information** (Dutch/English)

A few final words

If you are uncertain whether or not you should declare your use of GenAI tools, we suggest that you discuss this with your instructor or supervisor. It is always safer to declare GenAI use, even when it is not strictly required. However, declaring GenAI use does not entail that its use is

allowed; the right column in the table below provides more detailed instructions in this regard (code of conduct).

Moreover, advanced AI tools are evolving rapidly, and their capabilities have expanded significantly in a short period of time. As a result, we do not yet have all the answers about their responsible use. Finally, it is important to follow-up on the most recent evolutions in AI technologies, to have an open mind but also to be a bit cautious, to communicate with instructors, teaching assistants, supervisors and peers, to be as transparent as we can, and to learn together as we move along.

Student name: Stef Van Poucke .... **Student number: r1038576** ....

Please indicate with "X" whether it relates to a course assignment or to the Bachelor's or Master's thesis:

O This form is related to a **course assignment.**
Course name: ..... **Course number:** .....

This form is related to my O **Bachelor's** or X **Master's thesis.**
Title Bachelor's or Master's thesis: Using the OpenAPI specification for REST API security **Supervisor:** Wouter Joosen

Daily supervisor: Pieter Philippaerts

Please indicate with "X":

O I **did not use** any GenAI assistance tool.

X I **did use** GenAI Assistance. In this case **specify which ones** (e.g. ChatGPT, M365 Copilot, ...): ChatGPT and Claude

GenAI assistance used as/for:	**Name of the GenAI tool(s) used** If helpful, also describe in which way you were using GenAI related to what is specified as code of conduct.	**Code of conduct:** **For each of the categories below, always take into account the important guidelines and remarks mentioned above (e.g. copyrighted data, sensitive or personal data,. . .).**
As a language assistant for reviewing or improving texts I wrote myself	*ChatGPT as grammar, spelling, and consistency checker for texts I wrote myself, including consistency in the use of American vs British English.*	This use is similar to using spelling and grammar check tools. In general, you do not have to refer to such kind of GenAI use in the text. **However, be careful:** • When using GenAI tools on texts you did not write yourself to improve the text, **you have to refer to the original source or author**, otherwise you are committing plagiarism and thus an irregularity.
As a paraphrasing tool	*None*	This use is allowed except when it is prohibited by the teacher or the program of study. You may paraphrase your own text or texts by an author other than yourself and take inspiration from what a GenAI tool or another tool suggests (unless it is not allowed). In general, you do not have to refer to such kind of GenAI use or other paraphrasing tools in the text. **However, be careful**: • If it entails text by an author other than yourself, **you are not allowed to include that paraphrased text without reference to the original source or author**. Without such reference, you would be committing plagiarism and thus an irregularity.

For translation aid to improve texts I wrote myself or to better understand text from others	*None*	This use is allowed except when it is prohibited by the teacher or the program of study. It is similar to using translation tools (Google translate, DeepL, ...). In general, you do not have to refer to such kind of GenAI use in the text. **However, be careful:** • **You are not allowed to include that translated text without reference to the original source or author.** Without such reference, you would be committing plagiarism and thus an irregularity. • Always check the translated text for correctness and meaning.
As a search engine to get information on a topic or to search for existing research on the topic	*None*	This use is similar to e.g. a Google search or checking Wikipedia. If you write your own text based on this information, you do not have to refer to the use of GenAI in the text. **You only have to refer to the existing research and references you have checked and used** (without such references you would be committing plagiarism and thus committing an irregularity). **However, be careful:** • Be aware that the output of the GenAI tool cannot be guaranteed as a 100% reliable source of information. The output may not be entirely correct and/or be limited due to the databases it uses. Moreover, knowledge evolves and may change over time; therefore, the database of the GenAI tool may not be up to date. Therefore, **verify the information and do not just copy-paste it** as you should understand and critically process everything you are writing.

For literature search	*ChatGPT was used to suggest possibly related papers after my own regular literature searches. All suggested references were manually verified, and only sources I checked and used were cited*	This use is comparable to e.g. a Google Scholar search. You do not have to refer to such kind of GenAI use; **you only have to refer to the literature references you have checked and used** (without such references you would be committing plagiarism and thus committing an irregularity). **However, be careful:** • Be aware that the search output is restricted to the database the GenAI tool is built on. After this initial search, look for scientific sources and conduct your own analysis of the source documents. **Interpret, analyse and process the information you obtained; verify it and do not just copy-paste it** as you should understand and critically process everything you are writing. • Be aware that some GenAI tools may **output no or wrong references.** As a student you are responsible for further checking and verifying the absence or correctness of references; do not just copy-paste it.

For generating programming code	*ChatGPT and Claude were used as coding assistants for the proof-of-concept implementations in this thesis and to help understand existing code found online. ChatGPT was also used to help write and correct LaTeX code for formatting the thesis document, including tables, spacing, line wrapping, and layout. Generated suggestions were reviewed, tested, and adapted by me. No thesis content was generated through this use.*	Use of GenAI for coding is allowed except when it is prohibited by the teacher or the program of study. If used for coding, correctly mention the use of GenAI assistance in accordance with the instructions on the page on being transparant about the use of GenAI.

For generating new (research) ideas	*None*	This use of GenAI is allowed except when it is prohibited by the teacher or the program of study. Correctly mention the use of GenAI assistance in accordance with the instructions on the page on being transparent about the use of GenAI. **Be careful:** • Further verify in this case whether the idea is novel or not. It is likely that it is related to existing work. **If so, that existing work should be correctly referenced in the text** (without such reference you would be committing plagiarism and thus committing an irregularity).
For generating synthetic data	*None*	Use of GenAI for generating synthetic data is allowed, **provided that it is methodologically and ethically justifiable**, except when it is prohibited by the teacher or the program of study. Always correctly mention the use of GenAI assistance in accordance with the instructions on the page on being transparent about the use of GenAI. **Be careful:** • Always **carefully evaluate** the generated synthetic data for quality and possible bias since the output is highly dependent on the quality of the data on which the models are trained.

For generating blocks of text (other than the allowed use without referencing mentioned above)	*None*	According to Article 84 of the Regulations on Education and Examinations your text should allow to correctly and properly assess your own knowledge, understanding and skills. **Therefore, inserting blocks of text without quotes and a reference to GenAI assistance in your work is not allowed.** **Be careful:** • If it is really needed to insert a block of text from a GenAI tool, for instance because of the nature of your assignment, mention it as a citation by using quotes and correctly mention the use of GenAI assistance in accordance with the instructions page on being transparent about the use of GenAI. • However, in general, such GenAI use should be kept to an **absolute minimum**; you should always check the original sources.
For generating visuals, video or audio	*None*	Use of GenAI for generating visuals, audio or video is allowed except when it is prohibited by the teacher or the program of study. **Be careful:** • If used, refer to GenAI for visuals according to the usual referencing style, following the instructions on the page on referencing GenAI. • If you work with existing visuals, audio or video, **that existing work should be correctly referenced in the text** (without such reference you would be committing plagiarism and thus an irregularity). • Explain the usage in the methods section (if there is one) and optionally attach (or link to) the prompts with the full output (or history).

Other use (specify here; this may also include a combination of types of use mentioned above):	*None*	To make sure other use of GenAI is allowed within the course or thesis, and if so, the conditions that may apply, contact the teaching staff of the course or the supervisor of the thesis **beforehand** and explain the intended GenAI use. Also inform the programme director. Motivate how you would comply with Article 84 of the exam regulations. Explain the use and the added value of the AI tool you consider to use and how it is in accordance with the assignment or thesis and the KU Leuven guidelines on responsible use of Generative AI tools. Depending on the kind of GenAI use, it may be needed to properly reference it in the text, in accordance with the instructions page on being transparent about the use of GenAI.