MIND (Multimedia Artificial Intelligent Networking Database) Journal

MIND Journal | ISSN (p): 2528-0015 | ISSN (e): 2528-0902 | Vol. 10 | No. 2 | Halaman 235 - 249
DOI : https://doi.org/10.26760/mindjournal.v10i2.235-249 Desember 2025

Renovator: Resilience System in Microservices Architecture Using Fault Tolerance Factor

KURNIA RAMADHAN PUTRA, SOFIA UMAROH, NUR FITRIANTI FAHRUDIN, PRAMBUDHI WIBOWO PANDJI

Information System, Institut Teknologi Nasional Bandung, Indonesia
kurniaramadhan@itenas.ac.id

Received 23 Agustus 2025 **|** *Revised* 17 November 2025 **|** *Accepted* 25 November 2025

ABSTRAK

Arsitektur microservices menghadapi tantangan dalam menjaga ketahanan layanan akibat cascading failure, ketergantungan antarlayanan yang dinamis, serta keterbatasan mekanisme fault tolerance berbasis ambang statis. Pendekatan seperti circuit breaker dan bulkhead hanya memberi perlindungan terbatas ketika beban dan latensi berubah cepat. Untuk menjawab gap tersebut, penelitian ini memperkenalkan Renovator, kerangka kerja ketahanan yang memperluas circuit breaker melalui pemantauan adaptif dan pemulihan otomatis. Evaluasi pada simulasi Sistem Presensi dengan empat skenario yaitu cascading failure, latency spike, normal load, dan single-service failure menunjukkan peningkatan signifikan dibandingkan baseline: ketersediaan naik (86,31% menjadi 93,95%), MTTR berkurang 49–67%, tingkat kesalahan turun 55–63%, serta latensi membaik 20–27% tanpa memengaruhi throughput. Kontribusi utama penelitian ini adalah pengembangan Renovator sebagai mekanisme circuit breaker yang lebih adaptif dan otomatis untuk meningkatkan ketahanan microservices.

***Kata kunci**: microservices, fault tolerance, circuit breaker, MSA resilience*

ABSTRACT

Microservices architectures face resilience challenges due to cascading failures, dynamic dependencies, and the limitations of fault tolerance mechanisms that rely on static thresholds. Techniques such as Circuit Breaker and Bulkhead provide only partial protection under rapidly changing workloads. To address this gap, this study introduces Renovator, a resilience framework that enhances circuit breaker functionality through adaptive monitoring and automated recovery. Evaluated on a simulated Attendance System under four scenarios—cascading failure, latency spike, normal load, and single-service failure, Renovator shows notable improvements over the baseline: availability increases (from 86.31% to 93.95%), MTTR decreases by 49–67%, error rates drop 55–63%, and latency improves 20–27% with no throughput degradation. The main contribution is an adaptive and automated circuit-breaker–based framework to strengthen microservices resilience.

***Keywords**: microservices, fault tolerance, circuit breaker, MSA resilience*

1. INTRODUCTION

In building an information system, several stages are involved, including understanding, analysis, design, implementation, and testing **(Kopnova et al., 2022)**. In academic environments, information systems such as academic portals, library systems, financial systems, and presence systems play a crucial role in supporting institutional operations **(Munthe et al., 2024)**. However, the architectural choices underlying these systems significantly affect their reliability and ability to withstand failures.

The Presence System supports students and lecturers in recording attendance, which is then stored in the Academic Information System (Sikad). Based on interviews with the Information Communication Technology-Integrated Service Unit (UPT-TIK) and system observations reveal that the current Presence System is built on a monolithic architecture. This architectural style violates key microservices principles such as design for failure and the ability to isolate faults among interdependent components **(Auer et al., 2021; Gos & Zabierowski, 2020)**. An insident during the 2022 National Defense Awareness Education (PKBN) program demonstrated this limitation: Sikad experienced a major downtime while thousands of students were recording attendance, forcing manual data collection. This illustrated the low resilience of the existing system, primaly caused by thighly coupled modules typical of monolithic designs **(Mooghala, 2023)**.

System resilience refers to the ability to maintain acceptable service performance during disruptions and to restore normal operations afterward **(Kaloudis, 2024)**. Microservices architectures inherently provide higher resilience through loose coupling, service independence, and fault isolation **(Mailewa et al., 2025; Soldani et al., 2025)**. A widely adopted mechanism to enhance resilience is the circuit breaker pattern, which prevents cascading failures, limits error propagation, and allows degraded services to recover gracefully **(Asrowardi et al., 2020)**. However, conventional circuit breaker implementations rely on static thresholds and predefined states, making them less effective in dynamic microservices environments where workload fluctuations and dependency chains frequently change. These limitations create a significant research gap regarding the need for adaptive, intelligent, and automated fault tolerance mechanisms capable of responding to volatile system conditions.

Fault tolerance is a key requirement in microservices to handle failures without disrupting overall system operation **(Hosea et al., 2021).** One effective fault tolerance mechanism is the circuit breaker pattern, which prevents cascading failures and allows services to recover gracefully **(Falahah et al., 2021; Hosea et al., 2021)**. By implementing a circuit breaker, microservices can monitor failures, stop repeated calls to failing services, and provide fallback mechanisms, thereby enhancing overall system resilience**.** In addition, circuit breakers contribute to stability under dynamic workloads, since they allow degraded services to isolate themselves temporarily while recovery is attempted, rather than overwhelming the entire architecture. However, conventional circuit breaker implementations are often static and threshold-based, which can be insufficient in highly volatile environments where service latency and dependency chains fluctuate rapidly. This limitation highlights the need for adaptive and intelligent approaches that can extend the traditional circuit breaker model to achieve higher availability and fault tolerance in microservices-based systems **(Zhang et al., 2025)**. These limitations create a significant research gap regarding the need for adaptive, intelligent, and automated fault tolerance mechanisms capable of responding to volatile system conditions

Based on these considerations, this research focuses on redesigning the Presence System using a microservices architecture enhanced with a circuit-breaker-based fault-tolerance

mechanism to improve reliability and availability under various failure scenarios. The novelty and primary contribution of this research lie in the development of the Renovator framework, which extends the traditional circuit breaker into an adaptive and automated mechanism with dynamic monitoring and intelligent recovery strategies, advancing the current state of the art in microservices fault tolerance.

Research Questions

RQ1: How does the circuit breaker pattern affect the availability and reliability of the Presence System under various fault scenarios?

RQ2: How does the circuit breaker pattern influence mean time to recovery (MTTR) and error propagation between microservices in the Presence System?

Significance

The significance of this study lies in:

a. Practical contribution: Providing an architecture that enhances the resilience of academic systems, reducing risks of downtime during critical academic activities.
b. Theoretical contribution: Strengthening the understanding of fault tolerance patterns in microservices, particularly the circuit breaker, in improving availability and recovery metrics.
c. Institutional impact: Supporting Itenas in transitioning from monolithic to microservices-based systems, ensuring reliable academic services for students and lecturers.

2. RESEARCH METHODOLOGY

This study employs a design science research methodology to architect and evaluate a resilient microservices-based system. The research process follows a systematic sequence consisting of problem identification, solution design, implementation, and evaluation. The methodological foundation is guided by the Microservice Migration Pattern (MMP) framework **(Balalaie et al., 2018)**, chosen for its comprehensive step-by-step refactoring strategy from monolithic systems toward production-ready microservices. Compared with other approaches such as the Strangler Pattern or blue–green deployment strategies, MMP provides clearer operational granularity, explicit migration checkpoints, and standardized patterns suitable for academic system modernization. The overall research flow is illustrated in Figure 1.

2.1. Problem Identification and Requirements Analysis

The research commenced with a comprehensive analysis to define the problem space and establish clear requirements. This involved:

a. Literature Review: Conducting a state-of-the-art analysis to identify challenges in monolithic architectures and best practices for microservice resilience, fault tolerance, and refactoring patterns.
b. Structured Interviews: Engaging with key stakeholders (UPT TIK Itenas) to understand the real-world limitations, pain points, and failure modes of the existing attendance system.
c. System Observation: Technically profiling the current system's architecture, dependencies, and integration points to identify specific vulnerabilities. The output of this phase was a synthesized set of core resilience requirements that the proposed solution must address.

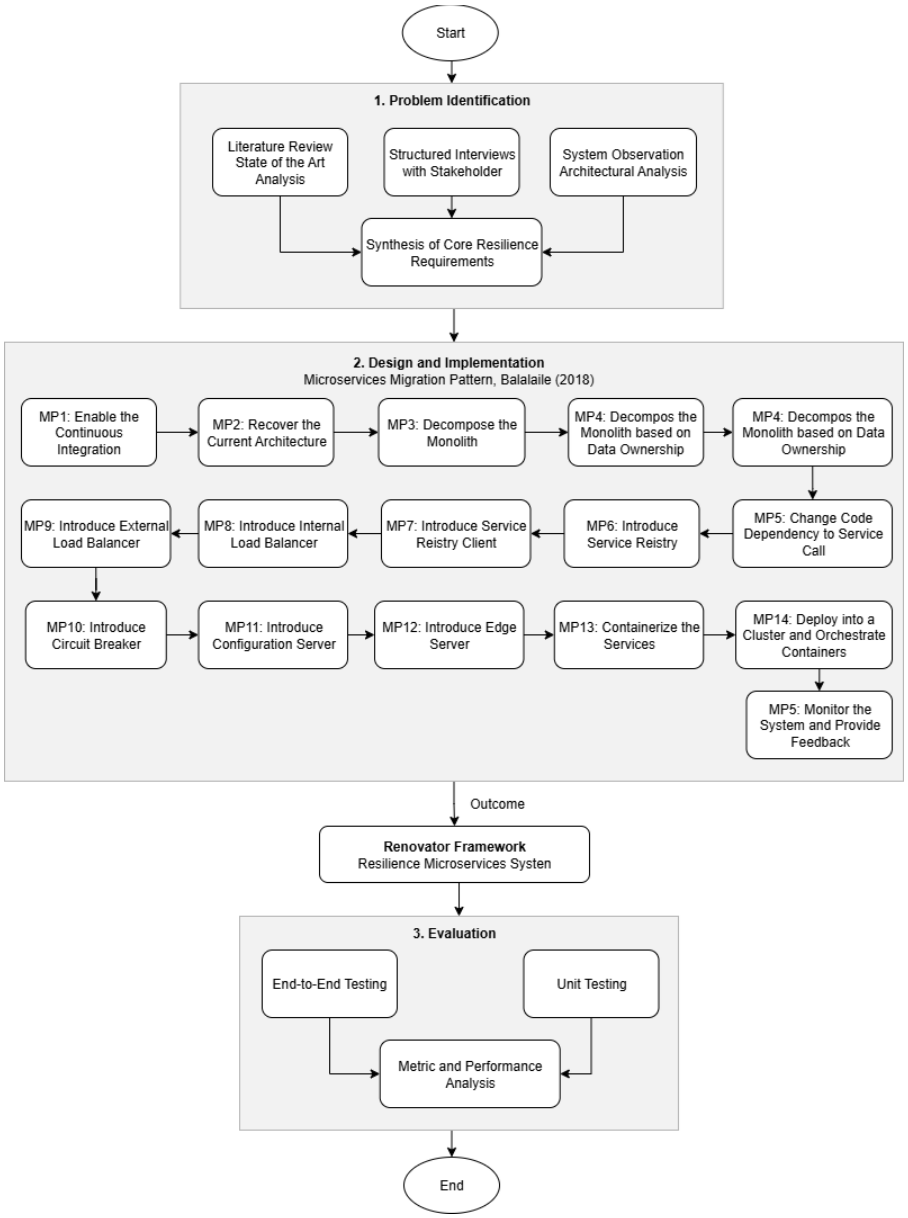

Figure 1. Research Flow

2.2. Solution Design

The microservices architecture was designed using the Microservice Migration Pattern (MMP). This framework consists of 15 sequential migration patterns, from MP1 (Continuous Integration) to MP15 (Monitoring and Feedback). This study adopted all 15 patterns, ensuring completeness in transitioning the existing monolithic system to a resilient microservices architecture.

MP1: Enable Continuous Integration

With the increase in the number of services in a microservices architecture, ensuring that every service produces production-ready artifacts consistently is essential. To address this, continuous integration pipelines were established to automate build and testing processes. Tools such as Gitlab CI 16.0, Jenkins 3.0, Nexus Repository Manager 3.49, Artifactory 8.40,

Travis CI, Bamboo 8.0, and Teamcity 2023.11 were employed to maintain a reliable and repeatable artifact delivery process.

MP2: Recover the Current Architecture
Before migrating to microservices, a clear understanding of the existing system was necessary. The architecture was documented in both textual and visual forms, including component diagrams, service interactions, technology stacks, and deployment procedures. Trello was used for textual documentation, while UML diagrams were created using Astah and draw.io to provide a comprehensive view for planning the migration.

MP3: Decompose the Monolithic
The monolithic system was divided into smaller, deployable units using Domain-Driven Design (DDD). Business subdomains were identified and defined as bounded contexts, which served as independent services. Java 17, Spring Boot 3.2, and Context Mapper 6.10.0 supported the decomposition process and service scaffolding.

MP4: Decompose the Monolithic based on Data Ownership
Further refinement of services was guided by data ownership. Cohesive sets of data entities were grouped into units with a single owner to ensure clarity in responsibility and reduce inter-service coupling. Spring Data JPA 3.2, MySQL 8.1, and PHPMyAdmin 5.2.2 were used to model and manage the data schemas accordingly.

MP5: Change Code Dependency to Service Call
To prevent build failures caused by shared code dependencies, service communication was refactored into RESTful API calls. OpenAPI/Swagger specifications 3.1 were used to define interfaces and facilitate loose coupling between services, maintaining independent development and deployment cycles

MP6: Introduce Service Registry Server
Dynamic service discovery was implemented through a registry server that maintains the addresses of all service instances. Each service registers itself upon startup, enabling clients and edge servers to locate instances dynamically. Eureka 3.1.3, Consul 1.51, Apache Zookeeper 3.8, and Spring Cloud Netflix Eureka server 3.2 supported this registry functionality.

MP7: Introduce Service Registry Client
Each service was equipped with a registry client to automatically register itself with the service registry upon deployment. This allowed the registry to be aware of new or terminated instances in real time, ensuring accurate service discovery. Eureka Client and Spring Cloud Netflix 3.2 components facilitated this mechanism.

MP8: Introduce Internal Load Balancer
Client-side load balancing was implemented within each service to distribute requests among available instances dynamically. Ribbon 2.10 and Spring Cloud Load Balancer 2023.0 were employed to fetch instance information from the service registry and balance the load without relying on external infrastructure.

MP9: Introduce External Load Balancer
An external load balancer was added to provide centralized load distribution with minimal changes to service code. Nginx 1.24 retrieved the list of service instances from the registry and applied a balancing algorithm to efficiently route traffic, ensuring system scalability and reliability.

MP10: Introduce Circuit Breaker
To improve system resilience, circuit breakers were applied to service calls. This mechanism enabled fast failure detection, fallback handling, and fault isolation for services that became unavailable. Hystrix 1.5, Resilience4j 2.1.0, and Spring Boot 3.2 were used to implement adaptive thresholds and monitoring capabilities.

MP11: Introduce Configuration Server
To enable configuration updates without redeployment, a centralized configuration server was established. Service configurations were stored separately from code repositories, allowing dynamic updates during runtime. Spring Cloud Config 2023.0.1, Archaius 2.0, and Git repositories supported this approach.

MP12: Introduce Edge Server
To hide the internal complexity of microservices and provide a single entry point for clients, an edge gateway was deployed. It handled routing, filtering, and monitoring of incoming requests. Spring Cloud Gateway 3.2 and Zuul 2.4 were used to implement this layer.

MP13: Containerize the Service
Each service was packaged into containers to ensure consistent environments across development and production. Docker 25.0, Docker Compose 2.20, and the Jib plugin facilitated containerization, allowing each service to operate in isolation with its required dependencies

MP14: Deploy into a Cluster and Orchestrate Containers
Service instances were deployed into a Kubernetes cluster, enabling scalable management and automated orchestration of multiple services. Helm 3.12 templates simplified deployment workflows, and Minikube 1.28 provided a local simulation environment for testing before production deployment.

MP15: Monitor the System and Provide Feedback
Comprehensive monitoring was established for each service to track performance, resource usage, and potential failures. Metrics and logs were collected and visualized using Collectd 5.12, Prometheus 2.44, ELK Stack 8.13.3, and Grafana 10. The monitoring data provided actionable feedback for continuous system improvement and operational decision-making.

All experiments were conducted in a controlled local cluster environment with the following specification can be seen in Table 1.

Table 1. Environment Configuration Specifications

Component	Description
Host Machine	Intel(R) Core(TM) i7-10750H CPU @2.60GHz (12 CPUs)
OS	Ubuntu 22.04 LTS (64-bit)
Container Runtime	Docker
Cluster	Minikube: 4 CPUs, 8 GB RAM, Containerd runtime
Network	Local Virtual Network

2.3. Evaluation and Validation
After implementing the proposed architecture and resilience framework, a comprehensive evaluation was conducted to compare performance against a baseline microservices system without resilience enhancements. The evaluation process consisted of three major aspects:

1. Evaluation Scenarios

Evaluation based on experimental scenarios were executed such as:

a. Cascading Failure: triggered by shutting down provider services sequentially
b. Latency Spike: induced artificial delays (300–800 ms)
c. Normal Load: 100–300 requests per second
d. Single-Service Failure: one microservice taken offline abruptly

2. Functional Testing

Functional correctness was verified through Unit Tests and End-to-End (E2E) Tests. Unit Tests validated the behavior of individual microservices, while E2E Tests ensured that the overall business workflows executed correctly, both under normal operating conditions and during partial failures. Some of the tools used for functionality testing are:

a. Unit tests: JUnit 5
b. End-to-end tests: executed through Postman
c. Failure simulation: chaos injection script using Chaos Monkey for Spring Boot
d. Load generation: Jmeter 5.6 and Locust 2.29

3. Metrics and Performance Analysis

a. Availability (A)

Availability measures the proportion of time the system remains operational and accessible, as defined in Equation (1).

$$A = \frac{Uptime}{Uptime + Downtime} \; x \; 100\% \tag{1}$$

Where, $Uptime$ is the total duration of correct operation, and $Downtime$ is the accumulated duration of service unavailability.

b. Mean Time to Recovery (MTTR)

MTTR represents the average time required to restore system functionality after a failure, as shown in Equation (2).

$$MTTR = \frac{\sum(Recovery\ Time)}{Number\ of\ Failures} \tag{2}$$

Where, $Recovery\ Time$ is the elapsed time between failure occurrence and full recovery.

c. Error Rate (ER)

Error Rate quantifies the percentage of failed requests relative to the total number of requests, as expressed in Equation (3).

$$ER = \frac{Failed\ Request}{Total\ Request} \; x \; 100\% \tag{3}$$

Where, $Failed\ Requests$ is the number of requests resulting in error responses, and $Total\ Requests$ is the overall number of incoming requests.

d. Throughput (T)

Throughput captures the volume of successfully processed requests per unit of time, as given in Equation (4).

$$T = \frac{Successful\ Request}{Elapsed\ Time} \tag{4}$$

Where, $Successful\ Requests$ is the number of requests completed without error, and $Elapsed\ Time$ is the duration of the measurement interval.

e. Latency (L)
Latency represents the average response time per request, as defined in Equation (5).

$$L = \frac{\sum(Response\ Time)}{Number\ of\ Requests} \quad (5)$$

Where, $Response\ Time$ is the time elapsed from sending a request until receiving the corresponding response, and $Number\ of\ Requests$ is the total requests measured.

3. RESULT AND DISCUSSION

Designing microservices using Microservices Migration Pattern (MMP) method introduced by Balalaile et al (2018) requires a Situational Method Engineering (SME) approach because there are various factors such as different needs and expertise of teams in different scenarios and companies, so that the methodology is unique, no rigid, and suitable for inadequate resources. So a needs analysis is carried out based on the results of interviews and observations to select a pattern for the microservice migration pattern and compose it into a design flow. The composition and sequence of MMP stages are illustrated in Figure 2.

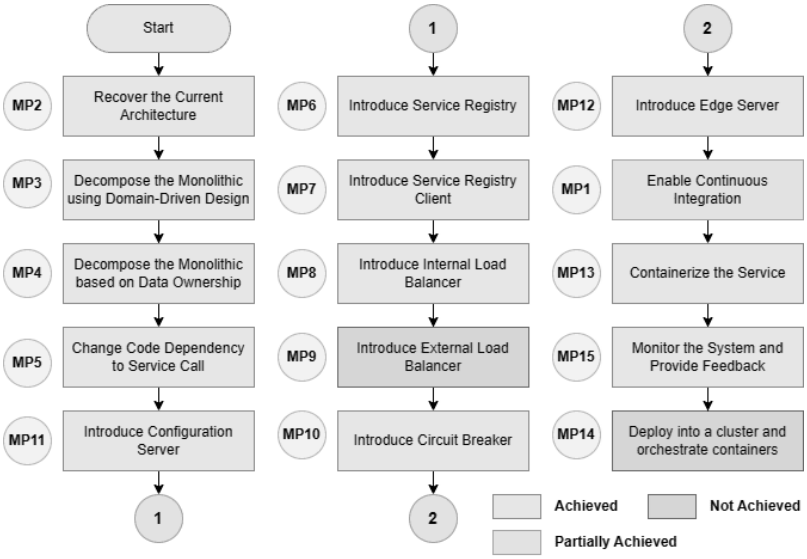

Figure 2. Implementation of MMP's Stages

Information:
a. Achieved: the Microservices Migration Pattern stages were successfully carried out.
b. Partially Achieved: the code repository has been successfully implemented, namely using Github, for the continuous integration server, namely Jenkins, which has been successfully installed, but cannot yet be configured and connected to Github.
c. Not Achieved: External load balancer implemented if we want to use public IP, and deploy the services to the cloud, but in this research just simulation in localhost. Deployment to

the orchestration server has not been successful because the service is paid using a credit card, that is Google Kuberneted Engine.

3.1 The Current Design of Presence System Architecture (As-is System)

The current architectural analysis is carried out using three approaches, namely Component and Service Architecture Analysis to analyze what components and services are in the system. Then the Technology Architecture Analysis approach is to analyze the technology used in system development and the final approach is Deployment Architecture Analysis to analyze how the system installation is carried out, which can be seen in Figure 3.

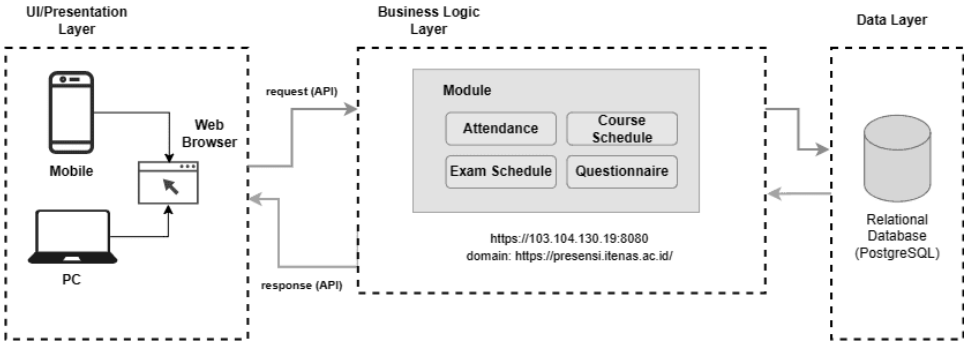

Figure 3. The Current Architecure

The current architecture of the Attendance System at Itenas follows a three-layered structure consisting of the UI/Presentation Layer, the Business Logic Layer, and the Data Layer. Overall, while the current architecture is functional for small-to-medium workloads, its monolithic and centralized nature limits its ability to address challenges in resilience, scalability, and dynamic workload management challenges that the proposed Renovator architecture is explicitly designed to overcome.

3.2 The Proposed Design of Presence System Architecture (To-be System)

The proposed system, named Renovator, was designed to enhance resilience, reliability, and availability in microservices-based systems. Key stages include:

a. Objective & Scope: Focus on backend layers (Business and Data Layer) to improve reliability and availability.
b. Architecture Principles: fault-tolerance, resilience, and load balancing.
c. High-Level Design: Cloud-based infrastructure for distributed data management.
d. Detailed Design: Microservices independently deployable and scalable; API gateways for traffic management; clustered servers for failover.
e. Technology Stack: Databases (PostgreSQL, MongoDB, and Redis), API Gateway (Spring Cloud Gateway), Application Framework (Spring Boot 3.2), DevOps (Jenkins 2.413), Containerization (Docker 24), and Monitoring & Observability (Prometheus 2.49 and Grafana 10)

The proposed design of Presence System, can be seen in Figure 4.

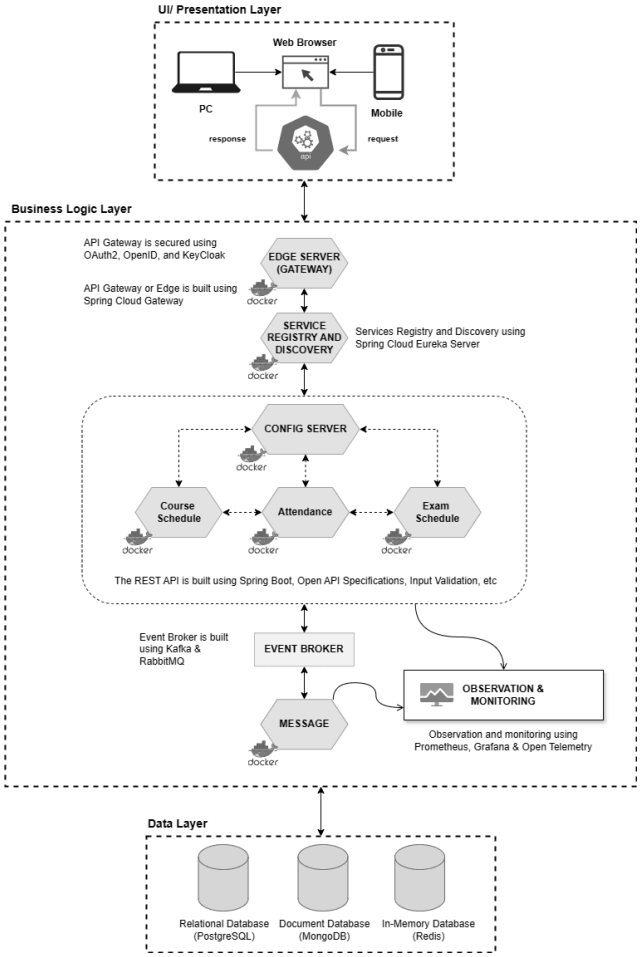

Figure 4. The Proposed Architecture (Renovator)

The proposed architecture, named Renovator, is designed to enhance the resilience of microservices-based systems by incorporating an adaptive fault tolerance Overall, the Renovator architecture goes beyond the adoption of standard microservices migration patterns by embedding an adaptive fault tolerance framework. Through the integration of event-driven communication, observability mechanisms, and automated recovery strategies, the system demonstrates improved service availability, reduced mean time to recovery (MTTR), and significantly lower error rates. These characteristics underscore Renovator's potential to strengthen resilience in microservices architectures, as validated in the experimental evaluation on the Attendance System simulation at Itenas.

3.3 Evaluation and Validation

End-to-end Testing

End-to-end testing is carried out by turning on microservices that are related to each other, but run independently on the Presence System, such as the Attendance microservice, Course Schedule microservice, and Exam Schedule microservice. Then testing was carried out on the Course Schedule microservice, by accessing one of the student data via registration number using tool Postman, it can be seen in Table 2 that the data is displayed well.

Table 2. End-to-End testing on Course Schedule Microservice

Endpoint	Method	Status
http://localhost:8072/presence-system/schedule/course	POST	200 (OK)

Request:

```
{
    "regNumber": 162019018
}
```

Response:

```
[
    {
        "id": 1,
        "regNumber": 162019018,
        "code": "ISB-404",
        "class": "AA",
        "day": "SENIN",
        "time": "10:00",
        "teacher": "NUR FITRIANTI FAHRUDIN",
        "room": "04-109",
        "activity": "KULIAH",
        "createDt": "2023-07-31"
    }
]
```

Unit Testing

In unit testing, testing is carried out by turning off one of the Exam Schedule microservice, so in Prometheus can be seen that the microservice is down, but the other microservices are still running well, as shown in Figure 5.

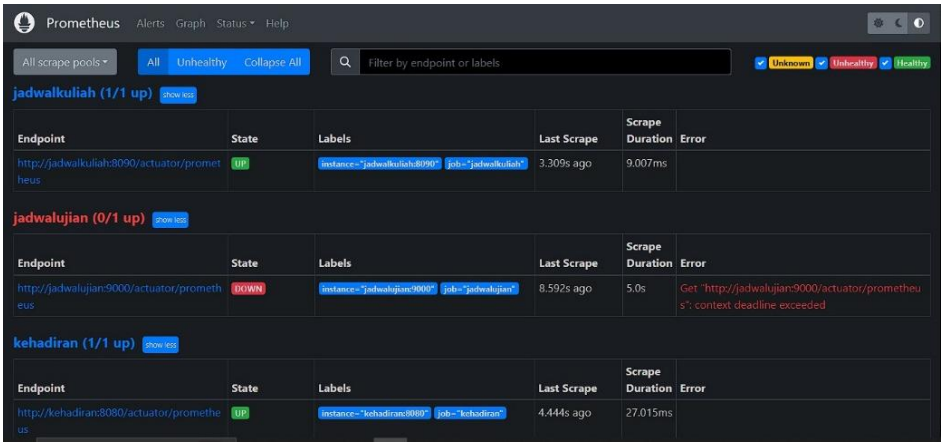

Figure 5. Microservices Status in Prometheus Dashboard

Metrics and Performance Analysis

All performance metrics were systematically collected during the simulations and exported into a structured comma-separated values (CSV) file, named Simulation_Dataset.csv. This dataset contains a total of 96 rows (4 scenarios x 2 variants x 12 runs), with each row representing a single experimental run.

A detailed description of each column in the dataset is provided in Table 3.

Table 3. The Dataset's Columns Description

Columns	Description
Scenario	Type of simulated incident: Normal Load, Latency Spike, Single-Service Failure
Variant	Basline (without optimization), Renovator (with recovery and optimization mechanism)
Run	Experiment run number, from 1 to 12
Uptime (s)	System operating time without disruption (in seconds)
Downtime (s)	System unavailability period (failed to respond)
Requests_total	Total number of requests sent per run
Requests_failed	Number of failed requests
Throughput (req/s)	Number of requests processed per second
Latency_1 to Latency_50	Response times (in milliseconds) of 50 randomly sampled requests measured per run

Metrics collected under four scenarios (Normal Load, Latency Spike, Single-Service Failure, Cascading Failure) for two system variants (Baseline vs. Renovator) with 12 independent runs each. Metrics include: Availability, MTTR, Error Rate, Throughput, and Latency (Avg & P95) as shown in Figure 6.

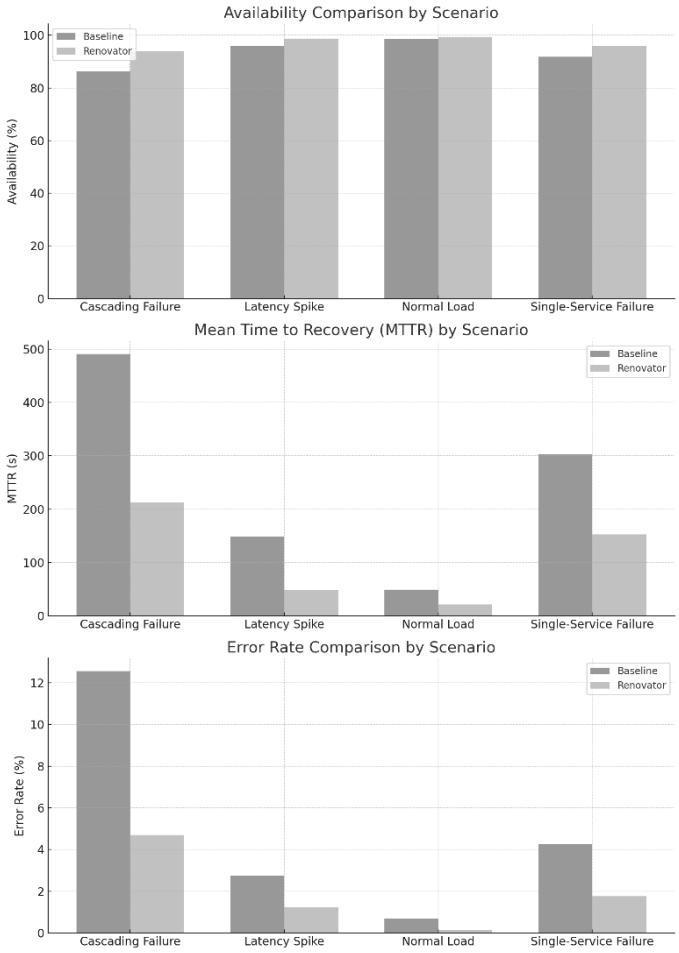

Figure 6. Performance Metrics By Scenario and Variant

The results, summarized in Table 4, provide a comprehensive comparison of system behavior under stress, highlighting the impact of proactive resilience strategies on overall system reliability and responsiveness.

Table 4. Summary of Performance Metrics by Scenario and Variant

Scenario	Variant	Availability (%)	MTTR (s)	Error Rate (%)	Throughput (req/s)	Avg Latency (ms)	P95 Latency (ms)
Cascading failure	Baseline	86,31	491,02	12,55	1003,87	519,68	548,29
Cascading failure	Renovator	93,95	212,51	4,70	1025,32	379,98	411,86
Latency spike	Baseline	95,89	147,92	2,76	1000,24	359,90	384,37
Latency spike	Renovator	98,67	48,15	1,23	997,40	279,66	304,94
Normal load	Baseline	98,65	48,74	0,68	998,68	180,08	192,64
Normal load	Renovator	99,41	21,42	0,15	996,60	170,15	182,49
Single-service failure	Baseline	91,78	302,96	4,25	977,65	320,19	342,80
Single-service failure	Renovator	95,82	152,89	1,77	986,12	249,55	271,99

The evaluation results show that Renovator consistently improves system resilience across all scenarios, with availability rising from 86.31% to 93.95% under Cascading Failure, MTTR decreasing by 56.7% through automated recovery, and error rates dropping by up to 62.5% due to adaptive monitoring and improved fault isolation. In addition, Renovator reduces latency by 20–27% while maintaining stable throughput, confirming that the framework introduces minimal performance overhead. These findings align with recent microservices resiliency research (**Hosea et al., 2021; Mooghala, 2023**), reinforcing that the integration of adaptive monitoring and automated recovery can substantially enhance microservices resilience without compromising throughput.

4. CONCLUSION

The results of this study demonstrate that the proposed Renovator framework effectively enhances microservices resilience by extending traditional circuit breaker mechanisms with adaptive monitoring and automated recovery, enabling faster failure detection, improved isolation, and more graceful service degradation. Experimental evaluation across multiple failure and load scenarios confirms consistent improvements in system stability, reduction of error propagation, and resilience gains achieved without imposing meaningful overhead on throughput or performance. This work contributes scientifically by showing that adaptive and self-healing mechanisms can significantly strengthen distributed architectures with complex inter-service dependencies, aligning with contemporary microservices resiliency theories from 2023–2025. Future research will focus on production-scale evaluation under real-world traffic, integration with auto-scaling and bulkhead isolation, incorporation of machine learning–based anomaly detection for predictive recovery and validation using tools such as Chaos Mesh.

REFERENCES

Asrowardi, I., Putra, S. D., & Subyantoro, E. (2020). Designing microservice architectures for scalability and reliability in e-commerce. *Journal of Physics: Conference Series*, *1450*(1). https://doi.org/10.1088/1742-6596/1450/1/012077

Auer, F., Lenarduzzi, V., Felderer, M., & Taibi, D. (2021). From monolithic systems to Microservices: An assessment framework. *Information and Software Technology*, *137*(4), 106600. https://doi.org/10.1016/j.infsof.2021.106600

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., & Lynn, T. (2018). Microservices migration patterns. *Software - Practice and Experience*, *48*(11), 2019–2042. https://doi.org/10.1002/spe.2608

Falahah, Surendro, K., & Sunindyo, W. D. (2021). Circuit Breaker in Microservices: State of the Art and Future Prospects. *IOP Conference Series: Materials Science and Engineering*, *1077*(1), 012065. https://doi.org/10.1088/1757-899x/1077/1/012065

Gos, K., & Zabierowski, W. (2020). The Comparison of Microservice and Monolithic Architecture. *International Conference on Perspective Technologies and Methods in MEMS Design*, *June*, 150–153. https://doi.org/10.1109/MEMSTECH49584.2020.9109514

Hosea, E., Palit, H. N., & Dewi, L. P. (2021). Fault Tolerance pada Microservice Architecture dengan Circuit Breaker dan Bulkhead Pattern. *Jurnal Infra*, *9*(12). https://doi.org/10.116/j.procs.2019.09.271

Kaloudis, M. (2024). Evolving Software Architectures from Monolithic Systems to Resilient Microservices : Best Practices , Challenges and Future Trends. *(IJACSA) International Journal of Advanced Computer Science and Applications*, *15*(9), 1–10. https://dx.doi.org/10.14569/IJACSA.2024.0150901

Kopnova, O., Shaporeva, A., Iklassova, K., Kushumbayev, A., Tadzhigitov, A., & Aitymova, A. (2022). Building an Information Analysis System Within a Corporate Information System for Combining and Structuring Organization Data (on the Example of a University). *Eastern-European Journal of Enterprise Technologies*, *6*(2–120), 20–29. https://doi.org/10.15587/1729-4061.2022.267893

Mailewa, A. B., Akuthota, A., & Mohottalalage, T. M. D. (2025). A Review of Resilience Testing in Microservices Architectures: Implementing Chaos Engineering for Fault Tolerance and System Reliability. *2025 IEEE 15th Annual Computing and Communication Workshop and Conference, CCWC 2025*, *January*, 236–242. https://doi.org/10.1109/CCWC62904.2025.10903891

Mooghala, S. (2023). A Comprehensive Study of the Transition from Monolithic to Micro

services-Based Software Architectures. *Journal of Technology and Systems*, *5*(2), 27–40. https://doi.org/10.47941/jts.1538

Munthe, R. G., Abbas, M., Fernandez, R., & Ulita, N. (2024). The Impact of Educational Technologies on Learning Outcomes in Higher Business Education. *Economic Annals*, *69*(241), 129–160. https://doi.org/10.2298/EKA2441129Z

Soldani, J., Forti, S., Roveroni, L., & Brogi, A. (2025). Explaining Microservices' Cascading Failures From Their Logs. *Software - Practice and Experience*, *55*(5), 809–828. https://doi.org/10.1002/spe.3400

Zhang, P., Xiang, L., Song, Z., & Yang, Y. (2025). Adaptive load balancing and fault-tolerant microservices architecture for high-availability web systems using docker and spring cloud. *Discover Applied Sciences*, *7*(7). https://doi.org/10.1007/s42452-025-07320-7