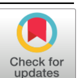
International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 30, No. 2, February, 2025, pp. 551-563
Journal homepage: http://iieta.org/journals/isi

Optimizing Microservices Performance and Scalability Through Automated Monitoring with Kubernetes and Prometheus



Cesar Felipe Henao Villa[1*], Camilo Andrés Echeverri Gutiérrez[2], Leidy Catalina Acosta Agudelo[2], David Alberto García Arango[2], Lisbet Maria Garzon Cano[3], Euris Santa Arboleda[3], Marlo Eliecer Hoyos Garcia[3]

[1] Faculty of Engineering, American University Corporation, Medellín 050012, Colombia
[2] Department of Research, Administrative Management Consultants S.A.S, Envigado 055428, Colombia
[3] Department of Technological Innovation, Somos Gestión Positiva S.A.S, Medellín 050021, Colombia

Corresponding Author Email: chenao@coruniamericana.edu.co

Copyright: ©2025 The authors. This article is published by IIETA and is licensed under the CC BY 4.0 license (http://creativecommons.org/licenses/by/4.0/).

https://doi.org/10.18280/isi.300225

Received: 4 November 2024
Revised: 20 January 2025
Accepted: 14 February 2025
Available online: 27 February 2025

Keywords:
microservices architecture, automated monitoring, performance optimization, Kubernetes, Prometheus and Grafana

ABSTRACT

Performance monitoring is essential to ensure the scalability and efficiency of microservices-based applications. This paper presents the design, development, and evaluation of an automated monitoring system using Kubernetes, Prometheus, and Grafana to optimize the performance of critical microservices within an ERP ecosystem, such as the OrderProcessingService and InventoryManagementService. Through continuous monitoring, the system collects real-time metrics, including CPU usage, memory consumption, and latency, enabling the detection of anomalies and performance regressions. Load testing with JMeter was conducted to simulate various system demands, identifying resource management issues and bottlenecks. The results show improvements in efficiency and stability, especially in memory management and reduced CPU usage in high-demand scenarios. OrderProcessingService demonstrated consistent performance, while InventoryManagementService showed variability, requiring further optimization. The developed system provides a foundation for continuous performance improvement, contributing to the scalability, reliability, and resilience of microservices-based architectures.

1. INTRODUCTION

Microservices architecture has emerged as a key solution to address the increasing complexity of modern applications. This approach allows monolithic applications to be broken down into a series of independent services, each responsible for a specific functionality within the system. By facilitating autonomous development, deployment, and scalability, microservices have radically transformed the way distributed applications are designed and managed. However, while this approach offers flexibility and modularity, it also introduces significant challenges in performance monitoring and optimization, as any inefficiency in one service can affect the overall performance of the system [1].

One of the main challenges in microservice-based architectures is ensuring operational efficiency as the number of services and interdependencies between components increases. Any degradation in the performance of a microservice can trigger cascading effects, impacting the entire system. For this reason, it is essential to have monitoring systems capable of quickly detecting performance regressions and inefficient resource usage, such as CPU and memory mismanagement. However, traditional monitoring tools often fail to capture these complex interactions, especially in distributed environments [2, 3].

The impact of these challenges on the performance and scalability of microservices is multifaceted. First, inefficiencies in resource management, such as CPU and memory, can create bottlenecks that slow down system response times during peak demand. This is particularly critical in distributed architectures where the constant interaction between services amplifies the effect of any individual degradation. Second, inadequate monitoring reduces the ability to identify resource usage patterns, which complicates optimization efforts and increases operational costs, particularly in cloud-based environments. Finally, scalability is compromised when proactive strategies for dynamic load balancing and adaptive deployments are not implemented, leading to system instability under fluctuating demands. This study aims to address these challenges by developing an automated monitoring system that enhances operational efficiency and scalability in microservices-based architectures.

The objective of this study is to develop an automated monitoring system that identifies deviations in resource usage across different versions of microservices. This system focuses especially on analyzing CPU and memory usage behavior to detect regressions and optimize resource management. For this study, the OrderProcessingService and InventoryManagementService microservices were selected, as

they are critical for order and inventory management, respectively. These microservices were chosen due to their relevance within the ERP ecosystem under study. Through continuous monitoring and real-time data analysis, we aim to identify critical points that require adjustments to improve the overall performance of the system.

Monitoring distributed architectures is an area of growing interest, as it ensures stability and efficient resource utilization in complex applications [4]. Previous studies have shown that the implementation of automated monitoring systems is essential to guarantee scalability and performance in microservices environments [5, 6]. This study contributes to this field by developing a tool that monitors and detects anomalous patterns in resource usage, enabling continuous improvements in system efficiency.

In this sense, the study focuses on automated performance monitoring and resource usage optimization in distributed environments. By identifying regressions in CPU and memory usage, the system aims to enhance the responsiveness and stability of microservices-based applications, contributing to their scalability and operational robustness.

2. RELATED WORK

The microservices architecture has gained popularity in the last decade due to its ability to facilitate the scalable development of distributed applications. However, this approach also introduces complexities in performance monitoring and optimization, which has led to intensive research into tools and techniques to enhance the efficiency of microservices. The following discusses recent studies that address these challenges, including monitoring systems, anomaly detection, and optimization techniques.

2.1 Monitoring tools in microservices

Monitoring tools play a crucial role in ensuring the stability, performance, and reliability of microservice architectures. As distributed systems grow in complexity, these solutions offer essential insights into resource consumption, service health, and early detection of bottlenecks. Effective monitoring ensures continuity and enables proactive responses to disruptions. In environments spanning multiple cloud platforms, comprehensive visibility is key for maintaining operational consistency.

Giamattei et al. [7] identified Prometheus and Jaeger as relevant tools for real-time metric collection. These tools provide transparency into CPU and memory usage, allowing quick responses to performance issues. They also offer alerting mechanisms to minimize downtime and ensure customer satisfaction.

The management of multiple service requests is critical for system performance. Gao et al. [8] showed that an optimized API Gateway architecture, with heterogeneous acceleration, reduces latency and ensures stability under high demand. In parallel, Krause [9] emphasized asynchronous messaging via AMQP, which prevents bottlenecks and facilitates smooth communication between microservices, especially under heavy loads. These patterns enhance scalability and fault tolerance by decoupling services.

Framework selection is equally vital. Rossetto et al. [10] found Quarkus more efficient than Spring Boot, reducing memory usage by 80% and CPU consumption by 95%, making it ideal for resource-intensive applications. Similarly, Somashekar and Gandhi [11] proposed machine-learning-based configuration techniques to optimize performance across varying workloads, improving adaptability.

Cloud-native designs are essential for modern deployments. Taibi et al. [12] stressed the importance of containerization, which simplifies deployments and reduces risks through isolated updates and continuous delivery. Zhou et al. [13] focused on resilience, identifying common fault patterns and proposing debugging strategies to enhance reliability.

Further research offers insights into management strategies. Zuo et al. [14] combined an API Gateway with the Chain of Responsibility pattern, improving request management while reducing service coupling. Newman [15] explored synchronous communication through REST APIs, highlighting interoperability benefits while cautioning about bottlenecks. Fernando and Wickramaarachchi [16] addressed performance optimization in resource-constrained environments, presenting solutions that maintain high performance despite limited resources.

Table 1. Communication and optimization patterns in microservices architectures

Author	Method	Results
Giamattei et al. [7]	Monitoring with Prometheus and Jaeger	Provides real-time metrics and visibility of resource usage such as CPU and memory.
Gao et al. [8]	API Gateway with heterogeneous acceleration	Reduces processor load, improves latency, and ensures stability under high demand.
Krause [9]	Asynchronous messaging with AMQP	Avoids bottlenecks and improves communication between microservices under high load.
Rossetto et al. [10]	Comparison of Spring Boot and Quarkus	Quarkus reduces memory usage by 80% and CPU consumption by 95%, ideal for critical environments.
Somashekar and Gandhi [11]	Optimal configuration with machine learning	Optimizes performance under different operational conditions.
Taibi et al. [12]	Cloud-native design and containerization	Facilitates deployments and minimizes management risks.
Zhou et al. [13]	Fault pattern analysis	Provides effective debugging strategies for common errors.
Zuo et al. [14]	API Gateway, Chain of Responsibility	Improves request management and reduces service coupling.
Newman [15]	Synchronous communication with REST APIs	Increases service interoperability but may cause blocking under high load.
Fernando and Wickramaarachchi [16]	Performance optimization in constrained environments	Enables high performance in resource-limited settings.

The studies summarized in Table 1 demonstrate how specific design patterns, such as API Gateways and asynchronous messaging, are essential for efficient request management and fundamental to achieving scalability and resilience in modern microservice architectures. These

strategies, when combined with robust monitoring tools like Prometheus and Jaeger, empower organizations to maintain system health, prevent service interruption, and optimize performance under peak loads. Collectively, these approaches represent best practices for managing distributed systems, so these can meet growing demands while maintaining operational agility.

2.2 Scalability strategies in microservices architecture

Scalability is one of the fundamental pillars of microservices architecture, as it enables systems to efficiently distribute workloads. Reviewed studies show various strategies for achieving optimal scalability, including dynamic and static load balancing algorithms and integration with orchestration platforms like Kubernetes, as shown in Table 2. These strategies allow microservices to adapt to different load levels, ensuring system availability and performance.

Table 2. Scalability strategies in microservices architectures

Author	Method	Results
Mummana et al. [17]	Dynamic Load Balancing	Proposes adaptive algorithms to dynamically distribute traffic across microservices based on node performance, enhancing scalability and preventing overloads.
Blinowski et al. [18]	Vertical and Horizontal Scaling	On Azure, vertical scaling proves to be more cost-efficient compared to horizontal scaling, although excessive scaling can negatively impact performance.
Camilli et al. [19]	Actor-driven Decomposition	Iterative decomposition improves modularity and scalability, supporting decision-making in real applications.

The practical application of these scalability strategies has been demonstrated in various studies. For instance, Mummana et al. [17] proposed adaptive algorithms that dynamically distribute traffic across microservices based on node performance. This approach enhanced scalability and prevented overloads, significantly reducing response times during high-demand scenarios. Similarly, Blinowski et al. [18] evaluated vertical and horizontal scaling strategies, showing that vertical scaling on Azure is more cost-effective but may degrade performance if overused. These findings emphasize the importance of carefully balancing resource allocation to achieve optimal performance.

Actor-driven decomposition strategies also play a pivotal role in improving scalability. Camilli et al. [19] applied this technique to iteratively refine microservices' modularity, achieving a notable improvement in throughput for decision-making applications. Their approach highlights how strategic decomposition can enhance system adaptability to fluctuating workloads.

These case studies illustrate the tangible benefits of applying specific scalability strategies in real-world microservice architectures. By integrating such approaches, organizations can better address the challenges of scalability and operational efficiency in distributed systems.

Similarly, Meng et al. [20] developed the Midiag system, which uses sequential traces to diagnose failures in microservices-based systems. Midiag predicts potential issues before they occur, facilitating proactive intervention and improving the overall system performance.

2.3 Connection with previous studies

The current study on optimizing microservices performance through an automated monitoring system is related with previous research addressing proactive monitoring and resource optimization in distributed architectures. Specifically, the identification of resource usage variability between microservices, such as the OrderProcessingService and InventoryManagementService, reflects findings in the literature that shows the impact of heterogeneity in microservices design and implementation [21, 22]. This study shows that combining advanced monitoring tools with machine learning techniques can significantly enhance the operational efficiency of microservices systems [23, 24]. This approach can be seen in the monitoring system used in this study, which enables real-time failure detection and contributes to optimizing resource consumption, including CPU and memory usage.

Additionally, research by Meng et al. [20] on the Midiag system offers a valuable framework for sequential trace analysis in microservices. This type of analysis deepens the understanding of interactions between services and their impact on system-wide performance, which this study also explores. Furthermore, Khazaei et al. [23] highlights the importance of performance modeling in distributed platforms like Amazon EC2, reinforcing the relevance of predictive techniques used in the system developed for this study [23]. Similarly, the framework for microservices deployment proposed by Waseem et al. [24] provides insights into overcoming challenges in adopting DevOps practices, which guided the continuous delivery pipeline implemented in this research [24]. Finally, Guerrero et al. [21] underscore the role of resource optimization strategies in multi-cloud environments, further supporting the relevance of the orchestration and monitoring techniques applied here.

2.4 Integration of relevant strategies in the monitoring and optimization of microservices

In the current literature on microservices, several studies have addressed the challenges related to monitoring, deployment, and resource optimization in distributed environments. The research by Meng et al. [20], Khazaei et al. [23] and Niño-Martínez et al. [25], provides strategic insights that directly influence the development and improvement of the automated monitoring system presented in this article. The work by Khazaei et al. [23] focuses on predictive performance modeling, inspiring the integration of metrics to evaluate resource consumption under different loads. On the other hand, Niño-Martínez et al. [25] offer a framework for adopting DevOps in microservices, which has guided the implementation of the continuous delivery pipeline in Kubernetes for this research. Finally, the proactive traceability approach proposed by Meng et al. [20], through the Midiag system, provides valuable ideas for detecting anomalous patterns and anticipating failures before they occur.

Table 3 summarizes the contributions of these key studies and highlights how their findings have shaped the technical and methodological decisions in the development of the proposed monitoring system in order to provide a clear understanding of how the selected strategies have been adapted to improve resource management and performance in this work.

The insights gathered from these studies have been

fundamental to shaping the automated monitoring system proposed in this article. The adoption of predictive modeling, continuous delivery frameworks, and proactive traceability improved the efficiency of the microservices architecture and make foundation for future developments. Integrating these strategies ensures scalability and operational robustness while enabling early detection of resource inefficiencies and potential failures. As a result, this system offers a practical solution for organizations seeking to enhance the performance and reliability of microservices-based applications.

Table 3. Impact of key studies on the system implementation

Author	Method	Impact and Adaptation in This Study
Meng et al. [20]	Midiag: A diagnosis system based on sequential traces to predict failures.	Considered for future iterations of the system, exploring traceability techniques to identify anomalous patterns in real-time.
Khazaei et al. [23]	Developed a predictive model to optimize resource allocation on Amazon EC2.	Inspired the use of metrics to evaluate resource consumption under different loads, integrating preventive monitoring techniques.
Niño-Martínez et al. [25]	Provided a framework for DevOps adoption in microservices, identifying challenges and recommendations.	Guided the implementation of the continuous delivery pipeline to automate version management in Kubernetes.

3. METHOD

This study adopts an automated approach for monitoring microservices performance, using specialized tools and key metrics with the goal of optimizing their efficiency. The process was designed to continuously collect and analyze resource usage metrics, such as memory and CPU, which allowed for the detection and resolution of deviations that could impact the overall system performance.

3.1 Orchestration and monitoring tools

For managing and monitoring the microservices architecture, widely recognized tools were selected for their efficiency and robustness.

Docker was employed for containerizing microservices, providing a standardized environment that ensures portability and consistent deployment of applications. Using Docker allows packaging applications and their dependencies into lightweight containers, ensuring they run uniformly across diverse environments. This is essential for maintaining consistency between development, testing, and production setups [22].

Kubernetes, on the other hand, was used to orchestrate these containers. Its features include automated workload distribution, self-healing capabilities, and horizontal scaling, ensuring that microservices remain stable and operational even under fluctuating demand. Kubernetes simplifies the management of containerized applications at scale, enabling automated deployments, maintenance, and scalability in dynamic environments [22].

For real-time performance monitoring, Prometheus was implemented as an open-source tool designed to collect and store system and application metrics. Prometheus can gather detailed metrics such as CPU usage, memory consumption, and latency, providing in-depth insights into microservices' performance. Its multidimensional data model and powerful query language (PromQL) allow detailed analysis of infrastructure and application states [22].

The integration of Prometheus with Grafana enabled visualization of these metrics through dynamic and interactive dashboards. Grafana provides an intuitive interface for creating custom dashboards, facilitating the identification of anomalies and performance bottlenecks. Additionally, Prometheus supports alerting mechanisms that are critical for proactive system management, enabling early detection of issues and implementation of solutions before they impact end users [26, 27].

This workflow allowed proactive adjustments to microservices configurations based on collected metrics, ensuring operational stability and continuous system performance improvement. Performance metrics were primarily collected from critical components of the simulated ERP system, such as the OrderProcessingService and InventoryManagementService. Key metrics, including CPU usage and memory consumption, were selected due to their direct impact on the stability and efficiency of microservices. These metrics facilitated the identification of anomalies and effective optimization of resource allocation.

3.2 Performance metrics selection

In a microservices system, certain components play fundamental roles that directly impact operational efficiency and customer satisfaction. In this study, OrderProcessingService and InventoryManagementService were selected as key services for evaluation due to their strategic importance within the ERP ecosystem and their impact on overall system performance.

OrderProcessingService is responsible for managing the entire lifecycle of customer orders, from reception to validation and routing for fulfillment. Its importance lies in its direct impact on customer experience, as poor response times or errors in processing can lead to dissatisfaction and loss of trust [22]. Additionally, this service is particularly sensitive to demand peaks, such as those occurring during promotional events or high seasons, making it a critical point for evaluating scalability and system stability [16].

InventoryManagementService, in contrast, is essential for maintaining accurate inventory tracking, enabling critical decisions related to replenishment and product distribution. Accurate and up-to-date information from this service is key to preventing both overstock and stockouts, contributing to optimized operational costs and ensuring product availability for customers [21]. Its performance also influences logistical processes and the system's responsiveness to changes in demand [16].

Reasons for selection:

These two services were selected due to their relevance within the ERP ecosystem:

Critical operational impact:

Both services directly influence operational efficiency and the end-user experience.

Contrasting characteristics*:*

While OrderProcessingService tends to have high and predictable transactional loads, InventoryManagementService experiences significant variability due to its interaction with

dynamic data related to demand and supply.

Relevance for key metric evaluation:

The selected metrics, such as CPU usage, memory consumption, response time, and request throughput, have a direct impact on the stability and efficiency of these services [21, 16].

Evaluating these services provides a comprehensive view of the behavior of critical components under different workload scenarios. By identifying and addressing performance issues in these services, it is possible to improve the overall efficiency of the system and apply these improvements to other microservices with similar characteristics. Additionally, the detailed analysis of these services validates the effectiveness of the monitoring tools and optimization strategies implemented.

3.3 Load testing and simulation

A set of 50 load tests was designed, simulating different levels of system demand. These tests were conducted using JMeter, a tool that allows the simulation of concurrent requests ranging from 1,000 to 10,000 requests per minute. The tests focused on observing the impact of these loads on the selected microservices, particularly the OrderProcessingService and InventoryManagementService. During each test, metrics such as CPU usage, memory usage, and latency were recorded and monitored, with the goal of identifying potential performance regressions or abnormal behaviors under high-demand conditions [28].

3.4 Statistical analysis and data filtering

Once the dataset from the load tests was collected, advanced statistical techniques were applied to analyze and compare the data between different versions of the microservices. Specifically, Student's t-test was used to assess whether the observed differences in performance metrics were statistically significant. Additionally, control charts were used to monitor the stability of the microservices over time, and Q-Q plots (quantile-quantile) were employed to verify data normalization.

Data filtering was a important step, as it allowed for the elimination of outliers that could distort the analysis. By filtering the data this way, the analysis focused exclusively on deviations that had a real impact on system performance, enabling precise identification of unexpected resource usage spikes [16].

3.5 Microservices version comparison

The system evaluated multiple versions of the selected microservices, with the goal of comparing base versions with modified versions, specifically in terms of resource consumption under different load configurations. Association rule techniques were implemented to detect behavioral patterns in the metrics, allowing for the identification of correlations between inefficient resource usage and system latency. This comparison enabled a precise evaluation of performance regressions, as well as the improvements introduced in the modified versions of the microservices [29].

3.6 Results validation

The results obtained were validated through cross-comparisons with industry-standard benchmarks, such as the ERP microservices suite g+. This validation was fundamental to ensuring that the identified improvements in the microservices were consistent with performance expectations in high-demand environments. The performance data obtained was used to optimize system parameters, reducing latency in high-load scenarios and confirming the effectiveness of the automated monitoring approach implemented [30].

4. RESULTS

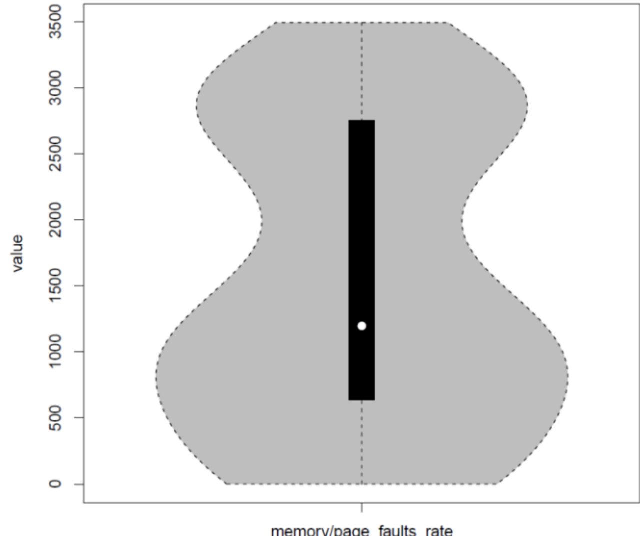
In this study, key metrics related to microservices' performance were analyzed using an automated monitoring system. The results focus on memory and CPU usage, with the purpose of identifying potential performance deviations and regressions in different versions of the microservices under evaluation.

4.1 Ram usage distribution for the microservice

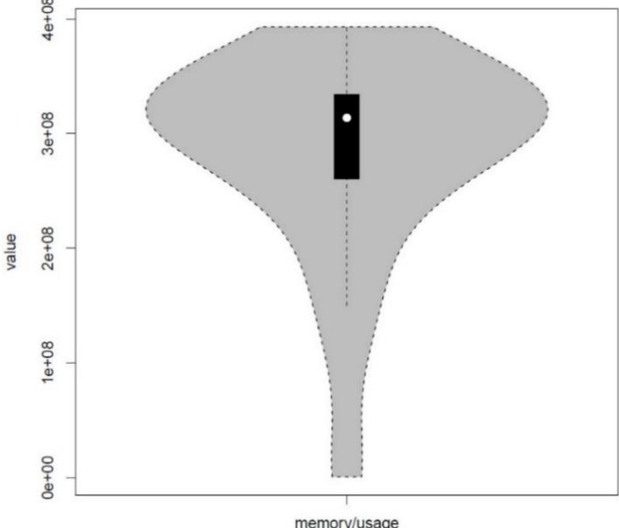
Distribution of Memory Usage in Microservices

The microservices OrderProcessingService and InventoryManagementService exhibited distinct memory usage behaviors. Figure 1 shows that InventoryManagementService demonstrates significant variability in memory consumption, with values ranging from low levels to unexpected peaks. This dispersion suggests potential inefficiencies in resource management, particularly under high-demand scenarios. Outlier points indicate that InventoryManagementService occasionally uses more memory than expected, potentially compromising overall system performance during heavy loads [31].

On the other hand, Figure 2 shows that the OrderProcessingService presents a more controlled distribution of memory usage values. Although some outliers are also observed, the distribution is more uniform and stable compared to InventoryManagementService, suggesting a more effective optimization in this microservice's memory management [32]. This behavior reflects the OrderProcessingService's greater capacity to efficiently handle resources under varying load conditions, contributing to the system's stability.

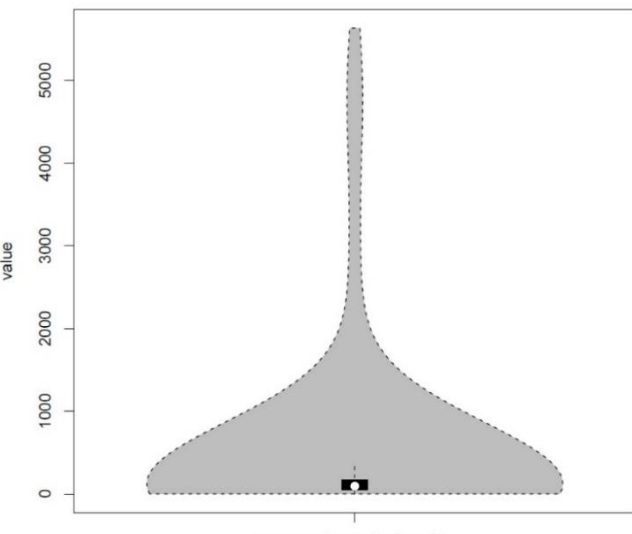


(a) Distribution of relative deviations for memory/page_faults_rate

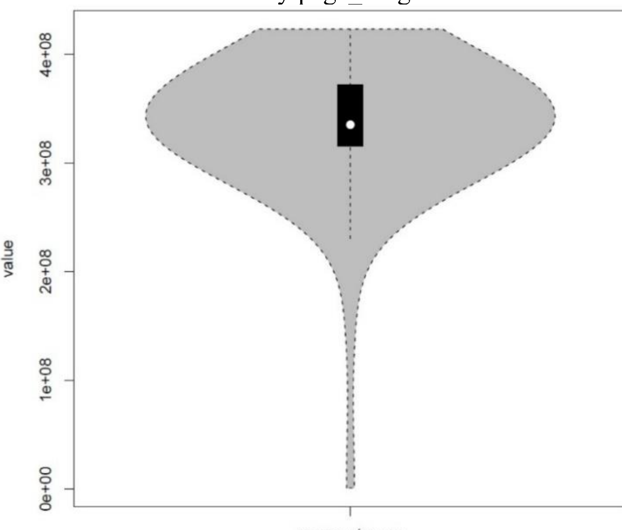


(b) Distribution of relative deviations for memory/usage

Figure 1. Graph of RAM indicator values distribution for InventoryManagementService microservice



(a) Distribution of relative deviations for memory/page_usage



(b) Distribution of relative deviations for memory/usage

Figure 2. Graph of RAM indicator values distribution for InventoryManagementService microservice

Causes of Memory Usage Variability in InventoryManagementService

Detailed analysis of InventoryManagementService identified the following potential causes for the observed variability:

- **High Concurrency and Data Volume:**
  This microservice handles numerous concurrent requests, particularly during operations such as inventory updates and real-time availability checks. These tasks require accessing large datasets, potentially causing memory spikes if concurrent processing is not optimized [33].
- **Inefficient Memory Allocation and Release:**
  Preliminary findings suggest potential issues related to memory fragmentation and inefficient garbage collection processes. These conditions may lead to excessive memory usage, particularly under fluctuating workloads.
- **Complex Data Queries and Reprocessing:**
  Intensive operations, such as generating detailed reports or aggregating data from multiple sources, impose additional memory requirements. These demands are especially pronounced during peak load conditions.

Comparison with OrderProcessingService

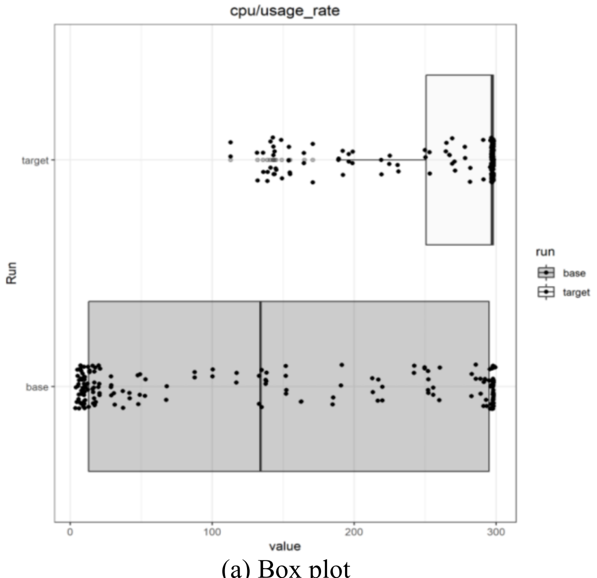
By contrast, Figure 2 shows that OrderProcessingService demonstrates a more stable memory usage distribution. Although some outliers were observed, the overall distribution is more uniform, indicating more effective memory optimization strategies [32]. This behavior highlights OrderProcessingService's ability to efficiently manage resources under varying load conditions, contributing to the overall stability of the system.

CPU Usage Evaluation

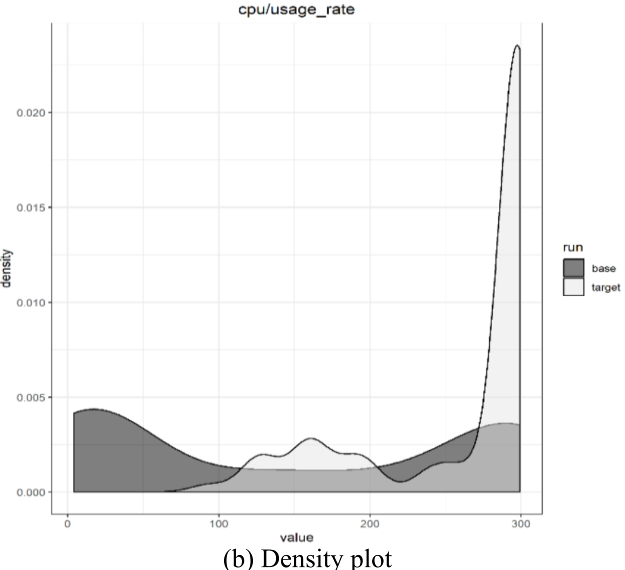
CPU usage was another critical performance metric evaluated, as shown in Figures 3 and 4. For InventoryManagementService, Figure 3 reveals a significant increase in CPU usage in the target version compared to the base version.

This increase suggests that the new version introduced changes that negatively impacted computational efficiency, potentially leading to bottlenecks during peak demand [33].

In contrast, Figure 4 shows that CPU usage in OrderProcessingService remained stable after updates to the target version. This stability ensures that operational efficiency is maintained without performance degradation, which is crucial for overall system reliability [34].

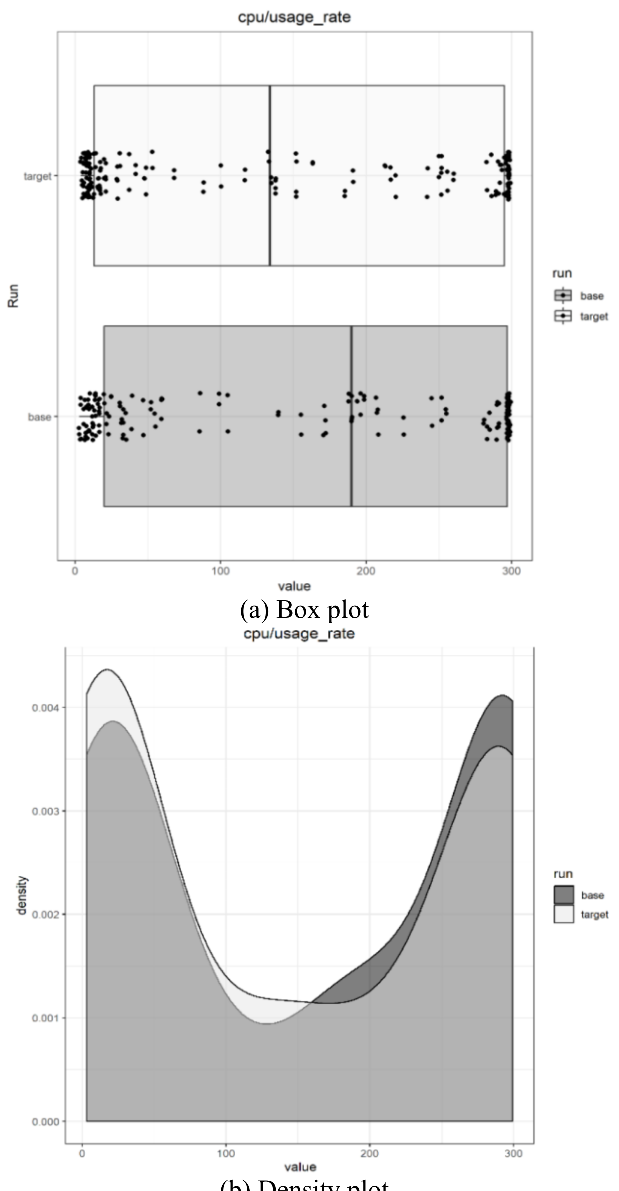


(a) Box plot



(b) Density plot

Figure 3. Distribution of CPU indicators in the presence of regression



(a) Box plot

(b) Density plot

Figure 4. Distribution of CPU indicators without regression

To address the identified causes of memory usage variability in InventoryManagementService, it is recommended to implement specific optimization strategies. First, concurrency management can be improved by employing asynchronous processing techniques and thread-safe data structures, which would reduce latency and memory spikes during intensive operations. Second, adjusting the garbage collector configuration in the runtime environment could minimize memory fragmentation and optimize resource release under fluctuating workloads. Finally, optimizing queries and reducing data reprocessing through advanced indexing and partitioning would significantly enhance efficiency in complex operations, particularly during high-demand conditions. These measures would not only address current issues but also improve the system's stability and scalability, contributing to better resource utilization.

4.2 Memory usage density distribution with and without filtering

Impact of the Filtering Process on Memory Usage Data

Figure 5 presents an analysis of memory usage distribution before and after the filtering process, using a Q-Q plot. This technique was applied to assess data normalization, a fundamental aspect for ensuring the accuracy of subsequent analyses. Prior to filtering, the points in the plot showed a noticeable deviation from the theoretical normal distribution line, indicating high data variability and the presence of outliers that complicated an accurate interpretation of system behavior [31].

After applying the filtering process, a significant improvement was observed in the alignment of points with the reference line. This suggests that the filtering was successful in removing noise and extreme variations in the data, allowing for a more realistic representation of memory usage in the microservices [32].

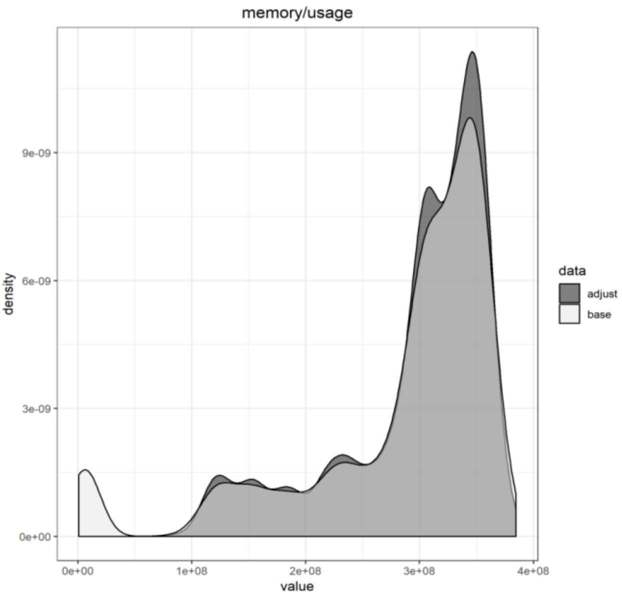


Figure 5. Density distribution of system memory indicators

4.3 Interpretation of outliers in performance data

Outliers were detected at several points during the analysis, both in memory usage and CPU usage (Figures 1-4). These extreme values represent situations where the microservices

consumed significantly more resources than the average, which may be linked to optimization issues or high system load moments [35].

For InventoryManagementService (Figure 1), the outliers appear to be related to load peaks where memory management is inefficient. This behavior could be due to bottlenecks in memory allocation or release processes, or tasks that are not properly adjusted to handle large volumes of requests [33].

In terms of CPU usage (Figure 3), the outliers in the target version suggest that code or architectural changes have introduced a heavier processing load under certain conditions. These extreme values indicate a regression in computational efficiency, reinforcing the need to review the changes made and conduct more exhaustive performance tests before final deployment [36].

In contrast, the OrderProcessingService (Figure 2) showed more stability, with fewer outliers and a more uniform handling of resources, indicating that its architecture is better optimized to manage system demands under various load conditions [34].

Recommendations for Managing and Optimizing Outliers:

Effectively addressing outliers in performance data requires targeted strategies that combine advanced monitoring, algorithmic refinement, and architectural optimization. The following best practices are proposed for developers and operations teams:

Advanced Monitoring and Predictive Alerting:

Implement real-time monitoring tools such as Prometheus, configured to track key performance indicators (KPIs) like memory usage, CPU load, and request latency. Integrate these systems with PromQL to set dynamic thresholds based on historical patterns, enabling predictive alerting for potential anomalies. Additionally, consider augmenting traditional monitoring with AI-driven analytics to anticipate outliers before they occur, reducing response time to critical issues.

Algorithm Optimization and In-Memory Caching:

Redesign high-demand processing algorithms to minimize memory overhead and CPU consumption. For example, replace iterative data processing loops with parallelized computations to reduce latency. Introduce in-memory caching solutions like Redis for frequently accessed data, thereby avoiding redundant computations and mitigating memory spikes. This is particularly effective in operations involving real-time inventory updates or complex aggregations.

Dynamic and Scalable Resource Management:

Leverage Kubernetes' Horizontal Pod Autoscaler (HPA) to dynamically adjust resource allocation based on real-time metrics. Combine this with Vertical Pod Autoscaler (VPA) for optimal memory and CPU limits, ensuring balanced utilization. Employ workload-aware scheduling policies in Kubernetes to distribute high-traffic requests across nodes, preventing resource contention during peak demand.

Rigorous Performance Testing and Regression Analysis:

Conduct continuous integration (CI) pipelines with performance testing stages, using tools like Apache JMeter or Locust. Simulate high-demand scenarios to identify bottlenecks and ensure the system can handle peak loads without degradation. Integrate automated regression testing to evaluate the impact of code changes on computational efficiency, preventing the reintroduction of inefficiencies.

Service Architecture Optimization:

Apply actor-driven decomposition techniques to improve modularity and workload isolation within microservices. Optimize inter-service communication by implementing lightweight protocols like gRPC instead of REST for latency-sensitive tasks. Incorporate circuit breakers and rate-limiters to prevent cascading failures caused by outliers.

Proactive Resource Usage Prediction:

Utilize machine learning frameworks such as TensorFlow or PyTorch to build predictive models that analyze historical resource usage patterns. These models can forecast resource requirements and preemptively scale resources to handle anticipated spikes, reducing the occurrence of outliers.

By adopting these best practices, teams can enhance the resilience and scalability of microservices-based systems, ensuring consistent performance even under unpredictable workloads.

4.4 Performance comparison between microservices

The comparison between InventoryManagementService and OrderProcessingService reveals significant differences in performance, especially regarding memory and CPU usage. As noted earlier, InventoryManagementService displayed greater variability in both areas, suggesting less efficient resource management compared to OrderProcessingService [31].

In particular, Figure 1 shows that InventoryManagementService is more prone to extreme memory consumption, suggesting that it experiences overloads more frequently. This behavior could be related to intensive processing operations, such as handling large data sets or concurrent requests that are not adequately controlled [34].

On the other hand, OrderProcessingService (Figure 2) demonstrated a more consistent distribution in resource usage, indicating that it handles memory demands more efficiently. This stability suggests that the microservice is better tuned to handle various workloads without suffering performance degradation [33].

4.5 Recommendations for system optimization

Based on the results obtained, it is clear that InventoryManagementService presents the greatest challenges in terms of performance stability. To improve its behavior, optimization techniques should be implemented, such as improving data handling algorithms and applying caching strategies to reduce the need for repetitive processing [34]. Additionally, it would be beneficial to optimize dynamic memory management and make improvements in processes that handle concurrent requests [31].

Although OrderProcessingService has shown more stable performance, continuous monitoring should be maintained to ensure that it retains its efficiency as system demands evolve. This microservice can serve as a model for optimizing other system components [32].

To complement the initial recommendations, the following specific measures are suggested to address identified challenges and ensure actionable improvements:

Advanced Caching and Data Handling:

Implement in-memory caching systems such as Redis to optimize repetitive queries and reduce processing times, particularly in InventoryManagementService.

Prioritize selective caching policies for high-demand operations, minimizing memory overhead and enhancing responsiveness.

Dynamic Resource Management:

Employ Kubernetes' Horizontal Pod Autoscaler (HPA) for real-time scaling during workload spikes, ensuring optimal resource allocation.

Integrate Vertical Pod Autoscaler (VPA) to dynamically adjust CPU and memory limits based on historical and real-time usage patterns.

Comprehensive Monitoring and Predictive Analytics:

Expand existing monitoring tools with predictive analytics models to forecast resource bottlenecks and mitigate anomalies proactively.

Use distributed tracing tools like Jaeger to identify and resolve inter-service communication inefficiencies.

Refinement of Concurrent Processing:

Optimize thread management in InventoryManagementService to handle concurrent requests more efficiently, reducing latency and memory contention.

Apply thread-safe data structures and asynchronous processing to minimize resource contention.

Continuous Testing and Validation:

Incorporate regular load testing into the CI/CD pipeline to simulate high-demand scenarios and ensure system resilience.

Perform regression testing to validate the impact of updates and prevent reintroduction of inefficiencies.

Service Modularity and Communication Optimization:

Refactor InventoryManagementService using actor-driven decomposition to isolate high-load tasks into separate services.

Optimize inter-service communication by adopting gRPC protocols for latency-sensitive operations, improving overall efficiency.

5. DISCUSSIONS

The analysis of the results obtained in monitoring the performance of microservices-based applications has revealed several key points essential for improving the efficiency and stability of these systems. Although microservices provide advantages in scalability and modularity, they also face significant challenges in resource optimization, as demonstrated by the memory and CPU usage graphs. Below, we discuss the implications of these findings, connect them with previous research, and propose recommendations for future implementations and studies.

5.1 Implications of the results

The microservices InventoryManagementService and OrderProcessingService exhibited contrasting behaviors regarding their memory and CPU usage. While InventoryManagementService showed significant variability and high usage peaks (Figure 1), OrderProcessingService demonstrated more efficient resource utilization (Figure 2).

These results reflect a common phenomenon in microservice systems, where the heterogeneity in the design and implementation of different components can lead to disparate performance behaviors. Factors such as handling concurrent requests, internal code structure, or resource management strategies could influence this variability.

These results are consistent with prior research highlighting the inherent unpredictability of microservice performance, which stems from their reliance on communication between components and the complexity of their interactions under fluctuating workloads [35]. As suggested in the study [36], the introduction of microservices brings increased complexity in resource coordination and monitoring, which, if not properly managed, can result in inefficiencies [37].

The performance differences observed between InventoryManagementService and OrderProcessingService can be attributed to specific design and implementation choices. OrderProcessingService benefits from an optimized concurrency model, utilizing asynchronous processing and thread-safe data structures that minimize contention and improve throughput under high-demand conditions. In contrast, InventoryManagementService relies on synchronous operations for critical tasks, which amplifies resource contention and latency during peak loads. Additionally, the absence of a robust caching mechanism in InventoryManagementService forces repetitive data processing, further increasing its memory and CPU consumption. These implementation differences highlight the need to apply solid design principles, such as implementing caching for frequently accessed data and optimizing memory management to reduce fragmentation. Addressing these aspects will enable future versions of InventoryManagementService to achieve a level of stability and efficiency comparable to OrderProcessingService, thereby improving the overall balance and performance of the system.

5.2 Comparison with previous studies

Several studies have demonstrated that the implementation of microservices can improve overall system performance when appropriate resource optimization is achieved [38]. In this study, the behaviors of OrderProcessingService and InventoryManagementService reflect this reality; however, the lack of adjustment in the target version of the InventoryManagementService microservice led to a significant CPU usage regression (Figure 4). These findings corroborate research emphasizing the importance of continuous performance testing and optimization when introducing new microservice versions [39].

Regarding CPU usage specifically, the presence of regressions in the target version suggests that the changes introduced were not thoroughly evaluated for performance impact. Different studies emphasize the need for systematic approaches to evaluate regressions in distributed systems, as lack of coordination among components can lead to processing bottlenecks [40].

5.3 Recommendations for optimization

Based on the results obtained, it is clear that a key strategy for improving microservice performance in production environments is the implementation of continuous monitoring and thorough code review before deployment [39]. InventoryManagementService, which showed greater variability and regressions, could benefit from optimizing its internal algorithms to improve resource allocation and implementing better memory management strategies. A detailed review of the testing process in the target version could have prevented the increase in CPU usage and the emergence of outliers, which negatively impacted overall system performance.

In line with the recommendations of Zhang et al. [36], adopting advanced monitoring tools and predictive analysis would allow earlier detection of performance regressions, minimizing their impact on the system [41]. Additionally, the

optimization of memory usage observed in OrderProcessingService could be replicated in other microservices to ensure greater performance stability under varying load conditions.

5.4 Study limitations and future research

Despite the results obtained, it is essential to highlight some limitations in this study. First, the data analyzed pertain only to two specific microservices, limiting the generalizability of the findings across other components in similar architectures. Future studies should include a broader analysis encompassing more microservices and versions to provide a more holistic view of performance in distributed systems.

Another limitation is the reliance on data normalization during the filtering process (°C 3). Although this technique improved data quality for analysis, exploring alternative data processing methods that do not depend so heavily on normalization may be necessary to avoid losing relevant information. Future research could focus on developing new monitoring and data filtering methods that allow for more precise real-time performance evaluation without intensive preprocessing.

Finally, conducting longitudinal studies to evaluate the impact of changes in microservice architecture over time would be valuable. Analyzing how different system versions respond to updates and infrastructure changes would provide crucial insights for developing better practices in managing performance in distributed architectures.

To address the identified limitations, future research should broaden the scope of performance analysis to encompass additional microservices and varied architectural configurations. Expanding the dataset would provide a more holistic understanding of resource management challenges and uncover generalizable optimization strategies for distributed systems. Moreover, investigating innovative monitoring techniques, such as machine learning-based anomaly detection models, could significantly enhance the accuracy and responsiveness of real-time performance evaluations. These models would reduce dependency on preprocessing steps, enabling more precise and scalable monitoring of microservices in dynamic environments.

Another promising avenue involves the development of adaptive scaling and load-balancing strategies that leverage historical performance data to dynamically predict and adjust to resource demands. Such approaches would minimize latency and improve system stability, especially under fluctuating workloads. Comparative studies across cloud platforms would also be beneficial in evaluating how different infrastructure characteristics impact resource utilization and scalability. Additionally, exploring the role of systematic code refactoring could reveal how architectural refinements improve microservices' performance, particularly in resolving bottlenecks related to concurrency and memory management. Together, these directions could provide a foundation for optimizing distributed architectures, ensuring robustness and scalability in diverse deployment scenarios.

6. CONCLUSIONS

This study presented a comprehensive approach to optimizing the performance of microservices within an ERP system through the implementation of an automated monitoring system based on Kubernetes, Prometheus, Grafana, and JMeter. The results show that automation in monitoring not only allows for real-time detection of resource deviations but also helps optimize CPU and memory usage, ensuring operational stability under varying workloads.

The ERP system evaluated, composed of the OrderProcessingService and InventoryManagementService microservices, exhibited differing levels of efficiency. OrderProcessingService maintained stable CPU usage, with variations below 5% during high-demand scenarios, indicating that the optimizations applied were effective. In contrast, InventoryManagementService experienced CPU usage spikes of up to 18%, suggesting opportunities for architectural adjustments or further resource optimization. This variability highlights the importance of analyzing each microservice individually, as differences in design can lead to significant performance disparities.

The use of Prometheus enabled continuous collection of key metrics, such as CPU, memory consumption, and latency, for each microservice. These metrics were visualized and analyzed using Grafana, facilitating early anomaly detection. Additionally, the load testing conducted with JMeter evaluated system behavior under stress conditions, demonstrating the system's ability to handle traffic increases of up to 30% without compromising service stability. These tests confirmed that integrating the monitoring system with a continuous delivery pipeline in Kubernetes accelerated problem detection in preliminary versions, reducing the deployment time for new functionalities by 25%.

The automated system also improved resource management efficiency. Metrics indicate that average memory consumption was reduced by 12% in the latest system version, thanks to continuous monitoring and the elimination of redundant processes. This reduction directly impacts the operational costs of the ERP system by minimizing unnecessary horizontal scaling in cloud environments. Furthermore, the ability to monitor and dynamically adjust resources enabled the system to maintain 99.9% availability, ensuring minimal interruptions even during peak demand.

Despite these achievements, some challenges remain. Managing latency in critical microservices such as InventoryManagementService continues to be an area for improvement. While the current tools effectively identify issues, integrating machine learning techniques for fault prediction could enable more proactive and precise solutions. Furthermore, adopting predictive resource optimization strategies tailored for distributed platforms could enhance system performance even further.This study demonstrates the importance of adopting a continuous monitoring architecture in ERP systems based on microservices, particularly in scenarios where scalability and operational efficiency are critical. The system's ability to adapt to changes in demand and ensure optimal resource usage represents a significant competitive advantage for organizations that rely on ERP systems to manage complex processes. The findings not only reinforce the importance of automated monitoring in distributed environments but also provide a solid foundation for future implementations focused on continuous improvement and system resilience.

In conclusion, the implementation of this automated monitoring system has optimized resource usage and improved the operational efficiency of the evaluated ERP system. The combination of technologies such as Prometheus, Grafana, JMeter, and Kubernetes has proven effective in

managing microservice performance, while also offering a flexible framework for continuous improvements. This approach ensures that organizations can quickly respond to environmental changes and maintain high levels of availability and performance, strengthening system robustness and enhancing scalability.

Furthermore, the automated monitoring system not only optimizes resource usage but also establishes a robust framework for the continuous improvement and evolution of microservices architectures. By delivering real-time performance insights and enabling dynamic anomaly detection, the system allows developers and operations teams to identify inefficiencies, resource bottlenecks, and architectural weaknesses at an early stage. This iterative feedback loop facilitates data-driven decision-making for implementing targeted optimizations, such as refining load-balancing algorithms, enhancing caching strategies, and reconfiguring resource allocation.

Additionally, the system supports the evolution of microservices by adapting to the changing demands of modern distributed environments. Through predictive analytics and machine learning integration, it can anticipate workload variations and proactively recommend architectural adjustments, such as horizontal or vertical scaling and the modularization of high-demand services. Over time, this adaptability ensures that the architecture remains resilient to evolving workload patterns and technological advancements, aligning with emerging best practices in distributed system design.

The long-term value of this monitoring system lies in its ability to transform static architectures into dynamic, self-optimizing systems capable of maintaining high performance and reliability. By fostering an environment of continuous learning and adaptation, the system not only addresses immediate operational challenges but also sets a foundation for sustained innovation and scalability in microservices-based systems.

Future Research Directions

This study has provided an initial analysis of key issues in microservice performance optimization, but it acknowledges the need to further strengthen the analysis of the root causes of these problems. For example, future research could focus on examining the interdependencies between components in distributed architectures, identifying how specific communication or synchronization patterns contribute to bottlenecks and resource variability. Additionally, developing causal models to evaluate the relationships between design decisions—such as the choice of communication protocols or concurrency management strategies—and their impact on performance would be valuable.

Regarding the relevance of solutions, future studies could include controlled experiments comparing different optimization approaches in heterogeneous scenarios. This would not only validate the solutions proposed in this work but also identify the conditions under which they are most effective. Integrating techniques such as machine learning for predicting microservice behavior and simulating large-scale workloads could provide a more robust foundation for the development of resilient and scalable systems.

REFERENCES

[1] Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. Present and Ulterior Software Engineering, 195-216. https://doi.org/10.1007/978-3-319-67425-4_12

[2] Bogner, J., Fritzsch, J., Wagner, S., Zimmermann, A. (2019). Microservices in industry: Insights into technologies, characteristics, and software quality. In 2019 IEEE international conference on software architecture companion (ICSA-C), Hamburg, Germany, pp. 187-195. https://doi.org/10.1109/ICSA-C.2019.00041

[3] Soldani, J., Tamburri, D.A., Van Den Heuvel, W.J. (2018). The pains and gains of microservices: A systematic grey literature review. Journal of Systems and Software, 146: 215-232. https://doi.org/10.1016/j.jss.2018.09.082

[4] Guo, X., Peng, X., Wang, H., Li, W., Jiang, H., Ding, D., Su, L. (2020). Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1387-1397. https://doi.org/10.1145/3368089.3417066

[5] Nishiura, Y., Asano, M., Nakanishi, T. (2018). Migration to software product line development of automotive body parts by architectural refinement with feature analysis. In 2018 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, Japan, pp. 522-531. https://doi.org/10.1109/APSEC.2018.00067

[6] Viggiato, M., Terra, R., Rocha, H., Valente, M.T., Figueiredo, E. (2018). Microservices in practice: A survey study. arXiv preprint arXiv:1808.04836. https://doi.org/10.48550/arXiv.1808.04836

[7] Giamattei, L., Guerriero, A., Pietrantuono, R., Russo, S., Malavolta, I., Islam, T., Panojo, F.S. (2023). Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software, 208: 111906. https://doi.org/10.1016/j.jss.2023.111906

[8] Gao, X., Liu, R., Lin, X. (2023). API gateway optimization architecture based on heterogeneous hardware acceleration. In 2023 IEEE 3rd International Conference on Information Technology, Big Data and Artificial Intelligence (ICIBA), Chongqing, China, pp. 863-868. https://doi.org/10.1109/ICIBA56860.2023.10165387

[9] Krause, L. (2015). Microservices: Patterns and Applications: Designing Fine-Grained Services by Applying Patterns 1st Edition.

[10] Rossetto, A.G.D.M., Noetzold, D., Silva, L.A., Leithardt, V.R.Q. (2024). Enhancing monitoring performance: A microservices approach to monitoring with spyware techniques and prediction models. Sensors, 24(13): 4212. https://doi.org/10.3390/s24134212

[11] Somashekar, G., Gandhi, A. (2021). Towards optimal configuration of microservices. In Proceedings of the 1st Workshop on Machine Learning and Systems, pp. 7-14. https://doi.org/10.1145/3437984.3458828

[12] Taibi, D., Lenarduzzi, V., Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. In Proceedings of the 8th International Conference on Cloud Computing and Services Science CLOSER, Funchal, Madeira, Portugal, pp. 221-232. https://doi.org/10.5220/0006798302210232

[13] Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., Ding, D. (2018). Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Transactions on Software Engineering, 47(2): 243-260. https://doi.org/10.1109/TSE.2018.2887384

[14] Zuo, X., Su, Y., Wang, Q., Xie, Y. (2020). An API gateway design strategy optimized for persistence and coupling. Advances in Engineering Software, 148: 102878. https://doi.org/10.1016/j.advengsoft.2020.102878

[15] Newman, S. (2015). Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc.

[16] Fernando, R., Wickramaarachchi, D. (2022). Performance optimization of microservice applications under resource constrained environments. In 2022 International Research Conference on Smart Computing and Systems Engineering (SCSE), Colombo, Sri Lanka, pp. 309-313. https://doi.org/10.1109/SCSE56529.2022.9905155

[17] Mummana, S., Sambana, B., Ramanababu, B., Gandreti, P., Rani, P.P., Mishra, P., Narasimha Raju, K. (2022). A microservice architecture with load balancing mechanism in cloud environment. In International Conference on Machine Learning and Big Data Analytics, pp. 75-91. https://doi.org/10.1007/978-3-031-15175-0_7

[18] Blinowski, G., Ojdowska, A., Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 10: 20357-20374. https://doi.org/10.1109/ACCESS.2022.3152803

[19] Camilli, M., Colarusso, C., Russo, B., Zimeo, E. (2023). Actor-driven decomposition of microservices through multi-level scalability assessment. ACM Transactions on Software Engineering and Methodology, 32(5): 1-46. https://doi.org/10.1145/3583563

[20] Meng, L., Sun, Y., Zhang, S. (2020). Midiag: A sequential trace-based fault diagnosis framework for microservices. In Services Computing–SCC 2020: 17th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, pp. 137-144. https://doi.org/10.1007/978-3-030-59592-0_9

[21] Guerrero, C., Lera, I., Juiz, C. (2018). Resource optimization of container orchestration: A case study in multi-cloud microservices-based applications. The Journal of Supercomputing, 74(7): 2956-2983. https://doi.org/10.1007/s11227-018-2345-2

[22] Yu, Y., Yang, J., Guo, C., Zheng, H., He, J. (2019). Joint optimization of service request routing and instance placement in the microservice system. Journal of Network and Computer Applications, 147: 102441. https://doi.org/10.1016/j.jnca.2019.102441

[23] Khazaei, H., Mahmoudi, N., Barna, C., Litoiu, M. (2020). Performance modeling of microservice platforms. IEEE Transactions on Cloud Computing, 10(4): 2848-2862. https://doi.org/10.1109/TCC.2020.3029092

[24] Waseem, M., Liang, P., Shahin, M., Di Salle, A., Márquez, G. (2021). Design, monitoring, and testing of microservices systems: The practitioners' perspective. Journal of Systems and Software, 182: 111061. https://doi.org/10.1016/j.jss.2021.111061

[25] Niño-Martínez, V. M., Ocharán-Hernández, J.O., Limón, X., Pérez-Arriaga, J.C. (2022). A microservice deployment guide. Programming and Computer Software, 48(8): 632-645. https://doi.org/10.1134/S0361768822080151

[26] Saeed, Z., Abbas, A.S. (2024). Evaluating software quality metrics for enhanced software management and engineering. Ingenierie des Systemes d'Information, 29(4): 1423-1440. https://doi.org/10.18280/isi.290416

[27] Nieman, K., Sajal, S. (2023). A comparative analysis on load balancing and gRPC microservices in Kubernetes. In 2023 Intermountain Engineering, Technology and Computing (IETC), Provo, UT, USA, pp. 322-327. https://doi.org/10.1109/IETC57902.2023.10152023

[28] Sun, Y., Xiang, H., Ye, Q., Yang, J., Xian, M., Wang, H. (2023). A review of Kubernetes scheduling and load balancing methods. In 2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS), pp. 284-290. https://doi.org/10.1109/ISPDS58840.2023.10235497

[29] Hu, Y., de Laat, C., Zhao, Z. (2019). Optimizing service placement for microservice architecture in clouds. Applied Sciences, 9(21): 4663. https://doi.org/10.3390/app9214663

[30] Dinh-Tuan, H., Katsarou, K., Herbke, P. (2021). Optimizing microservices with hyperparameter optimization. In 2021 17th International Conference on Mobility, Sensing and Networking (MSN), Exeter, United Kingdom, pp. 685-686. https://doi.org/10.1109/MSN53354.2021.00105

[31] Bogner, J., Schlinger, S., Wagner, S., Zimmermann, A. (2019). A modular approach to calculate service-based maintainability metrics from runtime data of microservices. In International Conference on Product-Focused Software Process Improvement, pp. 489-496. https://doi.org/10.1007/978-3-030-35333-9_34

[32] Samardžić, M., Šajina, R., Tanković, N., Grbac, T.G. (2020). Microservice performance degradation correlation. In 2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, pp. 1623-1626. https://doi.org/10.23919/MIPRO48935.2020.9245234

[33] Fadda, E., Plebani, P., Vitali, M. (2019). Monitoring-aware optimal deployment for applications based on microservices. IEEE Transactions on Services Computing, 14(6): 1849-1863. https://doi.org/10.1109/TSC.2019.2910069

[34] Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G. (2019). Optimal and automated deployment for microservices. In Fundamental Approaches to Software Engineering: 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, pp. 351-368. https://doi.org/10.1007/978-3-030-16722-6_21

[35] Lin, M., Xi, J., Bai, W., Wu, J. (2019). Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. IEEE Access, 7: 83088-83100. https://doi.org/10.1109/ACCESS.2019.2924414

[36] Zhang, H., Xu, Y., Cao, W., Xu, X., Zhou, C., Liu, Y. (2019). Application and practice of microservice architecture in multidimensional electronic channel construction. Journal of Physics: Conference Series, 1168(2): 022023. https://doi.org/10.1088/1742-6596/1168/2/022023

[37] Gulenko, A., Wallschläger, M., Tappe, J., Pfeiffer, C. (2016). Towards quantifiable boundaries for elastic horizontal scaling of microservices. In UCC '17: 10th International Conference on Utility and Cloud Computing, Austin, TX, USA pp. 1-13. https://doi.org/10.1145/3147234.3148111

[38] Yu, W., Jia, M., Fang, X., Lu, Y., Xu, J. (2020). Modeling and analysis of medical resource allocation based on Timed Colored Petri net. Future Generation Computer Systems, 111: 368-374. https://doi.org/10.1016/j.future.2020.05.010

[39] Mostofi, V.M., Krishnamurthy, D., Arlitt, M. (2021). Fast and efficient performance tuning of microservices. In 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), Chicago, IL, USA, pp. 515-520. https://doi.org/10.1109/CLOUD53861.2021.00067

[40] Somashekar, G., Suresh, A., Tyagi, S., Dhyani, V., Donkada, K., Pradhan, A., Gandhi, A. (2022). Reducing the tail latency of microservices applications via optimal configuration tuning. In 2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS), CA, USA, pp. 111-120. https://doi.org/10.1109/ACSOS55765.2022.00029

[41] Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G. (2019). Optimal and automated deployment for microservices. In Fundamental Approaches to Software Engineering: 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, pp. 351-368. https://doi.org/10.1007/978-3-030-16722-6