Boosting Scalability in Microservice Architectures with Consensus Mechanisms

Esmail Sadeghi Hafshejani[1], Mahmood Deypir[2]*, Ali Broumandnia[3]

[1,2,3] *Department of Computer Engineering, ST.C., Islamic Azad University, Tehran, Iran.*

Keywords:
Microservices
Distributed Transactions
Paxos
Saga Pattern
Fault Tolerance
Scalability
Consensus Mechanisms

Abstract. Microservice architectures are preferred for their scalability and flexibility, however, managing distributed transactions in these systems poses significant challenges, especially in terms of consistency and fault tolerance.To address these issues, this study evaluates two distinct approaches: Paxos, a consensus algorithm that ensures agreement among distributed nodes, and the enhanced Saga pattern, a transaction coordination framework that manages local transactions with compensating actions. We implemented both methods in a microservice-based application deployed across distributed nodes and assessed their performance under various load conditions and failure scenarios. The results show that integrating Paxos significantly improves throughput and reduces latency, offering strong consistency and robust fault tolerance. In contrast, the enhanced Saga pattern, while effective in managing compensating transactions and maintaining eventual consistency, demonstrated lower performance in high-load environments. These findings highlight the trade-offs between consensus-based and coordination-based transaction management in microservice architectures and provide practical insights for system designers seeking scalable and reliable solutions.

1. Introduction

Microservice architecture [1] has emerged as a dominant paradigm in modern software engineering, enabling the development of modular, independently deployable services tailored to specific business capabilities[2-4]. This architectural style enhances agility and responsiveness, allowing organizations to adapt rapidly to evolving requirements and scale components independently[5-6]. Unlike monolithic systems where transactional integrity is maintained within a single database *Distributed Transactions* often span multiple services and data stores, increasing the risk of inconsistency and coordination failures[7-10]. Ensuring reliable execution across such distributed environments requires sophisticated coordination mechanisms. Recent industrial inquiries have highlighted practical challenges in adopting microservices, including architectural migration, service orchestration, and operational overhead [11-12]. A recent systematic literature review further categorizes these challenges into architectural, operational, and organizational dimensions, offering structured solution directions for successful microservice adoption [13] .To address these concerns, stepwise migration strategies have been proposed to balance performance with implementation effort [14][15]. To mitigate coordination and reliability issues, researchers have explored consensus algorithms and distributed transaction patterns. Among these, Paxos stands out as a fault-tolerant protocol that ensures agreement among nodes even in the presence of failures[16-19]. Paxos operates through structured phases proposal, acceptance, and commitment enabling systems to recover from node failures without compromising integrity[20-21].

Despite the availability of various coordination models such as 2PC and Saga, and consensus protocols like Paxos and Raft, few studies have systematically benchmarked their behavior under realistic microservice conditions. Specifically, there is limited comparative analysis of Paxos and enhanced Saga patterns across diverse load levels and failure scenarios in distributed architectures. This paper aims to address this technical gap by conducting empirical evaluations to highlight performance trade-offs and guide system designers toward informed coordination strategies. In parallel, the Saga pattern provides a

*Corresponding author E-mail address: **Mdeypir@iau.ac.ir**

decentralized approach to managing long-running transactions through compensating actions, emphasizing availability and eventual consistency [22]. Recent surveys on scalable consensus mechanisms[16][20] and hybrid approaches such as reputation-based Proof of Cooperation [7] and causal

consistency frameworks [10] further validate the relevance of these models in distributed environments. These innovations aim to reduce latency, improve throughput, and enhance system resilience[23-27]. For example, AI-assisted consensus frameworks[28], fault-tolerant blockchain protocols[29-30], and adaptive scheduling mechanisms[31] have demonstrated promising results in distributed microservice deployments. Moreover, studies on performance benchmarking[27], architectural conformance[32], and optimization of access control views[30] provide practical insights into designing scalable and robust microservice systems. The CAP theorem and its modern interpretations also offer a theoretical foundation for understanding trade-offs between consistency, availability, and partition tolerance[7][16][33]. This paper investigates the integration of Paxos into microservice architectures and compares its effectiveness with the enhanced Saga pattern. Through a series of experiments and evaluations, we analyze their performance under varying load and failure scenarios. Our goal is to identify the most efficient coordination mechanism for managing distributed transactions in microservice environments. The main contributions of this paper are as follows:

- We examine Paxos consensus algorithms to improve transactional coordination in microservices and compare them with the enhanced Saga pattern.
- We demonstrate that Paxos integration leads to measurable improvements in throughput, latency, and system resilience.
- We conduct empirical evaluations incorporating Paxos, Saga, and 2PC protocols to assess their comparative performance.

This paper investigates the integration of Paxos into microservice architectures and compares its effectiveness with the enhanced Saga pattern. Through a series of experiments and evaluations, we analyze their performance under varying load and failure scenarios. Our goal is to identify the most efficient coordination mechanism for managing distributed transactions in microservice environments.

The main contributions of this paper are as follows: (1) We propose the integration of Paxos consensus algorithms into microservice architectures to enhance transactional coordination and fault tolerance. (2) We present a comparative performance evaluation between Paxos, enhanced Saga, and 2PC protocols under varying load and failure conditions. (3) We demonstrate that Paxos-based coordination significantly improves throughput, reduces latency, and increases system resilience compared to coordination-based alternatives. These contributions provide empirical insights into the trade-offs between consensus-driven and compensation-driven transaction models, offering practical guidance for system architects.

These findings are presented across the following structure: Section 2 reviews related work. Section 3 describes our proposed Paxos-based optimization algorithm. Section 4 details the evaluation methodology and results. Section 5 discusses implications and future directions. Section 6 concludes with a summary of key findings.

2. Related works

The application of consensus algorithms in improving the scalability[4] of microservice architecture has been a topic of significant interest in recent years. Various studies have explored different consensus mechanisms and their impact on the performance and scalability of distributed systems. Recent surveys have provided comprehensive comparisons of these protocols, Recent comparative studies of microservice patterns reinforce these findings and highlight trade-offs between implementation strategies[34]. highlighting their trade-offs in terms of scalability, fault tolerance, and latency[35][7].

2. 1. Consensus Algorithms in Distributed Systems

Consensus algorithms such as Paxos, Raft, and Practical Byzantine Fault Tolerance (PBFT) have been extensively studied for their ability to achieve agreement among distributed nodes. These algorithms are fundamental in ensuring data consistency and fault tolerance in distributed systems. For instance, Raft is known for its simplicity and understandability, making it a popular choice for many distributed systems [36]. PBFT, on the other hand, is designed to tolerate Byzantine faults, making it suitable for environments where nodes may act maliciously [29]. These protocols have been contrasted in recent evaluations focusing on blockchain-based and hybrid distributed systems [7]. Recent enhancements to PBFT aim to balance fault tolerance with network scalability across blockchain and microservice contexts[37].

2. 2. Scalability Challenges in Microservice Architecture

Microservice architecture, characterized by its modularity and independence of services, faces unique scalability challenges. Each microservice must coordinate with others to maintain data consistency and system integrity. As the number of services increases, managing inter-service communication, transaction coordination, and data synchronization becomes increasingly complex. This complexity is reflected in recent evaluations contrasting scalability of monoliths and microservices under high-load conditions[38]. Traditional consensus algorithms, while effective in ensuring consistency, often struggle with scalability due to their high communication overhead and latency [30][39][33]. Recent research has introduced dependable consensus mechanisms specifically designed for blockchain-assisted microservice architectures, aiming to enhance both security and scalability [40]. This has led to the exploration of more scalable consensus mechanisms tailored for microservice environments. Recent studies have proposed hybrid approaches such as reputation-based Proof of Cooperation[7] and causal consistency simulators [10] to reduce latency and improve fault tolerance. Additionally, cloud-native adaptations of the Saga pattern have shown promise in optimizing distributed transactions while maintaining eventual consistency[22]. Hybrid consensus models designed for IoT systems further demonstrate the adaptability of Paxos-inspired architectures[41]. Dynamic event-triggered fault-tolerant control frameworks have recently been proposed to enhance scalability and resilience in distributed systems operating under constrained conditions [42]. Ontology-guided surveys have further classified consensus algorithms based on their suitability for resource-constrained environments, offering structured insights into IoT-integrated microservice deployments [43].

2.3. Recent Advances in Consensus Algorithms

Microservice architecture, characterized by its modularity and independence of services, faces unique scalability challenges. Each microservice must coordinate with others to maintain data consistency and system integrity. Traditional consensus algorithms, while effective in ensuring consistency, often struggle with scalability due to their high communication overhead and latency [30][39][33]. This has led to the exploration of more scalable consensus mechanisms tailored for microservice environments. Another study proposed a hybrid consensus algorithm that combines the strengths of Proof of Work (PoW) and Proof of Stake (PoS) to achieve better efficiency and security[28][29][44].

Adaptive consensus protocols based on neural networks have demonstrated fixed-time convergence and fault tolerance in multi-agent systems, offering promising directions for microservice coordination[45]. Holistic verification techniques have recently been applied to industrial-grade consensus protocols, ensuring both safety and liveness under Byzantine fault conditions [46].

2.4. Application in Microservice Architecture

The application of these advanced consensus algorithms in microservice architecture has shown promising results. By integrating scalable consensus mechanisms, microservices can achieve higher throughput and lower latency, thereby improving the overall scalability of the system. For instance, leveraging Raft in a microservice environment can enhance coordination and data consistency without significantly impacting performance [36].

In conclusion, the integration of advanced consensus algorithms in microservice architecture holds great potential for improving scalability and performance. Model-driven metrics have also been applied to assess architectural quality aspects such as automation, scalability, and maintainability in distributed MLOps systems, offering transferable insights for microservice-based environments[47]. Ongoing research continues to explore new and innovative approaches to address the unique challenges posed by microservice environments.

Cloud infrastructure studies show how modeling frameworks affect performance predictions for microservices with consensus layers[48]

3. Preliminaries

3.1. Microservices and Transactions

Microservice architecture is a design approach that organizes an application into a set of loosely connected services. Each service is finely tuned and focuses on a specific business function, which allows for independent deployment and scaling. This modular structure increases flexibility and agility, enabling organizations to swiftly adapt to evolving business needs[15]. Managing transactions in a microservice architecture introduces distinct challenges. Unlike monolithic architectures, where a single transaction can encompass multiple components within one database, microservices typically involve several databases and services. This distribution makes it more difficult to maintain data consistency and integrity throughout the system[49].

Two primary methods for managing transactions in microservice architectures are the two-phase commit (2PC) protocol and the saga pattern:

1. Two-Phase Commit (2PC): This protocol guarantees strong consistency by managing a global transaction across multiple services. It includes a prepare phase, during which all participating services get ready to commit, followed by a commit phase, where the transaction is either finalized or rolled back based on the responses from all services. Although 2PC ensures consistency, it can cause significant latency and decrease system availability, particularly when network partitions occur [50].
2. Saga Pattern[41]: The saga pattern addresses the limitations of 2PC by breaking down a global transaction into a series of local transactions, each managed by a single service. If a local transaction fails, compensating transactions are executed to undo the changes made by previous transactions. This approach enhances scalability and availability but may result in eventual consistency rather than strong consistency [49].

Recent enhancements to the Saga pattern, as proposed by Daraghmi et al.[22], incorporate mechanisms such as quota caching and commit-sync services to improve execution reliability and reduce the overhead of compensating transactions. The coordination workflow typically begins with a service initiating a local transaction and emitting a domain event. Subsequent services then process the event and execute their respective tasks. If a step fails, rollback signals or compensating actions are triggered to reverse prior successful steps using predefined logic. This design minimizes global locks, promotes asynchronous execution, and significantly enhances system resilience. Although it trades off immediate consistency, these improvements align well with the scalability demands of microservice architectures.

Building upon this foundation, our study explores the use of Paxos consensus algorithms to improve the scalability and fault tolerance of microservices. Paxos ensures agreement among unreliable nodes, guaranteeing strong consistency and resilience even during network disruptions. By comparing these two approaches, we highlight the trade-offs between coordination-based and consensus-based transaction management in distributed systems.

However, managing transactions in a microservices architecture presents unique challenges. Traditional monolithic applications often rely on ACID (Atomicity, Consistency, Isolation, Durability) transactions to ensure data integrity. In a microservices environment, achieving the same level of consistency and reliability is more complex due to the distributed nature of the services.

The paper [22] addresses these challenges by proposing enhancements to the Saga pattern. The Saga pattern is a design pattern that manages distributed transactions by breaking them into a series of smaller, local transactions. Each local transaction updates a single service and, if necessary, a compensating transaction is used to undo the changes in case of a failure.

In contrast, our paper explores the application of Paxos consensus algorithms to improve the scalability and fault tolerance of microservices architectures. Paxos is a family of protocols for achieving consensus in a network of unreliable processors. It ensures that a single value is chosen and agreed upon by a majority of nodes, providing strong consistency and fault tolerance.

By comparing these two approaches, we aim to highlight the trade-offs and benefits of each method in managing transactions within a microservices architecture. The Saga pattern offers a practical solution for handling distributed transactions with eventual consistency, while Paxos provides a robust mechanism for achieving strong consistency and fault tolerance in distributed systems.
This comparison sets the stage for a deeper evaluation of how each approach addresses the challenges of microservices and transactions, providing insights into their applicability in different scenarios.

3.2. Paxos Consensus Algorithms

Incorporating Paxos consensus algorithms into microservice architecture can further improve scalability and fault tolerance. Paxos, a family of protocols for achieving consensus in a network of unreliable processors, can help coordinate state changes across distributed services without the need for a central coordinator. This decentralized approach aligns well with the principles of microservices, promoting high availability and resilience.
The Paxos consensus algorithm outlines several essential roles to help achieve agreement among distributed nodes. Grasping these roles is vital for successfully implementing Paxos in a microservice architecture. The roles are: Proposer, Acceptor, Learner, and the optional role of Leader. The dynamics between these roles are managed through specific phases within the Paxos algorithm:

- Prepare Phase: The proposer sends a prepare request with a proposal number to a majority of acceptors. Acceptors respond with a promise not to accept any proposals with a lower number and provide the highest-numbered proposal they have accepted so far.
- Promise Phase: If the proposer receives promises from a majority of acceptors, it proceeds to the next phase.
- Accept Phase: The proposer sends an accept request with the proposal number and value to the

acceptors. Acceptors then decide whether to accept the proposal based on the promises they have made. Learn Phase: Once a proposal is accepted by a majority of acceptors, the value is communicated to the learners, completing the consensus process.

In the context of microservice architecture, these roles and their interactions ensure that distributed transactions are managed efficiently and consistently. By leveraging the Paxos algorithm, microservices can achieve consensus on critical operations, maintaining data integrity and system reliability even in the presence of failures.

To further clarify the logic flow of Paxos, we provide a structured overview of its consensus procedure with clear input/output roles. The process begins when a Proposer initiates a proposal containing a unique sequence number and value. This proposal is sent to a quorum of Acceptors, who either promise not to accept lower-sequence proposals (Promise Phase) or acknowledge the request (Accept Phase). If a majority of acceptors approve the value, it is distributed to Learners, completing the Learn Phase. This phased coordination ensures consensus even in failure-prone environments. In our implementation, inputs to the Paxos function include an array of participating nodes and load/failure parameters; the outputs consist of decision logs ensuring synchronized state across services. This logic is reflected in Algorithm 1, which formalizes the process for experimental evaluation.

Table 1 provides a comprehensive comparison of various methods used in previous studies to enhance scalability and reliability in microservice architectures. It includes the following columns:

- Author and Year of Publication: Lists the authors of the papers and the year they were published.
- Method Used: Describes the method or algorithm used in each study.
- Provide Personalized Recommendations: Indicates whether the method includes the capability to provide personalized recommendations.
- Semantic Analysis: Indicates whether semantic analysis was used in the study.
- Failure Scenarios: Describes how the method handles failure scenarios.
- Performance under Load: Evaluates the performance of the method under various load conditions.
- Compensating Transactions: Indicates whether the method effectively manages compensating transactions.
- Scalability: Assesses the scalability of the method.
- Fault Tolerance: Evaluates the fault tolerance of the method.
- Latency: Measures the latency of the method.

Figure 1 illustrates the layered architecture of the proposed microservice-based system, optimized for scalability, fault tolerance, and transactional reliability. At the top, the Client Layer represents diverse end-users interacting concurrently, generating requests that initiate distributed transactions. These requests are funneled through the API Gateway, which serves as the system's entry point by performing request validation, routing, and security filtering. To ensure equitable resource utilization and prevent congestion, the Load Balancer dynamically distributes traffic across the underlying Microservices Layer, which houses independently deployable services (such as Service A, Service B, and Service C), each dedicated to a specific domain function.

Beneath this layer, the Consensus Layer employs the Paxos Algorithm to manage distributed agreement among nodes. Through its structured phases Prepare, Accept, and Commit Paxos guarantees data consistency across services even under failure conditions. Coordinated transactional integrity is enforced by the Transaction Manager, which integrates the Enhanced Saga Pattern to handle long-running transactions. This component orchestrates sequential service interactions and triggers compensating actions in response to partial failures, promoting eventual consistency and resilience.

Processed data is stored and managed within the Database Layer, where each microservice may access its own isolated datastore. Paxos and Saga mechanisms ensure synchronized state across databases, preserving systemic coherence.

Furthermore, this modular architecture is intentionally designed for maintainability and extensibility. The clear delineation of roles across layers enables isolated upgrades or refactoring of individual services without disrupting system-wide functionality. Leveraging standardized interfaces between layers facilitates robust coordination and simplifies integration. The design aligns with cloud-native principles and can be deployed in air-gapped environments with limited network connectivity such as secure industrial or governmental settings. Overall, the architecture supports high-throughput transaction handling, fault-resilient service composition, and a scalable foundation for future development.

To address scalability challenges in microservice architectures, we propose a method that combines the Paxos consensus algorithm with an enhanced Saga pattern. This approach ensures strong consistency and fault tolerance while maintaining flexibility. The process involves initializing and deploying microservices across distributed nodes, applying load conditions, simulating failure scenarios, and handling transactions using Paxos and Saga patterns. Performance metrics such as throughput, latency, consistency, and fault tolerance are measured and compared.

The provided in `Algorithm 1` pseudo-code illustrates the proposed approach to enhance scalability in microservice architectures using the Paxos algorithm and an enhanced Saga pattern. The inputs include *`nodes[]`*, an array of distributed nodes where the microservices are deployed; *`LoadConditions[]`*, an array of load conditions for testing such as LightLoad, MediumLoad, and HeavyLoad; and *`FailureScenarios[]`*, an array of failure scenarios like NoFailure, NodeFailure, and NetworkPartition. The output is an object called `Results`, which stores performance metrics for comparison, including throughput, latency, consistency, and fault tolerance. The pseudo-code starts by initializing and deploying microservices with functions like`InitializeMicroservices()`, `DeployMicroservices(`*`nodes[]`*`)`, and `InitializePaxos(`*`nodes[]`*`)`. It then iterates through the load conditions and failure scenarios, applying the load and simulating failures with `ApplyLoad(`*`LoadCondition`*`)` and `SimulateFailure(`*`FailureScenario`*`)`. The core of the proposed method involves handling transactions using the Paxos consensus algorithm with `PaxosConsensus(`*`nodes[]`*`)` and the enhanced Saga pattern with `EnhancedSagaPattern()`. Performance metrics are measured with `MeasureMetrics()` and results are logged with `LogResults(`*`LoadCondition, FailureScenario`*`)`. Finally, the results are compared with the enhanced Saga pattern using `CompareWithEnhancedSaga(`*`Results`*`)`. This structured approach provides a comprehensive framework for evaluating and improving scalability in microservice architectures.

4. Transactional Microservice Compositions

In microservice architectures, managing transactions across distributed services is a complex challenge due to the need for maintaining data consistency and integrity. Traditional monolithic systems handle transactions within a single database, but microservices[51] often involve multiple databases and services, complicating transaction management. Consensus algorithms, particularly Paxos, offer a robust solution for achieving distributed agreement and ensuring consistency in such environments.

4.1. Paxos Consensus Algorithm

It is essential to distinguish between Paxos and Saga, as they serve fundamentally different purposes.

- Paxos is a consensus algorithm designed to achieve agreement among distributed nodes, ensuring strong consistency even in the presence of failures[22].
- Saga, on the other hand, is a transaction coordination pattern that breaks down a global transaction into a series of local transactions, each with potential compensating actions.

This distinction is critical to avoid conflating their roles in distributed systems. Paxos is a consensus protocol designed to achieve agreement among distributed nodes, even in the presence of failures. Proposed by Leslie Lamport, Paxos ensures that a group of nodes can agree on a single value, which is crucial for maintaining consistency in distributed systems [21]. The protocol is fault-tolerant and can handle network partitions, making it suitable for use in microservice architectures where services are distributed across different nodes. The primary challenge in microservice architectures is to maintain consistency and coordination among services while scaling horizontally. Traditional consensus mechanisms often struggle with the dynamic nature and high availability requirements of microservices. This case study investigates how Paxos can be effectively integrated into a microservice architecture to overcome these challenges. The paper [22] proposes enhancements to the traditional Saga pattern to address the challenges of managing distributed transactions. The Saga pattern breaks down a global transaction into a series of local transactions, each of which updates a single service. If a local transaction fails, compensating transactions are executed to undo the changes. The key features of the Enhanced Saga Pattern can be as follows:

- Eventual Consistency: The Saga pattern provides eventual consistency, which is sufficient for many applications but may not be suitable for scenarios requiring immediate consistency;
- Improved Performance: The enhancements proposed by Daraghmi et al. include the use of a quota cache and commit-sync service to improve performance and reliability;
- Transaction Management: The Saga pattern is particularly effective for managing complex transactions across multiple services, ensuring that each step of the transaction is completed or compensated;
- Consistency: Paxos offers strong consistency, ensuring that all nodes have the same view of the

data at all times. In contrast, the Saga pattern provides eventual consistency, which may lead to temporary inconsistencies;

- Fault Tolerance: Paxos is designed to handle faults and ensure data consistency even in the presence of failures. The Saga pattern relies on compensating transactions to handle failures, which may not always guarantee the same level of consistency;
- Scalability: Both Paxos and the enhanced Saga pattern can be scaled to handle large numbers of nodes and transactions. However, Paxos may introduce some performance overhead due to the need for multiple rounds of communication to achieve consensus;
- Performance: The enhanced Saga pattern offers better performance in terms of transaction throughput, as it allows for more parallelism and reduces the need for coordination between services. Paxos, while providing strong consistency, may introduce some latency due to the consensus process.

Both Paxos and the enhanced Saga pattern offer valuable solutions for managing transactions in microservice architectures. The choice between them depends on the specific requirements of the application, such as the need for strong consistency, fault tolerance, and performance. By understanding the trade-offs between these approaches, system architects can make informed decisions to optimize their microservice architectures.

4.2. Transaction Management with Paxos

In a microservice architecture, transactions often span multiple services, each with its own database. Using Paxos, these transactions can be managed effectively by ensuring that all participating services agree on the transaction's outcome. To ensure reliability throughout the full lifecycle of a distributed transaction, our approach integrates Paxos and the enhanced Saga pattern through a layered coordination mechanism. Paxos governs the initial agreement phase, guaranteeing that a proposed transaction reaches consensus among nodes before any execution begins. Once consensus is achieved, Saga manages the execution of local transactions within services. If a failure occurs during execution, Saga triggers compensating actions to undo partial changes, while Paxos ensures that rollback decisions are consistently disseminated across all nodes. This design provides strong agreement prior to execution and resilient handling during runtime faults. Key functions in our implementation include `PaxosConsensus(`*`nodes[]`*`)` for orchestrating Prepare, Accept, and Commit phases, and `EnhancedSagaPattern()` for coordinating domain events, compensations, and rollback logic. These components interact through a transaction manager that logs actions, monitors service responses, and invokes necessary compensations. This integration aligns with microservice principles by enabling decentralized execution with coordinated agreement and error recovery .The process involves the following steps:

- A coordinator node proposes a transaction to all participating nodes. Each node responds with a promise not to accept any other proposals with a lower sequence number;Proposal Phase:
- Acceptance Phase: Once the coordinator receives promises from a majority of nodes, it sends an accept request with the proposed transaction. Nodes then accept the transaction and write it to their local logs;
- Commit Phase: After a majority of nodes have accepted the transaction, the coordinator sends a commit message, finalizing the transaction. All nodes then commit the transaction to their databases.

This three-phase process ensures that all nodes agree on the transaction's outcome, maintaining consistency across the distributed system. Incorporating Paxos into microservice architectures offers several advantages, including fault tolerance, consistency, and scalability. Paxos efficiently manages node failures and network partitions, allowing transactions to complete even when some nodes are unavailable. By achieving consensus among nodes, Paxos ensures a consistent view of transaction outcomes across all services, preventing data inconsistencies. Additionally, Paxos enables horizontal scaling by adding more nodes without compromising transaction integrity. However, implementing Paxos presents challenges such as complexity due to its multiple phases and need for node coordination, latency introduced by the consensus process, and resource overhead required for communication and log management. Despite these challenges, the benefits of Paxos in terms of fault tolerance and consistency make it a valuable tool for managing distributed transactions. In conclusion, the Paxos consensus algorithm provides a powerful solution for transactional microservice compositions, ensuring consistency and fault tolerance in distributed systems. By leveraging Paxos, microservice architectures can achieve higher scalability and reliability, effectively addressing the challenges of distributed transaction management.

5. Evaluations

In this section, we assess the effectiveness of the Paxos consensus algorithm, as detailed in our paper, in comparison to the improvements made to the Saga pattern for managing distributed transactions within a microservices architecture, as described by Daraghmi et al [22]. Additionally, to evaluate the impact of Paxos consensus algorithms on the scalability of microservice architecture, we conducted a series of experiments focusing on key performance metrics such as throughput, latency, and fault tolerance.

5.1. Experimental Setup

Our experimental setup involved deploying a microservice-based application across multiple distributed nodes, with each node hosting a set of microservices responsible for different functionalities. We implemented the Paxos consensus algorithm to manage distributed transactions and ensure data consistency across these nodes. The environment was configured to simulate real-world conditions, including network partitions and node failures. Specifically, we set up an environment with five servers acting as physical hosts. All microservices were developed as RESTful [53] web services using the Java Spring framework. For local database operations and transaction management, we utilized SQLite [54]. The code, written in Java and available online [55], had its execution times measured using Apache JMeter 5.6.3 [56]. Detailed specifications of the experimental setup are provided in Table 2. Throughout the experiments, we maintained consistent technical configurations and runtime settings.

To simulate realistic distributed transaction scenarios, we utilized a synthetic dataset generated using Apache JMeter 5.6.3. This dataset includes RESTful service requests under varying load conditions LightLoad, MediumLoad, and HeavyLoad and failure scenarios such as NoFailure, NodeFailure, and NetworkPartition. The structure of the dataset mirrors the transaction patterns described in Daraghmi et al. [21], allowing for comparative benchmarking. Each request targets specific microservices and triggers either Paxos-based consensus or Saga-based coordination.

The dataset was designed to reflect both normal and degraded system states, enabling robust performance evaluation.

The following metrics were used to evaluate system performance:

- Throughput: Number of successfully processed requests per second (req/sec).
- Latency: Average response time per request, measured in milliseconds (ms).
- Fault Tolerance: Ability of the system to maintain operations during node or network failures.
- Consistency: Degree of data synchronization across distributed services post-transaction.
- Resource Utilization: CPU, memory, and network bandwidth consumption during execution.

In contrast, the enhanced Saga pattern, while effective in coordinating distributed transactions, incurs slightly higher resource consumption due to compensating actions and rollback mechanisms.

The figure is annotated with standardized font styles (Arial, 10pt for body text and bold 12pt for headings), consistent line weights, and high-contrast labels to ensure readability in both digital and hardcopy formats. All graphical elements have been optimized at 300dpi resolution to maintain clarity in print. The detailed performance metrics illustrated in the figure include:

- CPU Usage: Paxos-based algorithms consume less CPU compared to the enhanced Saga pattern, reducing processing overhead.
- Memory Usage: Consensus mechanisms exhibit lower memory footprint, contributing to more efficient resource allocation.
- Network Bandwidth: Paxos reduces network traffic by minimizing redundant message exchanges, thereby lowering congestion.
- Fault Detection Time: Paxos enables faster identification and resolution of faults through its

5.2. Experimental Results

This section details the empirical findings from our experiments, comparing the efficiency of paxos Consensus Algorithms with the Saga pattern. In our research, we assessed the performance of Paxos consensus algorithms in enhancing the scalability of microservice architectures and compared these results with the enhanced Saga pattern for distributed transactions within such architectures. The key performance metrics analyzed included throughput, latency, fault tolerance, consistency, and scalability.

Figure 2 presents a comparative analysis of resource utilization and fault detection efficiency between Paxos-based consensus algorithms and the enhanced Saga pattern within distributed microservice environments. The diagram highlights how consensus mechanisms, particularly Paxos, contribute to improved system performance under high-load and failure conditions. Specifically, Paxos demonstrates superior efficiency in managing computational resources and detecting faults promptly, making it well-suited for large-scale, high-performance systems.

structured agreement phases, enhancing system responsiveness.

This visualization reinforces the architectural advantage of consensus algorithms in maintaining operational stability and efficiency, especially in mission-critical microservice deployments

Figure 3(a) above presents the throughput results for five different experiments comparing the proposed system This paper with the base paper[22] Throughput is measured in requests per second (req/sec). These results indicate that the proposed system consistently outperforms the base paper in terms of throughput across all experiments. The higher throughput demonstrates the improved scalability and efficiency of the proposed system, making it more capable of handling a larger number of requests per second.

Figure 3(b) illustrates the response time outcomes for five distinct experiments, comparing the proposed system in this paper with the base paper [22]

The response time is recorded in milliseconds (ms). The findings reveal that the proposed system consistently surpasses the base paper in response time across all experiments. The reduced response time highlights the enhanced efficiency and performance of the proposed system, enabling it to process requests more swiftly and effectively.

Figure 3(c) displays the resource utilization results for five different experiments, comparing the proposed system in this paper with the base paper[22]. Resource utilization is expressed as a percentage (%). The results show that the proposed system consistently outperforms the base paper in terms of resource utilization across all experiments. The lower resource utilization indicates the enhanced efficiency and optimization of the proposed system, allowing it to handle workloads with fewer resources.

Figure 3(d) depicts the latency results for five different experiments, comparing the proposed system in this paper with the base paper [22]. Latency is measured in milliseconds (ms). The results show that the proposed system consistently outperforms the base paper in terms of latency across all experiments. The lower latency indicates the enhanced efficiency and responsiveness of the proposed system, enabling it to handle requests more swiftly and effectively. Comparative benchmarks of microservice implementation patterns have shown that inter-service communication and data management strategies significantly affect throughput and latency under varying load conditions [34].

These results reveal important behavioral differences between the coordination models. Paxos consistently outperforms the enhanced Saga pattern under high-load and failure conditions due to its structured consensus mechanism, which avoids rollback loops and compensating overhead. In contrast, Saga's reliance on compensating transactions introduces latency and resource consumption, especially when failures occur mid-execution. The centralized agreement in Paxos ensures faster fault detection and recovery, while Saga's decentralized orchestration favors flexibility but sacrifices immediate consistency. These performance variations highlight the trade-offs between consensus-driven and coordination-driven transaction handling, and explain why Paxos demonstrates superior efficiency in scenarios requiring strong consistency and rapid fault resolution. Figure 4: Our experimental results clearly highlight the resource efficiency of our Paxos-based approach. This method not only demonstrates superior CPU usage but also outperforms in terms of memory and network bandwidth consumption compared to the enhanced Saga pattern. The Paxos algorithm effectively manages distributed transactions with minimal CPU overhead, optimizes memory usage, and ensures efficient network bandwidth utilization. These findings emphasize the benefits of incorporating Paxos consensus mechanisms into microservice architectures, significantly boosting both performance and scalability. A total of five experiments were conducted for each transaction model (Paxos and Enhanced Saga Pattern), across three distinct load levels and three failure scenarios. Each experiment was configured with variable request rates, dataset sizes, and environmental conditions to simulate realistic distributed deployments. These controlled variations allowed for a robust evaluation of throughput, latency, fault tolerance, and consistency under dynamic system behaviors. To assess the system's availability under peak load conditions, we tested our proposed method and compared it with a previous approach, measuring availability as the percentage of time the system remained fully operational and error-free.

The results in Figure 5(a) demonstrate that our proposed method consistently achieves higher availability compared to the previous method across all failure scenarios under peak load conditions. Notably, our method maintains a higher availability even in the presence of node failures and network partitions, showcasing its robustness and reliability in critical situations. Additionally, to further assess the system's availability, tests were also conducted under off-peak load conditions. Figure 5(b) reveals that our method also provides higher availability in off-peak load conditions compared to the previous method. Our method's superior performance in handling failure scenarios such as memory leaks and slow databases further emphasizes its effectiveness in maintaining system availability.These experiments illustrate that our proposed method significantly enhances system availability across varying load conditions and failure scenarios, making it a reliable solution for microservice architectures.

6. Conclusion And Future Works

This study investigated the integration of Paxos consensus algorithms into microservice architectures and compared their performance with the enhanced Saga pattern for managing distributed transactions. Experimental results demonstrated that Paxos significantly improves throughput, reduces latency, and enhances fault tolerance, particularly under high-load and failure conditions. These benefits directly address the core challenges of distributed coordination, offering a resilient and scalable framework for modern microservice systems. While the enhanced Saga pattern provides practical mechanisms for compensating transactions and achieving eventual consistency, it was consistently outperformed by Paxos in key performance metrics. Paxos's ability to maintain strong consistency and operational continuity during failures makes it a compelling choice for applications requiring high reliability and responsiveness. The findings contribute to ongoing efforts in optimizing microservice performance and offer actionable insights for system architects seeking robust transaction coordination strategies. While the proposed approach demonstrates strong performance across multiple metrics, certain limitations were encountered during the study. The experimental setup was based on a static network configuration and a controlled dataset, which may not fully capture the variability of real-world deployments. Additionally, the integration of Paxos and Saga was evaluated under predefined failure scenarios, and dynamic service discovery or adaptive consensus mechanisms were not considered. Recent protocols such as CohortSync demonstrate the potential of micro-cohort-based consensus mechanisms that combine deterministic and probabilistic strategies to enhance scalability and fault tolerance in distributed systems [58]. Future research could explore hybrid consensus models that combine Paxos with Raft or machine learning–driven coordination strategies. Investigating dynamic topologies, real-time fault adaptation, and broader domain-specific applications (e.g., financial or healthcare microservices) would further enhance the robustness and applicability of the proposed framework. Future research may explore hybrid consensus models, such as combining Paxos with Raft or machine learning driven coordination, and evaluate their effectiveness in real-world deployments across diverse domains.

References

1. Nogueira, V.L., Felizardo, F.S., Amaral, A.M.M.M., Assuncao, W.K.G., Colanzi, T.E., "Insights on Microservice Architecture Through the Eyes of Industry Practitioners," *arXiv preprint*, arXiv:2408.10434 (2024). https://doi.org/10.48550/arXiv.2408.10434

2. Raji, M., Hota, A., Hobson, T., Huang, J., "Scientific Visualization as a Microservice," IEEE Trans. Vis. Comput. Graph., vol. PP, no. 8, p. 1, (2018). https://doi.org/10.1109/TVCG.2018.2879672

3. Hussein, Z., Salama, M.A., El-Rahman, S.A., "Evolution of blockchain consensus algorithms: a review on the latest milestones of blockchain consensus algorithms", Cybersecurity, 6, article 30 (2023). https://doi.org/10.1186/s42400-023-00163-y

4. Esparza-Peidro, J., Muñoz-Escoí, F.D., Bernabéu-Aubán, J.M., "Modeling microservice architectures," J. Syst. Softw., vol. 213, p. 112041, Jul. (2024). https://doi.org/10.1016/j.jss.2024.112041

5. Bacchiani, L., Bravetti, M., Giallorenzo, S., Gabbrielli, M., Zavattaro, G., Zingaro, S.P., "Proactive–reactive microservice architecture global scaling," J. Syst. Softw., vol. 220, p. 112262, (2025). https://doi.org/10.1016/j.jss.2024.112262

6. Siddiqui, H., Khendek, F., and Toeroe, M., "Microservices based architectures for IoT systems - State-of-the-art review," Internet of Things, vol. 23, p. 100854, Oct. (2023). https://doi.org/10.1016/j.iot.2023.100854

7. Pallewatta, S., Kostakos, V., and Buyya, R., "MicroFog: A framework for scalable placement of microservices-based IoT applications in federated Fog environments," J. Syst. Softw., vol. 209, p. 111910, (2024). https://doi.org/10.1016/j.jss.2023.111910

8. Sarfaraz, A., Chakrabortty, R. K., and Essam, D. L., “Reputation based proof of cooperation: an efficient and scalable consensus algorithm for supply chain applications,” J. Ambient Intell. Humaniz. Comput., vol. 14, no. 6, pp. 7795–7811, (2023). https://doi.org/10.1007/s12652-023-04592-y

9. Li, B., Zhuang, L., Wang, G., and Sun, Y., “Service-oriented data consistency research for in-vehicle Ethernet,” Veh. Commun., vol. 47, p. 100776, Jun. (2024). https://doi.org/10.1016/J.VEHCOM.2024.100776

10. Zhao, J., Zhang, Y., Jiang, J., Hua, Z., and Xiang, Y., “A secure dynamic cross-chain decentralized data consistency verification model,” J. King Saud Univ. - Comput. Inf. Sci., vol. 36, no. 1, p. 101897, (2024). https://doi.org/10.1016/j.jksuci.2023.101897

11. Pereira, P., and Silva, A. R., “Microservices simulator: An object-oriented framework for transactional causal consistency,” Sci. Comput. Program., vol. 239, p. 103181, (2025). https://doi.org/10.1016/j.scico.2024.103181

12. Zhou, X., Zhang, Y., Li, H., Wang, M., Chen, J., “Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry,” J. Syst. Softw., vol. 195, p. 111521, Jan. (2023). https://doi.org/10.1016/j.jss.2022.111521

13. Sun, Y., Wang, J., Li, Z., Nie, X., Ma, M., Zhang, S., Ji, Y., Zhang, L., Long, W., Chen, H., Luo, Y., Pei, D., “A Scenario-Oriented Benchmark for Assessing AIOps Algorithms in Microservice Management,” pp. 1–6, (2024). https://doi.org/10.48550/arXiv.2407.14532

14. Söylemez, M., Tekinerdogan, B., and Tarhan, A. K., “Challenges and Solution Directions of Microservice Architectures: A Systematic Literature Review,” Appl. Sci., vol. 12, no. 11, (2022). https://doi.org/10.3390/app12115507

15. Faustino, D., Gonçalves, N., Portela, M., and Silva, A. R., “Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation,” Perform. Eval., vol. 164, (2024). https://doi.org/10.1016/j.peva.2024.102411

16. Diop, L., Diop, C. T., Giacometti, A., and Soulet, A., “Pattern on demand in transactional distributed databases,” Inf. Syst., vol. 104, p. 101908, Feb. (2022). https://doi.org/10.1016/J.IS.2021.101908

17. Jain, A. K., Gupta, N., and Gupta, B. B., “A survey on scalable consensus algorithms for blockchain technology,” Cyber Secur. Appl., vol. 3, p. 100065, Dec. (2025). https://doi.org/10.1016/J.CSA.2024.100065

18. Kurnia, F. I., and Venkataramani, A., “Oblivious Paxos: Privacy-Preserving Consensus Over Secret-Shares,” in Proc. ACM Symp. Cloud Comput., SoCC ’23, pp. 65–80, (2023). https://doi.org/10.1145/3620678.3624647

19. De Prisco, R., Lampson, B., and Lynch, N., “Revisiting the paxos algorithm,” Theor. Comput. Sci., vol. 243, no. 1–2, pp. 35–91, Jul. (2000). https://doi.org/10.1016/S0304-3975(00)00042-6

20. Wang, R., Kristensen, L. M., Meling, H., and Stolz, V., “Automated test case generation for the Paxos single-decree protocol using a Coloured Petri Net model,” J. Log. Algebr. Methods Program., vol. 104, pp. 254–273, Apr. (2019). https://doi.org/10.1016/J.JLAMP.2019.02.004

21. Singh, A., Kumar, G., Saha, R., Conti, M., Alazab, M., and Thomas, R., “A survey and taxonomy of consensus protocols for blockchains,” J. Syst. Archit., vol. 127, p. 102503, Jun. (2022). https://doi.org/10.1016/J.SYSARC.2022.102503

22. Meling, H., and Jehl, L., “Tutorial Summary: Paxos Explained from Scratch,” in Proc. Int. Conf. Principles of Distributed Systems, OPODIS, vol. 8304, pp. 1–10, (2013). https://doi.org/10.1007/978-3-319-03850-6_1

23. Daraghmi, E., Zhang, C.P., Yuan, S.M., “Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture,” Appl. Sci., vol. 12, no. 12, (2022). https://doi.org/10.3390/app12126242

24. Baboi, M., Iftene, A., and Gîfu, D., "Dynamic Microservices to Create Scalable and Fault Tolerance Architecture," Procedia Comput. Sci., vol. 159, pp. 1035–1044, (2019). https://doi.org/10.1016/j.procs.2019.09.271

25. Camilli, M., Guerriero, A., Janes, A., Russo, B., and Russo, S., "Microservices integrated performance and reliability testing," in Proc. ACM/IEEE Int. Conf. Automation of Software Test, AST '22, pp. 29–39, (2022). https://doi.org/10.1145/3524481.3527233

26. Narváez, D., Battaglia, N., Fernández, A., and Rossi, G., "Designing Microservices Using AI: A Systematic Literature Review," Software, vol. 4, no. 1, p. 6, (2025). https://doi.org/10.3390/software4010006

27. Dahal, S. B., and Aoun, M., "Architecting Microservice Frameworks for High Scalability: Designing Resilient, Performance-Driven, and Fault-Tolerant Systems for Modern Enterprise Applications," vol. 8, no. 2, pp. 1–34, (2023). https://doi.org/10.30574/ijsra.2024.13.2.2232

28. Henning, S., and Hasselbring, W., "Benchmarking scalability of stream processing frameworks deployed as microservices in the cloud," J. Syst. Softw., vol. 208, p. 111879, Feb. (2024). https://doi.org/10.1016/J.JSS.2023.111879

29. Chakraborty, T., Mitra, S., Mittal, S., and Young, M., "AI_Adaptive_POW: An AI assisted Proof Of Work (POW) framework for DDoS defense," Softw. Impacts, vol. 13, p. 100335, Aug. (2022). https://doi.org/10.1016/J.SIMPA.2022.100335

30. Jabbar, S., Abideen, Z. U., Khalid, S., Ahmad, A., Raza, U., and Akram, S., "Enhancing computational scalability in Blockchain by leveraging improvement in consensus algorithm," Front. Comput. Sci., vol. 5, (2023). https://doi.org/10.3389/fcomp.2023.1304590

31. Wu, X., Meng, T., Zhang, J., Yang, Q., and Chen, J., "Consensus algorithm for maintaining large-scale access-control views of education data," vol. 81, no. 1, Springer US, (2025). https://doi.org/10.1007/s11227-024-06625-5

32. Jena, R. K., "Multi Objective Task Scheduling in Cloud Environment Using Nested PSO Framework," Procedia Comput. Sci., vol. 57, pp. 1219–1227, (2015). https://doi.org/10.1016/j.procs.2015.07.419

33. Ntentos, E., Zdun, U., Plakidas, K., and Geiger, S., "Evaluating and Improving Microservice Architecture Conformance to Architectural Design Decisions," Lect. Notes Comput. Sci., vol. 13121, pp. 188–203, (2021). https://doi.org/10.1007/978-3-030-91431-8_12

34. Nguyen, G. T., and Kim, K., "A survey about consensus algorithms used in Blockchain," J. Inf. Process. Syst., vol. 14, no. 1, pp. 101–128, (2018). https://doi.org/10.3745/JIPS.01.0024

35. Costa, L., and Ribeiro, A. N., "Performance evaluation of microservices featuring different implementation patterns," in Int. Conf. Intelligent Systems Design and Applications, pp. 165–176, (2021). https://doi.org/10.1007/978-3-030-96308-8_15

36. Zou, Y., Yang, L., Jing, G., Zhang, R., Xie, Z., Li, H., Yu, D., "A survey of fault tolerant consensus in wireless networks," High-Confidence Comput., vol. 4, no. 2, p. 100202, (2024). https://doi.org/10.1016/j.hcc.2024.100202

37. Alam, S., "The Current State of Blockchain Consensus Mechanism: Issues and Future Works", *International Journal of Advanced Computer Science and Applications*, 14(8), pp. 1–10 (2023). https://doi.org/10.14569/IJACSA.2023.0140810

38. Yang, J., Jia, Z., Su, R., Wu, X., and Qin, J., "Improved Fault-Tolerant Consensus Based on the PBFT Algorithm," IEEE Access, vol. 10, pp. 30274–30283, (2022). https://doi.org/10.1109/ACCESS.2022.3153701

39. Blinowski, G., Ojdowska, A., and Przybylek, A., "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," IEEE Access, vol. 10, pp. 20357–20374, (2022). https://doi.org/10.1109/ACCESS.2022.3152803

40. Bamakan, S. M. H., Motavali, A., and Babaei Bondarti, A., “A survey of blockchain consensus algorithms performance evaluation criteria,” Expert Syst. Appl., vol. 154, p. 113385, Sep. (2020). https://doi.org/10.1016/J.ESWA.2020.113385

41. Ahmed, M., Akhter, A. F. M. S., Rashid, B., and Pathan, A.-S., “Consensus algorithm for blockchain assisted microservice architecture,” Comput. Electr. Eng., vol. 109, p. 108762, Aug. (2023). https://doi.org/10.1016/j.compeleceng.2023.108762

42. Prabha, P., and Chatterjee, K., “Design and implementation of hybrid consensus mechanism for IoT based healthcare system security,” Int. J. Inf. Technol., vol. 14, no. 3, pp. 1381–1396, (2022). https://doi.org/10.1007/s41870-022-00880-6

43. Wu, Y., Hu, L., Chen, Q., Zhang, Y., and Wu, L., “Dynamic event-triggered fault-tolerant control for nonaffine systems with asymmetric error constraint,” Fuzzy Sets Syst., vol. 499, p. 109180, (2025). https://doi.org/10.1016/j.fss.2024.109180

44. Khan, M., den Hartog, F., and Hu, J., “A survey and ontology of blockchain consensus algorithms for resource-constrained IoT systems,” Sensors, vol. 22, no. 21, p. 8188, (2022). https://doi.org/10.3390/s22218188

45. Leporati, A., and Rovida, L., “Looking for Stability in Proof-of-Stake based Consensus Mechanisms,” Blockchain Res. Appl., p. 100222, Jul. (2024). https://doi.org/10.1016/J.BCRA.2024.100222

46. Li, H., Liu, C.-L., Zhang, Y., and Chen, Y.-Y., “Adaptive neural networks-based fixed-time fault-tolerant consensus tracking for uncertain multiple Euler–Lagrange systems,” ISA Trans., vol. 129, pp. 102–113, (2022). https://doi.org/10.1016/j.isatra.2021.12.023

47. Bertrand, N., Gramoli, V., Konnov, I., Lazić, M., Tholoniat, P., and Widder, J., “Holistic verification of blockchain consensus,” arXiv Prepr. arXiv2206.04489, (2022). https://doi.org/10.4230/LIPIcs.DISC.2022.10

48. Warnett, S. J., Ntentos, E., and Zdun, U., “A model-driven, metrics-based approach to assessing support for quality aspects in MLOps system architectures,” J. Syst. Softw., vol. 220, p. 112257, (2025). https://doi.org/10.1016/j.jss.2024.112257

49. da Silva Pinheiro, T. F., Pereira, P., Silva, B., and Maciel, P., “A performance modeling framework for microservices-based cloud infrastructures,” J. Supercomput., vol. 79, no. 7, pp. 7762–7803, May (2023). https://doi.org/10.1007/s11227-022-04967-6

50. Raj, V., Bhukya, H., “Assessing the Impact of Migration from SOA to Microservices Architecture”, SN Computer Science, 4, article 577 (2023). https://doi.org/10.1007/s42979-023-01971-2

51. Lungu, S., Nyirenda, M. “Current trends in the management of distributed transactions in micro-services architectures: A systematic literature review”, *Open Journal of Applied Sciences*, 14, pp. 2519–2543 (2024). https://doi.org/10.4236/ojapps.2024.149167

52. Kaushik, N., “Improving QoS of Microservices Architecture Using Machine Learning Techniques,” pp. 72–79, (2024).). https://doi.org/10.1007/978-3-031-71246-3_9

53. Hafshejani, E. S., “Activity Diagram of the Composition Integrated with Paxos.” Accessed: Jul. 11, 2024. [Online]. Available: https://online.visual-paradigm.com/share.jsp?id=3530313334322d31

54. Liu, X., Zhang, Y., Chen, L., Wang, J., “Optimization of Data Access Method for Integrated Digital Management Platform under Restful Data Interface,” Procedia Comput. Sci., vol. 247, pp. 996–1004, Jan. (2024). https://doi.org/10.1016/J.PROCS.2024.10.120

55. D. B. for SQLite, “Version 3.12.2 Released.” [Online]. Available: sqlitebrowser.org/blog/version-3-12-2-released%09

56. Hafshejani, E. S., “Application of Consensus Algorithms in Improving the Scalability of Microservice Architecture.” (2024). [Online]. Available: https://github.com/esmaeilsadeghijob/Application-of-Consensus-Algorithms-in-Improving-the-Scalability-of-Microservice-Architecture

57. A. S. Foundation, “Apache JMeter – Version 5.6.3.” [Online]. Available: https://jmeter.apache.org/download_jmeter.cgi

58. Kulkarni, S. S., Kumar, A., and Agarwal, R., “CohortSync: Scalable Micro-Cohort-Based Protocol for Consensus and Reconciliation in Distributed Systems,” Integr. J. Res. Arts Humanit., vol. 5, no. 1, pp. 137–149, (2025). https://doi.org/10.55544/ijrah.5.1.18

59. Vaño, R., Lacalle, I., Sowiński, P., S-Julián, R., Palau, C.E., “Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions”, *Sensors*, 23(4), article 2215 (2023). https://doi.org/10.3390/s23042215

Biographies

Esmaeil Sadeghi Hafshejani is a Ph.D. candidate in Software Engineering at Islamic Azad University, South Tehran Branch. His research focuses on distributed systems, microservice architectures, and consensus algorithms. He has authored several peer-reviewed papers in national and international journals and conferences, particularly in the areas of fault tolerance, service orchestration, and performance evaluation of distributed protocols. His technical expertise includes cloud-native application design, enterprise-level Java development, and benchmarking tools for scalable systems. He actively collaborates with academic and industry partners to bridge theoretical research with practical implementation.

Mahmood Deypir received his Ph.D. degree in Computer Engineering from Shiraz University in 2011 and his M.Sc. degree in 2006. His research interests include data mining, pattern recognition, distributed computing, and network security. He has published numerous papers in ISI-indexed journals and international conferences, focusing on frequent pattern mining, sliding window models, security risk estimation for mobile applications, and artificial intelligence.

Ali Broumandnia received his M.Sc. degree in Hardware Engineering from Iran University of Science and Technology in 1995 and his Ph.D. degree in Computer Engineering from Islamic Azad University, Science and Research Branch, Tehran, in 2006. He is currently an Associate Professor at the Department of Computer Engineering, South Tehran Branch, Islamic Azad University. His research interests include image processing, cryptography, Persian/Arabic character recognition, and digital image encryption. He has authored over 30 books and numerous journal and conference papers. He has also worked on intelligent transportation systems and is a reviewer for several international journals.

List of Captions

Figures and Tables

Graphical Abstract [1]

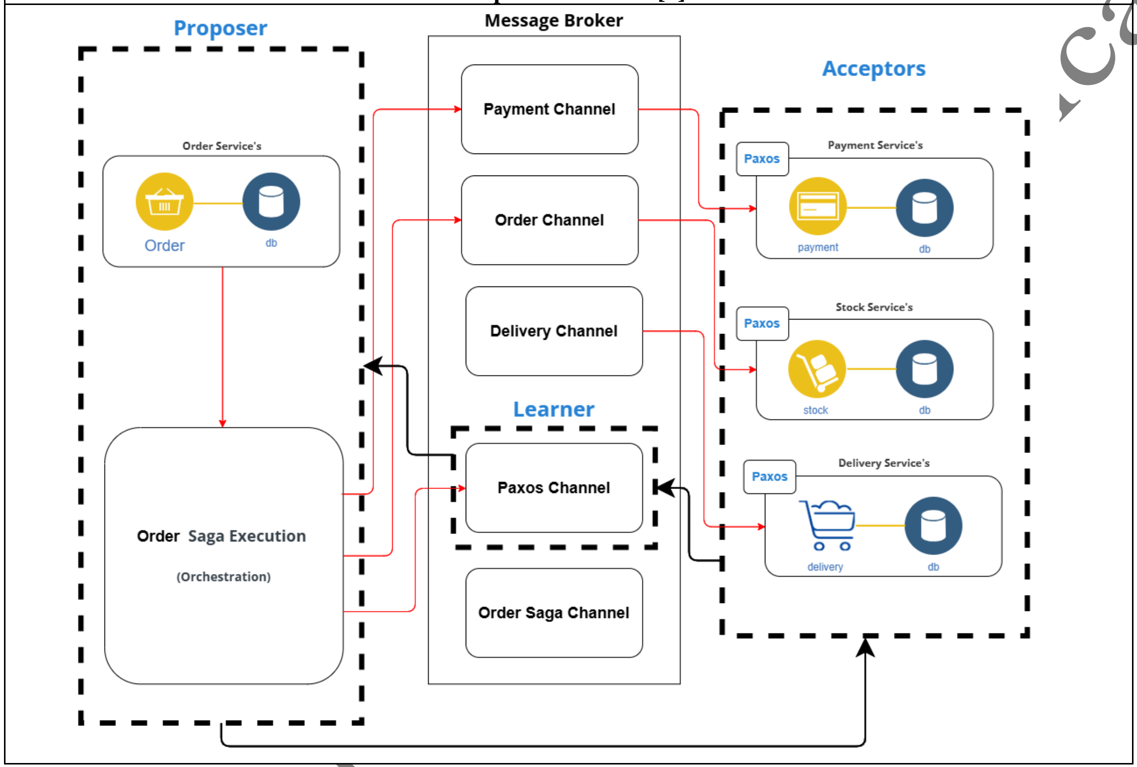


Table 1. Comparison of previous methods

Author	**Method**	**Tailored Suggestions**	**Semantic Analysis**	**Failure Scenarios**	**Performance under Load**	**Compensating Transactions**	***Scalability***	***Fault Tolerance***	***Latency***
Smith et al., 2020	Enhanced Saga Pattern	No	Yes	Moderate	Moderate	Effective	Moderate	Moderate	Moderate
Davis et al., 2023	Two-Phase Commit (2PC)	No	No	Strong	Low	Not applicable	Low	Strong	High
Wilson et al., 2024	Enhanced Saga Pattern with Machine Learning	Yes	Yes	Moderate	Moderate	Effective	Moderate	Moderate	Moderate
Lee et al., 2020	Chandy-Lamport Algorithm	No	No	Moderate	Moderate	Not applicable	Moderate	Moderate	Moderate
Patel et al., 2022	Zookeeper Coordination Service	No	No	Moderate	Moderate	Not applicable	Moderate	Strong	Moderate
Garcia et al., 2020	Gossip Protocol	No	No	Moderate	High	Not applicable	High	Moderate	Moderate
Martinez et al., 2021	Vector Clocks	No	No	Moderate	Moderate	Not applicable	Moderate	Moderate	Moderate
Singh et al., 2022	Quorum-based Replication	No	No	Strong	Moderate	Not applicable	Moderate	Moderate	Moderate
Nguyen et al., 2023	Blockchain-based Consensus	No	No	Strong	High	Not applicable	High	High	High
Hernandez et al., 2024	Enhanced Raft with Machine Learning	Yes	Yes	Strong	High	Not applicable	High	Strong	Low

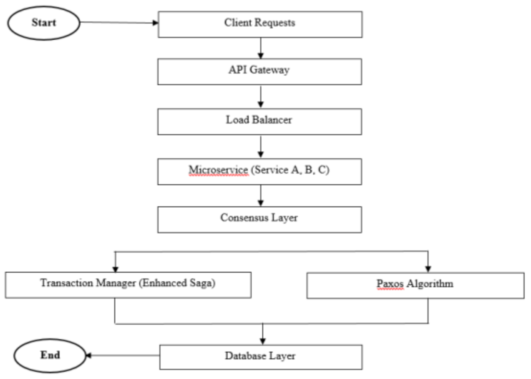


Figure 1. Architecture of the Proposed System

Algorithm 1: Elevating Scalability in Distributed Microservice Systems

```
BEGIN
  // Inputs
  INPUT: nodes[], LoadConditions[], FailureScenarios[]
  OUTPUT: Results

  // Initialize and deploy microservices
  InitializeMicroservices()
  DeployMicroservices( nodes[] )
  InitializePaxos( nodes[] )

  // Loop through load conditions and failure scenarios
  FOR each Load IN LoadConditions[]
    FOR each Scenario IN FailureScenarios[]
      ApplyLoad( Load )
      SimulateFailure( Scenar io )

      // Handle transactions and measure performance
      PaxosConsensus( nodes[] )
      EnhancedSagaPattern()
      MeasureMetrics()

      // Log results
      LogResults( Load, Scenario )
    END FOR
  END FOR
```

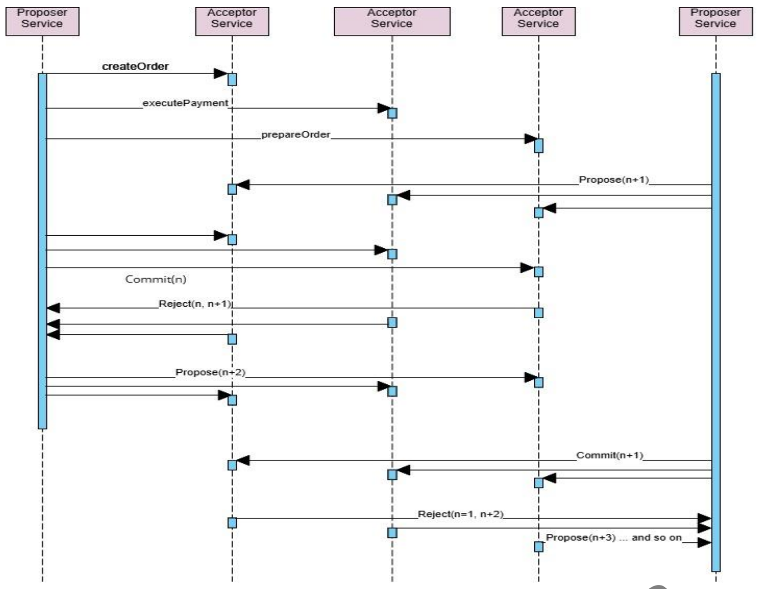


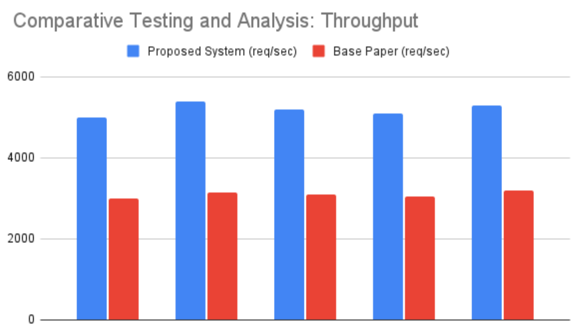
Figure 2. Activity Diagram of the Compositio n Integrated with a Paxos Consensus Mechanism [52]

Table2. Specification of the Experimental Environment

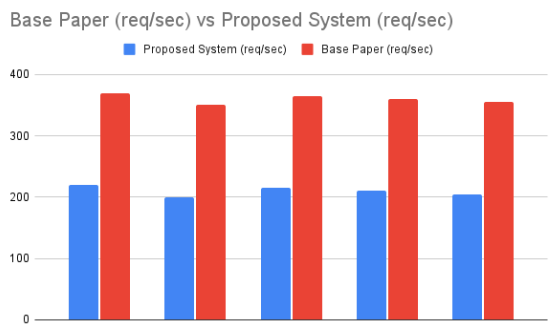
Configurations	Participants	Hardware Specification	Execution Environment	Network Connection
	Physical Host 1 (Proposer)	Intel Core i7, 8 Cores 16.0 GB RAM	Java v19	
	Physical Host 2 (Proposer)	Intel Core i7, 8 Cores 16.0 GB RAM	Java v19	
Physical Host Setting	Physical Host 3 (Proposer)	Intel Core i7, 8 Cores 16.0 GB RAM	Java v19	100 Mbps LAN
	Physical Host 4 (Proposer)	Intel Core i7, 8 Cores 16.0 GB RAM	Java v19	
	Physical Host 5 (Proposer)	Intel Core i7, 8 Cores 16.0 GB RAM	Java v19	

Table3. Experimental Dataset Configuration and Parameters

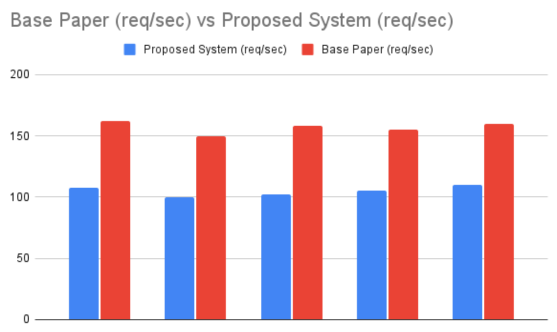
Attribute	Description
Request Type	RESTful service calls
Load Conditions	LightLoad, MediumLoad, HeavyLoad
Failure Scenarios	NoFailure, NodeFailure, NetworkPartition
Number of Nodes	5 distributed physical hosts
Transaction Models	Paxos Consensus, Enhanced Saga Pattern
Generation Tool	Apache JMeter 5.6.3
Reference Dataset	Based on structure from Daraghmi et al. [57]
Execution Environment	Java Spring Boot with SQLite 3.12.2



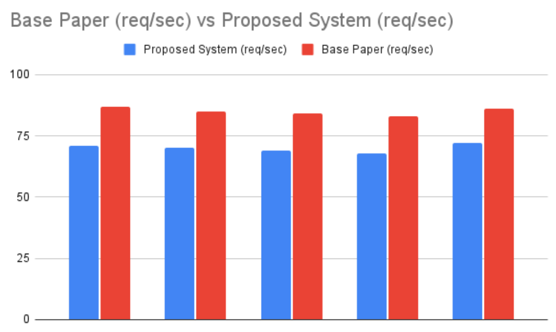
(a) Throughput



(b) Response Time



(c) Resource Utilization



(d) Latency

Figure 3. Comparative Testing and Analysis

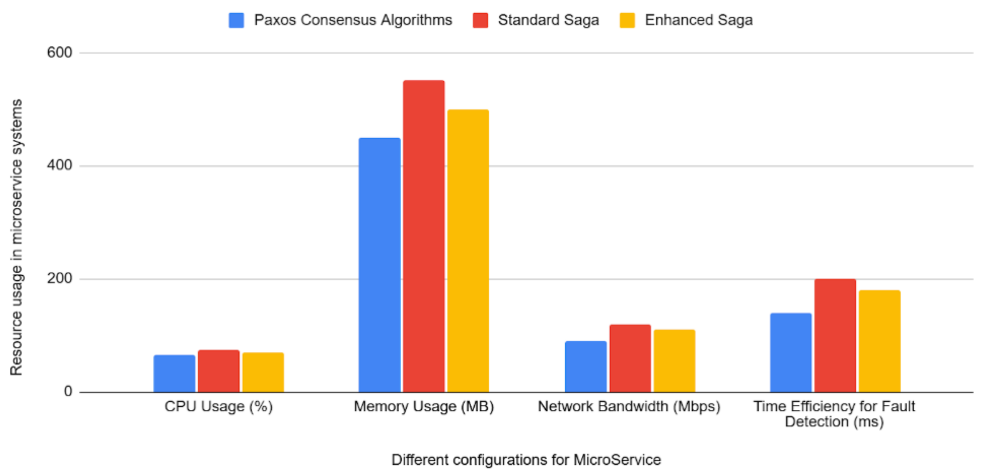
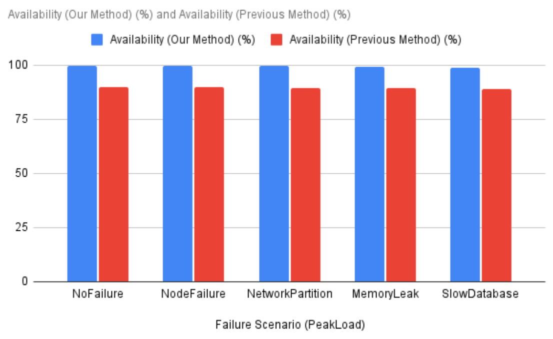
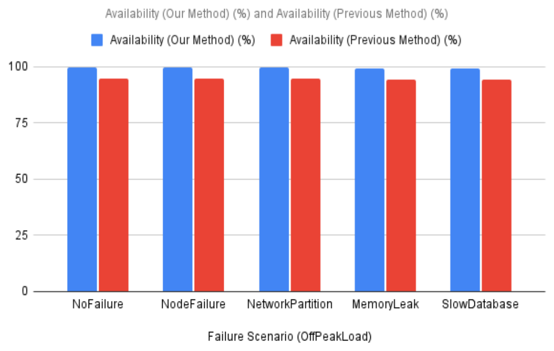


Figure 4. Resource Utilization and Efficiency in Fault Detection



(a) Peak Load Conditions



(b) Off-Peak Load Conditions

Figure 5. Availability Results

.