Available online at website: https://jurnal.iaii.or.id/index.php/RESTI

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 9 No. 3 (2025) 511 - 517 e-ISSN: 2580-0760

Performance Comparison of Monolithic and Microservices Architectures in Handling High-Volume Transactions

Mastura Diana Marieska[1]*, Arya Yunanta[2], Harisatul Aulia[3], Alvi Syahrini Utami[4], Muhammad Qurhanul Rizqie[5]
[1,2,3,4,5]Department of Informatics Engineering, Faculty of Computer Science, Universitas Sriwijaya, Palembang, Indonesia

[1]mastura.diana@ilkom.unsri.ac.id, [2]aryayun90@gmail.com, [3]haris.aulia404@gmail.com, [4]alvisyahrini@ilkom.unsri.ac.id, [5]qurhanul.rizqie@ilkom.unsri.ac.id

Abstract

Monolithic and microservices are two distinct approaches for designing and developing applications. However, these architectures exhibit contrasting characteristics. In monolithic architecture, all components of an application form a unified entity with closely interconnected parts, whereas microservices decompose an application into independent, lightweight services that can be developed, deployed, and updated separately. Microservices are often regarded as superior to monolithic architectures in terms of their performance. This study aims to compare the performance of monolithic and microservices architectures in handling a high volume of transactions. It is important to observe how the two architectures behave when handling transactions from a large number of concurrent users. A prototype of an online ticketing system was implemented for both architectures to enable comparative analysis. The selected performance metrics were response time and error rate. The experimental results reveal that under high-load conditions, microservices outperform monolithic architectures, demonstrating 36% faster response times and 71% fewer errors. However, under overload conditions—when CPU usage exceeds 90%—the performance of microservices degrades significantly. This does not imply that microservices cannot handle a large number of concurrent users but highlights the necessity for enhanced resource management.

Keywords: event-driven architecture; microservice; monolithic; online ticketing system

How to Cite: M. D. Marieska, Arya Yunanta, Harisatul Aulia, Alvi Syahrini Utami, and Muhammad Qurhanul Rizqie, "Performance Comparison of Monolithic and Microservices Architectures in Handling High-Volume Transactions", *J. RESTI (Rekayasa Sist. Teknol. Inf.)* , vol. 9, no. 3, pp. 511 - 517, Jun. 2025.
Permalink/DOI: https://doi.org/10.29207/resti.v9i3.6183

Received: November 20, 2024
Accepted: June 8, 2025
Available Online: June 19, 2025

1. Introduction

Monolithic services are a traditional software architecture where the entire application is a single, unified entity with closely linked components. This approach faces significant issues with scalability and flexibility. Scaling the system to handle higher demand requires scaling the entire application, which is often inefficient [1]. Additionally, a failure in one component can impact the entire application's performance [2].

In contrast, microservices architecture breaks down an application into independent, lightweight services that can be developed, deployed, and updated separately. This approach enhances flexibility and scalability compared to monolithic systems [3]. Each microservice operates independently and communicates through lightweight protocols like APIs [4]. Scalability is a key benefit organization seek when shifting to microservices [5].

The shift from monolithic architectures to microservices marks a crucial evolution in software development, driven by demands for greater scalability, flexibility, and maintainability. This is not a simple change, especially when coming from traditional object-oriented systems, because it requires rearranging a lot of system parts and adding a lot of interactions between services [6]. Furthermore, the migration process can be resource-intensive and time-consuming, often requiring a comprehensive rewrite of existing applications [7], [8]. The best level of detail for microservices is crucial to this migration because it affects application quality and resource use [9].

Comparative research highlights that while monolithic architectures feature a single and integrated codebase,

microservices offer separate and loosely coupled services [10]. Microservices provide improved scalability, agility, and independent service deployment, leading to enhanced performance [11]. Transitioning to microservices can improve stability, performance, and scalability, and promote more efficient resource use [12], [13]. One of the primary advantages of microservices is their ability to scale independently, which can significantly enhance performance under high demand. As a result, while monolithic applications might exhibit lower average latency for specific features, microservices often deliver superior performance overall [14].

To effectively manage systems that experience high performance demands due to a large number of concurrent users, it is crucial to conduct case studies that simulate such high-traffic environments. These studies are essential for assessing how well systems handle extensive user interactions and transaction loads, providing insights into their scalability and reliability under significant stress.

An illustrative example is an online ticketing system designed to handle large-scale events, such as concerts at expansive venues. Such systems must efficiently manage a high volume of transactions and simultaneous user access, highlighting the need for robust performance capabilities. Selling concert tickets online offers significant convenience compared to offline sales, which may involve long queues and large crowds. However, online ticket sales can generate extremely high traffic, with thousands of tickets potentially being sold within minutes.

Several other studies have compared monolithic and microservices architectures. One study [15] focuses on scalability, concluding that monolithic is more suitable for systems that do not require handling a large number of concurrent users. Another study [16], emphasized performance aspects of database usage, concluding that the latency of microservices database access is higher than monolithic. Another study [17] details the challenges encountered when migrating from monolithic to microservices.

Unlike other research, this study compares monolithic and microservices architectures to evaluate which one provides better performance and reliability for systems that handle a lot of transactions and multiple users interacting at the same time. The performance aspect will be analyzed by average and percentile 90 response time. The reliability aspect will be analyzed by error rate. By analyzing an online ticketing system designed for large-scale events, the study aims to determine how each architecture handles the demands of high-volume transaction processing and user access, ultimately providing insights into their effectiveness in managing such performance-intensive scenarios.

2. Methods

This section will explain the research methods employed in this study, focusing on the online ticketing system as the primary object of analysis. We will explore two distinct architectural approaches: monolith implementation and microservice implementation. Each approach will be examined to understand its structure, design principles, and operational characteristics.

2.1 Online Ticketing System

E-tickets, digital versions of traditional paper tickets for events, are becoming popular for their convenience and efficiency. Purchased online and stored on mobile devices, they eliminate the need for physical tickets and streamline access to events [18] . E-tickets simplify purchasing, reduce fraud, and enhance user experience with immediate access and updates. They also integrate with mobile apps for real-time notifications and easy transfers [18], [19].

The online ticketing system has been selected for this case study due to the significant transaction volume it must handle. For instance, during a major concert at a large stadium with online ticket sales, the system must efficiently process a large number of transactions in a very short period. This scenario highlights the need for a highly efficient and resilient system capable of managing rapid and numerous ticket purchases seamlessly, thereby ensuring a smooth and reliable experience for all users.

In this study, the online ticketing system to be developed will incorporate several key features, including the ability to view available tickets, purchase tickets, and process payments. Figure 1 shows a system flow design between key features. Additionally, the system will provide functionality for processing ticket cancellations within a specified time period.

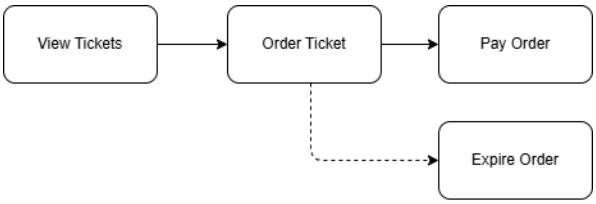

Figure 1. System Flows

This study employs a technology stack consisting of Go, PostgreSQL, Redis, and Apache Kafka. Go is used for its efficient concurrency and performance, PostgreSQL for robust relational data management, Redis for rapid in-memory data access and caching, and Kafka for real-time data streaming and processing.

2.2 Monolith Implementation

Monolithic services represent a conventional software architecture in which the entire application is developed as a single, unified entity. This design integrates all components into one cohesive system, which poses significant challenges for scalability and flexibility. In

a monolithic implementation, the system utilizes three components on the server: the web service, which is the monolith itself; a cache for temporary data storage; and a database for permanent data storage. Figure 2 illustrates interactions between three components. All features of the e-ticketing system are integrated into a single web service, which is the monolithic application.

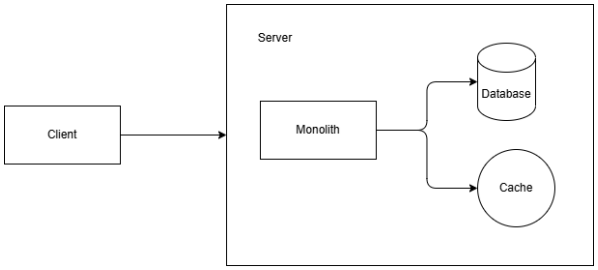

Figure 2. Online Ticketing System Design on Monolith

2.3. Microservice Implementation

Microservice is an architectural style where a system is divided into small, independently deployable services, each responsible for a specific function. This approach contrasts with monolithic systems by enhancing scalability and flexibility but introduces challenges in communication and management.

In a microservice implementation, the single web service is divided into multiple components based on business domains using Domain-Driven Design (DDD). DDD aligns software architecture with the business domain by deeply understanding the specific industry and creating a domain model that addresses its complexities and needs [20], [21]. This approach results in distinct services such as Category Service, Order Service, and Scheduler Service. Figure 3 illustrates interaction between components and services in microservices.

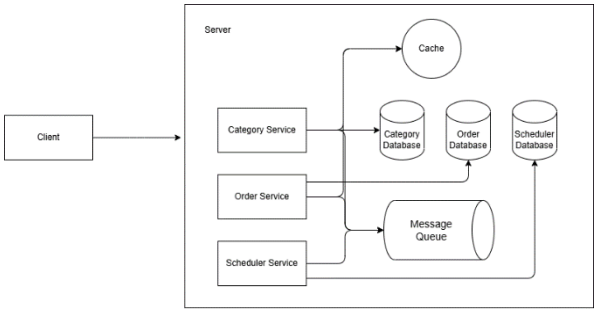

Figure 3. Online Ticketing System Design on Microservices

Category Service is responsible for managing the business logic related to ticket categories, including detailed category information and the remaining quantity of tickets available within each category. Order Service handles the business logic associated with ticket reservations, such as processing ticket bookings and paying for reservations of tickets. Lastly, the Scheduler Service is responsible for managing schedules to determine when a reservation will be cancelled in accordance with predefined deadlines.

In this research, implemented microservices use an event-driven for architecture design. Event-driven architecture (EDA) is a design pattern where system components communicate through events rather than direct interactions. This approach allows for a decoupled, flexible system that enhances scalability and responsiveness [22]-[24]. Asynchronous communication is supported by EDA, which lets components work separately and be easily changed without affecting the system [25].

Event-driven architecture does not interact directly with other services but instead maintains local copies of their data. This often leads to data duplication and requires synchronization by listening for changes, such as additions or modifications, in other services. The following is an implementation of event-driven architecture in this research.

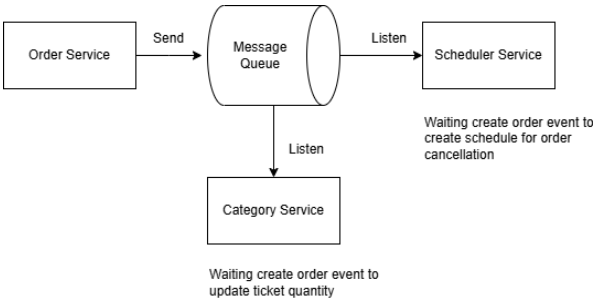

Figure 4. System Flow Design of Create Order Event

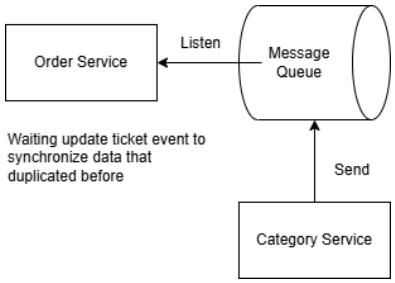

Figure 5. System Flow Design of After Create Order Event

Figure 4 illustrates the process of order creation, where the order service sends an event message to a message queue. Subsequently, other services, such as the scheduler and category services, capture the event data related to the order creation.

Following this, Figure 5 depicts a follow-up event focused on data synchronization post-order creation. This process involves synchronizing relevant data, specifically the ticket quantity. It's important to note that only the owner service has the authority to update its domain data, ensuring data integrity and proper access control.

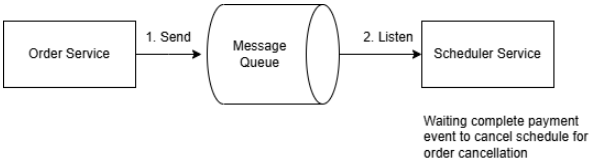

Figure 6. System Flow Design of Complete Order Event

Furthermore, Figure 6 illustrates the process of completing order payment. During this stage, the service sends an event message to the message queue, which is then captured by the service scheduler. This allows the scheduler to cancel any pending order cancellations.

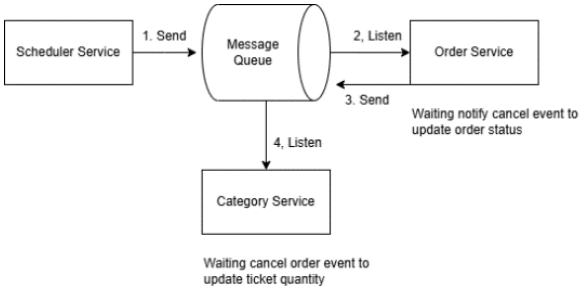

Figure 7. System Flow Design of Cancel Order Event

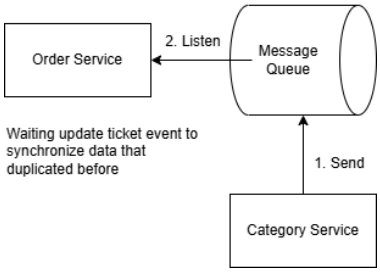

Figure 8. System Flow Design of After Cancel Order Event

Lastly, Figure 7 illustrates the order cancellation process. In this phase, the service scheduler sends an event message to the message queue, which is captured by the order service to initiate a status update. Subsequently, the order service sends another event message to the message queue, which the category service captures to update the ticket quantity accordingly. Also, Figure 8 illustrates the process that follows order cancellation, specifically focusing on the data synchronization process. This step ensures that all relevant systems are updated to reflect the changes resulting from the canceled order.

2.4 Performance Metric

Response time and software error rate are key metrics in software engineering that significantly impact application performance and reliability. While response time measures the speed at which a system provides feedback, the error rate reflects the frequency of defects within the software. Alongside response time, the P(90) metric, or the 90th percentile response time, measures the threshold within which 90% of requests are completed.

The 90th percentile provides a more reliable representation of system performance under high-load condition, reducing the influence of extreme outliers [26]. These metrics provide insight into user experience, indicating how many users experience slower response times during peak loads. The software error rate, specifically, indicates how often defects occur in systems.

2.5 Research Limitations and Managerial Implications

There are several limitations in this study, particularly related to the hardware and the developed testing application. The online ticketing system created is a prototype focused on the implementation of services and the flow of each event. This system does not have a GUI and has not been tested with real users. Virtual users were created using Grafana K6.

Hardware used for performance testing was a consumer-grade computer, with specifications detailed in the subchapter Test Configuration. This may differ from real-world cases where servers supporting high-performing applications would undoubtedly have high specifications with significant RAM capacity. However, testing with a standard consumer-grade computer is considered sufficient in the context of comparing the performance of the two architectures.

2.6 Test Configuration

Load testing is a critical component of performance evaluation, aimed at determining how a system performs under anticipated user traffic. Its primary goals are to identify potential bottlenecks, ensure system stability, and confirm that the application maintains performance levels under expected loads.

Grafana k6 is an open-source load testing tool designed to assess application performance effectively. It enables developers to simulate user interactions and evaluate how applications cope with different load conditions. This tool is particularly useful for ensuring sustained high performance throughout the application's lifecycle [27].

The testing methodology involves issuing requests via Grafana K6 for a duration of 30 seconds, simulating the complete ticketing process, including viewing tickets, ordering, and making payments. Furthermore, the test incorporates a failure scenario in which 1/3 of the transactions are simulated to fail due to non-payment.

In performance testing, "concurrent users" is a key metric for assessing how well a system manages simultaneous interactions. These are virtual users created to interact with the system concurrently during a load test. Tools like k6 facilitate the simulation and management of these virtual users, enabling comprehensive stress testing under various load conditions.

When multiple virtual users operate simultaneously, they generate a significant volume of requests, challenging the system's capacity to handle and process them effectively. This approach accurately reflects real-world scenarios, such as peak usage periods when numerous users access the application at once.

Requests Per Second (RPS) is a crucial performance metric that quantifies how many requests a server or service can handle in one second. It is vital for assessing the efficiency and scalability of web applications, APIs, and networked services. A clear understanding of RPS

enables developers to optimize user experience and allocate resources more effectively. In this research, we tested the system with concurrent user levels of 50, 100, 150, 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, and 700 to evaluate its performance under varying loads.

This approach allows us to assess how well the system handles increasing stress and identify any performance issues. The performance test was conducted on a laptop configured for thorough evaluation under varying loads, featuring an Intel Core i5-8350U processor, 24 GB of RAM, and Fedora 40 as the operating system.

3. Results and Discussions

This section will present the test results, followed by a discussion of the findings and their implications. We will analyze the outcomes of the performance tests conducted on the e-ticketing system, comparing the results between the monolith and microservice implementations. Finally, we will draw conclusions based on the insights gained from the analysis.

3.1 Test Results

We conducted a test within a defined operational environment specifically configured for load testing, ensuring that no other operations were running concurrently. This test was carried out over five iterations to enhance the consistency and reliability of the results. Following the completion of all iterations, we averaged the outcomes to construct a comprehensive measurement that accurately reflects the performance characteristics of the configuration under investigation. This methodology strengthens the validity of our findings by minimizing the influence of potential outlier results.

The collected data includes a variety of performance metrics, such as Requests Per Second (RPS), which is measured in requests per second (req/s), average response time, the 90th percentile response time (P90) expressed in milliseconds (ms), and the error rate represented as a percentage.

Here is the format of the test results that will be utilized later, as illustrated in Table 1. The upper section presents the results for the monolithic architecture, while the lower section outlines those for the microservices architecture.

Table 1. Load Test Result Format

User	RPS	Avg	P(90)	Error Rate
50	x1	x2	x3	x4
	y1	y2	y3	y4

Figure 9 shows a comparison of the results based on the response time metrics, specifically the average and the 90th percentile (P90).

Based on the average and 90th percentile (P90) metrics, the graph shows that the microservices architecture has a faster response time than the monolithic architecture at first. As the number of virtual users increases, the response time metrics rise; however, there are several fluctuations that will be addressed in the subsequent discussion session.

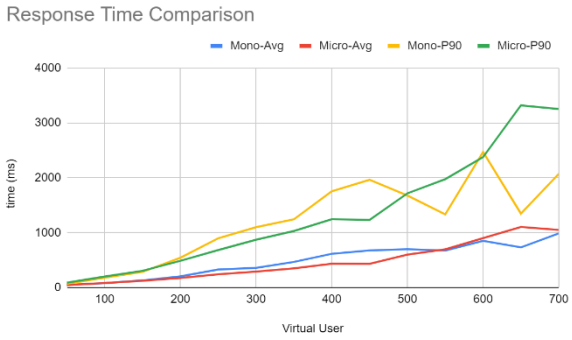

Figure 9. Response Time Graph Comparison

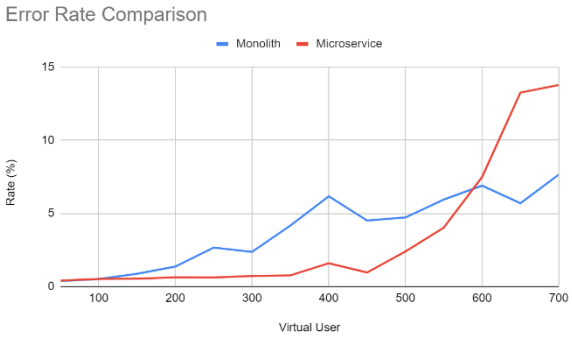

Figure 10. Error Rate Graph Comparison

Table 2. Load Test Result

User	RPS	Avg	P(90)	Error Rate
50	471	45	69	0.3
	453	45	89	0.4
100	699	80	177	0.5
	633	82	200	0.5
150	764	130	284	0.8
	718	121	304	0.5
200	736	203	542	1.3
	735	173	486	0.6
250	649	327	896	1.3
	711	240	682	0.6
300	700	357	1099	2.3
	735	290	870	0.7
350	661	466	1242	4.1
	742	348	1030	0.7
400	581	614	1753	6.1
	737	433	1244	1.5
450	603	675	1964	4.5
	797	431	1232	0.9
500	613	699	1676	4.7
	680	599	1716	2.3
550	718	673	1332	5.94
	657	696	1974	4
600	638	859	2470	6.8
	582	899	2376	7.4
650	818	731	1346	5.6
	516	1102	3318	13.2
700	651	986	2070	7.6
	588	1049	3252	13.7

Based on the error rate metric, a comparison graph of the results obtained, is shown in Figure 10. The graph depicts the error rates derived from the test results,

demonstrating that microservices generally yield lower error rates compared to monolithic architectures. However, it is important to note that fluctuations occur at certain points within the data.

Furthermore, the results of the tests and comparisons are presented in tabular format in Table 2. Based Table 2, the lowest recorded average response time for both microservices and monolithic architectures is 45 ms. The lowest p90 response time is 69 ms for the monolith and 89 ms for the microservices. Regarding error rates, the monolithic architecture shows a minimum error rate of 0.3%, while the microservices exhibit a minimum error rate of 0.4%. Furthermore, the maximum average response time for the monolith is 986 ms, compared to 1.1 s for the microservices. The maximum p90 response time for the monolith is 2.5 s, while it is 3.3 s for the microservices. Finally, the maximum error rate for the monolithic architecture is 7.6%, whereas the microservices demonstrate a maximum error rate of 13%.

3.2 Discussions

Based on the results of the experiments conducted, significant changes were observed in each scenario involving varying numbers of concurrent users, particularly when the number of users exceeded 450. The data revealed an irregular pattern of fluctuations, accompanied by noticeable spikes.

These fluctuations stem from limited computing resources, especially the CPU. During testing, CPU utilization reached an average of 90%, indicating that the system was operating at its maximum capacity. This situation led to slower response times during the testing process and an increase in errors.

The impact of these fluctuations is even more pronounced in a microservices architecture. This model typically consumes more resources than a monolithic architecture, as each microservice can operate multiple instances, including services, databases, and message brokers. As a result, with more components functioning simultaneously, CPU demand rises, which can negatively affect performance as the number of users increases.

In terms of response time, when the number of concurrent users is 450 or fewer, the microservices architecture demonstrates superior performance. Specifically, it is 36% faster than the monolith in average response time and 56% faster in the p90 metric. However, when there are more than 450 users at the same time, the response time for microservices goes up. In the p90 metric, it takes 25% longer for microservices to respond than for monoliths, and it takes 10% longer on average.

Regarding error rates, when the number of concurrent users is 450 or more, microservices achieve a 71% lower error rate compared to the monolith. Nevertheless, in scenarios with over 450 users, the error rate for microservices rises, becoming 35% higher than that of the monolith.

The findings of this study are consistent with one of the findings in another study [15], specifically regarding scalability limitations that contribute to the degradation of application performance in microservices architecture. A microservices requires more resources than a monolithic architecture, as it can run multiple instances in parallel. However, this capability can lead to a drastic decline in application performance in high-load environment. The test result indicates that when the number of concurrent users surpasses 450, there is noticeable rise in error rates and an increase in response time.

4. Conclusions

The results indicate that microservices architecture excels in both performance and error rates under moderate loads. While it is designed to manage high traffic, challenges related to resource limitations and communication overhead become apparent as load increases. This does not imply that microservices cannot handle a high number of users but highlights the necessity for enhanced resource management. Therefore, ongoing optimization and monitoring efforts are crucial to ensuring optimal performance during periods of increased user activity.

Acknowledgements

The research of this article was funded by DIPA of the Public Service Agency of Universitas Sriwijaya 2024. SP DIPA number: 023.17.2.677515/2024, on November 24th, 2023. In accordance with the Rector's Decree number: 0013/UN9/LP2M.PT/2024, on May, 20th, 2024.

References

[1] R. Bolscher and M. Daneva, "Designing software architecture to support continuous delivery and DevOps: A systematic literature review," in *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies*, SciTePress, 2019, pp. 27–39. doi: 10.5220/0007837000270039.

[2] F. H. Khoso, A. Lakhan, A. A. Arain, M. A. Soomro, S. Z. Nizamani, and K. Kanwar, "A Microservice-Based System for Industrial Internet of Things in Fog-Cloud Assisted Network", *Eng. Technol. Appl. Sci. Res.*, vol. 11, no. 2, pp. 7029–7032, Apr. 2021. https://doi.org/10.48084/etasr.4077

[3] F. Dai, G. Liu, X. Xu, Q. Mo, Z. Qiang, and Z. Liang, "Compatibility checking for cyber-physical systems based on microservices," *Softw Pract Exp*, vol. 52, no. 11, pp. 2393–2410, Nov. 2022, doi: 10.1002/spe.3131.

[4] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, Apr. 2019, doi: 10.1016/j.jss.2019.01.001.

[5] S. Hassan, R. Bahsoon, and R. Buyya, "Systematic Scalability Analysis for Microservices Granularity Adaptation Design Decisions," *Softw Pract Exp*, 2022, 52(6): 1378–1401, doi: 10.1002/spe.3069.

[6] J. H. Duarte Correia and A. R. Silva, "Identification of Monolith Functionality Refactorings for Microservices

Migration," *Softw Pract Exp*, 2022, 52(12): 2664–2683, doi: 10.1002/spe.3141.

[7] R. Gebler, "Supporting Regional Pandemic Management by Enabling Self-Service Reporting—A Case Report," *PLoS One*, 2024, Jan 31;19(1):e0297039, doi: 10.1371/journal.pone.0297039.

[8] J. Kazanavičius and D. Mažeika, "The Evaluation of Microservice Communication While Decomposing Monoliths", *Comput. Inform.*, vol. 42, no. 1, pp. 1–36, May 2023. https://doi.org/10.31577/cai_2023_1_1

[9] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, "Defining and measuring microservice granularity—a literature overview," *PeerJ Comput Sci*, vol. 7, p. e695, Sep. 2021, doi: 10.7717/peerj-cs.695.

[10] B. Salles and J. Cunha, "Visually-Assisted Decomposition of Monoliths to Microservices," 2023 *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Washington, DC, USA, 2023, pp. 293-295, doi: 10.1109/VL-HCC57772.2023.00057.

[11] H.-P. Huang, Y.-Y. Fanjiang, C.-H. Hung, H.-F. Tsai, and B.-H. Lin, "Evaluation of a Smart Intercom Microservice System Based on the Cloud of Things," *Electronics (Basel)*, 2023, vol 12, no. 11 p. 2406, doi: 10.3390/electronics12112406.

[12] S. Hassan *et al.*, "From Monolith to Microservices: Software Architecture for Autonomous UAV Infrastructure Inspection," *Softw Pract Exp*, vol. 52, 2022, doi: 10.14569/ijacsa.2017.081236.

[13] J. Kazanavicius *et al.*, "Microservice Identification by Partitioning Monolithic Web Applications Based on Use-Cases," *Softw Pract Exp*, 2022, doi: 10.1002/spe.3141.

[14] A. J. Lauwren, "Microservice and Monolith Performance Comparison in Transaction Application," *Proxies Jurnal Informatika*, 2024, doi: 10.24167/proxies.v5i2.12447.

[15] 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.

[16] A. Barczak and M. Barczak, "Performance comparison of monolith and microservices based applications.", *25th World Multi-Conference on Systemics, Cybernetics and Informatics, WMSCI 2021*, vol. 1, pp. 120–125. International Institute of Informatics and Systemics, IIIS.

[17] M. Seedat, Q. Abbas, and N. Ahmad, "Systematic Mapping of Monolithic Applications to Microservices Architecture," Sep. 2023, [Online]. Available: http://arxiv.org/abs/2309.03796

[18] C. H. Kristantyo, I. A. Putranto, E. S. Soegoto, R. Setiawan, and R. Jumansyah, "Impact of E-Ticketing Application on Bus Transportation in Bandung," *Kne Social Sciences*, 2020, doi: 10.2991/aebmr.k.200108.008.

[19] N. Bumanis, G. Vitols, I. Arhipova, and I. Mozga, "Mobile Ticket Lifecycle Management: Case Study of Public Transport in Latvia," 2017, doi: 10.22616/erdev2017.16.n015.

[20] J. Jordanov and S. K. Jaiswal, "Domain Driven Design Approaches in Cloud Native Service Architecture," *International Journal of Innovative Research in Engineering \& Management*, 2023, doi: 10.18421/tem124-09.

[21] S. K. Jaiswal, "Domain-Driven Design (DDD)- Bridging the Gap Between Business Requirements and Object-Oriented Modeling," *International Journal of Innovative Research in Engineering \& Management*, 2024, doi: 10.55524/ijirem.2024.11.2.16.

[22] R. Mikkilineni, "A New Class of Intelligent Machines With Self-Regulating, Event-Driven Process Flows for Designing, Deploying, and Managing Distributed Software Applications," 2023, doi: 10.20944/preprints202311.1104.v1.

[23] K. Farias and L. Lazzari, "Event-Driven Architecture and REST Architectural Style: An Exploratory Study on Modularity," *Journal of Applied Research and Technology*, 2023, doi: 10.22201/icat.24486736e.2023.21.3.1764.

[24] A. Rahmatulloh, F. Nugraha, R. Gunawan, and I. Darmawan, "Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems," 2022, doi: 10.1109/icadeis56544.2022.10037390.

[25] Ayoubi, "An Event-Driven Service Oriented Architecture Approach for E-Governance Systems," *Kjet*, 2019, doi: 10.31841/kjet.2021.1.

[26] R. Bhattacharya and T. Wood, "BLOC: Balancing Load with Overload Control In the Microservices Architecture," in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, 2022, pp. 91–100. doi: 10.1109/ACSOS55765.2022.00027.

[27] K. G. Sukadharma, "Implementasi CI/CD Pada Microservices Untuk Meningkatkan Availability Pada Pemrosesan Big Data," *Jeliku (Jurnal Elektronik Ilmu Komputer Udayana)*, 2024, doi: 10.24843/jlk.2023.v12.i03.p12.