Impact of API Rate Limit on Reliability of Microservices-Based Architectures

Amine El Malki, Uwe Zdun
Research Group Software Architecture
Faculty of Computer Science, University of Vienna
1090, Vienna, Austria
Email: {amine.elmalki|uwe.zdun}@univie.ac.at

Cesare Pautasso
University of Lugano, Faculty of Informatics
Software Institute, Switzerland
Email: cesare.pautasso@usi.ch

Abstract—**Many API patterns and best practices have been developed around microservices-based architectures, such as Rate Limiting and Circuit Breaking, to increase quality properties such as reliability, availability, scalability, and performance. Even though estimates on such properties would be beneficial, especially during the early design of such architectures, the real impact of the patterns on these properties has not been rigorously studied yet. This paper focuses on API Rate Limit and its impact on reliability properties from the perspective of API clients. We present an analytical model that considers specific workload configurations and predefined rate limits and then accurately predicts the success and failure rates of the back-end services. The model also presents a method for adaptively fine-tuning rate limits. We performed two extensive data experiments to validate the model and measured Rate Limiting impacts, firstly on a private cloud to minimize latency and other biases, and secondly on the Google Cloud Platform to test our model in a realistic cloud environment. In both experiments, we observed a low percentage of prediction errors. Thus, we conclude that our model can provide distributed system engineers and architects with insights into an acceptable value for the rate limits to choose for a given workload. Very few works empirically studied the impact of Rate Limit or similar API-related patterns on reliability.**

Index Terms—**API Rate Limit; Microservices; Cloud; Reliability; Modeling.**

I. Introduction

The increasing adoption of microservices-based architectures has posed many challenges regarding API design, including runtime properties such as reliability, availability, and performance [1]. Many API best practices and patterns have been identified [2], [3], [4]. A prominent and often applied practice is *API Rate Limit* [5]. *Rate Limit* aims to protect services from excessive usage and nudge *API clients* into becoming paying customers. It is supposed to increase the reliability of the services from *API clients* perspective. Many techniques and solutions exist that are, e.g., deployed in a Cloud-based architecture [6] or provided as configurable options of an *API Gateway* [7]. However, there is little guidance on which technique to use, how to use them, in which specific circumstances, and using which configurations.

Our study aims to fill this gap by developing an analytical model for accurately predicting the reliability impacts of *Rate Limiting* in microservice-based applications from *API clients* perspective. This model also presents a solid method for adaptively fine-tuning rate limits. We empirically validated this model using a set of microservice-based workload scenarios deployed on a private cloud (to minimize latency and other biases) and on the Google Cloud Platform (GCP) (to test our model in a realistic cloud environment). Note that *Rate Limit* also substantially impacts overall system performance and scalability, but this is beyond the scope of this paper. We simulate real-world API client workloads using different configurations and settings for empirical validation. Our goal is to answer the following research questions:

- **RQ1** How can we accurately predict *API Rate Limit* impacts on reliability properties of a microservices-based system from *API clients* perspective?
- **RQ2** What are the effects of *API Rate Limit* on those reliability properties?

We propose an analytical model to accurately predict the influence of *Rate Limit* on reliability properties from *API clients* perspective. We model each complete simulation run as a Bernoulli process composed of a series of Bernoulli trials, with each of these trials mapping to a predefined rate limit.

To validate our model, we have set up an experiment using a representative, modern Cloud-based architecture based on microservices and the Istio service mesh[1], testing realistic workload scenarios (see Section V-A). Then, we ran the experiment simulating 20 different configurations of API client workloads and repeated each experiment 50 times on the private cloud and GCP. The total runtime of the experiments was more than 2000 hours. We compared the results from the model with the data generated from each of these configurations. We have found that the error is significantly reduced with more runs or stays stable and stands at the maximum values of 17,7% for the private cloud and 16,73% for the GCP experiment, respectively. Those values are substantially below the target prediction error of up to 30% for Cloud-based architectures [8] and explainable with network infrastructure imperfections such as latency and unforeseen errors. The predictions are acceptable since they are close to reality and are sufficient to make broad or early architectural decisions.

The paper is organized as follows. In Section II, we discuss the related work of our study. Then, we give an overview of

[1]https://istio.io/docs/concepts/what-is-istio/

existing *Rate Limit* techniques and the technique we chose in our study in Section III. In Section IV, we present our analytical model. After that, we describe the experiment we have set up to empirically assess the model in Section V. In Section VI, we compare the results from the model and the experiments and discuss threats of validity. Finally, we conclude in Section VII.

II. Related work

Our work focuses on designing a prediction model to help software architects select the best *Rate Limit* strategy for a specific workload and a particular server configuration. Many approaches and methods have been presented for performance and reliability prediction in software architecture, such as performance model derivation and analysis based on model transformations [9] and software quality prediction based on Bayesian Nets [10]. Duzbayev et al. [11] present a runtime prediction model of software architecture QoS specific to queued systems. Adjepon-Yamoah [12] argues that most approaches have limitations, especially in unpredictable environments like Cloud-based architectures. Only a few studies proposed performance models dedicated to Cloud environments [13], [14]. Likewise, our approach is based on an analytical model and empirical data measured in a private cloud and GCP. It is generally Cloud compatible and can be applied to any microservice-based system. Since we cannot cover all microservice technologies and concepts, slight adaptations of measurements and models might be needed in environments with other characteristics, such as other public clouds [15].

Empirical assessments of software architecture reliability have also been the subject of many research papers. Katerina et al. [16] discuss the issue that most software reliability models research papers do not present empirical evaluation and assessment of those models. A simulation tool called Palladio Component Model (PCM) was extended by Brosch et al. [17] to close the gap between (UML-based) modeling of a system and its empirical evaluation. Another simulation tool is Cloudsim [18] which is sometimes applied for service simulations but generally more low-level, i.e., with a focus on cloud resources rather than architectural components. Chaos engineering [19] has been introduced as a discipline-specific to large-scale microservice-based systems for performing reliability testing in real-time. Our proposed reliability model is a novel parametric approach assessed using a widely-used service mesh deployable at a considerable scale. None of these approaches have been applied with an API-centric perspective or focused on specific API patterns to the best of our knowledge. Thus the research question of what the introduction of *Rate Limit* means for the results achieved with such approaches has not yet been answered, and our study aims to close this gap.

Rate Limit and similar techniques have been introduced as best practices or patterns [2], [3], [4], [5], [20] and in the context of Service-Level Agreements [21], [22]. However, the effects of *Rate Limit* have not yet been studied empirically in the scientific literature. This is interesting because *Rate Limit* is a major API practice, and it has a substantial influence not only on reliability and availability properties but also on other critical properties such as security and performance. Thus, measuring the impact of *Rate Limit* and related patterns is crucial to unveil their impacts on those properties and tradeoffs for decision making. We believe it is necessary to study and integrate the effects of such major practices that might influence every single distributed call between clients and servers in a system to improve the results of existing simulators and prediction models.

III. Background on Rate Limiting

Nowadays, service providers usually expose their APIs through API Gateways [23] to provide a central access point and ensure the security and availability of the services. *Rate Limit* [5], [2], [4] is one of the significant patterns that are used to ensure that the API provider is not overwhelmed by excessive requests of *API clients*, either intentional in API abuse or Denial of Service attacks or unintentional, e.g., as a result of system or user errors. *Rate Limit* also solves scalability (and thus overall system performance) and system reliability issues as it can help cope with situations where an API provider unexpectedly receives a burst of client requests. In some cases, this practice might be linked to extra functionalities like limiting access to a specific *API clients* from a specific device (e.g., mobile device) at a particular period. It can also detect *API clients* with shallow frequency access and who intentionally aim to keep connections open for a long time.

Rate Limit is usually linked to a *Rate Plan* [24], which enforces a billing strategy on API usage. Combining those patterns solves most of the problems described lately, as it provides an efficient way to control the behavior of *API clients*. However, this produces an additional computation and resource consumption overhead for authenticating and authorizing *API clients* and keeping track of their usage, especially when the *usage-based pricing* is selected as the base billing policy. Usually, a request is sent by an *API client* along with an *API Key* [25] or information for some other authorization mechanism to verify the legitimate use of the requested API resource.

There exist two types of *Rate Limit*: First, backend rate limiting is enforced by the backends' physical capacity and measured as Transaction Per Second (TPS). Second, application rate-limiting enforces limited requests per period or a quantity of data per user [26]. Our study focuses on the second rate-limiting type by enforcing limited user requests per minute.

Several techniques can enforce rate limiting, such as the token bucket, leaky bucket, or fixed and sliding window practices [6]. When using the token bucket technique, each request is considered a set of operations, and each of these operations consumes one token until reaching the maximum bucket capacity. The leaky bucket technique is similar to the first one, but the rate limit is applied directly to the requests, and any request that exceeds the capacity of the bucket is leaked. Our study covers all these rate-limiting techniques.

IV. Analytical model

In this section, we present our analytical model for the reliability of services with *Rate Limit* from *API clients* perspective, modeled as a Bernoulli process. The Bernoulli process represents an experiment during time T, where each time interval t represents a Bernoulli trial. We define n_t as the number of times we measure the success and failure rates of the system where:

$$\sum^{n_t} t = T \quad with \quad n_t = \frac{T}{t} \tag{1}$$

We model user profiles based on different usage levels from low to high. On client-side, let SU_i denote the share of users with level i of usage, where i ranges from 0 to ∞. Further, to precisely define each of these user groups, let RPM_i denote the number of requests per minute the users in SU_i send. Let TU denote the total number of users that send these requests. The total client requests TR^T in the time interval T can then be calculated as:

$$TR^T = T \cdot TU \cdot \sum_{i=0}^{\infty} SU_i \cdot RPM_i \cdot LF_i \tag{2}$$

Such that,

$$\sum_{i=0}^{\infty} SU_i = 100\%$$

And LF_i is the *load frequency occurrence* corresponding to SU_i. It represents the proportion of time T where the workload is being executed and is calculated using the formula:

$$LF_i = \frac{\frac{LOAD_i}{RPM_i}}{\frac{LOAD_i}{RPM_i} + SLEEP_i} \tag{3}$$

Where $LOAD_i$ is the number of concurrent requests sent by users in SU_i during time T and $SLEEP_i$ is the idle or no execution time corresponding to the users in SU_i during time T.

On the server-side, let $TR^T_{ratelimit}$ denote the total number of requests that failed due to rate limiting, and let $TR^T_{failure}$ indicate the total number of requests that failed for other reasons than rate-limiting during the time interval T. Then $TR^T_{success}$, the total number of succeeded requests, assuming that the reliability of the network infrastructure is very high, can be calculated as:

$$TR^T_{success} = TR^T - (TR^T_{failure} + TR^T_{ratelimit}) \tag{4}$$

Additionally, let $FR^t_{failure}$ and $FR^t_{ratelimit}$ denote the failure rates, during the time interval t, corresponding to $TR^T_{failure}$ and $TR^T_{ratelimit}$, respectively. $SR^t_{success}$ denotes the success rate, during the time interval t. The respective number of failed requests $TR^t_{failure}$, succeeded requests $TR^t_{success}$, and failed requests due to rate limiting $TR^t_{ratelimit}$, during the time interval t can be written as:

$$TR^t_{failure} = t \cdot FR^t_{failure} \tag{5}$$

$$TR^t_{success} = t \cdot SR^t_{success} \tag{6}$$

$$TR^t_{ratelimit} = t \cdot FR^t_{ratelimit} \tag{7}$$

We consider each time interval t as a Bernoulli trial. Let $P^t_{failure}$ denote the probability of request failure, $P^t_{ratelimit}$ the probability of request failure due to rate limiting, and $P^t_{success}$ the probability of a succeeded request. The corresponding expected numbers of failed requests $E^T_{failure}$, requests failed because of rate limiting $E^T_{ratelimit}$, and succeeded requests $E^T_{success}$, during the time interval T, can be written as:

$$E^T_{failure} = n_t \cdot P^t_{failure} \tag{8}$$

$$E^T_{ratelimit} = n_t \cdot P^t_{ratelimit} \tag{9}$$

$$E^T_{success} = n_t \cdot P^t_{success} \tag{10}$$

where:

$$P^t_{success} = 1 - (P^t_{failure} + P^t_{ratelimit}) \tag{11}$$

Thus, the total number of failures $TR^T_{failure}$, failures due to rate limiting $TR^T_{ratelimit}$, and succeeded requests $TR^T_{success}$ can be expressed as:

$$TR^T_{failure} = \sum^{n_t} E^T_{failure} \cdot TR^t_{failure} \tag{12}$$

$$TR^T_{ratelimit} = \sum^{n_t} E^T_{ratelimit} \cdot TR^t_{ratelimit} \tag{13}$$

$$TR^T_{success} = \sum^{n_t} E^T_{success} \cdot TR^t_{success} \tag{14}$$

which can be rewritten using equations (1), (5), (6), (7), (8), (9) and (10) as follows:

$$TR^T_{failure} = T \cdot \sum^{n_t} P^t_{failure} \cdot FR^t_{failure} \tag{15}$$

$$TR^T_{ratelimit} = T \cdot \sum^{n_t} P^t_{ratelimit} \cdot FR^t_{ratelimit} \tag{16}$$

$$TR^T_{success} = T \cdot \sum^{n_t} P^t_{success} \cdot SR^t_{success} \tag{17}$$

Using equations (2), (3), (4), (15), (16) and (17), the resulting formula including both client-side and server-side parameters is the following:

$$\begin{aligned} \overbrace{\sum^{n_t} \frac{P^t_{failure}}{TU} \cdot FR^t_{failure}}^{non-rate-limiting\ failure\ rate} \quad = \\ workload \left\{ \sum_{i=0}^{\infty} SU_i \cdot RPM_i \cdot \frac{\frac{LOAD_i}{RPM_i}}{\frac{LOAD_i}{RPM_i} + SLEEP_i} \right. \\ - \underbrace{\sum^{n_t} \frac{P^t_{success}}{TU} \cdot SR^t_{success}}_{success\ rate} - \underbrace{\sum^{n_t} \frac{P^t_{ratelimit}}{TU} \cdot FR^t_{ratelimit}}_{rate-limiting\ failure\ rate} \end{aligned} \tag{18}$$

Such that:

TABLE I: Definition of Parameters

Independent variable	**Description**
SU_i	Percentage of the user group with usage level i in the total users share.
RPM_i	Number of requests per minute the users in the share of users of level i usage send.
$LOAD_i$	Number of concurrent requests executed by level i usage users during time T.
$SLEEP_i$	Idle or no execution time for level i usage users during time T.
TU	Total number of users in the experiment.
T	Total duration of the experiment in minutes.
t	Trial duration in minutes.
Dependent variable	**Description**
NRL_FR	Per-minute number of requests failed for other reasons than rate limiting.
RL_FR	Per-minute number of requests failed due to rate limiting.
SR	Per-minute number of requests succeeded.

$$\begin{aligned} NRL\_FR &= \sum^{n_t} \frac{P^t_{failure}}{TU} \cdot FR^t_{failure} \\ SR &= \sum^{n_t} \frac{P^t_{success}}{TU} \cdot SR^t_{success} \\ RL\_FR &= \sum^{n_t} \frac{P^t_{ratelimit}}{TU} \cdot FR^t_{ratelimit} \end{aligned} \tag{19}$$

The resulting equation shows that the *non-rate-limiting failure rate* NRL_FR might decrease if both the *success rate* SR and *rate-limiting failure rate* RL_FR increase. Indeed, *Rate Limit* can considerably decrease NRL_FR. However, if not configured well, it can also lead to the decrease of SR and thus increase NRL_FR. Table I summarizes the variables used in our model.

V. Empirical evaluation

We have followed the eight methodological principles for reproducible Cloud performance evaluation [27]. Principle 1 emphasizes repeating the experiments many times and quantifying the results. Indeed, we repeated the experiments 50 times until we reached a reasonably low prediction error. As per Principle 2, each of these experiments has to run a different configuration, which is the case in our work, since we have run 50 different configurations (10 workload configurations $\times$ five rate limit configurations) using a private cloud and 50 additional configurations using GCP. Section V-C describes in detail the hardware and software configurations used in the experiments, which is required for fulfilling Principle 3. To satisfy Principle 4 and enable the reproducibility of our study, we published all artifacts as open-access[2]. In Section V-E, we analyze the results using box plots as recommended for Principle 5. Then, we evaluate the data's accuracy by comparing the experiments' results to our model, as described in Section VI, and required as Principle 6. As per Principle 7, all the measurement units are explicitly displayed. Principle 8 is about reporting cloud costs which we do for the GCP experiment in Table II (not applicable for the private cloud experiment).

[2]To enable reproducibility of our study, all artifacts used in the experiment, including code and data, are published as an open-access artifact on Zenodo: https://doi.org/10.5281/zenodo.6560270.

TABLE II: GCP cost details (in Euros)

Service	**Cost**	**Discounts**	**Promotions and others**	**Subtotal**
Compute Engine	416,56	-3,72	-243,22	169,62
Cloud Logging	74,2	0	-13,65	60,55
Kubernetes Engine	94,99	-94,99	0	0
Total	585,75	-98,71	-256,87	230,17

A. Benchmark workload scenarios

Existing benchmarks for studying application-level microservices, such as TeaStore [28], are based on examples implemented by researchers. We essentially followed the same approach, but to design realistic workload scenarios, we decided to perform our experiments using workload scenarios derived from typical kinds of interactions observed in a prior empirical study of open-source microservice architectures realized by practitioners [29] (e.g., we tried to model similar call chains length, number of database interactions, processing steps, etc.). In this context, we observed that typical E-commerce or business interactions such as authentication, product selection, basket interactions, and payment occur relatively often in open-source systems. Thus, we used such interactions for our workload scenarios. We plan to extend our study to other benchmarks and applications in the future.

Many open-source systems showcase specific technologies, such as specific tracing or deployment technologies, or microservice concepts, such as event sourcing. To avoid the substantial bias by those specific technologies or concepts in the implementations, we decided not to experiment based on one of those open-source systems. Instead, we created a benchmark based on those workload scenarios to model typical interactions in microservice-based business applications. In particular, we (1) modeled eight typical interactions in four services covering authentication, product, basket, and payment functionality as benchmark scenarios based on the steps repeatedly occurring in the open-source systems, and (2) implemented those scenarios from scratch as plain RESTful HTTP services. Based on the mentioned empirical study [29], we believe those scenarios to be representative of typical interactions in many business systems. We developed our RESTful microservices by using the widely used Java/Spring Boot[3] technology. Each microservice exposes a set of REST API endpoints that execute the corresponding operations on its dedicated Mongo[4] database instance.

A typical workload scenario in our experiments starts with the number – predefined earlier during workload initiation (see Tables III and IV) – of *API clients* of type *customer* sending *POST* login API calls to the *authentication* microservice. For instance, then, each of these customers views the list of available products using the provided *GET* API endpoint in the *product* microservice and adds his reserved product (for the reason described earlier) to the basket by calling *POST* API Endpoint in the *basket* microservice. After that, each customer proceeds to payment and checkout by sending a *POST* request to the *payment* microservice. The latter sends *DELETE*

[3]https://spring.io/projects/spring-boot
[4]https://www.mongodb.com

requests to both *basket* and *product* microservices to remove the purchased product from the customer's basket and from the list of available products to purchase, respectively. Numerous executions of such scenarios can be executed concurrently during our experiment runs.

B. Architecture overview

Service meshes have been studied as an infrastructure layer that manages communication in microservice architectures and performs numerous *Proxy Tasks* including API rate-limiting (see [30]). It has also been pointed out that service meshes provide numerous *Central Services* like collecting telemetry, traces, and metrics, which considerably reduce the complexity of data collection and analysis. We have chosen to use the Istio service mesh as a representative modern microservice platform in our current study. To handle *Ingress Communication*, we have decided to use a *Front Proxy* as an API Gateway [23], as we intend to assess rate limiting on the traffic coming from outside the service mesh. We have also selected the *Multi-Cluster Support* option for our private cloud, as it is simple and easy to administrate.

We have deployed our benchmark workload scenarios on the Istio service mesh. We aim to provide a realistic setup for a modern microservice deployment configuration. Istio is installed using its multi-cluster option with one shared control plane[5]. Each cluster represents a Minikube[6] instance running a Kubernetes engine[7] under a preconfigured Virtual Machine (VM). The benchmark workload scenarios run on four Java-based microservices: authentication, product, basket, and payment. Each microservice is a Docker[8] image deployed on a single cluster along with a dedicated Mongo database Docker image and exposes REST API endpoints accessible via the HTTP Protocol. The internal communication between these microservices is established through Sidecars [30] over a private network.

We have integrated the Kong API Gateway [31] into Istio to intercept incoming client communication. The reason for choosing this API Gateway is that it is open source and can be seamlessly integrated with Istio. Also, it natively supports rate limiting, which reduces its implementation complexity considerably. In addition, this way, we can avoid the threat to validity that – by implementing our rate limiting solution – we could have introduced some unrealistic implementation variant or other bias into the used rate limiting solution. Finally, the Kong API Gateway already provides *API metrics* [32], using the time series kong_http_status, that can be directly collected and visualized using tools like Prometheus[9] or Grafana[10].

In GCP, we used the same setup described previously but without the *Multi-Cluster Support* option. Instead, we used only one Google Kubernetes Cluster (GKE[11]) composed of four nodes pool.

[5]https://archive.istio.io/v1.4/docs/setup/install/multicluster/shared-vpn/

[6]https://minikube.sigs.k8s.io/docs/

[7]https://kubernetes.io

[8]https://hub.docker.com

[9]https://prometheus.io

[10]https://grafana.com

[11]https://cloud.google.com/kubernetes-engine

C. Configuration details

We have run the experiments on a private cloud, composed of 4 Ubuntu[12] 18.04.4 LTS Virtual Machines (VMs) running on vSphere[13] 6.7 environment. Each of these VMs runs a Minikube instance (version ranging from 1.8.2 to 1.9.2) with eight dedicated CPU cores Intel Xeon(R)™CPU E5-2650 v4 @ 2.20GHz and 20 GB of system memory. Each Minikube instance runs Kubernetes engine version 1.14.2 and Istio version 1.4.3.

The central Minikube instance hosts the central *Control Plane*, Kong Ingress Controller[14] version 0.8.0, the authentication microservice built using Java version 8 and its dedicated Mongo database (latest version used). The *Ingress Controller* accesses the 3 remaining microservices through YAML-defined Kubernetes endpoints[15] and Ingress Rules [33]. The data are collected using Prometheus version 11.0.3 and visualized using Grafana version 5.0.8.

On the client-side, nine physical desktop clients and six virtual desktop clients are used to sending HTTP requests to the private cloud. All desktop clients run Ubuntu 18.04.4 LTS, and eight physical desktops have 4 CPU cores Intel Core™2122 i3-2120T CPU @ 2.60GHz with 8 GB of system memory. The remaining physical desktop client has 4 CPU cores Intel Core™2122 i5-4670S CPU @ 3.10GHz, and 30 GB of system memory. All virtual desktop clients have 2 CPU cores Intel® Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory. An experiment involves three desktop clients, each dedicated to a specific category of users in terms of the number of requests per minute sent and a specific percentage share as described in Table I.

In GCP, we used one Cluster composed of 4 identical e2-medium VMs with eight dedicated Intel Broadwell vCPUs and 16 GB of system memory. The Cluster runs these newer Kubernetes engine 2.6, Istio 1.6.14, and Kong Gateway 2.5. We also used three virtual desktop clients identical to the ones described above.

D. Experimental setup

Each experiment starts by launching a workload script on three selected desktop clients, each corresponding to a specific category of users as described previously. These workloads execute the necessary setup requirements on the server-side, creating passwords and a product database by calling the corresponding microservices using their appropriate API endpoints. Tables III and IV list the different workload configurations (C1 ...C10) used in the experiments. Note that the following description of the values used in those workload configurations is specific to the current application and cloud settings and cannot be generalized to other experimentation settings without further adjustments. The value of $SU_{i\leq 10}$, reflecting

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

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

[14]https://github.com/Kong/kubernetes-ingress-controller

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

TABLE III: Workload configurations for experiments using the private cloud

Client parameter	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
TU	100	150	100	150	100	150	100	150	100	150
$SU_{i\leq10}$	50 %	46 %	50 %	50 %	48 %	50 %	50 %	47 %	54 %	53 %
$RPM_{i\leq10}$	5	3,12	5	2	4	10	4	5	4	7
$SU_{i>10}$	40 %	46 %	30 %	35 %	35 %	30 %	45 %	46 %	40 %	42 %
$RPM_{i>10}$	16,17	13	22	11	12	11	24	10,78	24,8	11
$SU_{i\geq50}$	10 %	8 %	20 %	15 %	17 %	20 %	5 %	7 %	6 %	5 %
$RPM_{i\geq50}$	80	100	140	62	85	150	260	70	180	150

TABLE IV: Workload configurations for experiments using GCP

Client parameter	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**	**C10**
TU	100	150	100	150	100	150	100	150	100	150
$SU_{i\leq10}$	50%	50%	55%	55%	50%	45%	50%	47%	55%	50%
$RPM_{i\leq10}$	3,12	3,01	3,08	2,97	3,11	3,02	3,13	3	3,17	2,99
$SU_{i>10}$	40%	40%	35%	35%	35%	35%	45%	42%	40%	30%
$RPM_{i>10}$	28,01	35,09	27,54	24,43	35	33,24	25,68	33,15	30,75	34,29
$SU_{i\geq50}$	10%	10%	10%	10%	15%	20%	5%	11%	5%	20%
$RPM_{i\geq50}$	303,49	450	295,61	400	430	500	342,88	420	745,18	380

the most common type of users that never overloads the system and has *API freemium* subscriptions only, is generally set as the greatest percentage of at least 46% of total users. The corresponding values of $RPM_{i\leq10}$ are usually less than ten requests per minute. Then, the value of $SU_{i>10}$, representing a lower proportion of users that might overload the system and possesses *API premium* subscriptions, fluctuates between 30% and 46%, without exceeding the value of $SU_{i\leq10}$. The respective values of $RPM_{i>10}$ are estimated at least ten requests per minute. Finally, the value of $SU_{i\geq50}$, reflecting the undesired set of users that usually creates bursts, crashes the system, and don't possess any valid API subscription, fluctuates between 5% and 20%, without exceeding the value of $SU_{i>10}$. These users can generate at least 50 requests per minute. On server-side, a configuration script is launched with the predefined variables T and t described in Table I. The script executes mainly two tasks:

- Enabling (and disabling) rate-limiting with a predefined value in the following: 5, 25, 75, 150, Infinite (for disabled rate limiting).
- Collecting data by querying the Prometheus database and storing them in JSON[16] format.

E. Results analysis: Experiments using the private cloud

To cover a wide range of possible workload configurations, the workload-related independent variables described in Table I are varied by respecting the constraints detailed in Section V-D. These workload configurations are listed in Table III.

The results of the experiments are collected and displayed as boxplots in Figures 1, 2 and 3. From these plots, we can gather a few general conclusions, such as: These plots show that as the rate limit increases for all experiments, both SR and NRL_FR generally increase. In contrast, RL_FR naturally decreases. However, we observe a slight decrease of SR for C4, C6, C7, and C10, and a considerable reduction of NRL_FR for C5 and C9 when we disable rate limiting. Then, a rate limit value of approximately 75 requests per minute should be selected for C4 and C6, where NRL_FR starts decreasing with a negative impact on RL_FR, especially for C4. However, for C7 and C10, it is more appropriate to apply a more significant rate limit value of around 150 requests per minute, where NRL_FR starts decreasing and RL_FR only slightly increases. On the other hand, for C5 and C9, rate-limiting only makes sense when applying a value of fewer than 25 requests per minute. However, in this case, the RL_FR increase is too high, and SR dramatically decreases, especially for C5. Applying rate-limiting for C1 and C3 is an obvious choice for a value of around 150 requests per minute, with a bit of impact on SR and a very slight increase in RL_FR for C3. Finally, disabling rate limiting is best for C2; unfortunately, we cannot make any visual decision regarding C8.

To evaluate the accuracy of our model, we compare the data collected from the experiments with the data calculated using the model. To have an approximation of the probabilities $P^t_{success}$, $P^t_{failure}$ and $P^t_{ratelimit}$, we have run each experiment many times and selected the probabilities that best fit our model. We plan to improve our proposed model and omit the last step in future work. The resulting data are shown in Table V. First of all, we notice that $P^t_{failure}$ is highest (more than 83%) when rate limiting is disabled. Also, $P^t_{success}$ is highest when rate limiting is enabled, except for C9, where its variation is hardly noticeable. This shows that *Rate Limit* plays an essential role in increasing system reliability properties, but it may also degrade these properties if not correctly configured.

F. Results analysis: Experiments using GCP

To generalize the results of our experiment on the private cloud, we have run it on GCP as well. This is to avoid any bias that might affect the results from using a private cloud, test our model on a realistic public cloud environment, and open the door to experimentation in other public clouds and environments. We used the workload configurations described in Table IV and compared the data collected from the experiments with the data calculated using the model in Table VI. We have used the workload configurations in the private cloud with few adjustments.

From Table VI we conclude that for C6, enabling rate limiting does not make sense since it drastically decreases SR and increases RL_FR, with only a slight decrease in NRL_FR. However, we notice a significant decline of NRL_FR when applying a rate limit of 150 requests per minute for all remaining configurations, although the increase in RL_FR and decrease in SR are still very high. Unlike when using the private cloud, both $P^t_{success}$ and $P^t_{failure}$ have the highest values when rate limiting is disabled, which confirms the latter conclusions. This is mainly due to the GCP's high reliability against the selected workload configurations. Notice that box plots corresponding to experiments using GCP are not shown here for space reasons and are included in the replication package[17].

[16]https://www.json.org/json-en.html

[17]https://doi.org/10.5281/zenodo.6560270

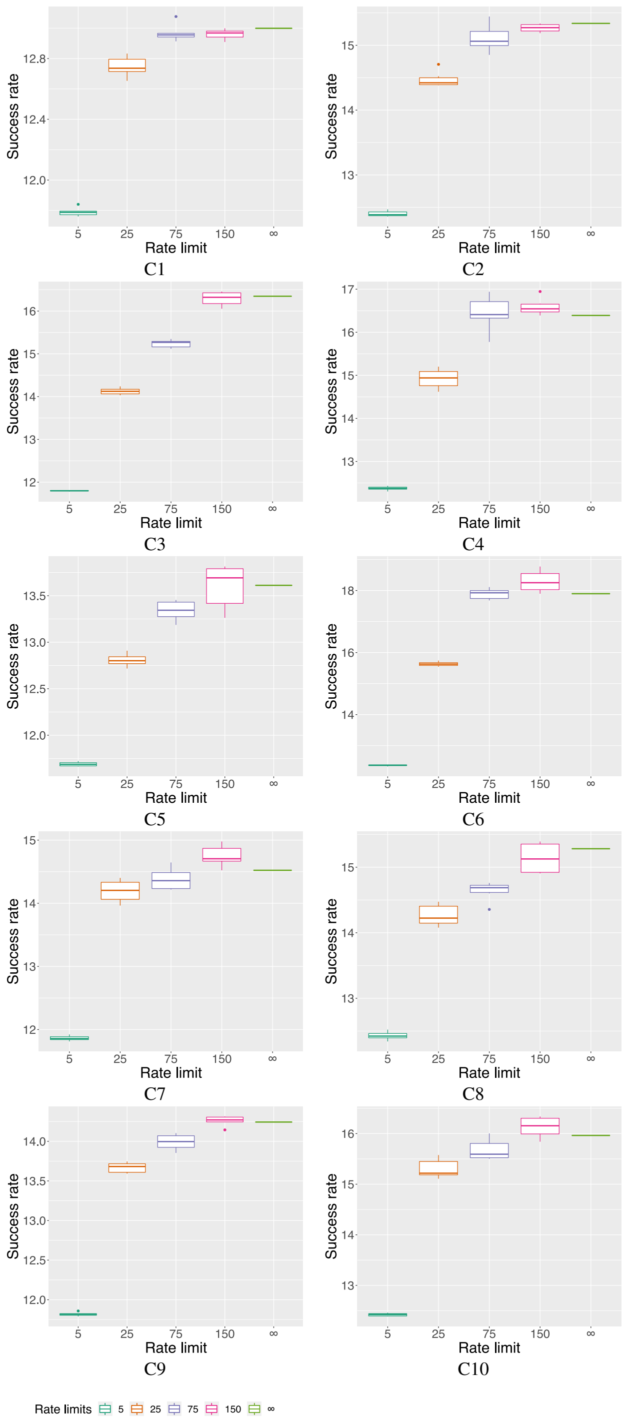

Fig. 1: SR results from all the experiments (50 runs)

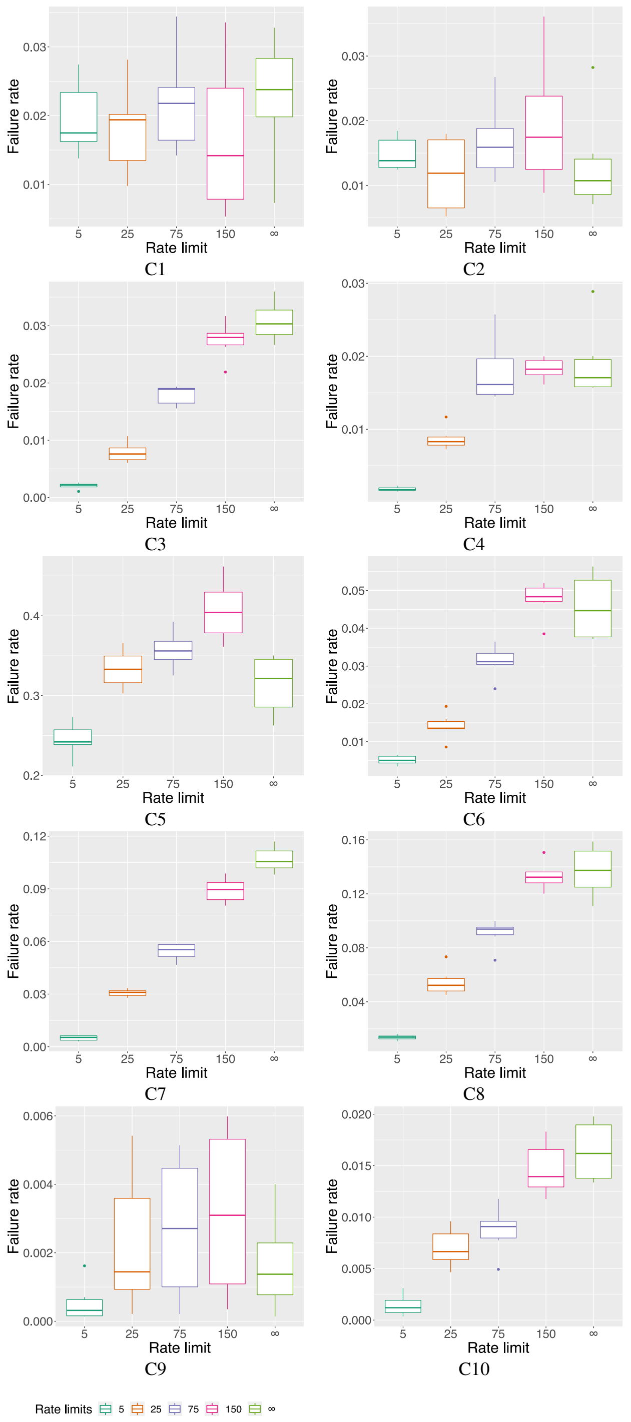

Fig. 2: NRL_FR results from all the experiments (50 runs)

VI. Prediction error and threats to validity

To evaluate the accuracy of our model, we calculate its prediction error using the Mean Absolute Percentage Error (MAPE) [34]. Then, we discuss the threats to validity.

A. Mean Absolute Percentage Error

We have run each experiment, using the workload configurations described in Tables III and IV, more than 50 times to show that the prediction error gets smaller with the increase of the number of runs and then stays relatively stable. A single run of an experiment takes about 2.5 hours without considering the intermittent crashes and recovery times. Let n_{config} be the number of configurations in our experiments, equal to 50 using the private cloud and 30 using GCP. Also, let $rate^{i}_{model}$ and $rate^{i}_{experiment}$ denote the model and experiment values

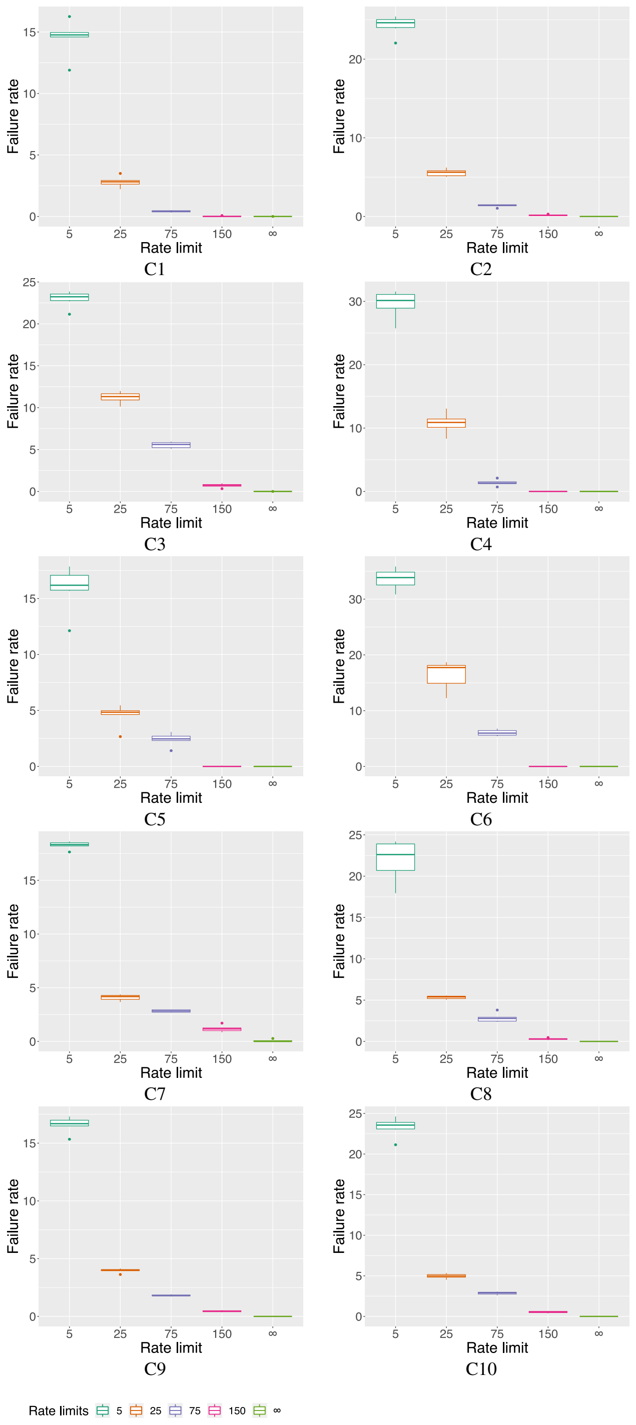

Fig. 3: RL_FR results from all the experiments (50 runs)

respectively, corresponding to SR, NRL_FR and RL_FR for each configuration i. Then, we have:

$$MAPE = \frac{100\%}{n_{config}} \cdot \sum_{i=1}^{n_{config}} \left| \frac{rate^i_{model} - rate^i_{experiment}}{rate^i_{experiment}} \right|$$

As seen in Table VII, the overall prediction errors decrease with the increasing number of runs for the experiment on the private cloud and reach stable values relatively early for the GCP experiment. After 50 runs we reach a prediction error of 17,7% for $SR\%$, 2% for $NRL\_FR\%$, and 16,89% for $RL\_FR\%$ using a private cloud and 16,73% for $SR\%$, 3,69% for $NRL\_FR\%$, and 16,24% for $RL\_FR\%$ using GCP. All the errors are below the target prediction error of up to 30% for Cloud-based architectures [8], and thus explainable with network infrastructure imperfections such as latency and unforeseen errors. Thus, the predictions are acceptable since they are close to reality and are sufficient to make broad architectural decisions.

TABLE V: Results from the model and the experiments using the private cloud (50 runs)

Config	RL	Probability %			Model			Experiment		
		$P^t_{success}$	$P^t_{failure}$	$P^t_{ratelimit}$	SR	NRL_FR	RL_FR	SR	NRL_FR	RL_FR
C1	5	16,61	66,85	16,54	2,96	0,05	4,91	7,01	0,05	8,66
	25	16,65	67,03	16,32	6,16	0,05	1,72	7,64	0,04	1,65
	75	16,59	68,61	14,8	7,73	0,05	0,14	7,74	0,05	0,22
	150	16,56	83,44	0	7,87	0,05	0	7,73	0,05	0
	∞	16,52	83,48	0	7,85	0,07	0	7,73	0,07	0
C2	5	16,62	66,77	16,61	2,64	0,02	4,77	5,8	0,02	9,68
	25	16,634	66,776	16,59	5,48	0,019	1,58	6,63	0,02	2,22
	75	16,648	67,252	16,1	6,9	0,026	0,51	6,89	0,03	0,53
	150	16,58	83,42	0	7,4	0,04	0	6,92	0,04	0
	∞	16,61	83,39	0	7,41	0,02	0	6,97	0,03	0
C3	5	16,61	66,709	16,681	3,71	0,005	9,22	8,33	0,005	13,79
	25	16,6	66,68	16,72	8,36	0,02	4,56	9,72	0,02	6,75
	75	16,56	66,63	16,81	10,3	0,04	2,59	10,36	0,04	3,35
	150	16,56	83,44	0	12,85	0,08	0	10,99	0,08	0
	∞	16,55	83,45	0	12,84	0,09	0	11,01	0,09	0
C4	5	16,591	66,717	16,692	2,33	0,003	5,11	5,78	0,003	11,85
	25	16,543	66,607	16,85	5,16	0,01	2,27	6,77	0,01	4,36
	75	16,59	83,41	0	7,41	0,034	0	7,4	0,036	0
	150	16,585	83,415	0	7,41	0,04	0	7,45	0,04	0
	∞	16,581	83,419	0	7,4	0,04	0	7,37	0,04	0
C5	5	13,63	69,59	16,81	2,98	0,61	6,16	6,78	0,61	9,61
	25	14,92	68,98	16,1	6,25	0,83	2,67	8,03	0,83	2,65
	75	15,1	68,1	16,8	8,42	0,87	0,47	8,41	0,88	1,46
	150	14,5	85,5	0	8,49	1,27	0	8,22	1,25	0
	∞	15	85	0	8,78	0,98	0	8,5	0,96	0
C6	5	19,84	63,468	16,692	3,53	0,008	7,07	6,9	0,008	13,47
	25	16,47	66,63	16,9	6,6	0,02	3,99	7,02	0,02	6,67
	75	16,1	65,6	18,3	7,97	0,05	2,59	7,74	0,05	2,65
	150	16,52	83,48	0	10,52	0,09	0	8,1	0,09	0
	∞	16,52	83,48	0	10,51	0,09	0	7,94	0,09	0
C7	5	16,24	66,96	16,8	2,97	0,01	8,41	8,18	0,01	11,05
	25	16,32	65,87	17,81	9,37	0,07	1,94	9,59	0,07	2,62
	75	16,4	66,6	17	9,8	0,13	1,46	9,75	0,13	1,73
	150	16,31	66,79	16,9	10,31	0,21	0,86	9,91	0,21	0,72
	∞	16,2	83,8	0	11,07	0,32	0	9,72	0,32	0
C8	5	15,27	67,41	17,32	2,24	0,02	4,87	5,34	0,02	9,13
	25	15,9	65,35	18,75	5,45	0,09	1,59	6,26	0,09	2,4
	75	16,24	83,76	0	6,95	0,18	0	6,54	0,18	0
	150	16,1	83,9	0	6,89	0,24	0	6,67	0,27	0
	∞	16	84	0	6,85	0,28	0	6,69	0,28	0
C9	5	16,654	66,677	16,669	3,09	0,001	7,63	8,37	0,001	9,95
	25	16,653	66,656	16,691	9,23	0,005	1,49	9,47	0,005	2,38
	75	16,652	66,648	16,7	9,71	0,006	1,01	9,67	0,006	1,09
	150	16,654	66,646	16,7	10,43	0,007	0,29	9,83	0,008	0,27
	∞	16,659	83,341	0	10,72	0,005	0	9,82	0,005	0
C10	5	16,62	66,69	16,69	3,34	0,002	5,06	5,81	0,002	9,33
	25	16,54	66,46	17	6,41	0,01	1,99	6,92	0,01	2,03
	75	16,55	66,25	17,2	7,19	0,01	1,21	7,08	0,014	1,18
	150	16,61	83,39	0	8,38	0,03	0	7,28	0,03	0
	∞	16,6	83,4	0	8,37	0,03	0	7,21	0,03	0

B. Threats to validity

In our study, we have simulated workloads from only a limited number of desktop clients per experiment, simultaneously generating the workload configurations of users' requests, as described in Section V-C. This could be a threat to validity since, in real-world scenarios, we usually have one desktop

TABLE VI: Results from the model and the experiments using GCP (50 runs)

Config	RL	Probability %			Model			Experiment		
		$P^t_{success}$	$P^t_{failure}$	$P^t_{ratelimit}$	SR	NRL_FR	RL_FR	SR	NRL_FR	RL_FR
C1	5	16,62	66,711	16,669	0,27	0	11,22	0,21	5E-5	14,88
	25	16,42	66,889	16,691	1,1	0,001	10,39	0,99	0,001	13
	75	16,51	66,79	16,7	2,96	0,01	8,52	2,76	0,01	10,92
	150	16,61	66,68	16,71	5,76	0,004	5,72	5,26	0,006	7,65
	∞	16,63	83,37	0	11,47	0,025	0	8,83	0,02	0
C2	5	16,62	66,711	16,669	0,15	0	10,67	0,14	0	13,93
	25	16,28	67,029	16,691	0,65	0,001	10,13	0,65	0,001	12,57
	75	16,378	66,922	16,7	1,89	0,015	8,91	1,83	0,015	11,04
	150	16,563	66,727	16,71	3,68	0,004	7,13	3,59	0,004	8,54
	∞	16,647	83,353	0	10,81	0,012	0	6,47	0,012	0
C3	5	16,62	66,711	16,669	0,24	0	10,79	0,21	2E-5	13,76
	25	16,473	66,836	16,691	1,14	0	9,89	1,01	0,001	12,31
	75	16,51	66,79	16,7	2,97	0,012	8,04	2,81	0,012	9,92
	150	16,61	66,68	16,71	5,71	0,005	5,31	5,23	0,005	6,56
	∞	16,63	83,37	0	11,01	0,02	0	8,52	0,02	0
C4	5	16,62	66,711	16,669	0,15	0	9,05	0,14	0	11,86
	25	16,34	66,969	16,691	0,68	0,001	8,52	0,66	0,001	10,79
	75	16,37	66,93	16,7	1,84	0,018	7,35	1,89	0,018	9,24
	150	16,586	66,704	16,71	3,55	0,002	5,65	3,63	0,002	6,4
	∞	16,655	83,345	0	9,2	0,006	0	5,81	0,007	0
C5	5	16,62	66,711	16,669	0,24	0	12,28	0,2	0	12,28
	25	16,367	66,942	16,691	0,96	8E-4	11,56	0,97	8E-4	11,56
	75	16,45	66,85	16,7	2,45	0,012	10,06	2,69	0,012	10,06
	150	16,583	66,707	16,71	4,58	0,002	7,94	5,15	0,002	7,94
	∞	16,656	83,344	0	12,52	0,008	0	8,94	0,008	0
C6	5	16,62	66,711	16,669	0,16	0	11,2	0,14	E-4	15,23
	25	16,3	67,009	16,691	0,73	9E-4	10,62	0,66	8E-4	14,21
	75	16,39	66,91	16,7	1,88	0,012	9,46	1,88	0,013	12,22
	150	16,56	66,73	16,71	3,56	0,002	7,79	3,53	0,002	9,83
	∞	16,662	83,338	0	11,35	0,003	0	6,67	0,003	0
C7	5	16,62	66,711	16,669	0,30	0	10,38	0,20	0	13,36
	25	16,473	66,836	16,691	1,23	6E-4	9,45	1	6E-4	12,09
	75	16,49	66,81	16,7	2,71	0,013	7,96	2,76	0,012	9,51
	150	16,61	66,68	16,71	4,94	0,002	5,73	5,29	0,003	5,75
	∞	16,646	83,354	0	10,67	0,01	0	8,96	0,01	0
C8	5	16,62	66,711	16,669	0,15	0	9,82	0,14	0	9,97
	25	16,31	66,999	16,691	0,69	0,001	9,28	0,66	0,001	10,79
	75	16,37	66,93	16,7	1,87	0,017	8,08	1,89	0,018	9,24
	150	16,581	66,709	16,71	3,64	0,002	6,32	3,63	0,002	6,4
	∞	16,655	83,345	0	9,96	0,006	0	5,81	0,007	0
C9	5	16,62	66,711	16,669	0,25	0	9,66	0,21	0	13,2
	25	16,436	66,873	16,691	0,99	9E-4	8,93	1	9E-4	12,66
	75	16,39	66,91	16,7	1,73	0,01	8,17	2,72	0,01	10,38
	150	16,525	66,765	16,71	2,65	0,003	7,26	5,01	0,003	7,08
	∞	16,65	83,35	0	9,91	0,009	0	9,42	0,009	0
C10	5	16,62	66,711	16,669	0,15	0	11,12	0,14	0	15,65
	25	16,345	66,964	16,691	0,66	0	10,6	0,66	0,001	14,84
	75	16,41	66,89	16,7	1,80	0,009	9,46	1,85	0,009	12,94
	150	16,556	66,734	16,71	3,47	0,002	7,79	3,60	0,003	10,68
	∞	16,85	83,42	0	11,21	0,05	0	6,71	0,06	0

TABLE VII: Calculated MAPE for all configurations up to 50 runs

Runs	Private Cloud			GCP		
	SR%	NRL_FR%	RL_FR%	SR%	NRL_FR%	RL_FR%
10	21,92	822,73	59,66	17,01	61,69	16,41
20	21,18	94,51	23,45	17,09	16,49	16,41
30	16,85	8,31	16,33	16,38	16,58	17,07
40	20,04	24	17,65	16,29	14,05	16,90
50	17,7	2	16,89	16,73	3,69	16,24

client per user. In our case, the shared network bandwidth would have reduced the values configured for $RPM_{i\leq10}$, $RPM_{i>10}$ and $RPM_{i\geq50}$. We have monitored these values in real-time and adjusted them accordingly to mitigate this issue.

The benchmark workload scenarios we used in our study are based on four homogeneous microservices, where we have only one service and one database per server. We have not considered the case where we have a heterogeneous environment, and we did not replicate the services to increase their availability, for example. Thus, our model does not consider server-side details like the number of services and type of services (such as database services, third-party services, and so on). Also, our model assumes that computational, and network resources are continuously available with a low risk for errors (e.g., issues with locks, authorization/authentication) and bottlenecks, which is not valid in the real world. To exclude this threat, server-side configurations and these resources, including potential errors, need to be integrated into our study. Please note that our model focuses on the API level only; from an abstract point of view, it does not matter whether a service is unreliable or slow because of the database or the service implementation. While this mitigates this threat to a large extent, it cannot be excluded that specific backend effects would influence the results on *Rate Limit* that can be observed at the API level.

A similar threat is that we used a limited number of E-commerce or business-related benchmark workload scenarios that might not represent non-business workflows. There is the threat that different applications and scenarios would yield substantially different experimental results. However, we plan to extend our study to other benchmarks and applications in the future. Again, we are only interested in the properties observable at the API level. Nonetheless, the model might have to be extended or changed to be applied for substantially different kinds of microservice technologies and concepts. However, as many microservice technologies and concepts exist, it is virtually impossible to cover them in one experiment. Hence the study is based on plain (vanilla) RESTful services, and we have chosen the E-commerce/business domain as enterprise applications with many users are often rate limited.

VII. Conclusion and Future Work

Concerning **RQ1**, we have developed an analytical model to provide distributed system engineers and architects insights into which specific value to choose as a rate limit given a workload situation. We have empirically validated the model using 50 different configurations in a private cloud and additional 50 different configurations in GCP with more than 2000 hours of runtime. We found that the prediction error generally decreases with more runs or stays stable, and stands at 16,89% for $RL\_FR\%$, 2% for $NRL\_FR\%$ and 17,7% for $SR\%$ using a private cloud and 16,73% for $SR\%$, 3,69% for $NRL\_FR\%$, and 16,24% for $RL\_FR\%$ using GCP, after 50 runs. These prediction errors are reasonably close to reality, given some indeterministic effects such as network latency that we have not included in our model.

Concerning **RQ2**, we conclude that applying *Rate Limiting* may increase the reliability properties of APIs, given a specific workload situation. However, finding the right balance between improving the success rate and keeping the failure rate at a certain minimum level is not trivial. We have provided the first step towards a solution by providing an empirically tested analytical model to accurately predict a configuration's and workload's impact on those variables. This model also presents a solid method for adaptively fine-tuning rate limits. Since we cannot cover all microservice technologies and concepts, we provide an approach that can be extended to other technologies and concepts. Hence, the study is based on plain RESTful services.

In future work, we aim to improve the analytical model by varying other parameters like the total duration of the experiment T and trial duration t and including different server configurations, workloads, benchmark scenarios, and resource management. Also, we plan to generalize the model by applying it to other technologies, applications, and architectures.

Acknowledgments. This work was supported by: FWF (Austria Science Fund) project API-ACE, no. I 4268.

References

[1] P. Francesco, I. Malavolta, and P. Lago, in *Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption*, 04 2017, pp. 21–30.

[2] 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.

[3] 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.

[4] 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.

[5] 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

[6] Google, "Techniques for enforcing rate limits," 2020. [Online]. Available: https://cloud.google.com/solutions/rate-limiting-strategies-techniques

[7] R. Paprocki, "How to design a scalable rate limiting algorithm," 2017. [Online]. Available: https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm/

[8] 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.

[9] A. Di Marco and R. Mirandola, "Model transformation in software performance engineering," in *Quality of Software Architectures*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, vol. 4214, pp. 95–110.

[10] Ł. Radliński, "A framework for integrated software quality prediction using bayesian nets," in *Computational Science and Its Applications - ICCSA 2011*, B. Murgante, O. Gervasi, A. Iglesias, D. Taniar, and B. O. Apduhan, Eds. Springer Berlin Heidelberg, 2011, pp. 310–325.

[11] N. Duzbayev and I. Poernomo, "Runtime prediction of queued behaviour," in *Quality of Software Architectures*, C. Hofmeister, I. Crnkovic, and R. Reussner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 78–94.

[12] D. E. Adjepon-Yamoah, "cloud-atam: Method for analysing resilient attributes of cloud-based architectures," in *Software Engineering for Resilient Systems*, I. Crnkovic and E. Troubitsyna, Eds. Cham: Springer International Publishing, 2016, pp. 105–114.

[13] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, "Efficiency analysis of provisioning microservices," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2016, pp. 261–268.

[14] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–32. [Online]. Available: https://doi.org/10.1145/3297663.3310309

[15] C. Laaber, J. Scheuner, and P. Leitner, "Software microbenchmarking in the cloud. how bad is it really?" *Empirical Software Engineering*, vol. 24, no. 4, pp. 2469–2508, 2019.

[16] K. Goseva-Popstojanova, M. Hamill, and R. Perugupalli, in *Large empirical case study of architecture-based software reliability*. IEEE, 2005, pp. 10 pp.–52.

[17] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Parameterized reliability prediction for component-based software architectures," in *Research into Practice – Reality and Gaps*, G. T. Heineman, J. Kofron, and F. Plasil, Eds. Springer Berlin Heidelberg, 2010, pp. 36–51.

[18] 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.

[19] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, May 2016.

[20] 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/

[21] 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.

[22] 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.

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

[24] 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

[25] ——, "Microservice api patterns: Api key," 2020. [Online]. Available: https://www.microservice-api-patterns.org/patterns/quality/qualityManagementAndGovernance/APIKey

[26] J. Simpson, "Everything you need to know about api rate limiting," 2019. [Online]. Available: https://nordicapis.com/everything-you-need-to-know-about-api-rate-limiting/

[27] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. Von Kistowski, A. Ali-eldin, C. Abad, J. N. Amaral, P. Tůma, and A. Iosup, "Methodological principles for reproducible performance evaluation in cloud computing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[28] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '18, September 2018.

[29] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Metrics for assessing architecture conformance to microservice architecture patterns and practices," in *18th International Conference on Service Oriented Computing (ICSOC 2020)*, December 2020.

[30] 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.

[31] K. Kevin Chen, "Kong ingress controller and service mesh: Setting up ingress to istio on kubernetes," March 2020. [Online]. Available: https://kubernetes.io/blog/2020/03/18/kong-ingress-controller-and-istio-service-mesh/

[32] H. Bagdi, "Integrate kong ingress controller with prometheus/grafana," 2020. [Online]. Available: https://github.com/Kong/kubernetes-ingress-controller/blob/master/docs/guides/prometheus-grafana.md

[33] ——, "Getting started with kong ingress controller," 2020. [Online]. Available: https://github.com/Kong/kubernetes-ingress-controller/blob/master/docs/guides/getting-started.md

[34] K. S. Trivedi and A. Bobbio, *Reliability and Availability Engineering: Modeling, Analysis and Applications*. Cambridge University Press, 2017.