Assessing the Impact of Asynchronous Communication on Resilience and Robustness: A Comparative Study of Microservice and Monolithic Architectures

Nathanael Bosilia[1], Gerald Weinberger[1][0009−0004−2965−1998], and Philipp Haindl[1*][0000−0001−6075−5286]

St. Pölten University of Applied Sciences, St. Pölten, Austria
Department of Computer Science and Security
`{is211308,firstname.lastname}@fhstp.ac.at`
[*] Corresponding author

Abstract. *Context:* Microservice architectures offer advantages such as scalability and independent deployment compared to traditional monolithic systems. Nonetheless, the specific impact of asynchronous communication patterns within microservice architectures on critical quality attributes, such as resilience and robustness, particularly when directly compared to monolithic systems under failure conditions, remains insufficiently explored empirically. *Objective:* This study empirically examines the influence of asynchronous communication mechanisms on the resilience and robustness of microservice architectures in comparison to equivalent monolithic architectures, identifying associated benefits and operational trade-offs. *Method:* Two equivalent systems representing a simplified e-commerce order process were developed: a synchronous monolith and an event-driven microservice architecture utilizing asynchronous communication. Controlled Chaos Engineering experiments simulating service crashes were conducted under load. Key metrics, including failure rate, system availability, request duration, and processed requests, were measured and compared across architectures under identical fault scenarios. *Results:* The asynchronous microservice architecture demonstrated significantly enhanced resilience and robustness under simulated failures. Compared to the monolith, it exhibited lower failure rates (up to a 12% reduction observed), maintained higher availability, and processed requests more effectively during fault conditions. Asynchronous decoupling localized failures and facilitated continued partial operation. However, the microservice architecture exhibited increased operational complexity and higher resource requirements. *Conclusion:* Asynchronous communication substantially improves the resilience and robustness of microservice architectures against failures compared to traditional monolithic approaches. While offering significant fault tolerance advantages, the adoption necessitates careful consideration of increased operational complexity, resource consumption, and the need for sophisticated monitoring and debugging infrastructures.

Keywords: Microservice Architecture · Monolithic Architecture · Asynchronous Communication · Resilience · Robustness · Event-Driven Architecture · Chaos Engineering.

1 Introduction

In recent years, software architecture has undergone significant transformations, primarily driven by the demands for scalability, flexibility, and maintainability. Microservice architecture has emerged as a prominent paradigm, characterized by the decomposition of applications into small, independently deployable services that communicate via lightweight protocols, often utilizing asynchronous messaging patterns [9, 12]. Each microservice typically encapsulates a specific business capability and operates autonomously, enabling independent development, deployment, and scaling without impacting other services [11, 15].

Newman [15] notes that effective microservice implementations usually embrace automation, decentralized governance, failure isolation, and observability to maintain service independence. Similarly, Richardson [20] advocates for each microservice to own its data store, which bolsters service isolation by facilitating independent development and scaling. Moreover, Fowler [9] defines microservices as an approach to developing a single application as a suite of small services, each running in its own process and communicating through lightweight mechanisms, with minimal centralized management and flexibility to employ diverse programming languages and data storage technologies.

The autonomy conferred by microservice architecture allows organizations to adopt heterogeneous technology stacks and deploy updates rapidly, thereby enhancing agility and resilience in dynamic environments [4]. However, this architectural approach typically requires higher computational resource allocation. In our experimental setup, the microservice architecture required three independent service instances plus message broker infrastructure, with each service allocated 1 vCPU and 512 MiB RAM, resulting in a total resource footprint that exceeded the monolithic counterpart by a factor of approximately four when including the Apache Kafka messaging infrastructure. In contrast, monolithic architectures encapsulate all application functionality within a single, tightly coupled codebase [4]. All modules share a common memory space and are executed as a single process, which simplifies development, testing, and deployment for small to moderately complex systems [6]. However, as systems expand, monoliths often encounter scalability bottlenecks, maintenance challenges, and an increased risk of cascading failures, as a fault in one module can propagate throughout the entire system [19].

1.1 Motivation and Problem Statement

While microservices offer notable benefits in terms of scalability and independent deployment [19], they also introduce complexities, particularly concerning inter-service communication, consistency management, and failure handling [23].

Asynchronous communication, frequently implemented through message brokers (e.g., Apache Kafka), has been proposed as a solution to address the challenges associated with synchronous communication, including tight coupling and cascading failures [8, 18, 24]. Event-driven architectures and messaging patterns play a crucial role in reducing resource consumption, enhancing fault tolerance, and decoupling service interactions [5, 18].

Nevertheless, the practical implications of asynchronous communication on critical quality attributes such as resilience and robustness remain insufficiently examined, particularly in direct comparison to monolithic systems. Existing research predominantly concentrates on general performance and scalability comparisons. For instance, Blinowski *et al.* [4] and Horváth *et al.* [10] present evidence indicating that microservice architectures achieve superior scalability in high-load scenarios, albeit occasionally at the expense of increased latency due to network overhead. Rodrigues *et al.* [21] and Razzaq and Ghayyur [19] explore the broader implications of transitioning from monolithic to microservice systems, highlighting challenges such as operational complexity and inconsistent performance gains.

Berry *et al.* [2] emphasize that while microservices can enhance resource utilization through horizontal scaling, certain components may become bottlenecks if not adequately replicated. Di Francesco *et al.* [7] and Soldani *et al.* [23] further note that the industrial adoption of microservices often encounters significant challenges alongside the benefits, particularly in managing distributed failures and ensuring robustness. Early reports, such as those by Thönes [25] and Fowler [9], offered optimistic perspectives on microservice adoption; however, empirical evidence addressing fault tolerance remains limited.

1.2 Research Questions

This study seeks to fill this gap by empirically examining the impact of asynchronous communication mechanisms on the resilience and robustness of microservice architectures, in direct comparison to traditional monolithic architectures. The comparison between asynchronous microservices and synchronous monoliths is valuable as it represents the most common architectural transition scenario in practice, where organizations migrate from synchronous, tightly-coupled monolithic systems to event-driven, loosely-coupled microservice architectures. This architectural transformation fundamentally changes both the deployment model (single process vs. distributed) and the communication paradigm (synchronous vs. asynchronous), making it essential to understand the combined impact of these changes on system resilience and robustness. To this end, we have formulated the following research questions:

RQ1. What is the impact of architectural style, specifically microservices with asynchronous communication versus monolithic architecture, on system resilience in the face of failures?

RQ2. How does the choice of architectural style influence system robustness, such as error rates and response times under load, during simulated fault conditions?

RQ3. What operational challenges and trade-offs, including complexity and resource usage, are observed when comparing these two architectural approaches in this context?

Analyzing the behavior of these architectures under failure scenarios is essential for assessing their appropriateness for systems that require high availability and reliability [4, 10].

1.3 Structure of the Paper

The structure of this paper is organized as follows: Section 2 provides a review of pertinent scholarly literature. Section 3 elaborates on the methodology, encompassing the design of the experimental systems and the application of the Chaos Engineering approach. Section 4 presents the empirical findings obtained from the experiments, addressing each research question. Section 5 interprets these findings, connecting them to existing literature and practical implications. Potential threats to validity are examined in Section 6. Finally, Section 7 offers concluding remarks and proposes directions for future research.

2 Related Work

We have identified three principal streams of related research.

2.1 Comparative Performance and Scalability Studies

Studies have compared the performance, scalability, and costs of monolithic and microservice architectures. Blinowski *et al.* [4] evaluated monolithic and microservice systems in Java and C# .NET across deployment environments. Their load tests showed monolithic systems perform better on single machines, while microservices excel in distributed environments, though scaling beyond certain instances caused performance issues. Horváth *et al.* [10] conducted load testing on file-sharing systems, confirming microservices' superior scalability under dynamic loads but noted increased complexity and latency. Rodrigues *et al.* [21] reviewed performance and costs of transitioning from monolithic to microservice systems, finding conflicting results that necessitate context-specific evaluations. Villamizar *et al.* [26] compared deployment strategies in cloud environments, concluding microservices' benefits depend heavily on workload patterns and system design. A common limitation across these studies is the insufficient focus on resilience and robustness metrics under fault conditions, with most experiments being performance-driven rather than quality-attribute-driven. Another relevant study in this area is by Bjørndal *et al.* [3], which benchmarks a case study of migration from monolith to microservices.

2.2 Impact of Communication Patterns in Microservices

Research explores the impact of communication patterns on microservice systems, with a particular focus on asynchronous communication. Srijith *et al.* [24] investigated the use of Apache Kafka Connect for inter-service communication among microservices, suggesting pipeline optimization to alleviate bottlenecks. Their findings indicated that adjusting Kafka producers and consumers reduced CPU and memory overhead. Rahmatulloh *et al.* [18] evaluated event-driven architecture (EDA) in microservices, discovering that EDA systems achieved a 19.18% faster response time and a 34.40% lower error rate compared to API-driven architectures, albeit with increased CPU usage. Cabane and Farias [5] examined the performance impact of event-driven architectures, focusing on the trade-offs between complexity and performance. Fehling *et al.* [8] cataloged cloud computing patterns, including event-driven resilience patterns for asynchronous microservice systems. Limitations of these studies include a focus on performance over resilience, with limited evaluation of real-world fault tolerance.

2.3 Quality Attributes and Design Patterns in Microservices

Recent research has focused on the impact of microservices on quality attributes and design patterns for addressing quality attributes. Li *et al.* [12] conducted a systematic review, identifying six key quality attributes: scalability, performance, availability, monitorability, security, and testability. They synthesized architectural tactics, noting maintainability needs further investigation. Pinciroli *et al.* [16] used queuing networks to analyze seven microservices design patterns' performance. Their evaluation quantified pattern trade-offs, though large-scale validation was limited. Meijer *et al.* [13] provided experimental validation of architectural patterns, focusing on service latency and resource utilization. Their findings showed real-world performance aligns with model predictions, highlighting evaluation complexity in heterogeneous environments. Jamshidi *et al.* [11] addressed challenges including resilience engineering, deployment complexity, and evaluation methodologies. Limitations include difficulty quantifying complex attributes and limited empirical studies complementing theoretical modeling.

3 Methodology

In order to systematically evaluate the impact of asynchronous communication mechanisms on the resilience and robustness of microservice architectures as compared to monolithic architectures, a controlled experimental framework was established. This section outlines the system prototypes developed, the experimental design, including baseline tests and fault injection methodology, as well as the metrics employed for evaluation. To enable replication of our study, we also provide the source code and the raw experimental data[1].

[1] https://doi.org/10.5281/zenodo.15740374

3.1 System Prototypes

Two software systems were developed as experimental artifacts: a monolithic system and a microservice-based system utilizing asynchronous communication. Both systems were designed to simulate the order processing workflow of a simplified e-commerce platform, including order creation, simulated shipment processing, and customer notification. Importantly, the core business logic was consistently maintained across both systems, and a uniform technology stack was employed where feasible to ensure a comparable baseline. A figure illustrating the high-level architecture of both applications can be found in the supplementary material to aid understanding.

Monolithic Architecture. The monolithic system was implemented as a single deployable Java (v23) application using the Spring Boot (v3.4.1) framework. All functionalities - order validation/creation, shipment initiation, and notification generation - were executed sequentially within a single process. Internal coordination between functional modules relied on synchronous, in-process method calls. The system utilized a PostgreSQL (v17.2) database. Deployment was managed as a single unit within a Kubernetes (v1.32.1) cluster using Docker (v27.4.0) containerization.

Microservice Architecture. The microservice system decomposed the application into three independent services: an *Ordering Service* (handling incoming order requests and validation), a *Shipping Service* (simulating shipment processing), and a *Notification Service* (handling user notifications). Each service was developed independently using Java (v23) and Spring Boot (v3.4.1), each connected to its own PostgreSQL (v17.2) database, and packaged as separate Docker containers. Orchestration was managed via Kubernetes (v1.32.1) using Minikube (v1.35.0) locally. Inter-service communication was implemented asynchronously through an event-driven architecture using Apache Kafka (v3.9.0), deployed via Strimzi (v0.45.0). Specific Kafka topics (*orderCreatedTopic*, *shipmentCreatedTopic*) facilitated the event flow between services.

3.2 Experiment Design

The experimental phase included three experiments. The first established baseline performance under load without failures, while the next two used Chaos Engineering [1, 22]. This involves controlled experimentation by injecting failures to build confidence in a system's ability to withstand turbulence. In these experiments, faults were introduced into systems under controlled load to evaluate behavior and resilience.

– **Experiment ❶ (Baseline Load Tests):** This experiment aimed to establish the baseline performance and behavior of both systems under high load conditions without induced faults. Workload was generated using k6 (v0.54.0) in *constant-vus* mode, maintaining 20 virtual users (VUs) for a duration of two minutes. This level of load was determined through preliminary testing to stress system resources near their limits without causing instability. Results from this experiment served as a reference point for evaluating the impact of failures in subsequent experiments.

– **Experiment ❷ and ❸ (Fault Injection Scenarios):** These experiments utilized Chaos Mesh (v2.7.0) to inject faults while the systems were under load generated by Grafana k6 (v0.54.0). The primary fault scenario involved simulating service crashes by terminating Kubernetes pods. For experiment ❷, the pod running the monolith application or the *Ordering Service* (the edge service in the microservice architecture) was terminated periodically. For experiment ❸, a pod corresponding to one of the three microservices was terminated randomly during the test runs.

A consistent workload was applied using k6 configured for a *constant-arrival-rate* of 10 requests per second over a duration of two minutes, utilizing up to 40 VUs to maintain the target rate under potential performance degradation. Each experimental run was preceded by checks to ensure service availability and followed by system restarts and database clearing to ensure consistent starting conditions for subsequent runs.

3.3 Metrics and Data Collection

Table 1 presents the systematically collected metrics during and subsequent to the experiments, aimed at evaluating the performance, resilience, and robustness of the two architectures. Data was obtained from k6 output logs and direct database queries following each test execution.

Table 1. Metrics Collected for the Evaluation.

Category	**Metric Name**	**Description**
Business Metrics	Orders Processed	Successfully processed orders.
	Shipments Processed	Simulated shipments created.
	Notifications	Generated notifications.
System Metrics	HTTP Requests	HTTP requests processed.
	HTTP Request Duration	Request duration statistics (ms).
	Failed HTTP Requests	HTTP request failure rate (%).

Results from multiple iterations were aggregated, and descriptive statistics, including mean, minimum, and maximum values, were computed using Python scripts. For experiment ❸, log data from the *Notification Service* was additionally utilized to distinguish between *Order* and *Shipment* notifications.

4 Results

This section delineates the empirical findings obtained from the comparative experiments conducted on the monolithic and microservice system prototypes. The results are systematically organized in accordance with the three research questions that guide this study. The data presented are derived from multiple iterations of each experimental scenario, including baseline load testing, targeted service failure, and random service failure.

4.1 RQ1: Impact on Resilience

In RQ1 we evaluated the systems' capacity to process requests successfully despite the presence of induced faults. The primary metrics considered were the number of successfully processed business transactions (*Orders*, *Shipments*, *Notifications*) and the HTTP request failure rate. Experiment ❷ involved simulating the failure of the primary entry point, which could be either the monolith itself or the *Ordering Service* within the microservice architecture. Table 2 presents a comparison of the business metrics and system-level request handling under this specific failure scenario.

Table 2. Targeted Failure of Entry Point (Mean Values).

Metric	**Monolith**	**Microservices**
Orders Processed	956	984
HTTP Requests	1,200	1,200
Failed HTTP Requests (%)	20.35	18.04

As illustrated, when the primary request-handling component failed, the asynchronous microservice architecture processed a slightly higher number of successful orders (approximately 3% more) and demonstrated a lower HTTP request failure rate (approximately 2.3 percentage points lower) compared to the monolithic architecture. This indicates a marginal resilience advantage even when the main entry point is compromised, potentially attributable to faster recovery or startup times of the smaller service instance, although this difference was minor in our setup. Experiment ❸ introduced random failures among any of the three microservices, while comparing against the monolith failure data from experiment ❷, as a monolith failure inherently affects all functions. Table 3 presents the results under these random internal failures.

The results from experiment ❸ demonstrate a pronounced resilience advantage for the asynchronous microservice architecture when internal components fail randomly. The microservice architecture processed significantly more successful orders (approx. 15.7% more) compared to the baseline monolith failure scenario. More notably, the average HTTP request failure rate dropped drastically from 20.35% in the monolith failure case to just 7.88% in the random

Table 3. Random Internal Service Failure (Mean Values).

Metric	**Monolith**	**Microservices**
Orders Processed	956	1,106
HTTP Requests	1,200	1,200
Failed HTTP Requests (%)	20.35	7.88

failure case of the microservice architecture. This highlights the fault isolation capability provided by the decoupled microservices and asynchronous communication; failures in downstream services (*Shipping*, *Notification*) did not prevent the *Ordering service* from accepting and acknowledging new requests, which were queued in Kafka for later processing.

4.2 RQ2: Impact on Robustness under Fault Conditions

The evaluation of robustness under fault conditions was primarily conducted by analyzing HTTP request duration and error rates. This involved comparing the system's behavior during experiments ❷ and ❸ with the baseline performance established in experiment ❶, which was a load test conducted without faults. Table 4 provides a summary of the baseline performance under load conditions without the introduction of failures.

Table 4. Baseline Performance Under Load.

Metric	**Monolith**	**Microservices**
Processed Orders (in 2 min)	21,421	61,124
Requests per Second (reqs/s)	178.41	509.19
HTTP Req. Duration (ms) - Mean	112.11	39.18
HTTP Req. Duration (ms) - Min	3.20	2.13
HTTP Req. Duration (ms) - Max	1,622.00	1,170.00
Failed HTTP Requests (%)	0.00	0.00

Under the specified baseline conditions, the asynchronous microservice architecture exhibited a markedly higher throughput, approximately 185% more requests per second, and a significantly reduced average request duration, approximately 65% faster than the monolithic architecture. This performance enhancement is likely attributable to the asynchronous nature of the system, which enables the edge service (*Ordering Service*) to respond promptly without necessitating the completion of the entire downstream process (*Shipping*, *Notification*) in a synchronous manner, as is required in the monolithic architecture. Table 5 provides the request duration metrics observed during the targeted failure scenario of experiment ❷.

Table 5. Request Duration with Entry Point Failure.

Metric	**Monolith**	**Microservices**
HTTP Req. Duration (ms) - Mean	59.57	33.58
HTTP Req. Duration (ms) - Min	7.42	5.48
HTTP Req. Duration (ms) - Max	2,680.00	1,925.00

As demonstrated in Table 5, during the targeted service failure (experiment ❷), the microservice architecture exhibited significantly improved response times, with an average duration approximately 43% shorter than that of the monolithic architecture. Additionally, the maximum latency was considerably reduced in the microservice architecture. This suggests enhanced robustness in terms of performance consistency, even when the primary service endpoint encounters instability.

Table 6 presents the request duration metrics during the random internal failure scenario of experiment ❸ (microservices only), compared to the monolith's performance during its failure scenario (experiment ❷).

Table 6. Request Duration with Random Internal Failure.

Metric	**Monolith**	**Microservices**
HTTP Req. Duration (ms) - Mean	59.57	24.87
HTTP Req. Duration (ms) - Min	7.42	4.99
HTTP Req. Duration (ms) - Max	2,680.00	1,333.42

When subjected to random internal failures, the asynchronous microservice architecture demonstrated superior robustness in performance relative to the monolithic architecture under similar conditions. Specifically, the average request duration was reduced by approximately 58%, and the maximum latency experienced a significant decrease. This observation underscores that decoupling through asynchronous communication effectively mitigates the impact of downstream issues on the responsiveness of the user-facing service endpoint.

4.3 RQ3: Operational Challenges and Differences

While quantitative results show advantages for the asynchronous microservices architecture in resilience and robustness under failure, the development process revealed several operational challenges.

- **Increased Complexity:** The microservice implementation required more components (services, databases, Kafka cluster, service discovery) than the monolith. Asynchronous communication setup demanded specific expertise.

- **Resource Utilization:** Baseline load testing required careful resource allocation (1 vCPU, 512 MiB RAM per pod) for both architectures. While the microservice architecture handled higher throughput, its total resource footprint (3 services + Kafka) exceeded the monolith, suggesting higher operational costs.
- **Monitoring and Debugging:** Tracing requests across distributed services and Kafka proved more challenging than monolith debugging. Message processing issues required analysis across multiple logs, necessitating distributed tracing solutions.
- **Consistency Management:** Asynchronous communication requires careful design for eventual consistency across distributed databases, adding complexity compared to monolithic consistency.

These findings indicate that while asynchronous microservices provide resilience benefits, they require greater operational maturity and infrastructure investment than monoliths.

5 Discussion

This section discusses the interpretation of these findings in relation to the research questions and existing literature, highlighting both the advantages and the inherent trade-offs.

5.1 Resilience and Robustness Findings (RQ1 & RQ2)

The findings indicate a positive influence of the asynchronous, event-driven microservice architecture on system resilience (RQ1). The significantly reduced failure rate observed during random internal service failures (experiment ❸, Table 3) compared to the monolithic scenario (experiment ❷, Table 2) highlights the advantage of fault isolation [15]. Within the microservice architecture, the failure of a downstream service (e.g., *Shipping Service* or *Notification Service*) did not impede the acceptance of new orders by the user-facing *Ordering Service*. Asynchronous communication via Apache Kafka facilitated the buffering of events (*orderCreated*), allowing downstream services to process them upon recovery. This stands in stark contrast to the monolithic architecture, where any internal processing failure within the synchronous request chain resulted in the failure of the entire transaction. Even in the event of an entry-point failure (experiment ❷), the microservice architecture demonstrated a slight resilience advantage, potentially due to the faster restart/recovery time of the smaller, independent service instance compared to the larger monolith, although this effect was less pronounced in our specific setup.

Regarding robustness (RQ2), the asynchronous microservice architecture consistently outperformed the monolith under fault conditions in terms of request duration (Tables 5 and 6). The decoupling achieved through asynchronous messaging prevented latency in downstream services from directly affecting the response time of the initial order request. This is consistent with findings suggesting performance benefits of event-driven architectures, although some studies report increased baseline latency due to network overhead [4, 5, 18]. Our baseline tests (experiment ❶, Table 4) actually demonstrated lower average latency for the microservice architecture, likely because the initial HTTP response could be sent much earlier without waiting for the full synchronous chain required by the monolith. The ability of the microservice architecture to maintain superior performance and lower failure rates under stress confirms its enhanced robustness compared to the monolithic counterpart in our experiments.

5.2 Operational Challenges and Trade-offs (RQ3)

Despite the demonstrated enhancements in resilience and robustness, the implementation of an asynchronous microservice architecture introduced substantial operational challenges (RQ3), which is consistent with findings in literature.

First, the inherent complexity of a distributed system increases significantly. The management of multiple services, databases, and the Kafka messaging infrastructure necessitated more sophisticated deployment orchestration (Kubernetes) and a deeper level of specialized knowledge compared to a single-unit monolith. This complexity extends to development, debugging, and maintenance [6]. Diagnosing issues requires correlating events across multiple service logs and potentially Kafka itself, underscoring the critical need for robust distributed tracing and observability tools in production environments, which represents an additional operational overhead.

Second, resource utilization and potential operational costs were elevated. While individual microservices may be small, the aggregate resource consumption (CPU, memory) of multiple service instances, along with the messaging broker infrastructure, typically surpasses that of an equivalent monolith, as indicated by comparisons in other studies [2] and implied by the need for adequate resource allocation in our setup.

Third, maintaining data consistency across distributed services communicating asynchronously necessitates careful design, often involving complex patterns, such as Sagas [20], whereas the monolith benefits from simpler transactional consistency within a single database. Although our simple application does not require complex consistency patterns, it remains a significant challenge in real-world scenarios.

5.3 Relation to Existing Work and Implications

Our findings empirically validate the theoretically proposed resilience advantages of asynchronous microservices. While prior studies, such as [4, 10], primarily concentrated on scalability and performance under normal load conditions, our research specifically quantifies the resilience and robustness enhancements under simulated failure scenarios using Chaos Engineering. The results underscore the significance of asynchronous, event-driven patterns [5, 18] in constructing fault-tolerant distributed systems.

However, the operational challenges observed reflect the "pains" frequently reported alongside the "gains" of microservices [11, 23]. Consequently, the decision to adopt asynchronous microservices is complex, involving significant trade-offs between resilience and robustness improvements and increased operational complexity and cost. Organizations must possess the requisite operational maturity, tools, and expertise to manage these complexities effectively. For systems where absolute resilience is critical and operational capacity is available, the benefits likely justify the costs. Conversely, for simpler applications or organizations with limited operational resources, a well-structured monolith, potentially a modular monolith [17], might remain a more pragmatic choice [6, 14].

6 Threats to Validity

We recognize the following potential threats to the validity of this study.

6.1 Construct Validity

Construct validity concerns whether the operationalizations and metrics accurately represent the theoretical constructs under study, specifically resilience and robustness. Standard metrics like HTTP failure rate, request duration, and processed transactions were used based on literature, but these may not fully capture all aspects of these complex quality attributes. The user-perceived system behavior during degradation and data consistency under failure were not directly measured. Additionally, the implemented system prototypes represent a simplified e-commerce workflow. This simplification, while necessary for controlled experimentation, may limit how well the observed behaviors represent resilience and robustness in complex, real-world enterprise systems.

6.2 Internal Validity

Internal validity addresses whether outcomes can be confidently attributed to experimental manipulations rather than confounding factors. Internal validity may be threatened by differing resource allocations, as the microservice architecture used three times the CPU/RAM limits of the monolith, potentially confounding performance comparisons. Additionally, inherent differences in failure semantics between the monolithic transactional model and event-driven buffering in the asynchronous microservice architecture could influence outcome metrics. Mitigation included using identical core logic and consistent deployment environments.

6.3 External Validity

External validity concerns the generalizability of findings beyond this study's context. Generalizability may be limited by the simplified e-commerce system prototype, which might not fully represent the complexities of real-world enterprise applications. The experiments were conducted in a controlled local Kubernetes environment, and results might vary in production conditions. The comparison was restricted to a classic synchronous monolith, limiting applicability to other monolithic patterns such as modular monoliths. The experimental apps were custom-built prototypes, which restricts the findings' broad applicability.

7 Conclusion and Future Work

This study empirically examines the impact of asynchronous communication mechanisms on the resilience and robustness of microservice architectures in comparison to traditional monolithic systems. By employing equivalent system prototypes subjected to controlled fault injection through Chaos Engineering, we systematically assessed key quality attributes under failure conditions.

The results indicate that the asynchronous, event-driven microservice architecture demonstrates significantly enhanced resilience and robustness relative to the synchronous monolithic architecture. Notably, the microservice architecture exhibited lower failure rates, particularly in the context of random internal component failures, and maintained superior performance (request duration) during fault scenarios. This confirms that the decoupling and fault isolation provided by asynchronous communication effectively mitigate the impact of failures and prevent cascading issues common in tightly coupled systems.

However, the study also identified significant operational challenges and trade-offs associated with the asynchronous microservice architecture. Increased architectural complexity, higher aggregate resource consumption (approximately 4x the resources of the monolith when including messaging infrastructure), and the necessity for sophisticated monitoring and debugging capabilities represent critical considerations. Therefore, while asynchronous communication demonstrably enhances fault tolerance, the decision to adopt this pattern must balance these benefits against the required operational maturity and potential costs. The answer to the central research question is thus nuanced: asynchronous communication significantly enhances microservice resilience and robustness compared to synchronous monoliths, but introduces substantial operational complexities that must be managed.

7.1 Future Work

Future work should validate these findings using more complex, realistic system prototypes and explore comparisons with alternative architectures, such as asynchronous modular monoliths. Further research could also investigate the nuanced impact of resource allocation strategies on resilience trade-offs and explore lightweight observability frameworks to address the monitoring challenges inherent in distributed, event-driven systems. Finally, expanding the Chaos Engineering experiments to include a broader range of fault types would provide a more comprehensive resilience assessment.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., Rosenthal, C.: Chaos Engineering. IEEE Software **33**(3), 35–41 (2016). https://doi.org/10.1109/MS.2016.60

2. Berry, V., Castelltort, A., Lange, B., Teriihoania, J., Tibermacine, C., Trubiani, C.: Is it Worth Migrating a Monolith to Microservices? An Experience Report on Performance, Availability and Energy Usage. In: 2024 IEEE International Conference on Web Services (ICWS), pp. 944–954 (2024). https://doi.org/10.1109/ICWS62655.2024.00112
3. Bjørndal, N., Jonatã Pires de Araújo, L., Bucchiarone, A., Dragoni, N., Mazzara, M., Dustdar, S.: Benchmarks and performance metrics for assessing the migration to microservice-based architectures. Journal of Object Technology **20**(2), 2:1–17 (2021). https://doi.org/10.5381/jot.2021.20.2.a3. http://www.jot.fm/contents/issue_2021_02/article3.html
4. Blinowski, G., Ojdowska, A., Przybyłek, A.: Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. IEEE Access **10**, 20357–20374 (2022). https://doi.org/10.1109/ACCESS.2022.3152803
5. Cabane, H., Farias, K.: On the impact of event-driven architecture on performance: An exploratory study. Future Generation Computer Systems **153**, 52–69 (2024). https://doi.org/10.1016/j.future.2023.10.021
6. Christian, J., Steven, Kurniawan, A., Anggreainy, M.S.: Analyzing Microservices and Monolithic Systems: Key Factors in Architecture, Development, and Operations. In: 2023 6th International Conference of Computer and Informatics Engineering (IC2IE), pp. 64–69 (2023). https://doi.org/10.1109/IC2IE60547.2023.10331155
7. Di Francesco, P., Lago, P., Malavolta, I.: Architecting with microservices: A systematic mapping study. Journal of Systems and Software **150**, 77–97 (2019). https://doi.org/10.1016/j.jss.2019.01.001
8. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications. Springer, Vienna (2014)
9. Fowler, M.: Microservices: A definition of this new architectural term, (2014). https://martinfowler.com/articles/microservices.html. Last access 2025-04-08.
10. Horváth, M., Sakhnenko, V., Gurbáľ, F.: Comparison of scalability and performance in Microservices and Monolithic Architectures. In: 2024 IEEE 17th International Scientific Conference on Informatics (Informatics), pp. 82–87 (2024). https://doi.org/10.1109/Informatics62280.2024.10900892
11. Jamshidi, P., Pahl, C., Mendonça, N.C., Lewis, J., Tilkov, S.: Microservices: The Journey So Far and Challenges Ahead. IEEE Software **35**(3), 24–35 (2018). https://doi.org/10.1109/MS.2018.2141039
12. Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., Babar, M.A.: Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. Information and Software Technology **131**, 106449 (2021). https://doi.org/10.1016/j.infsof.2020.106449
13. Meijer, W., Trubiani, C., Aleti, A.: Experimental evaluation of architectural software performance design patterns in microservices. Journal of Systems and Software **218**, 112183 (2024). https://doi.org/10.1016/j.jss.2024.112183
14. Mendonça, N.C., Box, C., Manolache, C., Ryan, L.: The Monolith Strikes Back: Why Istio Migrated From Microservices to a Monolithic Architecture. IEEE Software **38**(5), 17–22 (2021). https://doi.org/10.1109/MS.2021.3080335
15. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Sebastopol, CA (2015)

16. Pinciroli, R., Aleti, A., Trubiani, C.: Performance Modeling and Analysis of Design Patterns for Microservice Systems. In: 2023 IEEE 20th International Conference on Software Architecture (ICSA), pp. 35–46 (2023). https://doi.org/10.1109/ICSA56044.2023.00012
17. Prakash, C., Arora, S.: Systematic Analysis of Factors Influencing Modulith Architecture Adoption over Microservices. In: 2024 TRON Symposium (TRONSHOW), pp. 1–8 (2024)
18. Rahmatulloh, A., Nugraha, F., Gunawan, R., Darmawan, I.: Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems. In: 2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS), pp. 01–06 (2022). https://doi.org/10.1109/ICADEIS56544.2022.10037390
19. Razzaq, A., Ghayyur, S.A.K.: A systematic mapping study: The new age of software architecture from monolithic to microservice architecture—awareness and challenges. Computer Applications in Engineering Education **31**(2), 421–451 (2023). https://doi.org/10.1002/cae.22586
20. Richardson, C.: Microservices Patterns: With examples in Java. Manning, Shelter Island, New York (2018)
21. Rodrigues, H., Rito Silva, A., Avritzer, A.: Performance Comparison of Monolith and Microservice Architectures. In: Tekinerdoğan, B., Spalazzese, R., Sözer, H., Bonfanti, S., Weyns, D. (eds.) Software Architecture. ECSA 2023 Tracks, Workshops, and Doctoral Symposium, pp. 185–199. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-66326-0_12
22. Al-Said Ahmad, A., Al-Qora'n, L.F., Zayed, A.: Exploring the impact of chaos engineering with various user loads on cloud native applications: an exploratory empirical study. Computing **106**(7), 2389–2425 (2024). https://doi.org/10.1007/s00607-024-01292-z
23. Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J.: The pains and gains of microservices: A Systematic grey literature review. Journal of Systems and Software **146**, 215–232 (2018). https://doi.org/10.1016/j.jss.2018.09.082
24. Srijith, R, K.B., N, G., R, A.M.: Inter-Service Communication among Microservices using Kafka Connect. In: 2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS), pp. 43–47 (2022). https://doi.org/10.1109/ICSESS54813.2022.9930270
25. Thönes, J.: Microservices. IEEE Software **32**(1), 116–116 (2015). https://doi.org/10.1109/MS.2015.11
26. Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC), pp. 583–590 (2015). https://doi.org/10.1109/ColumbianCC.2015.7333476