Scientific Journal of Informatics
Vol. 12, No. 4, November 2025

p-ISSN 2407-7658 https://journal.unnes.ac.id/journals/sji/index e-ISSN 2460-0040

Migrating Monolithic to Microservices: Comparative Performance Testing Evaluation Between Architectures

Asep Wahyudin[1]*, Herbert Siregar[2], Ani Anisyah[3], Sekar Madu Kusumawardani[4], Erlangga[5]
[1, 2, 3, 4, 5]Department of Computer Science, Universitas Pendidikan Indonesia, Indonesia

Abstract.
Purpose: Various organizations and industries have migrated their systems from previously adopting monolithic architecture to microservices architecture. One of the advantages of microservices architecture that is desired to be achieved from the migration process is the performance side. Therefore, this research aims to conducted performance testing on the system that was migrated from monolithic to microservices in the previous study.
Methods: This research was conducted in several stages, such as designing and implementing software and applications, creating performance testing scenarios, executing scenario testing with load testing, stability testing (soak testing), and stress testing such as load testing, soak testing, and stress testing, and finally analyzing and reporting testing results using performance indicator in terms of response time, throughput, and error rate.
Result: The test results showed a significant increase in performance before and after the migration of the monolithic system to microservices. Application response time became faster, more requests could be handled, and the failure rate experienced by the system was smaller. This shows that system performance is better with the implementation of microservices architecture.
Novelty: This research presents a novelty in the form of a comparative evaluation of real deployment-based system performance between Laravel monolithic architecture and Golang gRPC-based microservices on the same application, with a seven-stage performance testing approach and the use of in-depth quantitative metrics using Apache JMeter.

Keywords: Performance testing, Monolithic architecture, Microservices architecture, Software migration testing
Received June 2025 / **Revised** November 2025 / **Accepted** December 2025

INTRODUCTION

There are a number of software architectures that are changing to better fit the needs and features of system applications [1]. The goal is to get the most out of applications in terms of performance, scalability, flexibility, complexity, maintainability, and resilience. More and more, monolithic design is becoming less popular in favor of different architectures [2]–[4]. This architecture is simple and easy to use in small-scale settings [5]. However, as application development has become more popular, monolithic architecture has become less scalable, less performant, less fault-tolerant, and harder to maintain [3], [6]–[9]. Because of this, a lot of companies, like Uber, SoundCloud, Netflix, Amazon, eBay, and Spotify, moved their monolithic apps to microservices [2], [10]–[15].

Several studies have shown that moving from a monolithic system to microservices can make the system better in many ways, such as by making it run faster [14]–[20]. As an increasing number of individuals move from monolithic architectures to microservices, we can compare how well both designs work in the system migration we did for the same application. We do performance testing and look at the results based on a set of non-functional requirements, like figuring out how fast the system responds, how much data it can handle, and how reliable it is under a certain workload [21]. When the reaction time is low and the throughput is high, the application works better [22].

Monolithic and microservice architectures offer contrasting blueprints for organizing web applications, each bringing distinct advantages and disadvantages in terms of performance, scalability, maintenance, and day-to-day operational complexity. Monolithic designs combine the user interface, business logic, and data layers into a single, self-contained unit, which facilitates rapid in-process communication, easier deployment, and minimal overhead during early development [23], [24]. When studied in controlled

*Corresponding author.
Email addresses: away@upi.edu (Wahyudin)
DOI: 10.15294/sji.v12i4.28499

laboratory settings as well as typical cloud configurations, monolithic applications typically deliver higher throughput and lower latency compared to their microservice counterparts [23], [25]. For example, Wang and colleagues note that monolithic systems outperform microservices by about 6 percent in experiments with many connections, a gap that is partially closed by service discovery frameworks such as Consul [26]. Similarly, findings published in ACM SIGAPP show that monolithic architectures outperform microservices in CPU-constrained workloads on both on-premises and in Azure, especially when the majority of processing time is spent executing business rules [23]. However, these benefits do not eliminate the inherent scalability limitations of monoliths; adding capacity requires cloning the entire application, a wasteful approach when only certain components face spikes in demand [4], [27]. Additionally, interconnected codebases can hamper the pace of feature releases and amplify the propagation of errors—a modification in one module may inadvertently break behavior in another module, triggering failures far beyond the intended scope [28]. As an application's footprint grows, monoliths often accumulate technical debt that resists clean refactoring and makes it difficult to experiment with different programming languages or frameworks [29].

A microservice architecture breaks a large application into smaller, independently deployable services, each of which handles a specific business function [30], [31]. This separation increases modularity, making it possible to scale a single service, release updates, or choose a different technology stack without affecting the entire system [4], [27]. One major financial platform migration reported tighter internal cohesion, lower CPU and memory overhead, and clearer boundaries between services after the monolith was split [28]. A long-term study by Lenarduzzi and colleagues shows that, while moving to microservices briefly accelerates technical debt, the rate of new debt accumulation is subsequently slower than with a maintained monolith [29]. However, these benefits do not come without costs. Combining multiple microservices turns an application into a distributed system that requires service discovery, API gateways, container orchestration such as Kubernetes, and a shared dashboard for logging and tracing [32], [33]. All of these moving parts add overhead, especially additional network hops, which can hinder performance when workloads are already CPU-bound [23], [25]. Keeping different services in step, versioning their contracts, and running thorough end-to-end testing can push teams toward richer CI/CD pipelines and tighter operational habits [23], [27].

The choice between monolithic and microservice architectures ultimately depends on the planned scale of the system, the skills of the development team, the performance characteristics required, and the long-term roadmap for maintaining and evolving the software. For new products or medium-sized applications, monoliths still make sense because their single code base is easier to maintain and tends to perform better from the start [23], [24]. In contrast, very large systems that grow rapidly benefit from microservices, because independent teams can scale, deploy, and maintain modules with minimal disruption, even if that independence adds complexity to monitoring and operations [4], [27]. Future empirical work should quantify this trade-off across industries, particularly by identifying the system sizes and levels of infrastructure maturity that truly warrant a migration to a microservice model.

Performance testing is usually broken down into three types: load testing, soak/stability testing, and stress testing. Load testing involves continuously testing the system by introducing traffic or load within specific limits that are gradually increased. This helps to find the system's capacity and load limits. Soak/stability testing checks how the system performs over a long period to ensure it can handle requests consistently and identify any errors that may arise over time. Stress testing applies maximum load and pressure to a system until it fails. The aim is to find out which conditions and levels of load lead to failure. Furthermore, there are several metrics that are most commonly used to evaluate system performance, namely response time (the time required for the system to respond to a client request, which is determined by the start of the request to the end of the response from the system received), throughput (the number of requests that the system can handle in a certain unit of time), and error rate (the number of transactions that are successfully and failed to be handled) [6], [30], [31].

Performance testing was carried out on the UPI Tracer Study application in both monolithic (pre-migration) and microservices (post-migration) configurations. The UPI Tracer Study monolithic application was built with Laravel technology, while the microservices application was created with Golang gRPC technology. Application development using the Go language is ideal for building API Gateways in microservices architectures due to its high performance, low latency, and efficient concurrency support through goroutines. Compared to other languages such as Node.js, Python, or Java, Go offers smaller memory usage and very fast startup times. This makes API Gateways more responsive, scalable, and resource-efficient in cloud-native environments. Furthermore, the mature Go ecosystem supports the creation of stable, lightweight, and easily deployable gateways.

Testing was carried out with the Apache JMeter tool, which is an open-source desktop-based testing software developed in Java that supports a variety of API connection protocols, including REST, SOAP, and gRPC.

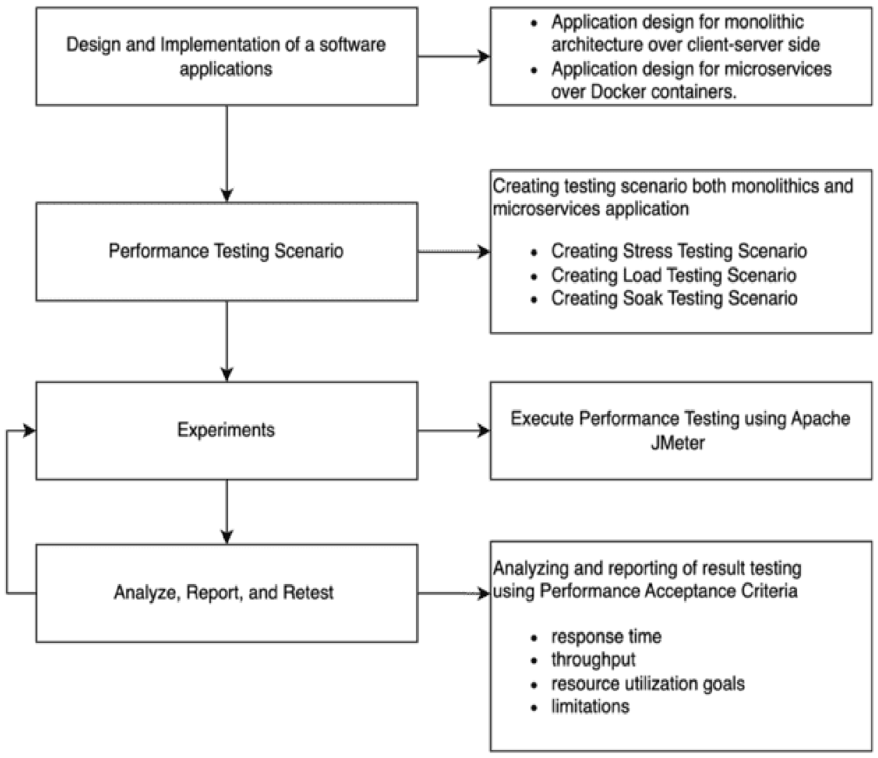
RELATED RESEARCH

The following is Table 1 as a list of research related to the research conducted. Several previous studies have been conducted with different objectives and performance tests. Based on previous research, it can be concluded that the differences between this study and previous studies in the type of testing, measurement parameters, and application development approach. This study not only focus on one type of testing, but also includes three main types of testing such as load testing, soak testing, and stress testing to conduct more comprehensive performance evaluation result. In addition, system performance measurements in this study include various important metrics such as response time, throughput, resource utilization, goals, limitations, and error rates, thus providing a more detailed picture of system performance under various load conditions. Another difference is in the application development approach, where this study uses a microservice architecture with REST API and gRPC API implementations as a comparison to analyze communication efficiency between services, which has not been widely done in previous studies.

Table 1. List of related research

Contributions	Architecture/Method	Output	Matric	Author
Testing the feasibility and benefits of the MMSD (Microservices Migration using the Strangler pattern and DDD) approach on a monolithic Green Button application. [16]	- Migration: Strangler Fig Pattern and Domain-Driven Design (DDD) - Testing: Performance Testing with first AUT (Application Under Test)	The Green Button system that has been migrated to MMSD has higher reliability and a lower error rate compared to the monolithic Green Button system.	- Average (ms) - Error (%) - Throughput (sec) - Average Response Size (byte)	Li, C., Ma, S., Lu, T. (2020)
Improve scalability and performance of Open Library applications by migrating applications from monolithic to microservices. [34]	Method: Domain-Driven Design (DDD) dan Strangler Fig Pattern Performance Testing: load test, stress test, spike test, endurance test, scalability test and volume test	Improved application performance and increased freedom for organizational developers in choosing development technologies	- Average Response Time - Failure Request - Peak User	Ferdinand, J., Syahrina, A., & Musnansyah A. (2021)
Refactoring the SmartCampus application from monolithic to microservices to improve application performance. [7]	- Refactoring: Domain decomposition and Strangler Pattern. - Testing: Performance Testing : load testing, spike testing, stress testing, and soak testing	The SmartCampus system implements a microservices architecture and performs better than previous monolithic systems.	- Average Response Time (ms) - Latenncy (ms) - Throughput	Riyanto, Hermadi, I., Nurhadryani, Y. (2023)
Analyzing the practical aspects of using REST APIs and gRPC for communication processes within applications in a microservices-based ecosystem. [14]	Performance Testing: load testing, stress testing, endurance testing, spike testing, and volume testing	The research results show that both technologies (REST and gRPC API) are excellent for intra-service communication within a microservices ecosystem.	Response Time	Paszkiewicz, A., Ganzha, M., Paprzycki, M., Sowinski, P., Lacalle, I., Palau, C. (2022)

METHODS



Design and Implementation of a software applications
Application design for monolithic architecture over client-server side
Application design for microservices over Docker containers.
Performance Testing Scenario
Creating testing scenario both monolithics and microservices application
Creating Stress Testing Scenario
Creating Load Testing Scenario
Creating Soak Testing Scenario
Experiments
Execute Performance Testing using Apache JMeter
Analyze, Report, and Retest
Analyzing and reporting of result testing using Performance Acceptance Criteria
response time
throughput
resource utilization goals
limitations

Figure 1. WorkFlow diagram of the process research conducted, methods, and evaluation

This section explains the research process used to compare the performance of monolithic and microservices systems. It begins by outlining the application architectural design used for the evaluation. This section covers scenario development, performance testing, data collection, experiments, and preliminary results. Finally, it discusses the performance results of testing tools and their employability based on performance indicators. Figure 1 depicts a workflow diagram for the research process, techniques, and evaluation.

Design and implementation of a software application

Application design for monolithic architecture over client-server side

The system architecture diagram of the Tracer Study UPI provides a high-level representation of the software components, emphasizing their functions, implementations, and interrelationships. As illustrated in Figure 2, the diagram depicts the integration of servers, web servers, databases, and the application of specific design patterns within the Tracer Study UPI system. The architecture is structured as a monolithic system, in which all core functionalities are consolidated within a single deployment unit. Analyzing this architectural model reveals the role of each component in supporting the system's functionality, as well as the flow of data and the operational processes within the system.

The application architecture integrates both client-side and server-side components within a single, unified codebase. All services within the application access a common database, reflecting a shared data layer. Core functionalities such as configuration management, logging, and task scheduling are centralized within the application. Since the services are developed as part of the same project, they utilize a consistent technology stack. Furthermore, the services exhibit a high degree of interdependence; thus, failures, updates, or modifications in one service—or changes to the database—can propagate and affect other components. The system is deployed as a monolithic application, running within a single process. Consequently, when one service experiences increased traffic, the system must be scaled by replicating the entire application instance, rather than scaling individual services independently.

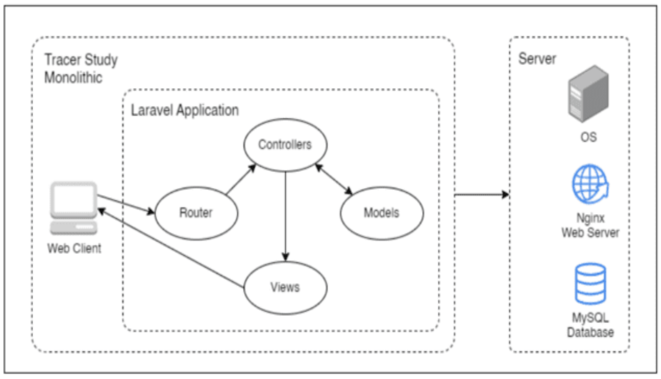


Figure 1. Monolithic architecture

Application design for microservices architecture over docker containers

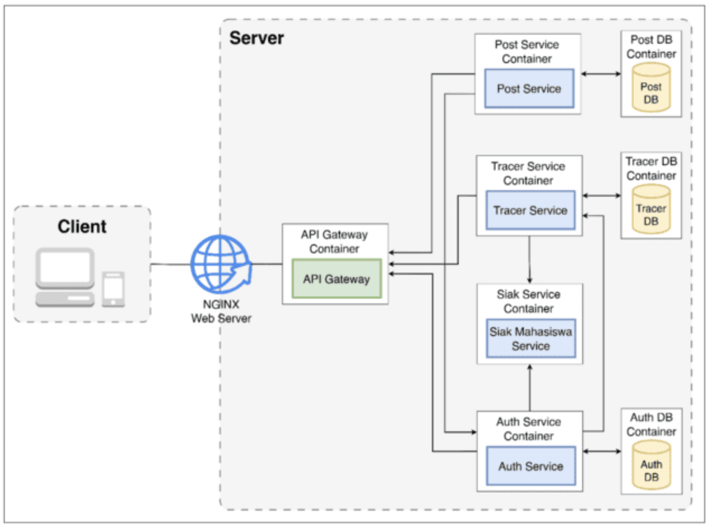


Figure 3. Microservices architecture

The design of the microservices architecture is represented through a comprehensive architectural diagram (Figure 3), which outlines the individual services, their associated databases, inter-service relationships, and the interactions between microservices, clients, and third-party services. This diagram illustrates the communication among several predefined services, including the Post Service, Tracer Service, SIAK Service, and Auth Service. Additionally, two new services—Data Pipeline Service and API Gateway—have been introduced to facilitate the system's migration from a monolithic to a microservices architecture. The Data Pipeline Service is responsible for synchronizing data between the monolithic database and the new Tracer database. This synchronization process encompasses a sequence of operations such as data copying, transformation, cleaning, validation, and normalization.

The development result of architecture are packaged using containers that carried out using Docker. Each service represents a single container, and each database is also created as a separate container. This method aims to simulate the implementation of microservices deployed on different servers. Containerization isolates services so that the resources and environments of each service are not mixed, and it also isolates operating system failures that occur in one service from those of another.

Containers on the server are run using the base Docker image of each service that has been pushed to Docker Hub. Each service runs as a separate container, as shown in Figure 4.24. Each service communicates internally through the Docker Network. External clients can only communicate with the API Gateway service through the NGINX Web Server. Once running, the containers can be managed using Docker commands.

The API Gateway serves as an intermediary layer between clients and backend services. Its core functions include protocol translation, centralized routing, and request mapping from clients to the appropriate microservices. Each service in the architecture is equipped with its own authentication middleware, which enforces security by performing token-based authorization for every incoming request. This process ensures that access to resources is granted only to authorized users, in accordance with Zero Trust Architecture (ZTA) principles, thereby strengthening the security of inter-service communications.

Furthermore, inter-service communication within the microservices environment is implemented using the gRPC (Google Remote Procedure Call) framework. gRPC leverages the HTTP/2 protocol and Protocol Buffers (a compact binary serialization format), offering performance and efficiency benefits ideal for internal service-to-service interactions. In contrast, external communication with clients is established using RESTful APIs (Representational State Transfer), providing greater flexibility and compatibility with various client types.

Performance testing scenario

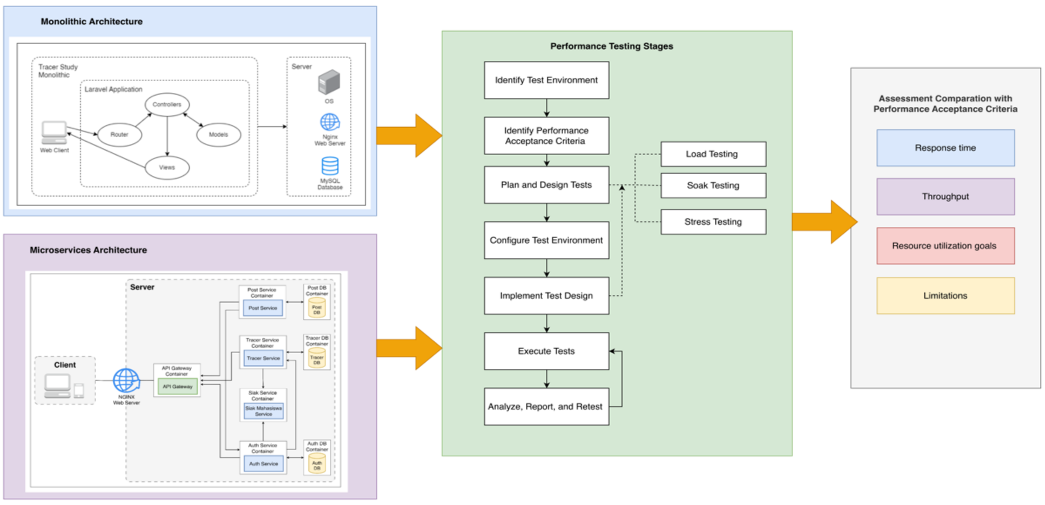


Figure 4. Performance testing stage

The performance testing stages on the monolithic and microservice systems that have been developed as depicted in Figure 4 are carried out by following several stages based on the testing stages according to Meier et al. [2] as follows: identifying the test environment, identifying performance acceptance criteria, planning and designing tests, configuring the test environment, implementing the test design, running tests, and analyzing, reporting results and retesting.

The test environment identification stage is carried out by selecting testing tools and preparing the server. The next stage is to identify performance acceptance criteria, at this stage the performance acceptance criteria indicators are determined. The indicators used as a reference for performance acceptance criteria are in accordance with Table 2.

Table 2. Performance testing indicator assessment

No	Indicator	Description
1.	Response Time	The time from the first byte of page interest sent to the last byte received from the server response.
	Min	Minimum response time
	Average	Average response time
	Max	Maximum response time
2.	Throughput	The measure of the number of transactions or requests that can be processed by a system or application in a given time period, usually measured in units of transactions per second (TPS) to transactions per day (TPD).
	Request per Second	Number of requests handled per second
	Request per Day	Number of requests handled per day
3	Error Rate	Percentage of requests that encountered problems against all requests. This percentage counts all HTTP status codes that returned an error to the server. In addition, it can also count requests that timed out.

The system performance assessment indicators by looking at response time can be categorized into key response times as explained in Table 3.

Table 3. Response time values description

Response Time	Description
0,1 second	This response time is the best time. If the response time is 0.1 indicates that the user feels that the system responds instantly, and the user feels that there is no interference with the system.
1,0 second	This response time is the maximum acceptable response time. Users do not feel any disruption although they may experience some delays. Response times longer than 1 second can disrupt the user experience.
10 second	This response time is the maximum limit after the response time exceeds the acceptable limit. However, currently, if the response time exceeds 6 seconds, the user will leave the site or exit the application.

In general, response times should be as fast as possible, in the range of 0.1 – 1 second. However, users can adapt to slower response times, but users will never be happy with a response time of more than 2 seconds. In addition, performance indicators are measured from the throughput value which can be calculated using the throughput calculation formula as follows:

Throughput = Number of Samples / Time (seconds)
Number of Samples: Total number of requests successfully completed.
Time: Total time required to complete all requests (from start to finish).

For example, if there are 1000 requests and the average time for each request to be completed is 10 seconds, then the throughput value is 1000 requests / 10 seconds = 100 requests/second.

These two performance assessments, namely the response time value and the throughput value of a request indicate that the smaller the response time value and the greater the throughput value, the better the performance of the web application [7]. This good performance will certainly optimize the use of the application in terms of access speed. Good access speed can make users more comfortable when using the application [14]. Throughput represents the server's ability to handle heavy loads. The higher the Throughput, the better the server performance [5].

The next stage is planning and designing testing carried out by creating various test scenarios. Designing test scenarios includes determining the name of the scenario/feature/transaction, sampler/function call/method, number of threads/users/requests, ramp-up period, and loop count. Classification and Performance Testing Indicators, the types of performance testing applied at this testing stage are load testing, soak testing, and stress testing. Load testing is carried out to test the resilience and capacity of the system with a test load according to the scenario that is carried out continuously at a certain time. Soak testing is carried out to determine the stability of the system and evaluate failures that occur over a fairly long period of access. Stress testing is carried out by giving a large load and stopping when the load given results in 100% failure, which means that the system is completely unable to handle the load.

At the configuration and preparation stage of the testing environment, the application is installed/deployed on the server and the testing tools used are set up. The source code deployment process for the UPI monolithic Tracer Study application is carried out using containers. The application is built into a Docker Image and run on the server in the container. The monolithic database is also run in a different MySQL container. In order for the application to run, the environment configuration is also carried out.

Experiment, analyze, and report
The implementation stage of the test design is carried out by implementing scenarios and test specifications with testing tools. The total number of scenarios implemented is a scenario or thread group. To create a sampler, recording and tracking of application usage activities on the browser is also carried out using JMeter. The results of the activity recording will be saved as a step in each test scenario. In addition, to record test results, it is necessary to install a listener on each scenario or thread group, such as the View Results Tree, View Results in Table, and Summary Report listeners. After that, the test execution is carried out in stages starting with the load testing, soak testing, and stress testing stages. The test results that are run are then analyzed and written in a report in the form of a table and if needed, retesting can be carried out.

RESULTS AND DISCUSSIONS

Identifying the test environment

Applications tested on the UPI Tracer Study monolithic system with URL http://203.194.113.32:8000. Testing was conducted using Apache JMeter v5.6.3 software. This software is open-source and cross-platform. In this test, the server used to deploy all services is a Virtual Private Server (VPS) with Operating System: Linux Ubuntu 20.04 LTS 64-bit, CPU 2 Core, Memory 8 GB RAM, and storage 40 GB SSD.

Identifying performance acceptance criteria

Performance acceptance criteria indicators or metrics used in this test are response time and throughput. Response time indicators include minimum response time (min), average response time (Average), and maximum response time. Meanwhile, Throughput criteria calculated by number of handled requests per second (request per second), number of handled requests per day (Request Per Day), and percentage of errors in handling requests (error rate).

Planning and designing testing

In applications with monolithic architecture, testing planning and design are done by creating several test scenarios. In JMeter, one test scenario is a thread group. Scenarios are created based on features that can be used by users. Scenario design includes creating a scenario name, a list of steps or actions, test data needed, the number of users/requests/threads, ramp-up period, and loop count. The following Table 4 shows a list of several performance test scenarios that have been designed for the UPI Tracer Study monolithic system.

Table 4. Performance testing scenario at monolithic scenario

No	Scenario	Step	URL Path
1.	User lists	Access control access page > users	/admin/users
2.	Export data PKTS	Access dashboard questionnaire report > click download	/admin/report_by_year/{year}
3.	Report By Year	Access dashboard questionnaire report	/admin/report_by_year/{year}
4.	Report By Study Program	Access dashboard questionnaire report by Study Program	/admin/perstudyprogram{year}/{program_study_code}
5.	Posts Lists	Access Home Page or Posts Page	/ or /posts
6.	Post Details	Access detail post page	/posts/{id_post}/{slug}

In applications with microservices architecture, testing planning and design are done by creating various test scenarios. One test scenario can include 1 or more endpoints/function calls from microservices. This is because the UPI Tracer Study microservices system breaks down functionality in a monolithic system following the Single Responsibility Principal (SRP). So that one feature or transaction in a monolithic system can be divided into several functions in the microservices system.

Test scenario design includes determining the name of the scenario/feature/transaction, sampler/function call/method, number of threads/users/requests, ramp-up period, and loop count. The following are some performance test scenarios for microservices applications that have been designed as shown in Table 5. While the number of threads, ramp-up period, and loop count are determined based on the type of performance testing performed.

Table 5. Performance testing scenario at microservices scenario

No	Scenario	Sampler
1.	User lists	GetAllUsers
2.	Export data PKTS	ExportPKTSReport
3.	Report By Year	GetPKTSRekapByYear
4.	Report By Study Program	GetPKTSRekapByStudyProgram
5.	Posts Lists	GetAllPosts
6.	Post Details	GetPostById AddVisitor GetCommentsByPostsId

As mentioned, the types of performance testing applied in this testing phase are load testing, soak testing, and stress testing. The following is the design of performance testing carried out based on its type.

Load testing

Load testing is performed for 30-45 minutes per scenario continuously with gradually increasing loads. This is intended to test the resilience and capacity of the system. The design of the load testing test case for each scenario is shown in Table 6. The design was also used to test the previous monolithic system.

Table 6. Load testing scenario

Scenario	*Test*	*Threads*	*Ramp-Up Period*	*Loop Count*
User Lists	PT-a.01-001/001	10	30	5
	PT-a.01-002/002	50	150	3
	PT-a.01-003/003	100	300	3
Export Data	PT-a.02-001/004	10	30	5
	PT-a.02-002/005	50	150	3
	PT-a.02-003/006	100	300	3
Summary Data By Year	PT-a.03-001/007	10	30	5
	PT-a.03-002/008	50	150	3
	PT-a.03-003/009	100	300	3
Summary Data By Study Program	PT-a.04-001/010	10	30	5
	PT-a.04-002/011	50	150	3
	PT-a.04-003/012	100	300	3
Post List	PT-a.05-001/013	10	30	5
	PT-a.05-002/014	50	150	3
	PT-a.05-003/015	100	300	3
Detail Post	PT-a.06-001/016	10	30	5
	PT-a.06-002/017	50	150	3
	PT-a.06-003/018	100	300	3

Soak testing

Soak testing is performed for 24 hours per scenario continuously with a fixed load. This is intended to determine the stability of the system and evaluate failures that occur over a long period of access. The soak testing test case design for each test scenario is shown in Table 7. The design was also used to test the previous monolithic system.

Table 7. Soak testing scenario

Scenario	*Test*	*Threads*	*Ramp-Up Period*	*Loop Count*
User Lists	PT-b.01-001/001	5	30	~
Export Data	PT-b.02-002/002	5	30	~
Summary Data By Year	PT-b.03-003/003	10	30	~
Summary Data By Study Program	PT-b.04-004/004	10	30	~
Post List	PT-b.05-005/005	10	30	~
Detail Post	PT-b.06-006/006	10	30	~

Stress testing

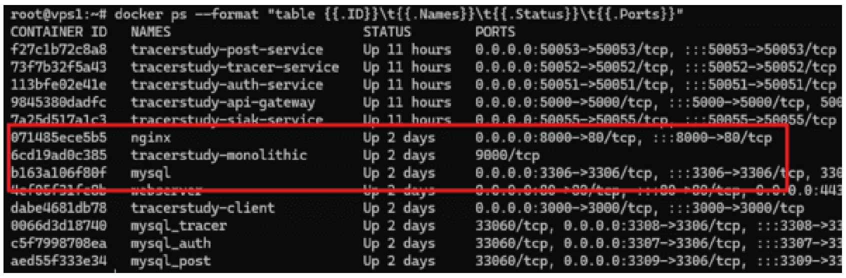
Stress testing is done by giving a large load and stopping when the load given results in 100% failure, which means the system is completely unable to handle the load. This is intended to find out what conditions or loads cause complete failure of the system. The design of stress testing test cases for each test scenario is shown in Table 8. The load is increased by 10x each time it does not meet the 100% error rate condition. This design was also used to test the previous monolithic system.

Table 8. Stress testing scenario

Scenario	*Test*	*Threads*	*Ramp-Up Period*	*Loop Count*
Daftar Users	PT-c.01-001/x	5-?	30	~
Export PKTS	PT-c.02-001/x	5-?	30	~
PKTS Rekap By Year	PT-c.03-001/x	10-?	30	~
PKTS Rekap By Prodi	PT-c.04-001/x	10-?	30	~
Daftar Posts	PT-c.05-001/x	10-?	30	~
Detail Post	PT-c.06-001/x	10-?	30	~

Configurating testing environment

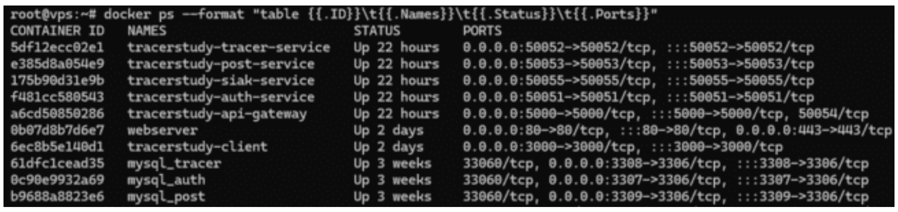
At this stage, the configuration and preparation of the testing environment is carried out, such as installing/deploying the application on the server and setting up the testing tools used. The process of deploying the source code of the UPI monolithic Tracer Study application is carried out using containers. The application is built into a Docker Image and run on the server in the container. The monolithic database is also run in a different MySQL container. In order for the application to run, the environment configuration is also carried out. The application can be accessed after the DNS configuration and web server installation are carried out. Checking the running of the application can be seen in Figure 5.



```
root@vps1:~# docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Status}}\t{{.Ports}}"
CONTAINER ID   NAMES                         STATUS        PORTS
f27c1b72c8a8   tracerstudy-post-service      Up 11 hours   0.0.0.0:50053->50053/tcp, :::50053->50053/tcp
73f7b32f5a43   tracerstudy-tracer-service    Up 11 hours   0.0.0.0:50052->50052/tcp, :::50052->50052/tcp
113bfe02e41e   tracerstudy-auth-service      Up 11 hours   0.0.0.0:50051->50051/tcp, :::50051->50051/tcp
9845380dadfc   tracerstudy-api-gateway       Up 11 hours   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp, 500
7a25d517a1c3   tracerstudy-siak-service      Up 11 hours   0.0.0.0:50055->50055/tcp, :::50055->50055/tcp
071485ece5b5   nginx                         Up 2 days     0.0.0.0:8000->80/tcp, :::8000->80/tcp
6cd19ad0c385   tracerstudy-monolithic        Up 2 days     9000/tcp
b163a106f80f   mysql                         Up 2 days     0.0.0.0:3306->3306/tcp, :::3306->3306/tcp, 330
4cf05f31fc8b   webserver                     Up 2 days     0.0.0.0:80->80/tcp, :::80->80/tcp, 0.0.0.0:443
dabe4681db78   tracerstudy-client            Up 2 days     0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
0066d3d18740   mysql_tracer                  Up 2 days     33060/tcp, 0.0.0.0:3308->3306/tcp, :::3308->33
c5f7998708ea   mysql_auth                    Up 2 days     33060/tcp, 0.0.0.0:3307->3306/tcp, :::3307->33
aed55f333e34   mysql_post                    Up 2 days     33060/tcp, 0.0.0.0:3309->3306/tcp, :::3309->33
```

Figure 5. Deploying monolithic tracer study application to VPS

Meanwhile, for microservices applications, it is required to inspect all services that have been installed on the server by examining the running Docker Container as shown in Figure 6. The 'Up' state on the container implies that the container and the services in it are functioning well (there are no runtime issues that cause the service to terminate).



```
root@vps:~# docker ps --format "table {{.ID}}\t{{.Names}}\t{{.Status}}\t{{.Ports}}"
CONTAINER ID   NAMES                         STATUS        PORTS
5df12ecc02e1   tracerstudy-tracer-service    Up 22 hours   0.0.0.0:50052->50052/tcp, :::50052->50052/tcp
e385d8a054e9   tracerstudy-post-service      Up 22 hours   0.0.0.0:50053->50053/tcp, :::50053->50053/tcp
175b90d31e9b   tracerstudy-siak-service      Up 22 hours   0.0.0.0:50055->50055/tcp, :::50055->50055/tcp
f481cc580543   tracerstudy-auth-service      Up 22 hours   0.0.0.0:50051->50051/tcp, :::50051->50051/tcp
a6cd50850286   tracerstudy-api-gateway       Up 22 hours   0.0.0.0:5000->5000/tcp, :::5000->5000/tcp, 50054/tcp
0b07d8b7d6e7   webserver                     Up 2 days     0.0.0.0:80->80/tcp, :::80->80/tcp, 0.0.0.0:443->443/tcp
6ec8b5e140d1   tracerstudy-client            Up 2 days     0.0.0.0:3000->3000/tcp, :::3000->3000/tcp
61dfc1cead35   mysql_tracer                  Up 3 weeks    33060/tcp, 0.0.0.0:3308->3306/tcp, :::3308->3306/tcp
0c90e9932a69   mysql_auth                    Up 3 weeks    33060/tcp, 0.0.0.0:3307->3306/tcp, :::3307->3306/tcp
b9688a8823e6   mysql_post                    Up 3 weeks    33060/tcp, 0.0.0.0:3309->3306/tcp, :::3309->3306/tcp
```

Figure 6. Status deployed services status on microservices application in server

In addition, at this stage, preparations are made for using JMeter as a testing tool. The performance testing tool used, namely Apache JMeter, is shown in Figure 7.

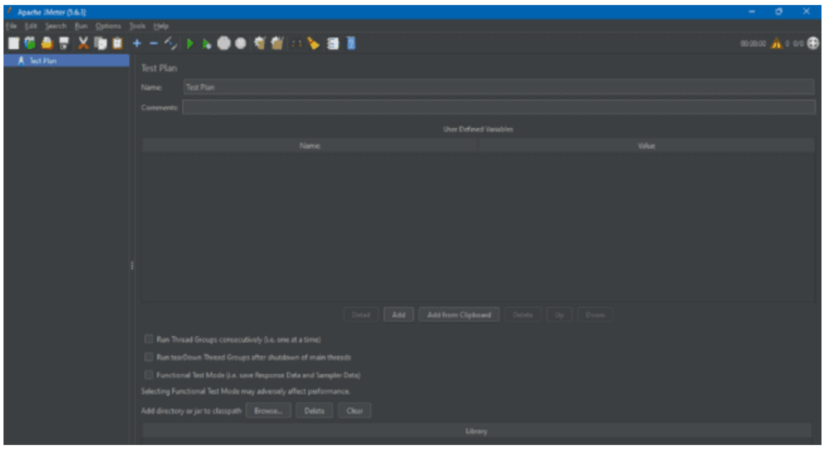
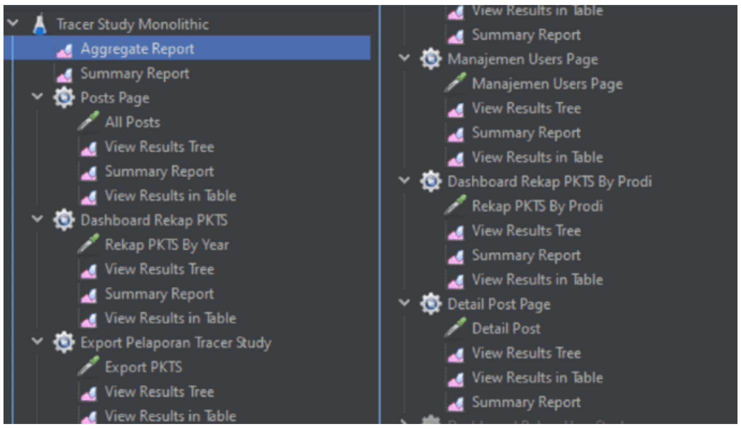

Apache JMeter (5.6.3)
File Edit Search Run Options Tools Help
Test Plan
Test Plan
Name: Test Plan
Comments:
User Defined Variables
Name
Value
Detail
Add
Add from Clipboard
Delete
Up
Down
Run Thread Groups consecutively (i.e. one at a time)
Run tearDown Thread Groups after shutdown of main threads
Functional Test Mode (i.e. save Response Data and Sampler Data)
Selecting Functional Test Mode may adversely affect performance.
Add directory or jar to classpath
Browse...
Delete
Clear
Library

Figure 7. JMeter user interface

Implementing scenario testing

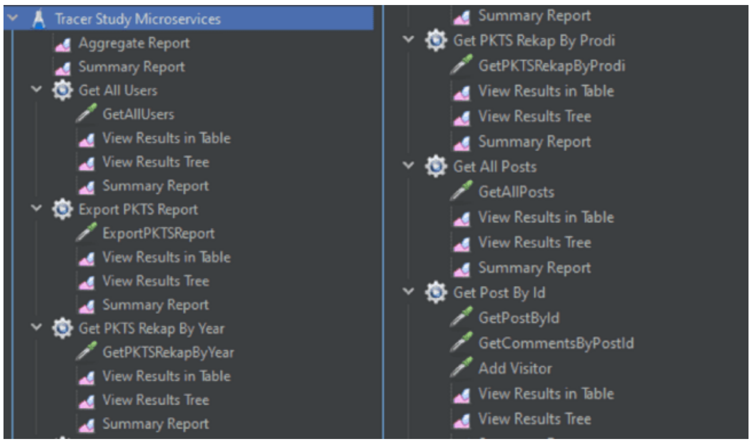
The previously designed test scenarios and specifications are implemented into the Apache JMeter testing tool as shown in Figure 8 for monolithic application testing and Figure 9 for microservices application testing. The total number of scenarios implemented is 6 scenarios or thread groups. In monolithic testing, each thread group represents the activity of using the application on the browser and records user activity (step). While in microservices testing, each thread group has a sampler which is a gRPC function involved in the scenario. In addition, in both tests, each thread group is also paired with a listener to record test results, namely View Results in Table, View Results Tree, and Summary Report. The result of this implementation is the JMeter test plan file which is stored in .jmx format in the Apache JMeter /bin folder. This design implementation represents equivalent features in both monolithic and microservices applications.



```
Tracer Study Monolithic
    Aggregate Report
    Summary Report
    Posts Page
        All Posts
        View Results Tree
        Summary Report
        View Results in Table
    Dashboard Rekap PKTS
        Rekap PKTS By Year
        View Results Tree
        Summary Report
        View Results in Table
    Export Pelaporan Tracer Study
        Export PKTS
        View Results Tree
        View Results in Table
```

```
        View Results in Table
        Summary Report
    Manajemen Users Page
        Manajemen Users Page
        View Results Tree
        Summary Report
        View Results in Table
    Dashboard Rekap PKTS By Prodi
        Rekap PKTS By Prodi
        View Results Tree
        Summary Report
        View Results in Table
    Detail Post Page
        Detail Post
        View Results Tree
        View Results in Table
        Summary Report
```

Figure 8. Implementation of performance testing for monolithic application



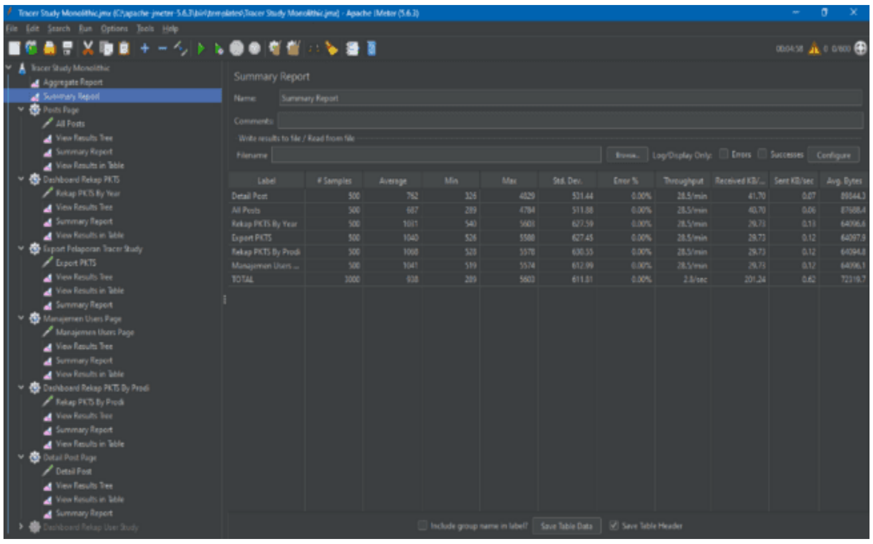
```
Tracer Study Microservices
    Aggregate Report
    Summary Report
    Get All Users
        GetAllUsers
        View Results in Table
        View Results Tree
        Summary Report
    Export PKTS Report
        ExportPKTSReport
        View Results in Table
        View Results Tree
        Summary Report
    Get PKTS Rekap By Year
        GetPKTSRekapByYear
        View Results in Table
        View Results Tree
        Summary Report
```

```
        Summary Report
    Get PKTS Rekap By Prodi
        GetPKTSRekapByProdi
        View Results in Table
        View Results Tree
        Summary Report
    Get All Posts
        GetAllPosts
        View Results in Table
        View Results Tree
        Summary Report
    Get Post By Id
        GetPostById
        GetCommentsByPostId
        Add Visitor
        View Results in Table
        View Results Tree
```

Figure 9. Implementation of performance testing for microservice application

Executing testing

At this stage, all tests are run in stages. Starting from load testing, soak testing, to stress testing. Apache JMeter used to execute the test is run on a local machine. The following Figure 10 (monolithic) and Figure 11 (microservices) show a preview when load testing is being run using the Graphical User Interface (GUI) of the Apache JMeter application. The results of all tests are recorded by each listener installed on each thread group. The results are analyzed and reported in the next testing stage.

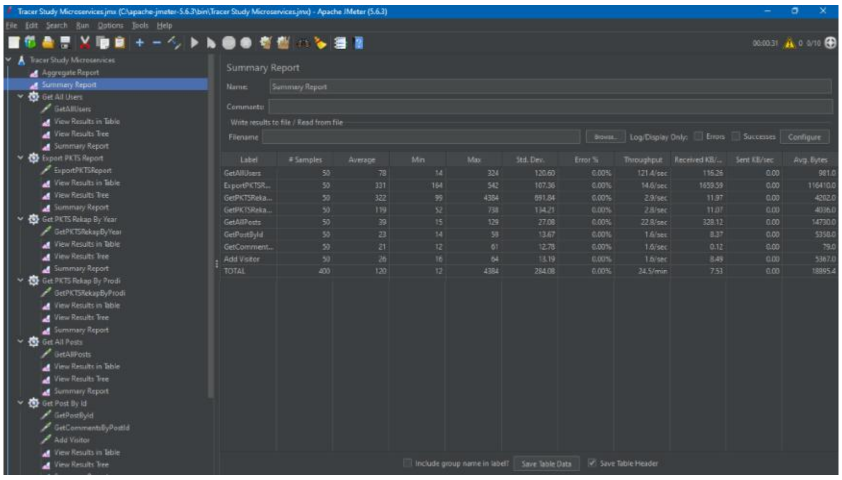


Summary Report

Name: Summary Report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Detail Post	500	762	326	4829	531.44	0.00%	28.5/min	41.70	0.07	89844.3
All Posts	500	687	289	4784	511.88	0.00%	28.5/min	40.70	0.06	87688.4
Rekap PKTS By Year	500	1031	540	5603	627.59	0.00%	28.5/min	29.73	0.13	64096.6
Export PKTS	500	1040	526	5580	627.45	0.00%	28.5/min	29.73	0.12	64097.9
Rekap PKTS By Prodi	500	1060	520	5578	630.55	0.00%	28.5/min	29.73	0.12	64094.8
Manajemen Users ...	500	1041	519	5574	612.99	0.00%	28.5/min	29.73	0.12	64096.1
TOTAL	3000	938	289	5603	611.01	0.00%	2.8/sec	201.24	0.62	72319.7

Figure 10. Running load testing for monolithic application


Summary Report

Name: Summary Report

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
GetAllUsers	50	78	14	324	120.60	0.00%	121.4/sec	116.26	0.00	981.0
ExportPKTSR...	50	331	164	542	107.36	0.00%	14.6/sec	1659.59	0.00	116410.0
GetPKTSReka...	50	322	99	4384	891.84	0.00%	2.9/sec	11.97	0.00	4202.0
GetPKTSReka...	50	119	52	738	134.21	0.00%	2.8/sec	11.07	0.00	4036.0
GetAllPosts	50	39	15	129	27.08	0.00%	22.8/sec	328.12	0.00	14730.0
GetPostById	50	23	14	59	13.67	0.00%	1.6/sec	8.37	0.00	5358.0
GetComment...	50	21	12	61	12.78	0.00%	1.6/sec	0.12	0.00	79.0
Add Visitor	50	26	16	64	13.19	0.00%	1.6/sec	8.49	0.00	5367.0
TOTAL	400	120	12	4384	284.08	0.00%	24.5/min	7.53	0.00	18895.4

Figure 11. Running load testing for microservice application

Analyzing and report testing

Based on result of executing testing, test result data was obtained that shows the performance of the two applications with different architectures, namely monolithic and microservices. The following are the test results on load testing, soak testing, and stress testing.

Load testing

The results of the load testing on the monolithic application are shown in Table 9. From the data, it can be seen that the entire scenario has an average response time of 1138ms (milliseconds) and a throughput of 1rps (request per second). Meanwhile, to calculate the estimated throughput per day, the following formula can be used [rps x 60 x 60 x 24] [16]. From this calculation, the throughput value per day is 86,400 requests. From this load testing, it can also be seen that the average system error rate in handling requests is 0.23%.

Table 9. Performance result of load testing at monolithic architecture

Test	Reqs	*Response Time (ms)*			*Throughput*		Error Rate
		Min	*Avg*	*Max*	*Sec*	*Day*	
PT-a.01-001/001	50	693	1557	2232	1,5	129.600	0,00%
PT-a.01-002/002	200	641	1382	1984	1,1	95.040	0,00%
PT-a.01-003/003	500	519	1041	5574	0,5	43.200	0,21%
PT-a.02-001/004	50	618	1497	2178	1,5	129.600	0,00%
PT-a.02-002/005	200	587	1428	2486	0,8	69.120	0,00%
PT-a.02-003/006	500	526	1040	5588	0,5	43.200	0,15%
PT-a.03-001/007	50	612	1468	2174	1,5	129.600	0,00%
PT-a.03-002/008	200	584	1395	2341	1,3	112.320	0,02%
PT-a.03-003/009	500	540	1031	5603	0,5	43.200	0,24%
PT-a.04-001/010	50	710	1547	2224	1,5	129.600	0,00%
PT-a.04-002/011	200	649	1193	3146	1,1	95.040	0,00%
PT-a.04-003/012	500	528	1068	5578	0,5	43.200	0,48%
PT-a.05-001/013	50	389	849	1677	1,6	138.240	0,00%
PT-a.05-002/014	200	345	811	1798	0,9	77.760	0,07%
PT-a.05-003/015	500	289	687	4784	0,5	43.200	0,59%
PT-a.06-001/016	50	366	909	1532	1,6	138.240	0,00%
PT-a.06-002/017	200	242	827	2357	1,2	103.680	0,00%
PT-a.06-003/018	500	326	762	4829	0,5	43.200	0,03%
Average	**700**	**242**	**1138**	**5578**	**1**	**86.400**	**0,23%**

Meanwhile, in the microservices application, the test results data are shown in Table 10 below. From the data, it can be seen that the entire scenario has an average response time of 154ms (milliseconds) and a throughput of 16.4rps (requests per second). Meanwhile, to calculate the estimated throughput per day, the following formula can be used [rps x 60 x 60 x 24]. From this calculation, the throughput value per day is 1,451,067 requests. From this load testing, it can also be seen that the average system error rate in handling requests is 0.00%.

Table 10. Performance result of load testing at microservices architecture

Scenario	*Reqs*	*Response Time*			*Throughput*		*Error Rate*
		Min	*Avg*	*Max*	*Sec*	*Day*	
PT-a.01-001/001	50	14	78	324	121,4	10.488.960	0,00%
PT-a.01-002/002	200	12	39	324	8,5	734.400	0,00%
PT-a.01-003/003	500	12	34	943	12,4	1.071.840	0,00%
PT-a.02-001/004	50	164	331	542	14,6	1.261.440	0,00%
PT-a.02-002/005	200	164	523	1451	8,1	699.840	0,00%
PT-a.02-003/006	500	164	529	1451	12,1	1.045.440	0,00%
PT-a.03-001/007	50	99	322	4384	2,9	250.560	0,00%
PT-a.03-002/008	200	96	203	4384	8,5	734.400	0,00%
PT-a.03-003/009	500	96	226	4383	13,4	1.157.760	0,00%
PT-a.04-001/010	50	52	119	738	2,8	241.920	0,00%
PT-a.04-002/011	200	50	85	738	9,5	820.800	0,00%
PT-a.04-003/012	500	49	97	1266	14,7	1.270.080	0,00%
PT-a.05-001/013	50	15	39	129	22,8	1.969.920	0,00%
PT-a.05-002/014	200	15	31	129	9,4	812.160	0,00%
PT-a.05-003/015	500	14	31	129	14,2	1.226.880	0,00%
PT-a.06-001/016	50	12	24	64	4,8	414.720	0,00%
PT-a.06-002/017	200	12	31	1053	2,8	803.520	0,00%
PT-a.06-003/018	500	12	31	1053	12,9	1.114.560	0,00%
Average	**700**	**12**	**154**	**4384**	**16,4**	**1.451.067**	**0,00%**

The figure 12,13,14 and 15 explaining table 9, shows how response time graphs can help you see how well a system is working. The throughput, error rate, and heatmap all show a consistent pattern when it comes to stability and processing power at different load levels. The average response time for 50 and 200 users is usually between 580 and 710 ms. This is longer than some 500 user load scenarios, which actually show shorter response times. This suggests that the system is optimized internally in some cases, but not always. When there are few users, the system can handle the most transactions, with 138,240 transactions per day. But when there are 500 users, it can only handle 43,200 transactions per day, which shows that it can't handle more load. The error rate graph shows that most of the 50 and 200 user load scenarios don't have any errors. However, there are big spikes in a few 500 user load scenarios, especially in the 04-003 and 05-003 series, where errors range from 0.48% to 0.59%. The heatmap highlights this pattern by showing that there are a lot of light colors in the 500 loads column. This shows that errors are mostly happening in high-flow situations. In general, these results show that the system works very well and stably when the load is low to medium. However, when the load is high, the performance drops, the number of errors goes up, and the throughput goes down. This means that architectural improvements or load handling mechanisms are needed for larger applications.

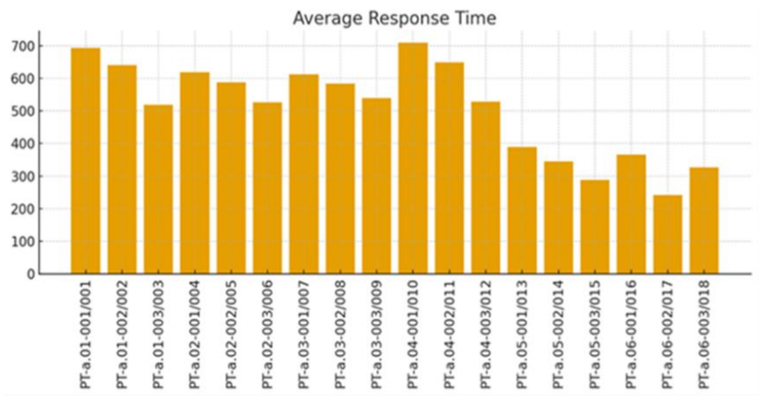


Figure 12. Bar chart response time (average), displays a comparison of avg response time values for all test cases

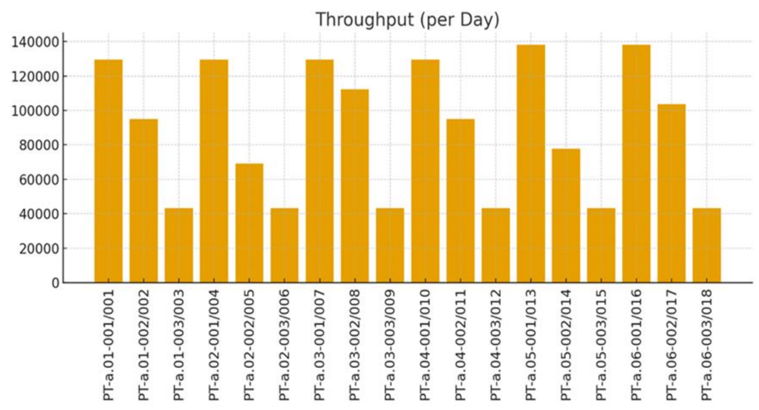


Figure 13. Bar chart throughput (per day), shows the processing capacity of each scenario per day

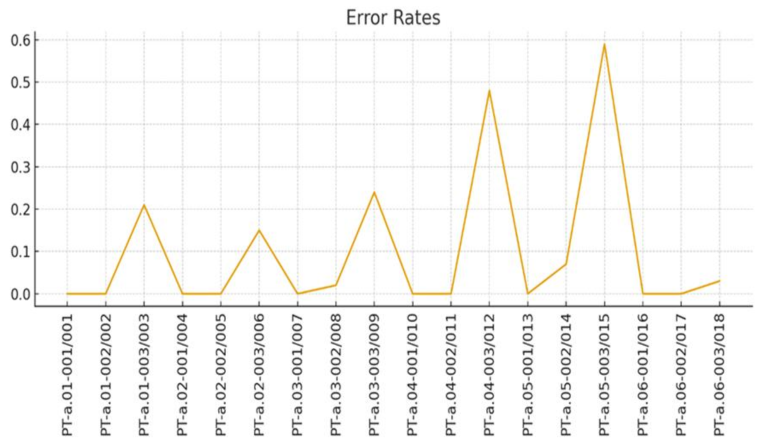


Figure 14. Line chart error rates, eror trend graph for each test

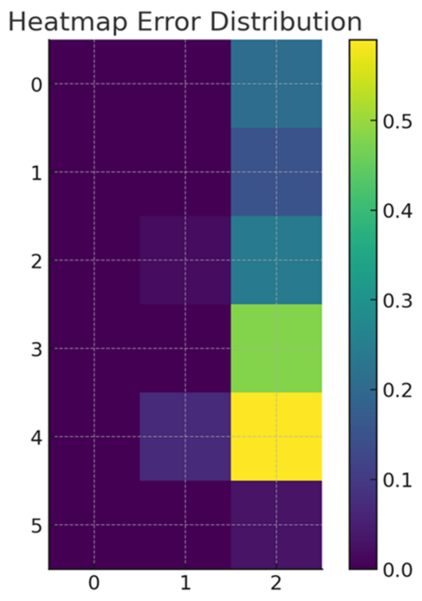


Figure 15. Error distribution heatmap created from 18 error rate datapoints mapped onto a 6×3 grid

Soak testing

Soak testing that has been done for 24 hours found that each request has a different percentage of request handling failure or error rate. In monolithic applications, the average error rate of all scenarios is 2.53%. Meanwhile, detailed error rate data per scenario can be seen in Table 11. Based on data recorded by the listener of each thread group, the errors that occur include having the status codes Internal Server Error, Time Out, and Bad Gateway.

Table 11. Performance result of soak testing at monolithic architecture

Scenario	***Reqs***	***Response Time (ms)***			***Throughput***		***Error Rate***
		Min	***Avg***	***Max***	***Sec***	***Day***	
PT-b.01-001/001	5	325	697	1226	1,2	103.680	0,96%
PT-b.02-002/002	5	618	1256	1978	1,8	155.520	2,21%
PT-b.03-003/003	10	635	1394	2575	1,1	95.040	1,57%
PT-b.04-004/004	10	713	1344	2087	1,2	103.680	2,13%
PT-b.05-005/005	10	392	874	2349	1,3	112.320	2,95%
PT-b.06-006/006	10	386	1034	1825	1,9	164.160	3,46%
Average	**8**	**325**	**1099**	**2575**	**1,4**	**120.960**	**2,53%**

Meanwhile, in microservices applications, the average error rate of all scenarios is 0.03%. Meanwhile, detailed error rate data per scenario can be seen in Table 12. Based on data recorded by the listener of each thread group, the errors that occur include having the status codes Unavailable, Deadline Exceeded, and Resource Exhausted.

Table 12. Performance result of soak testing at microservices architecture

Scenario	***Reqs***	***Response Time***			***Throughput***		***Error Rate***
		Min	***Avg***	***Max***	***Sec***	***Day***	
PT-b.01-001/001	5	12	24	65	4,5	388.800	0,00%
PT-b.02-002/002	5	168	245	412	5,5	475.200	0,04%
PT-b.03-003/003	10	49	87	231	5	432.000	0,01%
PT-b.04-004/004	10	93	149	375	4,6	397.440	0,01%
PT-b.05-005/005	10	14	25	86	1,3	112.320	0,02%
PT-b.06-006/006	10	14	22	76	4,5	388.800	0,03%
Average	**8**	**12**	**92**	**412**	**4,2**	**430.560**	**0,03%**

Stress testing

Stress testing that is carried out continuously until the system experiences 100% failure, finds that in monolithic applications the system will go down when given an average load of 23,000 requests per second. This shows that the system has a limit in handling the maximum load received at that amount every second. In addition to the system, this limit is also influenced by the specifications of the machine/server used for the implementation of the system. The maximum load limit that can cause failure in each scenario is shown in Table 13.

Table 13. Performance result of stress testing at monolithic architecture

Scenario	***Reqs***	***Response Time***			***Throughput***		***Error Rate***
		Min	***Avg***	***Max***	***Sec***	***Day***	
PT-c.01-001/011	5	53	69	84	9,2	794.880	0,00%
PT-c.01-002/012	50	47	62	78	10,8	933.120	0,17%
PT-c.01-003/013	500	91	154	188	7,6	656.640	37,19%
PT-c.01-004/014	5.000	90	189	265	3,3	285.120	71,82%
PT-c.01-005/015	50.000	86	203	478	0,2	17.280	100,00%
PT-c.02-001/020	5	128	152	212	5,3	457.920	0,00%
PT-c.02-002/021	50	111	161	249	3,2	276.480	44,62%
PT-c.02-003/022	500	117	178	285	2,7	233.280	100,00%
PT-c.03-001/023	10	166	192	239	7,4	639.360	0,00%
PT-c.03-002/024	100	159	210	240	5,8	501.120	19,78%
PT-c.03-003/025	1.000	151	485	793	3,1	267.840	75,33%
PT-c.03-004/026	10.000	164	397	726	2,2	190.080	100,00%
PT-c.04-001/027	10	160	182	214	4,7	406.080	0,00%
PT-c.04-002/028	100	154	189	223	3,4	293.760	0,25%
PT-c.04-003/029	1.000	162	294	328	2,5	216.000	80,31%
PT-c.04-004/030	10.000	171	288	331	1,6	138.240	100,00%
PT-c.05-001/043	10	73	91	140	12,4	1.071.360	0,00%
PT-c.05-002/044	100	99	168	227	12,1	1.045.440	9,12%
PT-c.05-003/045	1.000	102	139	162	13,4	1.157.760	64,37%
PT-c.05-004/046	10.000	159	237	285	10,5	907.200	100,00%
PT-c.06-001/047	10	70	120	161	11,1	959.040	0,00%
PT-c.06-002/048	100	92	139	192	10,6	915.840	19,04%
PT-c.06-003/049	1.000	105	146	189	11,1	959.040	46,71%
PT-c.06-004/050	10.000	144	229	341	8,3	717.120	100,00%
Average	**306.100**	**47**	**178**	**793**	**21,9**	**672.224**	**68,31%**

Meanwhile, in the microservices application, it was found that the system would go down when given an average load of 39,000,000 requests. This shows that the system has a limit in handling the maximum load received at that amount. In addition to the system, this limit is also influenced by the machine/server specifications used for system implementation. Table 14 below shows the maximum load limits that can cause failure in each scenario.

Table 14. Performance result of stress testing at microservice architecture

Scenario	***Reqs***	***Response Time***			***Throughput***		***Error Rate***
		Min	*Avg*	*Max*	*Sec*	*Day*	
PT-c.01-001/001	10	64	81	112	6,4	552.960	0,00%
PT-c.01-002/002	100	83	148	218	5,9	509.760	0,07%
PT-c.01-003/003	1.000	127	199	294	5,1	440.640	23,00%
PT-c.01-004/004	10.000	119	243	326	4,4	380.140	70,23%
PT-c.01-005/005	100.000	121	266	314	0,2	17.280	100,00%
PT-c.02-001/006	5	81	128	183	4,2	362.880	0,00%
PT-c.02-002/007	50	79	119	162	5,3	457.920	0,02%
PT-c.02-003/008	500	65	129	154	4,8	414.720	13,28%
PT-c.02-004/009	5.000	67	182	229	3,2	622.080	49,65%
PT-c.02-005/010	50.000	92	201	412	1,4	120.960	100,00%
PT-c.03-001/011	5	53	69	84	9,2	794.880	0,00%
PT-c.03-002/012	50	47	62	78	10,8	933.120	0,17%
PT-c.03-003/013	500	91	154	188	7,6	656.640	37,19%
PT-c.03-004/014	5.000	90	189	265	3,3	285.120	71,82%
PT-c.03-005/015	50.000	86	203	478	0,2	17.280	100,00%
PT-c.04-001/016	10	113	148	225	2,7	233.280	0,00%
PT-c.04-002/017	100	102	166	231	2,3	198.720	0,01%
PT-c.04-003/018	1.000	133	190	274	1,4	120.960	64,80%
PT-c.04-004/019	10.000	112	237	389	0,6	51.840	100,00%
PT-c.05-001/020	5	128	152	212	5,3	457.920	0,00%
PT-c.05-002/021	50	111	161	249	3,2	276.480	44,62%
PT-c.05-003/022	500	117	178	285	2,7	233.280	100,00%
PT-c.06-001/023	10	166	192	239	7,4	639.360	0,00%
PT-c.06-002/024	100	159	210	240	5,8	501.120	19,78%
PT-c.06-003/025	1.000	151	485	793	3,1	267.840	75,33%
PT-c.06-004/026	10.000	164	397	726	2,2	190.080	100,00%
PT-c.07-001/027	10	160	182	214	4,7	406.080	0,00%
PT-c.07-002/028	100	154	189	223	3,4	293.760	0,25%
PT-c.07-003/029	1.000	162	294	328	2,5	216.000	80,31%
PT-c.07-004/030	10.000	171	288	331	1,6	138.240	100,00%
PT-c.08-001/031	5	128	156	219	5,2	449.280	0,00%
PT-c.08-002/032	50	172	237	291	5,1	440.640	0,78%
PT-c.08-003/033	500	224	251	299	4,3	371.520	36,10%
PT-c.08-004/034	5.000	241	277	319	2,8	241.920	100,00%
PT-c.09-001/035	10	94	112	146	11,1	959.040	0,00%
PT-c.09-002/036	100	91	138	172	10,4	898.560	21,94%
PT-c.09-003/037	1.000	115	168	202	10,1	872.640	88,12%
PT-c.09-004/038	10.000	149	211	276	7,3	630.720	100,00%
PT-c.10-001/039	10	62	97	150	8,8	760.320	0,00%
PT-c.10-002/040	100	79	105	162	7,7	665.280	4,92%
PT-c.10-003/041	1.000	97	133	151	7,2	622.080	37,28%
PT-c.10-004/042	10.000	155	218	270	5,2	449.280	100,00%
PT-c.11-001/043	10	73	91	140	12,4	1.071.360	0,00%
PT-c.11-002/044	100	99	168	227	12,1	1.045.440	9,12%
PT-c.11-003/045	1.000	102	139	162	13,4	1.157.760	64,37%
PT-c.11-004/046	10.000	159	237	285	10,5	907.200	100,00%
PT-c.12-001/047	10	70	120	161	11,1	959.040	0,00%
PT-c.12-002/048	100	92	139	192	10,6	915.840	19,04%
PT-c.12-003/049	1.000	105	146	189	11,1	959.040	46,71%
PT-c.12-004/050	10.000	144	229	341	8,3	717.120	100,00%
Average	**306.100**	**47**	**178**	**793**	**21,9**	**672.224**	**68,31%**

Comparative analysis of performance test results of monolithic systems with microservices

Table 15. Comparison of load testing performance on monolithic and microservices systems

Scenario	Requests	*Monolithic*			*Microservices*		
		Response Time	*RPS*	*Error Rate*	*Response Time*	*RPS*	*Error Rate*
	50	1557 ms	1,5/s	0,00%	78 ms	121,4/s	0,00%
PT-a.01	200	1382 ms	1,1/s	0,00%	39 ms	8,5/s	0,00%
	500	1041 ms	0,5/s	0,21%	34 ms	12,4/s	0,00%
	50	1497 ms	1,5/s	0,00%	331 ms	14,6/s	0,00%
PT-a.02	200	1428 ms	0,8/s	0,00%	523 ms	8,1/s	0,00%
	500	1040 ms	0,5/s	0,15%	529 ms	12,1/s	0,00%
	50	1468 ms	1,5/s	0,00%	322 ms	2,9/s	0,00%
PT-a.03	200	1395 ms	1,3/s	0,02%	203 ms	8,5/s	0,00%
	500	1031 ms	0,5/s	0,24%	226 ms	13,4/s	0,00%
	50	1547 ms	1,5/s	0,00%	119 ms	2,8/s	0,00%
PT-a.04	200	1193 ms	1,1/s	0,00%	85 ms	9,5/s	0,00%
	500	1068 ms	0,5/s	0,48%	97 ms	14,7/s	0,00%
	50	849 ms	1,6/s	0,00%	39 ms	22,8/s	0,00%
PT-a.05	200	811 ms	0,9/s	0,07%	31 ms	9,4/s	0,00%
	500	687 ms	0,5/s	0,59%	31 ms	14,2/s	0,00%
	50	909 ms	1,6/s	0,00%	24 ms	4,8/s	0,00%
PT-a.06	200	827 ms	1,2/s	0,00%	31 ms	2,8/s	0,00%
	500	762 ms	0,5/s	0,03%	31 ms	12,9/s	0,00%
Average		1138 ms	1/s	0,23%	154 ms	16,4/s	0,00%

Table 15 presents a comparison of the results of load testing performance on monolithic and microservices systems. Based on research, the test case start with small number of concurrent users/request per second for example 1,20, 50, 100 request per second. On this test case, testing conduct from 50 request until 200 request to understand how your system behaves under light load. Gradually increase the number of users to identify the point at which your system begins to experience stress [16].

The data presented in Table 15 shows that there is no response time in the microservices system that exceeds the monolithic system in all scenarios. The average response time improved from 1138ms to 154ms. There was also an increase in the number of requests that can be processed in 1 second (requests per second/rps), namely from an average of 1rps to 16.4rps. System failures (marked by the error rate) that occurred also decreased from the previous average of 0.23% to 0%. These data show that there has been an improvement in the system, especially in terms of performance with the implementation of the microservices architecture. In the soak testing test carried out for 24 hours with a constant load, the results of the two system architectures also showed differences. The following Table 16 presents a comparison of the results of soak testing performance tests on monolithic and microservices systems.

Table 16. Comparation result of soak testing monolithic and microservices system

Scenario	Requests	*Monolithic*			*Microservices*		
		Response Time	*RPS*	*Error Rate*	*Response Time*	*RPS*	*Error Rate*
PT-b.01	5	697	1,2	0,96%	24	4,5	**0,00%**
PT-b.02	5	1256	1,8	2,21%	245	5,5	**0,04%**
PT-b.03	10	1394	1,1	1,57%	87	5	**0,01%**
PT-b.04	10	1344	1,2	2,13%	149	4,6	**0,01%**
PT-b.05	10	874	1,3	2,95%	25	1,3	**0,02%**
PT-b.06	10	1034	1,9	3,46%	22	4,5	**0,03%**
Average		**1099**	**1,4**	**2,53%**	**92**	**4,2**	**0,03%**

Based on Table 16, the soak testing results show significant performance differences between the monolithic system and the microservices system. The microservices system consistently outperforms the monolithic system, showing a significantly lower average response time (92 ms compared to 1099 ms) and higher requests per second (RPS) (4.2 rps compared to 1.4 rps). In addition, the microservices system maintains a much lower error rate, with an average of only 0.03% compared to the monolithic system of 2.53%. These results indicate that the microservices architecture is more resilient and stable in its availability in handling loads over a sustained period of time.

Practical implications

Migrating from a monolithic architecture to microservices is a step organizations should consider to improve resilience and scalability. This migration process is generally undertaken when the existing architecture faces limitations in terms of scalability and flexibility, which impacts the speed of feature development and deployment. While it offers advantages, this migration process presents challenges such as distributed system complexity, data consistency management, and increased monitoring and automation requirements. These challenges must be addressed by implementing DevOps principles. To support an effective transition, organizations are advised to utilize containerization (Docker), orchestration tools (Kubernetes), API Gateway, and monitoring tools such as Prometheus and Grafana. A phased migration strategy using the strangler pattern (Fowler, 2018) is also recommended to ensure the transformation process does not disrupt existing systems.

The findings of this study were implemented contextually on the UPI Tracer Study system using Laravel and Golang gRPC. While the microservices design principles found in this study are generally applicable, their effectiveness may vary in other programming languages or frameworks, such as Node.js or Python. Therefore, these results are best viewed as a limited empirical contribution that can be extended through studies on other technologies.

The following adjustments can be made to implement a microservices architecture in other languages or frameworks, requiring adjustments at the communication, infrastructure, performance, and security levels.

1. Inter-Service Communication Model
   Different languages and frameworks generally use varying communication mechanisms, such as REST API (HTTP/JSON), gRPC (Protocol Buffers), or GraphQL. Communication efficiency, latency, and overhead need to be re-examined because performance between protocols can vary significantly.
2. Dependency Management and Framework Ecosystem
   Each language has a different ecosystem of packages and libraries to support microservices. For example, Node.js has strong support for REST APIs and event-driven architecture, while Go is more optimized for high-performance services. These differences impact modularization, testing, and dependency management strategies.
3. Performance and Scalability
   Microservices architectures are highly dependent on the runtime efficiency of the language used. Go, for example, excels in concurrency and memory efficiency, while Python tends to be slower but excels in AI/analytics integration. Therefore, performance testing (load testing and stress testing) needs to be tailored to each language.
4. Distributed Data Management Strategy
   Database management in microservices systems depends on the data integration pattern used, such as a database per service or a shared database.
5. DevOps Deployment and Integration Mechanisms
   Different languages have different preferences for DevOps tools and pipelines. For example, Java Spring Boot is often integrated with Jenkins or ArgoCD, while Node.js often uses GitHub Actions or Docker Compose. This impacts the CI/CD process, containerization, and monitoring.
6. Service Security and Authentication Management
   Security implementations, such as token-based authentication (JWT), OAuth 2.0, or mTLS, can differ across ecosystems. Each framework provides different libraries for authorization and encryption, so security configurations need to be reviewed for consistency across services.
7. Adapting Patterns to Observability and Logging
   Microservices systems require comprehensive monitoring and tracing. Tools like Prometheus, Grafana, Jaeger, or ELK can be used, but integration may vary depending on the language and framework. For example, the implementation of logging middleware in Node.js differs from that in Golang or Python.
8. Development Team Readiness and Resources
   Migrating or reimplementing to another language requires considering the technical capabilities of the development team. Proficiency in the language, framework, and supporting tools will directly impact implementation efficiency and system stability.

Results from a Laravel–Golang gRPC-based system can serve as a conceptual reference, but retesting and configuration calibration on other platforms are necessary to ensure the results remain valid and optimal.

CONCLUSION

The results of performance tests on monolithic and microservices systems indicate how implementing microservices affects system performance. It was discovered that implementing the microservices system improved system performance in terms of response time, which decreased from an average of 1138ms to 154ms. Furthermore, the throughput, or number of requests that can be processed per day, jumped from an average of one per second (8400 requests per day) to 16.4 per second (1,419,960 requests per day). The microservices system also has a lower failure or error rate in normal use than the monolithic system, with an error rate of 0.00% vs an average of 0.23% in the monolithic system. These results show that the microservices design performs better than the monolithic architecture.

REFERENCES

[1] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform*. O'Reilly Media, 2020.

[2] J. D. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, *Performance Testing Guidance for Web Applications*, 1st ed. Microsoft Press, 2007.

[3] A. Mili and F. Tchier, *Software Testing: Concepts and Operations*, 1st ed. John Wiley & Sons, 2015.

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

[5] A. Ismail, A. Y. Ananta, S. N. Arief, and E. N. Hamdana, "Performance Testing Sistem Ujian Online Menggunakan Jmeter Pada Lingkungan Virtual," *J. Inform. Polinema*, no. Vol. 9 No. 2 (2023): Vol 9 No 2 (2023), pp. 159–164, 2023, [Online]. Available: https://jurnal.polinema.ac.id/index.php/jip/article/view/1728/1347

[6] R. Mufrizal and D. Indarti, "Refactoring Arsitektur Microservice Pada Aplikasi Absensi PT. Graha Usaha Teknik," *J. Nas. Teknol. dan Sist. Inf.*, vol. 5, no. 1, pp. 57–68, Apr. 2019, doi: 10.25077/TEKNOSI.v5i1.2019.57-68.

[7] Riyanto, Irman Hermadi, and Yani Nurhadryani, "Analisis Uji Performa Aplikasi Dari Hasil Implementasi Refactoring Arsitektur Monolitik Ke Mikroservis dengan Decomposition dan Strangler Pattern," *J. Sist. Cerdas*, vol. 6, no. 3, pp. 189–203, Dec. 2023, doi: 10.37396/jsc.v6i3.352.

[8] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From Monolithic Systems to Microservices: A Comparative Study of Performance," *Appl. Sci.*, vol. 10, no. 17, p. 5797, Aug. 2020, doi: 10.3390/app10175797.

[9] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Inf. Softw. Technol.*, vol. 137, p. 106600, Sep. 2021, doi: 10.1016/j.infsof.2021.106600.

[10] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating Towards Microservice Architectures: An Industrial Survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2018, pp. 29–2909. doi: 10.1109/ICSA.2018.00012.

[11] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann, "Microservices Migration in Industry: Intentions, Strategies, and Challenges," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 481–490. doi: 10.1109/ICSME.2019.00081.

[12] D. Wolfart *et al.*, "Modernizing Legacy Systems with Microservices: A Roadmap," in *Evaluation and Assessment in Software Engineering*, Jun. 2021, pp. 149–159. doi: 10.1145/3463274.3463334.

[13] A. de Camargo, I. Salvadori, R. dos S. Mello, and F. Siqueira, "An architecture to automate performance tests on microservices," in *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, Nov. 2016, pp. 422–429. doi: 10.1145/3011141.3011179.

[14] M. Bolanowski *et al.*, "Eficiency of REST and gRPC realizing communication tasks in microservice-based ecosystems," Aug. 2022, doi: 10.3233/FAIA220242.

[15] R. Chen, S. Li, and Z. Li, "From Monolith to Microservices: A Dataflow-Driven Approach," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2017, pp. 466–475. doi: 10.1109/APSEC.2017.53.

[16] C.-Y. Li, S.-P. Ma, and T.-W. Lu, "Microservice Migration Using Strangler Fig Pattern: A Case Study on the Green Button System," in *2020 International Computer Symposium (ICS)*, Dec. 2020, pp. 519–524. doi: 10.1109/ICS51289.2020.00107.

[17] L. Khoirunnisa', "Rancang Bangun Sistem E-Learning Berbasis Microservices dan Domain Driven

Design (Studi Kasus Probistek UIN Maulana Malik Ibrahim Malang)," UIN Maulana Malik Ibrahim Malang, 2019. [Online]. Available: http://etheses.uin-malang.ac.id/15301/1/14650045.pdf
[18] H. Knoche and W. Hasselbring, "Using Microservices for Legacy Software Modernization," *IEEE Softw.*, vol. 35, no. 3, pp. 44–49, May 2018, doi: 10.1109/MS.2018.2141035.
[19] K. S. Swarnalatha, A. Mallya, G. Mukund, and R. Ujwal Bharadwaj, "Solving Problems of Large Codebases: Uber's Approach Using Microservice Architecture," 2023, pp. 653–662. doi: 10.1007/978-981-19-5482-5_57.
[20] S. Li *et al.*, "A dataflow-driven approach to identifying microservices from monolithic applications," *J. Syst. Softw.*, vol. 157, p. 110380, Nov. 2019, doi: 10.1016/j.jss.2019.07.008.
[21] K. Sharma *et al.*, "Review of Software Architecture and Design Practices: Current Trends and Future Direction," *Tuijin Jishu/Journal Propuls. Technol.*, vol. 43, no. 4, pp. 1001–4055, 2022, doi: 10.52783/tjjpt.v43.i4.2341.
[22] A. Trichur Ramachandran, Abhishek, Mamatha, Rashmi, Badrinath, and M. Parmar, "Understanding Migration from Monolithic to Microservice Architecture and its Challenges," *Int. J. Sci. Res. Eng. Dev.*, vol. 4, no. 3, pp. 1742–1752, 2021, [Online]. Available: https://ijsred.com/volume4/issue3/IJSRED-V4I3P243.pdf
[23] O. Al-Debagy and P. Martinek, "A Comparative Review of Microservices and Monolithic Architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, Nov. 2018, pp. 000149–000154. doi: 10.1109/CINTI.2018.8928192.
[24] I. Molyneaux, *The Art of Application Performance Testing: From Strategy to Tools*, 1st ed. O'Reilly Media, 2009.
[25] P. Tanuska, O. Vlkovic, and L. Spendla, "The Usage of Performance Testing for Information Systems," *Int. J. Comput. Theory Eng.*, pp. 144–147, 2012, doi: 10.7763/IJCTE.2012.V4.439.
[26] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," in *2020 IEEE XVIth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, Apr. 2020, pp. 150–153. doi: 10.1109/MEMSTECH49584.2020.9109514.
[27] P. Jatkiewicz and S. Okrój, "Differences in performance, scalability, and cost of using microservice and monolithic architecture," in *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, Mar. 2023, pp. 1038–1041. doi: 10.1145/3555776.3578725.
[28] 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. dan Teknol. Informasi)*, vol. 9, no. 3, pp. 594–600, Jun. 2025, doi: 10.29207/resti.v9i3.6183.
[29] T. R. H. Barzotto and K. Farias, "Evaluation of the impacts of decomposing a monolithic application into microservices: A case study," Aug. 2022, [Online]. Available: http://arxiv.org/abs/2203.13878
[30] C. M. Aderaldo, N. C. Mendonca, C. Pahl, and P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, May 2017, pp. 8–13. doi: 10.1109/ECASE.2017.4.
[31] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does Migrate a Monolithic System to Microservices Decrease the Technical Debt?," Jul. 2020, doi: 10.1016/j.jss.2020.110710.
[32] H. M. AlGhamdi, C. Bezemer, W. Shang, A. E. Hassan, and P. Flora, "Towards reducing the time needed for load testing," *J. Softw. Evol. Process*, vol. 35, no. 3, Mar. 2023, doi: 10.1002/smr.2276.
[33] Z. M. Jiang and A. E. Hassan, "A Survey on Load Testing of Large-Scale Software Systems," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1091–1118, Nov. 2015, doi: 10.1109/TSE.2015.2445340.
[34] J. F. Lombogia, A. Syahrina, and A. Munansyah, "Perancangan Arsitektur Perangkat Lunak Microservices pada Aplikasi Open Library Telkom University Menggunakan GRPC," Telkom University, 2021.