Performance Comparison between a Monolithic and a Microservice Application

Reinhard Bernsteiner[1,*,†], Marco Blasisker[1,†], Michael Kohlegger[1,†], Christian Ploder[1,†] and Stephan Schlögl[1,†]

[1] *Management Center Innsbruck, Universitaetsstrasse 15, 6020 Innsbruck, Austria*

Abstract

The evolution from monolithic to cloud-native microservices architectures represents a fundamental shift in software engineering practices, driven by the increasing demands for scalability, reliability, and rapid deployment in modern digital environments.

The fundamental distinction between monolithic and microservices architectures lies in their codebase configuration and component organization. Microservices employ multiple, independent codebases distributed across separate services within a distributed system, while monolithic architectures operate as singular, cohesive units with unified codebases that create interdependencies among all components. This structural difference has cascading effects on various system design, deployment, and maintenance aspects.

Due to these advantages, migrating existing monolithic systems to a microservice architecture might be beneficial because of a potential high. This decision must be evaluated thoroughly since the migration effort can be very high.

To gain practical insights into migrating a typical monolithic application, selected components of a webshop were migrated as a representative example. This approach aims to highlight the essential steps involved in the migration process and identify potential challenges. Subsequently, the newly developed system is evaluated against the original monolithic version, focusing on performance. The findings from this prototype migration are intended to serve as a foundation for guiding future migration decisions

Keywords

Monolithic Application, System Performance, Microservice Architecture, System Migration, Cloud-native Application

1. Introduction

The evolution from monolithic microservices architectures represents a fundamental shift in software engineering practices, driven by the increasing demands for scalability, reliability, and rapid deployment in modern digital environments [1]. Monolithic architectures, characterized by unified codebases and integrated components, have traditionally served as the foundation for software development but face significant limitations in contemporary cloud computing environments [2]. In contrast, cloud-native applications built on microservices principles offer enhanced scalability, fault tolerance, and development agility through their containerized, independently deployable service components [3].

While monolithic systems may remain viable for specific use cases, the transition to cloud-native microservices architecture delivers substantial operational efficiency, system resilience, and organizational adaptability, particularly for enterprises experiencing rapid growth or requiring high-availability services [4].

SQAMIA 2025: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 20--12, 2025, Maribor, Slovenia

* Corresponding author.

† These authors contributed equally.

reinhard.bernsteiner@mci.edu (R. Bernsteiner); marco.blasisker@mci.edu (M. Blasisker); michael.kohlegger@mci.edu (M. Kohlegger); christian.ploder@mci.edu (Ch. Ploder); stephan.schloegl@mci.edu (St. Schlögl)

0000-0002-8142-3544 (R. Bernsteiner); 0000-0002-4538-973X (M. Kohlegger); 0000-0002-7064-8465 (Ch. Ploder); 0000-0001-7469-4381 (St. Schlögl)

While the benefits of microservices architecture are substantial, organizations must carefully evaluate their specific circumstances and requirements before initiating migration efforts [5]. The scope of migration varies depending on how a monolithic application is structured and built [6]. The project's central aim is to gain experience migrating a typical monolithic application to a microservice system using state-of-the-art technologies and tools. The newly developed application is compared with the original monolithic version in terms of performance. Performance is a key characteristic of microservices and includes several dimensions. In this project, the services' response time and CPU usage are measured. Parts of a monolithic webshop are migrated as an example.

This leads to the research question for this project: What impact does the migration of a monolithic application to a microservice-based application have on performance (response time and CPU usage)?

Developing and validating a prototype can be justified as a scientific method to answer the research question. Prototypes allow to test whether proposed ideas, theories, or designs are practical and feasible before committing to full development [7]. This early-stage validation reduces the risk of pursuing unworkable concepts and saves time and resources [8]. The results of this prototype migration are intended to help establish a foundation for future migration decisions.

2. Theoretical Background

This section presents the central theoretical concepts relevant to the migration project.

2.1. Monolithic Architecture

Monolithic architecture represents the traditional approach to software development, where applications are constructed as unified, self-contained units with a single codebase that couples all business concerns together. This architectural model treats the entire application as a singular, extensive computing network, requiring comprehensive updates to the whole stack when any modifications are necessary. The monolithic approach has historically provided simplicity in development and deployment, as it involves managing a single executable file that encompasses all application functionality [9].

However, monolithic systems present significant scalability challenges as applications grow in complexity and user base. The integrated structure of monoliths creates bottlenecks when attempting to scale specific components independently, often requiring scaling the entire application even when only particular features experience increased demand [10]. This limitation becomes particularly problematic for organizations experiencing rapid growth, as demonstrated by Netflix's infrastructure challenges in 2009, ultimately leading to their pioneering migration from monolithic to microservices architecture [11].

The coupling of components in monolithic systems also creates reliability concerns, where failures in one component can potentially compromise the entire application's functionality. This lack of fault isolation means that a single point of failure can result in complete system downtime, making monolithic architectures less resilient compared to distributed alternatives. Additionally, monolithic systems often constrain development team organization and technology choices, as all teams must work within the same technological framework and coordinate changes across the entire codebase [12].

2.2. Microservice Architecture

This kind of software architecture represents a paradigm shift toward microservices-oriented, containerized, and dynamically orchestrated systems designed to optimize resource utilization in cloud computing environments. These applications are fundamentally structured as collections of small, independent services that can be developed, deployed, and scaled autonomously while maintaining loose coupling between components. The microservices architecture adheres to several key principles that distinguish it from traditional monolithic approaches [13].

To meet the flexible requirements of microservices, containers are used to encapsulate each module, which replaces the traditional virtual machine (VM) operating mode. Container can be started and stoped quickly to optimize resource consumption [14]. Kubernetes is a container orchestration system that manages the deployment, scaling, and operation of containers across a cluster. Kubernetes typically runs on clusters of VMs provided by cloud platforms or private infrastructure [15].

The single responsibility principle ensures that each microservice focuses on a specific business function or capability, promoting maintainability and enabling easy replacement of individual services without affecting the broader system [16]. Decentralized governance empowers development teams to make independent decisions regarding technology choices, programming languages, and frameworks that best suit their specific service requirements. This autonomy facilitates faster development cycles and allows organizations to leverage the most appropriate tools for each service's unique needs [17].

Independent deployment capabilities represent another crucial advantage of microservices architecture, enabling faster release cycles and reducing the risk of deployment failures. Each service can be updated, scaled, or modified without requiring changes to other system components, significantly improving development velocity and reducing coordination overhead [18]. Fault isolation mechanisms ensure that failures in one microservice do not cascade throughout the system, enhancing overall resilience and maintaining service availability even when individual components experience issues [19].

Containerization serves as the foundation for cloud-native microservices deployment, providing lightweight, standardized units that package application code with all necessary dependencies. Containers enable rapid scaling capabilities and portability across different environments, while orchestration platforms like Kubernetes manage the complexity of deploying and scaling containerized applications. The stateless nature of many microservices further enhances scalability by eliminating persistent memory requirements and enabling dynamic resource allocation based on demand [20].

2.3. Performance Testing

Performance testing is part of software testing and is an essential phase in the software development process for evaluating a software product. New problems can arise with web applications due to high user numbers and the associated higher load. The performance of a web application is typically determined under normal load using a load test and under very high load using a stress test. Different metrics can be specified with the tests.

3. Description of the Monolithic and Microservice Application

This section gives an overview of the architecture of the monolithic and microservices application. Furhtermore, the use technologies and toos for both systems are presented.

3.1. Monolithic System

ShoppingCart is an e-commerce website developed with Java. Users can look closely at products and add them to their virtual shopping cart. The user must be logged in and registered to use the shopping cart function. To successfully place the order, there is a typical order flow where the user's data is entered again, and a final order button. There is also an About Us page with information about the store and a contact page to contact the operator. Admins can also add products and edit existing products.

The application of the monolithic application is based on a model-view-controller architecture. The front end is realized by JSP pages, which are coupled with the Spring controllers. An H2 database is used as the database. The data models are created with Hibernate in Java. The build tool Maven is

used for compilation and deployment. Maven creates a war (Web Application Resource) file, which is executed using Apache Tomcat Server.

3.2. Microservice System

The individual components of the sample application are ported into independent microservices. The new microservice architecture creates a distributed system for which new interfaces must be made. To determine how large a microservice should be, the various data models are considered and analyzed to determine which data is closely related and which is independent. Similar data is grouped, and the individual services are derived from this. This results in the microservices product service, user service, cart service, and order service for the ShoppingCart application. In case of doubt, services should be roughly granulated, as these can always be broken down further if necessary or advantageous. REST interfaces are created to communicate with the microservices.

An API gateway is also required, containing the front end and the interface between users and all microservices [21]. The API gateway uses the newly implemented REST interfaces to communicate with the various microservices. A discovery service is also built to provide better name resolution during communication and an overview of all running microservices. With the help of a load balancer, it is possible to have a balanced load even with multiple instances of a microservice. This enables the scalability of a microservice-based system.

3.3. Overview of the Used Technologies and Tools

ShoppingCart uses Spring MVC as the basis. Dynamic HTML pages are rendered with the help of JSP, and Spring Security is also used for authentication, as well as Spring Webflow for the check-out process. An H2 database is used, and Hibernate is the interface. Maven is also used to manage dependencies and as a build tool. The migration to microservices means that Spring MVC can be dispensed completely. Spring Boot is used instead. In addition, JSP is replaced with AngularJS and the H2 database is replaced with a MariaDB. Finally, Spring Webflow must be removed, as Spring Webflow is not compatible with AngularJS. The technologies are summarized once again in table 1:

Table 1
Overview of the Used Technologies and Tools

Monolith	Microservices	Comments
Spring-Webmvc	Spring Boot	Spring MVC is a Model-View-Controller framework for web applications. Spring Boot is used for REST APIs and requires less configuration. It is, therefore, well suited as a foundation for microservices.
JSP (Java Server Pages)	AngularJS	Java Server Pages are compatible with Spring Web MVC. A different frontend technology is required since REST APIs are used for microservices. AngularJS was chosen here because it is compatible, and there is much online information for the migration.
H2 Database	MariaDB	MariaDB has a large community and will make cloud deployment easier in the future.
Hibernate	Hibernate	Not Changed
	Spring Webflux	Spring Webflux includes WebClient, an interface that enables web requests for reactive websites. Spring Security will also be migrated to Spring Webflux Security.
Spring Security	Spring Security	Despite Spring Webflux Security, Spring Security is still used to configure web filters.
Spring Webflow		Spring Webflow is not compatible with AngularJS. The order flow must, therefore, be rebuilt.

Maven	Maven	Not Changed
	Spring Cloud	Spring Cloud is used for discovery services and internal load balancing.

3.4. Runtime Environment for the Monolith System

To reduce possible differences in performance caused by the runtime environment, a branch of the original version is used. This branch has already been converted to Spring Boot and MariaDB. The application is started in the Google Compute Engine. This is based on the same "e2-medium" computer type and the same image as the Kubernetes instances. This is to avoid performance differences due to different computing power.

3.5. Runtime Environment for the Microservice System

The microservices are started in the Google Kubernetes Engine, as described in the Deployment section. An external IP address is assigned for the API gateway, and port 8080 is forwarded. The cluster is set up so that 6 nodes are used. The "e2-medium" machine is selected as the computer type for the nodes. This offers 4GB RAM and 2 virtual CPUs. For the performance comparison, automatic scaling is deactivated for the time being, and instead, a fixed scaling factor of 2 instances is set for each service. This represents approximately how the system would run in standard operation.

4. Performance Measurements

In this chapter, comparisons are made between the original monolithic application and the migrated microservice application. The performance for various use cases is recorded and compared for both application versions.

The microservices are packaged into containers using Docker and deployed in a Kubernetes cluster at any cloud provider. When Kubernetes operates on virtualized infrastructure, VM clusters can be provisioned, managed, and scaled up or down with relative ease. Most cloud providers offer virtual machines as their fundamental compute units, along with services specifically designed to support Kubernetes deployments.

All measurements are carried out using the Apache JMeter tool. The following dimensions are used to measure performance [22,23]:

a. Response time: comparison of the monolith with microservices; response time is the time between sending a request and receiving the complete response
b. CPU usage for the following services:
    - API-gateway: API gateway required, which contains the front end and serves as an interface between users and all microservices
    - The discovery service can recognize unreachable services, remove them, or redirect requests to available instances.
    - Cart-Service: This service represents the shopping cart. For each customer, a new cart is created
    - Order-Service: This service handles customer orders
    - User-Service: The User-Service is responsible for registration and login

4.1. Test Plan

To measure performance, a fixed sequence of queries is defined, which should cover the functions and workflow of the application (Table 2). This is to ensure that a real workload is simulated. JSP pages are called in the monolithic architecture, whereas, in the microservice architecture, the newly implemented REST interfaces can be accessed.

Table 2
Tasks and Related Requests

Task	Monolith	Microservices
AboutUs	GET /shopeasy/aboutus	GET /#!/aboutus
Login	POST /shopeasy/login	POST /login
AddToShoppingCart	GET /shopeasy/cart/add/1	PUT /api/cart/carts/addProductToCart/2
GetCart	GET /shopeasy/cart/getCart/6	GET /api/cart/carts/6
CheckOut	GET /shopeasy/order/6	POST /api/order/orders/newOrder

There are four test cases (Table 3) whereby the HTTP query process is always the same. However, the number of threads that simulate the number of users and the number of repetitions changes for each test. Each test triggers a different load, and it is possible to compare how the two applications behave. Ramp-up is the time it takes for all users to be added to the test. Even in a productive environment, the number of users does not typically increase to 100 percent from one second to the next. The ramp-up time should, therefore, simulate a natural increase in the number of users.

Table 3
Test Cases

Test case	Nr Users	Nr Repetitions	Ramp-up
Test 1	1	1000	1
Test 2	5	100	5
Test 3	20	100	10
Test 4	50	150	30

4.2. Performance Results – Response Times

Table 4
Performance Results – Response Times (in milliseconds)

Test case	Average Microservice	Average Monolith	90% percentile Microservice	90% percentile Monolith
Test case 1	12 ms	14 ms	19 ms	16 ms
Test case 2	13 ms	29 ms	19 ms	21 ms
Test case 3	16 ms	308 ms	26 ms	1256 ms
Test case 4	197 ms	1075 ms	610 ms	2032 ms

Table 4 clearly shows that an increasing workload leads to huge differences between the two architectures. In Test case 1, both systems have similar response times. With just 5 users and 100 repetitions, differences in reaction times can already be determined. When testing with 20 simulated users simultaneously, apparent differences between the two architectures can be seen. Test case 4 also represents a very high load for microservices; the load on the monolith is more than 5 times as high.

4.3. CPU Usage – Test Case 1

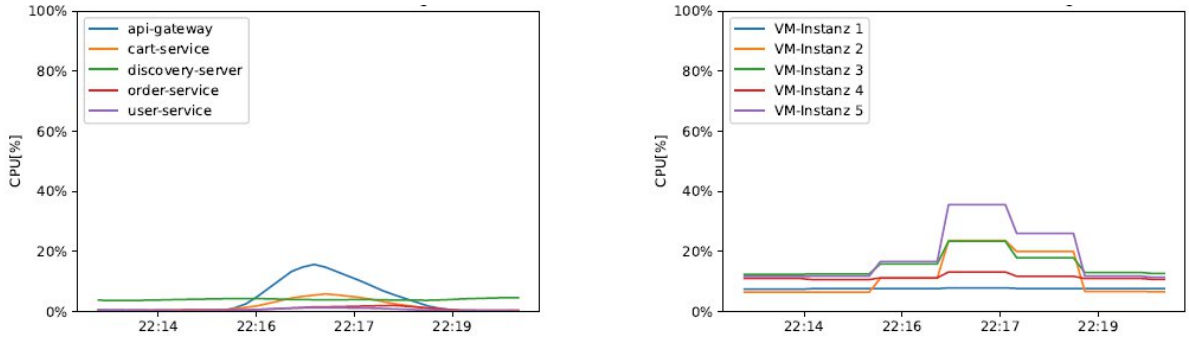

Figure 1: Test Case 1 - CPU Usage of Services and VMs - Microservices

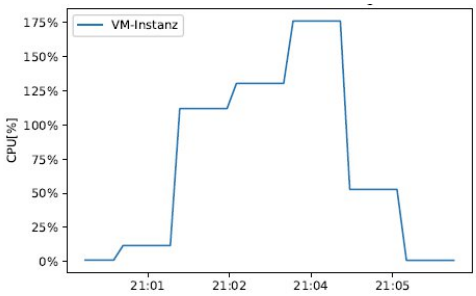

Figure 2: Test Case 1 - CPU Usage Virtual Machine - Monolith

During the test period, it can be seen that the CPU utilization of the microservices hardly increases (Figure 1, left). In this case the microservices are spread across 5 VMs. For the microservice solution, the metrics show that the CPU usage is always below 50 percent (Figure 1, right), compared to over 100 percent for the monolithic solution (Figure 2). This is possible because the e2-medium computer type of the instance has CPU bursting, which allows the instance to temporarily access additional CPU capacity.

4.4. CPU Usage – Test Case 2

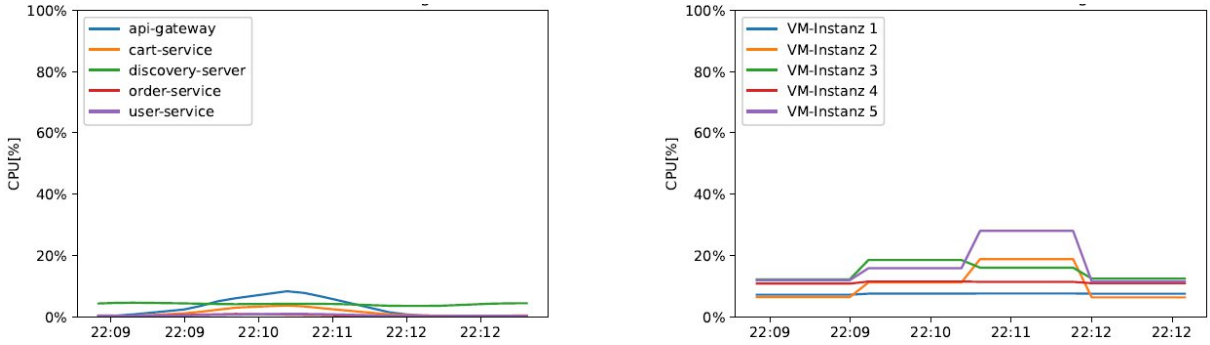

Figure 3: Test Case 2 - CPU Usage of Services and VMs - Microservices

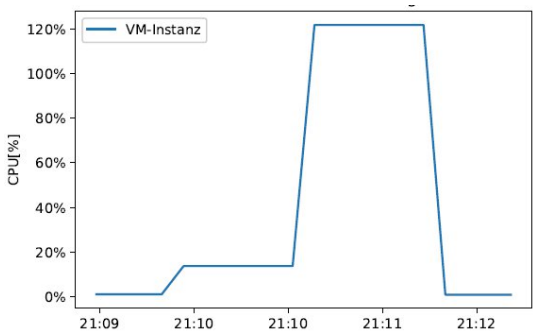

Figure 4: Test Case 2 - CPU Usage Virtual Machine - Monolith

With five simulated users, where all requests are repeated 100 times, no microservice uses more than 20 percent (Figure 3, left) of the CPU. This picture is also reflected in the utilization of the VM instances (Figure 3, right). The VM of the monolithic software activates burst mode again, as the CPU has already reached its limits, as depicted in Figure 4.

4.5. CPU Usage – Test Case 3

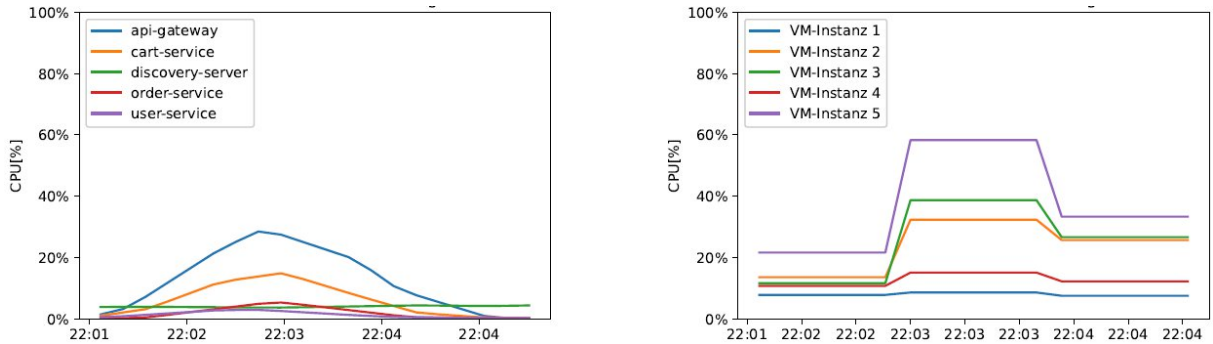

Figure 5: Test Case 3 - CPU Usage of Services and VMs - Microservices

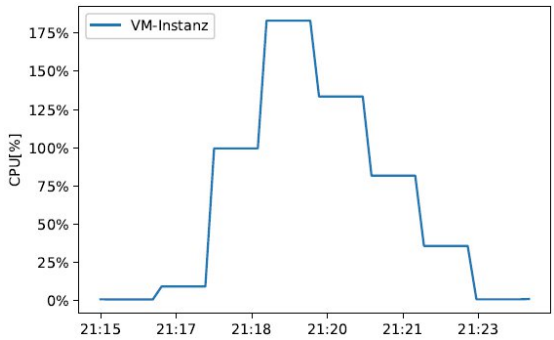

Figure 6: Test Case 3 - CPU Usage Virtual Machine - Monolith

The microservices (services and VMs) have not yet reached their limits. The left part of Figure 5 represents the CPU utilization of the services, the VMs CPU utilization can be seen on the right side of Figure 5. The VM of the monolith cannot cope with the load even in bursting mode, as shown in Figure 6. These additional resources are limited to around two minutes. From this point onwards, there are substantial deviations in the server response times.

4.6. CPU Usage – Test Case 4

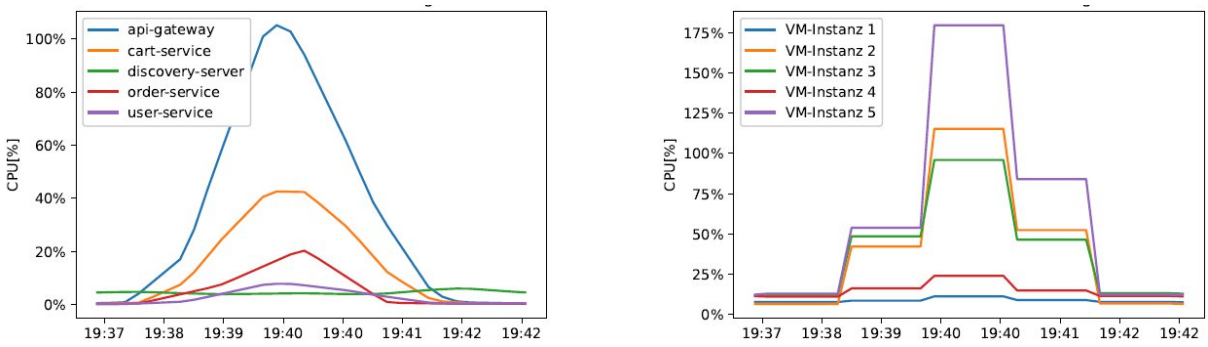

Figure 7: Test Case 4 - CPU Usage of Services and VMs - Microservices

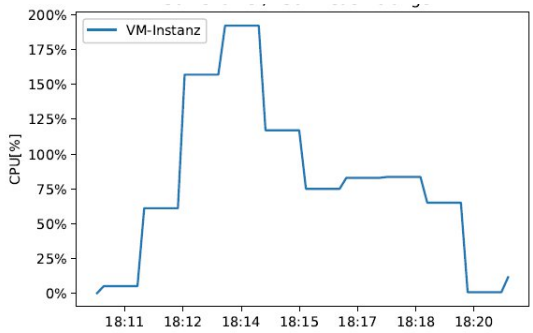

Figure 8: Test Case 4 - CPU Usage Virtual Machine - Monolith

In Google Cloud Monitoring, it can be seen that microservices also reach their limits with 50 users (Figure 7, left). The advantage, however, is that five VM instances are available for this workload. The bottleneck in this case are the two VM instances, each with an API gateway, which also enter burst mode during the test (Figure 7, right). To increase performance, the two API gateways' fixed scaling would have to be removed or increased. The VM with the monolithic architecture is running at full capacity throughout this test. It can be seen how the load initially rises to up to 200 percent (Figure 8)with the help of burst mode and then levels off at around 100 percent.

5. Conclusions

Regarding performance, particularly response time and CPU usage, it has been shown that microservices deliver significantly better results than the monolithic system. With Kubernetes, the services and the number of nodes in a cluster can be scaled automatically. This is a particular advantage when the system is used very dynamically. If the load is higher, the services can be split across several VM instances; if the load is lower, the number of nodes can be reduced again, saving costs.

An architecture based on microservices has further advantages compared to a monolithic architecture. Defective services can be detected with health check metrics and restarted automatically. With several replicas of a service, functionality is still guaranteed even if one of these replicas fails. Such a system's maintenance and further development are also more efficient than a monolithic system, especially if the system's complexity is high or increasing.

On the other hand, there is the migration effort. Therefore, It is essential to conduct a thorough analysis to clarify how the new system will be structured, which new interfaces need to be created, and which technologies and tools will be used.

Declaration on Generative AI

During the preparation of this work, the authors used Grammarly in order to: Grammar and spelling check. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

6. References

[1] P. Jatkiewicz, S. Okrój, Differences in performance, scalability, and cost of using microservice and monolithic architecture. In: J. Hong (ed.). *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, 1038–1041. New York,NY,United States: Association for Computing Machinery; 2023.

[2] H. Hassan, M. A. Abdel-Fattah, W. Mohamed, Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review, *International Journal of Advanced Computer Science and Applications* 15 (2024); doi: 10.14569/IJACSA.2024.0151013.

[3] G. Blinowski, A. Ojdowska, A. Przybylek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, *IEEE Access* 10, 20357–20374 (2022); doi: 10.1109/ACCESS.2022.3152803.

[4] S. Chinamanagonda, Revitalizing Legacy Systems: Proven Strategies for Successful Application Modernization, *Journal of Computing and Information Technology* 4 (2024).

[5] C. Zhong, H. Huang, H. Zhang, S. Li, Impacts, causes, and solutions of architectural smells in microservices: An industrial investigation, *Software: Practice and Experience* 52, 2574–2597 (2022); doi: 10.1002/spe.3138.

[6] D. Faustino, N. Gonçalves, M. Portela, A. R. Silva, Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation, *Performance Evaluation* 164, 102411 (2024).

[7] E. Bjarnason, F. Lang, A. Mjöberg, An empirically based model of software prototyping: a mapping study and a multi-case study, *Empirical Software Engineering* 28 (2023); doi: 10.1007/s10664-023-10331-w.

[8] I. Koskinen, J. Frens, Research Prototypes, *Archives of Design Research* 30, 5–14 (2017); doi: 10.15187/adr.2017.08.30.3.5.

[9] R. P. Singh, M. Thakur, S. M. A. Razavi, S. Sankhla, K. K. Singh, I. H. Limbasiya, Monolithic and Microservice Architecture: A Sustainable Approach. In: *2025 3rd IEEE International Conference on Industrial Electronics: Developments & Applications (ICIDeA)*, 1–6. IEEE; 2025.

[10] V. Velepucha, P. Flores, A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges, *IEEE Access* 11, 88339–88358 (2023); doi: 10.1109/ACCESS.2023.3305687.

[11] C. Henríquez, J. R. Valencia, G. Sánchez, Architectural Evolution from Monolithic to Microservices in Scalable Systems: A Case Study of Netflix, *Prospectiva* 23, 12 (2025).

[12] V. Velepucha, P. Flores, Monoliths to microservices - Migration Problems and Challenges: A SMS. In: *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*, 135–142. IEEE; 2021.

[13] Y. Abgaz, A. McCarren, P. Elger, D. Solan, N. Lapuz, M. Bivol, G. Jackson, M. Yilmaz, J. Buckley, P. Clarke, Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review, *IEEE Transactions on Software Engineering* 49, 4213–4242 (2023); doi: 10.1109/TSE.2023.3287297.

[14] Z. Ding, S. Wang, C. Jiang, Kubernetes-Oriented Microservice Placement With Dynamic Resource Allocation, *IEEE Transactions on Cloud Computing* 11, 1777–1793 (2023); doi: 10.1109/TCC.2022.3161900.

[15] S. Taherizadeh, M. Grobelnik, Key influencing factors of the Kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications, *Advances in Engineering Software* 140, 102734 (2020); doi: 10.1016/j.advengsoft.2019.102734.

[16] G. Buchgeher, M. Winterer, R. Weinreich, J. Luger, R. Wingelhofer, M. Aistleitner,

Microservices in a Small Development Organization. In: A. Lopes, R. de Lemos (eds.). *Software Architecture*, 208–215. Cham: Springer International Publishing; 2017.

[17] S. Pinto-Agüero, R. Noel, Microservices Evolution Factors: A Multivocal Literature Review, *IEEE Access* 13, 88707–88730 (2025); doi: 10.1109/ACCESS.2025.3570658.

[18] A. Sen, I. Skrobot, Implementation of DevOps paradigm to deployment and provisioning of microservices, *Issues In Information Systems* (2021); doi: 10.48009/1_iis_2021_136-148.

[19] L. Wang, Y. X. Jiang, Z. Wang, Q. E. Huo, J. Dai, S. L. Xie, R. Li, M. T. Feng, Y. S. Xu, Z. P. Jiang, The operation and maintenance governance of microservices architecture systems: A systematic literature review, *Journal of Software: Evolution and Process* 35 (2023); doi: 10.1002/smr.2433.

[20] P. Mahanta, S. Chouta, Translating a Legacy Stack to Microservices Using a Modernization Facade with Performance Optimization for Container Deployments. In: C. Debruyne, H. Panetto, W. Guédria, P. Bollen, I. Ciuciu, G. Karabatis, R. Meersman (eds.). *On the Move to Meaningful Internet Systems: OTM 2019 Workshops*: *Confederated International Workshops: EI2N, FBM, ICSP, Meta4eS and SIAnA 2019, Rhodes, Greece, October 21–25, 2019, Revised Selected Papers*, 143–154. Cham: Springer International Publishing; Imprint Springer; 2020.

[21] A. Makris, K. Tserpes, T. Varvarigou, Transition from monolithic to microservice-based applications. Challenges from the developer perspective, *Open Research Europe* 2, 24 (2022); doi: 10.12688/openreseurope.14505.1.

[22] V. M. Mostofi, D. Krishnamurthy, M. Arlitt, Fast and Efficient Performance Tuning of Microservices. In: C. A. Ardagna (ed.). *2021 IEEE 14th International Conference on Cloud Computing*: *CLOUD 2021 : virtual conference, 5-11 September 2021 : proceedings*, 515–520. Piscataway, NJ: IEEE; 2021.

[23] V. Ramamoorthi, AI-Enhanced Performance Optimization for Microservice-Based Systems, *Journal of Advanced Computing Systems* 4, 1–7 (2024); doi: 10.69987/JACS.2024.40901.