# Optimizing Microservices Performance and Resource Utilization through Containerized Grouping: An Experimental Study

Fernando H. L. Buzato
*Department of Computer Science*
*University of São Paulo (USP)*
São Paulo, Brazil
fbuzato@ime.usp.br

Alfredo Goldman
*Department of Computer Science*
*University of São Paulo (USP)*
São Paulo, Brazil
gold@ime.usp.br

***Abstract*—This study investigates microservices grouping within containers, exploring its impact on performance, resource utilization, and availability. Experiments using the Sock-Shop microservice benchmark tool across varying workloads revealed benefits in terms of optimizing performance, computational resources and improving availability in scenarios grouping loosely-coupled microservices. However, this approach can incur operational complexity and costs. Therefore, grouping microservices within containers offers potential advantages, but careful consideration is needed to balance benefits and challenges effectively.**

***Index Terms*—Microservices architecture, Benchmark, Containerization, Performance optimization, Resource utilization, Availability, Workload, Kubernetes, Microservices grouping, Operational complexity, Cost analysis**

## I. Introduction

With the advent of technologies and practices such as cloud computing, on-demand virtualization, DevOps and agile methodologies, it was necessary to adapt applications and teams organization for this new scenario. With that, a new architectural style called microservices emerged [1]. In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies [2]. This new architectural style brought several benefits to the applications and developers such as resilience, scalability, technological heterogeneity, organizational alignment and autonomy.

However, the micorservices architectural style has also brought new challenges and complexities for developers and companies that use this architectural style for deliver their applications. The most commons challenges and complexities in the microservices architecture are related to the distributed systems challenges and complexities like deployment, resilience, scalability and communication [3]. Furthermore, other challenges and complexities in this type of architecture are related to the microservices size definition (granularity).

With these complexities and challenges, several design patterns have emerged to be used in microservices architecture and help developers to solve common problems [4]. One of these common problems faced for the developers is related to the microservices deployment model. A microservices based application can have thousands of microservices what makes almost impossible for developers to deploy it manually. To minimize this complexity, some tools and techniques like continuous integration, cloud computing, infrastructure as code, containerization, Docker [1] and Kubernetes [2] began to be used by developers, and one of the most used pattern to deploy microservices in a cloud environment it is the one service per container pattern [4]. This pattern says that each service must be packaged in a container image and each instance of the service must be deployed as a container. The beneficial results of using this pattern can be the following: Ease of horizontal scalability of services, abstraction of technologies used to deploy services, isolation of services and limitation in CPU and memory consumption consumed by each instance of services.

However, some studies [5] [6] [7] have been carried out to assess the impact that the use of containers can have on the performance and resource consumption of applications based on the microservices architecture and concluded that the use of containers can decrease the application performance and increase the network resource consumption.

In addition to the problems related to deploying microservices with containers, there is another major challenge related to microservice architecture, which is the definition of microservice granularity. Some subject matter experts have different approaches to defining the granularity of microservices. At Amazon, a microservice is characterized as an

Thanks for USP and Amazon AWS partnership, CAPES and the project FAPESP 19/26702-8 (https://hpc.ime.usp.br/) for the funding.

[1]https://www.docker.com/ (visited on Jul. 15, 2023)
[2]https://kubernetes.io/ (visited on Jul. 15, 2023)

979-8-3503-8160-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBAC-PADW60351.2023.00027

application that can be maintained by a team of no more than a dozen people (*two pizzas team*) [2]. Jon Eaves characterizes a microservice as an application that can be rewritten in two weeks [1]. [2] and [1] believe that a microservice represents a business capability and the bounded context of *Domain Driven Design* [8] helps to define the size of a microservice and its boundaries.

Therefore, because there are several approaches to defining the size of a microservice and because these approaches are abstract, the size of a microservice can vary depending on the developer or architect who is creating a microservice-based application, that is, in a real scenario it is possible to verify different microservices granularity in an application, some very granular and some not so granular. Because of this, there are also studies [9] [10] that evaluated the impact of defining the microservices granularity in an application and these studies concluded that the decomposition of an application into more granular microservices can decrease the application performance.

Given the problems related to the one service per container deployment pattern and the problems related to the impact on the applications performance with very granular microservices, it is possible that there is a potential problem of performance and resource consumption in applications based on microservices architecture and that are deployed with one service per container.

Thus, this paper aims to investigate, in a real scenario of microservices architecture that use the deployment pattern of one service per container, cases in which the consolidation of more than one microservice within the same container would be interesting, verifying impacts related to performance, resource consumption (computational and financial) and availability of the application. To do this investigation there are some research questions to be answered. The main question that need to be answered in this study is: Are there cases where it is interesting to group microservices in the same container? However, to answer this research question, there is a subset of questions that need to be answered first. So it is necessary to answer the following sub-questions before answer the main research question: (1) How the microservices grouping could impact the application performance? (2) How the microservices grouping could impact the computational and financial resources in a public cloud? (3) How the microservices grouping could impact the application availability?

In order to allow the replication of the experiments, all implemented code is publicly available under the GNU *General Public License* v3.0 license on https://github.com/fernandohlb/microservices-container-grouping.git.

The rest of this text is organized as follows: the Section 2 lists and compares the related works to the purpose of this article; the Section 3 describes the methods used in this study, the criteria used to grouping the microservices and how the workloads were established; the Section 4 describes each experiments, how the benchmark tool were selected, how the microservices were grouped, how the experiments were performed and the results and discussions about them; the Section 5 highlights the conclusions about the study.

## II. Related and previous works

### A. *Microservices with containers performance and computational resources evaluation*

The authors M. Amaral et al. [5] evaluated the computational resource consumption and performance impact that the use of containers could have on a system based on microservices architecture. In the observed results, the use of container did not significantly impact the CPU consumption when compared with virtualized applications without containers. In addition, network consumption in containerized applications and virtualized applications without containers was also very close, except when configurations with Linux Bridge or OpenvSwitch were used in the experiment, which significantly impacted the network consumption of containerized applications. In this case the authors demonstrated that the use of containers with Linux Bridge or OpenvSwitch network configurations can decrease the transfer rate and increase the network latency by about two times. The same way, the author N. Kratzke [6] presented an evaluation about the impact on network performance when there is communication between two services executing in separated containers. On this paper the author executed four experiments with the following configurations: (i) two services communicating directly over the network, without the use of containers, (ii) two services communicating directly over the network, with one container per service, (iii) two services communicating over the network through an unencrypted SDVN (Software Defined Virtual Network), using one container per service, and (iv) two services communicating over the network through an encrypted SDVN, using one container per service. The conclusion reached by the author was that the use of a container can degrade the network throughput performance about 10% to 20%. When using an SDVN (Software Defined Virtual Network) the network throughput performance can degrade about 30% to 70%. In this study the experiments were performed considering the use or not of containers with microservices separately and the scenario with the deployment of more than one microservice inside the same container was not verified.

### B. *Microservices architecture impact on systems performance*

The authors O. Al-Debagy and P. Martinek [11] proposed to conduct an investigation into the performance impact that the microservice architecture can cause in the system in terms of response time and throughput. Basically the author performs some experiments comparing the same system in a monolithic architecture and in a microservice architecture, capturing the results of the response time and throughput metrics. The conclusion of this study was that in monolithic architecture the system presented a better response time for a small number of users, however, for a large number of users the microservices architecture presented a better result than the monolithic architecture. In a concurrency test, the monolithic architecture showed a better throughput than the microservices architecture.

With a similar approach, the authors T. Ueda, T. Nakaike, and M. Ohara [12] investigated the impact in systems performance that the microservices architecture can cause. The authors used a widely used benchmark called ACME Air [3] to execute the experiments. This benchmark simulates a flight reservation system and was executed in a scenario with a microservice architecture and in a scenario with a monolithic architecture. The author also explored the issue of deployment using container and comparing two programming languages Node.JS and Java. Basically, the conclusion reached by the author was that monolithic architecture can be 79% more performative than microservice architecture. However, this result was based only on the observation of throughput, not observing other metrics that are also important when it comes to system performance.

The authors D. Shadija, M. Rezai, and R. Hill [9] presented the question about the microservices granularity and their effects in systems latency. The experiments was made through a university admissions system use case and explored basically two scenarios: in the first scenario, all functionalities was deployed together in the same web server. The second scenario, some of these functionalities was separated in a second web server, simulating a more granular microservice. Some workloads were executed in both scenarios to verify the response time and the conclusion was that there was no significant increase in response time in the second scenario (with more granular microservices).

The same way, the authors M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, and S. Perera [10] investigates the impact on performance that the decomposition of microservices can cause in a system based on a microservice architecture. In the study, the author used some specific benchmarks to evaluate some situations such as Echo-Service to evaluate the decomposition in an I/O investigation scenario and Prime-check to evaluate the decomposition in a CPU usage investigation scenario. The conclusion reached by the authors was that the microservices decomposition linked to I/O can degrade the system's performance when analyzing the throughput and response time. On the other hand, when the decomposition is related to CPU processing, the decomposition is beneficial for the system's performance, increasing throughput and decreasing response time.

### C. Previous Works

In a previous work [7] we followed a different approach than the studies previously presented. In this paper, it was verified whether the resources consumption (I/O, network, CPU and memory) could be optimized depending on the microservice deployment model. To carry out the study, two microservices deployment models with container were verified: (i) One container for each microservice layer and (ii) A single container with all microservice layers. The study showed interesting results, such as the optimization of about 99% in network consumption when all microservice layers were deployed within a single container.

These related and previous works presented that containers can impact directly the system performance and resources consumption in a microservice architecture. Also the microservice architecture as well their granularity can impact directly the system performance.

## III. Methods

In order to carry out the study and check if there were scenarios in which was interesting to group more than one microservice in the same container, the study was executed in two stages, an exploratory stage, in which some inputs were collected to base the experiments, and another experimental stage, in which the experiments were executed and the results were collected and analyzed. In the experimental stage a microservices benchmark tool was used to run the experiments. This tool provides an application based on the microservice architecture and tools for executing the workload and capturing the results.

### A. Exploratory Stage

At this stage, systematic bibliographical reviews were carried out in order to find answers, even if incomplete, to evaluate some scenarios where it would be interesting to group more than one microservice in the same container. The main works analyzed in this stage, as well as their results, were presented in the section II.

Still in this stage, a pre selection of two *benchmark* tools that was used in the experiments was carried out. First, it was selected six microservices benchmark tools that were used in important published scientific papers: $\mu$Suite [13], NDBench [14], DeathstarBench [15], TeaStore [16], ACME Air benchmark [12] and Sock Shop [17] [15]. After, it was defined some requirements to be evaluated and filter the two best qualified benchmark tools. The requirements evaluated was based on the authors C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi work [18] and was adapted to the study necessities. So the benchmark tools were evaluated considering the following requirements: R1 - Explicit Topological View, R2 - Pattern-based distributed architecture, R3 - Easy Access from a Version Control Repository, R4 - Support for deployment in multiple environments, R5 - Container orchestration support, R6 - Alternate Versions, R7 - Community Usage and Interest, R8 - Configurable workloads for testing, R9 - Metrics generation for application performance, R10 - Metrics generation for application resource consumption, R11 - Reports/Visualization of results and R12 - Scenarios for running the benchmark. After defining the requirements, each benchmark were evaluated according to the requirements fulfillment and a score were attributed to each benchmark. After calculating the final scores for each microservice benchmark tool and classified them, the Sock Shop and DeathstarBench benchmark tools were selected to be evaluated in the experimental stage.

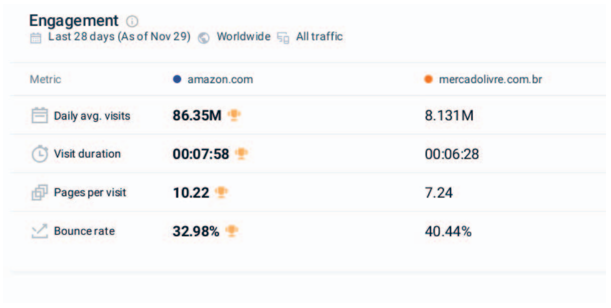
[3] https://github.com/acmeair/

## B. Experimental Stage

After the exploratory stage III-A, 2 experiments were executed: One to select the benchmark tool and other experiment to compare the microservices grouping scenarios and capture the results about performance, resources consumption and availability.

To define the microservices grouping scenarios, 4 criteria were considered: The first, was considering the benchmark itself in which there was no microservice grouping in containers and each microservice was executed in its own container. The second, was considering the opposite of the benchmark, where all the application's microservices were grouped in a single container. The third, was considering the dependency between the microservices, where the microservices that had more dependency on each other were grouped into a single container. And the fourth and last one, was considering the execution stack of each microservice, where the microservices that execute with the same stack were grouped in the same container. Therefore, 4 execution scenarios were defined to be compared in each experiment: Benchmark, All-in-one, By-dependencies and By-stack.

In this study it was established 3 workloads to be executed: a small workload with few users accessing the application, a high workload with lot of users accessing the application and a medium workload with the average number of users considering the small and high workloads. As the application family provided by the benchmark tool selected in the first experiment is an e-commerce, workloads from large e-commerces used nowadays by users around the world were considered to define the number of users in each workload for the experiments. The Fig. 1 shows the average number of users on a day that accessed these e-commerces during the period between November 29, 2022 and December 2, 2022. These data was consulted in the website Similarweb [4]. Thus, for the maximum workload, the Amazon.com [5] workload was considered, and for the minimum workload, the Mercadolivre.com.br [6] workload was considered. To calculate the number of users of each workload in the experiments, the number of users of each e-commerce was linearly distributed during the 24-hour period. Thus, for example, Amazon's workload presented an average amount of approximately 60000 users accessing the platform every minute. From this number, users who visited a single page on the websites before leaving (Bounce rate) were excluded, therefore, in the example of Amazon, the workload considered was 40200 users per minute. It was also considered a load balance factor for this workload on these websites, so for this study the factor considered was 100 processing replicas and therefore the Amazon workload would be 402 users per minute for each replica. For this study, the high workload based on this Amazon workload was 450 users per minute for a single replica of the services. A processing period of 10 minutes was also considered for each workload, in which the first 5 minutes were used for processing and entering new users in the application and the final 5 minutes only for processing with the maximum number of users. Therefore, in the high workload, 4500 users were processed in total, with a spawn rate of 15 users per second during the first 5 minutes and during the final 5 minutes with the maximum number of simultaneous users in the application. The low and medium workloads followed the same method as the high workload, however, the low workload was calculated considering the Mercado Livre workload and the medium workload was calculated by the average of the low and high workloads. The workloads details it is presented in the Table I


Engagement
Last 28 days (As of Nov 29) Worldwide All traffic

| Metric | ● amazon.com | ● mercadolivre.com.br |
|---|---|---|
| Daily avg. visits | 86.35M | 8.131M |
| Visit duration | 00:07:58 | 00:06:28 |
| Pages per visit | 10.22 | 7.24 |
| Bounce rate | 32.98% | 40.44% |

Fig. 1: E-commerces workloads

TABLE I: Table Experiments Workloads

| Workloads | Users | Spawn rate |
|---|---|---|
| Small | 300 | 1 user/second |
| Medium | 2400 | 8 users/second |
| High | 4500 | 15 users/second |

## IV. Experiments and Results

In this study two experiments were executed to verify cases in which the consolidation of more than one microservice within the same container would be interesting. In the subsection III-B were presented the methods used in each one experiment and the following sections will present the experiments details the results and discussions about the results.

### A. Benchmark tool selection

To selected the benchmark tool it was executed a simplified experiment in a local machine comparing the two best qualified benchmark tools that were presented in the subsection III-A. So it was compared the Sock-Shop benchmark tool and the DeathstarBench benchmark tool. The experiment was executed in a local kubernetes clustes using the Minikube tool [7] and the microservices benchmark tools were compared considering four requirements: Ease of deployment, Alternate Version, Workload customization and results completeness in the generated reports. Some of these requirements are part of the requirements presented in the subsection III-A. Comparing the deployments between DeathstarBench and SockShop, both

[4]https://www.similarweb.com/ (visited on Jul. 26, 2023)
[5]https://www.amazon.com/ (visited on Jul. 26, 2023)
[6]https://www.mercadolivre.com.br/ (visited on Jul. 26, 2023)

[7]https://minikube.sigs.k8s.io/docs/ (visited on Jul. 26, 2023)

provided a very similar deployment model in which, the tools provide kubernetes manifests that can be executed by the kubectl command to create services and deployments inside the kubernetes cluster. When comparing the tools by Alternate Versions requirements, the Sock Shop benchmark tool presented a more heterogeneous architecture than the DeathstarBench benchmark tool. The DeathstarBench tool provides three applications, however, each of these applications are homogeneous in relation to the language in which each microservice was built, for example, the hotel reservation application, has microservices all written in GO. On the other hand, the Sock Shop application, despite providing only a single application scenario, has heterogeneous microservices in terms of language. Therefore, the Sock Shop application is built using microservices in NodeJS, Go and Java. Regarding the workload, both tools provide scripts that simulate users' navigation within the system and both tools allow for customizations in these scripts. However, the Sock Shop tool uses the Locust.io [8] tool as a test tool, which has a greater dynamism to parameterize workloads, such as the users spawn rate. In addition, the Locust.io tool allows the export of various performance information to csv and html files with graphics, while the *wrk2* tool used in DeathstarBench, exports the data only to the terminal where the workload is being executed. So, after running locally the benchmark tools and comparing the results, it was concluded that the Sock Shop benchmark tool obtained better results for the evaluated requirements and therefore was chosen for the other experiments.

## B. Workload execution experiments

Before running the workloads, it was necessary to develop and create the docker images for each grouping scenario. Considering the Sock-Shop benchmark tool, there are 8 microservices and 5 infra-services. Following the microservices grouping criteria presented in the subsection III-A, the docker image for the all-in-one scenario were developed consolidating the 8 microservices from the benchmark tool in just one docker image. The docker images for the by-stack scenario were developed consolidating 3 microservices in the by-stack-go docker image, 4 microservices in the by-stack-jvm docker image and 1 microservice in the by-stack-node docker image. The docker images for the by-dependencies scenario were developed consolidating 3 microservices in the by-dependencies docker image and the other 5 microservices were kept in their own images as in the benchmark tool. In addition to grouping the microservices in each docker image, it was necessary to calibrate the resources in the Kubernetes deployments. So, based on the cpu and memory requests and limits in each microservice deployment in the benchmark scenario, it was necessary to sum the cpu and memory requests and limits in the deployments with docker images that grouped microservices. For example, in the benchmark scenario the Orders deployment had 275Mb of requested memory and 550Mb of limit memory, the Shipping deployment had 200Mb of requested memory and 400Mb of limit memory and the Payment deployment had 15Mb of requested memory and 30Mb of limit memory. The by-dependencies docker image consolidated the Orders, Shipping and Payment microservices, so the by-dependencies deployment had 490Mb of requested memory and 980Mb of limit memory. The Table II presents the information about the docker images consolidation and the resources for each deployment.

|  |  |  | **Resources Metrics** |  |  |  |
|---|---|---|---|---|---|---|
|  |  |  | **CPU Milicores** |  | **Memory Mb** |  |
| **Scenarios** | **Microservices** | **Containers** | **Requested** | **Limit** | **Requested** | **Limit** |
| Benchmark | Carts | Carts | 460 | 920 | 275 | 550 |
|  | Catalogue | Catalogue | 30 | 60 | 15 | 30 |
|  | Front-end | Front-end | 465 | 930 | 340 | 680 |
|  | Orders | Orders | 400 | 800 | 275 | 550 |
|  | Payment | Payment | 15 | 30 | 15 | 30 |
|  | Queue | Queue | 150 | 300 | 200 | 400 |
|  | Shipping | Shipping | 150 | 300 | 200 | 400 |
|  | User | User | 150 | 300 | 50 | 100 |
| All-in-one | Carts<br>Catalogue<br>Front-end<br>Orders<br>Payment<br>Queue<br>Shipping<br>User | All-in-one | 1820 | 3640 | 1370 | 2740 |
| By-Stack | Catalogue<br>Payment<br>User | bystack-go | 195 | 390 | 80 | 160 |
|  | Carts<br>Orders<br>Queue<br>Shipping | bystack-jvm | 1160 | 2320 | 950 | 1900 |
|  | Front-end | by-stack-node | 465 | 930 | 340 | 680 |
| By-Dependencies | Payment<br>Orders<br>Shipping | by-dependencies | 565 | 1130 | 490 | 980 |
|  | Carts | Carts | 460 | 920 | 275 | 550 |
|  | Catalogue | Catalogue | 30 | 60 | 15 | 30 |
|  | Front-End | Front-End | 465 | 930 | 340 | 680 |
|  | Queue | Queue | 150 | 300 | 200 | 400 |
|  | User | User | 150 | 300 | 50 | 100 |

TABLE II: Microservices per Container and Kubernetes deployment resource requested and limits

After develop the docker images and configure the Kubernentes deployments, the experimentes were executed. All experiments were executed in the Amazon AWS with an EKS cluster, Kubernetes 1.22 and Docker Engine 20.10.23. It was created one Kubernetes nodegroup for each microservices grouping scenario and each nodegroup had just one node. This configuration was necessary to evict the interference from the network in the communication between the services inter nodes. Each node used in each scenario was created using EC2 instances. The instances flavors used in this study was a t3.xlarge with 4 Intel Xeon Scalable processor vCPUs of 3.1 GHz, with 16Gb ram memory and with 50Gb of EBS-Storage disk. In addition to the nodes for each scenario, we created three accessory nodegroups for the experiments: one for the load testing tool, for executing workloads, one for the application monitoring tool to capture resource consumption and performance metrics, and a third node for the financial resource consumption monitoring tool. The results related to performance were generated generated in html and csv files from locust tool and the results related to resource consumption, cost and availability were generated in csv files from Grafana's dashboards. The Fig. 2 shows the experiments architecture design.

The experiments were executed from a bash file that sent the configurations of each workload to the Locust tool and controlled the beginning of the execution of the tests. Each workload ran the 4 microservices grouping scenarios presented

[8]https://locust.io/ (visited on Jul. 26, 2023)

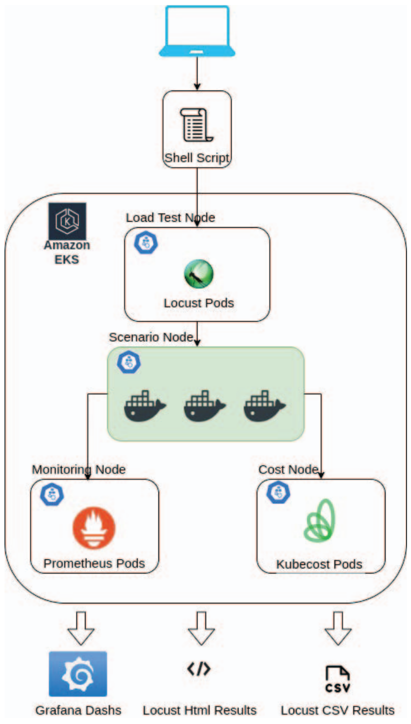
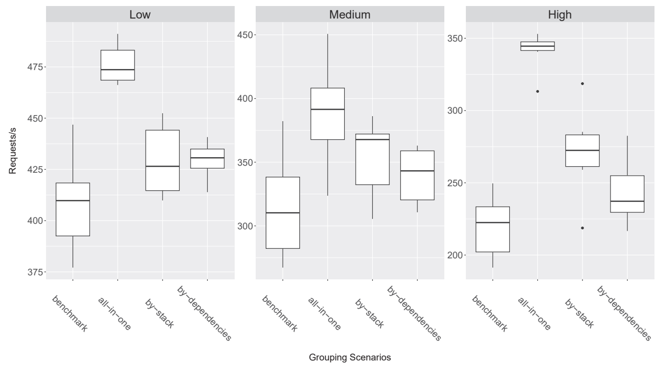


Fig. 2: Experiments Architecture

in the subsection III-B, and each scenario ran for 10 minutes. Therefore, each workload took at least 40 minutes to run all scenarios. Due to cost and execution time, 6 samples were run for each workload. After performing the experiments, the results of the six samples were consolidated and analyzed using the boxplot due to the small number of samples and the lack of data normality.

*1) Results:* One of the mainly results captured in the experiments was the results related to the application performance. The Fig. 3 represents the boxplots generated from the results and the Table III presents the metrics and the median results for each scenario and workload. When comparing the benchmark scenario with other scenarios, it was possible to verify that the scenarios with some microservices grouping had better performance than the benchmark scenario. For example, in the medium workload, with 2400 users, the All-in-One scenario presented a throughput about 26% better than the benchmark scenario and a response time about 20.5% better than the benchmark scenario. The other scenarios also presented expressive improvement in the performance metrics compared to the benchmark scenario. The By-stack scenario presented a throughput improvement about 18% and a latency improvement about 15.5%. The By-dependencies presented a throughput improvement about 10% and a latency improvement about 9.4%. The error rate in this workload were very similar between the scenarios, but in the high workload the All-in-One scenario was the only one that presented a better error rate than the Benchmark scenario. Even with worst error rate than the Benchmark, the scenarios By-stack and By-dependencies presented better response time and throughput than the Benchmark scenario, which showed that the error rate did not significantly interfered in the performance.



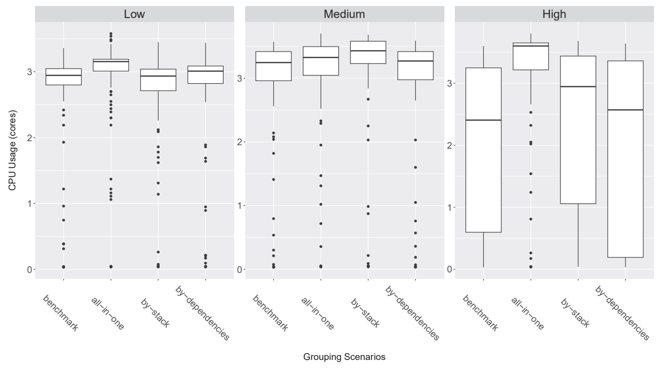
(a) Requests/s

Fig. 3: Performance results - Comparison between scenarios in different workloads

| | | Scenarios | | | |
|---|---|---|---|---|---|
| **Workload** | **Metrics** | **Benchmark** | **All-In-One** | **By-Stack** | **By-Dependencies** |
| Low | Throughput (Requests/s) | 409.76 | 473.66 | 426.51 | 430.65 |
| | Latency (ms) | 547.51 | 473.95 | 525.79 | 520.40 |
| | Error Rate (Failures/s) | 0.0100 | 0.0033 | 0.0092 | 0.0083 |
| Medium | Throughput (Requests/s) | 310.26 | 391.47 | 367.75 | 343.18 |
| | Latency (ms) | 5,608.42 | 4,459.34 | 4,737.41 | 5,082.69 |
| | Error Rate (Failures/s) | 0.0450 | 0.0558 | 0.0524 | 0.0458 |
| High | Throughput (Requests/s) | 222.48 | 344.55 | 272.45 | 237.24 |
| | Latency (ms) | 14,035.36 | 9,211.59 | 11,479.73 | 13,464.91 |
| | Error Rate (Failures/s) | 39.37 | 36.51 | 44.15 | 46.99 |

TABLE III: Performance metrics median results

The Fig 4 presents the CPU boxplots results and the Table IV presents all computational resources consumption median results. The scenarios with microservices grouping presented higher CPU consumption than the benchmark scenario in all workloads. On the other hand, scenarios with microservices grouping presented lower memory consumption than the benchmark scenario in all workloads. The high CPU consumption in the scenarios with microservices grouping can be justified by the requested and limits resource configuration in each deployment, mainly in the all-in-one scenario that grouped the Front-End microservices in the same container. But even with high requested and limits configuration the memory presented an opposite behavior, which means that there was a memory consumption optimization in scenarios with microservices grouping. Considering the disk usage metric, there was expressive results in the low and medium workloads. In the low workload, the disk usage presented an optimization about 40%, 20% and 15% in the scenarios all-in-one, by-stack and by-dependencies, respectively and in the medium workload the all-in-one and by-stack presented an optimization about 35% and 21%, respectively. In the high workload, there were no optimizations in this resource, mainly due to the availability that was affected in each scenario. In the network metric was possible to verify an expressive optimization just in the all-in-one scenario. In the low workload and medium workload, this scenario presented an optimization about 15% and 6%, respectively.



(a) Cpu Consumption (cores)

Fig. 4: Computational Resources Consumption results - Comparison between scenarios in different workloads

| | | Scenarios | | | |
|---|---|---|---|---|---|
| Workload | Metrics | Benchmark | All-In-One | By-Stack | By-Dependencies |
| Low | CPU (cores) | 2.95 | 3.16 | 2.94 | 3.01 |
| | Memory (Gb) | 2.42 | 2.25 | 2.38 | 2.42 |
| | Network (Mb/s) | 30.63 | 25.96 | 30.16 | 30.33 |
| | Disk (Mb) | 163.74 | 98.74 | 132.94 | 141.58 |
| Medium | CPU (cores) | 3.25 | 3.33 | 3.43 | 3.27 |
| | Memory (Gb) | 2.74 | 2.66 | 2.70 | 2.74 |
| | Network (Mb/s) | 21.65 | 20.30 | 23.08 | 21.85 |
| | Disk (Mb) | 179.41 | 115.08 | 140.99 | 183.31 |
| High | CPU (cores) | 2.41 | 3.60 | 2.95 | 2.57 |
| | Memory (Gb) | 3.03 | 2.86 | 2.84 | 2.88 |
| | Network (Mb/s) | 13.61 | 17.79 | 18.30 | 14.32 |
| | Disk (Mb) | 100.19 | 124.82 | 122.38 | 123.32 |

TABLE IV: Computational Resources consumption median results

The table V presents the financial costs results. The financial cost estimates are based on cpu, memory, and persistent volume allocation costs. The scenarios used in this study didn't use persistence storage, so the monthly cost was directly impacted by the cpu and memory consumption in each scenario. The CPU hourly price it is considerably more expensive than the memory price and because of it the financial monthly costs results were very similar to the cpu results. Observing the results, it was possible to verify that the benchmark scenario presented a lower cost than the other scenarios in the low and high workloads. In the medium workload, the scenarios all-in-one and by-dependencies presented lower costs than the benchmark scenario.

| | | Scenarios | | | |
|---|---|---|---|---|---|
| Workload | Metrics | Benchmark | All-In-One | By-Stack | By-Dependencies |
| Low | Monthly Cost (U$) | 109.95 | 111.02 | 111.88 | 114.3 |
| Medium | Monthly Cost (U$) | 123.36 | 121.57 | 130.49 | 119.85 |
| High | Monthly Cost (U$) | 100.92 | 127.56 | 121.97 | 71.67 |

TABLE V: Estimated Monthly Costs median results

Another important result that was analyzed in this study was about the scenarios availability. The availability can be verified by two aspects: The availability by error and the availability by time. The first, refers to the number of errors that occurred when the application received the call and was unable to process and consequently was unable to respond to the client. The second, refers to the time that the application was down and could not receive requests from the client. The second one can affect directly the first one. The availability in both cases can be calculate by the following formulas:

$$AvailabilityByError = \frac{(TotalRequests - TotalErrors)}{TotalRequests} \tag{1}$$

$$AvailabilityByTime = \frac{(TotalTime - TotalDownTime)}{TotalTime} \tag{2}$$

The availability results can be verified in the Table VI. In the low workload all scenarios presented an availability by error of 100%. In the medium workload all scenarios presented a similar availability: The scenarios benchmark, all-in-one and by-dependencies presented an availability by error of 99.99% and the scenario by-stack presented an availability of 99.98%. In the high workload, the scenarios presented a large variation in the availability by error between them. Comparing the differences of the availability by error with the benchmark, the all-in-one and by-stack presented better results than the benchmark and the by-dependencies presented a worst result than the benchmark scenario. In this workload, the availability by error was impacted directly by the availability by time. The availability by time in the low and medium workloads was 100% in all scenarios because in these workloads didn't occurred pod restarts during the experiments. But in the high workload occurred pod restarts in all scenarios and these pod restarts affected the scenario availability. Comparing the differences of the availability by time with the benchmark also the all-in-one and by-stack presented better results than the benchmark scenario and the by-dependencies presented a worst result than the benchmark scenario.

| | | Scenarios | | | |
|---|---|---|---|---|---|
| Workload | Metrics | Benchmark | All-In-One | By-Stack | By-Dependencies |
| Low | Availability by Error (%) | 100.00 | 100.00 | 100.00 | 100.00 |
| | Availability by Time (%) | 100.00 | 100.00 | 100.00 | 100.00 |
| Medium | Availability by Error (%) | 99.99 | 99.99 | 99.98 | 99.99 |
| | Availability by Time (%) | 100.00 | 100.00 | 100.00 | 100.00 |
| High | Availability by Error (%) | 81.35 | 89.38 | 84.30 | 80.30 |
| | Availability by Time (%) | 63.75 | 95.00 | 66.25 | 55.00 |

TABLE VI: Availability median results

## V. Discussions

The outcomes of the conducted experiments offer a valuable perspective on identifying the most suitable scenarios for microservices grouping within containerized deployments. These findings not only help organizations harness the advantages of microservices architecture but also guide them in optimizing performance, resource utilization, availability, and operational costs.

In a low workload all scenarios presented similar performance metrics. The all-in-one scenario and the by-dependencies presented best performance metrics. On the other hand, these scenarios presented higher cpu and memory consumption that affects directly the financial costs in a public cloud. The benchmark scenario presented the best resource consumption and the lowest financial cost. But this scenario presented the worst performance metric. The by-stack scenario in these workload presented better performance than the benchmark and worst performance than the all-in-one and by-stack scenarios. But this scenario presented a similar financial cost and similar cpu and memory consumption compared to benchmark scenario. Therefore, in the low workload, the by-stack microservices grouping could be a good approach for applications that needs a good performance, a balanced resource consumption and a low financial cost.

With a medium workload, the all-in-one and by-stack scenarios presented best performance metrics and lowest disk consumption. Also the all-in-one scenario presented a better financial cost and a lower memory consumption compared to the benchmark scenario. On the other hand, the by-stack scenario presented high cpu, memory and network consumption which led to a high financial cost. The by-dependencies presented the lowest financial cost and the lowest cpu and memory consumption. But this scenario did not present a good performance and this scenario presented the highest disk consumption. Considering all these facts, in the medium workload the all-in-one could be a good approach for applications that needs a good performance and a balanced resource consumption. On the other hand, if the main requirement it is a low financial cost, the by-dependencies scenario could be a better choice.

Under heavy workloads, the all-in-one scenario also presented best performance metrics and higher availability than the other scenarios. But the resource consumption in this scenario also was higher than the other scenarios. This high resource consumption may have been influenced by the scenario availability. The scenario by-stack was the second with the highest availability. Therefore, it was also the second highest in resource consumption and financial cost. The benchmark scenario presented the worst performance metrics, but it presented a higher availability than the by-dependencies scenario. The by-dependencies scenario presented the worst availability and consequently the lowest resource consumption and financial costs. But, even with a lower availability than the benchmark scenario, this scenario presented higher performance metrics than the benchmark. Therefore, in a heavy workload the all-in-one or by-stack scenarios could be good choices because the high availability and performance metrics.

## VI. Conclusion

The microservices architecture and the deployment model using containers are widely used in the industry due to the ease of use of this model with the microservices architecture. Through the results obtained in this study, it was possible to verify that the microservices grouping in some scenarios can bring benefits mainly in terms of performance and computational resources optimizations such as memory, network and disk. Furthermore, this approach can improve application availability in scenarios where we grouped microservices that are not highly dependent. However, microservices grouping in the same container can be an operationally costly process, in addition to not bringing significant optimizations in terms of financial costs. So, a possible future work would be doing the same study with the Kubernetes horizontal pod auto scaling and verify the results and the scenarios behavior in the high workload.

## References

[1] S. Newman, *Building microservices: designing fine-grained systems.* " O'Reilly Media, Inc.", 2015.

[2] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term."

[3] M. Van Steen and A. S. Tanenbaum, *Distributed systems.* Maarten van Steen Leiden, The Netherlands, 2017.

[4] C. Richardson, *Microservices patterns.* Manning Publications Company,, 2018.

[5] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on.* IEEE, 2015, pp. 27–34.

[6] N. Kratzke, "About Microservices, Containers and their Underestimated Impact on Network Performance," *Proceedings of CLOUD COMPUTING*, vol. 2015, pp. 165–169, 2015.

[7] F. H. Buzato, A. Goldman, and D. Batista, "Efficient resources utilization by different microservices deployment models," in *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA).* IEEE, 2018, pp. 1–4.

[8] E. Evans, *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[9] D. Shadija, M. Rezai, and R. Hill, "Microservices: granularity vs. performance," in *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, 2017, pp. 215–220.

[10] M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, and S. Perera, "An analysis of throughput and latency behaviours under microservice decomposition," in *International Conference on Web Engineering.* Springer, 2020, pp. 53–69.

[11] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI).* IEEE, 2018, pp. 000 149–000 154.

[12] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE international symposium on workload characterization (IISWC).* IEEE, 2016, pp. 1–10.

[13] A. Sriraman and T. F. Wenisch, "$\mu$ suite: a benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2018, pp. 1–12.

[14] I. Papapanagiotou and V. Chella, "Ndbench: Benchmarking microservices at scale," *arXiv preprint arXiv:1807.10792*, 2018.

[15] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18.

[16] 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 *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* IEEE, 2018, pp. 223–236.

[17] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, "Benchmarking microservice performance: a pattern-based approach," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 232–241.

[18] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE).* IEEE, 2017, pp. 8–13.