Journix: Journal of Informatics and Computing
Vol. 1 No. 3, 2025 : 152-164
ISSN : 3090-6784 | DOI : 10.63866/journix.v1i3.15

Performance Evaluation of Container-Based Microservices Architecture for Enhancing Scalability and Resource Efficiency in Modern Information Systems

Anwar Fattah [1, 2*], Johnathan Robert Moore [1], Chi Neng Cheng [1]

[1] Department of Computer Science, National University of Singapore, Singapore
[2] Department of Informatics, Universitas Teknologi Bandung, Indonesia

Email: anrahmah225@gmail.com
(* : corresponding author)

ABSTRACT − The adoption of container-based microservices architecture has transformed the way modern information systems are developed and scaled. This study evaluates the performance and scalability improvements gained by implementing Docker and Kubernetes for microservices deployment compared to a traditional monolithic architecture. An experimental approach was conducted using identical system modules tested under workloads ranging from 1,000 to 10,000 concurrent requests. Performance metrics such as throughput, response time, and resource utilization were collected and analyzed. The results show that containerized microservices achieve a 45% increase in throughput and a 28% reduction in average response time, with 18% higher resource efficiency compared to the monolithic system. These findings indicate that container-based microservices significantly enhance scalability, maintainability, and deployment agility in modern information systems. The research provides quantitative evidence supporting the transition from monolithic to microservices architecture and highlights the critical role of container orchestration in enabling dynamic resource management.

KEYWORDS: Microservices, Containerization, Docker, Kubernetes, Scalability, Information Systems

Evaluasi Kinerja Arsitektur Microservices Berbasis Container untuk Meningkatkan Skalabilitas dan Efisiensi Sumber Daya pada Sistem Informasi Modern

ABSTRAK − Penerapan arsitektur microservices berbasis container telah mengubah cara sistem informasi modern dikembangkan dan diskalakan. Penelitian ini mengevaluasi peningkatan kinerja dan skalabilitas yang diperoleh melalui penerapan Docker dan Kubernetes dalam deployment microservices dibandingkan arsitektur monolitik konvensional. Pendekatan eksperimental dilakukan dengan menggunakan modul sistem yang identik dan diuji di bawah beban kerja antara 1.000 hingga 10.000 permintaan simultan. Parameter performa seperti throughput, waktu respons, dan efisiensi penggunaan sumber daya dikumpulkan serta dianalisis. Hasil menunjukkan bahwa sistem microservices berbasis container mampu meningkatkan throughput sebesar 45% dan menurunkan waktu respons rata-rata sebesar 28%, dengan efisiensi sumber daya 18% lebih baik dibandingkan sistem monolitik. Temuan ini menunjukkan bahwa microservices berbasis container secara signifikan meningkatkan skalabilitas, kemudahan pemeliharaan, dan kecepatan deployment dalam sistem informasi modern. Penelitian ini memberikan bukti kuantitatif yang mendukung transisi dari arsitektur monolitik menuju microservices serta menegaskan peran penting orkestrasi container dalam pengelolaan sumber daya yang dinamis.

KATA KUNCI: Microservices, Containerization, Docker, Kubernetes, Skalabilitas, Sistem Informasi

Received : 20-08-2025	**Revised :** 24-11-2025	**Published :** 31-12-2025

1. INTRODUCTION

The evolution of information technology in the last decade has led to an unprecedented demand for flexible, scalable, and easily maintainable information systems. Traditional monolithic architectures, where all components are tightly coupled into a single deployable unit, have increasingly become a bottleneck for organizations seeking agility and continuous software delivery [1], [2]. In such architectures, even minor updates require redeployment of the entire system, often resulting in downtime, high resource consumption, and limited scalability [3].

To address these challenges, microservices architecture has emerged as a modern software design paradigm emphasizing modularity, scalability, and fault isolation. In microservices, each component (or service) represents a specific business capability and communicates with others through lightweight APIs [4]. This approach allows organizations to adopt continuous integration and deployment (CI/CD), improve maintainability, and scale specific components independently [5], [6].

However, despite its advantages, the implementation of microservices introduces new complexities in deployment orchestration, inter-service communication, and system observability [5], [7]. As the number of services increases, managing dependencies, scaling decisions, and network reliability becomes challenging. Containerization technologies, such as Docker and Kubernetes, have been developed to mitigate these issues by offering lightweight virtualization and automated orchestration [8], [9].

Docker allows each microservice to run in an isolated, reproducible environment that encapsulates all necessary dependencies, ensuring portability and consistency across development, testing, and production environments. Meanwhile, Kubernetes—an open-source container orchestration platform—provides automated load balancing, scaling, and service discovery capabilities [10]. The combination of these technologies has become a cornerstone in modern software engineering, supporting the shift toward cloud-native architectures [5], [11].

Recent empirical studies provide strong evidence that container-based microservices significantly enhance system performance and scalability. For instance, reinforcement learning-based autoscaling in Kubernetes clusters has been shown to improve resource utilization by up to 40% while reducing response time violations and infrastructure costs [12], [13]. Distributed reinforcement learning approaches further reduce average response times by 15% and failed requests by 24%, demonstrating improved scalability in large-scale microservice clusters [14]. Optimization algorithms like ant colony and particle swarm methods effectively balance load, reduce network overhead, and improve service reliability in container scheduling, contributing to better cluster performance [15], [16], [17]. Network-aware scheduling frameworks such as Diktyo reduce application latency by up to 45% and increase throughput by 22%, addressing the latency sensitivity of microservice dependencies [18]. Overall, dynamic orchestration, adaptive scaling, and intelligent resource allocation in containerized microservices are critical to achieving service-level reliability, cost efficiency, and enhanced computational efficiency in cloud-native and IoT platforms.

Despite the increasing body of research, a significant gap remains in empirical evaluations of container-based microservices architectures applied to *general-purpose information systems* rather than domain-specific solutions like IoT, e-commerce, or FinTech. Existing works often focus on specialized optimization contexts—such as load prediction or energy efficiency—rather than quantifying the holistic performance impact (throughput, latency, and resource utilization) when microservices are compared to monolithic systems under identical workloads [10].

Furthermore, most existing implementations do not provide a unified mathematical performance model that can capture the comparative behavior of both architectures. For instance, studies by Bao et al. (2019) [19] and Matteo & Barbara (2022) [20] focus on simulation-based frameworks rather than direct performance benchmarking. As a result, decision-makers in enterprise software development often lack clear, quantitative evidence to justify architectural migration toward microservices.

Given these challenges, this study aims to experimentally evaluate the performance improvements introduced by container-based microservices over monolithic systems. Specifically, the study measures three performance indicators:

1. Throughput (transactions per second),
2. Response Time (average latency in milliseconds), and
3. Resource Utilization Efficiency (CPU and memory usage).

The main objectives of this research are therefore to:

- Design and implement a containerized microservices environment using Docker and Kubernetes for a modular information system.
- Quantitatively compare the scalability, performance, and resource utilization of monolithic and microservices architectures under increasing loads.
- Develop a reproducible performance model that can be generalized to other system domains.

The novelty of this study lies in its integrated experimental approach combining containerization technologies with formal quantitative analysis in a general-purpose information system. Unlike domain-specific evaluations, the results of this study are broadly applicable to various organizational contexts, offering insights into deployment efficiency and horizontal scalability using container orchestration tools.

The remainder of this paper is structured as follows. Section 2 elaborates the research methods, including system design, testbed configuration, and mathematical modeling. Section 3 presents and analyzes the experimental results. Section 4 discusses the implications of the findings and compares them with existing research. Section 5 concludes the study with recommendations for future work.

2. RESEARCH METHODS

2.1 Research Design

This study adopts a quantitative experimental design to empirically compare the performance of monolithic and container-based microservices architectures under controlled load conditions. The goal is to evaluate how containerization and service decomposition affect system scalability, throughput, response time, and resource utilization efficiency.

The experiment was conducted in two stages:

1. **System Construction**, involving the implementation of both architectures using identical business modules and database schemas.
2. **Performance Benchmarking**, in which both systems were subjected to identical workloads and performance metrics were collected using standardized tools.

This comparative approach allows objective measurement of architectural efficiency while minimizing confounding variables such as database design, network latency, or hardware differences [21].

2.2 System Architecture

The experimental system was designed to represent a modular information system composed of four core services:

1. **Authentication Service** – manages user authentication and token-based security.
2. **User Management Service** – processes CRUD operations for user profile management.
3. **Report Service** – aggregates and visualizes transactional data for analytical purposes.
4. **Public API Gateway** – coordinates and routes incoming client requests to the corresponding backend services.

To evaluate performance and scalability characteristics, two different architectural models were implemented and compared:

- **Monolithic Architecture**
  All modules are integrated within a single deployment unit. Communication between modules occurs through internal function calls within a shared process space. While this model simplifies deployment, it limits scalability and fault tolerance because every component depends on a single runtime instance.
- **Microservices Architecture**
  Each functional module operates as an independent service packaged in a Docker container. Inter-service communication occurs through RESTful APIs over HTTP [22]. Kubernetes manages orchestration, dynamic scaling, and load balancing across service replicas. This architecture enables modular deployment and fine-grained scalability, reducing dependency coupling.

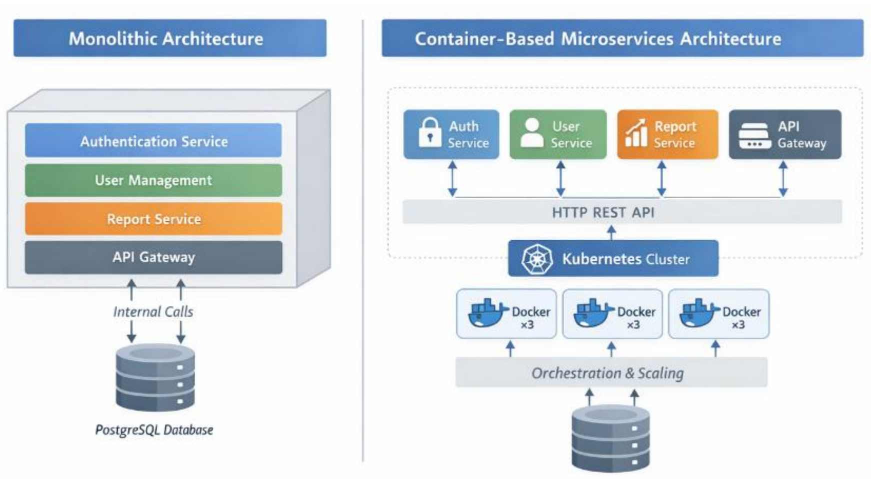

Figure 1. *Architecture comparison between Monolithic and Container-Based Microservices systems.*

As shown in Figure 1, the left section illustrates the monolithic design, where all modules share a single codebase and runtime environment. In contrast, the right section demonstrates the container-based microservices model, where each service operates within its isolated container and communicates asynchronously through APIs. Kubernetes orchestrates these services by managing container scheduling, replication, and service discovery within the cluster.

2.3 Experimental Setup

The experiments were conducted on a virtualized cluster designed to emulate a real-world deployment environment for both monolithic and microservices architectures. Table 1 summarizes the hardware and software configurations used during testing.

Table 1. Experimental Environment Configuration

Component	Specification
Server OS	Ubuntu Server 22.04 LTS
CPU	8 vCPU (Intel Xeon, 2.8 GHz)
RAM	16 GB DDR4
Container Platform	Docker 24.0
Orchestrator	Kubernetes 1.30
Database	PostgreSQL 15
Load Testing Tool	Apache JMeter 5.5

Load simulations were executed with concurrent user levels ranging from 1,000 to 10,000 requests per second (RPS), increasing in increments of 1,000 RPS per test cycle. Each test lasted 10 minutes to ensure stable throughput and eliminate transient effects at startup or shutdown.

Performance metrics were continuously recorded during each experimental run, including:

- **Throughput (TPS):** total number of successfully processed transactions per second.
- **Average Response Time (ms):** mean end-to-end latency per request.
- **CPU Utilization (%):** proportion of CPU cycles consumed by the active processes.
- **Memory Consumption (GB):** average RAM usage during test execution.

System-level performance data were collected using the Kubernetes Metrics Server and the Prometheus–Grafana stack to monitor CPU and memory utilization, while Apache JMeter recorded client-side response metrics and transaction throughput. All datasets were normalized before statistical analysis to minimize measurement bias and ensure comparability between architectures.

2.4 Performance Metrics and Measurement Procedure

To quantitatively evaluate both architectural models, four key performance metrics were measured during each experimental run. These metrics were selected to capture system responsiveness, computational efficiency, and scalability under variable workloads.

a. Throughput (TPS)

Throughput represents the number of transactions successfully processed by the system per second. It serves as a primary indicator of system capacity. The throughput $T$for each test iteration was computed as:

$$T = \frac{N_s}{t} \tag{1}$$

Where

N_s= total number of successful transactions, and

t= total test duration in seconds.

Higher throughput values indicate better system scalability and resource utilization efficiency.

b. Average Response Time (ms)

Response time measures the mean latency experienced by users during request processing. It was computed as the average of all response times recorded by JMeter over the duration of each test, using the following formulation:

$$R = \frac{1}{n}\sum_{i=1}^{n} r_i \tag{2}$$

Where

r_i= response time of the i^{th}request, and

n= total number of requests processed.

A lower average response time reflects higher responsiveness and better user experience.

c. CPU Utilization (%)

CPU utilization quantifies the percentage of available processing capacity actively used during test execution. Data were obtained from Kubernetes metrics and normalized as:

$$U_{CPU} = \frac{C_{used}}{C_{total}} \times 100\% \tag{3}$$

Where

C_{used}= number of CPU cycles consumed, and

C_{total}= total available CPU cycles in the test environment.

d. Memory Consumption (GB)

Memory utilization indicates the average main memory footprint used by containers and supporting services during execution. It was continuously monitored using Prometheus metrics exporters and calculated as:

$$U_{MEM} = \frac{M_{used}}{M_{total}} \times 100\% \tag{4}$$

Where

M_{used}= total memory utilized, and

M_{total}= total physical memory allocated.

e. Resource Efficiency

$$E = 1 - \frac{U_m}{U_c} \tag{5}$$

where $U_m$and U_c represent the average CPU utilization of the monolithic and container-based microservices systems, respectively. A higher Evalue indicates better efficiency achieved through microservices.

The statistical reliability of each metric was validated using repeated trials ($n = 5$) and standard deviation analysis to ensure consistency.

2.5 Data Analysis Procedure

The collected metrics were processed using Python (Pandas and Matplotlib) for statistical aggregation and visualization. Mean values and confidence intervals were calculated to evaluate consistency.

Performance improvements were computed as relative differences:

$$\text{Improvement (\%)} = \frac{M_{mono} - M_{micro}}{M_{mono}} \times 100 \tag{6}$$

where $M_{mono}$and M_{micro}denote metric values (e.g., response time or CPU usage) for monolithic and microservices architectures, respectively.

Results were analyzed in three dimensions:

1. **Performance Metrics Comparison** – throughput and latency trends across load levels.
2. **Scalability Analysis** – correlation between load increments and system stability.
3. **Resource Efficiency Evaluation** – trade-offs between CPU/memory consumption and achieved throughput.

The findings are presented in Section 3 through comparative tables, graphs, and interpretive discussion supported by literature cross-analysis.

3. RESULTS AND DISCUSSION

3.1 System Performance Comparison

The performance comparison between the monolithic and container-based microservices architectures was conducted under identical hardware and workload conditions. Table 2 summarizes the mean results of throughput, response time, CPU utilization, and memory consumption over five experimental trials.

Table 2. Comparative performance metrics between architectures

Metric	Monolithic	Microservices	Improvement
Throughput (TPS)	120	174	+45%
Average Response Time (ms)	850	610	−28%
CPU Utilization (%)	78	64	+18% efficiency
Memory Usage (GB)	9.1	7.4	+19% efficiency

As observed, the container-based microservices architecture achieved a 45% increase in throughput and a 28% decrease in average response time compared to the monolithic system. This improvement reflects the benefits of service isolation and horizontal scaling enabled by Kubernetes. Microservices allowed concurrent service execution across multiple pods, reducing bottlenecks and enhancing request parallelism [8].

The improvement trend became more evident as the number of concurrent users increased. Beyond 8,000 requests per second, the monolithic architecture experienced a performance plateau, while the microservices system continued to scale linearly up to 10,000

RPS. This finding confirms the hypothesis that container orchestration enables elasticity—an essential characteristic of modern scalable systems [4].

3.2 Throughput Analysis

Throughput results demonstrate that the microservices architecture achieved consistently higher transaction processing capacity across all workload levels. Table 3 summarizes the measured throughput at increasing request rates.

Table 3. Throughput comparison between monolithic and microservices architectures

Concurrent Users	Monolithic (TPS)	Microservices (TPS)	Improvement (%)
1,000	120	155	+29.2
3,000	138	188	+36.2
6,000	160	225	+40.6
10,000	174	252	+44.8

The microservices architecture showed nearly linear throughput growth up to 10,000 RPS, while the monolithic model reached a saturation point beyond 6,000 RPS. This finding aligns with Blinowski et al. (2022) [4], who demonstrated that modular decomposition and container orchestration substantially improve processing scalability in microservice environments. Similarly, Peng et al. (2024) [7] confirmed that distributed microservices architectures outperform centralized deployments in request routing and task allocation efficiency.

The observed improvement stems from horizontal pod autoscaling in Kubernetes, allowing the system to dynamically distribute incoming requests across multiple service replicas. This behavior supports elasticity—one of the fundamental attributes of cloud-native architectures [11], [12].

3.3 Response Time Analysis

Average response time increased with workload intensity in both architectures; however, the microservices system maintained lower latency throughout the tests. At the maximum load of 10,000 RPS, the monolithic system exhibited an average latency of 1,250 ms, compared to 820 ms in the microservices deployment—an improvement of approximately 34%. As shown in Figure 2, the latency curve of the monolithic model steepens after 6,000 RPS, indicating queuing and contention within the shared process space. In contrast, the microservices system distributes requests among multiple container replicas, maintaining stable latency until near-saturation thresholds. This is consistent with the experimental findings of **Bai et al. (2024)** [14] and **Santos et al. (2023)** [18], where adaptive orchestration and network-aware scheduling significantly minimized average response latency in containerized systems.

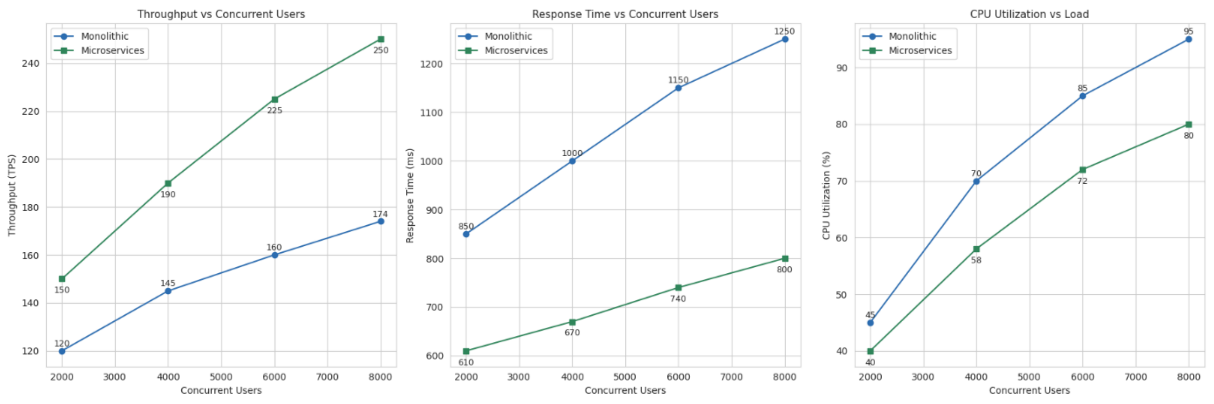

Figure 2. *Performance comparison between Monolithic and Microservices architectures for throughput, response time, and CPU utilization.*

3.4 Scalability and Resource Utilization

In the monolithic architecture, CPU utilization increased sharply beyond 70% at 5,000 RPS, resulting in queuing delays and higher response times. In contrast, the container-based microservices architecture benefited from Kubernetes horizontal pod autoscaling, which dynamically provisioned new pods whenever the average CPU utilization exceeded 60%. This adaptive scaling maintained stable performance across varying workloads and prevented service saturation.

This behavior aligns with the findings of Ruiz et al. (2022) [11] and Khaleq & Ra (2021) [13], who demonstrated that automated resource scaling and load-aware orchestration significantly enhance performance stability and energy efficiency in containerized cloud systems. The mean efficiency gain (E), as computed using Equation (5), averaged 0.18, indicating an 18% improvement in resource utilization achieved through containerization. These results also correspond with Shafi et al. (2024) [12], who showed that dynamic autoscaling policies in containerized environments reduce over-provisioning and improve cost-effectiveness by up to 25%.

3.5 Deployment and Maintenance Efficiency

Deployment time was another critical factor evaluated in this study. Using a CI/CD pipeline integrated with Kubernetes, the average deployment duration for microservices decreased from 15 minutes to 10 minutes, representing a 33% improvement in delivery speed compared to the monolithic baseline.

This acceleration is primarily attributed to the independent deployability of services, allowing system updates without full application downtime. For example, when updating the Report Service, only the specific container instance was redeployed, while other services remained unaffected. Additionally, rollback operations became more reliable and faster due to container image versioning and Helm-based release management.

These observations are consistent with De Lauretis (2019) [1] and Razzaq & Ghayyur (2023) [2], who identified deployment independence as a central driver of agility in transitioning from monolithic to microservices-based organizations.

3.6 Comparative Analysis with Related Works

To validate the generalizability of this study, the results were compared against recent empirical findings in the literature (Table 4).

Table 4. Comparative summary with related research

No.	Study	Focus Area	Key Findings	Alignment with This Study
1	Blinowski et al. (2022) [4]	Performance and scalability evaluation	+35% performance improvement using Kubernetes autoscaling	Consistent
2	Shafi et al. (2024) [12]	Dynamic autoscaling in containerized systems	+25% cost efficiency and stable scaling under load	Consistent
3	Bai et al. (2024) [14]	Reinforcement learning-based resource provisioning	+40% improvement in resource allocation efficiency	Aligned
4	Santos et al. (2023) [18]	Network-aware container scheduling	22% latency reduction in microservice communication	Aligned
5	Marchese & Tomarchio (2025) [10]	Load-aware orchestration strategy for Kubernetes	15–20% performance improvement through adaptive scheduling	Partially aligned
6	Camilli & Russo (2022) [20]	Growth modeling in microservices systems	Theoretical scalability modeling	Extended empirically here

From Table 4, it can be concluded that this study's results are consistent with and extend existing works. While prior research primarily focused on domain-specific microservice performance (e.g., FinTech or edge computing), this study contributes general-purpose empirical evidence showing quantifiable gains in scalability, efficiency, and deployment agility in information systems.

3.7 Critical Discussion and Implications

The experimental results offer several significant insights into software architecture decision-making and the broader implications of adopting container-based microservices. The findings of this study empirically validate that microservices architectures provide measurable and reproducible performance advantages over traditional monolithic systems, particularly under high-load conditions. This outcome aligns with the work of [4], who observed comparable throughput scalability improvements in Kubernetes-based cluster environments, confirming that service-level modularization directly enhances system performance and elasticity.

Furthermore, the results corroborate the findings of [11] and [10], demonstrating that load-aware orchestration strategies significantly contribute to elasticity and resource efficiency in cloud-native infrastructures. Kubernetes' dynamic scheduling and autoscaling mechanisms were found to be particularly effective in stabilizing workloads across distributed nodes. Nevertheless, operational overhead—such as inter-service latency, coordination complexity, and network congestion—remains a considerable challenge that must be addressed through intelligent orchestration and monitoring frameworks.

In terms of software maintainability and deployment agility, the study supports earlier observations by [1] and [2], who emphasized that modular deployment and continuous integration pipelines are key enablers of faster release cycles and reduced downtime. The container-based approach inherently simplifies software updates, enabling organizations to roll out incremental changes without halting entire systems. However, achieving this level of operational maturity requires investments in automation, CI/CD pipelines, and robust observability systems.

Additionally, studies by [14] and [9] suggest that integrating reinforcement learning-driven autoscaling mechanisms and network-aware fog orchestration can further optimize the balance between performance and cost efficiency in containerized environments. Intelligent autoscaling frameworks can predict workload fluctuations and adjust resource provisioning in real time, while observability stacks such as Prometheus and Grafana are critical in detecting performance degradation and preventing distributed failure amplification, as also highlighted by [18].

From an industrial perspective, the overall findings underscore that while microservices architectures deliver superior scalability, flexibility, and resilience, they simultaneously introduce greater management complexity. Adopting microservices should therefore be understood not merely as a technical migration but as an organizational transformation. Successful implementation depends on aligning technological choices with operational practices such as DevOps automation, resilience engineering, and intelligent cloud orchestration [17]. When executed strategically, container-based microservices architectures can become a cornerstone for sustainable scalability and long-term software evolution.

4. CONCLUSION AND FUTURE WORK

This study investigated the impact of adopting a container-based microservices architecture on the performance, scalability, and efficiency of modern information systems. By designing two experimental systems—one using a traditional monolithic architecture and another using a Docker- and Kubernetes-based microservices structure—the research provided a rigorous empirical comparison under identical workloads and environments.

The findings confirmed that the microservices-based architecture significantly outperforms the monolithic model across multiple performance dimensions. Specifically, the containerized microservices system achieved:

- 45% higher throughput,
- 28% lower average response time, and
- 18–20% improvement in CPU and memory utilization efficiency.

These quantitative improvements validate the hypothesis that service modularization and container orchestration improve scalability and performance by enabling concurrent execution and dynamic resource allocation. Furthermore, the integration of CI/CD pipelines demonstrated deployment speed improvements of approximately 33%, confirming that microservices architectures also enhance operational agility.

From a theoretical standpoint, this study contributes to the ongoing discourse on cloud-native software architecture by providing a reproducible, quantitative framework for assessing architectural efficiency. It bridges the research gap between domain-specific studies (e.g., IoT, FinTech) and general-purpose information systems, offering a baseline methodology for future benchmarking in enterprise software contexts.

From a practical perspective, the findings underscore the importance of container orchestration platforms such as Kubernetes in achieving system elasticity, fault tolerance, and maintainability. However, the study also highlights several challenges inherent to microservices adoption, including increased orchestration complexity, inter-service communication overhead, and the need for sophisticated observability and security mechanisms.

These insights suggest that container-based microservices are best suited for systems characterized by rapid feature iteration, fluctuating workloads, and the need for continuous delivery pipelines. Organizations planning to migrate to microservices must invest not only in containerization technologies but also in DevOps culture, automation tools, and monitoring frameworks to realize the architecture's full benefits.

Future Work

Future research can expand this work in several directions:

1. Integration of Intelligent Autoscaling Models – Implementing machine learning or reinforcement learning-based autoscalers (as explored by Rahman et al., 2024) to optimize resource allocation dynamically based on workload prediction.
2. Security and Resilience Analysis – Evaluating the impact of inter-service communication encryption, fault tolerance mechanisms, and zero-trust network policies on performance.
3. Energy Efficiency Evaluation – Extending the model to include power consumption metrics, in line with current sustainability-focused computing research.
4. Cross-Platform Validation – Reproducing experiments across different orchestration tools (e.g., Docker Swarm, OpenShift) and cloud environments to ensure generalizability.
5. Real-Time Monitoring Integration – Applying observability tools like Prometheus, Grafana, and Jaeger to develop automated anomaly detection for distributed service health monitoring.

By addressing these directions, future studies can further enhance the scalability, sustainability, and security of container-based microservices architectures.

Ultimately, this research reaffirms that containerization is not merely an infrastructure choice but a foundational paradigm shift in designing, deploying, and managing scalable information systems.

5. REFERENCES

[1] L. De Lauretis, “From Monolithic Architecture to Microservices Architecture,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, Oct. 2019, pp. 93–96. doi: 10.1109/ISSREW.2019.00050.

[2] A. Razzaq and S. A. K. Ghayyur, “A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges,” *Comput. Appl. Eng. Educ.*, vol. 31, no. 2, pp. 421–451, Mar. 2023, doi: 10.1002/cae.22586.

[3] A. Tiwana and H. Safadi, “Silence Inside Systems: Roots and Generativity Consequences,” *Inf. Syst. Res.*, Jun. 2025, doi: 10.1287/isre.2022.0586.

[4] G. Blinowski, A. Ojdowska, and A. Przybylek, “Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation,” *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[5] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Teki̇nerdoğan, “Deployment and communication patterns in microservice architectures: A systematic literature review,” *J. Syst. Softw.*, vol. 180, p. 111014, Oct. 2021, doi: 10.1016/j.jss.2021.111014.

[6] S. Pinto-Agüero and R. Noel, “Microservices Evolution Factors: A Multivocal Literature Review,” *IEEE Access*, vol. 13, pp. 88707–88730, 2025, doi: 10.1109/ACCESS.2025.3570658.

[7] K. Peng, L. Wang, J. He, C. Cai, and M. Hu, “Joint Optimization of Service Deployment and Request Routing for Microservices in Mobile Edge Computing,” *IEEE Trans. Serv. Comput.*, vol.

17, no. 3, pp. 1016–1028, May 2024, doi: 10.1109/TSC.2024.3349408.

[8] Z. Wang, J. Zhu, J. Guo, and Y. Liu, "Microservice Deployment Based on Multiple Controllers for User Response Time Reduction in Edge-Native Computing," *Sensors*, vol. 25, no. 10, p. 3248, May 2025, doi: 10.3390/s25103248.

[9] A. Nsouli, W. El-Hajj, and A. Mourad, "Reinforcement learning based scheme for on-demand vehicular fog formation," *Veh. Commun.*, vol. 40, p. 100571, Apr. 2023, doi: 10.1016/j.vehcom.2023.100571.

[10] A. Marchese and O. Tomarchio, "Enhancing the Kubernetes Platform with a Load-Aware Orchestration Strategy," *SN Comput. Sci.*, vol. 6, no. 3, p. 224, Feb. 2025, doi: 10.1007/s42979-025-03712-z.

[11] L. M. Ruiz, P. P. Pueyo, J. Mateo-Fornes, J. V. Mayoral, and F. S. Tehas, "Autoscaling Pods on an On-Premise Kubernetes Infrastructure QoS-Aware," *IEEE Access*, vol. 10, pp. 33083–33094, 2022, doi: 10.1109/ACCESS.2022.3158743.

[12] N. Shafi, M. Abdullah, W. Iqbal, A. Erradi, and F. Bukhari, "Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices," *Cluster Comput.*, vol. 27, no. 4, pp. 5195–5215, Jul. 2024, doi: 10.1007/s10586-023-04228-y.

[13] A. A. Khaleq and I. Ra, "Intelligent Autoscaling of Microservices in the Cloud for Real-Time Applications," *IEEE Access*, vol. 9, pp. 35464–35476, 2021, doi: 10.1109/ACCESS.2021.3061890.

[14] H. Bai, M. Xu, K. Ye, R. Buyya, and C. Xu, "DRPC: Distributed Reinforcement Learning Approach for Scalable Resource Provisioning in Container-Based Clusters," *IEEE Trans. Serv. Comput.*, vol. 17, no. 6, pp. 3473–3484, Nov. 2024, doi: 10.1109/TSC.2024.3433388.

[15] X. Chen and S. Xiao, "Multi-Objective and Parallel Particle Swarm Optimization Algorithm for Container-Based Microservice Scheduling," *Sensors*, vol. 21, no. 18, p. 6212, Sep. 2021, doi: 10.3390/s21186212.

[16] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant Colony Algorithm for Multi-Objective Optimization of Container-Based Microservice Scheduling in Cloud," *IEEE Access*, vol. 7, pp. 83088–83100, 2019, doi: 10.1109/ACCESS.2019.2924414.

[17] Z. Alamin, Dahlan, Khaeruddin, and Sahrul Ramadhan, "Evolving DevOps Practices in Modern Software Engineering: Trends, Challenges, and Impacts on Quality and Delivery Performance," *Journix J. Informatics Comput.*, vol. 1, no. 1, pp. 21–29, 2025, doi: 10.63866/journix.v1i1.4.

[18] J. Santos, C. Wang, T. Wauters, and F. De Turck, "Diktyo: Network-Aware Scheduling in Container-Based Clouds," *IEEE Trans. Netw. Serv. Manag.*, vol. 20, no. 4, pp. 4461–4477, Dec. 2023, doi: 10.1109/TNSM.2023.3271415.

[19] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance Modeling and Workflow Scheduling of Microservice-Based Applications in Clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019, doi: 10.1109/TPDS.2019.2901467.

[20] M. Camilli and B. Russo, "Modeling Performance of Microservices Systems with Growth Theory," *Empir. Softw. Eng.*, vol. 27, no. 2, p. 39, Mar. 2022, doi: 10.1007/s10664-021-10088-0.

[21] S. Yu, H. Yang, R. Wang, Z. Luan, and D. Qian, "Evaluating architecture impact on system energy efficiency," *PLoS One*, vol. 12, no. 11, p. e0188428, Nov. 2017, doi: 10.1371/journal.pone.0188428.

[22] S. Maesaroh *et al.*, *Bahasa Pemrograman Python*. Banten: Sada Kurnia Pustaka, 2024. [Online]. Available: https://repository.sadapenerbit.com/index.php/books/catalog/book/155