Automated Pattern-Based Recommendation for Improving API Operation Performance and Reliability in Cloud-Based Architectures

Amine El Malki and Uwe Zdun

Research Group Software Architecture, Faculty of Computer Science
University of Vienna, Währinger Straße 29, A-1090, Vienna, Austria
Email: {amine.elmalki|uwe.zdun}@univie.ac.at

***Abstract*—The extensive use of APIs as the entry point to many Cloud-based applications has created challenging problems, especially concerning API quality properties such as performance and reliability. API best practices and patterns, such as bundling requests, rate limiting, or load balancing, have been proposed to solve these challenges. Unfortunately, no study investigating the impact of existing API practices and patterns on such quality properties exists beyond informal recommendations. In this paper, we fill this gap by proposing a pattern-based, automated recommendation approach to improve the performance and reliability of API operations. We provide a benchmark suite based on a realistic open-source microservice application to enable the automatic generation of comprehensive decision tree models. These models are then processed to generate API design recommendation algorithms to improve API operations regarding performance and reliability stored in catalogs for reuse. We validate our algorithms using extensive data sets generated by running the benchmark on a private cloud and AWS. For both environments, based on the decision tree models automatically generated from the measured data, API design recommendation algorithms have been calculated using our approach.**

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

I. Introduction

The widespread use of cloud-based applications accessible through APIs has led to many challenges regarding the quality of these APIs, especially when deployed on a large scale. To cope with such challenges, API design has become an essential building block in achieving the desired level of quality properties for the APIs. API design prevents degrading such quality properties and concerns refactoring APIs based on runtime usage observations. While already an extensive set of API patterns and best practices has been proposed [1], [2], [3], [4], today, no support beyond informal guidance on API quality exists. This is particularly unsatisfactory for measurable properties such as performance and reliability. In addition, it is often unclear which effect a *combination* of different patterns would have when just following such informal guidance. Thus, API designers need to improve their APIs based on these informal recommendations by trial and error, which is costly and likely does not lead to optimal results.

Some notable API patterns specifically focusing on improving performance and reliability properties of APIs are *Request Bundle*, *Rate Limit*, and *Load Balancing*:

- *Request Bundle* [1], [5] is achieved by bundling multiple message chunks into one combined message to avoid chatty communications between API clients and servers. As such, message chunks can get large enough to produce complexity and congestion on both the client and server side, which should be avoided [6].
- *Rate Limit* [1], [7] limits the number of requests that can be handled by API servers during a specific amount of time to avoid problems caused by excessive and abusive API clients' requests [8]. This pattern often makes use of *Front* or *Edge proxies* [9] and may combine both of them.
- *Load Balancing* is usually achieved using *API Gateways* [10], [11] and aims to balance the load between multiple instances in the backend [12]. Many algorithms supporting *Load Balancing* exist but must be selected carefully to avoid problems like bottlenecks and performance degradation.

This paper aims to study the possible combinations of such API patterns and, based on the gathered data, automatically generate recommendations to enable refactoring or adaptions of API design to optimize these combinations concerning their performance and reliability impact. This work is based on a prior work in which possible pattern combinations have been studied by providing an initial empirical study resulting in a regression model, which had the aim to quantify the impact of combining *Request Bundle*, *Rate Limit*, and *Load Balancing* patterns on API performance and reliability. However, such a model is static and does not consider parameters like the application type and usage period. For this reason, in this paper, we have substantially extended these empirical studies to provide a novel automatic and dynamic API design recommendation approach based on decision tree modeling, which is the core contribution of this paper. We aim to answer the following research questions:

- **RQ1** What are the different decision criteria that should

be considered to evaluate the impact of combining *Request Bundle*, *Rate Limit*, and *Load Balancing* patterns on API performance and reliability?
- **RQ2** How can we select the best combinations of these patterns and API operations in terms of performance and reliability impacts?

We provide a benchmark suite based on a real-life application deployed on modern cloud-based infrastructure. In the prototype implementation for experiments, the Istio[1] service mesh and the Envoy[2] proxy were used as a widely used combination of representative modern infrastructure tools that enable to combine features like *Rate Limit* and *Load Balancing* [9].

We use the data generated in the experiment to automatically construct decision tree models for each combination of those API patterns to answer those questions. Then, we develop API design recommendation algorithms to select the best combination of API patterns and operations in terms of performance and reliability impacts using these decision trees. These algorithms are stored in catalogs for reuse. Please note that such reuse of our empirical results makes sense in similar settings (business application on a similar private cloud or AWS configuration) to get initial estimates. Also, we have derived generic recommendations on pattern combinations that were yet unknown or not yet empirically proven (see Section VI). We propose an API design recommendation pipeline that essentially re-runs our benchmark in such a new setting or configuration to get exact recommendation algorithms or estimates in vastly different configurations. Our approach automatically generates and validates the necessary decision trees and API design recommendation algorithms in this context.

The paper is organized as follows. Section II describes related work. Our research methods are then summarized in Section III. Next, we describe our approach in Section IV. In Section V, we analyze the data and develop the API design recommendation algorithm. Then, we provide a discussion and threats to validity in Section VI. Finally, we conclude and describe future work in Section VII.

II. Related work

Many studies have suggested different approaches for dealing with self-adaptive systems to improve quality properties like performance and reliability. Self-adaptation was a topic that emerged in early Service-oriented Architecture (SOA) research as a way to increase resiliency [13]. Garlan et al. [14] presented Rainbow as a reusable infrastructure for system developers to realize self-adaptation cost-effectively. Layaida et al. [15] presented PLASMA as a component-based framework for building self-adaptive multimedia applications. The framework is based on a hierarchical composition and reconfiguration model. Adapt! [16] plug-in architectures were proposed as an external adaptation solution to achieve full self-adaptation functionality. The middleware approach [17] was introduced to enable this externalization through application and service mirroring. However, all of these approaches are general and not API-centric. They are also not applicable to highly versatile environments like Cloud-based architectures.

The decision tree modeling approach has also been the subject of many studies in the literature. Rathore et al. [18] presented a decision tree-based approach to predict software faults. Another similar military-based research developed a decision tree model to identify fault-prone software modules [19]. Chui et al. [20] provided an approach based on decision tree modeling considering the time of data collection to improve performance prediction rate without affecting prediction accuracy. Decision tree modeling has also been combined with other techniques, like Bayes in the transportation world, to build a lane-changing assistance simulation model [21] to increase the safety of passengers. These extensive use cases and others for decision tree modeling to predict the performance and reliability of diverse applications show that the approach followed in this paper is relevant.

Several prior studies tried measuring the real impact of API patterns on reliability and performance [22], [23]. Lawin et al. [24] evaluated the effect of using GraphQL and REST technologies on the performance of massively used API. They figured out that the first technology is the better choice when data frequently change, while the second stands out when there is a need for multiple requests. While these studies and others have proposed robust models and interesting results to derive API configuration recommendations based on specific settings and technologies, those models and recommendations lack important properties like self-adaptation and easy reuse. Also, they treat each API pattern individually, not in combination with others, and deal with specific technologies. In our study, we try to fill this gap by proposing a complete self-adapted and agnostic API design recommendation pipeline.

III. Research methods

Our approach is based on a dataset generated in a prior study which we describe first. Then, we provide details about the benchmark used to evaluate the model.

A. Regression model

The study considered two types of variables: numerical and categorical. The first type of variables includes rpm, api_users, and $bundle\_size$, collected at the client-side, $rate\_limit$, which (if enabled) is set at the server-side, and $total\_time$ and $failure\_rate$ which are collected at the server-side. 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 (see Table I).

B. Benchmark description

As no benchmark for studying API operations existed, a novel benchmark was developed for the study based on the

[1]https://istio.io/

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

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 ranges from 100 to 1200 (in increments of 100).
$request\_bundle$	Categorical variable indicating whether request bundling is used or not.
$bundle\_size$	The request bundle size ranges from 10 to 50 (in increments of 10).
$load\_balancing$	Categorical variable indicating whether load balancing is used or not.
$method$	Categorical variable indicating the method executed by API users (get, delete, update, or create).
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.

open-source Lakeside Mutual[3] application that is realistic and developed based on real-life experience and makes use of several microservice API patterns such as *Request Bundle* [5], [6] (already implemented for the API operation: get customers).

Based on a manual analysis of which other API operations could benefit from *Request Bundle*, the application was extended to support *Request Bundle* in the following API operations: $create$, $delete$, and $update$. The implementation follows the same style as the existing bundle implementation in Lakeside Mutual. The application is composed of four Java-based microservices packaged using Maven[4], containerized using Docker[5] and deployed on the Istio service mesh. Containers orchestration is realized using Kubernetes[6]. Database per service model is used when *Load Balancing* is inactive. Otherwise, the database is shared by the service and its replica. Istio is used since it provides various functionalities out of the box like *Rate Limit* and *Load Balancing* [9]. On the client side, the workload is defined using Shell[7] script covering all possible combinations of the API patterns under study.

To evaluate the impact of the combination of those API patterns on performance and reliability, we present the benchmark described in Table III generated from earlier experimental data. In the benchmark, the authors of this study define $bundle\_size$ ranging from 10 to 50, which reflects the bundle size, in case *Request Bundle* is active. They also define $rate\_limit$ ranging from 100 to 1200, in case *Rate Limit* is active. Otherwise, $rate\_limit = \infty$. $load\_balancing$ is a categorical value indicating whether *Load Balancing* is used or not.

The benchmark was run on a private Cloud provided by the University Data Center and AWS[8]. Infrastructure details are provided in Table II. Both experiments were executed covering 130 different configurations 500 times on the private cloud and 200 times on AWS, totaling 130000 repetitions, more than 4500 hours of runtime in the private cloud, and 52000 repetitions on AWS, and more than 3900 hours of runtime. A small portion of the data collected is summarized by calculating the mean and shown in Table III. The full dataset is provided in an online appendix[9].

[3]https://github.com/Microservice-API-Patterns/LakesideMutual
[4]https://maven.apache.org
[5]https://hub.docker.com
[6]https://kubernetes.io
[7]https://www.shellscript.sh
[8]https://aws.amazon.com

IV. APPROACH DESCRIPTION

Modern applications expose many APIs accessible by API clients and interoperable with other third-party applications. However, part of these APIs is rarely (and sometimes never) exploited by the API clients and applications. The regression model described briefly in Section III-A could be used to find the right combination of API patterns to improve performance and reliability. However, this approach is static and does not include parameters like the application type and usage period. Since this kind of parameter is hard to consider in the model, we further developed an approach introducing decision trees using the model parameters and the data generated in the experiment described in Section III-B. We then automatically generate an API design recommendation algorithm from the decision trees that could enable APIs to adapt at runtime. Details are provided in the next section.

Figure 1 represents the API design recommendation benchmark pipeline that continuously delivers API improvement recommendations to a target platform, described in Step 1. The benchmark is configured using the target cloud setting during this initial step, and the application is deployed. In Step 2, the benchmark suite is executed to generate data which are then collected along with this target configuration to verify whether it exists in the configurations catalog in Step 3. If that is the case, the benchmark selects the corresponding algorithm with the collected data to generate API recommendations in Step 4. Otherwise, data are extracted and analyzed in Step 3 to create decision trees, which are then used to develop new algorithms with the collected data. After validation, these algorithms are used to reconfigure API and added to the algorithms catalog in Step 4. The configurations and algorithms catalogs are continuously synchronized and updated for each newer version of the target application and cloud setting.

V. API ANALYSIS AND RECOMMENDATIONS

In this section, we provide details of our API analysis approach. Then, we use this analysis approach to generate API

[9]https://doi.org/10.5281/zenodo.7691350

Algorithm 1 API design recommendation (AWS)

```
for each method ∈ methods do
    if api_users < 14 then
        bundle_size = 10 and
        range(rate_limit, [150, ∞]) and
        combination = LB ∧ RB ∧ RL
    else if api_users < 24 then
        bundle_size = 20 and
        range(rate_limit, [150, ∞]) and
        combination = RL ∧ RB
    else
        range(bundle_size, {30, 40, 50}) and
        range(rate_limit, [150, ∞]) and
        combination = LB ∧ RB ∧ RL
    end if
end for
```

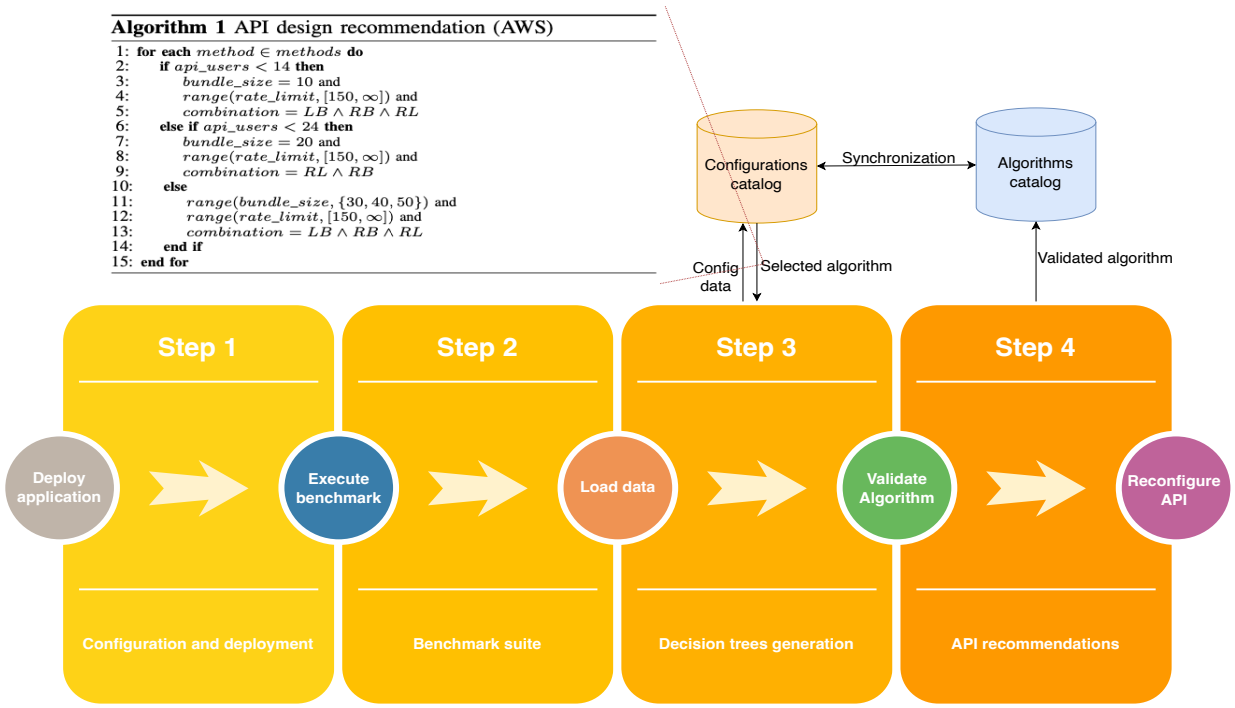

Fig. 1. API design recommendation pipeline

TABLE II
INFRASTRUCTURE DETAILS OF THE EXPERIMENTS

	Private cloud	**AWS**
Cluster	11 Ubuntu[10] 18.04.5 LTS Virtual Machines (VMs) installed on vSphere[11] 6.7 environment.	Amazon EKS Cluster[12] composed of four EC2[13] instances pool.
Clients	Ubuntu 18.04.5 LTS virtual desktops are used to inject HTTP requests into the private cloud. Each of the virtual desktops has 2 CPU cores Intel® Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.	
Minikube	1.20.0	
Kubernetes	1.20.2	
Istio	1.10.0	
Number of clusters	2	1

design recommendation algorithms from the private cloud and AWS data collected using the proposed benchmark.

A. API analysis

The data generated using the experimentation described in Section III-B are analyzed utilizing the package rpart[14] provided by R language[15]. This package builds classification and regression models using a two-phase procedure. First, it selects the variables that best split the data into two groups and recursively repeats this until the subgroup is too small or there is no further improvement. Secondly, the resulting tree is reevaluated by selecting the subtree with the lowest risk. We have generated 14 decision trees covering all API patterns and method combinations for each cloud type (private cloud and AWS). The decision trees in Figures 2 to 7 are used to generate the algorithms for AWS. The remaining figures are for the private cloud.

[14]https://cran.r-project.org/web/packages/rpart/rpart.pdf
[15]https://www.r-project.org

Let's take as an example the case when combining *Rate Limit*, *Request Bundle*, and *Load Balancing* on AWS in Figures 2 to 7. The first figure shows the case where all the patterns are combined. The performance here is best when $api\_users < 14$ with $total\_time = 853ms$. Regarding reliability, Figure 3 adjusts the value of $rate\_limit$ to ≥ 150 for maximum reliability. When $api\_users < 24$, the best combination is *Request Bundle* and *Rate Limit* in Figure 6, where $total\_time$ ranges between $730ms$ and $1270ms$ depending on the value of rpm. The range values of $rate\_limit$ do not change as described in Figure 7. The remaining and extreme case with $api\_users \geq 24$ is illustrated by the first figure with a maximum value of $total\_time = 1534ms$ (not considering the case where $rpm < 207$ with the probability of occurrence $\approx 0\%$). Those conditions are generated for specific applications and cloud settings in a certain period. To do that in real-time, we need to construct algorithms based on those decision trees that must be run continuously, as explained in Section IV. The following Section describes our algorithms in

TABLE III
SAMPLE DATA: LB=LOAD BALANCING, RB=REQUEST BUNDLE, OP=OPERATION, SZ=BUNDLE SIZE, TT=TOTAL TIME(MS), FR=FAILURE RATE(%)

				Rate limit														
				100		200		300		400		500		600		700		∞
LB	RB	OP	SZ	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT	FR	TT
Yes	Yes	Get	50	173,32	31,91	222,56	15,96	239,01	10,64	249,17	7,98	386,46	12,76	588,87	10,64	814,30	9,12	280,98
			40	100,60	49,66	161,85	24,83	182,33	16,55	194,88	12,42	235,96	19,87	446,65	16,55	683,52	14,19	235,89
			30	79,,00	53,54	127,72	26,77	145,45	17,85	155,48	13,39	249,52	21,42	507,21	17,85	725,92	15,30	204,03
			20	94,84	29,97	109,34	21,78	116,87	14,52	122,38	10,89	317,50	14,71	541,16	14,52	746,28	12,44	164,43
			10	64,18	31,20	75,23	15,60	79,02	10,40	81,55	7,80	277,31	12,48	501,92	10,40	690,79	8,92	125,88
		Update	50	1694,05	47,87	2347,08	23,93	2829,87	15,96	3202,48	11,97	3417,89	9,57	3440,80	7,98	3062,27	13,68	2206,88
			40	690,41	74,17	1501,51	37,25	2054,19	24,83	2412,76	18,62	2616,78	14,90	2686,67	12,42	2333,14	21,24	1557,92
			30	443,77	80,31	1237,47	40,16	1659,41	26,77	1998,13	20,08	2182,29	16,06	2227,49	13,39	1925,88	22,95	1264,42
			20	917,49	44,96	1227,73	22,48	1404,52	21,78	1689,10	16,33	1826,23	13,07	1841,12	10,89	1621,98	15,76	836,63
			10	510,90	46,81	686,79	23,40	873,89	15,60	1035,38	11,70	1131,87	9,36	1180,39	7,80	1032,16	13,37	451,94
		Delete	50	1878,47	47,87	2802,29	23,93	3008,76	15,96	3039,55	11,97	3041,84	9,57	3021,96	7,98	2685,27	13,68	2152,24
			40	544,65	74,17	1611,92	37,25	2084,14	24,83	2218,97	18,62	2262,23	14,90	2268,20	12,42	1980,38	21,24	1484,82
			30	280,74	80,31	1340,56	40,16	1600,04	26,77	1674,18	20,08	1687,21	16,06	1693,75	13,39	1471,86	22,95	1316,66
			20	823,02	44,96	1194,01	22,48	1037,57	21,78	1108,92	16,33	1140,91	13,07	1148,58	10,89	1033,16	15,76	934
			10	516,88	46,81	691,40	23,40	742,61	15,60	734,47	11,70	721,58	9,36	707,89	7,80	633,57	13,37	459,28
		Create	50	1804,28	47,87	2541,28	23,93	2676,87	15,96	2702,85	11,97	2700,02	9,57	2677,38	7,98	2505,30	13,68	2686,54
			40	763,51	74,17	1625,92	37,25	1934,97	24,83	2019,48	18,62	2060,32	14,90	2065,04	12,42	1862,34	21,24	1867,13
			30	413,74	80,31	1292,95	40,16	1548,42	26,77	1584,90	20,08	1609,71	16,06	1609,33	13,39	1439,68	22,95	1580,91
			20	905,41	44,96	1196,48	22,48	1033,29	21,78	1086,63	16,33	1123,11	13,07	1115,16	10,89	1056,33	15,76	1084,51
			10	456,18	46,81	670,42	23,40	733,00	15,60	735,22	11,70	723,65	9,36	700,85	7,80	652,40	13,37	551,97
	No	Get	50	689,96	64,67	855,37	71,25	1206,95	49,61	1406,60	37,39	1275,43	42,84	1397,88	48,68	1788,78	42,63	1643,10
			40	563,35	64,15	703,43	69,93	985,75	48,07	1137,54	36,34	1039,13	41,90	1182,27	47,54	1530,61	41,37	1302,57
			30	463,46	64,03	574,90	65,59	777,26	44,37	883,25	33,75	812,65	39,80	922,42	44,36	1224,45	38,30	1034,79
			20	270,45	96,51	348,13	81,03	487,17	54,30	575,15	40,73	559,09	51,88	671,44	54,16	955,22	46,54	719,87
			10	148,16	95,85	224,22	75,96	279,79	50,64	314,54	37,98	299,71	49,56	437,46	50,64	689,05	43,41	388,14
		Update	50	918,42	48,69	1068,77	71,84	1459,03	71,25	2212,97	55,82	2662,99	44,65	2758,25	37,39	2469,20	39,00	2923,57
			40	789,48	48,77	936,57	71,34	1253,94	69,93	1831,40	54,08	2172,24	43,26	2239,12	36,34	2010,26	38,12	2198,13
			30	696,78	48,49	757,38	71,10	1017,43	65,59	1455,35	49,91	1708,43	39,93	1742,78	33,75	1570,58	35,86	1767,02
			20	429,95	97,55	491,03	95,25	668,80	81,03	966,40	61,09	1147,01	48,87	1227,86	40,73	1091,59	48,84	1182,39
			10	205,06	96,96	263,27	93,86	431,99	75,96	597,72	56,97	690,20	45,58	710,35	37,98	628,98	46,41	597,28
		Delete	50	895,41	48,69	824,58	71,84	1482,69	71,25	3084,93	55,82	4308,49	44,65	4349,28	37,39	3810,93	39,00	3528,01
			40	821,04	48,77	722,60	71,34	1225,45	69,93	2583,96	54,08	3486,26	43,26	3492,44	36,34	3063,40	38,12	2633,99
			30	744,96	48,49	606,85	71,10	1213,10	65,59	2252,40	49,91	2846,37	39,93	2785,86	33,75	2448,73	35,86	2363,27
			20	240,22	97,55	257,92	95,25	676,70	81,03	1169,42	61,09	1554,57	48,87	1730,57	40,73	1519,18	48,84	1577,66
			10	131,79	96,96	133,15	93,86	344,33	75,96	600,10	56,97	791,03	45,58	878,57	37,98	771,18	46,41	786,28
		Create	50	895,41	48,69	824,58	71,84	1482,69	71,25	3084,93	55,82	4308,49	44,65	4349,28	37,39	3810,93	39,00	4486,12
			40	821,04	48,77	722,60	71,34	1225,45	69,93	2583,96	54,08	3486,26	43,26	3492,44	36,34	3063,40	38,12	3137,15
			30	744,96	48,49	606,85	71,10	1213,10	65,59	2252,40	49,91	2846,37	39,93	2785,86	33,75	2448,73	35,86	2801,52
			20	240,22	97,55	257,92	95,25	676,70	81,03	1169,42	61,09	1554,57	48,87	1730,57	40,73	1519,18	48,84	1815,61
			10	131,79	96,96	133,15	93,86	344,33	75,96	600,10	56,97	791,03	45,58	878,57	37,98	771,18	46,41	899,42

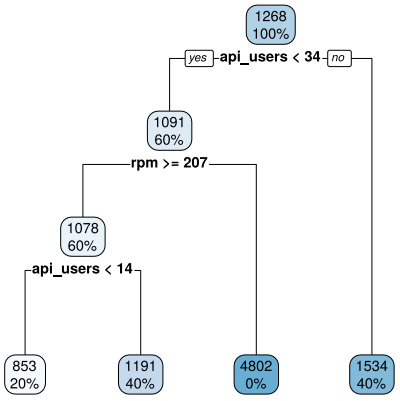

Fig. 2. Load balancing, request bundle & rate limit combined: Performance decision tree on AWS (milliseconds)

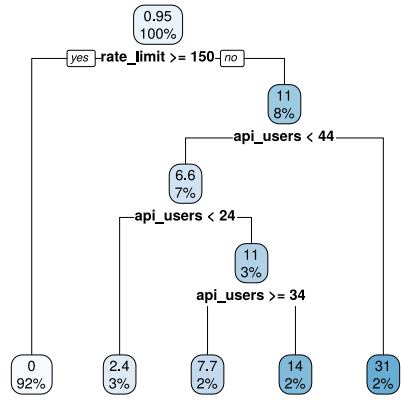

Fig. 3. Load balancing, request bundle & rate limit combined: Reliability decision tree on AWS (% failure)

more detail.

B. API design recommendation algorithms

Let LB, RB, and RL denote *Load Balancing*, *Request Bundle*, and *Rate Limit*, respectively. Each possible $combination$ is described as:

$$combination \in \{LB \wedge RB, LB \wedge RL, RL \wedge RB, LB \wedge RB \wedge RL\}$$

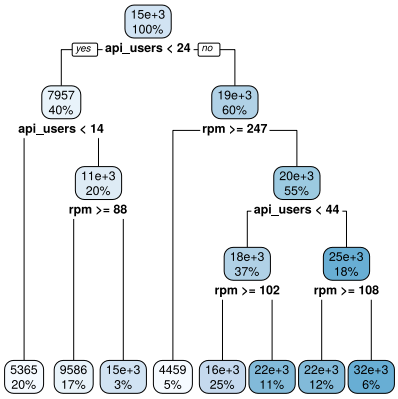

Fig. 4. Load balancing & rate limit combined: Performance decision tree on AWS (milliseconds)

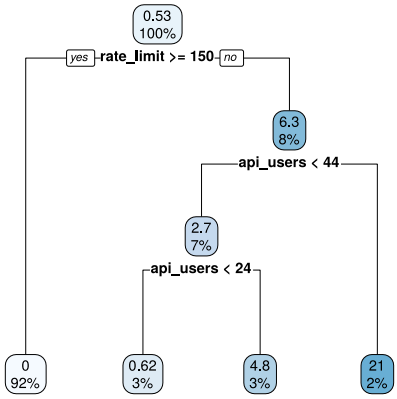

Fig. 7. Rate limit & request bundle combined: Reliability decision tree on AWS (% failure)

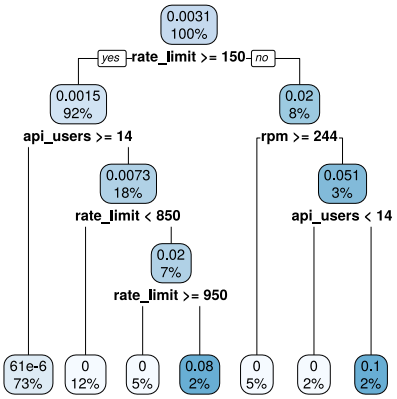

Fig. 5. Load balancing & rate limit combined: Reliability decision tree on AWS (% failure)

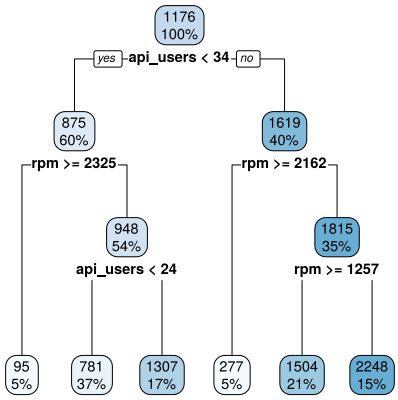

Fig. 8. Load balancing, request bundle & rate limit combined: Performance decision tree on the private cloud (milliseconds)

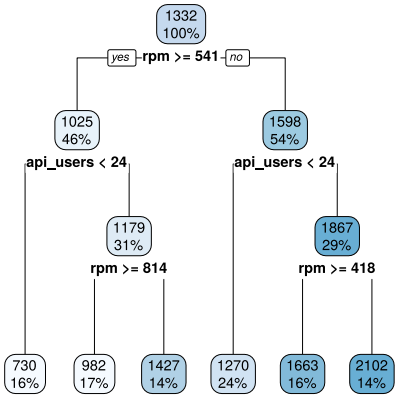

Fig. 6. Rate limit & request bundle combined: Performance decision tree on AWS (milliseconds)

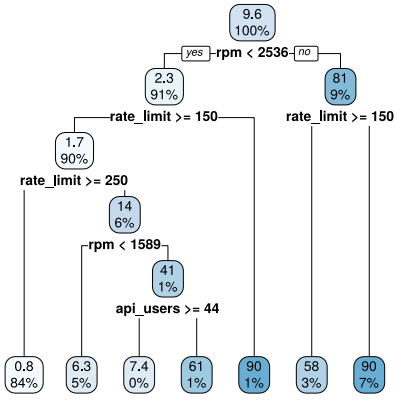

Fig. 9. Load balancing, request bundle & rate limit combined: Reliability decision tree on the private cloud (% failure)

Algorithms 1 and 2 are generated from the data collected on both AWS and the private cloud, respectively, to determine the best API patterns and methods combinations. The values corresponding to $rate\_limit$, rpm, and api_users are generated

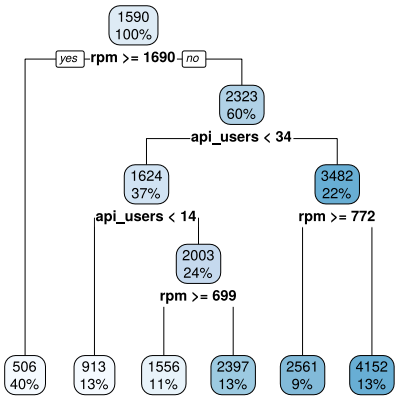

Fig. 10. Load balancing & rate limit combined: Performance decision tree on the private cloud (milliseconds)

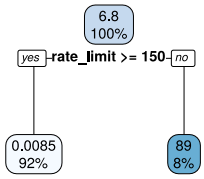

Fig. 13. Rate limit & request bundle combined: Reliability decision tree on the private cloud (% failure)

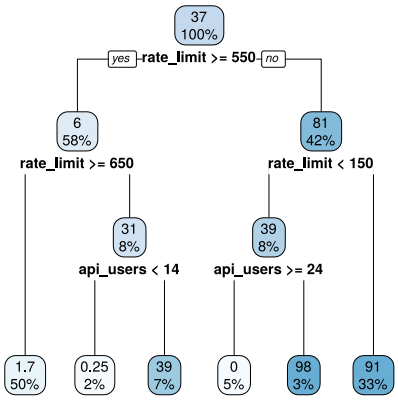

Fig. 11. Load balancing & rate limit combined: Reliability decision tree on the private cloud (% failure)

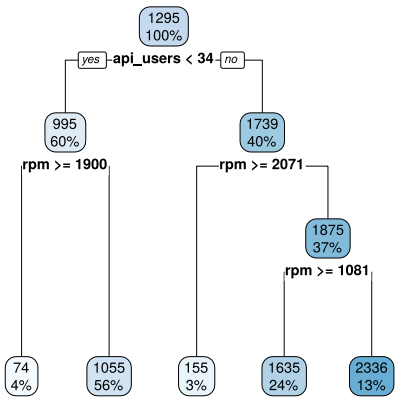

Fig. 12. Rate limit & request bundle combined: Performance decision tree on the private cloud (milliseconds)

from the collected data. Each new data collection will yield values specific to the application settings and usage period. In Algorithm 1, in case $api\_users < 14$ or $api\_users \geq 24$, then the conditions that are displayed are extracted from the decision trees depicted in Figures 2 and 3, as described in the last section. The case in the middle is illustrated by Figures 6 and 7. Algorithm 2 follows the same logic using different decision trees depicted in Figures 8 to 13.

Algorithm 1 API design recommendation (AWS)

for each $method \in methods$ **do**
  if $api\_users < 14$ **then**
    $bundle\_size = 10$ and
    $range(rate\_limit, [150, \infty])$ and
    $combination = LB \wedge RB \wedge RL$
  else if $api\_users < 24$ **then**
    $bundle\_size = 20$ and
    $range(rate\_limit, [150, \infty])$ and
    $combination = RL \wedge RB$
  else
    $range(bundle\_size, \{30, 40, 50\})$ and
    $range(rate\_limit, [150, \infty])$ and
    $combination = LB \wedge RB \wedge RL$
  end if
end for

VI. Discussion and Threats to Validity

In this section, we first discuss the generalizability of the results and the possible design advice. Then, we provide some threats of validity that we may encounter.

A. Discussion

The following results are observed from both generated algorithms 1 and 2. While the outcomes of these algorithms differ in several cases, we also observe some similarities, which are discussed in the following.

Load balancing & request bundle combined: This combination is beneficial when rpm is not very high, if at all since else it would overload the API backend with large messages and create congestion in the load balancer. It is illustrated in Algorithm 2 that this combination is only recommended with relatively low api_users and $bundle\_size$ in the private cloud. However, for AWS, our algorithms do not recommend

Algorithm 2 API design recommendation (private cloud)

```
for each method ∈ methods do
    if api_users < 14 then
        bundle_size = 10 and
        range(rate_limit, [650, ∞]) and
        combination = LB ∧ RL
    else if api_users < 24 then
        bundle_size = 20 and
        combination = LB ∧ RB
    else if api_users < 34 then
        bundle_size = 30 and
        if rpm ≥ 1900 then
            range(rate_limit, [150, ∞]) and
            combination = RL ∧ RB
        else
            range(rate_limit, [650, ∞]) and
            combination = LB ∧ RL
        end if
    else
        range(bundle_size, {40, 50}) and
        if rpm ≥ 2071 then
            range(rate_limit, [150, ∞]) and
            combination = RL ∧ RB
        else
            range(rate_limit, [250, ∞]) and
            combination = LB ∧ RB ∧ RL
        end if
    end if
end for
```

this combination. The combination should be used in applications with a limited or dedicated number of users, like a private company website.

Rate limit & request bundle combined: When using the private cloud, this combination might be used in highly demanding environments with high values of api_users and rpm. *Rate Limit* plays a significant role in preventing the backend from being overwhelmed. Also, when combined with *Request Bundle*, *Rate Limit* does not cause a high failure rate, increasing reliability from the API clients' perspective. However, *Request Bundle* has to be supported on both the client and server side, increasing complexity and costs.

Load balancing & rate limit combined: The combination is to be used when api_users and rpm are relatively low due to using one load balancer. Adding more load balancers will make the combination more scalable. It should also be used with relatively high values of $rate\_limit(\geq 650)$. This shows that *Load Balancing* plays a significant role in distributing the load and alleviating the role of *Rate Limit* in preventing server overload and decreasing both performance and reliability. The combination should not be used in highly versatile environments like AWS.

Load balancing, request bundle & rate limit combined: The combination is recommended in relatively extreme situations when either api_users or rpm are relatively high. *Request Bundle* plays a crucial role, especially on AWS, by preventing the load balancer from overloading when api_users and rpm are very high. The combination is the most flexible and should be used in highly versatile applications like Social Media and E-commerce.

B. Threats to validity

In any modeling study, a couple of threats to validity need to be considered and mitigated by workaround solutions or future improvements.

The first threat to validity is regarding the algorithms presented in the last section. We considered all $methods$ to be the same because the type of the method had a negligible effect on the $total\_time$ calculated. However, we do not exclude the general case where in some situations, some types of $methods$ might take much longer execution times than others. To mitigate this issue, we plan to add a $method\_time$ variable to our model in future work.

Also, the API pattern implementations used in our model are particular to the configuration used in a prior study. For instance, only one load balancer and one type of rate limiting are used. This could lead to an external threat of validity regarding the general applicability of our model. While the dataset used for different configurations might be completely different, we believe that the automation and cyclic process of the tree decision modeling to generate best-case scenarios is still valid. Nevertheless, we must validate that using different datasets from different configurations and settings.

The proposed benchmark might be biased because it is based on only one open-source application. We tried to mitigate this issue by selecting a realistic open-source application (based on real-life experience in the domain) by former practitioners. As the authors were not involved in the application writing, the bias that we could have influenced the experiment outcomes this way is mitigated. We selected diverse operations from the application to represent a realistic operation set in the benchmark. However, the application is a business system. Systems with vastly different loads might have different characteristics and need other benchmarks. For these reasons, we cannot claim generalizability beyond the system represented well by this benchmark.

Our results might not be generalizable beyond the environments and technologies used in our study. We tried to mitigate this threat by using two very different environments, the private cloud and AWS. We only tested one configuration on each, but as our approach requires rerunning the benchmark on a new environment to provide accurate recommendations, dealing with this threat is part of our approach. We believe that for similar private or public cloud scenarios, our generated algorithms will likely give a reasonable estimate but no precise recommendations. This would be good enough for design advice but not for automatic adaptations based on our approach.

VII. Conclusion and Future Work

To answer **RQ1** and **RQ2**, we have developed algorithms based on decision trees generated for each combination of the studied API patterns.

Regarding **RQ1**, we have found that the api_users and rpm values are the main criteria for each decision on those algorithms. We also conclude that each recommended combination of API patterns is relevant to a specific configuration setting. Specifically to the configuration in this paper, *Load Balancing* and *Request Bundle* combined should only be used with a limited number of api_users, preferably in a dedicated

environment. Similarly, the combination *Load Balancing* and *Rate Limit* should be avoided in public clouds, for example. Instead, all the patterns should be combined unless the cost and complexity of *Request Bundle* is an issue.

Concerning **RQ2**, not only have we developed algorithms from decision tree models to generate the best combinations of these patterns in terms of performance and reliability impacts, but we also propose an entire API design recommendation pipeline that generates those algorithms automatically by re-running our benchmark using new settings and configurations. The validated algorithms are then stored in a catalog for re-use if the same configuration setting is triggered.

Our generated algorithms can provide for private or public cloud scenarios similar to our experiment setting a good estimate which would be good enough for design advice. Also, using our approach in a self-adaptation scenario is possible, i.e., to run automatic adaptations based on our approach. But for this, we need to re-run our benchmark on the target environment to provide precise recommendations.

In future work, we plan to run the API design recommendation pipeline for new configurations and settings. We also aim to integrate new parameters into our model, like the methods execution time.

REFERENCES

[1] O. Zimmermann, M. Stocker, D. Lübke, C. Pautasso, and U. Zdun, "Microservice api patterns," https://microservice-api-patterns.org/, 2022.

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

[9] 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.

[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] P. den Hamer and T. Skramstad, "Autonomic service-oriented architecture for resilient complex systems," in *2011 IEEE 30th Symposium on Reliable Distributed Systems Workshops*, 2011, pp. 62–66.

[14] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, 2004.

[15] O. Layaida and D. Hagimont, "Designing self-adaptive multimedia applications through hierarchical reconfiguration," in *Distributed Applications and Interoperable Systems*, L. Kutvonen and N. Alonistioti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 95–107.

[16] S. Jahan, I. Riley, A. Sabino, and R. Gamble, "Towards a plug-in architecture to enable self-adaptation through middleware," in *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, 2021, pp. 214–219.

[17] E. Gjørven, F. Eliassen, K. Lund, V. S. W. Eide, and R. Staehli, "Self-adaptive systems: A middleware managed approach," in *Self-Managed Networks, Systems, and Services*, A. Keller and J.-P. Martin-Flatin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 15–27.

[18] S. S. Rathore and S. Kumar, "A decision tree regression based approach for the number of software faults prediction," *SIGSOFT Softw. Eng. Notes*, vol. 41, no. 1, pp. 1–6, feb 2016. [Online]. Available: https://doi.org/10.1145/2853073.2853083

[19] T. Khoshgoftaar, E. Allen, L. Bullard, R. Halstead, and G. Trio, "A tree-based classification model for analysis of a military software system," in *Proceedings. IEEE High-Assurance Systems Engineering Workshop (Cat. No.96TB100076)*, 1996, pp. 244–251.

[20] B. Cheung Chiu and G. I. Webb, "Using decision trees for agent modeling: Improving prediction performance," *User Modeling and User-Adapted Interaction*, vol. 8, no. 1, pp. 131–152, 1998. [Online]. Available: https://doi.org/10.1023/A:1008296930163

[21] Y. Hou, P. Edara, and C. Sun, "Modeling mandatory lane changing using bayes classifier and decision trees," *IEEE Transactions on Intelligent Transportation Systems*, vol. 15, no. 2, pp. 647–655, 2014.

[22] 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/

[23] 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/

[24] A. Lawi, B. L. E. Panggabean, and T. Yoshida, "Evaluating graphql and rest api services performance in a massive and intensive accessible information system," *Computers*, vol. 10, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2073-431X/10/11/138