International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 16, Number 2, pages 52–75, January 2026

Extended version of Microservices Performance Optimization Through Horizontal Pod Autoscaling: A Comprehensive Study

Fernando H. L. Buzato
University of São Paulo (USP), Instituto de Matemática e Estatística (IME), Department of Computer Science, Rua do Matão, 1010
São Paulo, São Paulo, 05508-090, Brazil

Alfredo Goldman
University of São Paulo (USP), Instituto de Matemática e Estatística (IME), Department of Computer Science, Rua do Matão, 1010
São Paulo, São Paulo, 05508-090, Brazil

Received: July 30, 2025
Accepted: September 12, 2025
Communicated by Masahiro Shibata

Abstract

This paper investigates the integration of Horizontal Pod Autoscaling (HPA) into containerized microservices architectures, focusing on optimizing performance, resource utilization, and operational costs. Building upon prior research on microservices grouping strategies, experiments were conducted using the Sock-Shop benchmark tool across low, medium, and high workload scenarios. Results reveal that HPA significantly enhances scalability, throughput, and latency—achieving up to 66% improved throughput and 32% reduced response times under certain grouping strategies. However, these advantages come with trade-offs, such as increased disk usage and operational complexity. This study provides a detailed analysis of HPA's impact on dynamic environments and offers practical recommendations for balancing performance and cost in deployment strategies. Future research directions include exploring alternative scaling models, diverse workload impacts, serverless integration, and operational simplifications.

Keywords: Microservices, Containers, Autoscaling, HPA, Resource Optimization, Cloud Computing, Benchmark Tools, Performance, Resource Consumption, Availability, Scalability

Preliminary Considerations

This is an extended version of the paper Extending Microservices Performance Optimization Through Horizontal Pod Autoscaling: A Comprehensive Study [3]. In addition to the original findings, several enhancements and new insights were introduced in this version to strengthen the analysis and provide more practical guidance. The main additions are highlighted below:

- **Correlation Analysis:** This version includes a detailed correlation analysis between key performance metrics—such as CPU usage, memory usage, response time, and financial cost—across different workload scenarios and grouping strategies. This analysis aims to uncover deeper performance patterns and support more informed decisions.
- **New Section on Practical Insights:** A dedicated section was added to present practical insights on microservice grouping strategies. This section synthesizes the experimental findings into actionable recommendations for software architects and DevOps engineers.
- **Decision Tree for Grouping Strategy:** A visual decision tree was introduced to assist practitioners in choosing the most appropriate microservice grouping strategy, based on the specific context and workload characteristics. This tree is grounded on the experimental data collected in this study.

These enhancements aim to transform the previous work into a more robust and applicable resource for both academic and industry contexts.

1 Introduction

Microservices architectures have emerged as a cornerstone of modern software development [9], transforming the way applications are built and deployed. By breaking down complex systems into smaller, independently deployable services, microservices enable developers to achieve scalability, resilience, and technological flexibility [5]. Each service focuses on a specific business function, allowing for modularity and ease of updates without disrupting the entire system.

Despite these advantages, microservices come with significant challenges. As the number of services in a system grows, so does the complexity of managing inter-service communication, resource allocation, and deployment processes [14]. The decentralized nature of microservices often results in resource inefficiencies, increased network overhead, and performance bottlenecks, particularly under fluctuating workloads [11] [6].

Containerization has emerged as a solution to some of these challenges, providing a lightweight environment to deploy microservices. Tools such as Docker [1] and Kubernetes [2] have revolutionized deployment practices by enabling services to run in isolated environments with predictable resource requirements. However, while containers simplify deployment and scaling, they also introduce new performance trade-offs, especially in network-intensive applications [7] [10] [8] [4].

Horizontal Pod Autoscaling (HPA) in Kubernetes extends the scalability of containerized microservices by dynamically adjusting the number of pod instances based on real-time resource usage. By monitoring metrics such as CPU and memory utilization, HPA ensures that applications can handle sudden spikes in demand without manual intervention. This capability is particularly critical in industries with unpredictable traffic patterns, such as e-commerce and streaming services.

Prior research has explored the benefits of containerized microservices and static scaling strategies. Studies have shown that grouping related services within the same container can reduce inter-service communication overhead, leading to performance gains. However, the lack of dynamic scaling mechanisms in these studies limits their applicability in environments with highly variable workloads [2]. Therefore, this study aims to evaluate cases in which consolidating more than one microservice within the same container is beneficial in auto-scaling environments, combining the advantages identified in the previous study with the dynamic scalability of HPA. This study also has the following key contributions:

- A general understanding of the impact of using HPA in microservices-based applications and the effect of grouping multiple microservices within the same container in environments with HPA enabled;
- Analyzing the impact of HPA on application performance in scenarios where multiple microservices are grouped within the same container;
- Evaluating the impact of HPA on application resource consumption in scenarios where multiple microservices are grouped within the same container;
- Assessing the impact of HPA on application availability in scenarios where multiple microservices are grouped within the same container;
- Understanding the impact of grouping microservices within the same container on the application's horizontal scalability.

The study focuses on real-world scenarios using the Sock-Shop benchmark tool [3], a simulated e-commerce application, to evaluate the performance impact of HPA. The experimental setup incorporates a range of workload scenarios, from low user activity to peak demand, to assess the effectiveness of HPA across different traffic patterns. Metrics related to the application performance (throughput, latency, error rate), to the application resource consumption (CPU, memory, network, disk and financial costs) and to the application scalability and availability were analyzed to

[1]`https://www.docker.com/` (visited on Dez. 29, 2024)
[2]`https://kubernetes.io/` (visited on Dez. 29, 2024)
[3]`https://github.com/microservices-demo` (visited on Dez. 29, 2024)

provide a holistic view of the benefits and trade-offs of integrating HPA with microservices grouping. 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`.

This paper is organized as follows: Section 2 lists and compares the related work to the purpose of this paper. Section 3 describes the experimental methodology, including workload configurations and evaluation metrics. Section 4 presents the details about the experiments and how they were configured to be executed. Section 5 presents the results, highlighting the performance, resource utilization, and cost implications of HPA and the correlation between them. Section 6 discusses the findings, and the impacts of the microservices grouping strategies with HPA. Section 7 presents practical insights on microservices grouping. Finally, Section 8 concludes the paper and outlines directions for future research.

2 Related Work

2.1 Microservices architecture impact on systems performance

O. Al-Debagy and P. Martinek [1] proposed to conduct an investigation of the performance impact that microservice architecture can cause in the system in terms of response time and throughput. Basically, the authors performed some experiments that compare the same system in a monolithic architecture and in a microservice architecture, capturing the results of response time and throughput metrics. The conclusion of this study was that in monolithic architecture the system presented a better response time for a few users, however, for many users, the microservice architecture presented a better result than the monolithic architecture. In a concurrency test, the monolithic architecture showed better throughput than the microservice architecture.

Using a similar approach, T. Ueda, T. Nakaike and M. Ohara [13] investigated the impact on system performance that the microservice architecture can cause. The authors used a benchmark tool called ACME Air [4] 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. They also explored the issue of deployment using containers and compared two programming languages, Node.js and Java. Basically, the conclusion reached by the authors was that monolithic architecture can be 79% more performative in terms of throughput than microservice architecture.

M. Jayasinghe, et al. [6] investigated the impact of microservice decomposition on the performance of systems based on a microservice architecture. In their study, the authors used specific benchmark tools to evaluate different scenarios. For example, they used Echo-Service to investigate the impact of decomposition in an I/O-bound scenario and Prime-check to evaluate CPU usage. The Echo-Service represents an I/O-bound microservice, simply echoing back the request sent to it [12]. The Prime-check tool is part of the well-known sysbench benchmark [5], which evaluates the performance of CPU-bound applications in multithreaded systems.

The authors concluded that microservice decomposition linked to I/O operations can degrade system performance, reducing throughput and increasing response time. Conversely, when decomposition is related to CPU processing, it improves system performance by increasing throughput and decreasing response time.

These related work were important for our study, as they provided a general understanding of how the decomposition of an application into microservices could impact key performance metrics, such as response time and throughput. This supports one of the challenges identified with this type of architecture: the potential degradation of application performance in certain contexts using microservice architecture.

In addition, these works allowed us to identify some gaps in relation to our study. Although the works evaluated the impact on performance metrics, they did not assess other important metrics,

[4]`https://github.com/acmeair/` (visited on Dec. 29, 2024)

[5]`https://github.com/akopytov/sysbench` (visited on Dec. 29, 2024)

such as the consumption of computational and financial resources. Furthermore, no metrics related to the availability and scalability of the application were considered. Another significant gap was that these studies did not use a real application or a scenario based on a microservice architecture provided by a benchmarking tool. These related work basically presented that the microservice architecture and their granularity can directly impact the system performance. Because of it, the microservice architecture needs to be carefully designed.

2.2 Performance evaluation of applications with containers

N. Kratzke [7] presented an evaluation of the impact on network performance when communication occurs between two services running in separate containers. In this work, the author conducted 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;
- (iv) two services communicating over the network through an encrypted SDVN, using one container per service.

The author concluded that the use of containers can degrade network throughput performance by about 10% to 20%. When using an SDVN (Software Defined Virtual Network), the network throughput performance can degrade by about 30% to 70%.

C. Ruiz, E. Jeanvoine, and L. Nussbaum [10] conducted a performance evaluation of containers for HPC (High-Performance Computing). One of the evaluations in their study assessed the impact of moving an HPC workload, with several MPI (Message Passing Interface) processes per machine, to containers. They found that, for some benchmarks used in the experiments, the use of containers introduced an overhead in the execution time of MPI processes of approximately 180%, due to message congestion in the virtual network layer used by the containers. Even after switching the network layer to Linux's macvlan, SR-IOV, or OpenvSwitch, the overhead remained as high as 24% compared to the scenario without container usage.

Although the focus was not on microservices applications, K. Lee, Y. Kim, and C. Yoo, [8] evaluated the impact of container usage on IoT (Internet of Things) devices. They conducted two experiments: the first compared network throughput without containers and with containers using the default network (Bridge), while the second compared network throughput when using one container versus multiple containers. In the first experiment, the authors observed a degradation of the network of 6% to 18% when using containers. In the second, they found a network degradation of approximately 10% when using more than four containers.

These related work presented that containers can directly impact system performance and resource consumption in a microservice architecture. Some of these papers used benchmarks to carry out the experiments, but none of them simulated a real microservice system scenario.

2.3 Previous work

In a previous work [4], we adopted a different approach from the studies previously presented. In that study, we investigated whether resource consumption (I/O, network, CPU, and memory) could be optimized depending on the microservice deployment model. To conduct the research, we evaluated two microservice deployment models with containers: (i) One container for each microservice layer, and (ii) A single container containing all microservice layers.

The study yielded interesting results, such as a 99% optimization in network consumption when all microservice layers were deployed within a single container. However, this work did not assess the impact on application performance nor explore the potential resource optimizations when multiple microservices are deployed within the same container.

We also did another study [2] were we observed in an experiment without horizontal scalability that grouping microservices into the same container, in certain scenarios, provided some benefits, particularly in terms of performance and optimization of computational resources, such as memory, network, and disk. However, the grouping of microservices proved to be an operationally costly process and did not yield significant financial optimizations.

Although these studies have evaluated the grouping of microservices (or part of a microservice) within the same container and have evaluated metrics related to performance and resource consumption, none of them have evaluated the results with horizontal scalability of these containers.

3 Methods

The methods followed in this study are the same as those used in the previous study, which consisted of 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. What we did in this study, in addition to what was conducted in the previous one, was to include an additional experiment in the experimental stage, enabling the use of HPA to evaluate the same metrics and microservice grouping scenarios with autoscaling enabled.

This complemented the necessary analyzes to validate whether there were scenarios in which it was beneficial to group more than one microservice in the same container.

3.1 Experimental Setup

To conduct our experiments in this study, we used the same microservices grouping scenarios and benchmarking tool from the previous study. The benchmark tool that we used was the Sock-Shop benchmark, and this benchmarking tool provides an e-commerce application scenario based on microservices with 8 microservices and 5 infrastructure services. The Fig 1 presents an overview of the architecture of this application. The microservices in this application are developed using Java, Go, and Node.js. They communicate via REST APIs and asynchronously through message queues. Additionally, they rely on MySQL, MongoDB, and Redis databases for data storage.

To define the microservices grouping scenarios, 4 criteria were considered:

1. The first scenario defined was the benchmark scenario itself in which there was no microservice grouping in containers and each microservice was executed in its own container.

2. The second defined scenario was the opposite scenario of the benchmark, where the eight application microservices were grouped in a single container.

3. The third was to consider the dependency between the microservices, where the microservices that had more dependency on each other were grouped into a single container. That is, in the case of the Sock Shop application, we grouped in the same container the Order, Payment, and Shipping microservices due to their high functional dependency and synchronous communication with each other.

4. The fourth criteria was considering the execution stack of each microservice, where the microservices that execute with the same stack were grouped in the same container. In this case, we grouped the Catalogue, User, and Payment microservices into a container with Go-based applications, the Cart, Order, Shipping, and Queue microservices into a container with Java-based applications, and the Front-End microservice into a container with Node.js-based applications.

Therefore, 4 execution scenarios were defined to be compared in each experiment: Benchmark, All-in-one, By-dependencies and By-stack.

In this study, we used the same three workloads employed in the previous work: 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.

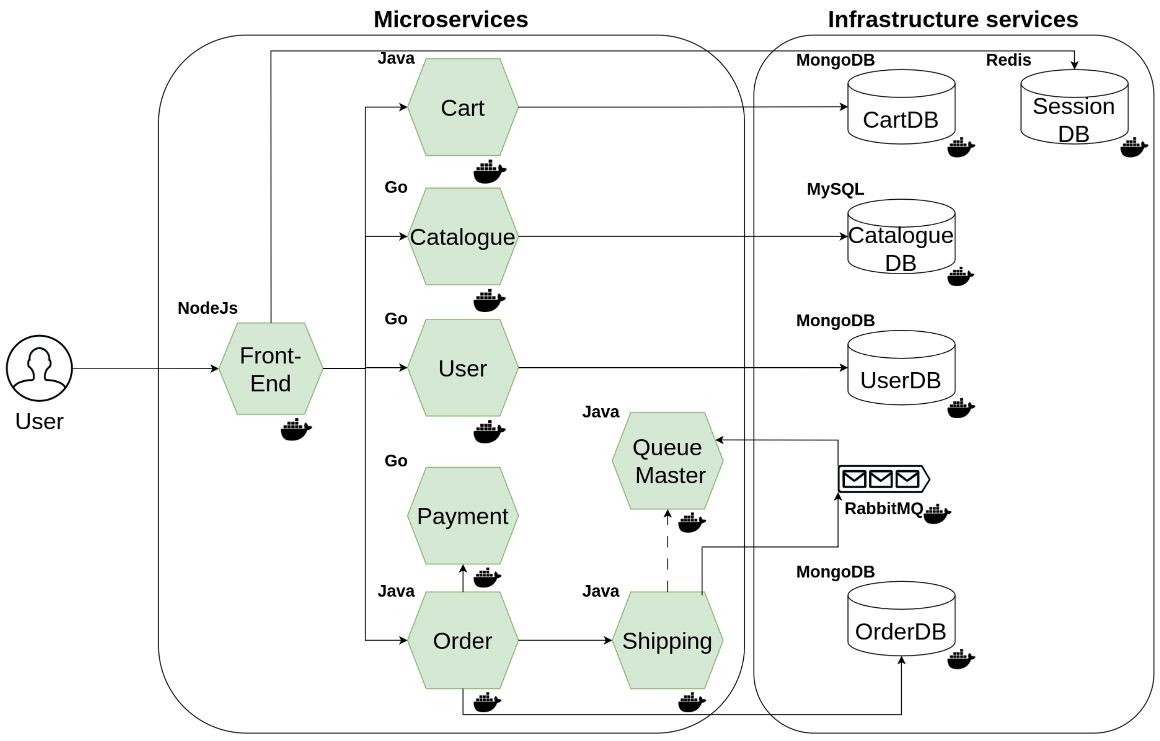


Figure 1: Sock Shop benchmark application scenario architecture

As the application family provided by the benchmark tool selected in the first experiment was 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. 2 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 [6]. Thus, for the maximum workload, the Amazon.com [7] workload was considered, and for the minimum workload, the Mercadolivre.com.br [8] workload was considered.

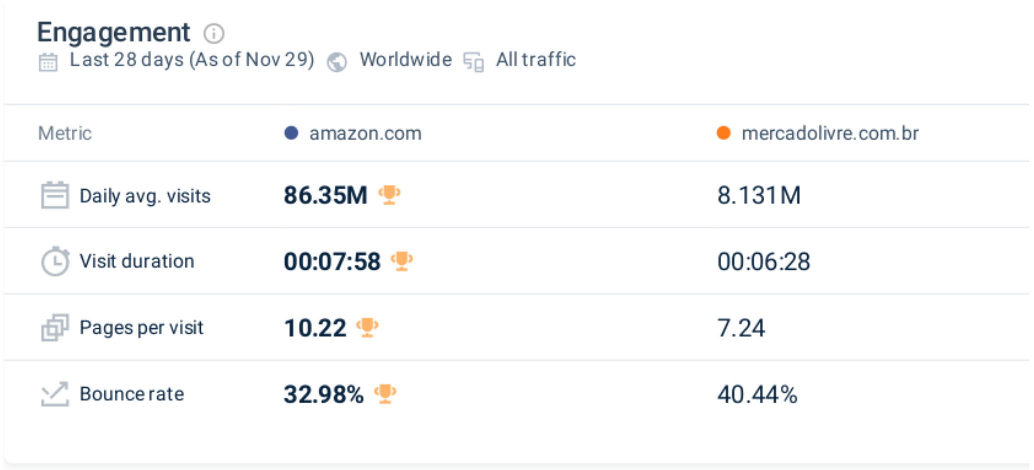
To calculate the number of users in each workload in the experiments, the number of users of each e-commerce was linearly distributed during the 24-hour period. Thus, for example, Amazon's workload presented an average amount of approximately 60.000 users accessing the platform every minute. From this 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, 4.500 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

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



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%

Figure 2: E-commerces workloads

calculated by the average of the low and high workloads. The workloads details it is presented in the Table 1.

Table 1: Table Experiments Workloads

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

4 Experiments

4.1 Infrastructure

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.

However, in this work, each node group in each scenario was configured with two nodes to allow horizontal pod scalability. This was necessary due to the maximum pod limit per instance, which for t3.xlarge instances is 58 pods. In the benchmark scenario, where the number of pods is higher, scaling all pods could exceed this limit, preventing the experiment from running.

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. 3 shows the experiments architecture design.

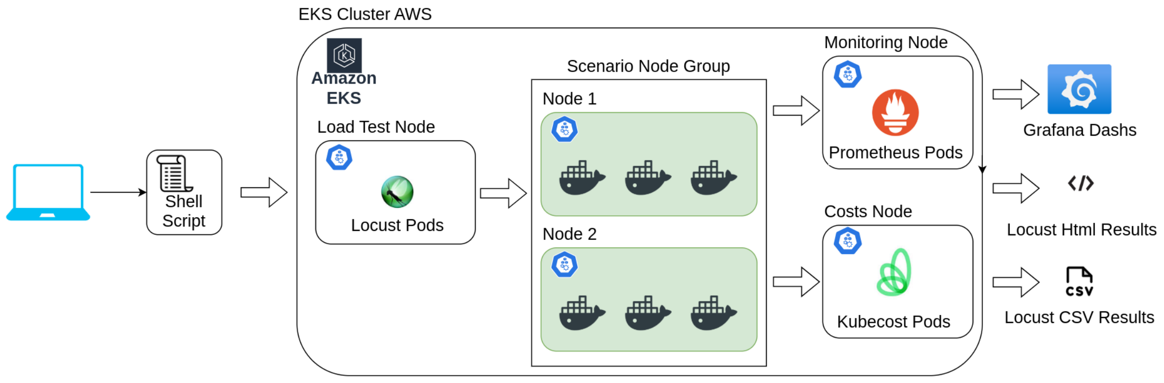


Figure 3: Experiments with HPA

4.2 HPA Configuration

After defining the number of nodes, we configured horizontal pod autoscaling (HPA) for each scenario. We set the minimum number of pods to 1 and the maximum to 4, considering the pod limit per instance.

Autoscaling targets for CPU and memory usage were configured at 150%, meaning that when CPU or memory usage reaches 150% of the requested resources, the pod scales up. In contrast, when usage drops below 150%, the pod scales down. The Kubernetes algorithm for upscaling and down scaling can be found in the official documentation [9].

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 four microservices grouping scenarios defined in the previous work, and each scenario ran during 10 minutes. Therefore, each workload took at least 40 minutes to run all scenarios.

Due to cost and execution time, six samples were executed 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.

5 Results

5.1 Performance results

In the previous study, our experiments without HPA showed that the use of microservice grouping strategies generally improved the application's *throughput* and *latency* across all evaluated workloads. The only performance-related metric negatively affected by this approach was the *error rate* under high workload conditions, where the *all-in-one* scenario exhibited the worst result compared to the other configurations.

However, in this study, with the use of HPA, we observed that the results related to *throughput* and *latency* metrics were further improved with HPA. Additionally, the *error rate* also benefited from this approach, effectively reducing it to near zero in the *all-in-one* scenario.

Table 2 presents a comparison between the median results obtained from the evaluated performance boxplots with and without HPA. For example, in the previous study, it was observed that under medium workload conditions, without HPA, the *all-in-one* and *by-stack* scenarios achieved a *throughput* 18% and 10% higher than the benchmark scenario, respectively.

With HPA, these same scenarios under the same workload showed a *throughput* increase of 66% and 33% compared to the benchmark scenario. Fig. 4 illustrates the variation in median requests per second (*requests/s*) for each grouping scenario.

[9]`https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/` (visited on Oct. 06, 2024)

We observed a similar pattern in the response time results as in the requests/s metric. In the previous work, it was observed that the application in scenarios with a higher number of microservices grouped together demonstrated better response times compared to scenarios with fewer microservices grouped. For example, in the all-in-one scenario, the application achieved a response time that was 20.5% faster than the benchmark scenario. In experiments with HPA, the improvement in response time was evident across all scenarios. In the medium workload, the all-in-one and by-stack scenarios achieved response times approximately 32% and 24% better, respectively, than the benchmark scenario.

Finally, when evaluating the median results of failures/s (error rate), we observed that HPA significantly improved this metric across all scenarios. However, the all-in-one scenario benefited the most, achieving an error rate close to zero. Fig. 4 presents the boxplot of the requests/s results evaluated and Table 2 provides a summary of the medians obtained for the performance metrics in experiments with HPA.

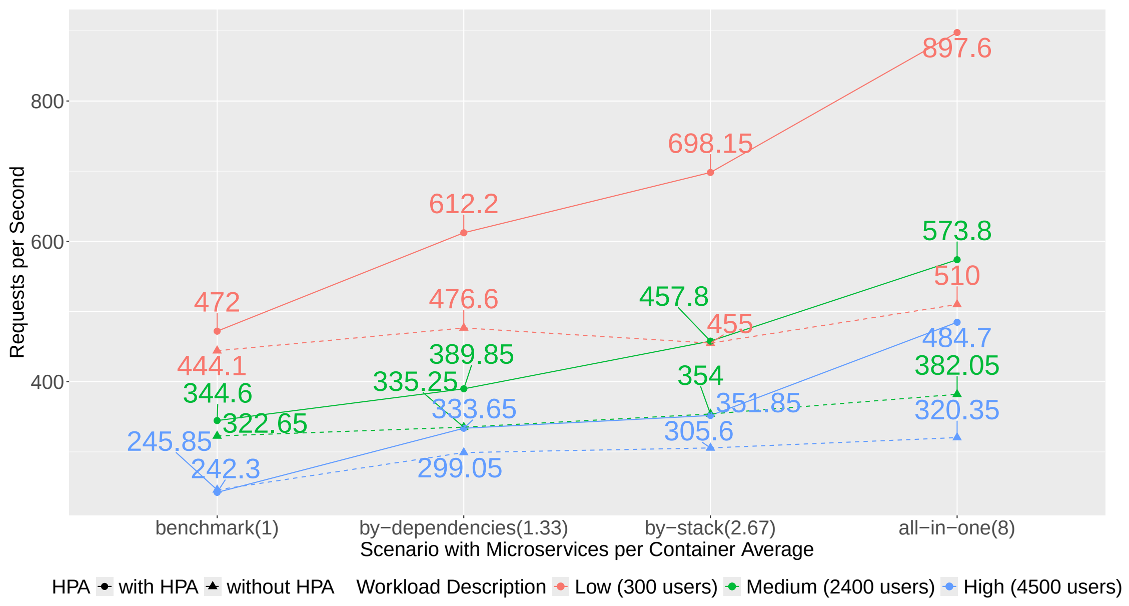


Figure 4: Throughput (requests/s) median results

5.2 Computational resources consumption results

When analyzing the median results related to computational resource consumption presented in the table 3, the experiments with HPA showed patterns similar to those observed in the previous work without HPA. An important point to highlight it is that all resource consumption results presented in this study correspond to the total sum of the resource usage of all pods that were scaled during the experiments. In other words, if the *all-in-one* scenario scaled two pods under a given workload, and each pod consumed **1MB** of memory, the total memory consumption in this case would be **2MB**.

Considering the CPU consumption metric, also in the experiments with HPA the CPU usage was more excessive in scenarios with a higher number of microservices grouped together. In the case of results with HPA, CPU consumption was up to 50% higher in the all-in-one scenario under high workload compared to the benchmark scenario. On the other hand, memory consumption was lower in scenarios with a higher number of microservices grouped. With HPA, memory usage in the all-in-one and by-stack scenarios was up to 27% and 20% lower, respectively, compared to the

Table 2: Performance metrics median results comparative

			Scenarios			
Workload	**Metrics**	**HPA**	**Benchmark**	**By-Dep.**	**By-Stack**	**All-in-one**
Low	Throughput (Requests/s)	Without HPA	444.10	476.60	455.00	510.00
		With HPA	472.00	612.20	698.15	897.60
	Latency (ms)	Without HPA	420.00	397.00	382.00	349.00
		With HPA	433.00	328.00	327.00	253.00
	Error Rate (Failures/s)	Without HPA	0.0	0.0	0.0	0.0
		With HPA	0.0	0.0	0.0	0.0
Medium	Throughput (Requests/s)	Without HPA	322.65	335,25	354.00	382.05
		With HPA	344.60	389.85	457.80	573.80
	Latency (ms)	Without HPA	3,602.00	3,103.00	2,919.00	2,754.00
		With HPA	3,476.00	2,871.00	2,654.00	2,364.00
	Error Rate (Failures/s)	Without HPA	0.0	0.0	0.0	0.0
		With HPA	0.10	0.10	0.10	0.10
High	Throughput (Requests/s)	Without HPA	245.85	299.05	305.60	320.35
		With HPA	242.30	333.65	351.85	484.70
	Latency (ms)	Without HPA	6,153	4,782	4,931	5,105
		With HPA	6,153.00	5,105.00	4,931.00	4,782.00
	Error Rate (Failures/s)	Without HPA	4.45	5.00	2.10	18.15
		With HPA	2.10	9.15	3.80	0.20

benchmark scenario under high workload. Fig. 5 presents the median results related to memory consumption in each microservices grouping scenario and workload with and without HPA.

Also, in this study, it was observed that there were no significant optimizations in the network and disk consumption metrics. However, unlike the previous work, it was noted that disk consumption increased with microservice grouping compared to the no-grouping strategy, indicating that grouping in scenarios with HPA leads to higher disk usage.

5.3 Financial resource consumption results

In addition to computational resource metrics, we also evaluated the financial resource consumption metric in a public cloud. Table 4 and Fig. 6 presents the median results about the monthly costs for each microservices grouping scenario, workload and experiment (without HPA and with HPA).

In the experiments without HPA, the differences in costs between the medians of each scenario were not particularly significant. Only in the high workload did we observe a notable difference, with the all-in-one and by-stack scenarios showing much higher median costs compared to the benchmark and by-dependencies scenarios. This difference was primarily influenced by the availability metric (which we will analyze in another section), as the scenarios with higher costs were those that remained available for a longer duration during the experiment.

In the experiments with HPA, the grouping scenarios exhibited similar availability, and thus costs were less affected by this metric. But, the costs was directly influenced by the scalability metric in these experiments, that is, the scenarios with higher costs were those with a higher scalability rate, as they required scaling more pods. For instance, under high workload conditions, the scenarios with the lowest costs were benchmark and all-in-one, which were also the ones that scaled the least. On the other hand, the scenarios that scaled the most (by-stack and by-dependencies) had the highest costs. Fig. 7 presents the example of the cost variation by the scalability variation.

5.4 Availability results

Another important result that was analyzed in this study was about the scenarios availability. As in the previous work, we evaluated the availability of the scenarios from two perspectives: The availability by error and the availability by time.

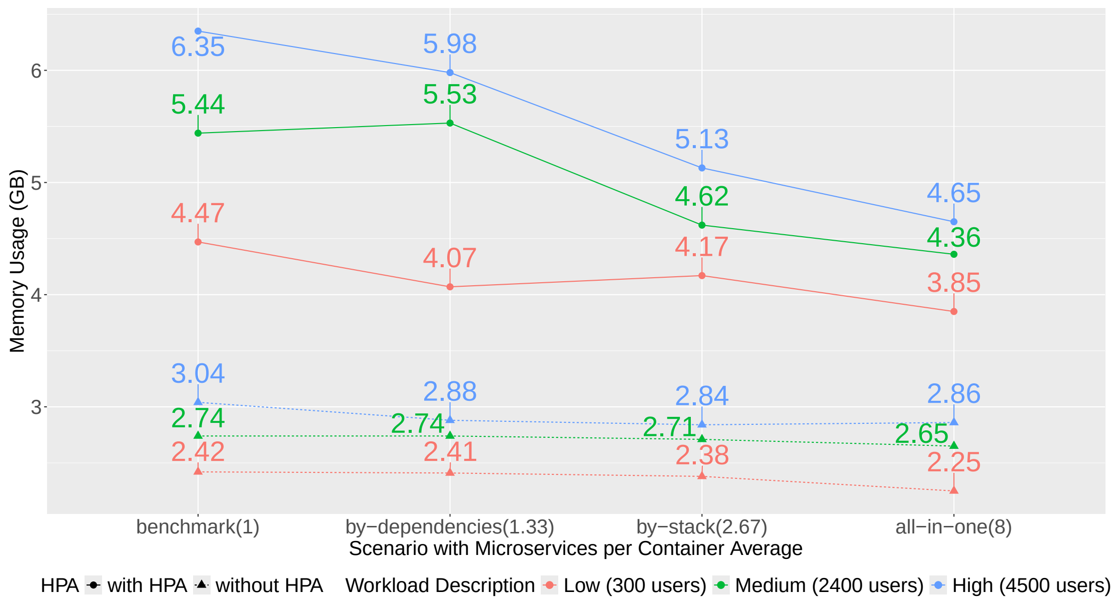


Figure 5: Memory consumption median results

Table 3: Computational Resources consumption median results

			Scenarios			
Workload	**Metrics**	**HPA**	**Benchmark**	**By-Dep.**	**By-Stack**	**All-in-one**
Low	CPU (cores)	Without HPA	2.97	3.02	2.92	3.14
		With HPA	3.71	4.74	5.04	5.74
	Memory (GB)	Without HPA	2.42	2.41	2.38	2.25
		With HPA	4.47	4.07	4.17	3.85
	Network (MB/s)	Without HPA	30.16	30.20	29.16	26.26
		With HPA	36.79	45.32	48.85	51.75
	Disk (MB)	Without HPA	164	142	133	99
		With HPA	178	201	177	186
Medium	CPU (cores)	Without HPA	3.25	3.30	3.41	3.35
		With HPA	4.47	4.64	4.98	5.38
	Memory (GB)	Without HPA	2.74	2.74	2.71	2.65
		With HPA	5.44	5.53	4.62	4.36
	Network (MB/s)	Without HPA	21.70	21.67	22.85	20.39
		With HPA	26.86	27.57	35.03	33.19
	Disk (MB)	Without HPA	180	183	140	114
		With HPA	193	216	204	201
High	CPU (cores)	Without HPA	2.41	2.46	2.82	3.61
		With HPA	3.56	4.81	4.91	5.37
	Memory (GB)	Without HPA	3.04	2.88	2.84	2.86
		With HPA	6.35	5.98	5.13	4.65
	Network (MB/s)	Without HPA	13.76	12.15	17.76	17.15
		With HPA	18.76	24.11	27.48	28.70
	Disk (MB)	Without HPA	100	121	122	124
		With HPA	170	214	191	203

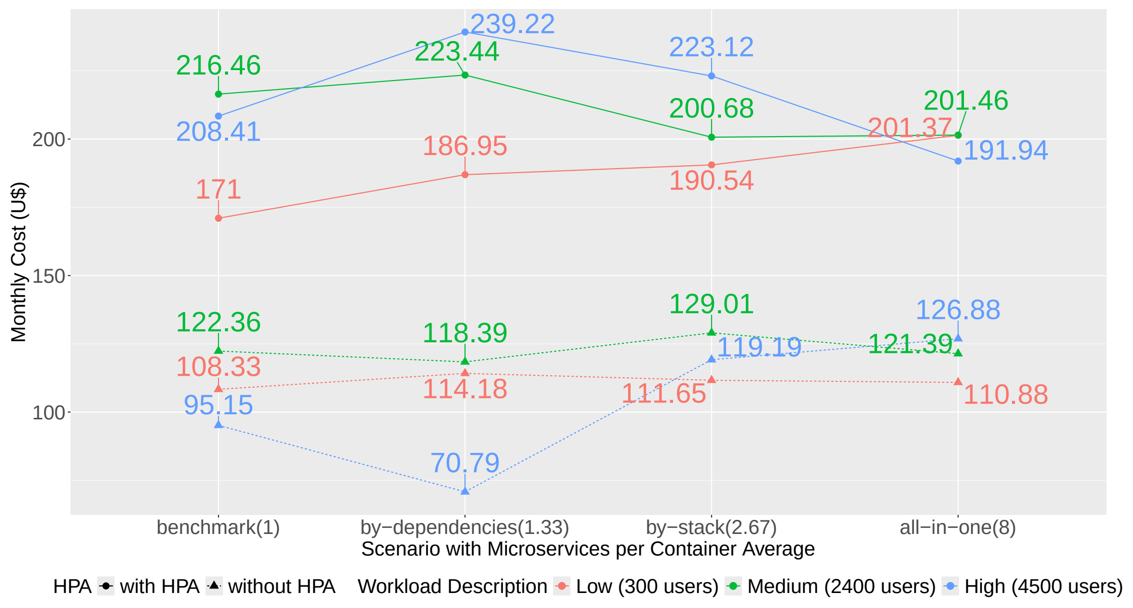


Figure 6: Costs in public cloud median results

Table 4: Estimated monthly costs median results

			Scenarios			
Workload	**Metrics**	**HPA**	**Benchmark**	**By-Dep.**	**By-Stack**	**All-in-one**
Low	Monthly Cost (U$)	Without HPA	108.34	114.18	111.65	110.88
		With HPA	171.01	186.95	190.54	201.37
Medium	Monthly Cost (U$)	Without HPA	122.36	118.39	129.02	121.40
		With HPA	216.46	223.45	200.68	201.46
High	Monthly Cost (U$)	Without HPA	95.15	70.79	119.19	126.88
		With HPA	208.41	239.22	223.12	191.94

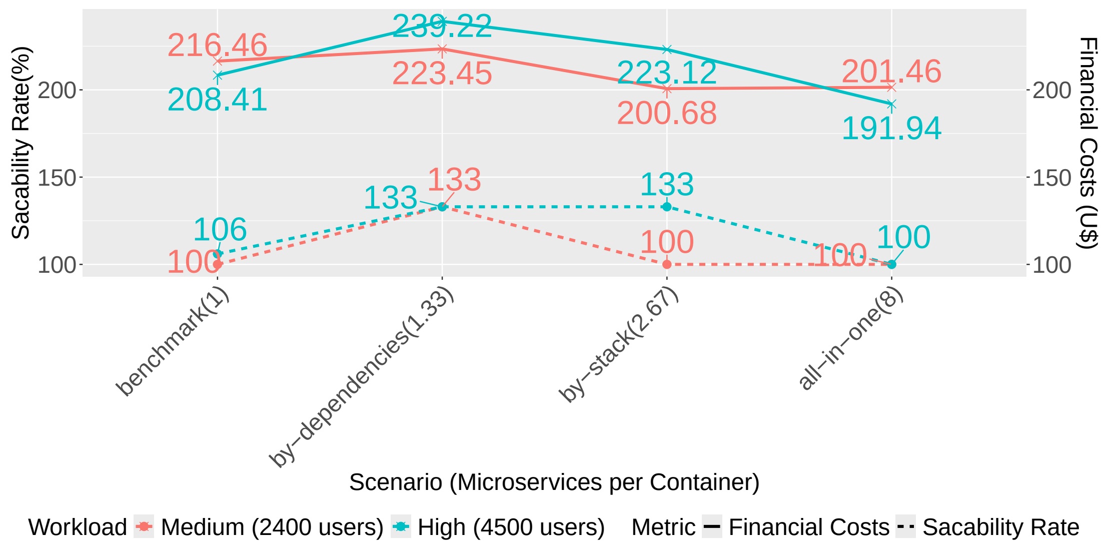


Figure 7: Estimated monthly costs median results and scalability rate median results

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.

5.4.1 Availability by error

When observing the median results of the *availability by error* metric in Table 5, it is possible to verify that this metric showed significant results only under high workload conditions. In low and medium workloads, this metric remained close to 100% across all scenarios.

The analysis of this metric's results revealed that, in general, the scenarios with higher availability by error were those with a greater number of microservices grouped within the same container. This trend was observed both in experiments without HPA and in those with HPA. Furthermore, the results related to this metric showed a significant improvement when HPA was enabled.

This improvement was primarily due to *availability by time*, which directly impacted this metric. In the HPA experiments, none of the grouped microservices scenarios experienced pod downtime.

5.4.2 Availability by time

Observing the results related to *availability by time*, it is possible to verify that under high workload conditions without HPA, all scenarios experienced a period of downtime. Among them, the most affected scenarios were *benchmark* and *by-dependencies*.

However, the use of HPA mitigated this issue for all scenarios with microservice grouping. The only scenario that still experienced downtime with this strategy was the *benchmark* scenario.

Figure 8 presents the median results obtained for this metric in both experiments, with and without HPA.

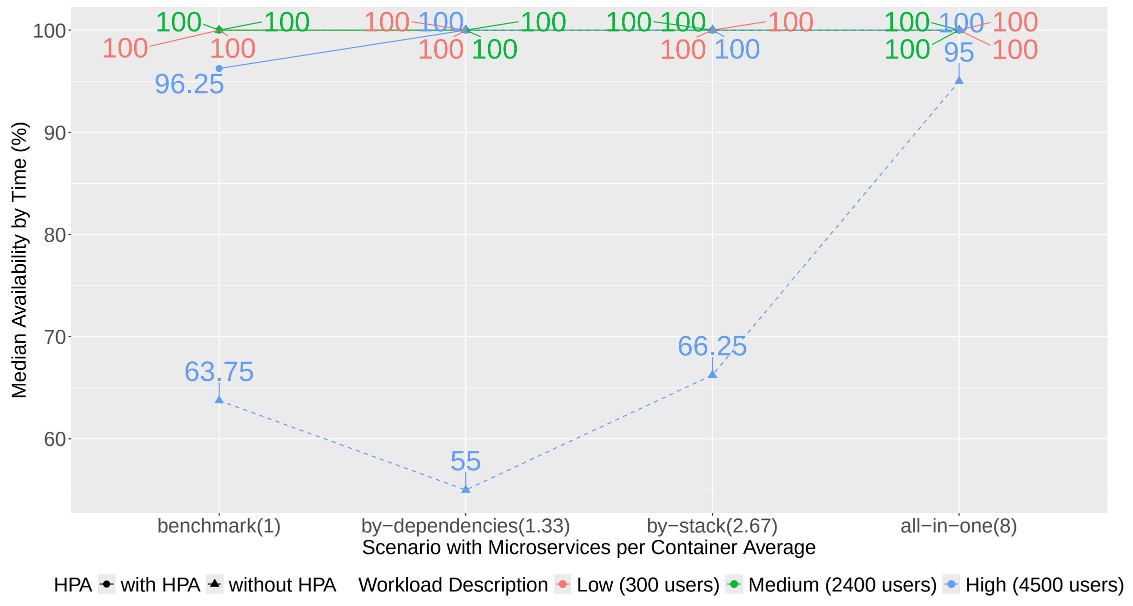


Figure 8: Availability by time median results

5.5 Scalability results

Regarding the horizontal scalability of each scenario, we evaluated only the results from the experiments with HPA, as no horizontal scaling of pods occurred in the experiments without HPA. To

Table 5: Availability median results

			Scenarios			
Workload	**Metrics**	**HPA**	**Benchmark**	**By-Dep.**	**By-Stack**	**All-in-one**
Low	Availability by Error	Without HPA	100.00	100.00	100.00	100.00
		With HPA	100.00	100.00	100.00	100.00
	Availaiblity by Time	Without HPA	100.00	100.00	100.00	100.00
		With HPA	100.00	100.00	100.00	100.00
Medium	Availability by Error	Without HPA	99.99	99.99	99.98	99.99
		With HPA	99.42	99.97	99.97	99.97
	Availaiblity by Time	Without HPA	100.00	100.00	100.00	100.00
		With HPA	100.00	100.00	100.00	100.00
High	Availability by Error	Without HPA	81.35	80.30	84.30	89.38
		With HPA	91.78	94.29	94.51	99.48
	Availaiblity by Time	Without HPA	63.75	55.00	66.25	95.00
		With HPA	96.25	100.00	100.00	100.00

assess scalability, we used the scalability rate percentage as the metric, calculated using the following formula:

$$ScalabilityRate = \frac{(NumberofScaledPods - InitialNumberofPods)}{InitialNumberofPods} \quad (1)$$

This formula allows us to determine the scalability percentage relative to the initial number of pods in each scenario. For example, in the benchmark scenario, the initial number of pods is 8, with one pod per microservice. Therefore, if the scalability rate in this experiment is 50%, it means that 4 pods were scaled (0.5 * 8).

In the All-in-One scenario, which starts with a single pod containing all microservices, a 50% scalability rate would indicate that 0.5 pods were scaled (0.5 * 1). However, since Kubernetes does not support fractional pod scaling, the minimum scalability rate in the All-in-One scenario is 100%, scaling at least one pod (1 * 1). It is important to note that the lower this metric, the more efficient the scenario is for processing a given workload. In other words, a lower scalability rate indicates that fewer pods are required to handle the same number of users across the evaluated scenarios.

Therefore, observing the consolidated scalability rates and comparing the median results of each scenario, we found that in the lowest workload, the benchmark scenario had the lowest scalability rate, scaling only 75% of the initial number of pods. In contrast, the by-stack scenario had the highest scalability rate, scaling 150% of the initial number of pods. In the All-in-One and by-dependencies scenarios, the scalability rate was 100%, effectively doubling the initial number of pods.

In the medium workload, the scalability rates were quite similar across all scenarios, with the by-dependencies scenario standing out with a scalability rate of 133%, while the other scenarios maintained a scalability rate of 100%.

In the high workload, the All-in-One and benchmark scenarios had the lowest scalability rates at 100% and 106%, respectively. Meanwhile, both the by-stack and by-dependencies scenarios achieved scalability rates of 133%.

It is also worth examining the scalability over time. When analyzing the results, we observed that across all workloads, during the first 300 seconds (as users were entering the workload), the All-in-One scenario scaled the fastest, reaching a 100% scalability rate within the first 60 seconds.

The by-stack scenario was the second fastest, reaching 100% scalability in 180 seconds in the low workload, 120 seconds in the medium workload, and 60 seconds in the high workload.

The by-dependencies scenario followed, reaching 100% scalability in 300 seconds in the low workload, 180 seconds in the medium workload, and 116% scalability in 180 seconds in the high workload. The benchmark scenario was the slowest to scale within the first 300 seconds, achieving 75% scalability in the low workload, 106% in the medium workload, and 112% in the high workload, all within 300 seconds.

After the first 300 seconds, when all users were in the system, we observed that in the low workload, the by-stack scenario maintained the highest scalability rate of 167% until the end of the experiment. The All-in-One and benchmark scenarios had the lowest scalability rates, maintaining 100% and 93.75%, respectively, throughout the experiment.

In the medium workload, the results differed slightly. The by-dependencies scenario scaled the most, reaching a rate of 150% at 480 seconds. The benchmark scenario followed, reaching 125% at 540 seconds. The by-stack scenario maintained a rate of 116% from 300 to 480 seconds, then dropped to 100% in the final moments. The All-in-One scenario remained stable at 100% from the first 60 seconds until the end of the experiment.

In the high workload, the All-in-One and benchmark scenarios scaled the least during this period, while the by-stack scenario scaled the most, reaching a rate of 166% at 480 seconds of execution. Figure 9 and table 6 present a comparison of the median scalability rates over time.

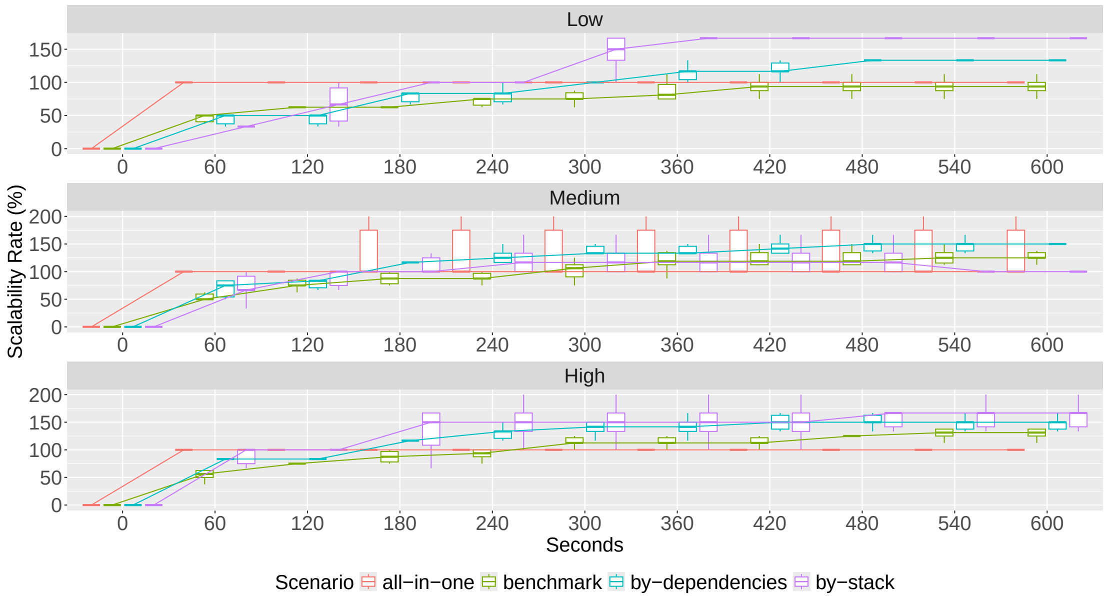


Figure 9: Median scalability rate (%) over time

Table 6: Median scalability rate (%) over time

		Experiment execution time (seconds)										
Workloads	**Scenarios**	**0**	**60**	**120**	**180**	**240**	**300**	**360**	**420**	**480**	**540**	**600**
Low	all-in-one	0	100	100	100	100	100	100	100	100	100	100
	benchmark	0	50	62.50	62.50	75	75	81.25	93.75	93.75	93.75	93.75
	by-dep.	0	50	50	83.33	83.33	100	116.67	116.67	133.33	133.33	133.33
	by-stack	0	33.33	66.67	100	100	150	166.67	166.67	166.67	166.67	166.67
Medium	all-in-one	0	100	100	100	100	100	100	100	100	100	100
	benchmark	0	50	75	87.50	87.50	106.25	118.75	118.75	118.75	125	125
	by-dep.	0	75	83.33	116.67	125	133.33	133.33	141.67	150	150	150
	by-stack	0	66.67	100	100	116.67	116.67	116.67	116.67	116.67	100	100
High	all-in-one	0	100	100	100	100	100	100	100	100	100	100
	benchmark	0	56.25	75	87.50	93.75	112.50	112.50	112.50	125	131.25	131.25
	by-dep.	0	83.33	83.33	116.67	133.33	141.67	141.67	150	150	150	150
	by-stack	0	100	100	150	150	150	150	150	166.67	166.67	166.67

5.6 Correlation between the results

Based on the results obtained from the different microservices grouping scenarios, we identified important correlations between **performance**, **resource consumption**, and **scalability**. By relating these metrics, we gained a broader understanding of how microservices grouping impacts the overall behavior of the application in terms of efficiency and cost-effectiveness. To identify correlations, we used a strategy comparing the medians obtained in each result and analyzing them using Pearson correlation coefficient as the statistical method. We observed correlation values close to 1 for each comparison, indicating a strong correlation between the analyzed data.

5.6.1 Correlation between throughput and CPU consumption

We observed that the throughput (requests/s) varied significantly across the different microservices grouping scenarios. A similar pattern was found in CPU consumption for each grouping scenario. By analyzing the medians of the obtained results, we identified a correlation between these metrics. As shown in table 7, higher CPU consumption in each scenario corresponds to an increase in the number of requests/s.

Figure 10 provides an example of this variation in the medium workload during experiments with HPA, although this behavior was also observed across other workloads and in experiments without HPA. Additionally, we found that scenarios with a greater number of microservices grouped exhibited higher CPU consumption and throughput. This is because the reduction in inter-pod communication enhances local processing, which increases CPU demand and results in higher throughput.

Workload	Scenario	Microservices per Container (Average)	Requests/s with HPA	CPU Usage (cores) With HPA	Pearson Correlation Coefficient
Low (300 users)	benchmark	1.00	472	3.71	0.93
Low (300 users)	by-dependencies	1.33	612.2	4.74	
Low (300 users)	by-stack	2.67	698.15	5.04	
Low (300 users)	all-in-one	8.00	897.6	5.74	
Medium (2400 users)	benchmark	1.00	344.6	4.47	0.67
Medium (2400 users)	by-dependencies	1.33	389.85	4.64	
Medium (2400 users)	by-stack	2.67	457.8	4.98	
Medium (2400 users)	all-in-one	8.00	573.8	5.38	
High (4500 users)	benchmark	1.00	242.3	3.56	0.71
High (4500 users)	by-dependencies	1.33	333.65	4.81	
High (4500 users)	by-stack	2.67	351.85	4.91	
High (4500 users)	all-in-one	8.00	484.7	5.37	

Table 7: Correlation between requests/s and CPU usage (median results)

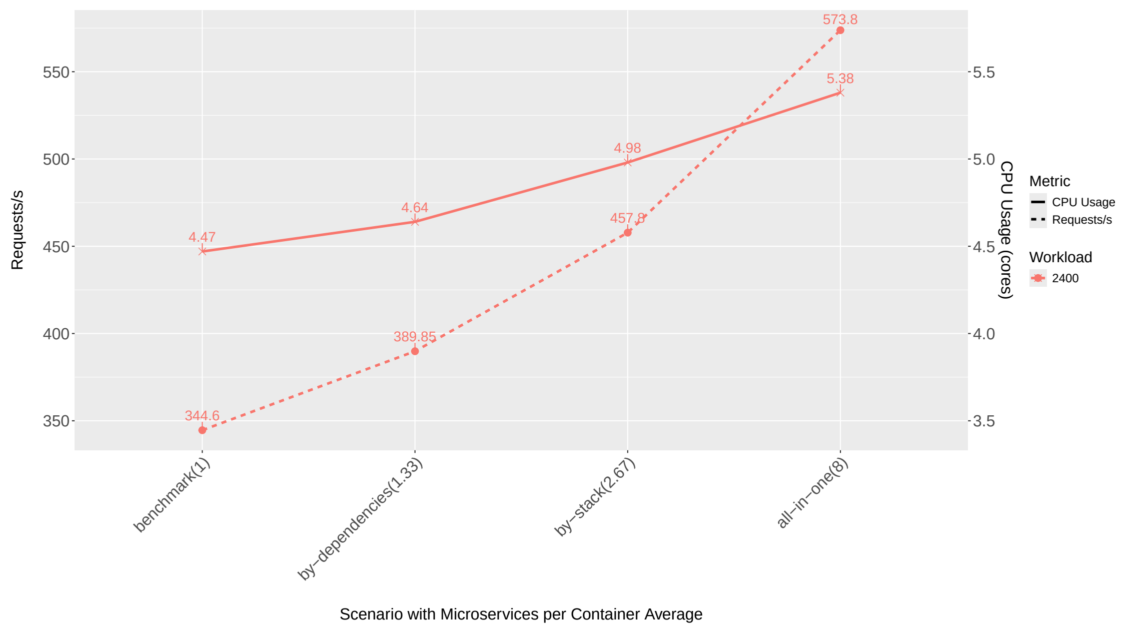


Figure 10: Correlation between Requests/s and Cpu Usage (core) in HPA experiment and medium workload

5.6.2 Correlation between memory consumption and latency reduction

We also observed a correlation between memory consumption and response time. In general, the scenarios with lower response times also exhibited lower memory consumption. These scenarios were the ones with a higher average of microservices grouped. This correlation between lower memory consumption and faster response times can be explained by several technical factors:

- **Reduction of Communication Overhead**: When microservices are in the same container, they can communicate directly through shared memory or inter-process communication (IPC), eliminating the need for network calls. This reduces latency overhead and avoids the need for additional memory buffers that would be used in external communications;
- **Optimization of Shared Resource Usage**: Grouping microservices in the same container allows them to share libraries and runtime environments, reducing the overhead of loading multiple instances of identical resources. This contributes to an overall reduction in memory consumption, as common dependencies are loaded only once;
- **Improved Cache Utilization**: Grouping microservices can increase the efficiency of in-memory cache usage, especially when microservices share common data or processing. With fewer costly transactions between different containers, internal caches can be used more effectively, contributing to lower latency.

These results can be seen in table 8, and an example of this correlation can be observed in Figure 11.

Workload	Scenario	Microservices per Container (Average)	Response time without HPA	Memory (Gb) without HPA	Pearson Correlation Coefficient
Low (300 users)	benchmark	1,00	420	2,42	0.93
Low (300 users)	by-dependencies	1,33	397	2,41	
Low (300 users)	by-stack	2,67	382	2,38	
Low (300 users)	all-in-one	8,00	349	2,25	
Medium (2400 users)	benchmark	1,00	3602	2,74	0.76
Medium (2400 users)	by-dependencies	1,33	3103	2,74	
Medium (2400 users)	by-stack	2,67	2919	2,71	
Medium (2400 users)	all-in-one	8,00	2754	2,65	
High (4500 users)	benchmark	1,00	6594	3,04	0.78
High (4500 users)	by-dependencies	1,33	6197	2,88	
High (4500 users)	by-stack	2,67	5838	2,84	
High (4500 users)	all-in-one	8,00	5314	2,86	

Table 8: Correlation response Time (ms) and memory usage (median results)

5.6.3 Correlation between scalability and financial cost

The relationship between scalability and financial cost is crucial when evaluating the performance and efficiency of microservices in public cloud environments. Table 9 and the Figure 7 show that as the system scales automatically to meet increased demand, especially in the HPA experiments, computational resource costs directly increase.

In our experiments with HPA, we observed significant variations in pod scalability across the different grouping scenarios. The All-in-One scenario, for example, demonstrated greater efficiency in scalability, requiring fewer pods to scale compared to other scenarios. This efficiency was reflected in lower financial costs, particularly in the medium and high workloads.

Workload	Scenario	Microservices per Container (Average)	Scalability rate (%)	Monthly costs (U$)	Pearson Correlation Coefficient
Low (300 users)	benchmark	1,00	75	171,01	0.48
Low (300 users)	by-dependencies	1,33	100	186,95	
Low (300 users)	by-stack	2,67	150	190,54	
Low (300 users)	all-in-one	8,00	100	201,37	
Medium (2400 users)	benchmark	1,00	100	216,46	0.77
Medium (2400 users)	by-dependencies	1,33	133,33	223,45	
Medium (2400 users)	by-stack	2,67	100	200,68	
Medium (2400 users)	all-in-one	8,00	100	201,46	
High (4500 users)	benchmark	1,00	106,25	208,41	0.93
High (4500 users)	by-dependencies	1,33	133,33	239,22	
High (4500 users)	by-stack	2,67	133,33	223,12	
High (4500 users)	all-in-one	8,00	100	191,94	

Table 9: Correlation scalability rate(%) and estimated monthly costs (U$)

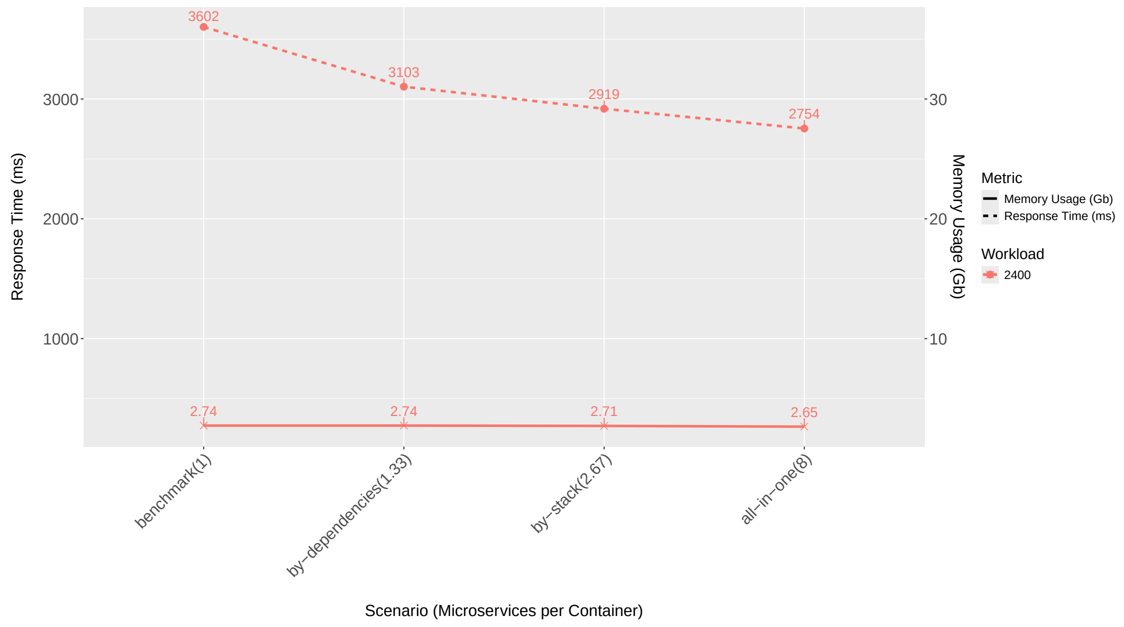


Figure 11: Correlation between response time (ms) and memory usage (Gb) in without HPA experiment and medium workload

Low workloads: In lower intensity workloads (300 users), our results showed that the benchmark scenario, with fewer microservices grouped per container, scaled more gradually, reaching a scalability rate of 75%. In comparison, the by-stack scenario scaled to 150%, and the All-in-One scenario scaled to 100%. These values indicate that, although the by-stack scenario scaled more rapidly, the All-in-One scenario was more efficient in maintaining its resources, scaling in a more controlled manner. From a financial perspective, the estimated monthly cost for the All-in-One scenario was 10% higher than the benchmark scenario in the low workload without HPA. However, with HPA enabled, the All-in-One scenario managed its scalability more efficiently, resulting in costs close to the benchmark. In contrast, the by-stack and by-dependencies scenarios incurred higher costs due to the need to scale more pods.

Medium workloads: In medium workloads (2,400 users), the All-in-One scenario demonstrated a notable balance between scalability and cost. Scalability in this scenario was limited to 100%, compared to 133% in the by-dependencies scenario and 116% in the by-stack scenario. This lower scalability was reflected in a reduced monthly cost for the All-in-One scenario, approximately 7% lower than the benchmark with HPA enabled. These results suggest that grouping microservices can indeed reduce costs by limiting the need to scale large numbers of pods. The reduction in financial cost observed in the All-in-One scenario can be explained by the more efficient local communication between microservices, which reduces network overhead and the number of transactions between containers, thereby decreasing the need to scale additional pods to handle the workload.

High workloads : In high workloads (4,500 users), the All-in-One scenario maintained a scalability rate of 100%, while the by-stack and by-dependencies scenarios scaled to 133% each. This significant difference in scalability had a direct impact on financial costs. The All-in-One scenario resulted in costs approximately 8% lower than the by-dependencies scenario and 15% lower than the by-stack scenario with HPA enabled. These results suggest that the All-in-One scenario, despite requiring more CPU due to the higher concentration of microservices, compensates in financial costs by scaling more efficiently, maintaining a smaller number of active pods. In contrast, scenarios with fewer microservices per container needed to scale more pods to handle the same load, leading to increased costs.

6 Discussion

In the previous study, without using HPA, it was observed that microservice grouping in certain scenarios could benefit the application, particularly in terms of performance and optimizations in resource consumption, such as memory and disk.

Moreover, this approach improved availability in scenarios with a greater number of microservices grouped within a single container. However, this strategy could be operationally more costly than the one-container-per-microservice approach and did not lead to significant optimizations in financial costs.

In this study, with the use of HPA, the same benefits observed in the previous study were confirmed, with performance and memory consumption optimizations being further enhanced by HPA. However, with HPA, resource consumption in some areas was adversely affected, such as network and disk usage, which increased. This indicates that microservice grouping in HPA scenarios can lead to higher disk and network consumption.

Regarding financial costs, the previous study could not determine the benefits of grouping microservices as this metric was directly impacted by the availability of each scenario. In the first study, under higher workloads, the grouped scenarios showed higher financial costs.

However, in the experiments with HPA, where scenario availability remained similar under higher workloads, it was observed that scenarios with more microservices grouped generally presented lower financial costs. This is mainly due to scenarios with more microservices grouped requiring fewer pods to scale and process the workloads.

As for availability, a significant improvement in this metric was observed across all scenarios with the use of HPA, especially in the scenarios with microservice grouping, which showed no downtime. Thus, adopting the microservice grouping strategy with HPA proved to be beneficial for the application.

Lastly, in this study, scalability in each scenario and workload was also evaluated. To assess this metric, the scalability rate was considered relative to the initial number of pods in each scenario.

It was found that, in general, scenarios with microservice grouping showed a higher scalability rate. However, this does not mean these scenarios had a higher absolute number of pods. For example, while the benchmark scenario had a lower scalability rate than the by-dependencies scenario in some workloads, the by-dependencies scenario concluded the experiment with a lower absolute number of pods.

It is also important to highlight that the study revealed how the results of certain metrics are intrinsically correlated and influenced by both the workload and the grouping strategy adopted.

The results showed a clear correlation between CPU consumption and application throughput across all grouping scenarios and workload levels. For example, scenarios with a higher number of microservices grouped per container — such as the all-in-one scenario — not only achieved a throughput 66% higher than the benchmark scenario but also showed increased CPU usage.

This correlation was quantified by Pearson coefficients close to 1 in most cases (see Table 7), indicating that increasing grouping density intensifies local computation and reduces network overhead, at the cost of increased CPU load.

Conversely, we observed in the Table 8 that groupings with higher microservice density systematically led to lower memory consumption and improved latency. The underlying mechanisms include reduced inter-container communication overhead, improved cache utilization, and elimination of redundant runtime environment. The strong correlation, as evidenced by coefficients ranging from 0.76 to 0.93, suggests that grouping microservices can be particularly beneficial for latency-sensitive and memory-constrained applications.

Additionally, our analysis revealed a robust correlation between scalability rate (the increase in the number of pods relative to the initial state) and financial costs in public cloud. In HPA-enabled experiments, scenarios that required fewer scaled pods consistently resulted in a lower estimated monthly cost, particularly under medium and high workloads 9.

7 Practical Insights on Microservice Grouping

The results of this study reveal that microservice grouping strategies can play a central role in performance tuning and cost efficiency within autoscalable cloud environments. Building on our experimental findings, this section outlines practical insights to guide architects and engineers in making informed deployment decisions based on trade-offs between performance, resource usage, and operational maintainability.

7.1 Choosing the Right Grouping Strategy

Each grouping configuration studied exhibited specific advantages depending on the workload level and the metric being prioritized:

- **Benchmark (1 service/container)** offered greater operational flexibility, easier debugging, and clearer service isolation. However, it incurred higher communication overhead and lower throughput, especially under medium and high workloads.
- **All-in-One** provided the best throughput and lowest latency, particularly under autoscaling, due to local service communication and shared dependencies. Nonetheless, it presented higher CPU and disk usage, limited service-level scalability, and increased complexity for updates and testing.
- **By-Dependencies** struck a balance between performance and resource optimization. It grouped services with high interdependence, reducing latency and memory consumption. However, it required precise knowledge of service relationships and suffered from limited scaling granularity.
- **By-Stack** showed consistent performance and maintainability benefits by grouping services by technology stack. Although it was less aggressive in resource savings, it simplified deployments and presented good latency and autoscaling behavior.

To facilitate the decision-making process, we synthesized the main trade-offs into a decision tree shown in Figure 12, which guides the selection of grouping strategies based on workload intensity, service coupling, and operational priorities.

7.2 Guidelines for Deployment Design

Our analysis supports the following recommendations:

1. **Correlate grouping with workload profiles:** In high-load environments, grouping services (especially all-in-one or by-dependencies) can significantly boost performance metrics. However, for volatile or heterogeneous loads, more modular grouping may be preferable.
2. **Use HPA to offset grouping limitations:** Autoscaling mechanisms effectively mitigate resource contention issues introduced by tight service coupling. Scenarios with grouped services, when combined with HPA, showed notable improvements in availability and error reduction.
3. **Evaluate inter-service communication costs:** Reducing network latency through co-location of services can enhance CPU efficiency, but this comes at the cost of reduced scaling independence. Trade-offs must be assessed per application domain.
4. **Avoid over-grouping in frequently updated components:** Services requiring frequent releases or hotfixes should remain isolated or grouped minimally to prevent downtime propagation.
5. **Consider grouping inside Kubernetes Pods:** If container-level grouping introduces too much rigidity, Kubernetes Pods offer a middle ground—allowing shared resources and fast communication without full service fusion.

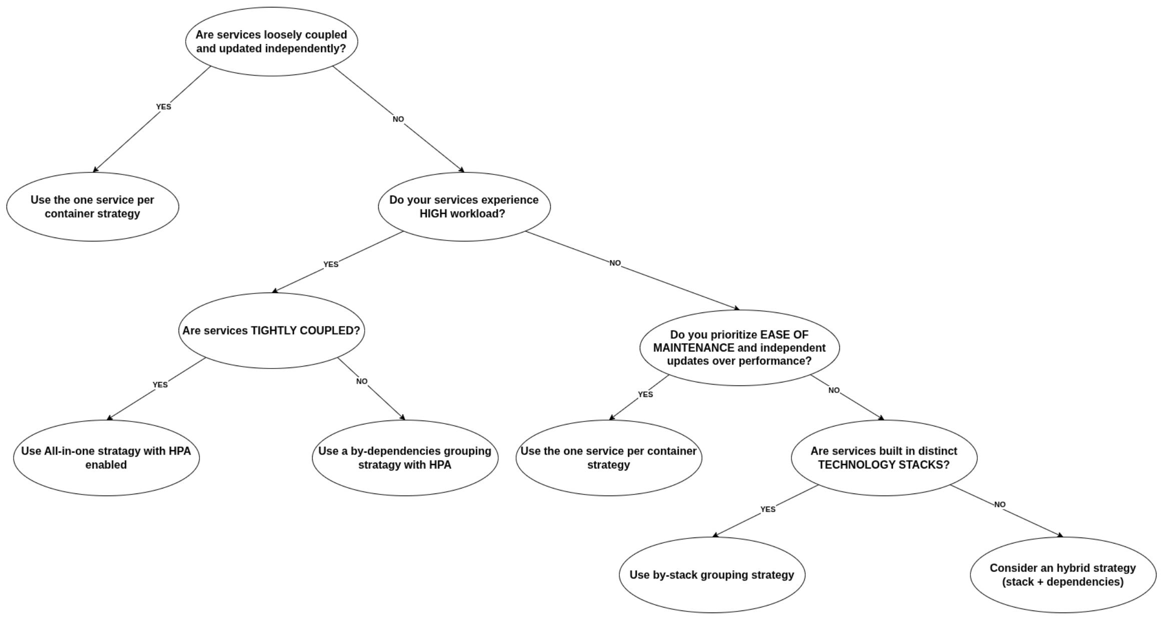

Are services loosely coupled and updated independently?
YES
NO
Use the one service per container strategy
Do your services experience HIGH workload?
YES
NO
Are services TIGHTLY COUPLED?
Do you prioritize EASE OF MAINTENANCE and independent updates over performance?
YES
NO
YES
NO
Use All-in-one stratagy with HPA enabled
Use a by-dependencies grouping stratagy with HPA
Use the one service per container strategy
Are services built in distinct TECHNOLOGY STACKS?
YES
NO
Use by-stack grouping strategy
Consider an hybrid strategy (stack + dependencies)

Figure 12: Decision tree for selecting microservice grouping strategies.

6. **Test and iterate based on real workload traces:** Simulated workloads alone may not expose the operational impact of grouping strategies. Real traffic traces and long-term observation help to refine grouping configurations.

7.3 Strategic Takeaways

Overall, our experiments reinforce that grouping microservices is not a binary decision but a strategic design parameter. The optimal configuration depends on workload stability, autoscaling readiness, and service evolution needs.

While grouped strategies, especially All-in-One, can unlock substantial performance and cost benefits when paired with autoscaling, their application should be constrained to scenarios with tightly coupled logic and relatively stable service boundaries. Hybrid approaches—such as combining grouping by stack with modular boundary enforcement—offer a viable middle path, balancing gains and maintainability.

In sum, microservice grouping should be treated as an architectural lever, to be tuned dynamically, guided by empirical performance data, system observability, and business objectives.

8 Conclusion and Future Work

This study presented a comprehensive analysis of integrating Horizontal Pod Autoscaling (HPA) into containerized microservices architectures, focusing on the impact of microservices grouping strategies on performance metrics, computational resource consumption, and financial costs.

The results demonstrated that HPA provides significant improvements in scalability, resource efficiency, and availability, especially in high-load scenarios. However, increased disk usage and operational complexity were observed in scenarios with greater microservices grouping.

Among the main insights obtained, it was highlighted that scenarios with higher microservices grouping tend to show better response times and higher throughput, but the effectiveness of this strategy is closely tied to workload characteristics and the level of granularity adopted.

Additionally, the integration of HPA proved particularly advantageous in reducing failures and increasing application availability, even under high-demand conditions.

Also, the results demonstrate that microservices grouping can offer significant benefits in terms of performance, resource efficiency, and cost-effectiveness. However, these gains are not unlimited and depend on several factors, such as workload type, infrastructure management capabilities, and the appropriate use of scalability mechanisms like Horizontal Pod Autoscaling (HPA).

The benefits of grouping microservices into the same container depend on careful implementation and a balance between performance and flexibility. Grouping is most advantageous in high-load scenarios and environments that can take advantage of efficient internal communication and resource optimization.

However, there are limits to the effectiveness of this strategy. Excessive grouping can lead to resource contention, maintenance difficulties, and challenges in managing updates and releases. The ideal approach involves balancing grouping and scalability, prioritizing the grouping of services that benefit from local communication, while other services with specific update and scalability needs are kept independent.

The use of HPA is essential to ensure that the infrastructure responds dynamically to workload variations, avoiding overloads, and optimizing costs. Therefore, the choice of grouping strategy should be based on a careful analysis of the application's requirements and the characteristics of the environment, aiming to maximize benefits without compromising flexibility and the system's ability to evolve.

For future work, the following directions are suggested:

- **Grouping within Kubernetes Pods**: Extending the research to evaluate the impact of grouping microservices within the same Kubernetes pod rather than the same container.
- **Impact studies on diverse workloads:** Expand experiments to include more varied workloads, such as data-intensive applications or financial transactions, which have specific performance requirements.
- **Integration with serverless architectures:** Evaluate how combining microservices grouping with serverless architectures can contribute to cost reductions and improve application elasticity.
- **Automation and operational simplification:** Develop tools and frameworks to reduce the operational complexity associated with microservices grouping, facilitating the adoption of this strategy in production environments.

These future works have the potential to expand knowledge on the use of microservices grouping strategies and HPA, contributing to the advancement of more efficient and scalable practices in microservices-based systems.

References

[1] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. In *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pages 000149–000154. IEEE, 2018.

[2] Fernando HL Buzato and Alfredo Goldman. Optimizing microservices performance and resource utilization through containerized grouping: An experimental study. In *2023 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 115–122. IEEE, 2023.

[3] Fernando HL Buzato and Alfredo Goldman. Extending microservices performance optimization through horizontal pod autoscaling: A comprehensive study. In *2025 International Parallel and Distributed Processing Symposium*. IEEE, 2025.

[4] Fernando HL Buzato, Alfredo Goldman, and Daniel Batista. Efficient resources utilization by different microservices deployment models. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–4. IEEE, 2018.

[5] Martin Fowler and James Lewis. Microservices a definition of this new architectural term.

[6] Malith Jayasinghe, Jayathma Chathurangani, Gayal Kuruppu, Pasindu Tennage, and Srinath Perera. An analysis of throughput and latency behaviours under microservice decomposition. In *International Conference on Web Engineering*, pages 53–69. Springer, 2020.

[7] Nane Kratzke. About Microservices, Containers and their Underestimated Impact on Network Performance. *Proceedings of CLOUD COMPUTING*, 2015:165–169, 2015.

[8] Kyungwoon Lee, Youngpil Kim, and Chuck Yoo. The impact of container virtualization on network performance of iot devices. *Mobile Information Systems*, 2018(1):9570506, 2018.

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

[10] Cristian Ruiz, Emmanuel Jeanvoine, and Lucas Nussbaum. Performance evaluation of containers for hpc. In *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers 21*, pages 813–824. Springer, 2015.

[11] Dharmendra Shadija, Mo Rezai, and Richard Hill. Microservices: granularity vs. performance. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 215–220, 2017.

[12] Yao Sun, Lun Meng, Peng Liu, Yan Zhang, and Haopeng Chan. Automatic performance simulation for microservice based applications. In *Methods and Applications for Modeling and Simulation of Complex Systems: 18th Asia Simulation Conference, AsiaSim 2018, Kyoto, Japan, October 27–29, 2018, Proceedings 18*, pages 85–95. Springer, 2018.

[13] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *2016 IEEE international symposium on workload characterization (IISWC)*, pages 1–10. IEEE, 2016.

[14] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems.* Maarten van Steen Leiden, The Netherlands, 2017.