# Combining API Patterns in Microservice Architectures: Performance and Reliability Analysis

Amine El Malki
UniVie Doctoral School Computer Science
DoCS, University of Vienna
Währinger Straße 29, 1090, Vienna, Austria
Email: amine.elmalki@univie.ac.at

Uwe Zdun
Research Group Software Architecture
Faculty of Computer Science, University of Vienna
Währinger Straße 29, 1090, Vienna, Austria
Email: uwe.zdun@univie.ac.at

*Abstract*—There are many challenges in maintaining the desired quality of service levels in modern microservice and cloud applications. Numerous techniques and patterns, such as API Rate Limit, Load Balancing, and Request Bundle, have been suggested for API services and clients to improve quality properties related to performance and reliability. However, no study has measured the impact of these techniques and their combinations in a specific configuration, especially using a large distributed system workload setting. This paper experimentally studies the effects of combining the API Rate Limit, Load Balancing, and Request Bundle patterns based on a realistic, third-party microservice-based application deployed in a private cloud and on the Amazon Web Services cloud (AWS) using 130 different configurations. We have run each configuration 500 times in the private cloud, totaling more than 4500 hours of runtime, and 200 times on AWS, totaling more than 3900 hours of runtime. We developed regression models from the collected data to predict the performance and reliability impacts of combining such techniques and patterns. We found that the models provide acceptable prediction errors below 30% on the private cloud and AWS. Further, we found that the models work best in highly reliable environments like AWS. In addition to the concrete analyses provided in our work, we propose a general and largely automated method that can be followed iteratively to evaluate similar techniques and patterns for their quality properties.

*Index Terms*—API Patterns; Microservices; Cloud; Performance; Reliability.

## I. INTRODUCTION

With the emergence of cloud and microservices-based architectures, many challenges and issues have arisen due to these architectures' high versatility and volatility. As an essential aspect of such architectures, API design plays a significant role in having seamless and reliable access from API clients to backends [1]. Hence, it has become crucial to evaluate current techniques, best practices, and their combinations to achieve the desired quality of service properties. This paper focuses on reliability and performance qualities and the tradeoffs between them. Improving one of these quality properties should not degrade the other. Unfortunately, objective support for choosing the right combination of API patterns to balance such quality properties in distributed systems is lacking so far. Also, no realistic and large-scale datasets exist that can be used as a basis for that support.

Many API patterns and techniques related to microservices APIs are recommended in the literature [2], [3], [4] to achieve desired quality properties such as performance and reliability. Three of the most common API patterns in this context are:

- *Request Bundle* [5] is a technique used on both client and server sides to reduce communication overhead caused by exchanging too many messages between both sides. This is usually achieved programmatically on both the API client and backend by adding up the data in those messages to form one block of data to be sent simultaneously. However, some complexity might be introduced since the client and server sides must handle large chunks of data [6].
- *Rate Limit* [7] can be enabled only on the server side to limit the number of requests from API clients at a specific time. This can be achieved by keeping track of the requests per API client by storing this information in a database. When the maximum limit of requests is reached, the API client is blocked until the reset time. This pattern is usually realized using *Front* or *Edge Proxies* [8], depending on the target scope of the application. As a result, it can prevent excessive and abusive API clients' requests [9]. However, it might also create bottlenecks and traffic congestion.
- *Load Balancing* at the API level is usually realized using *API Gateways* [10], [11]. It works by distributing the load over multiple node instances to avoid overloading or underloading them [12]. This is realized using *Load Balancing* algorithms like round-robin, random, and weighted algorithms that keep track of the API requests and their assigned node instances utilizing a database. However, this might also create bottlenecks and performance degradation, especially when the *Load Balancing* algorithm is not configured correctly.

Those are essential API patterns to study because they are commonly used techniques for optimizing the performance and scalability of APIs. By studying these patterns, developers can learn how to implement them effectively in their APIs and gain a deeper understanding of how they work and how they can be used to improve the performance and scalability of their APIs.

Studying these patterns in combination is particularly useful because they are often used in real-world API architectures

and might influence each other's performance and scalability. By understanding how these patterns can be combined and used, developers can understand more comprehensively how to design and build performant and scalable APIs. Additionally, studying these patterns in combination can help developers understand the trade-offs and potential challenges in using them together. It can help them to make informed decisions about which patterns are most appropriate for their specific use cases.

Our study first presents a regression model we have developed to quantify the performance and reliability impacts of such combinations of patterns. Then, we experimentally evaluate the model using a large-scale setup and a realistic, third-party microservices-based application. Please note that besides providing concrete data and analyses for the studied patterns that can, e.g., be used during early architecting, our approach can also be used as a general and largely automated method that can be applied iteratively to study other combinations of optimization techniques and patterns.

We aim to answer the following research questions:

- **RQ1** What method(s) can be used to experimentally study and evaluate possible combinations of the *Rate Limit*, *Load Balancing*, and *Request Bundle* patterns concerning the performance and reliability impacts on microservices API?
- **RQ2** What are the performance and reliability impacts of possible combinations of those patterns in a realistic application and workload setting?

We have set up a modern cloud-based infrastructure to answer these questions using the Istio[1] service mesh. Istio provides several features natively, like *Rate Limit* and *Load Balancing* [8], using its integrated Envoy[2] proxy. We used this infrastructure to measure widely-used industry practices. We have selected the Lakeside Mutual application[3] as our benchmark application, which is a realistic, third-party application (i.e., the authors were not involved in its development) developed based on practitioner experiences using several API best practices.

The benchmark tests were realized based on nine different API operations of the benchmark application using or not using *Request Bundle*. Then the other two patterns were added in different combinations for each case. The tests were executed 500 times on a private cloud and 200 times on AWS[4] for each configuration, equivalent to 130000 repetitions in total and more than 4500 hours of runtime using the private cloud and 52000 repetitions on AWS and more than 3900 hours of runtime, generating a huge data set.

We believe this is a realistic, large-scale experimental study (designed to fulfill the first methodological principle for reproducible performance evaluation in Cloud Computing [13]). Our benchmark is provided as open access[5] and can be used

to run similar tests for other API patterns and combinations of API patterns and reproduce our study.

Next, we developed regression models that could be used for guidance to architect similar types of microservice applications, as in our benchmark application, and present a complete and largely automated method to apply iteratively in different contexts and settings. We found that the models provide acceptable prediction errors on the private cloud and AWS since they are lower than 30% for Cloud-based architectures [14]. The models work best in highly reliable environments such as AWS.

The paper is structured as follows. Section II describes related works. More detailed background about API *Load Balancing*, *Request Bundle*, and *Rate Limit* is provided in Section III. Then we describe our approach in Section IV. After that, we present our regression model in Section V. Section VI provides details about our experiments setup and running the experiments. Then, the data generated are collected and analyzed in Section VII-A. Section VIII thoroughly discusses the results and threats to validity. Finally, we conclude in Section IX.

## II. RELATED WORK

Many experimental studies have studied performance and reliability prediction models in software architecture. Wartika et al. [15] proposed applying a Bayesian algorithm to evaluate the performance of a software complexity prediction model based on three requirements: user, system, and software specifications. Another study [16] performed a similar parametric approach by using the Palladio component model and a simulation tool based on that model to realize performance predictions. In contrast to the previous studies, Wang et al. [17] proposed an approach to defining a reliability prediction model based on a deep learning neural network. The authors also claim that their approach is experimentally proven to be more accurate than the other approaches. A more relevant experimental study [18] suggested the methodology MIPaRT to evaluate the performance and reliability of microservice systems continuously. The authors use the Markov model to randomly generate valid and invalid requests to the backend to explicitly force failure and ex-vivo testing to generate raw data for performance-reliability analysis. None of these studies have focused on API patterns and their combination impact on performance and reliability.

Many studies have proposed solutions to improve load balancing efficiency in microservices architecture and their future directions [19]. Yipei et al. [20] argue that existing load-balancing strategies lack optimization and suffer from network congestion and complexity. They propose a chain-oriented load-balancing algorithm (COLBA) to solve these problems and claim it is more efficient than existing load-balancing strategies. Similarly, XiaoDong et al. [21] propose a shortest queue-waiting-time load balancing algorithm (SQLB) that outperforms classical load balancing algorithms. In addition, the authors experimentally demonstrate that this algorithm generates a low prediction error. On the other hand, Rusek

---

[1]https://istio.io/

[2]https://www.envoyproxy.io

[3]c.f. https://github.com/Microservice-API-Patterns/LakesideMutual

[4]https://aws.amazon.com

[5]https://doi.org/10.5281/zenodo.7692007

et al. [22] show that decentralized load-balancing algorithms are more efficient than centralized ones by analyzing their time complexity. Another study used scenario calculus to formally analyze load balancing impact on scalability and capability of microservices [23]. It demonstrates that elastic round-robin scheduling is more scalable than shortest-waiting queue scheduling. Li et al. [24] developed the DWLOAD (Dynamic Weight Loading) algorithm to maintain high efficiency and reliability of load balancing in a microservices environment. This was mainly realized by calculating and recording the real-time weights of the hosting servers. BLOC, a self-managing load balancing system, was also proposed [25] to provide a second layer control for service mesh load balancers in microservices architectures using other patterns like *Rate Limit*. Lastly, an online decision algorithm based on Lyapunov optimization techniques was proposed to distribute the load in a hybrid cloud cost-effectively when a flash crowd event occurs [26]. As a result, only a few studies have evaluated the impact of load balancing on reliability. Also, they have yet to consider combining it with other patterns such as *Request Bundle* either.

*Request Bundle* and *Rate Limit* have been defined as best practices or patterns [27], [28], [29], [30], and particularly in the context of Service-Level Agreements [31], [32]. Although these API patterns have been studied experimentally [5], [33], their combinations with other API patterns and techniques have not been the subject of any experimental study in the scientific literature. This represents a significant gap in understanding the impact of widely used API practices such as *Rate Limit* and *Load Balancing* and their combinations. Also, there is no way to measure the impact of those API patterns and their combinations on some important quality properties of a specific configuration setting, such as performance and reliability. Our study closes this gap by presenting an experimentally validated regression model using a real-life microservice-based application and setting. This model could also support the existing simulators and prediction models.

## III. Background about Load Balancing, Request Bundle and Rate Limit

*Load Balancing* is a technique to distribute the load evenly between two or more resources to avoid overloading or under-loading them. This technique can be performed either on cloud resources level [34] or service level [35]. Our study mainly focuses on service level load balancing, a feature provided by service meshes like Istio [8]. Indeed, Istio provides four methods to achieve load balancing. While the *round robin* method dispatches API requests to replicas using a particular order, the *random* method does it randomly. On the other hand, the *weighted* method explicitly assigns a percentage weight to each node instance. More interestingly, the *least requests* method gives the largest weight dynamically to the node instance with the least number of requests.

*Request Bundle* [6] generally consists of containerizing multiple messages into one containing additional information related to those messages, such as identifiers. Instead of sending many messages to the API backend, request bundling encapsulates those messages into one, avoiding unnecessary chatty communication. However, some complexity might be introduced on both the API client and server, which might also cause bottlenecks and traffic congestion. In our study, we limited the bundle size to 50 to reduce the likelihood of those issues. *Request Bundle* is a general practice already adopted in many APIs [6]. Since those APIs and others without request bundling constantly evolve, measuring the impact of using such a practice and its combination with other API patterns regarding performance and reliability is crucial.

*Rate Limit* has also been the focus of many studies [7], [27], [29], but has also not been studied with combinations of other patterns and techniques. It is usually implemented using an API Gateway [36] and ensures that API providers are not overwhelmed by an intentional or unintentional flood of requests from API clients. *Rate Limit* plays an essential role in maintaining the desired level of quality property related to performance and reliability, and combining it with a billing strategy or a *Rate Plan* [37] makes it even more effective. However, efficiency drastically decreases by including mandatory patterns like *API key* [37] to keep track of API clients. In our study, we use Istio, which supports two types of rate-limiting: global and local rate limiting. While the former is activated globally on the service mesh level, the latter triggers each service instance of the mesh.

## IV. Approach overview

Figure 1 illustrates the patterns and combination scenarios we have considered in our study. Finding an optimal point or area in possible solutions where performance and reliability are maximized is not straightforward without additional information. So, we need first to identify parameters and metrics to estimate the localization of this point or area. Also, this optimal point or area might change in time due to unforeseen circumstances like resource saturation. Thus, we need to identify a model that accurately considers chosen and third-party parameters related to resource management. We have also defined metrics that could estimate the localization of the optimal point or area in the space of possible solutions.

Combining the three patterns and techniques we have considered in our study must be carefully decided. For example, in many situations, it will not make sense to use *Request Bundle* with *Rate Limit*, since otherwise, we would have large API requests that overload the backend before *Rate Limit* is triggered. Also, using *Request Bundle* with *Load Balancing* is a logical choice to not overwhelm one node on the backend with large API requests to handle. However, using *Request Bundle* along with *Rate Limit* might not work correctly since the backend would be already overloaded after only a few API requests. Then, it would be wiser to introduce *Load Balancing*, too, to avoid such a scenario. Also, combining patterns generates costs and complexity, which should be justified in advance by accurately measuring the foreseen improvements of those combinations on performance and reliability. There should be a way to find a balance between high performance
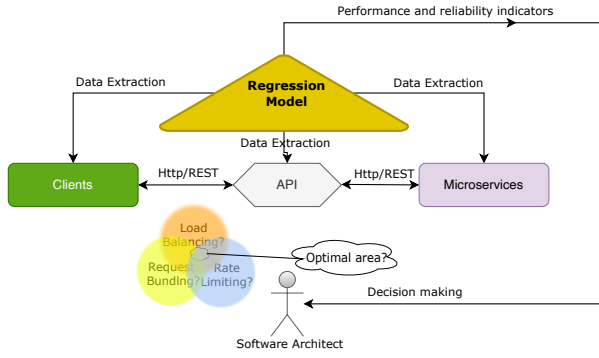
Fig. 1. Approach overview

and reliability in conjunction with low costs and complexity and, if required, find the best combination to handle a specific workload configuration, which is the aim of our model.

Our proposed regression model employs data extracted from API or API Gateway, API clients, and API backend or microservices. The data extraction is realized by considering each possible combination of the patterns in our study. The iterative process illustrated in Figure 1 makes the approach dynamic and adaptive, depending on the usage period and the cloud setting. That means the same regression model could dynamically extract the performance and reliability indicators in a completely different cloud environment. Please note that these indicators would only represent a rough estimation to the software architect for decision-making. More accurate predictions, like the usage period, would need more parameters to consider in the regression model.

## V. REGRESSION MODEL DESCRIPTION

### A. Independent and dependent variables

As previously stated in Section III, our work is motivated by the high uncertainty in the scenarios covered in Figure 1 to localize the optimal point or area in the combination of the patterns to achieve optimal performance and reliability. In addition, the optimal selected point or area might change in time due to unforeseen circumstances like resource saturation. Hence, given a specific workload configuration, we first need to define a model that could help us measure the performance and reliability impact.

Our study has considered two types of independent variables: numerical and categorical. The first type of variables includes $rpm$, which is the number of requests per minute for a specific workload setting, $api\_users$ which is the number of API users that generate load for that particular workload setting, $rate\_limit$, which is the maximum predefined number of requests allowed per API user per minute, $bundle\_size$ which indicates the size of the request bundle, in case it is used and $methods\_count$ which reflects the number of methods that are considered in the study. The second type of variables includes $request\_bundle$ and $load\_balancing$, which indicate whether *Request Bundle* and *Load Balancing* are used respectively. It also includes $method$, which indicates

the API operation executed on the backend: Get, Update, Delete, or Create. (see Table I).

The first dependent variable we have considered is $total\_time$, equivalent to the total round trip time spent by API requests to be completely fulfilled. It is defined as:

$$total\_time = network\_time + backend\_time$$

$network\_time$ is equivalent to the round-trip time spent by API requests from the API client and the API Gateway, and $backend\_time$ is the round-trip time spent from the API Gateway backend service(s) and maybe database(s). The prior study needed to include server details concerning program execution and database time. As these details are not required here, we employ $rpm$ as an API client parameter that is more straightforward to calculate to define $total\_time$ as follows:

$$total\_time = \sum_{i=1}^{api\_users} \frac{bundle\_size_i}{rpm_i}$$

where,

$$bundle\_size = \begin{cases} 10\ldots50, & \text{if } request\_bundle = Yes. \\ 1, & \text{otherwise.} \end{cases}$$

The second dependent variable is $failure\_rate$, which indicates these API requests' failure percentage. Here, we do not distinguish failure rate due to rate limiting and failure rate due to other reasons than rate limiting. We define:

$$failure\_rate = \begin{cases} \sum_{i=1}^{methods\_count} rpm_i \cdot \frac{load\_weight_i}{load\_balancers} - rate\_limit_i, \\ \text{if } rpm_i \cdot \frac{load\_weight_i}{load\_balancers} > rate\_limit_i \\ \wedge \ load\_balancers \geq 1 \\ \\ \sum^{n_t} \frac{P_{failure}^t}{api\_users} \cdot failure\_rate^t, \text{otherwise.} \end{cases}$$

The load balancing algorithm determines $load\_weight_i$. In our study, we exclusively used the *weighted load balancing* algorithm, and thus their values are fixed and known in advance along with $methods\_count$ and $load\_balancers$. Therefore, these variables are not included in the following regression analysis. $P_{failure}^t$ and $failure\_rate^t$ denote the probability of failure during each time interval $t$ and the corresponding failure rate as proposed in a prior study [33]. This is a Bernoulli distribution composed of $n_t$ trials and is not considered further in the current study, so these parameters are not included in the regression analysis.

## VI. EXPERIMENTATION SETUP

### A. Application under study

We have decided to choose Lakeside Mutual[6] application, which is a real-life application developed to make in practice various microservice API patterns like *Request Bundle* [6]. Indeed, we have primarily extended the application to support request bundling on the following functions (provided in the

---

[6]https://github.com/Microservice-API-Patterns/LakesideMutual

TABLE I
DEFINITION OF PARAMETERS

| Independent variable | Description |
|---|---|
| $rpm$ | The number of requests per minute for API users. |
| $api\_users$ | The number of API users sending the requests. |
| $rate\_limit$ | The value of the applied rate limit ranging from 100 to 1200 (100 increments). |
| $request\_bundle$ | Categorical variable indicating whether request bundling is used or not. |
| $bundle\_size$ | The request bundle size ranges from 10 to 50 (10 increments). |
| $load\_balancing$ | Categorical variable indicating whether load balancing is used or not. |
| $load\_balancers$ | Number of load balancers. |
| $load\_weight$ | Percentage of requests that is forwarded to a specific backend instance. |
| $method$ | Categorical variable indicating the method executed by API users. |
| $methods\_count$ | Number of methods executed by API users. |

| Dependent variables | Description |
|---|---|
| $total\_time$ | The total roundtrip time spent for requests between API client and backend. |
| $failure\_rate$ | The percentage failure rate of API requests. |

TABLE II
AWS COST DETAILS (IN US DOLLARS)

| Service | Cost |
|---|---|
| EC2-Other | 1783,53 |
| EC2-Instances | 1369,52 |
| Elastic Container Service for Kubernetes | 356,67 |
| EC2-ELB | 356,67 |
| Tax | 701,98 |
| Total | 4211,89 |

replication package[7] and designed to fulfill the fourth methodological principle for reproducible performance evaluation in Cloud Computing [13]):

- Update customers using their IDs by application customers or administrators. Get customers already supports request bundling.
- Create/Get/Delete insurance quota requests using their IDs, which application customers do.
- Create/Get/Update/Delete insurance policies done by the application administrator using *policy-management-backend* microservice.

*Request Bundle* is enabled by implementing a new dedicated chain of methods on the backend supporting a list as an input parameter instead of one parameter at a time for each API operation described above. For example, *getCustomers()* method now gets a list of customers' IDs instead of one ID at a time. *Rate Limit* and *Load Balancing* are both supported by Istio through EnvoyFilter[8] and virtual services[9] respectively. The full implementation details are provided in the replication package.

We are executing nine different API functions on the backend with and without using *Request Bundle*. In our study, we could have used an existing benchmark. Other cloud benchmarks exist, like those proposed in the context of Cloudsim [38] and PerfCompass [39], but they are relatively low-level. They are defined at the network rather than the business application level, which is relevant for request bundling. As a consequence, there are no usable cloud application-level benchmarks. Even if some existed, they likely would not apply to our study, as we would need them to use request bundling. Instead, we defined realistic scenarios based on Lakeside Mutual's request bundling functions described above.

## B. Infrastructure details

We have used a private cloud hosted in our University data center for our experiment. It is composed of 11 Ubuntu[10] 18.04.5 LTS Virtual Machines (VMs) installed on vSphere[11] 6.7 environment. Each of these VMs runs a Minikube instance version 1.20.0 with eight dedicated CPU cores Intel Xeon(R)™CPU E5-2650 v4 @ 2.20GHz and 20 GB of system dedicated memory. Each Minikube instance executes a Kubernetes[12] engine version 1.20.2 and Istio version 1.10.0.

There are two Cluster environments that we have set up to differentiate API requests that are handled with and without load balancing. The first Cluster comprises two replicated central Minikube instances containing the central *Control Plane* and the *customer-self-service-backend* microservice built using Java version 8 each. Replication is also done for Minikube instances running *customer-management-backend* and *policy-management-backend*, which are built using the same version of Java and accessible using *Ingress Controller* YAML-defined Kubernetes Endpoints[13] and Ingress Rules[14]. The replicas share one Mongo[15] (latest version) database instance. The second Cluster does not support load balancing, and thus there is no replication.

On the client side, two Ubuntu 18.04.5 LTS virtual desktops inject HTTP requests into the private cloud. Each virtual desktop has 2 CPU cores Intel® Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.

We have also run the same experiment on AWS using a four-node Amazon Elastic Kubernetes Services (EKS[16]) cluster. The costs of using AWS are described in Table II (designed to fulfill the eighth methodological principle for reproducible performance evaluation in Cloud Computing [13] and not applicable for the private cloud).

## C. Running the experiment

Each virtual desktop executes a Shell[17] script with 130 different configurations of bundle sizes ranging from 10 to 50, rate limits ranging from 100 to 1200 set on server-side ($\infty$ if rate-limiting is disabled), and whether request bundling

---

[7]https://doi.org/10.5281/zenodo.7692007

[8]https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/

[9]https://istio.io/latest/docs/concepts/traffic-management/#virtual-services

[10]https://releases.ubuntu.com/18.04/

[11]https://www.vmware.com/products/vsphere.html

[12]https://kubernetes.io

[13]https://kubernetes.io/docs/concepts/services-networking/service/

[14]https://docs.konghq.com/kubernetes-ingress-controller/1.1.x/guides/getting-started/

[15]https://www.mongodb.com

[16]https://aws.amazon.com/eks/

[17]https://www.shellscript.sh

250

## TABLE III
### Combined models' description on the private cloud: total time prediction

| Model | Total time | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Intercept | Load balancing | Request bundle | Rate limit | API users | RPM | Method delete | Method get | Method update |
| 1 | 745,1 | 294,9 | -79,37 | 0,0051 | 36,78 | -0,12 | -334,9 | -1009 | 17,53 |
| 2 | 338,4 | 294,9 | -79,73 | 0,0051 | 36,78 | -0,12 | X | X | X |
| 3 | 297,6 | 295,2 | X | 0,0052 | 36,74 | -0,12 | X | X | X |
| 4 | 489,7 | X | -80,66 | 0,0048 | 36,85 | -0,13 | X | X | X |
| 5 | 448,4 | X | X | 0,0049 | 36,81 | -0,13 | X | X | X |

## TABLE IV
### Combined models' description on AWS: total time prediction

| Model | Total time | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Intercept | Load balancing | Request bundle | Rate limit | API users | RPM | Method delete | Method get | Method update |
| 1 | 8504 | 620,5 | -5872 | 0,19 | 310,4 | -16,71 | -662,4 | -1852 | -1349 |
| 2 | 7406 | 592,8 | -5845 | 0,19 | 310,4 | -16,72 | X | X | X |
| 3 | 297,6 | 295,2 | X | 0,0052 | 36,74 | -0,12 | X | X | X |
| 4 | 489,7 | X | -80,66 | 0,0048 | 36,85 | -0,13 | X | X | X |
| 5 | 448,4 | X | X | 0,0049 | 36,81 | -0,13 | X | X | X |

## TABLE V
### Combined models' description on the private cloud: failure rate prediction

| Model | Failure rate | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Intercept | Load balancing | Request bundle | Rate limit | API users | RPM | Method delete | Method get | Method update |
| 1 | 46,02 | -11,63 | -38,59 | -0,0025 | -0,088 | 0,0058 | n/a | n/a | n/a |
| 2 | 46,02 | -11,63 | -38,59 | -0,0025 | -0,088 | 0,0058 | X | X | X |
| 3 | 26,2 | -11,46 | X | -0,0024 | -0,1 | 0,0062 | X | X | X |
| 4 | 40,05 | X | -38,54 | -0,0024 | -0,091 | 0,0059 | X | X | X |
| 5 | 20,34 | X | X | -0,0024 | -0,11 | 0,0063 | X | X | X |

## TABLE VI
### Combined models' description on AWS: failure rate prediction

| Model | Total time | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Intercept | Load balancing | Request bundle | Rate limit | API users | RPM | Method delete | Method get | Method update |
| 1 | -1,76 | 0,81 | 3,13 | 0,00017 | 0,059 | -0,0045 | -0,07 | n/a | n/a |
| 2 | -1,77 | 0,81 | 3,13 | 0,00017 | 0,059 | -0,0045 | X | X | X |
| 3 | -1,43 | 0,66 | X | 0,00024 | 0,041 | -0,0004 | X | X | X |
| 4 | -1,38 | X | 3,01 | 0,00017 | 0,059 | -0,0045 | X | X | X |
| 5 | -1,11 | X | X | 0,00024 | 0,041 | -0,0043 | X | X | X |

is used or not. The script executes the following depending on the configuration:

- Create a predefined number of API users;
- Get customers' data with or without request bundling;
- Update customers' addresses with or without request bundling;
- Create/get/delete insurance quota requests with or without request bundling;
- Create/get/update/delete policies with or without request bundling;

Those API requests or methods are executed using load balancing in only the first Cluster, as stated in Section VI-B. We have run the script 500 times for each configuration in the private cloud, totaling 130000 repetitions and more than 4500 hours of runtime, and 200 times on AWS, counting 52000 repetitions and more than 3900 hours of runtime (designed to fulfill the second methodological principle for reproducible performance evaluation in Cloud Computing [13]). $total\_time$ is calculated at the end of each trial, and $failure\_rate$ is calculated when switching from one configuration to another.

## VII. Multivariate regression analysis

### A. Data collection and analysis

All the data are collected from output log files on the client side. Due to the large set of data collected (130 configurations tested), we have decided to include it in an online Appendix[18].

We have realized data analysis using R language[19]. Tables III and IV list the combined models that we generated from the data collection for $total\_time$ prediction on the private cloud and AWS, respectively. Tables V and VI list the combined models that we generated from the data collection for $failure\_rate$ prediction on the private cloud and AWS, respectively. We have only collected data concerning the

independent variables that we have experimentally varied as described in Section V-A. Notice that the $method$ variable is not significant using model 1 concerning $failure\_rate$ prediction but highly significant using the same model concerning $total\_time$ prediction. Also, the coefficient values relative to $load\_balancing$ and $request\_bundling$ reflect the case where they are enabled. In other words, when they are disabled, the values of those coefficients are zero, as described [40]. The same applies to $method$ create that gets zero value for its coefficient in model 1.

On the private cloud and as shown in Table VII, the F-statistic p-value is very low for all the models, and the Adjusted R-Squared is relatively high, especially for models 1, 2, and 4 regarding $failure\_rate$ prediction. The latter is even higher on AWS regarding $total\_time$ prediction but lower regarding $failure\_rate$ prediction. However, no conclusion can be made without analyzing the models' residuals. We used Anderson-Darling's and Henze- Zirkler's tests to realize residuals' univariate and multivariate normality. As clearly shown in the models (provided in the online Appendix[18]), the histograms are not curvy-shaped, indicating that the residuals are not normally distributed. This is confirmed by the Chi-Square Q-Q plots and Scatter plots showing that the points are widely spread and do not form any obvious straight line. The perspective 3D plots deny residuals multivariate normality as

## TABLE VII
### Combined models' summary: total time & failure rate prediction

| # | Private Cloud | | | | AWS | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | Total time | | Failure rate | | Total time | | Failure rate | |
| | Adjusted R-squared | F-Statistic p-value | Adjusted R-squared | F-Statistic p-value | Adjusted R-squared | F-Statistic p-value | Adjusted R-squared | F-Statistic p-value |
| 1 | 0,2452 | <2.2e-16 | 0,5925 | <2.2e-16 | 0,6438 | <2.2e-16 | 0,09809 | <2.2e-16 |
| 2 | 0,1683 | <2.2e-16 | 0,5925 | <2.2e-16 | 0,6383 | <2.2e-16 | 0,09806 | <2.2e-16 |
| 3 | 0,1677 | <2.2e-16 | 0,3641 | <2.2e-16 | 0,6113 | <2.2e-16 | 0,05358 | <2.2e-16 |
| 4 | 0,1599 | <2.2e-16 | 0,5717 | <2.2e-16 | 0,6374 | <2.2e-16 | 0,08846 | <2.2e-16 |
| 5 | 0,1593 | <2.2e-16 | 0,3439 | <2.2e-16 | 0,6094 | <2.2e-16 | 0,04708 | <2.2e-16 |

| Variable | Model 1 | | Model 2 | | Model 3 | | Model 4 | | Model 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Pillai | p-value | Pillai | p-value | Pillai | p-value | Pillai | p-value | Pillai | p-value |
| $rpm$ | 0,15 | <2.2 e-16 | 0,15 | <2.2 e-16 | 0,14 | <2.2 e-16 | 0,16 | <2.2 e-16 | 0,15 | <2.2 e-16 |
| $api\_users$ | 0,08 | <2.2 e-16 | 0,07 | <2.2 e-16 | 0,07 | <2.2 e-16 | 0,07 | <2.2 e-16 | 0,07 | <2.2 e-16 |
| $rate\_limit$ | 0,1 | <2.2 e-16 | 0,1 | <2.2e-16 | 0,07 | <2.2 e-16 | 0,1 | <2.2 e-16 | 0,07 | <2.2 e-16 |
| $request\_bundle$ | 0,32 | <2.2 e-16 | 0,32 | <2.2 e-16 | n/a | n/a | 0,32 | <2.2 e-16 | n/a | n/a |
| $load\_balancing$ | 0,04 | <2.2 e-16 | 0,04 | <2.2 e-16 | 0,03 | <2.2 e-16 | n/a | n/a | n/a | n/a |
| $method$ | 0,07 | <2.2 e-16 | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |

| Variable | Model 1 | | Model 2 | | Model 3 | | Model 4 | | Model 5 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Pillai | p-value | Pillai | p-value | Pillai | p-value | Pillai | p-value | Pillai | p-value |
| $rpm$ | 0,16 | <2.2 e-16 | 0,16 | <2.2 e-16 | 0,54 | <2.2 e-16 | 0,15 | <2.2 e-16 | 0,54 | <2.2 e-16 |
| $api\_users$ | 0,34 | <2.2 e-16 | 0,33 | <2.2 e-16 | 0,38 | <2.2 e-16 | 0,33 | <2.2 e-16 | 0,38 | <2.2 e-16 |
| $rate\_limit$ | 0,02 | <2.2 e-16 | 0,02 | <2.2e-16 | 0,02 | <2.2 e-16 | 0,02 | <2.2 e-16 | 0,02 | <2.2 e-16 |
| $request\_bundle$ | 0,11 | <2.2 e-16 | 0,11 | <2.2 e-16 | n/a | n/a | 0,11 | <2.2 e-16 | n/a | n/a |
| $load\_balancing$ | 0,01 | <2.2 e-16 | 0,01 | <2.2 e-16 | 0,01 | <2.2 e-16 | n/a | n/a | n/a | n/a |
| $method$ | 0,01 | <2.2 e-16 | n/a | n/a | n/a | n/a | n/a | n/a | n/a | n/a |



Fig. 2. Performance analysis (on the private cloud)
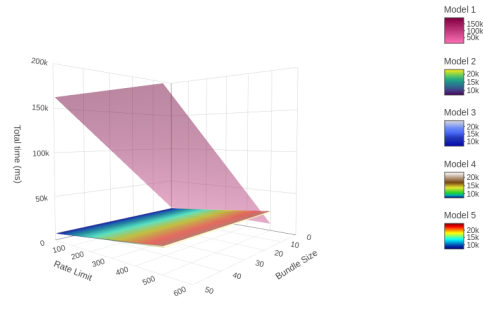


Fig. 3. Performance analysis (on AWS)

well. Since the normality of residuals, one of the assumptions of MANOVA (Multivariate Analysis of Variance), is violated, we have decided to perform MANOVA Pillai's Trace test statistics for further analysis [41].

As shown in Table VIII, all the variables have very high significance since their p-values are very low. As a reminder, this significance increases as a variable's Pillai value gets closer to 1. As a result, we notice that $request\_bundle$ is the best predictor in models 1, 2, and 4 on the private cloud. On the other hand, $load\_balancing$ has the lowest Pillai value in models 1, 2, and 3. Also, each specific variable's Pillai value remains stable in the models where the variable is used. Concerning AWS, Table IX also shows that all the variables have very high significance. However, we see clearly that $api\_users$ and $rpm$ are the best predictors in all the models, and the rest of the variables have lower significance.

## VIII. DISCUSSION AND THREATS TO VALIDITY

### A. Models comparison

Let us compare the models we have found in detail, first for their performance and then for their reliability impacts. Figure 2[20] shows that $total\_time$ in model 1 is very sensitive to $bundle\_size$, thus a large performance degradation starts very early. In contrast, $rate\_limit$ does not show any clear impact on performance. All the remaining models overlap, meaning they have similar behavior concerning performance.

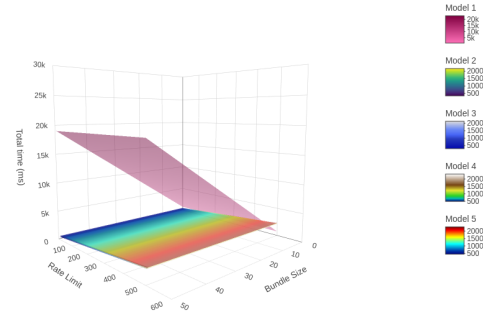[20]All the figures in this section were realized using the Plotly package provided by R language; see https://plotly.com.

$bundle\_size$ does not play a role in models 2 and 4, but increasing $rate\_limit$ slightly decreases performance. Also, $load\_balancing$ has no apparent effect on performance. As a result, model 1 is only usable in architectures where *Request Bundle* is not the preferred option and high performance is required. For example, this is where API messages are already large and need a high amount of bandwidth. On the other hand, the remaining models are more suitable when API messages are small and should use *Request Bundle*. This reduces the impact of *Rate Limit* on performance consequently and keeps it manageable, as clearly shown in the figure. Figure 3, realized using AWS-generated data, confirms the conclusions above with a degraded performance overall due to latencies and timeouts usually occurring in public clouds.

In Figure 4, we can see that the models diverge but show a similar trend as in Figure 2. Model 1 is also susceptible here and witnesses a massive increase in reliability at the highest value of $bundle\_size$ for all values of $rate\_limit$. The remaining models exhibit a gradual reliability increase, with the increase of $rate\_limit$. In models 2 and 4, $bundle\_size$ does not play a role, too. Here, $load\_balancing$ impacts reliability, especially in model 2. Like the conclusions drawn regarding performance analysis, model 1 is suitable for architecting systems that exchange large API messages where $bundle\_size$ is less than or equal to 10. Also, the remaining models can be used by systems exchanging smaller API messages using (or not using) *Request Bundle*. However, at this time, the impact
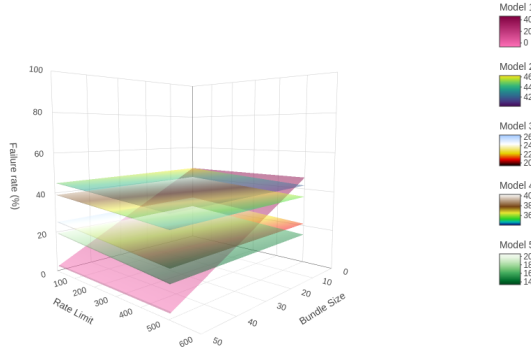
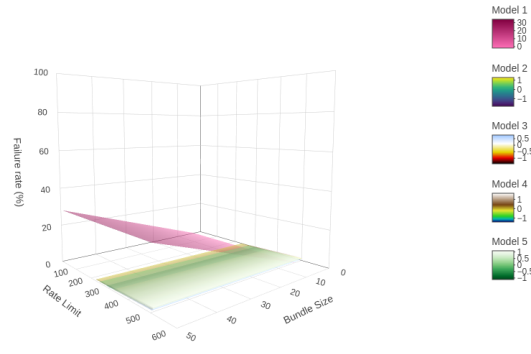Fig. 4. Reliability analysis (on the private cloud)
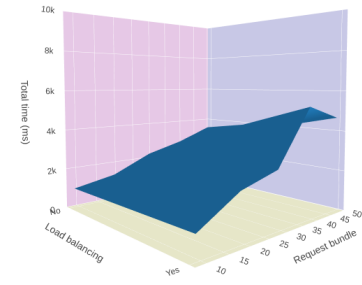


Fig. 5. Reliability analysis (on AWS)



Fig. 6. Load Balancing vs. Request Bundle: performance perspective (on the private cloud)



Fig. 7. Load Balancing vs. Request Bundle: performance perspective (on AWS)

of *Rate Limit* on reliability is influenced by $load\_balancing$ and illustrated specifically by models 2 and 3 in the figure.

Figure 5, realized using AWS-generated data, shows a different trend for the models. The increase in $failure\_rate$ when increasing bundle size is mainly due to the lack of scalability since we have only used one AWS cluster as described in Section VI-B. For models 2 to 5, $failure\_rate$ is higher in Figure 4 due to lower latency, which increases $failure\_rate$ due to *Rate Limit*. For model 1, as the bundle size increases, more requests get bundled, which produces fewer $rpm$ and thus less $failure\_rate$ due to *Rate Limit*.

### B. Tradeoff analysis

We have investigated different combinations of API patterns like *Circuit Breaker* and *Rate Limit*. We have not considered such combinations since they did not show any interesting dependencies from which design advice can be derived. For instance, *Circuit Breaker* may directly hinder the real impact of *Rate Limit* on reliability. In contrast, the selected combinations have revealed a clear understanding of their impact on reliability and performance. We perform an API patterns tradeoff analysis from the experimental data to validate and complement the models' analysis presented above. We analyze tradeoffs of using *Load Balancing* with firstly *Rate Limit* and secondly *Request Bundle* in terms of performance and
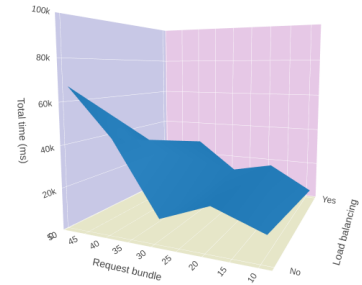
reliability impact. Thus we also study tradeoffs between the two investigated quality properties.

**Load Balancing vs. Request Bundle.** As seen in Figure 6, *Load Balancing* has a clear negative impact on performance, especially when $bundle\_size$ gets higher starting from approximately 30 requests. That is because processing large chunks of data by only one load balancer quickly creates a long waiting queue, which could be resolved by adding more load balancers. This trend is partially confirmed by Figure 7 realized using AWS-generated data, with a significant degradation in performance explainable by latency and timeouts that usually occur on public clouds. However, as $bundle\_size$ gets higher, *Load Balancing* improves performance considerably, unlike in the private cloud. Also, both patterns do not impact reliability, as depicted in Figure 8. Models 1 and 2 include $load\_balancing$ and $request\_bundle$, but only Model 2 is recommended in case of *Request Bundle* usage, as stated in the previous section.

**Load Balancing vs. Rate Limit.** On AWS, the failure rate remains very low, mainly due to the high reliability provided by public clouds and latency that prevents *Rate Limit* activation. On the other hand, Figure 9 shows that *Load Balancing* reduces the failure rate on private cloud, especially
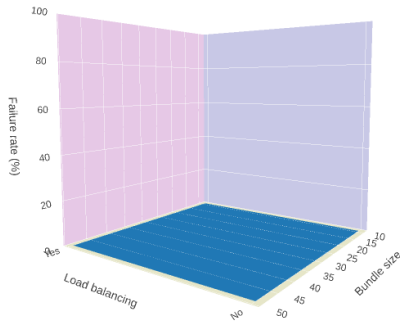
253

Fig. 8. Load Balancing vs. Request Bundle: reliability perspective (on private cloud and AWS)
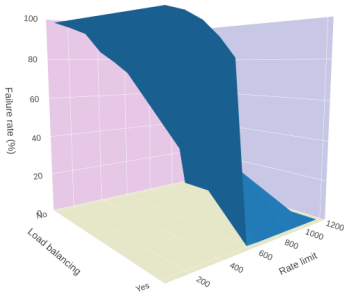


Fig. 9. Load Balancing vs. Rate Limit: reliability perspective (on the private cloud)
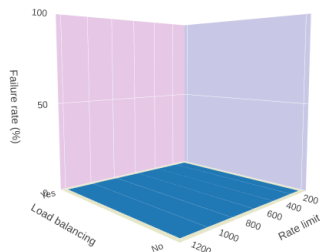


Fig. 10. Load Balancing vs. Rate Limit: reliability perspective (on AWS)

when the $rate\_limit$ value is very high; though, this is hardly noticeable when its value is between 200 and 600 requests per minute. Otherwise, as mentioned earlier, *Load Balancing* reduces the impact of *Rate Limit* on reliability. This could also be reinforced by using additional $load\_balancers$. However, this could also lead to bottlenecks and traffic congestion, as
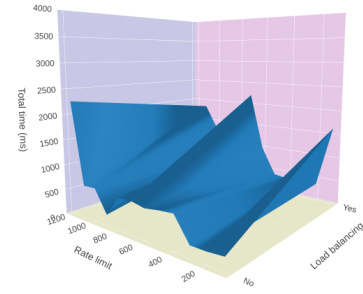


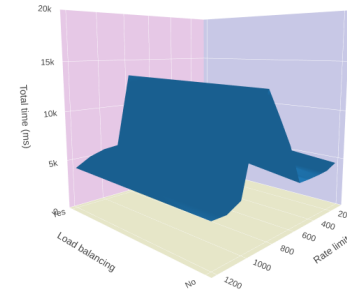Fig. 11. Load Balancing vs. Rate Limit: performance perspective (on the private cloud)



Fig. 12. Load Balancing vs. Rate Limit: performance perspective (on AWS)

depicted in Figures 11. Performance degradation is higher on AWS, and the highest $total\_time$ value is collected at a $rate\_limit$ of approximately 800 requests per minute, as depicted in Figure 12. Notice that *Load Balancing* has no obvious impact on performance because, as stated previously, only one cluster is used. Here, we consider Models 2 and 3 since Model 1 is not influenced by the $rate\_limit$ as far as reliability is concerned, as mentioned in the previous section. Also, model 3 is best where $load\_balancing$ has the lowest impact on reliability.

### C. Models evaluation

To evaluate the accuracy of the models, we have used the Mean Average Error (MAE) calculations for $failure\_rate$ prediction errors using the private cloud and AWS data. We have also used the Mean Absolute Percentage Error (MAPE) estimates for $total\_time$ prediction errors. The results are reported in Table X. Usually, it is advised to use MAPE for calculating model prediction error [42]. However, as $failure\_rate$ usually equals 0, MAPE is not applicable, and MAE is the next preferred choice. We see that all models' prediction errors on both clouds are lower than 30%. This is often considered to be acceptable for Cloud-based architectures [14]. These numbers are good enough, especially when

| Model | Private Cloud | | AWS | |
|---|---|---|---|---|
| | MAE% | MAPE% | MAE% | MAPE% |
| 1 | 19,72 | 28.51 | 1,56 | 24,83 |
| 2 | 19,72 | 27,99 | 1,56 | 24,83 |
| 3 | 27,25 | 28,44 | 1,33 | 24,58 |
| 4 | 19,67 | 28,97 | 1,48 | 24,82 |
| 5 | 27,95 | 29,51 | 1,29 | 25,3 |

used for early architecting, where orders of magnitude are needed rather than precise predictions. We also observe that the prediction errors are lower on AWS compared to the private cloud, especially regarding $failure\_rate$. This is explainable by the large latency gap that diverges the two environments and that we have not included in the models. Indeed, in the private cloud, the latency is very low, and thus $failure\_rate$ due to *Rate Limit* is high compared to AWS. It demonstrates that the models work best in high-latency environments like public clouds.

### D. Threats to validity

In the model presented, we have carefully selected the independent variables and varied them in such a way as to comprehend their causal effect relationship to performance and reliability. We have also generated five models of all possible combinations of those independent variables. As described in Table VIII, all the models present very high *statistical significance*, which emphasizes the causal effect relationship, and thus mitigates many related internal threats to validity. However, we could have unwillingly made errors in the experiment's selection and implementation of some scenarios. The workloads could also not be representative of a real-world benchmark. We have mitigated this issue by covering several widely used scenarios in the experiment.

As described in Table VIII, all the models present very high *statistical significance* but with different Pillai values. We have also found that these values for specific variables remain stable from one model to another. This could be biased because we only used a unique setup for the experiment described in Section VI-B, leading to a construct threat of validity. To mitigate this threat, we have run the same experiment in the public cloud AWS.

In our experiment, we have considered the API patterns *Request Bundle* and *Rate Limit* and their combinations with *Load Balancing*. We have used a limited and predefined range for $bundle\_size$, $rate\_limit$, and $load\_balancers$. Also, we have used only local rate-limiting and weighted load balancing and ran the experiment 500 times on the private cloud but only 200 times on AWS. This is mainly due to the limited scope and time of the experiment. However, this could lead to an external threat to validity regarding the generalizability of the results. We plan to mitigate this issue by extending our research to other types of *Rate Limit* and *Load Balancing*. We also plan to generalize our results by increasing the $load\_balancers$,

$bundle\_size$ values, and the number of experiment runs on AWS.

A similar threat to validity could be caused by using only one application deployed on one private cloud and one public cloud for the experiment. However, the application is a large open-source application developed using industry best practices by experienced professionals. Also, we plan to further mitigate this issue by running the experiment in other infrastructure environments and settings.

In Section VI-A, we have explained that extended modifications of the Lakeside mutual application have been made to enable request bundling. This might engender the *reliability of treatment implementation* thread of validity. We will validate our models using other applications and settings to waive this threat.

### IX. CONCLUSION AND FUTURE WORK

In this paper, we evaluated the impact of combining the API patterns *Request Bundle*, *Rate Limit*, and *Load Balancing* on the performance and reliability of a cloud-based benchmark based on a realistic application and setting.

To answer **RQ1**, we have first presented a regression model to predict the impact of combining the API patterns above on performance and reliability. Then, we validated that model by experimenting with a real-life application and cloud setting. We repeated the experiment of 130 configurations 500 times in the private cloud, which took more than 4500 hours of runtime and 200 times on AWS, totaling more than 3900 hours of runtime, generating a vast dataset. The data collected are then used to create five distinctive and highly significant models. Besides providing general experimental insights, our models can be used during early architecting. In cases where an environment needs concrete numbers, our approach presents a complete and largely automated method applicable iteratively in different contexts and settings.

As of answering **RQ2**, we used the experiment-generated data to present five models with very high statistical significance. As described in Section VIII, we have found a distinctive impact of those API patterns on performance and reliability. These models could be used in the early phases of architecting distributed systems to evaluate the impact of combining such patterns on those quality properties. As discussed in Section VIII, Model 2 is recommended when combining *Request Bundle* and *Load Balancing*, and Model 3 is best when combining *Rate Limit* and *Load Balancing*. Also, Model 1 is only usable when we have large API messages and *Request Bundle* is not used. The remaining models are suitable otherwise. All the models provide the lowest prediction errors on AWS.

For future work, we plan to add more variables to our model, including those already defined in Section V-A and fixed in advance due to the time and scope of the experiment and include other *Load Balancing* algorithms and API patterns.

## References

[1] C. Surianarayanan, *Essentials of Microservices Architecture : : Paradigms, Applications, and Techniques.* Taylor & Francis, 2019.

[2] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, "Guiding architectural decision making on quality aspects in microservice apis," in *Service-Oriented Computing*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, pp. 73–89.

[3] J. Higginbotham, "Cloud native cloud native api management," 2020. [Online]. Available: https://www.enable-u.nl/wp-content/uploads/2021/03/White-Paper-Cloud-Native.pdf

[4] ——, *Principles of Web API Design: Delivering Value with APIs and Microservices*, ser. Addison-Wesley Signature Series. Pearson Education (US), 2021. [Online]. Available: https://books.google.at/books?id=3dIOzgEACAAJ

[5] A. E. Malki and U. Zdun, "Evaluation of api request bundling and its impact on performance of microservice architectures," in *IEEE International Conference on Services Computing (SCC 2021)*, September 2021. [Online]. Available: http://eprints.cs.univie.ac.at/6898/

[6] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, "Microservice api patterns: Request bundle," 2021. [Online]. Available: https://microservice-api-patterns.org/patterns/quality/dataTransferParsimony/RequestBundle.html

[7] ——, "Microservice api patterns: Rate limit," 2020. [Online]. Available: https://www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/RateLimit

[8] A. El Malki and U. Zdun, "Guiding architectural decision making on service mesh based microservice architectures," in *Software Architecture*, T. Bures, L. Duchien, and P. Inverardi, Eds. Cham: Springer International Publishing, 2019, pp. 3–19.

[9] N. Madden, *API Security in Action*. Manning Publications, 2020. [Online]. Available: https://books.google.at/books?id=hkEKEAAAQBAJ

[10] C. Richardson, *Microservices Patterns: With examples in Java.* Manning, 2018. [Online]. Available: https://books.google.at/books?id=QTgzEAAAQBAJ

[11] Z. Houmani, D. Balouek-Thomert, E. Caron, and M. Parashar, "Enhancing microservices architectures using data-driven service discovery and qos guarantees," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 290–299.

[12] G. J. Mirobi and L. Arockiam, "Dynamic load balancing approach for minimizing the response time using an enhanced throttled load balancer in cloud computing," in *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 2019, pp. 570–575.

[13] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. v. Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, vol. 47, no. 8, pp. 1528–1543, 2021.

[14] D. A. Menascé and V. A. F. Almeida, "Capacity planning for web services; metrics, models, and methods," *Prentice Hall PTR*, vol. 26, no. 1, 2002.

[15] W. Wartika, F. L. Gaol, A. Nugroho, and B. S. Abbas, "Application of bayesian regularization algorithm for evaluation of performance software complexity prediction model based on requirement," *International journal of recent technology and engineering*, vol. 8, no. 3, pp. 2530–2535, 2019.

[16] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *The Journal of systems and software*, vol. 82, no. 1, pp. 3–22, 2009.

[17] J. Wang and C. Zhang, "Software reliability prediction using a deep learning model based on the rnn encoder–decoder," *Reliability engineering & system safety*, vol. 170, pp. 73–82, 2018.

[18] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo, "Microservices integrated performance and reliability testing," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, ser. AST '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 29–39. [Online]. Available: https://doi.org/10.1145/3524481.3527233

[19] H. Wang, Y. Wang, G. Liang, Y. Gao, W. Gao, and W. Zhang, "Research on load balancing technology for microservice architecture," *MATEC web of conferences*, vol. 336, p. 8002, 2021.

[20] Y. Niu, F. Liu, and Z. Li, "Load balancing across microservices," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 198–206.

[21] X. Liu, Y. Jin, Y. Song, and X. Zhao, "Queue-waiting-time based load balancing algorithm for fine-grain microservices," in *Services Computing – SCC 2018*, J. E. Ferreira, G. Spanoudakis, Y. Ma, and L.-J. Zhang, Eds. Cham: Springer International Publishing, 2018, pp. 176–191.

[22] M. Rusek and J. Landmesser, "Time complexity of an distributed algorithm for load balancing of microservice-oriented applications in the cloud," *ITM web of conferences*, vol. 21, p. 18, 2018.

[23] H. Zhu, H. Wang, and I. Bayley, "Formal analysis of load balancing in microservices with scenario calculus," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, vol. 2018-. IEEE, 2018, pp. 908–911.

[24] J. Li, G. Yi, B. Wu, Z. Cao, and X. Xu, "Research and improvement of dynamic highly available load balancing algorithm for microservices," in *2022 International Conference on Artificial Intelligence and Computer Information Technology (AICIT)*, 2022, pp. 1–6.

[25] R. Bhattacharya and T. Wood, "Bloc: Balancing load with overload control in the microservices architecture," in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (AC-SOS)*, 2022, pp. 91–100.

[26] Y. Niu, B. Luo, F. Liu, J. Liu, and B. Li, "When hybrid cloud meets flash crowd: Towards cost-effective service provisioning," in *2015 IEEE Conference on Computer Communications (INFOCOM)*, 2015, pp. 1044–1052.

[27] U. Zdun, M. Stocker, O. Zimmermann, C. Pautasso, and D. Lübke, "Guiding architectural decision making on quality aspects in microservice apis," in *International Conference on Service-Oriented Computing (ICSOC)*. Springer, 2018, pp. 73–89.

[28] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 2: Service integration and sustainability," *IEEE Software*, vol. 34, no. 2, pp. 97–104, 2017.

[29] M. Stocker, O. Zimmermann, U. Zdun, D. Lübke, and C. Pautasso, "Interface quality patterns: Communicating and improving the quality of microservices apis," in *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, 2018, pp. 1–16.

[30] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, "Microservice api patterns: Rate limit," 2020. [Online]. Available: https://www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/RateLimit

[31] A. Gamez-Diaz, P. Fernandez, A. Ruiz-Cortés, P. J. Molina, N. Kolekar, P. Bhogill, M. Mohaan, and F. Méndez, "The role of limitations and slas in the api industry," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1006–1014.

[32] A. Gámez Díaz, P. Fernández Montes, and A. Ruiz Cortés, "Fostering sla-driven api specifications," *JCIS 2018: XIV Jornadas de Ciencia e Ingeniería de Servicios (2018),*, 2018.

[33] A. E. Malki, U. Zdun, and C. Pautasso, "Impact of api rate limit on reliability of microservices-based architectures," in *16th IEEE International Conference on Service-Oriented System Engineering (SOSE2022)*, 2022. [Online]. Available: http://eprints.cs.univie.ac.at/7399/

[34] P. Kumar and R. Kumar, "Issues and challenges of load balancing techniques in cloud computing: A survey," *ACM Comput. Surv.*, vol. 51, no. 6, Feb. 2019. [Online]. Available: https://doi-org.uaccess.univie.ac.at/10.1145/3281010

[35] X. XIE and S. S. Govardhan, "A service mesh-based load balancing and task scheduling system for deep learning applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 843–849.

[36] C. Richardson, "Pattern: Api gateway / backends for frontends," 2020. [Online]. Available: https://microservices.io/patterns/apigateway.html

[37] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, "Microservice api patterns: Rate plan," 2020. [Online]. Available: https://www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/RatePlan

[38] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, and R. Buyya, "Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Software: Practice and experience*, vol. 41, no. 1, pp. 23–50, 2011.

[39] D. J. Dean, H. Nguyen, P. Wang, and X. Gu, "Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-

a-service clouds," in *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'14.  USA: USENIX Association, 2014, p. 16.

[40] J. Starkweather, "Reference category and interpreting regression coefficients in r," Nov. 2018. [Online]. Available: https://it.unt.edu/interpreting-glm-coefficients

[41] "What is pillai trace? (definition & example)," 01 2021. [Online]. Available: https://www.statology.org/pillais-trace/

[42] S. Allright. (2021, June) Mae vs mape, which is best? [Online]. Available: https://stephenallwright.com/mae-vs-mape/