AGE: Automatic Performance Evaluation of API Gateways

Pedro Moreira†
INESC TEC &
University of Minho
pedro.m.moreira@inesctec.pt

António Ribeiro
Department of Informatics
University of Minho
anr@di.uminho.pt

João Marco Silva⋆
University of Minho &
INESC TEC
joaomarco@di.uminho.pt

***Abstract*—The increasing use of microservices architectures has been accompanied by the profusion of tools for their design and operation. One relevant tool is API Gateways, which work as a proxy for microservices, hiding their internal APIs, providing load balancing, and multiple encoding support. Particularly in cloud environments, where the inherent flexibility allows on-demand resource deployment, API Gateways play a key role in seeking quality of service. Although multiple solutions are currently available, a comparative performance assessment under real workloads to support selecting the more suitable one for a specific service is time-consuming. In this way, the present work introduces AGE, a service capable of automatically deploying multiple API Gateways scenarios and providing a simple comparative performance indicator for a defined workload and infrastructure. The designed proof of concept shows that AGE can speed up API Gateway deployment and testing in multiple environments.**

***Index Terms*—API Gateways, Automatic testing, Load testing, Performance assessment**

I. Introduction

For decades, the standard way to develop an application in software engineering was through monolithic architectures. Despite the numerous monolithic approaches, the most common pattern of this architecture is a system with all of the code deployed as a single process [1], [2]. This highly coupled architecture limits not only the application's scalability but also its maintenance and may lead to inefficient resources use.

More recently, microservices architecture rose as a popular and robust strategy to enhance systems scalability and code maintainability. It consists of a Service-Oriented Architecture (SOA) type in which applications are composed of independent components (i.e., services) running as isolated processes that typically communicate through the network. Therefore, scaling up (or down) a system involves only the required component, not the whole monolithic application. Moreover, each service is technology agnostic, allowing the adoption of the most suitable solution independently [1]. However, it introduces complexity in managing multiple APIs, extends attack surfaces and affects overall performance, as the network latency is higher than function calls within a monolithic architecture.

Dealing with these challenges involves resorting to API Gateways, a reverse proxy to microservices that acts as a single-entry point into the system, decoupling consumers from the backend services, their internal structure and location [3]–[7]. By aggregating requests and responses, they can also provide load-balancing mechanisms, service throttling, circuit breaker, multiple encoding and protocol support, authentication and authorisation mechanisms, and fine-grained metric collection [6], [8]–[10].

Besides its effectiveness, choosing the most suitable API Gateway for a specific scenario is a challenging and time-consuming task. Also, an inadequate solution can affect the perceived quality of connected services regardless of network performance. To this extent, no automatic tools allow comparative performance assessment in either on-premises or cloud-hosted deployments. Furthermore, allied to the challenge of configuring heterogeneous testbeds, the lack of real-world benchmarks hinders drawing meaningful conclusions.

Aiming to address these challenges, this work introduces a service able to (*i*) parse a generic API documentation file into different API Gateways' configuration files; (*ii*) deploy selected API Gateways on user's cloud providers; (*iii*) load tests to all deployed API Gateways based on workloads described by the user; (*iv*) monitor the performance achieved by different hardware specification; and (*v*) provide a simple comparative metric to assist user's decision. The applicability of this proposal is assessed by resorting to an auction system as a case study, as it has the desired requirements for scalability and intensive message exchange.

This paper is organised as follows: Section II outlines the related work on comparative analyses of the main API Gateways; the design goals for an automatic system aiming for a simplified way to analyse different solutions in real scenarios are listed in Section III; the architecture and main developed components of the proposed system are described in Section IV; the proof of concept designed to demonstrate the solution and the obtained results are presented in Section V; and the conclusions are summarised in Section VI along with the planned further developments.

II. Related Work

Although current API Gateways also provide tools to support microservice-based systems' development, deployment, and management, deciding the most suitable solution typically relies on their performance related to the number of requests per second (i.e., throughput) they sustain and the average response time. Nevertheless, publicly available benchmarks are released mainly by the API Gateway's developers and do not consider real-world workloads.

One common approach is used by *KrakenD's* team [11]. It consists of generating a uniform synthetic workload towards the same API deployment running in different hardware configurations. For this benchmark, the workload is generated using the tool *hey*[1] and a web server is created with *LWAN*[2]. The results show that throughput increases and response time decreases with the computational power hosting the API Gateway.

Tyk benchmark is provided by the developers of the open source *Tyk* API Gateway [12]. Differently to the approach adopted by *KrakenD*, *Tyk benchmark* ensures the requests produced by *Locust*[3] fall into a predefined request quota and rate limit. Therefore, there is no request dropping or throughput variation, which does not resemble a real scenario. Moreover, the assessment relies only on the proxy's analytics data, and no comparative evaluation with different solutions is provided.

The *KrakenD* team also provides a comparative assessment considering two other open source solutions, i.e., *Tyk* and *Kong* [13]. To do so, the authors use the *Varnish* [4] benchmark system with a small group of simple tests. Although presenting better global performance, the limited features considered and yielded indicators hamper real-world conclusions to be taken from these tests.

Since different services pose heterogeneous performance requirements, providing developers with flexible and realistic benchmarking systems is paramount. Therefore, this work proposes AGE[5], an Automatic Performance Evaluator for multiple API Gateways that support real-world workloads and maps their performance into a single comparative indicator.

III. Design goals

AGE consists of a service capable of automatically deploying different API gateway solutions in users' cloud providers and submitting them to comparative performance assessments based on realistic system usage. Seeking flexibility and coverage, its development is driven by the following design goals:

- AGE should parse multilanguage documentation files (e.g., OpenAPI, RAML) into multiple API gateway configuration inputs. It aims at allowing the reuse of configuration files, which reduces the effort of configuring multiple gateways individually;
- The same documentation input should provide the hardware configuration to be tested, the underlying workload profile and cloud credentials;
- AGE should automatically deploy all the indicated API gateways instances in the client's cloud provider accordingly to the specified hardware;
- It also should automatically deploy load test instances and apply them to all defined API gateways;
- As a result, the system should provide a scalar score to each API gateway based on a comprehensive performance analysis;
- Detailed performance results should also be provided for each analysed parameter;
- Operating AGE should be user-friendly, i.e., hiding the complexity of configuring different API gateways, workloads and performance analysis.

IV. AGE architecture

Figure 1 presents the architectural view of AGE, including the operational flow from the documentation file to the load testing. In this figure, the input file is represented by its two main components, i.e., the API documentation and the testing workload description. The first component defines the API gateways' configurations by mapping the domain *endpoints* and the underlying *path*, *HTTP method*, and *parameters*. It also includes the infrastructure in which the API gateways will be tested, e.g., hardware specification and operating system configurations. The workload description includes test scheduling, load distribution, and the request composition to each API endpoint.

Once submitted, AGE's backend parses the input file to the configurator modules. The first one is responsible for deploying and configuring all the selected API gateways. At this point, AGE supports three mainstream solutions, i.e., *Kong*, *KrakenD*, and *Tyk*. However, new options can be included by extending the parser and configuration components. The second configurator deploys and runs the described workload, collecting and processing performance metrics.

Notice the API gateways and the load generator are deployed in the user's cloud infrastructure, while the AGE's backend might be deployed elsewhere. This provides flexibility and prevents the proposed system from affecting performance analyses.

Since the load generator requires the public IP addresses of all tested systems, AGE first deploys the API gateways, retrieves their addresses and adds them to the workload configuration file. To ensure consistency among the distributed components, the processing steps are regulated by a state machine running within AGE's backend.

Sections IV-A to IV-C provide implementation details for the main developed functions.

A. Configuration parser and instances deployment

The first module parses the generic description input into each API Gateway configuration file format. The challenge in this process is determining the right granularity in which the information within input files should be divided to adapt it to all tested systems. For instance:

- *Kong* uses a different syntax for variables that represent an endpoint path, e.g., `/bid/(?<id>[^/]+)` instead of `/bid/{id}` as in *KrakenD*;

[1]https://github.com/rakyll/hey
[2]https://lwan.ws/
[3]https://github.com/locustio/locust
[4]https://github.com/varnish/api-gateway-benchmarks
[5]https://github.com/Bishop19/AGE

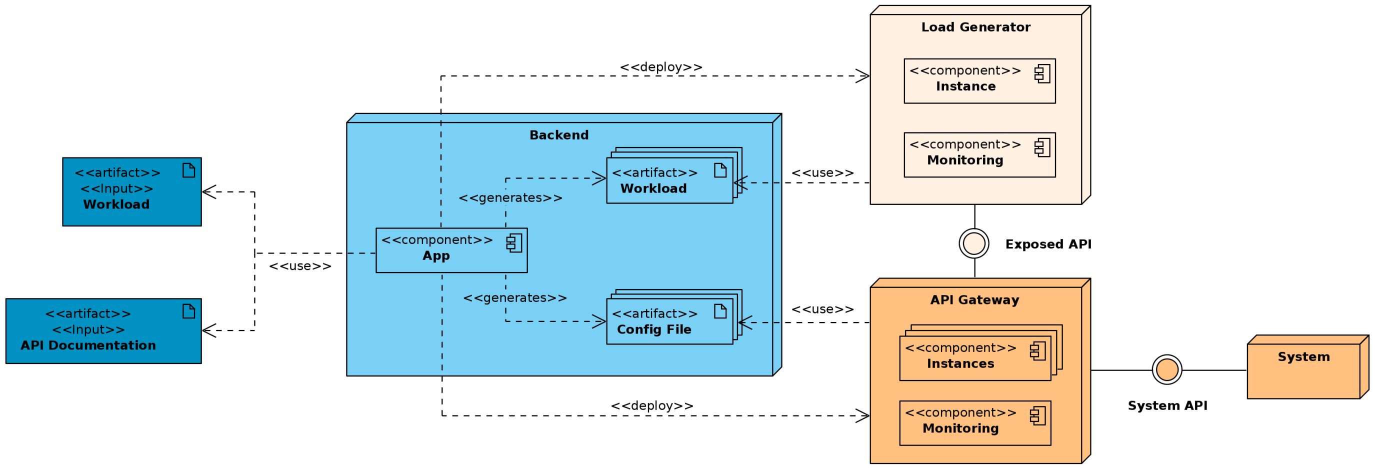


Fig. 1: AGE's architectural scheme

- *KrakenD* needs to define which headers are allowed to be forwarded explicitly, e.g., a protected API endpoint is required to allow the *Authorization* header by adding it to the *headers_to_pass* property within *KrakenD* configuration.

To address this heterogeneity challenge, AGE first generates an intermediate internal representation to be further parsed to the target API Gateway format. The internal representation is created by dividing each request URL description into representative components. For example, the request URL *https://biwi.com/bid/*$\{auctionID\}$ can be divided in two: the *base path* (or domain), *https://biwi.com*; and the *endpoint path*, */bid/*$\{auctionID\}$. Each endpoint request also has an HTTP *method* linked to it, e.g., GET or POST.

The transformations from Listing 1, which uses OpenAPI format[6], to Listing 2 (i.e., the internal representation) demonstrate the underlying process. Based on this intermediate data structure, multiple dialects used as input descriptions can be parsed to any target API Gateway configuration format.

AGE deploys the API Gateways and the load generators using the output files from the parsing phase, the described workload to be submitted during the tests, and the provided cloud credentials. Each instance is individually deployed following hardware specifications, and operating system configurations also provided through the input files.

B. Load Testing

To reduce the resources required to test multiple API Gateways, AGE deploys only one load generator per testing session, using the tool *JMeter*[7]. Nevertheless, flexibility is ensured since AGE adopts the *Strategy* design pattern. This is accomplished by the workload configurator module, which is responsible for installing the load generator and selecting the specific methods for each API Gateway under analysis in runtime.

[6]It could be in any other supported format.
[7]https://jmeter.apache.org/

The monitoring component is responsible for starting a test session, collecting all data required for comparative analysis, finishing the session, retrieving overall results, and running the *Scoring system* (see Section IV-C).

The comparative analysis is based on request throughput (i.e., the number of attended requests per second) and the response time, including the minimum, maximum, deviation, and average time, as well as the 50th, 90th, 95th, and 99th percentiles.

C. Scoring system

The comparative performance analysis is provided by a module that collects all the metrics described in Section IV-B for the endpoints being tested and calculates a global relative score (i.e., between 0 and 1) for each API Gateway system. The scoring system considers each value of the API Gateway for that endpoint metric, giving the score value of 1 to the API Gateway with the best metric value. This value could be the minimum value of the API Gateways for that specific endpoint metric:

$$min_{me} = \min(value_{meg}), \forall g \in gateways \tag{1}$$

Or the maximum:

$$max_{me} = \max(value_{meg}), \forall g \in gateways \tag{2}$$

For example, consider the minimum response times are 412 ms, 334 ms, and 587 ms for *Kong*, *KrakenD* and *Tyk*, respectively, meaning that *KrakenD* would have a score of 1 for the minimum response time metric. To calculate the other relative scores, Equations (3) and (4) are used:

$$score_{meg} = \begin{cases} 1, value_{meg} = min_{me} \\ 1 - \frac{|min_{me} - value_{meg}|}{value_{meg}}, otherwise \end{cases} \tag{3}$$

$$score_{meg} = \begin{cases} 1, value_{meg} = max_{me} \\ 1 - \frac{|max_{me} - value_{meg}|}{max_{me}}, otherwise \end{cases} \tag{4}$$

Listing 1: OpenAPI input

```
{
  "/bid/{auctionID}": {
    "post": {
      "servers": [{
          "url": "https://biwi.com/"
      }],
      "parameters": [{
          "name": "auctionID",
          "in": "path",
          "schema": {"type": "integer"}
      }],
      "requestBody": {
        "bid": { "type": "number" }
      },
      "security": "bearerAuth"
    }
  }
}
```

Listing 2: Internal structure representation

```
{
  "base_path": "https://biwi.com/",
  "endpoint_path": "/bid/{auctionID}",
  "method": "POST",
  "query_params": {},
  "path_params": {
    "auctionID": "integer"
  },
  "body_params": {
    "bid": "number"
  },
  "security": "bearer"
}
```

After all API Gateways have received a score for each metric, they are summed and scaled up, giving a value between 0 and 100 for the endpoint performance. The total score is the average from all endpoint scores, as detailed in Equations (5) and (6).

$$score_{eg} = \frac{\sum_{m}^{metrics} score_{meg} * 100}{\dim(metrics)} \quad (5)$$

$$score_{g} = \overline{score_{eg}}, \forall e \in endpoints \quad (6)$$

V. Proof of Concept

This section presents the proof of concept designed to demonstrate the usefulness of AGE in providing a simple way to compare different API Gateways under realistic workloads. To do so, it is used an open-source auction platform (i.e., BIWI[8]) which implements heterogeneous microservices connected through REST API.

A. *Test scenario*

As demonstrated in Figure 2, the proof of concept scenario comprises three virtual machines deployed in the Google Cloud provider to run BIWI, the API Gateways, and the Load tester, while the AGE backend is deployed locally. Although it is possible to deploy each API Gateway independently, in these tests, all the supported systems (i.e., *Kong*, *KrakenD* and *Tyk*) are deployed in a single machine and evaluated one at a time. As presented in Table I, all the used machines have the same basic specification and zone location.

After deploying all the required entities through AGE[9], three realistic workloads are applied to different BIWI's endpoints through the tested API Gateways. They include *bid auction*, *create auction*, and *retrieve auction* tasks, as follows:

[8]https://github.com/ambystomatidae/biwi-backend
[9]The input files are available in https://github.com/Bishop19/AGE/blob/main/doc-parser/tests/biwi-short.json

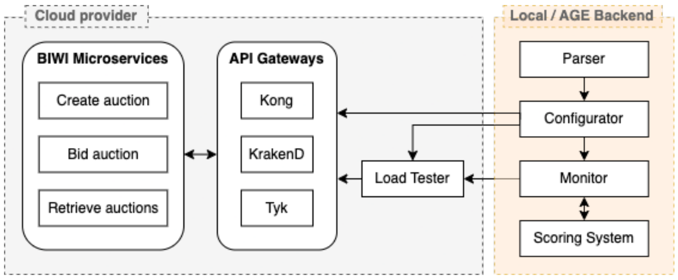


Fig. 2: Test scenario

1) Create auction: It consists of publishing new auction items, which requires POST HTTP methods containing the item description and images. This scenario aims at evaluating how the system works under heavy payload requests. Thus, the workload comprises the creation of 50 auctions with one image of size 8.6MB.

2) Bid auction: This scenario's objective is to assess system performance under time-critical tasks. The workload represents multiple users bidding on an auction concurrently. More specifically, AGE generates and submits 50 concurrent auction bids to the tested API Gateways.

3) Retrieve auction: This test scenario evaluates the API Gateways under high-demand request periods by randomly retrieving 400 auction items sequentially using the GET HTTP method. Each request returns the URLs to the item description and images.

TABLE I: Basic hardware specifications

Machine	**CPUs**	**RAM**	**Type** (Google)	**Zone**
BIWI	2	8GB	n2-standard-2	europe-central2-a
API Gateways	2	8GB	n2-standard-2	europe-central2-a
Load tester	2	8GB	n2-standard-2	europe-central2-a

All test scenarios are also compared with a *baseline*, which consists of the same workloads applied directly to the BIWI backend without using an API Gateway as a proxy. The obtained results correspond to the average of 10 executions for each workload. As discussed in Section IV, the comparison metrics include the request throughput, the response time distribution, and the unified AGE scoring.

B. Results

The results for the auction creation workload presented in Figure 3a do not show a significant difference in the average throughput achieved by the tested API Gateways. Since this task tends to pressure the backend more than its API, even the baseline scenario does not perform significantly better for the throughput analysis. However, considering the average time required to run the entire workload (i.e., creating 50 auctions), Table II shows a higher variation in results from *Tyk* that, in the worst case, took almost four times the time required by *Kong*. Such higher variation explains the lower grade assigned by the AGE scoring system (see Figure 4), in which, for this workload, the best-evaluated system was *KrakenD*, with a score of 96.

Another important result presented in Table II is the absence of errors during the tests. It means that all the auctions were successfully created in the BIWI system. This information might be used to validate whether the hardware specification provides the expected quality of service.

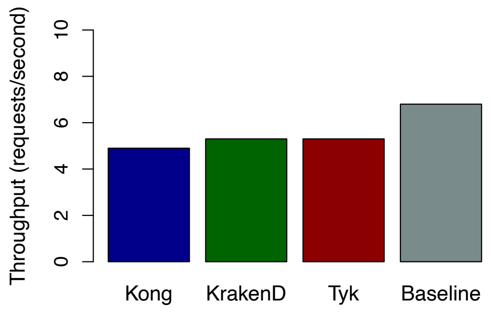
TABLE II: Summary of results - Workload: Create auction

	Response Times (ms)					
Gateway	**Min**	**Avg**	**Max**	**Std. Dev.**	**P95%**	**%Error**
Kong	139	202	307	43	282	0
KrakenD	136	186	343	36	232	0
Tyk	129	187	1186	146	257	0
Baseline	120	145	195	17	174	0

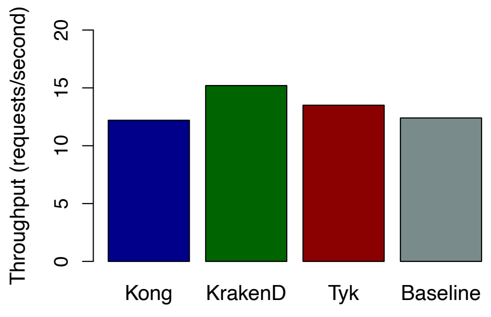
For the bidding auction workload, Figure 3b shows that *KrakenD* achieved a higher mean throughput with no errors. In this case, an error means that a received bid has a lower value than a previously processed one. Considering that all systems are tested with the same workload distribution, the errors identified for *Tyk* and the baseline scenario might be caused by network jitter. The average response time in Table III seems to confirm this for the baseline scenario, which has the worst result even without an active load balancer in the API Gateways. Comparing all the analysed metrics, the AGE scoring system points to no significant overall variation among the tested API systems. In this case, even with a few errors, *Tyk* received a slightly higher score (see Figure 4). In this case, despite the better flow rate, *KrakenD* is penalised by the higher percentile in response time, i.e., P95% in Table III.

TABLE III: Summary of results - Workload: Bid auction

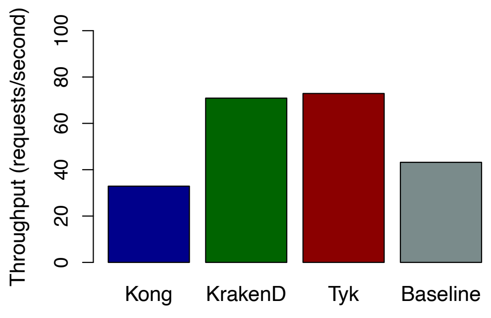
	Response Times (ms)					
Gateway	**Min**	**Avg**	**Max**	**Std. Dev.**	**P95%**	**%Error**
Kong	37	74	192	43	161	0
KrakenD	34	68	202	50	198	0
Tyk	32	73	181	49	175	2
Baseline	37	114	323	65	257	2



(a) Create auction



(b) Bid auction



(c) Retrieve auction

Fig. 3: Throughput per endpoint

The workload related to auction retrieval provides interesting comparative results. Firstly, *KrakenD* and *Tyk* outperform the throughput achieved by *Kong* and the baseline scenario by a significant margin, i.e., up to 45% (see Figure 3c). In both cases, such a difference is explained by the caching system provided by the two best performers. Similar results are observed when analysing the distribution in the response times along the tests. Notice, in Table IV, that even with lower overall throughput and higher response time, the API Gateway *Kong* still manages to finish the retrieval process without errors. Moreover, without a caching system, the baseline scenario faced a significantly higher variation in response time, demonstrated by the standard deviation of its results across the performed tests.

TABLE IV: Summary of results - Workload: Retrieve auction

	Response Times (ms)					
Gateway	**Min**	**Avg**	**Max**	**Std. Dev.**	**P95%**	**%Error**
Kong	7548	9670	11592	1408	11429	0
KrakenD	826	3360	4812	1140	4583	0
Tyk	728	3416	4618	975	4440	0
Baseline	264	5874	8436	2026	8125	0

Figure 4 shows that such performance difference is reflected by the AGE scoring system, which, for this workload, assigned the grade 58 to *Kong*, while *KrakenD* and *Tyk* received 96 and 99, respectively.

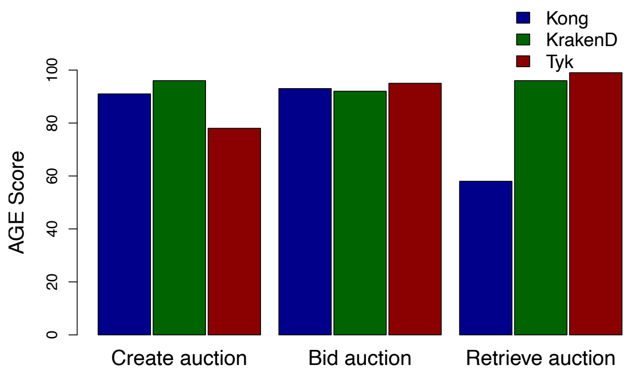


Fig. 4: AGE scoring system

Although all the comparative metrics are available through AGE's user interface, the unified scoring system can provide a simple and comprehensive indicator to support deciding on an API Gateway solution. For example, considering the workloads defined to test BIWI's services and for the specified hardware and service location, the *KrakenD* system achieved a better overall performance. Table V presents the consolidated results for all defined tests. Due to the AGE's flexibility, different workloads or hardware can be quickly evaluated by adapting the input file and repeating the test sessions.

TABLE V: Overall AGE score

API Gateway	Kong	KrakenD	Tyk
Global score	80	94	90

VI. Conclusion

With the widespread use of microservice-based services, selecting adequate API Gateways became a relevant part of system design and development, as they act as a single entry point capable of hiding internal API and load-balancing external requests. Nevertheless, the complexity and time required to deploy, configure and test multiple available solutions frequently lead to the adoption of inadequate solutions for a specific service and constraints.

This work introduces AGE, a service capable of automatically deploying different API gateway solutions in users' cloud providers and submitting them to comparative performance assessments based on realistic system usage. The service uses as input high-level description formats, such as OpenAPI documents, based on which all the infrastructure is deployed and tested under described workloads. After running the tests, AGE provides metrics related to throughput and response time distribution achieved by each analysed API Gateway. Moreover, a unifying scoring system provides a single scalar indicator to easily compare their overall performance.

The proof of concept has demonstrated the relevance of AGE by showing that, for some workloads, a subset of tested API Gateways can even outperform an implementation without using such type of proxy. It also validated the proposed scoring system, as the provided results reflect the overall performance achieved by each evaluated solution.

Further developments of AGE will include (*i*) support for distributed backends, which allows for assessing load-balancing functions; (*ii*) extended support for more API Gateways solutions; and (*iii*) support for API-level authentication.

Acknowledgements*:* †This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020. *It is also financed by National Funds through the FCT - Fundação para a Ciência e a Tecnologia, I.P. within the project FLEXCOMM, with reference EXPL/CCI-INF/1543/2021.

References

[1] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith.* O'Reilly Media, 2020.
[2] N. Ford, R. Parsons, and P. Kua, *Building Evolutionary Architectures.* O'Reilly Media, 2017.
[3] D. Geethika, M. Jayasinghe, Y. Gunarathne, T. A. Gamage, S. Jayathilaka, S. Ranathunga, and S. Perera, "Anomaly Detection in High-Performance API Gateways," *2019 International Conference on High Performance Computing and Simulation, HPCS 2019*, pp. 995–1001, 2019.
[4] J. T. Zhao, S. Y. Jing, and L. Z. Jiang, "Management of API Gateway Based on Micro-service Architecture," *Journal of Physics: Conference Series*, vol. 1087, 2018. [Online]. Available: https://iopscience.iop.org/article/10.1088/1742-6596/1087/3/032032
[5] F. Ponce, J. Soldani, H. Astudillo, and A. Brogi, "Smells and refactorings for microservices security: A multivocal literature review," *Journal of Systems and Software*, vol. 192, p. 111393, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122200111X
[6] M. Mathijssen, M. Overeem, and S. Jansen, "Identification of Practices and Capabilities in API Management: A Systematic Literature Review," 2020. [Online]. Available: https://arxiv.org/abs/2006.10481
[7] M. G. de Almeida and E. D. Canedo, "Authentication and Authorization in Microservices Architecture: A Systematic Literature Review," *Applied Sciences*, vol. 12, no. 6, 2022. [Online]. Available: https://www.mdpi.com/2076-3417/12/6/3023
[8] S. Preibisch, *API Development: A Practical Guide for Business Implementation Success.* Apress Berkeley, CA, 2018.
[9] D. Lopez. (2022, Feb.) An API Gateway is not the new unicorn. [Online]. Available: https://www.krakend.io/blog/what-is-an-api-gateway/
[10] (2022, Feb.) Open source and managed API gateway for modern applications. [Online]. Available: https://geekflare.com/api-gateway/
[11] (2022, Feb.) Benchmarks overview. [Online]. Available: https://www.krakend.io/docs/benchmarks/overview/
[12] (2022, Mar.) Benchmark - Tyk API Gateway and API Management. [Online]. Available: https://tyk.io/why-tyk/benchmark/
[13] (2022, Mar.) Comparison of KrakenD Vs other products in the market (benchmark). [Online]. Available: https://www.krakend.io/docs/benchmarks/api-gateway-benchmark/