Performance evaluation of microservices featuring different implementation patterns

Leandro Costa[1,3][0000−0003−4973−1093] and António Nestor Ribeiro[1,2,3][0000−0002−0681−8909]

[1] University of Minho, Braga Portugal
[2] Dept. of Informatics/University of Minho, Braga Portugal
[3] HASLab/INESC TEC Braga, Portugal

Abstract. The process of migrating from a monolithic to a microservices based architecture is currently described as a form of modernizing applications. The core principles of microservices, which mostly reside in achieving loose coupling between the services, highly depend on the implementation approaches used. Being microservices a complete change of paradigm that contrasts with the traditional way of developing software, the current lack of established principles often results in implementations that conflict with its alleged benefits. Given its distributed nature, performance is affected, but specific implementation patterns can further impact it. This paper aims to address the impact that microservices-based solutions, featuring different implementation patterns, have on performance and how it compares with monolithic applications. To do so, benchmarks are conducted over one application developed following a traditional monolithic approach, and two equivalent microservices-based implementations featuring distinct inter-service communication mechanisms and data management methodologies.

Keywords: Microservices · Monolithic · Inter Process Communication · Distributed Systems · Performance evaluation.

1 Introduction

Due to the monolithic applications' tightly coupled nature keeping up with the high demand and request for new functionalities constitutes a challenge [1]. Emphasized by the current demand for continuous delivery, as applications grow more complex, traditional architectural styles start to exhibit limitations. This has resulted in different architectural styles being considered [2].

Microservices [3] allegedly solve the inconveniences presented by monolithic applications by introducing modularity. Contrasting with the existence of a singular codebase that runs in a single process, microservices are composed of multiple small services that collaborate and run on its isolated process. These characteristics promote a wide range of quality attributes often required by modern applications, namely, maintainability, reusability, scalability, availability, and more [4]. Despite that, it also comes with their set of overheads, which extend to

the overall complexity of the system and even performance [7], which is affected by the added communication latency of a distributed system.

Given that microservices are not the traditional way of developing applications, there is still a degree of unfamiliarity [5] which often results in implementations that fail to reach their potentialities [6]. To achieve the promised benefits there are a set of principles to follow, mostly related to maintaining the services decoupled. To assure that, the introduction of specific patterns is required, from the communication approach used to the data management methodologies applied. The multiple existent microservices patterns can impact performance differently, something not particularly sustained in the existent research work.

The present paper conducts performance benchmarks to microservices solutions featuring different implementation patterns in order to evaluate what is their impact on performance and how it compares with a monolithic counterpart.

1.1 Related Work

There are multiple studies that analyze the performance impact of microservices. Some experiments are conducted over applications that feature a synchronous inter-service communication approach [8, 9] and inappropriate data management patterns, which inevitably introduces coupling between the services. This constitutes an obstacle in achieving microservices' alleged benefits, which requires the introduction of specific patterns that can have an impact on performance.

As an example, Flygare and Holmqvist [10] conducted an experiment in order to compare performance between monoliths and microservices. However, similarly to most empirical studies, it features a synchronous collaboration style between the services, resulting in tight coupling and lack of fault isolation. Moreover, the setting was not ideal as it featured only two hosts, lacking the amount of communication that would take place in a realistic scenario.

The experiments conducted by Akbulut and Perros [11] analyze 3 different microservices implementation patterns, including asynchronous messaging. However, it features different types of applications for each pattern, which provides inconclusive data to compare results across the implementations.

In general, most of the studies point towards worse performance results for microservices architectures. However, comparative studies which analyze the same monolithic and microservices applications that follow the best practices and patterns are remarkably scarce. This means that the available research on the impact on performance of the different existent implementation patterns is very limited and can be somewhat misleading.

2 Methodology

In order to evaluate the impact on performance of a microservices-based architecture in a concrete and empirical way, one same e-commerce platform was developed from scratch following a monolithic architectural style and two architectural patterns based on microservices. The microservices solutions feature distinct inter-service communication mechanisms and ways of dealing with data.

To analyze the performance and capacity of the developed applications benchmark experiments using a load testing approach were conducted, which simulates user load in a realistic scenario. The approach used was to apply a workload of multiple requests, at a specific rate per time unit and assess how the system behaves under the load. Several test scenarios are conducted and for each iteration, the number of requests per minute (the ramp-up period selected) is progressively increased and metrics, such as response time and throughput, are analyzed. Together with the different implementation patterns used, the number of requests per unit of time constitutes the independent variables of the experiments.

The analysis of the response time, the time taken to fully complete the requests, may indicate what is the saturation point of an application. This point corresponds to the stage in which the load is so high that the application cannot respond in a timely fashion, drastically increasing the response time.

Another relevant metric is the throughput, which is the measure of the number of successful requests per time unit. This metric helps to determine the performance capacity of a system and indicates how much load it can handle. Along with the hardware metrics collected, throughput and response times are part of the dependent variables of the experiments.

The used workload for the experiments consists of the creation of an order, one of the most intensive operations of the system, which requires the interaction of multiple services and database accesses. Regarding the asynchronous microservices version, this specific operation spans over five different services and requires the publication of four events. All the performance tests were conducted using Apache Jmeter[4], which simulates traffic of requests that would take place in a real scenario. It also provides the hardware and performance-related metrics necessary to evaluate the performance of the systems, including the dependent variables specified above. JMeter was installed on a local laptop connected to the applications' network in which the test plans and thread groups were created.

3 Experiments setting

The experiments are conducted over different solutions that feature distinct implementation patterns, whose impact is evaluated. The technical details of the experiments and the used settings are described in the following sections.

3.1 Monolith

The monolithic version of the reference application was developed using Java combined with the Spring framework (version 2.4.4), Hibernate as the ORM, and the database used in production is MySQL. This version aims to portrait a traditional legacy application scenario with high availability assured. With the conduction of the performance benchmarks and subsequent comparison of results, it is targeted to evaluate how traditional applications perform in comparison to modern distributed solutions. The infrastructure used for the monolithic

[4] https://jmeter.apache.org/

version consists of two physical machines that host a virtual machine (VM) for each component. Although it was possible, containers were not used for the deployment of the monolithic version. Targeting a more traditional environment for a legacy application, VMs were used instead. Each of the VMs used, represented in Figure 1, consists of a machine with 2Gb of memory and a CPU with 2 cores. The MySQL database used across all experiments is hosted on a dedicated Linux server. Targeting a production-ready infrastructure, high availability was promoted by eliminating single points of failure. Through the introduction of redundancy on the applicational server, the use of Linux Virtual Servers as load balancers, and a virtual IP address, the architecture became more resilient and fault-tolerant.

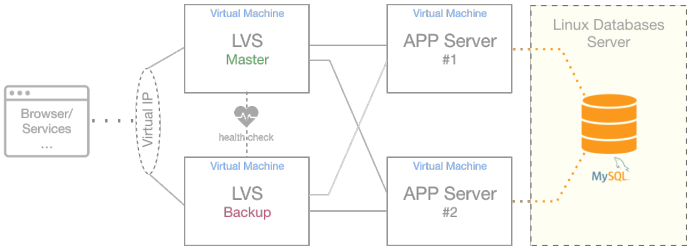

Fig. 1. Test case - Monolith

3.2 Microservices

The developed microservices solutions are built on the same domain model and use the same technological stack. The domain is split across multiple independent services that are responsible for a portion of the capabilities. Nonetheless, all the functionalities are identical and exposed via the same endpoints specification as the monolithic version, being the internal implementation the only distinction.

Assuring high availability in microservices requires different approaches comparing with monolithic applications. The selected orchestration tool to assist the management of the cluster, Docker Swarm, offers self-sufficient mechanisms which guarantee that if a host from the cluster fails then all the services running on that machine will be restarted in another. These mechanisms take action without human interaction, keeping the cluster constantly at a designated state. Furthermore, Docker Swarm offers easy access to scalability, making it possible to scale the number of instances of a service. Regarding internal communication, all services are reachable by hostname thanks to the internal DNS server existent within the Swarm cluster, which nullifies the need for a discovery service.

For the experiments, Raspberry Pi 4[5] boards compose a four-node Swarm cluster (represented in Figure 2), a low cost and energy-efficient solution. The several services are grouped in stacks, a concept maintained by Docker Swarm, in which the services are initiated together and distributed across the nodes.

[5] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/

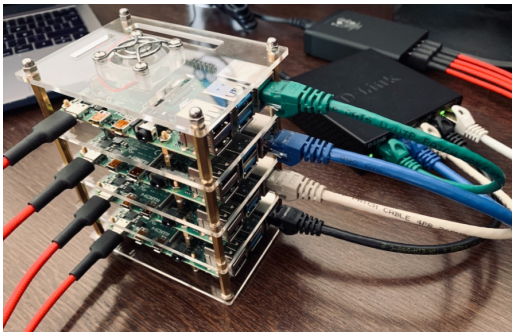

Fig. 2. Raspberry Pi cluster used in the experiments

Synchronous request/response microservices This microservices version developed is the one often featured in performance benchmarks research, which makes use of a request/response communication methodology, in this case, HTTP. This type of inter-service communication ultimately adds dependencies between the services, requiring them to be simultaneously healthy at a given time [3], which highly contrasts with the microservices principles of having decoupled and isolated services. Using this communication approach, a failure in a service may propagate to others and potentially cause a chain of failures.

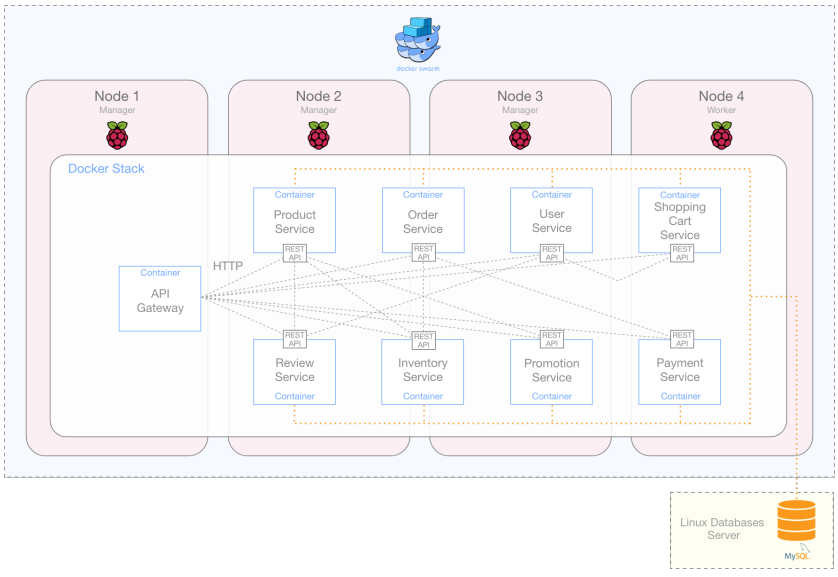

Fig. 3. Test case - Synchronous request/response microservices

By comparing this version's benchmark results with the monolithic version it is possible to directly evaluate the impact on the latency that comes from a distributed system. Given that this version does not feature data synchronization mechanisms, the only distinction between the monolithic version is the use of Saga Transactions[6] and different database schemas for each service. Containers were used for this version, being deployed on a Docker Swarm cluster composed of four Raspberry Pi 4 SBCs as the cluster nodes. For the database, an external Linux server is used, the same featured in the monolithic test case. All microservices versions deployed for the benchmarks use the same orchestration tool and physical hosts.

Asynchronous event-driven microservices The asynchronous version of the microservices solution consists of the same business services as the synchronous version, but with the addition of a message broker and two additional services, the read models.

The introduction of a message broker as the middleware responsible for all communication enables the existence of producer and consumer services that are completely unaware of each other, promoting loose coupling and fault isolation. Messaging was implemented using RabbitMQ[7] and messages are published to a topic that one or more services may subscribe to. Those events when consumed by the subscriber services will trigger a set of defined actions. The described behavior represents the flow of Saga Transactions, the approach used to perform operations that span multiple services, since each one has its own database.

The read models introduced in this version are the result of implementing the CQRS pattern[8]. By applying this pattern, views of frequently joined data are maintained and queried whenever a client application needs data, segregating the write and read operations. This pattern solves the complex querying-related issues by having replicas of the data that otherwise would have to be distributively joined. In order to maintain the replicated data, the services responsible for the read operations have to subscribe to all events that affect the maintained data. Although synchronizing replicas of data makes the services more independent from each other, there is an overhead associated with the data synchronization for the maintenance of replicas. Hence, it was predicted beforehand that this overhead may affect the benchmarks, as well as the eventual consistency outcome of using an event-driven approach along with Saga Transactions.

The conduction of benchmarks to this version aims to contribute to the lack of performance analysis of microservices that complies with the best practices, with no coupling between the services. The comparison with the other versions enables the evaluation of the impact of several factors, such as the use of a message broker and the overhead of having replicated data being synchronized.

For all the described experiments, the databases were kept outside of the cluster, since it is not recommended to keep such sensitive data inside a volatile

[6] https://microservices.io/patterns/data/saga.html
[7] https://www.rabbitmq.com/
[8] https://microservices.io/patterns/data/cqrs.html

cluster of containers. Regardless, there are scenarios that demand the synchronization of the data from the stateful services. These components when initialized in any of the nodes of the cluster should have access to a previously maintained state in order to continue operating. To assure the state is available across all nodes, a distributed file system (DFS) is normally used.

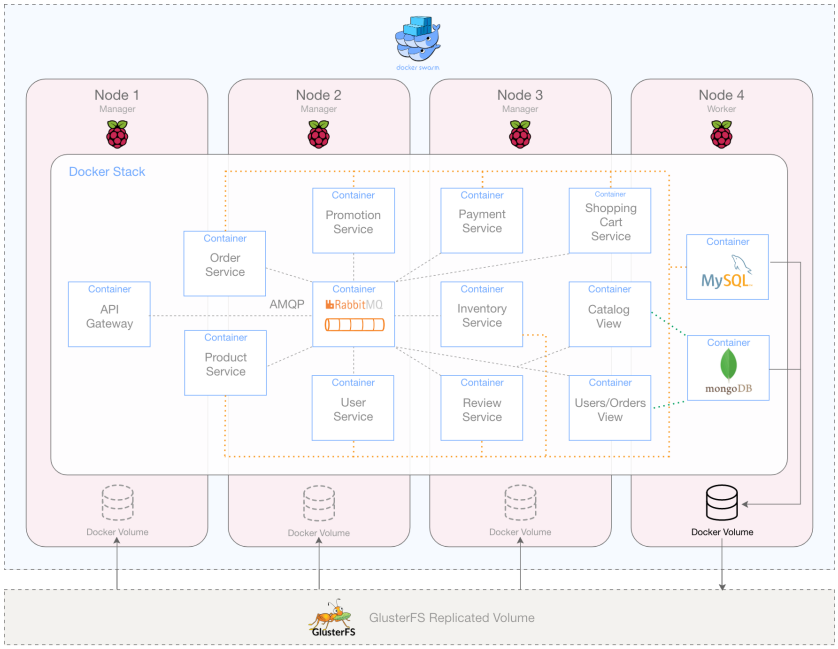

Fig. 4. Test case - Asynchronous event-driven microservices with a DFS

Thus, to evaluate the impact of using a DFS, the asynchronous microservices version was further split into two test cases, one with the databases hosted on a dedicated server, and one featuring database data synchronization directly across the nodes of the cluster (Figure 4). Given that the databases are stateful components, a DFS keeps the service's data available across all nodes and the one selected was GlusterFS[9], installed on each node that composes the cluster.

Both used databases offer cluster capabilities that could have been used instead, which deals with the data synchronization at an application level. However, the use of a DFS is something commonly necessary in a microservices system, so evaluating the impact of its introduction provides valuable data. GlusterFS was used to synchronize the MySQL and MongoDB data, making it available across all nodes and allowing its initialization in any physical instance. Although it was predicted beforehand that there would be a lot of I/O

[9] https://www.gluster.org/

load due to the intensive data synchronization mechanisms, how it impacts performance composes one of the endorsed test cases. The comparison between the results from the two asynchronous versions provides information on how the introduction of a DFS for state synchronization within a cluster affects the overall performance of a system.

4 Results and discussion

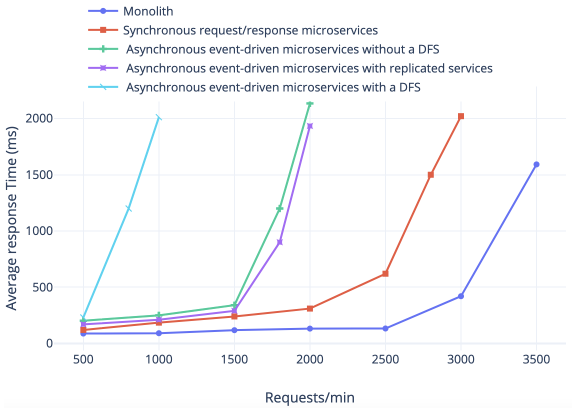

Fig. 5. Response times obtained from the benchmarks

The performance tests (Figure 5) revealed that the monolithic version exhibited the best performance results and the asynchronous versions the worst. That is partially explained by the fact that within a monolithic application all operations take place locally at method-level calls. In contrast, event-based microservices communicate with multiple services to perform the same operations, asynchronously. Regardless, simply by being a distributed system, there is latency in the communication between the services which may reside in different physical hosts. The existence of latency also explains why the synchronous microservices version has performed worse than the monolithic solution. Furthermore, in the monolith, database operations are executed using local ACID transactions, while in the microservices versions they proceed through Saga Transactions. This approach often requires multiple database accesses, which is the case of this workload, further contributing to higher response times in the asynchronous versions.

The steps of an operation within the asynchronous versions take place eventually, in which notifications are placed in queues to be eventually processed. This methodology, as stated, provides a great degree of resilience since services do not directly depend on each other. Additionally, the asynchronous versions maintain replicated data that is constantly being synchronized, adding load that is not present in the synchronous version nor the monolith. The differences in

performance between the asynchronous and synchronous microservices versions are also justified by the fact that the first one communicates through a message broker, adding even more latency to the communication process. The cost of not having a sequential flow of events and the overhead of synchronizing replicated data, as expected, had a negative impact on the performance benchmarks. There is a clear trade-off between having loosely coupled services and performance.

For these reasons, the worst performance values came from the asynchronous versions, which reaches the saturation point earlier comparing with the synchronous version and monolith. Although it was expected, the collected hardware metrics were analyzed to identify possible bottlenecks and potentially mitigate them.

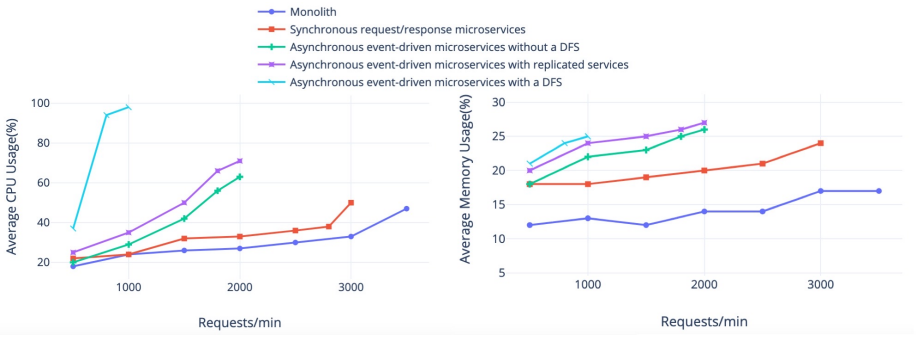

Fig. 6. Hardware metrics collected during the load tests

As an attempt of improving the results and delay the saturation point of the asynchronous versions, the main participant services were scaled. However, replicating the number of running instances of some containers did not considerably improve the response time results nor delay the saturation point. This is explained by the fact that the resource capacity of the services had not reached its limits, which is showcased by the acceptable CPU usage percentage during the entirety of the load tests (see Figure 6). The effect of scaling the services only resulted in the CPU utilization being more uniformly distributed across the nodes, which did not even reach its full capacity after hitting the saturation point. Furthermore, inside the configuration file of a Docker Swarm stack, it is possible to set resource constraints. Several combinations of CPU and memory limits were tested but failed to improve the results.

Hence, the performance bottleneck did not reside in the CPU nor memory usage, but in the database systems. The asynchronous versions perform up to five times more database operations than the other versions since it maintains replicated data that needs to be synchronized. Essentially, at the cost of having more isolated services with all the data needed to fulfill its operations, databases are more often accessed. Therefore, since the database systems can be described as being the bottleneck and given that the asynchronous version features more

database operations, it means its performance is the most affected by this bottleneck, which explains its eager saturation point.

However, when the database is still capable of handling the load, while the cadence of requests is still low, the response times were better when replicated instances were introduced due to the possibility of parallel processing. If the bottleneck was due to capacity limitation from the containers, showcased by a high CPU usage, having replicated instances would not only improve the response times but also delay the saturation point. Nonetheless, in order to delay the saturation point across all versions whose bottleneck is the database systems, a more performant database infrastructure would have to be introduced, for instance, a dedicated cluster with sharding mechanisms and parallel processing. Regardless, with the exact same database system, the benchmark results highly differ across the different implementations.

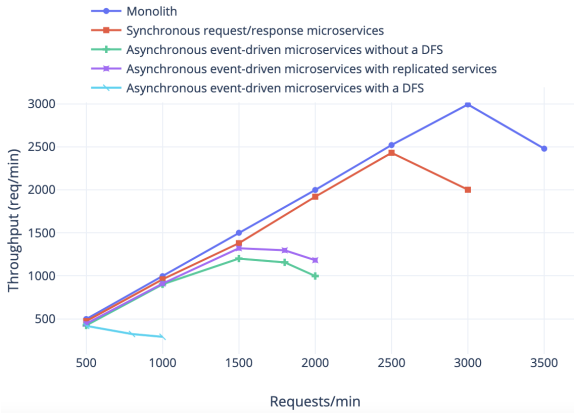

Fig. 7. Throughput values obtained from the benchmarks

By comparing the benchmarks from the monolithic and synchronous microservices version it is possible to inspect what is the direct impact of having a legacy application made distributed. The synchronous microservices version does not maintain replicas of data. Thus, the only difference between the monolithic version is that operations are executed through Saga Transactions, in which the different services spawn each other in a request/response manner. Although the monolith featured better response times and a later saturation point, the benchmarks results between the two are similar.

In terms of throughput values, they were also analyzed as they provide insights of what is the performance capacity of the system (Figure 7). Regarding the asynchronous versions, since the inter-service communication does not occur through a synchronous request/response style, the timeout periods that are associated with it do not apply, making the error rate zero due to its eventual consistency nature. The throughput values of this approach are affected by that

factor, while in the synchronous versions, requests that take too long, or are discarded, simply provoke an increase in the error rate.

The monolithic version was capable of successfully complete up to 3000 requests per minute before reaching the saturation point. This indicates that any load bigger than that has a substantially higher response time or will increase the error rate. The maximum capacity of the synchronous microservices version was slightly lower, capable of completing up to 2500 requests per minute. The throughput starts to decrease once the saturation point is reached in both versions, which showcases what is the maximum capacity of the system.

Regarding the two asynchronous versions, it was evaluated how the introduction of a DFS impacts the benchmark results. The database systems are the bottleneck across all versions, except for the version that uses a DFS, whose bottleneck resides in resource capacity limitations, highlighted by the high CPU percentage across all nodes (Figure 6). This version requires the synchronization of the database data every time a write operation takes place, which represents a limitation with significant overheads in terms of resources utilization. The CPU usage percentage when using a DFS has increased significantly which ends up becoming the performance bottleneck in this case, reaching the saturation point considerably earlier. Since GlusterFS is primarily a file-system storage, replication does not occur at a block level, which causes a great impact on the CPU utilization, since during load tests there is constantly synchronization happening.

As previously mentioned, the asynchronous version is composed of services that possess shared replicated entities, which demands the publication of events whenever data changes so the consumer services can keep all the replicas consistent. The maintenance of shared entities is already a considerable overhead, increasing the number of database operations per request when comparing with the synchronous versions. On top of that, having database data files being constantly synchronized across multiple instances requires a substantial amount of resources. The combination of all these mechanisms demands a very performant infrastructure to have similar performance results as its synchronous counterparts. The results have clearly demonstrated that when data is synchronized across the nodes of the cluster, the overall performance substantially decreases.

5 Conclusions

In this paper, performance benchmarks were conducted on monolithic and microservices applications. Furthering the comparison between the two architectural styles, the effect on performance of multiple variations of microservices scenarios was tested. The impact of having different inter-service communication mechanisms, data management methodologies, and even the utilization of a distributed file system was evaluated.

The microservices version that features the best practices in terms of having loosely coupled services, the event-based version, was the one with worse response times. Besides the expected influence from the added latency due to its distributed nature, the results demonstrated the impact on performance of hav-

ing the necessary mechanisms to achieve service isolation. The use of a different database for each service, the introduction of patterns such as Saga Transactions and CQRS, which introduces eventual consistency and data synchronization overhead, showcased a negative impact on the benchmark results.

The typical test case scenario for benchmark experiments involving microservices features synchronous communication without data synchronization. This version was tested and proved to be the one with the best performance results among all microservices implementations. The other test cases exhibited worse benchmark results, indicating that microservices implementations that follow best practices will perform worse than what is presented in existent experiments.

Nonetheless, regardless of which implementation patterns are featured, the performance results are in accordance with most of the existent related work, in the sense that microservices will perform worse than a monolithic counterpart.

Acknowledgments This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDB/50014/2020.

References

1. N. Kratzke, P. Quint: Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. pp. 1–16. Journal of Systems and Software (2017)
2. D. Taibi, V. Lenarduzzi, and C. Pahl: Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing (2017)
3. N. Dragoni et al.: Microservices: Yesterday, Today, and Tomorrow. Present and Ulterior Software Engineering, pp. 99–110. Springer (2016)
4. R. Chen, S. Li and Z. Li: From Monolith to Microservices: A Dataflow-Driven Approach. In: 2017 24th APSEC on Proceedings, pp. 466—475. (2017)
5. A. Carrasco, B. Bladel, and S. Demeyer: Migrating towards microservices: migration and architecture smells. In: 2nd International Workshop on Refactoring on Proceedings, pp. 1–6. Association for Computing Machinery (2018)
6. P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. IEEE Software, pp. 24–35. (2018)
7. S. Baskarada, V. Nguyen, and A. Koronios: Architecting microservices: Practical opportunities and challenges. Journal of Computer Information Systems (2018)
8. V. Singh and S. K. Peddoju: Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation (ICCCA) on Proceedings, pp. 847–852. IEEE (2017)
9. O. Al-Debagy and P. Martinek: A Comparative Review of Microservices and Monolithic Architectures. In: IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI) on Proceedings, pp. 149–154. IEEE (2018)
10. R. Flygare and A. Holmqvist: Performance characteristics between monolithic and microservice-based systems. Bachelor's Thesis, Faculty of Computing Blekinge Institute of Technology Karlskrona, Sweden, (2017)
11. A. Akbulut and H. G. Perros: Performance Analysis of Microservice Design Patterns. IEEE Internet Computing, pp. 19–27. IEEE (2019)