COMPARATIVE ANALYSIS OF NEXT.JS AND GOLANG BACKEND ARCHITECTURE IN THE CIVIL RECORDS MANAGEMENT SYSTEM

Firman Firdaus*, Tedi Budiman

Institut Pendidikan Indonesia, Garut, Indonesia
Email: *firmanfird23@gmail.com*, tedi1976bdmn@gmail.com*

Article Info

Article history:
Received Mar 30, 2026
Revised Apr 12, 2026
Accepted Apr 30, 2026

ABSTRACT

Sistem Evaluasi Layanan Lengkap Individu dan Catatan Aktivitas (SELLICA) was initially built using a monolithic architecture based on Next.js API routes. However, as the user scale increased, the system experienced performance and efficiency bottlenecks under heavy workloads. This study aims to conduct a comprehensive comparative analysis between an integrated Next.js backend architecture and a separated, service-oriented Go (Golang) backend for government-scale applications. The research method employs quantitative evaluation through performance testing (load testing with K6 and benchmarks) as well as qualitative evaluation of architectural complexity and development experience. The results indicate that migrating to a Go backend provides a response time improvement of 20 to 289 times faster, 20.25 times higher throughput, and memory usage efficiency up to 4 to 5 times lower compared to Next.js. Conversely, Next.js maintains advantages in rapid initial development and prototyping ease. In conclusion, for large-scale systems with stringent performance and concurrency requirements, a Go backend is highly recommended. A hybrid architectural approach, combining a static Next.js frontend with a Go backend, proves to be the most optimal solution in balancing high performance and cost efficiency.

Keywords: *Backend Architecture, Golang, Government System, Next.js, Web Performance*

ABSTRAK

Sistem Evaluasi Layanan Lengkap Individu dan Catatan Aktivitas (SELLICA) pada awalnya dibangun menggunakan arsitektur monolithic berbasis Next.js API routes. Namun, seiring dengan peningkatan skala pengguna, sistem mengalami kendala performa dan efisiensi di bawah beban kerja yang tinggi. Penelitian ini bertujuan untuk melakukan analisis komparatif yang komprehensif antara arsitektur backend Next.js yang terintegrasi penuh dengan backend Go (Golang) terpisah (service-oriented) pada aplikasi berskala pemerintahan. Metode penelitian menggunakan evaluasi kuantitatif melalui pengujian performa (load testing dengan K6 dan benchmark) serta evaluasi kualitatif terhadap kompleksitas arsitektur dan pengalaman pengembangan. Hasil penelitian menunjukkan bahwa migrasi ke backend Go memberikan peningkatan waktu respons sebesar 20 hingga 289 kali lebih cepat, throughput 20,25 kali lebih tinggi, dan efisiensi penggunaan memori hingga 4-5 kali lebih rendah dibandingkan Next.js. Di sisi lain, Next.js tetap menunjukkan keunggulan pada kecepatan pengembangan awal dan kemudahan purwarupa. Kesimpulannya, untuk sistem berskala besar dengan persyaratan performa dan konkurensi tinggi, backend Go sangat direkomendasikan. Pendekatan arsitektur hibrida, yaitu menggabungkan frontend Next.js statis dan backend Go, terbukti menjadi solusi paling optimal dalam menyeimbangkan performa tinggi dan efisiensi biaya.

Kata Kunci: *Arsitektur Backend, Golang, Next.js, Performa Web, Sistem Pemerintahan*

Corresponding Author:
Firman Firdaus
Institut Pendidikan Indonesia
Jl. Terusan Pahlawan No.32, RW.01, Sukagalih, Kec. Tarogong Kidul, Kabupaten Garut, Jawa Barat 44151
Email: firmanfird23@gmail.com

1. INTRODUCTION

The Individual Comprehensive Service Evaluation and Activity Record System (SELLICA) is a large-scale web-based application designed to modernize civil registration and vital statistics systems in Indonesia. As a digital infrastructure that bridges government employees and the public, SELLICA was initially built using a full-stack monolithic architecture with the Next.js framework and integrated API routes. Although this approach provided the advantages of a unified codebase with TypeScript and high initial development speed, as the workload increased to more than 1,000 concurrent users, the system began to face several critical obstacles. These challenges included performance degradation marked by response times reaching 500-2000 milliseconds during peak hours, memory leaks in long-running Node.js processes, and high latency in WebSocket-based real-time communication features.

Several recent studies have compared the performance of monolithic and microservices-based backends in high-scale applications. Pratama & Farisi compared the performance of API backends using PHP, Go, and JavaScript, and found that Go was significantly more efficient under high load, with faster response times and lower memory usage [1]. Marieska et al. benchmarked large-scale transaction systems, showing that microservices architecture can handle higher throughput and lower errors than monolithic architecture [2]. In addition, a study by Dirgantara et al. confirmed that the Docker-based microservices approach is superior to monolithic architecture in terms of system scale and performance management, including real-time communication [3]. A synthesis of these findings highlights the importance of empirical evaluation of backends in the context of large-scale government applications, as decisions regarding architecture and programming language selection have a direct impact on performance and operational cost efficiency.

Previous literature on backend architectures frequently discusses the trade-offs between monolithic and microservices designs. JavaScript-based full-stack frameworks such as Next.js provide efficient data flow and rapid development, but are built on the Node.js runtime, which relies on a single-threaded event loop to handle concurrency. While this model excels at managing asynchronous I/O-bound workloads, its single-thread execution can make it less effective when handling CPU-intensive tasks, as all concurrent request executions are funneled through the same thread and may propagate performance issues across pending operations [4]. On the other hand, compiled languages such as Go (Golang) offer lightweight concurrency models through goroutines, native code execution, and significantly lower memory footprint efficiency [5]. Other research shows that refactoring from monolithic to microservices can have a direct impact on improving response time and scalability, as separate services maximize parallel execution and component deployment automation [6]. Several studies related to web application performance have proven that response time is directly correlated with system reliability, where execution delays are highly intolerable, especially in Indonesia's Electronic-Based Government System (SPBE) standards. Furthermore, contemporary research shows that performance and scalability are not only a function of the programming language, but also the service architecture that supports workload distribution and separation of responsibilities between components [7].

Recent research also evaluates performance after migration from a monolithic architecture to microservices, showing significant changes in metrics such as response time, throughput, and failure rates in real test environments. Wahyudin et al. conducted load, soak, and stress testing on a system migrated from monolithic Laravel to Golang-based microservices and found that the microservices implementation consistently accelerated response times, increased request handling capacity, and substantially reduced the error rate [8]. These results show that migration strategies can have a real impact if they are designed and implemented in a measured manner.

Although many studies have compared various frameworks theoretically or through synthetic benchmarks, there is still very little literature that presents empirical data from architectural migrations in government-scale production applications. Therefore, the scientific novelty of this research lies in the empirical comparative analysis of the real architectural transformation process from an integrated Next.js API routes system to a separate Go-based modular backend while maintaining the Next.js frontend. This research not only measures performance metrics quantitatively, but also evaluates architectural implications and developer experience qualitatively.

However, although recent literature has discussed performance comparisons extensively, there is still a lack of research that empirically tests architectural migration in government-scale production applications, especially those that combine Next.js frontend with Go backend, thus creating a gap that this research will address. This gap includes

empirical evaluation of performance, development complexity, and trade-offs in a large-scale production context, which is the focus of this research.

The main issues in this research are how to overcome performance degradation and scalability limitations in the Next.js monolithic architecture when handling government-level service workloads, and whether migrating to a Go backend can provide a solution that is commensurate with its development complexity. Based on these issues, this study aims to comprehensively measure and compare the quantitative performance differences between Next.js and Go backends, evaluate trade-offs in development and maintenance complexity, and formulate an objective decision framework for government organizations or institutions in choosing the backend architecture that best suits the scale and needs of the project.

2. RESEARCH METHODS

This research was conducted through several main stages, namely: (1) experimental design and determination of test variables, (2) configuration and implementation of two backend architectures to be compared, (3) performance testing using structured load scenarios, (4) collection and analysis of quantitative and qualitative data, and (5) interpretation of results to develop architectural recommendations. This research used a mixed methods approach that combined quantitative performance analysis with qualitative architectural evaluation and development experience.

The research was conducted in the SELLICA (Civil Registry Electronic System) system environment by testing and comparing two main backend architectures: an integrated Next.js framework (based on Node.js v20.x, Next.js 15.3.0, and Supabase Client 2.52. 1) and a separate Go-based backend (version 1.23.0 with the Gin 1.10.0 framework).

The testing infrastructure was run on main hardware specifications consisting of an Intel/AMD 8-core processor with 16GB DDR4 RAM, operating on the Linux Ubuntu 22.04 / Windows 11 operating system in a local network environment with minimal latency. The choice of performance testing tools was decided based on a comparative study of modern load testing tools commonly used in software engineering research. Sulistiyani & Rosul showed that k6 and tools such as JMeter and Gatling are among the top tools for evaluating performance metrics such as response time and system resource utilization [9]. Quantitative performance measurements were executed using K6 Load Testing software to progressively simulate concurrent user loads, and Go Native Benchmarks (go test -bench=. -benchmem) were utilized to obtain metrics at the basic function level.

The load testing methodology refers to a scientific method library that discusses systematic steps in load scenario design, test execution, and performance result interpretation to determine system scalability and stability limits [10]. The load testing procedure was carried out progressively through four main scenarios to measure system resilience. The first test is a light load simulated with 25-50 concurrent users for 30 seconds to obtain baseline metrics. The second test is a normal load run with 100-200 concurrent users for 60 seconds to simulate traffic levels during operational hours. The third test is a heavy load conducted using 500 concurrent users for 90 seconds to represent the peak demand for government services. Finally, a stress test is executed involving more than 1000 concurrent users for 120 seconds to identify the breaking point and recovery limits of each architecture.

Research by Neelapu confirms that to measure application performance comprehensively, monitoring metrics such as response time, throughput, and memory and CPU utilization is very important because it provides a clear picture of the resilience and efficiency of a system under high load [11]. The performance benchmarks collected from the test results were then extracted and processed to compare several critical parameters. These parameters include: response time measured in percentiles (Average, P50, P95, and P99), throughput or requests per second (RPS), error rate, and server resource utilization, which includes main memory usage (MB) and CPU utilization rate (%). In addition to quantitative testing, qualitative evaluation was conducted by measuring lines of code, the complexity of cache and WebSocket system integration, and the time taken by the development team to implement features in order to assess the maintainability of both architectures.

To ensure validity and reliability, each test scenario was run three times and analyzed using average values to minimize bias due to system fluctuations. During testing, monitoring was performed to ensure there were no external anomalies. Quantitative data was then analyzed descriptively and comparatively by comparing the averages and percentile distributions between architectures to identify practical performance differences [11].

3. RESULTS AND DISCUSSION

3.1 Quantitative Performance Analysis (Response Time and Throughput)

Baseline testing results show significant performance differences between Next.js and Go architectures under the same operating conditions. The average response time measurement for the Go backend was recorded in the range of 1.7 to 28 milliseconds, which is far superior to Next.js, which requires 500 to 2000 milliseconds. The Go backend throughput is also consistently higher than Next.js. These findings are consistent with a study by Jarmoszewicz et al., which shows that microservices with efficient communication protocols result in lower latency and higher throughput compared to traditional monolithic systems, especially under high load [12].

A comparison of the overall performance data can be seen in Table 1.

Table 1. Comparison of Performance Metrics Between Next.js and Go Backends

Metrics	Next.js (Node.js)	Go (Golang)	Improvements
Response Time (Average)	500–2000 ms	1,7–28 ms	20–289× faster
Throughput (RPS)	20–50 RPS	126–405 RPS	20.25× higher
Memory Usage	200–500 MB	50–100 MB	4–5× more efficient
Concurrent User Capacity	50–100	1000+	10× larger

The superior performance of the Go backend is consistent with other research findings that show that microservices or separate backend architectures often result in better response and throughput performance than traditional monolithic architectures under high load [2].

These results are in line with research by Pratama & Farisi, which shows that Golang has lower response times than JavaScript (Node.js) when tested under high API loads [1]. The study confirms that the characteristics of a natively compiled language provide a significant advantage in processing concurrent requests compared to interpreter-based or JIT runtime environments.

Furthermore, the response and throughput data across load scenarios (light, normal, heavy, stress) show a stable linear pattern for the Go backend, while Next.js experiences a drastic performance degradation in heavy and stress scenarios.

These findings are also reinforced by research by Effendy et al. comparing Node.js and Golang in database-based backend testing, where Golang maintained better latency stability as the number of connections increased [13]. In that test, Node.js showed an exponential increase in response time when concurrency exceeded a certain threshold, a phenomenon also seen in the SELLICA system.

Furthermore, Azzahidi et al. in their evaluation of modern REST API frameworks (Express.js vs Gin) found that the Go-based framework (Gin) consistently produced higher throughput and lower memory usage compared to Express.js [14]. These results provide empirical comparisons that support the quantitative results of this study.

In addition to the average values, the response percentile distribution (P50, P95, and P99) shows increasingly significant differences under high loads. In the Go architecture, the difference between P50 and P99 is relatively small, indicating latency stability despite a surge in the number of users. In contrast, in Next.js, the P99 value increases sharply compared to P50 in heavy and stress load scenarios, indicating latency spikes and request queuing. This pattern shows that response variability in Next.js is much higher than in Go, which has the potential to reduce service quality during peak hours.

This phenomenon of increased tail latency was also reported by Azzahidi et al., where Express.js showed a significant high percentile deviation compared to Gin when concurrency increased [14]. The study emphasized that stability at P95 and P99 is more crucial in production systems than just the average value, as it reflects the user experience in the worst-case scenario.

In the context of public service systems such as SELLICA, tail latency stability is an important factor because demand is not always evenly distributed. Public access patterns tend to be volatile, especially as administrative deadlines approach. Therefore, the ability of the Go backend to maintain consistent latency distribution demonstrates a structural advantage in dealing with unexpected loads.

3.2 Interpretation of Performance Differences

Scientific findings from the data in Table 1 show a drastic difference between the two backend architectures. Scientifically, this disparity stems from fundamental differences in the execution and concurrency models of the technologies used. Next.js runs on Node.js, which uses a single-threaded event loop model based on asynchronous I/O. In other research on backend performance, monolithic architectures such as those commonly used in Node.js also show increased latency and

error rates under high loads if not balanced with strong parallelism strategies [2].

Vilhelmsson show that event-driven architectures such as Node.js have advantages in I/O-bound tasks, but suffer limitations when faced with CPU-bound workloads due to their reliance on a single main event loop [15]. This explains why, in encryption and data-intensive processing tests, Next.js's response time increased significantly.

In contrast, Go is a language that compiles directly into native machine code and has a concurrency model using goroutines, which are lightweight threads that are multiplexed into a small number of operating system threads. This approach helps the Go backend maximize the efficient utilization of all processor cores, as seen in CPU utilization measurements showing that Go utilizes 45–60% per core, while Next.js is often constrained to a single core.

These findings are in line with other studies showing that the choice of architecture and execution model greatly contributes to system performance stability, especially in the context of web backends under heavy load.

Pragestu et al. in their analysis of the scalability of Apache Tomcat, Node.js, and Go web servers, also reported that Go is able to maintain more stable performance on HTTP and MQTT protocols when concurrency increases progressively [16]. This indicates that Go's built-in concurrency model is more adaptive to increases in the number of simultaneous connections than the single-threaded approach.

When analyzed further from a system architecture perspective, this performance difference is also influenced by the memory management and garbage collection models. Node.js, which runs on the V8 Engine, relies on an automatic garbage collection mechanism that can trigger pause time when the memory cleanup cycle takes place. Under high loads with many temporary objects, the frequency of garbage collection increases and contributes to latency spikes. In contrast, Go has a garbage collector optimized for low-latency environments with a concurrent mark-and-sweep approach, resulting in relatively smaller stop-the-world pauses under intensive loads.

These findings are in line with recent research showing that the choice of programming language and its runtime environment significantly affects performance and scalability in distributed, cloud-native systems. Empirical comparative studies demonstrate that languages optimized for lightweight concurrency (e.g., Golang) and efficient memory management tend to achieve higher throughput and stability under load compared to alternatives such as Node.js [17], [18]

In addition to technical language factors, architectural design also affects load distribution patterns. In a monolithic approach, all business logic, authentication, and database access run in one main process. This increases the risk of contention when one module experiences a bottleneck. Conversely, a service-oriented approach allows for workload isolation, so that a failure in one service does not directly interfere with other modules. This principle is in line with the findings of Marieska et al. regarding fault isolation in microservices [2].

3.3 Resource Utilization and Scalability Stability

In peak load testing (stress test), the Go backend maintained a request success rate of 98% with an error rate close to 0%. On the other hand, Next.js API routes experienced an increase in errors of up to 35%, and in some repetitions even caused the server to crash. This phenomenon has also been reported in other contexts, where optimized microservices are able to maintain more consistent performance than monolithic ones when the load continues to increase [2].

Marieska et al. in a comparative study of monolithic and microservices architectures in high-volume transaction systems, found that microservices exhibit better fault isolation, so that a failure in one service does not directly affect the entire system [2]. This is relevant to the stability of the modularly designed Go backend in this study.

Additional testing of real-time aspects and WebSocket communication also supports these findings in a WebSocket benchmarking study, Go libraries such as gorilla/websocket showed higher scalability efficiency compared to similar libraries in Node.js, especially when the number of simultaneous clients increased dramatically [17].

Fernando & Engel report that Go-based WebSocket implementations maintain lower latency and more efficient memory consumption compared to Socket.io on Node.js when the number of connections exceeds thousands of clients [17]. These findings reinforce the results of real-time communication testing on the SELLICA system.

Horizontal scalability is an important indicator in this test. The Go backend demonstrated the ability to maintain performance for more than 1000 concurrent users without significant degradation. This shows that the goroutine-based concurrency approach is capable of handling thousands of connections with minimal overhead. Recent research highlights that the choice of programming language and architectural style can significantly affect performance and resource utilization in

distributed service-oriented systems. Comparative analysis of programming languages in microservices reveals that compiled languages such as Go often achieve superior efficiency and scalability compared to interpreted counterparts [18].

Furthermore, the stress test results show that the error rate in Next.js increases progressively before crashing, while Go maintains a high success rate until it approaches the system capacity limit. This pattern indicates that Go-based systems have more gradual performance degradation (graceful degradation), which is an important characteristic in critical systems such as government services.

The concept of graceful degradation in distributed systems has been highlighted in recent comparative research on software architectures. Empirical evidence shows that microservices architectures are able to isolate faults and maintain partial service availability even when individual components fail, whereas monolithic systems tend to exhibit cascading failures that impact the entire application. This resilience is supported by mechanisms such as circuit breakers and autonomous fault isolation inherent in service-oriented designs [19], [20].

3.3 Architecture Evaluation and Development Experience (Developer Experience)

From a developer experience perspective, Next.js shows an initial advantage in development productivity thanks to the use of TypeScript, code sharing, and integrated monolithic development flows. However, as complexity increases, the Go service-oriented backend approach shows advantages in terms of maintainability and explicit error handling, which is in line with software architecture literature that emphasizes modularity to reduce technical debt in large systems [2].

Redha in a study on Point of Sales systems reported that microservices architecture requires longer initial development time but provides better flexibility and scalability in the long run [21] . This is consistent with the experience of the SELLICA development team, which saw an increase in system stability after migrating to a separate Go backend.

Furthermore, Szewczyk & Skublewska-Paszkowska stated that the choice of backend framework has a significant effect on REST API maintainability and ease of scaling [22]. The study emphasized that frameworks with a modular structure and static compilation tend to result in lower defect rates in the post-deployment phase.

From a Quality of Service (QoS) perspective, parameters such as latency, throughput, and availability are interrelated in determining user satisfaction. In public service systems, Service Level Agreement (SLA) standards generally require response times below a certain threshold so as not to disrupt administrative processes. The results of the study show that the Go backend consistently falls well below the 100 ms threshold, while Next.js exceeds 500 ms under high loads and even reaches 2000 ms.

According to Stabili microservices-based systems with lightweight concurrency are better able to meet SLAs in elastic cloud environments than traditional monolithic architectures [23]. These findings support the empirical results of the SELLICA system, particularly in the context of readiness to handle spikes in public demand.

3.4 Infrastructure Impact and Return on Investment

In a reverse migration simulation, the full monolithic approach showed a negative ROI due to greater horizontal scaling requirements and increased infrastructure costs. This is consistent with the recommendations found in exploratory studies that suggest adapting microservices for high-scale systems in order to reduce long-term operational costs [2].

Lauwren in a transaction application study concluded that although monolithic architecture is simpler in the early stages, infrastructure costs increase significantly when system load grows exponentially [24]. With service separation, load distribution can be done more efficiently, thereby reducing excess resource requirements. These findings reinforce the projected 59% cost increase in the SELLICA system simulation if the Next.js-based monolithic approach is maintained. Therefore, a hybrid architecture approach—combining a Next.js frontend with a separate Go backend—is the most technically and economically rational solution for large-scale government services.

In terms of infrastructure efficiency, the lower memory usage of the Go backend (50–100 MB) compared to Next.js (200–500 MB) has a direct impact on the number of server instances required. With more efficient memory consumption, a single physical or virtual machine can accommodate more service instances without having to perform aggressive horizontal scaling. This reduces monthly operational costs, especially in pay-as-you-go cloud environments.

Dirgantara et al. research comparing monolithic and Docker-based microservices architectures shows that microservices provide more precise scaling flexibility, reducing resource waste by up to 30% in dynamic loads [3]. These findings are in line with the projected 59% cost increase in the SELLICA

system reverse migration simulation if a fully monolithic approach is maintained.

Considering factors such as performance, stability, QoS, and cost efficiency, a hybrid architecture approach is the optimal compromise solution. Next.js continues to be utilized for development speed and interface rendering, while Go handles core business processes that require high stability and large concurrency. This model reflects modern architectural practices that strictly separate the presentation layer and business logic layer to maximize overall system efficiency.

4. CONCLUSIONS

Based on the results of the comparative analysis that has been conducted, it can be concluded that the separate Go (Golang) language-based backend architecture is significantly superior to the monolithic Next.js API routes architecture in handling government-scale application workloads. The migration from Next.js to Go successfully addressed performance degradation and scalability limitations through the implementation of the goroutines concurrency model and optimal memory management efficiency. The decision to use Go resulted in substantial infrastructure cost savings and ensured system reliability (zero error rate) under peak loads, making it a highly recommended architectural choice for applications with high concurrency needs and strict response time requirements.

Based on the research results, it can be concluded that a separate Go (Golang)-based backend architecture is significantly superior to the monolithic Next.js architecture in handling government-scale application workloads. The Go backend is capable of maintaining low response times, high throughput, and near-zero error rates even under peak loads, ensuring system reliability under heavy operational conditions. The migration from Next.js to Go successfully addressed the performance degradation and scalability limitations experienced by the previous system, while also providing substantial infrastructure cost savings. On the other hand, Next.js still provides advantages in early development and prototyping thanks to its unified code base and ease of prototyping.

These results show that combining both technologies in a hybrid architecture is the most optimal approach: Next.js can be utilized for the frontend to maximize user experience, while the Go backend handles all business logic, database integration, and real-time synchronization. This approach not only improves overall system performance but also provides a flexible foundation for further development.

Furthermore, this research shows that implementing a Go backend can reduce long-term maintenance overhead, lower the risk of bugs, and minimize the need for hotfixes, thereby providing significant benefits to operational reliability. This hybrid system also opens up opportunities for real-time analytics implementation, automated performance monitoring, and the integration of Artificial Intelligence and Machine Learning technologies for automated document processing and data-driven decision making.

As a practical implication, these conclusions support the use of modular and service-oriented architectures for large-scale government applications, enabling increased scalability, resource efficiency, and better ROI. This study suggests that the evolution of the system towards full microservices and the adoption of modern communication protocols such as gRPC could be the next strategic step to improve flexibility and integration capabilities with external services.

REFERENCES

[1] F. Pratama and A. Farisi, “Analisis Perbandingan Kinerja Backend API Menggunakan PHP, Golang, dan JavaScript,” *Techno.Com*, vol. 24, no. 1, pp. 153–165, Feb. 2025, doi: 10.62411/tc.v24i1.12080.

[2] 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 (Reka yasa Sist. dan Teknol. Informasi)*, vol. 9, no. 3, pp. 594–600, Jun. 2025, doi: 10.29207/resti.v9i3.6183.

[3] D. P. Dirgantara, D. S. Kusumo, and R. G. Utomo, “Docker-Based Monolithic And Microservices Architecture Performance Comparison,” *J. Tek. Inform.*, vol. 5, no. 2, pp. 357–365, Apr. 2024, doi: 10.52436/1.jutif. 2024.5.2.1338.

[4] H. M. Kabamba, M. Khouzam, and M. R. Dagenais, “Vnode: Low-Overhead Transparent Tracing of Node.js-Based Microservice Architectures,” *Futur. Internet*, vol. 16, no. 1, p. 13, Dec. 2023, doi: 10.3390/fi16010013.

[5] A. A. A. Donovan and B. W. Kernighan, *The Go* programming *language*. Addison-Wesley Pro fessional, 2015.

[6] S. F. Yusri, D. D. J. Suwawi, and M. Adrian, “Analysis The Impact Of Refactoring From Monolithic Applications To Microservices On Response Time Using The Mda And Sca Approaches,” *J. Tek. Inform.*, vol. 5, no. 6, pp. 1861–1872, Dec. 2024, doi: 10.52436/1.jutif. 2024.5.6.2617.

[7] V. Chernohor, "Performance and scalability analysis of monolithic and microservice architectures in social networks," *J. Comput. Sci. Inst.*, vol. 37, pp. 405–409, Dec. 2025, doi: 10.35784/jcsi.7939.

[8] A. Wahyudin, H. Siregar, A. Anisyah, S. M. Kusumawardani, and Erlangga, "Migrating Monolithic to Microservices: Comparative Performance Testing Evaluation Between Architectures," *Sci. J. Informatics*, vol. 12, no. 4, pp. 797–816, Jan. 2026, doi: 10.15294/ sji.v12i4.28499.

[9] E. Sulistiyani and Q. M. Rosul, "Analisis Perbandingan Kinerja Alat Pengujian Beban Perangkat Lunak: Apache JMeter, Gatling, dan K6," *J. Inform. Polinema*, vol. 12, no. 2, pp. 393–400, Feb. 2026, doi: 10.33795/jip.v12i2.8693.

[10] D. M. D. Amit, "Performance Testing: Methodology for Determining Scalability of Web Systems," *Int. J. Sci. Res.*, vol. 13, no. 1, pp. 1254–1261, Jan. 2024, doi: 10.21275/ SR24121010827.

[11] M. Neelapu, "The Effectiveness of Load and Performance Testing on Application Scalability," *ESP J. Eng. Technol. Adv.*, vol. 4, no. 3, pp. 171–180, 2024, doi: 10.56472/25832646/JETA-V4I3P118.

[12] J. Jarmoszewicz, P. Iwanowski, and M. Plechawska-Wójcik, "Analysis of the performance and scalability of microservices depending on the communication technology," *J. Comput. Sci. Inst.*, vol. 33, pp. 323–330, Dec. 2024, doi: 10.35784/jcsi.6499.

[13] F. Effendy, Taufik, and B. Adhilaksono, "Performance Comparison of Web Backend and Database: A Case Study of Node.JS, Golang and MySQL, Mongo DB," *Recent Adv. Comput. Sci. Commun.*, vol. 14, no. 6, pp. 1955–1961, Oct. 2021, doi: 10.2174/2666255813666191219104133.

[14] A. S. Azzahidi, B. Wijayanto, and A. Darmawan, "Performance Evaluation of Backend Frame works for REST API: A Comparative Study of Spring Boot, Flask, Express.js, Laravel FrankenPHP, and Gin," *J. Tek. Inform.*, vol. 6, no. 4, pp. 2405–2419, Aug. 2025, doi: 10.52436/1.jutif.2025.6.4.4811.

[15] I. Vilhelmsson, "A performance comparison of an event-driven node. js web server and multi-threaded web servers," 2021. [Online]. Available: https://www.diva-portal.org/smash/ record. jsf? pid=diva2:1605998

[16] S. Pragestu, H. Sujaini, and E. F. Ripanti, "Analisis Skalabilitas Web Server Apache Tomcat, Node. Js Dan Go Pada Protokol Hypertext Transfer Protocol (HTTP) Dan Message Queue Telemetry Transport (MQTT)," *JUSTIN (Jurnal Sist. dan Teknol. Informasi)*, vol. 11, no. 4, pp. 605–611, 2023, doi: 10. 26418/justin.v11i4.71607.

[17] L. Fernando and M. M. Engel, "Comparative Performance Benchmarking of WebSocket Libraries on Node.js and Golang," *sinkron*, vol. 9, no. 4, pp. 2051–2060, Oct. 2025, doi: 10.33395/ sinkron.v9i4.15266.

[18] S. Hussein, S. J. Abbas, G. F. Ali, N. K. Hadi, and M. M. M. Maadi, "A Comparative Analysis of Programming Languages Used in Microservices," *Int. J. Comput. Exp. Sci. Eng.*, vol. 11, no. 1, Feb. 2025, doi: 10.22399/ ijcesen.977.

[19] M. Bhandari, "Microservices vs. Monolithic Architectures in Real-Time Distributed Systems: A Comparative Analysis," *J. Inf. Syst. Eng. Manag.*, vol. 10, no. 62s, pp. 496–503, Nov. 2025, doi: 10.52783/jisem.v10i62s.13614.

[20] S. S. A. Yalamati, "Resilient microservice patterns using Java 17 and Spring Boot 3.2 in Cloud-Native Systems," *Int. J. Sci. Res. Arch.*, vol. 16, no. 3, pp. 431–447, Sep. 2025, doi: 10.30574/ijsra.2025. 16.3.2559.

[21] F. S. Redha, "Performance Comparison of Web Services Built Using Monolithic Architecture and Microservices on Point of Sales Systems," *JATISI*, vol. 10, no. 1, pp. 406–420, 2023, doi: 10.35957/jatisi.v10i1.3210.

[22] M. Szewczyk and M. Skublewska-Paszkowska, "Performance comparison of development frameworks in selected environments in REST API architecture," *J. Comput. Sci. Inst.*, vol. 35, pp. 121–128, Jun. 2025, doi: 10.35784/jcsi.7041.

[23] D. Stabili, R. Romagnoli, M. Marchetti, B. Sinopoli, and M. Colajanni, "A multidisciplinary detection system for cyber attacks on Powertrain Cyber Physical Systems," *Futur. Gener. Comput. Syst.*, vol. 144, pp. 151–164, Jul. 2023, doi: 10.1016/j.future.2023.02.019.

[24] A. J. Lauwren, "Microservice and Monolith Performance Comparison in Transaction Application," Universitas Katholik Soegijapranata Semarang, 2022. [Online]. Available: https://repository.unika.ac.id/28277/