# Evaluation of API Request Bundling and its Impact on Performance of Microservice Architectures

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

***Abstract*—The widespread adoption of Microservice architectures has posed many challenges regarding API design for these architectures. Several API best practices and patterns have been proposed that could help API designers ensure API quality attributes such as reliability, availability, and performance. API Request Bundling, which is in focus of this paper, is one of those patterns that aims at optimizing performance. The pattern promises substantial performance gains, but can also lead to significant drawbacks such as increased development effort and application complexity. So far, there is little to no rigorously acquired knowledge to judge whether applying Request Bundling is worth the costs in a given Microservice architecture. To improve this situation, we performed an empirical study based on a Microservice-based open source business application using realistic workload scenarios. To estimate the performance impact of Request Bundling, we derived a regression model and performed a multivariate regression analysis. These selected regression models can help distributed system engineers and architects to estimate the gain in performance, in terms of round-trip time, with or without Request Bundling. Our approach followed in the paper, can be customized to other Microservice architectures or to study other performance-related Microservice patterns.**

***Index Terms*—API Request Bundling; Modeling; Microservices; Cloud; Performance.**

## I. Introduction

Numerous quality aspects of Microservices APIs have been identified in the context of critical application design decisions to be made by software architects [1]. The *Request Bundle* pattern [2], described as part of the Microservice API (MAP) patterns[1], is a commonly used Microservice API pattern that can be used to optimize Microservice API performance-related qualities. In particular, it can be used to reduce communication overhead such as throughput, latency, and bandwidth use between API clients and backends. Instead of sending many messages, API clients can bundle all these messages and send them in fewer chunks, which can considerably reduce network use and overall roundtrip times of messages. As a downside, some complexity and latency maybe be introduced on both client and server side due to the larger amount of data to be processed at once. Usually, the number of messages is decreased and size of messages is increased. Support for request bundling requires design and implementation efforts both on client and server sides. It also increases the complexity of the messages and the message processing architecture.

In order to figure out the real impact of request bundling on performance and whether the aforementioned complexity and efforts are justifiable and worth the costs, we have conducted an empirical study based on a Microservice-based open source business application using realistic workload scenarios. Firstly, we aim to establish a better understanding of the *Request Bundle* pattern through its empirical evaluation. Secondly, we try to quantify the possible performance gains using such a pattern, and provide a model to estimate it in applications and workloads similar to the tested open source application. Thirdly, we provide a method of studying such optimization techniques, which can be easily adapted to other patterns or to study the *Request Bundle* pattern in the context of substantially different applications. We aim to answer the following research questions:

- **RQ1** What is the performance impact of the *Request Bundle* pattern in a realistic application and workload setting?
- **RQ2 a)** How can we evaluate the *Request Bundle* pattern empirically and how can we measure its performance impacts? **b)** What method(s) can be used to study such optimization techniques (or related patterns) for Microservices API communication?

In our empirical study, we have gathered data from a representative and modern Microservice-based setup. We have deployed the Lakeside Mutual application[2], an open source system implemented based on experiences with real-life applications, on the Istio[3] service mesh. The installation is hosted on a private Cloud. We have studied different scenarios in this application with and without request bundling, and with different message bundle sizes. In total, we ran 30 tests on a private Cloud with a total running time of more than 15 hours (including setup times) and a total number of 8160 messages per repeated experiment run (at least 5 times). Based on the data from this empirical experiment, we created four regression models that can be used to estimate the total round trip time from the API client to the backend, to evaluate response time or performance of an API request with or without request bundling.

The paper is organized as follows. Section II compares to related works. Then we present some background about request bundling in Section III. In Section IV, we provide

[1]See: https://microservice-api-patterns.org/

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

[3]https://istio.io/v1.4/

details about our proposed regression model. We evaluate this model empirically in Section V. Section VI presents an analysis of the data collected and the regression models selected. Finally, threats of validity are discussed in Section VII, and we conclude in Section VIII.

## II. Related Work

There are two types of request bundling. The first one consists of sending one single bundled request and receiving one single bundled response. The second one sends one single bundled request but receives instead multiple responses. Further, a *Request Bundle* might consist of non-dependent or dependent batch requests [3]. In our study, we focus on the first type of request bundling using non-dependent batch requests.

Request bundling is a well-known technique already implemented widely for optimization in Web browsers [4]–[6]. Also, it has been introduced as a way to optimize energy consumption in mobile applications by detecting and bundling multiple HTTP requests into one [7]. Some Cloud providers (e.g. [8]) propose request bundling as a paid feature for reaching better performance and lowering costs. However, there are no indications or guidelines on how and when to use this technique. In addition, although some studies have praised the overall performance gains achievable with request bundling [3], there is no concrete information about what performance improvements to expect in a given situation.

Many API-related empirical studies have exhibited a set of recommendations to improve APIs performance. Wittern et al. [9] have studied GraphQL[4] interfaces in practice, as a query language that can improve API performance by using fewer client-server roundtrips and reduced response message sizes. An OpenAPI[5]-based framework has been proposed by Bucaille et al. [10] to test non-functional properties of REST APIs such as performance and availability. Park et al. [11] presented a REST-MapReduce framework, specific to mobile Cloud computing, to increase the performance of both REST OpenAPI service and MapReduce. However, those studies are technology-dependent and environment-specific. There is no platform-agnostic model proposed, as it is the case in our study by using the well-known and widely-used *Request Bundle* pattern as the object of our study.

## III. Background: Request Bundling

*Request Bundle* advises to define a bundle of messages as a container that assembles multiple individual requests in a single request message. The bundle message usually contains metadata such as number and identifiers of individual messages. *Request Bundle* can be found in many public and private APIs (see e.g. the known uses in [2]). The basis of this optimization is that in many cases, clients are able to "foresee" that requests can be bundled automatically; that is, in those cases, if supported by the Microservice API, the optimization can be applied automatically by client-side tools, without human intervention. For example, consider users on client-side visiting linked pages of data, and for retrieving each page a message is needed. The client application could profile user behavior and prefetch the typical next pages visited by the user with a *Request Bundle*.

To make the decision for or against using a *Request Bundle*, regardless whether it is human or automated decision making, it needs to be understood whether the effort is worth the costs, and if in the specific design situation (for human decision making) or runtime decision (if decision making is automated), a substantial performance gain can be achieved. In particular, the additional effort needed to implement a more complex API endpoint and corresponding clients needs to be considered as costs of the use of the pattern. The reduction of the number of messages does not imply that less information is exchanged. Thus, the remaining messages need to carry more complex payloads. This paper deals with the problem that so far empirical data and models that can help to make such decisions based on a solid and empirical ground are missing.

## IV. Regression model

### A. Independent and dependent variables

We define the dependent variable $total\_time$ as the total round-trip time taken to process an API client request:

$$total\_time = network\_time + backend\_time \tag{1}$$

where $network\_time$ is the round-trip time spent between the API client and the API Gateway, and $backend\_time$ is the round-trip time spent from the API Gateway to the database or backend service (in case no database call is made).

On the backend, we define a number of variables to characterize the operations executed in the backend (see Table I). Those variables should enable our model to estimate the time spent during backend computations. Then, we have:

$$\begin{aligned} backend\_time &= inmem\_time + db\_time + dist\_time \\ &= inmem\_weight \cdot inmem\_calls \\ &\quad + db\_weight \cdot db\_calls + dist\_weight \cdot dist\_calls \end{aligned} \tag{2}$$

Here, $inmem\_weight$ is the weight factor corresponding to the number of in-memory operations $inmem\_calls$, $db\_weight$ is the weight factor corresponding to the number of database operations $db\_calls$, and $dist\_weight$ is the weight factor corresponding to the number of distributed operations $dist\_calls$. We introduce the weights to ease the application of our model and enable adjusting our model to different applications and environments. That is, the effort in applying the model can be eased because by using the weights it is possible to simply count the number of operations to generate a new estimate using our model, if acceptable weight values have been measured before.

In summary, to make an estimation with our model, a user of the model needs to count the number of calls of the different types in the operations that are candidates for request bundling. In addition, to adjust our model, a number of calls of each type should be measured to obtain mean

[4]https://graphql.org

[5]https://swagger.io/specification/

TABLE I: Definition of Parameters

| Independent variable | Description |
|---|---|
| $inmem\_time$ | Time spent in milliseconds by in-memory operations when executing a function. |
| $inmem\_calls$ | Number of in-memory operations when executing a function. |
| $db\_time$ | Time spent in milliseconds by database operations when executing a function. |
| $db\_calls$ | Number of database operations when executing a function. |
| $dist\_time$ | Time spent in milliseconds by distributed calls when executing a function. |
| $dist\_calls$ | Number of distributed calls when executing a function. |
| $request\_bundle$ | Categorical variable indicating whether request bundling is used or not. |
| $method$ | Categorical variable indicating which function is executed. |
| **Dependent variable** | **Description** |
| $total\_time$ | Total round trip time in milliseconds between the API client and backend. |

performance measurements to determine the weights. Both is rather straightforward (e.g., with a simple counting script and simple instrumentation of the source code).

Using equations (1) and (2), we get:

$$\begin{aligned} total\_time &= network\_time + inmem\_time + db\_time \\ &\quad + dist\_time \\ &= network\_time + inmem\_weight \cdot inmem\_calls \\ &\quad + db\_weight \cdot db\_calls + dist\_weight \cdot dist\_calls \end{aligned} \tag{3}$$

Finally, we introduce two additional categorical variables, which describe if request bundling is used or not, and which kind of RESTful operation type that is executed, respectively: Get, Update, or Create (see Table I). Please note that Delete could be added here, but is not included, as the Lakeside Mutual application does not use it in our scenarios.

## V. Empirical study

In this section, we describe our empirical study. First, we define the scope of the study. Second, we give an overview of the open source application or *prototype* we have selected for our study and that executes the *tasks* or functions that we measure in our experiment. Then, we provide details about the used hardware and software configuration of our *Private Cloud*. After that, we describe the developments and deployments realized as well as the *measurement tools*. Finally, we describe how to launch the experiment or the *workload*. Finally, we discuss our experiment design and hypotheses. We have followed the guidelines proposed in [12], [13].

The study's data set and further analyses are provided as an open data set for enabling reproducability of our results[6].

### A. Scope definition

The goal of the empirical study is to analyze API request bundling usage for the purpose of predicting its impact on the performance of a Microservice-based application, from the perspective of API clients. It is a *mutli-test within object study quasi-experiment* where one single *object* is evaluated using a set of *subjects*. The *context* involves as *subjects* the sizes of the requests, which were preselected, the *methods* used, and a flag indicating whether request bundling is used or not. Details are provided in Section V-F. The *object* is a close to real-life microservices-based application, available online, and developed with the goal to mimic professionals microservice/API design (details provided in Section V-B). We have deployed this application on a modern Private Cloud (details are described in Sections V-C, V-D and V-E).

### B. Lakeside Mutual application

Lakeside Mutual[7] is a Java-based open source system implemented based on experience with real-life applications. It implements an insurance company services to customers and employees. It is composed of four frontend microservices and four backend microservices and the latter are in the focus of our study. We are particularly interested in three functions that are provided by these microservices:

- Get customers returns a list of customers using their IDs. It can be used by both customers and employees. However, a customer can only get information using the *customer-self-service-backend* microservice. On the other hand, an employee can get information about all customers using the *customer-management-backend* microservice. Both call the *customer-core* microservice that takes care of the fetch database operations. This function can be invoked with or without using request bundling.
- Update customers' addresses changes the addresses of a list of customers. Again, it can be used by both customers and employees under the same conditions listed for the first function. This function did not support request bundling in the original application. Request bundling support for it has been implemented by us.
- Create insurance quota requests creates a list of insurance quota requests for a given customer. It can be realized by customers using the *customer-self-service-backend* microservice, which then forwards the request to the *policy-management-backend* microservice. Request bundling for this function had to be implemented by us.

### C. Configuration details

We have executed the experiments on a Private Cloud that consists of 3 Ubuntu[8] 18.04.5 LTS Virtual Machines (VMs) on top of vSphere[9] 6.7 environment. Each of these VMs hosts a Minikube instance version 1.14.2 with 8 dedicated CPU cores Intel Xeon(R)™CPU E5-2650 v4 @ 2.20GHz and 20 GB of system dedicated memory. Each Minikube instance hosts a Kubernetes engine version 1.14.2 and Istio version 1.4.3.

The central Minikube instance contains the central *Control Plane*, Kong Ingress Controller[10] version 0.8.0 and the *customer-self-service-backend* microservice built using Java

[6]https://doi.org/10.5281/zenodo.5107982

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

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

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

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

version 8. The 3 remaining microservices are also built using the same version of Java and accessed using the aforementioned *Ingress Controller* through YAML-defined Kubernetes Endpoints[11] and Ingress Rules[12]. On client side, one Ubuntu 18.04.5 LTS virtual desktop is used to launch HTTP requests into the Private Cloud. The virtual desktop has 2 CPU cores Intel® Xeon(R) CPU E5-2650 0 @ 2.00GHz with 8 GB of system memory.

### D. Development and deployment details

To test request bundling in the three functions described in Section V-B, we have implemented two additional methods in the microservices source code, which are the following:

- changeAddresses(), which takes as input a comma-separated list of IDs and addresses, is added to both *customer-management-backend* and *customer-core* microservices API specification. This method updates those customers data with the new provided addresses.
- createInsuranceQuoteRequests(), which takes as input a comma-separated list of IDs and other information about the insurances, is added to the *customer-selfservice-backend* microservice API specification. This method creates insurance quota requests for the customers.

Also, the source code is instrumented to compute the independent variables defined in our regression model described in Section IV, which are $db\_time$, $inmem\_time$ and $dist\_time$, which required only trivial instrumentation.

Each of the microservices is running as a containerized Docker[13] image deployed on a multi-clustered Istio service mesh with one single shared control plane[14]. With each of these microservices running on a separate VM, the communication between them is established using *Edge proxies* or *Sidecars* [14], over a private network. An *API gateway* [15] or *Front proxy* is responsible of intercepting incoming or ingress communication from API clients. For that purpose, the Kong API Gateway is used by integrating it to Istio [16].

### E. Launching the experiment

Each experiment consists of executing a Shell[15] script on the virtual desktop that executes cURL[16] HTTP requests on the Private Cloud API Gateway and does the following:

- Create a list of preauthorized customers;
- Get customers data with and without request bundle;
- Update customers' addresses with and without request bundle;
- Create insurance quota requests with and without request bundle;

The dependent variable $total\_time$, defined in our regression model described in Section IV, is recorded in the output file for each of the above operations. Also, $backend\_time$ is calculated using the HTTP headers *X-Kong-Upstream-Latency* and *X-Kong-Proxy-Latency* provided by Kong API Gateway[17]. On server side, the independent variables $db\_time$, $inmem\_time$ and $dist\_time$ are collected and computed using the output log file of each microservice.

TABLE II: Data collected during the experiments

| Request bundle? | Function | Size | Inmem_calls | Inmem_time | Db_calls | Db_time | Dist_calls | Dist_time | Total_time |
|---|---|---|---|---|---|---|---|---|---|
| No | Get | 50 | 2600 | 55,07355 | 50 | 46,597509 | 50 | 929,328941 | 1673 |
| | | 40 | 2080 | 45,576909 | 40 | 38,590982 | 40 | 1112,832109 | 1949 |
| | | 30 | 1560 | 40,305603 | 30 | 38,190849 | 30 | 772,503548 | 1432 |
| | | 20 | 1040 | 19,344074 | 20 | 15,071418 | 20 | 380,584508 | 819 |
| | | 10 | 520 | 11,842218 | 10 | 11,648897 | 10 | 301,508885 | 511 |
| | Update | 50 | 1500 | 15,338561 | 100 | 97,869806 | 50 | 828,791633 | 1794 |
| | | 40 | 1200 | 13,156796 | 80 | 90,622794 | 40 | 722,22041 | 1710 |
| | | 30 | 900 | 9,710725 | 60 | 74,760225 | 30 | 581,52905 | 1330 |
| | | 20 | 600 | 5,964218 | 40 | 33,33821 | 20 | 343,697572 | 701 |
| | | 10 | 300 | 3,493898 | 20 | 22,846122 | 10 | 168,65998 | 332 |
| | Create | 50 | 1600 | 491,892331 | 100 | 95,342467 | 50 | 1721,765202 | 3088 |
| | | 40 | 1280 | 437,888573 | 80 | 82,19147 | 40 | 1485,919957 | 2833 |
| | | 30 | 960 | 327,063494 | 60 | 60,522121 | 30 | 1143,414385 | 1999 |
| | | 20 | 640 | 252,230347 | 40 | 37,559331 | 20 | 856,210322 | 1422 |
| | | 10 | 320 | 118,169791 | 20 | 23,961 | 10 | 364,869209 | 678 |
| Yes | Get | 50 | 241 | 5,02067 | 50 | 17,598524 | 1 | 46,380806 | 97 |
| | | 40 | 201 | 8,473363 | 40 | 20,126167 | 1 | 43,40047 | 112 |
| | | 30 | 161 | 7,883633 | 30 | 17,62739 | 1 | 49,488977 | 92 |
| | | 20 | 121 | 2,044953 | 20 | 6,320683 | 1 | 37,634364 | 70 |
| | | 10 | 81 | 1,563875 | 10 | 7,096915 | 1 | 23,33921 | 55 |
| | Update | 50 | 937 | 14,533892 | 100 | 75,450795 | 1 | 63,015313 | 302 |
| | | 40 | 757 | 16,697279 | 80 | 84,001458 | 1 | 28,301263 | 330 |
| | | 30 | 577 | 10,811141 | 60 | 64,232514 | 1 | 17,956345 | 187 |
| | | 20 | 397 | 5,966065 | 40 | 29,861325 | 1 | 18,17261 | 115 |
| | | 10 | 217 | 3,649556 | 20 | 15,518327 | 1 | 12,832117 | 70 |
| | Create | 50 | 3488 | 46,408423 | 100 | 120,666903 | 50 | 1660,924674 | 1945 |
| | | 40 | 2798 | 41,364438 | 80 | 81,855359 | 40 | 1291,780203 | 1945 |
| | | 30 | 2108 | 36,585093 | 60 | 68,053868 | 30 | 1414,361039 | 1600 |
| | | 20 | 1418 | 25,210233 | 40 | 38,04053 | 20 | 1129,749237 | 1244 |
| | | 10 | 728 | 19,248005 | 20 | 25,998748 | 10 | 355,753247 | 440 |

### F. Experiment design and hypotheses

In our study, we aim to measure the effect of using request bundling on API performance or response time. The size of the request bundle is chosen in a predefined interval setting. We have limited the maximum request bundle size to 50 to be realistic. Our experiment follows the *one factor with two treatments balanced* design since in each trial, we either use request bundling or not on all the functions described in V-B.

Hence, we define the following experiment hypotheses:

- The *null hypothesis* $H_0$ states that request bundling has negative or no effect on API performance.
- The *alternative hypothesis* $H_1$ states the opposite.

Those hypotheses were checked for each combination of using request bundle (Yes/No), tested function (Get/Update/Create) and request bundle size (5 intervals), which totals 30 tests.

## VI. Data collection and analysis

### A. Multivariate regression analysis

As described in Section V-E, the variables $total\_time$, $inmem\_time$, $db\_time$ and $dist\_time$ data are directly collected from output log files on client and server sides. The remaining variables $inmem\_calls$, $db\_calls$ and $dist\_calls$

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

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

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

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

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

[16] https://curl.haxx.se/docs/manpage.html

[17] https://docs.konghq.com/gateway-oss/0.10.x/proxy/

TABLE III: Models' description

| Model | Intercept | Inmem_time | inmem_calls | db_time | db_calls | dist_time | dist_calls | request_bundle(Y) | method (Get) | method (Update) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | -54,585 | 1,506 | X | 4,074 | X | 1,192 | X | -312,011 | 331,758 | 312,936 |
| 2 | -94,019 | 1,795 | -0,081 | X | X | 0,968 | 17,01 | -76,891 | 226,742 | 330,915 |
| 3 | 165,215 | 1,695 | -0,077 | X | X | 0,576 | 25,034 | -75,963 | X | X |
| 4 | 602,925 | X | -0,194 | X | 7,85 | X | 35,105 | -296,56 | -386,275 | -637,184 |

TABLE IV: Models' summary

| | Adjusted R-squared | F-Statistic: p-value | Jarque-Bera: p-value | Ljung-Box: p-value |
|---|---|---|---|---|
| Model 1 | 0.9815 | <2.2e-16 | 0.8407 | 0.1378 |
| Model 2 | 0.9835 | <2.2e-16 | 0.7316 | 0.09669 |
| Model 3 | 0.9767 | <2.2e-16 | 0.2867 | 0.03966 |
| Model 4 | 0.9247 | 6.338e-13 | 0.8405 | 0.1169 |

are directly counted in the source code. Table II shows the data collected for experiments using request sizes of 10, 20, 30, 40 and 50.

As a result of our analysis using R language[18], we obtained the four regression models described in Tables III. Notice that the categorical variable $method(Create)$ is not shown since it was selected as the reference category and so its coefficient is equal to zero [17]. The same applies to the categorical variable $request\_bundle(N)$, in case no request bundling is used.

As Table IV shows, all the models have very low (F-Statistic) p-value and very high Adjusted R-squared values which clearly hints that they have a very high *statistical significance*. However, we cannot give a definite conclusion without an analysis of the residuals.

In order to verify the applicability of the linear models described in Table III, we first have to check that the residuals follow a normal distribution. Figures 1 to 4 (provided in our open data set[6]) show the Histograms of Residuals of all those models. Except for Model 2 and 3, the rest of the histograms display approximately a bell curve shape which suggests that the residuals of Models 1 and 4 are probably normally distributed. Model 2 and 3's Normal Q-Q Plots in addition show that most of the points lie straight on the line. So, there is no reason to believe that the residuals are not normally distributed.

Multi-Fit Studentized Residual diagrams are Studentized Residual diagrams with multiple variables as independent variables. Studentized Residuals can detect potential outliers in the models. They are reliable since they consider each observation as a potential outlier, remove it, and refit the regression model. This process is repeated until all the observations are tested. As we can see in Figures 1 to 4, very few outliers are observable with no obvious outliers in Models 2 and 3, and there is an overall clear constant variance exhibited by all the models.

To validate these observations, we have run Jarque-Bera tests on all models and the results are shown in Table IV. The null hypothesis $H_0$ for these tests tells us that the residuals are normally distributed. With a level of significance $\alpha$ = 0.05, we fail to reject the null hypothesis for all the models, which confirms that these models' residuals are normally distributed.

[18]https://www.r-project.org

To check whether these residuals are independent, we use the Ljung-Box tests as described in Table IV. The null hypothesis $H_0$ for these tests tells us that the residuals are not autocorrelated. With a level of significance $\alpha$ = 0.05, we fail to reject the null hypothesis for all models except for Model 3, which is thus discarded from further analysis. Therefore, Models 1, 2 and 4 residuals are independent and are selected for $total\_time$ prediction calculations.

### B. Selected models

Based on the analysis from the previous section, we have selected Models 1, 2 and 4 to produce our prediction data. Model 4 produces two negative values for the predicted $total\_time$, and so it is discarded from further analysis. Figures 5 to 10 (provided in our open data set [6]) compare the models' values of predicted $total\_time$ and its actual measured values when using the different methods with and without using request bundle. A visual inspection confirms that the estimation for most methods with or without request bundling provide reasonably close values. Some notable specific observations are: It is observable that all models work best in case we do not use request bundling and for the *Create* method using request bundle. In case of the *Get* method using request bundle, the models are best when we have at least 30 requests bundled. For the *Update* method using request bundle, Model 1 makes the best predictions for $total\_time$.

To compare the two selected models, we have calculated their prediction errors using the Mean Absolute Percentage Error (MAPE) [18]. We found that models 1 and 2 have prediction errors that are equal to 16% and 21% respectively, which are below the target prediction error of up to 30% for Cloud-based architectures [19], and thus explainable with network infrastructure imperfections such as latency and unforeseen errors.

## VII. Validity evaluation

As shown in Table IV, the models proposed display a very high *statistical significance*, which provides some confirmation for the conclusions from our experiment. Also, in Section VI-A, we have accurately verified our *statistical test assumptions* concerning residuals' normality, and the *error rate* is very low since we have conducted the tests using a significance level of 0.05. The *instrumentation* used in the tests and described in Section V encourages a very high *reliability of the measures*. However, there might be a threat of validity regarding the *reliability of treatment implementation* since we have made some changes to the application as explained in Section V-D. While those were rather small and following the same development methods used in the rest of Lakeside mutual application, an impact of those changes on the results cannot be fully excluded.

On client-side, the dependent variable $total\_time$ was measured using Linux provided *date* utility for calculating elapsed time between the start and end of an API call. Similarly on the server-side, the independent variables $inmem\_time$, $db\_time$ and $dist\_time$ are measured using JAVA provided

*time* utility integrated into the source code. There is no reason to believe that these measurements lack accuracy. The rest of independent variables $inmem\_calls$, $db\_calls$ and $dist\_calls$ are manually counted, which was convenient considering the limited number of calls in the application we used. However, this is considered as a threat of validity for other large applications. To alleviate this threat, we plan to develop a script that can generate the counting results automatically.

We studied only the Lakeside Mutual open source application. While it is a realistic business application, realized with state-of-the-art technologies, it cannot be excluded that no results can be generalized to other applications. Especially, generalization to non-business applications might be problematic. In our study, we only considered three methods (although the most common) which might be a threat of validity, especially for other types of applications. Also, the *instrumentation* employed is very specific and technology dependent. In order to generalize our approach, we plan to apply our models to other applications and technologies.

## VIII. Conclusion and future work

In this paper, we evaluated the impact of API request bundling on the performance of Cloud-based benchmark based on a realistic application.

To answer **RQ1**, we have evaluated the impact of request bundling by empirically calculating the total round-trip time using 30 different *workload* settings. As shown in Table II and Figure 5 to 10, it is clear that $total\_time$ decreases considerably when using request bundle and independently of the *workload* setting. As a result, we successfully reject the null hypothesis $H_0$ described in Section V-F and confirm that request bundling has a positive impact on performance, independently of the *workload* setting.

Concerning **RQ2**, we presented a regression model that can be used to estimate performance impact of request bundling. This can be easily elaborated by counting $inmem\_calls$, $db\_calls$ and $dist\_calls$, given appropriate mean weights values measured before as described in Section IV. We have empirically validated the model using a realistic application and reported four regression models using the data collected. After data analysis, we selected two models that best fit. These models generate prediction errors that are substantially below the target prediction error of up to 30% for Cloud-based architectures. Further, it is important to note that a rough estimation is usually good enough for making architectural design decisions, as those are usually decided at an early project stage. In addition to those models, we have defined an approach that not only can be used for request bundling evaluation using a specific *workload* setting and application, but can also be applied to other optimization techniques and patterns.

In future work, we plan to validate our model using other *workload* settings, applications and technologies. For that purpose, we aim at automating the *instrumentation* used for counting $inmem\_calls$, $db\_calls$ and $dist\_calls$, in order to be able to use it in larger applications. We also plan to integrate other parameters to our model, especially those related to network latency and error management.

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

## References

[1] 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.

[2] 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

[3] A. Sinha, "Batch: An api to bundle multiple rest operations," Nov. 2018. [Online]. Available: https://medium.com/paypal-engineering/batch-an-api-to-bundle-multiple-paypal-rest-operations-6af6006e002

[4] M. Schulz, "Bundling and minification: an introduction," January 2015. [Online]. Available: https://mariusschulz.com/blog/bundling-and-minification-an-introduction

[5] S. Addie and D. Pine, "Bundle and minify static assets in asp.net core," Februar 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/client-side/bundling-and-minification?view=aspnetcore-5.0

[6] K. Gysen, "Bundle / code splitting revised," May 2019. [Online]. Available: https://medium.com/@kimgysen/bundle-code-splitting-revised-d9719e9219c1

[7] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated energy optimization of http requests for mobile applications," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 249–260.

[8] brainCloud, "Making s2s api calls from postman," 2021. [Online]. Available: https://getbraincloud.com/apidocs/portal-usage/using-postman-with-s2s/

[9] E. Wittern, A. Cha, J. C. Davis, G. Baudart, and L. Mandel, "An empirical study of graphql schemas," in *Service-Oriented Computing*, S. Yangui, I. Bouassida Rodriguez, K. Drira, and Z. Tari, Eds. Cham: Springer International Publishing, 2019, pp. 3–19.

[10] S. Bucaille, J. L. Cánovas Izquierdo, H. Ed-Douibi, and J. Cabot, "An openapi-based testing framework to monitor non-functional properties of rest apis," in *Web Engineering*, M. Bielikova, T. Mikkonen, and C. Pautasso, Eds. Cham: Springer International Publishing, 2020, pp. 533–537.

[11] J.-H. Park, H.-Y. Jeong, Y.-S. Jeong, and M. Choi, "Rest-mapreduce: An integrated interface but differentiated service," *Journal of applied mathematics*, vol. 2014, pp. 1–10, 2014.

[12] R. Ré, R. M. Meloca, D. N. Roma, M. A. da Cruz Ismael, and G. C. Silva, "An empirical study for evaluating the performance of multi-cloud apis," *Future Generation Computer Systems*, vol. 79, pp. 726–738, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17301802

[13] C. Wohlin, *Experimentation in software engineering*. Berlin Heidelberg: Springer, 2012.

[14] 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.

[15] C. Richardson, "Pattern: Api gateway / backends for frontends," 2021. [Online]. Available: https://microservices.io/patterns/apigateway.html

[16] 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/

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

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

[19] 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.