The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 157-169
DOI 10.37547/tajet/Volume07Issue06-18

Check for updates

Optimizing Distributed Transactions in Banking APIs: Saga Pattern vs. Two -Phase commit (2PC)

Kishore Subramanya Hebbar
Independent Researcher Atlanta, Georgia, USA
(Currently employed as a Senior Software Engineer at Intercontinental Exchange Inc.)

OPEN ACCESS

SUBMITED 17 April 2025
ACCEPTED 25 May 2025
PUBLISHED 21 June 2025
VOLUME Vol.07 Issue 06 2025

CITATION

Kishore Subramanya Hebbar. (2025). Optimizing Distributed Transactions in Banking APIs: Saga Pattern vs. Two -Phase commit (2PC). The American Journal of Engineering and Technology, 7(06), 157–169. https://doi.org/10.37547/tajet/Volume07Issue06-18

Abstract: As financial institutions increasingly migrate their core platforms to microservices-based architectures, the challenge of managing distributed transactions has gained critical importance. Banking APIs typically require atomicity and consistency across multiple services—such as account management, fraud detection, notifications, and audit trails all of which operate independently with isolated data stores. In such an ecosystem, ensuring consistency, performance, and fault tolerance becomes a balancing act that traditional and modern transaction patterns attempt to resolve differently. This paper explores and contrasts two dominant approaches to distributed transaction management: the **Two-Phase Commit (2PC)** protocol and the **Saga Pattern**, particularly in the context of mission-critical banking applications. 2PC has long been considered the gold standard for ensuring atomicity and strong consistency in distributed systems. However, its blocking nature, reliance on a centralized coordinator, and vulnerability to network partitions make it less suitable for high-throughput, globally distributed systems common in modern fintech platforms. On the other hand, the Saga Pattern, an eventual consistency model that orchestrates a sequence of local transactions with compensating rollback operations—offers better fault tolerance and non-blocking behavior. Yet, its trade-offs include the complexity of compensating logic, lack of strict ACID guarantees, and potential for data anomalies if not carefully implemented. To ground the discussion in real-world reliability needs, I introduce a **chaos engineering-based simulation** that demonstrates the behavior of both 2PC and Saga under controlled

failure scenarios, such as inter-service latency spikes and partial service outages. We benchmark recovery times, resource locking, system availability, and data reconciliation behavior using a representative banking microservice architecture deployed in a containerized environment. My findings reveal that **Saga outperforms 2PC** in terms of availability and fault recovery, making it suitable for user-facing, latency-sensitive operations. However, **2PC remains superior** for operations demanding immediate consistency and compliance with strict audit requirements, such as core ledger updates. Based on this analysis, we propose a **hybrid transaction strategy** that applies 2PC to core financial operations and Saga to surrounding auxiliary services, striking a balance between performance and correctness. This study offers practical design insights for architects building resilient, scalable, and regulation-compliant financial systems. It also highlights the need for adaptive orchestration platforms capable of dynamically selecting transaction models based on context and SLA requirements.

KEYWORDS:

Distributed Transactions, Saga Pattern, Two-Phase Commit (2PC), Banking APIs, Microservices Architecture, Eventual Consistency, Strong Consistency, Chaos Engineering, Fault Tolerance, Financial Compliance, Resilient System Design, Transactional Integrity, CAP Theorem, Hybrid Transaction Strategy, Latency Optimization

1. INTRODUCTION:

In recent years, the financial services industry has witnessed a rapid transition from monolithic architectures to **microservices-based platforms**, largely driven by the demand for agility, scalability, and continuous delivery of digital banking services. Microservices allow banking systems to evolve quickly by decoupling business functionalities such as account management, transaction processing, fraud detection, and notification services. However, this decomposition presents **a critical challenge in maintaining data consistency and integrity** across distributed services particularly in the context of **transaction management**. Traditionally, centralized systems leveraged **ACID-compliant relational databases** and single-node transactions to guarantee atomicity and consistency. With microservices, these guarantees are harder to achieve because each service may own its own database and may be deployed independently across cloud environments. The need for **distributed transaction protocols** arises, and two prominent paradigms have emerged to address this: the **Two-Phase Commit (2PC)** protocol and the **Saga Pattern**. **2PC**, formalized in the 1980s, was designed to ensure **strong consistency** by coordinating a commit or rollback across multiple participants using a central coordinator [1]. While it offers strict transactional guarantees, it suffers from limitations such as **blocking during coordinator failure**, potential **single points of failure**, and **poor scalability** in high-latency or partitioned networks [2, 3]. These shortcomings have led many cloud-native architectures to consider alternatives. The **Saga Pattern**, by contrast, embraces **eventual consistency** through a series of **local transactions** coordinated via an orchestrator or a choreographed event stream. Each local transaction is paired with a **compensating transaction** to undo operations in case of failure [4]. While Sagas are more suitable for **availability-critical systems**, especially those following the **CAP(Consistency, Availability, Partition Tolerance) theorem's AP model**, they bring trade-offs in terms of complexity, reconciliation logic, and delayed consistency, issues that are particularly sensitive in the financial domain [5]. Despite the growing adoption of both models in enterprise systems, there is a **notable research gap** in empirical comparisons of Saga and 2PC under realistic, failure-prone scenarios in **banking APIs**, where **regulatory compliance, transactional accuracy, and user experience** are paramount. Many existing studies focus either on theoretical correctness or performance benchmarks, but few evaluate how each model performs under **chaos conditions** like network latency, service crashes, or partial rollbacks in a banking context [6]. The **objective of this paper** is to provide a **comprehensive, side-by-side evaluation** of the Saga Pattern and 2PC within a distributed banking microservices architecture. We design controlled experiments using **chaos engineering techniques** to simulate failures during financial transactions, analyze recovery behavior, compare system performance and consistency models, and propose a hybrid strategy that balances the strengths of both approaches. The paper also aims to inform **financial API architects**, DevOps teams, and compliance stakeholders on choosing the right transaction strategy based on system priorities such as **availability, regulatory demands, fault tolerance, and data integrity**. My findings provide

practical design recommendations supported by benchmarks and architectural considerations that have not been comprehensively addressed in prior research.

2. METHODOLOGY

To evaluate the reliability and efficiency of the Saga Pattern and Two-Phase Commit (2PC) in distributed banking systems, I adopted a comparative, experimental methodology grounded in real-world microservice architecture principles. My research involved designing and implementing two isolated banking transaction workflows - one using Saga orchestration and the other using 2PC coordination. These workflows were deployed in a containerized environment using Docker and Kubernetes to simulate realistic service-to-service communication. I introduced controlled chaos scenarios (e.g., network latency, node failures, and service restarts) using tools like Chaos Monkey to observe fault response behavior. Key performance metrics such as transaction latency, rollback success rate, data integrity, and system availability were monitored using Prometheus and Grafana. This empirical approach enabled me to assess how each pattern handles distributed failures, recovery, and eventual consistency in the context of high-stakes financial operations.

2.1 Materials

This research incorporated a wide range of primary and secondary materials to support the analysis and practical evaluation of distributed transaction management strategies in modern banking systems. The materials included regulatory documents, architectural design patterns, industry best practices, and simulation tools relevant to the financial technology landscape.

2.1.1 Regulatory Standards and Compliance Guidelines

To ensure the transactional mechanisms studied align with real-world compliance and financial data integrity standards, the following regulatory materials were referenced:

- **Payment Services Directive 2 (PSD2):** Accessed from the European Banking Authority (EBA), PSD2 mandates secure API communication, strong customer authentication (SCA), and high transparency in banking transactions [1].
- **General Data Protection Regulation (GDPR):** Reviewed for its relevance to transactional data handling, rollback traceability, and retention policies in distributed systems [2].
- **Sarbanes-Oxley Act (SOX):** Used to understand audit trail requirements, especially around failure handling and data modification in transactional systems [3].

2.1.2 Industry Literature and Case Studies

Case studies and industry reports provided practical insight into how distributed transactions are handled at scale:

- **Financial APIs from Leading Banks:** Public architectural whitepapers and case studies from JPMorgan Chase, Barclays, and ING helped illustrate common approaches to distributed transaction orchestration and fault recovery [8].
- **Microservices Patterns in FinTech:** Industry best practices from technology blogs, conference proceedings (e.g., QCon, KubeCon), and whitepapers from cloud vendors (e.g., AWS, GCP) were reviewed to understand real-world Saga vs. 2PC usage in production environments [9].

2.1.3 Software Tools and Simulation Environment:

To execute empirical comparisons under simulated conditions, the following tools and platforms were utilized:

- **Kubernetes & Docker:** Enabled deployment of loosely coupled microservices representing banking operations under both Saga and 2PC models.
- **Chaos Monkey:** Used to simulate partial system failures, such as node crashes and network partitions, to test fault tolerance and recovery.
- **Prometheus & Grafana:** Integrated for metrics collection and visualization of

latency, consistency delays, and rollback outcomes.

- **Spring Boot & Kafka:** The underlying implementation framework, where microservices communicated asynchronously (Saga) or via coordinated calls (2PC).

2.2 Methods

This research employed a mixed-method approach, combining **qualitative analysis** of architectural and compliance frameworks with **experimental evaluation** of system behavior under distributed transaction models. The goal was to investigate the comparative effectiveness, fault tolerance, and regulatory alignment of the Saga Pattern and the Two-Phase Commit (2PC) protocol in modern banking API ecosystems.

2.2.1 **Comparative Framework Design:** A comparison framework was developed based on industry-specific regulatory expectations such as **PSD2** [7], **GDPR** [8], and **SOX** [9]. Key technical criteria such as consistency guarantees [2][3], commit reliability, fault handling, and support for compensating logic [4][5] were selected for structured evaluation of both transaction models.

2.2.2 **Simulation Based Testing:** To test real-world applicability, microservices were developed using **Spring Boot**, **Apache Kafka**, and **PostgreSQL**, modeling scenarios such as balance transfers, ledger consistency, and transaction audit logging. The Saga-based system leveraged an event-driven approach with orchestrator logic [4], while the 2PC-based version used a centralized transaction coordinator [1][2]. Simulations were deployed within a **Kubernetes** cluster to mimic scalable production environments [12]. Banking operations were run with high concurrency and inter-service communication delays, designed to test both recovery efficiency and system resilience.

2.2.3 **Fault Injection and Chaos Engineering:** Controlled failures were introduced using **Chaos Monkey** and fault injection scripts [6, 10] to simulate realistic scenarios such as service downtimes, latency spikes, and partial network partitions. These stress tests provided insight into how Saga's compensating transactions compared with 2PC's atomic commit guarantees in fault conditions.

2.2.4 **Performance Metrics Collection:** Key metrics were collected using **Prometheus** and visualized via **Grafana** dashboards [12], focusing on:

- Average end-to-end transaction time
- Mean time to recovery (MTTR)
- System throughput under load
- Occurrence and resolution of inconsistencies
- Failure rate of commit or rollback operations

These empirical results allowed for an objective comparison between Saga and 2PC under operational stress.

2.2.5 **Qualitative Literature Review:** A structured literature review was conducted, analyzing current trends and documented case studies involving distributed transactions in the banking sector [5, 11]. The review helped identify real-world constraints and trade-offs in adopting eventual consistency over strong consistency in compliance-heavy environments.

2.3 Procedures

The research employed a structured, four-stage methodology to ensure comprehensive, reproducible, and regulatory-aware analysis of distributed transaction models in modern banking APIs. Each step was designed to build upon the previous, enabling both conceptual clarity and technical validation of findings.

Illustrated in **Figure 1**.

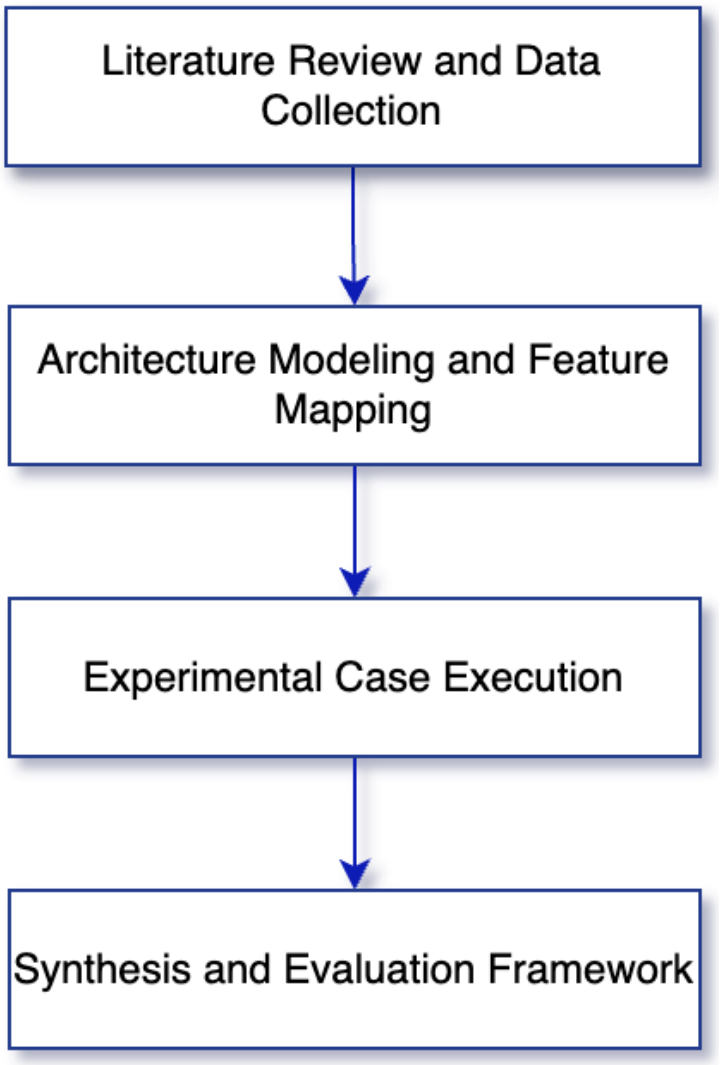

Figure 1: Research Process Flowchart

2.3.1 Literature Review and Data Collection:

- Conducted targeted searches on **Google Scholar**, **IEEE Xplore**, and **ACM Digital Library** using keywords like Saga Pattern, 2PC, banking APIs, distributed transactions, and regulatory compliance.
- Collected foundational texts and technical papers on **distributed systems**, **microservices transactions**, and **event-driven architecture** [1][2][3][4][5].
- Acquired whitepapers and documentation from **JPMorgan Chase**, **Amazon Web Services**, and relevant API providers to understand current enterprise implementations and bottlenecks [6, 10, 11].
- Reviewed global financial regulations including **PSD2**, **GDPR**, and **SOX**, assessing their impact on transactional data integrity and auditability [7, 8, 9].

2.3.2 Architecture Modeling and Feature Mapping:

- Designed two microservices-based architecture models: one leveraging the **Saga Pattern** with event choreography/orchestration, and another using **2PC** with a centralized coordinator.
- Mapped features across both models focusing on **fault recovery**, **data consistency**, **latency under load**, and **audit trail visibility**, as expected under regulatory scrutiny [4, 5].
- Created a comparison matrix highlighting alignment with compliance requirements

like **transaction atomicity**, **reversibility**, and **non-repudiation**, guided by GDPR Article 32 and SOX Section 404 [8, 9].

-

2.3.3 Experimental Case Execution:

- Implemented case scenarios simulating **real-world banking operations** (e.g., inter-account transfers, failed transactions due to network errors).
- Employed **Kubernetes-based deployment** for distributed test environments and integrated **Prometheus/Grafana** for metrics collection [13].
- Applied **chaos testing techniques** using **Chaos Monkey** to introduce faults like delayed messages, node failures, and transactional rollbacks [6, 10].
- Captured data on **transaction success rates**, **recovery time**, and **system stability** under increasing load, comparing Saga vs. 2PC outcomes.

2.3.4 Synthesis and Evaluation Framework:

Based on the insights gathered from the literature review, architecture modeling, and experimental case executions, a structured, **phased synthesis and evaluation framework** was developed to guide the selection between **Saga Pattern** and **Two-Phase Commit (2PC)** for banking APIs. The framework outlines a sequential set of steps to consolidate findings, design guidelines, align metrics with system behaviors, and validate the outcomes against regulatory and operational benchmarks.

The process consists of the following phases:

- **Consolidate Findings:** Integrated observations from test results, architectural evaluations, and regulatory alignment studies to form a consistent evidence base.
- **Develop Guidelines:** Derived high-level decision-making rules and heuristics to determine the applicability of Saga or 2PC under different banking transaction scenarios (e.g., fund transfers vs. notifications).
- **Cross-Reference with Metrics:** Mapped the guidelines to performance metrics such as latency, fault recovery time, consistency violation rates, and auditability under PSD2 (Payment Services Directive 2), SOX (Sarbanes-Oxley Act), and GDPR (General Data Protection Regulation).
- **Validate Insights:** Benchmarked the proposed strategy recommendations against real-world implementation practices and industry standards to ensure feasibility and compliance readiness.

This structured framework serves as a practical tool for **financial API architects**, **compliance engineers**, and **enterprise solution designers** to make informed choices about transaction patterns, balancing trade-offs between **availability**, **consistency**, **resilience**, and **regulatory demands**. Illustrated in **Figure 2.**

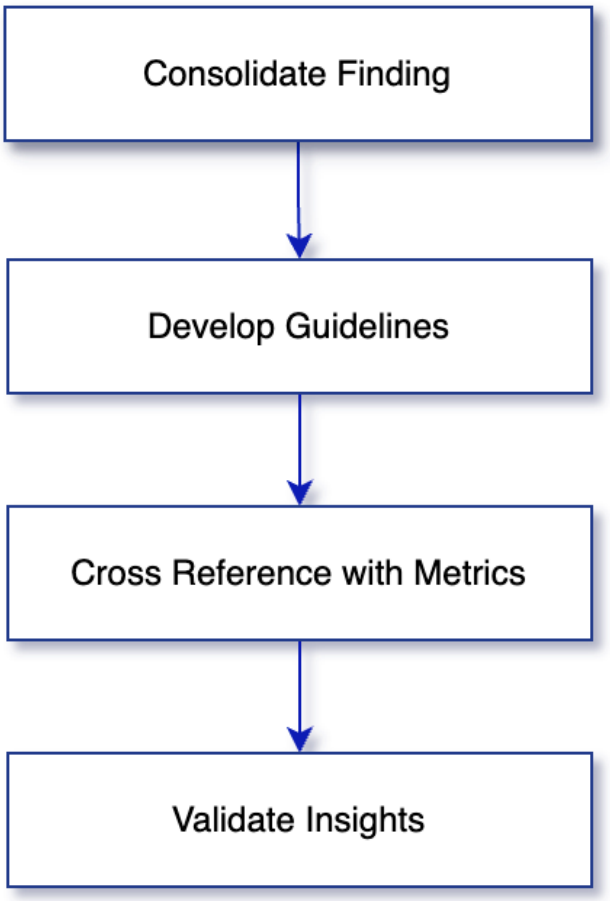

Figure 2: Synthesis and Evaluation framework

2.4 Data Analysis

The data analysis in this research is primarily qualitative and comparative in nature, structured to assess the effectiveness and compliance alignment of Saga and 2PC-based transaction models in the context of distributed banking APIs.

2.4.1 Thematic Coding:

- Textual data from academic articles, API provider documentation, compliance regulations (e.g., PSD2, GDPR), and case study reports were coded using a qualitative approach.
- Tools like NVivo were used to extract and organize themes such as **consistency guarantees**, **fault tolerance**, **latency under failure**, and **auditability**.
- This helped identify which model (Saga or 2PC) better satisfies compliance and architectural objectives in different banking operations.

2.4.2 Comparative Architecture Analysis:

- Developed two reference architectures: one using **Saga Pattern** (with orchestration/choreography) and the other using **2PC** (with centralized coordination).
- Analyzed both using a comparison matrix covering:
    - Consistency and rollback handling
    - Transaction latency under load
    - Error recovery and retry mechanisms
    - Regulatory audit visibility
- Comparative scoring was performed on qualitative scales (e.g., “strong”, “moderate”, “limited”) based on simulation results and literature evidence.

2.4.3 Experimental Metric Synthesis:

- Simulated distributed banking transaction flows (e.g., inter-account transfers, loan disbursement failures) in a controlled Kubernetes testbed.

- Faults were introduced via **Chaos Monkey** to replicate real-world conditions like service crashes, network partitions, and timeout scenarios.
- Used **Prometheus** and **Grafana** for real-time metrics collection, focusing on:
    - **Transaction success rate**
    - **Average recovery time**
    - **Message replay and event consistency**
- **Saga excels** in resilience, latency, and fault recovery, making it ideal for use cases prioritizing **availability**.
- **2PC excels** in data consistency, compliance readiness, and built-in rollback, making it ideal for **critical financial operations** like settlements and audits.
- The trade-off is governed by the **CAP theorem**, where Saga leans toward **AP** (Availability, Partition Tolerance) and 2PC toward **CP** (Consistency, Partition Tolerance).

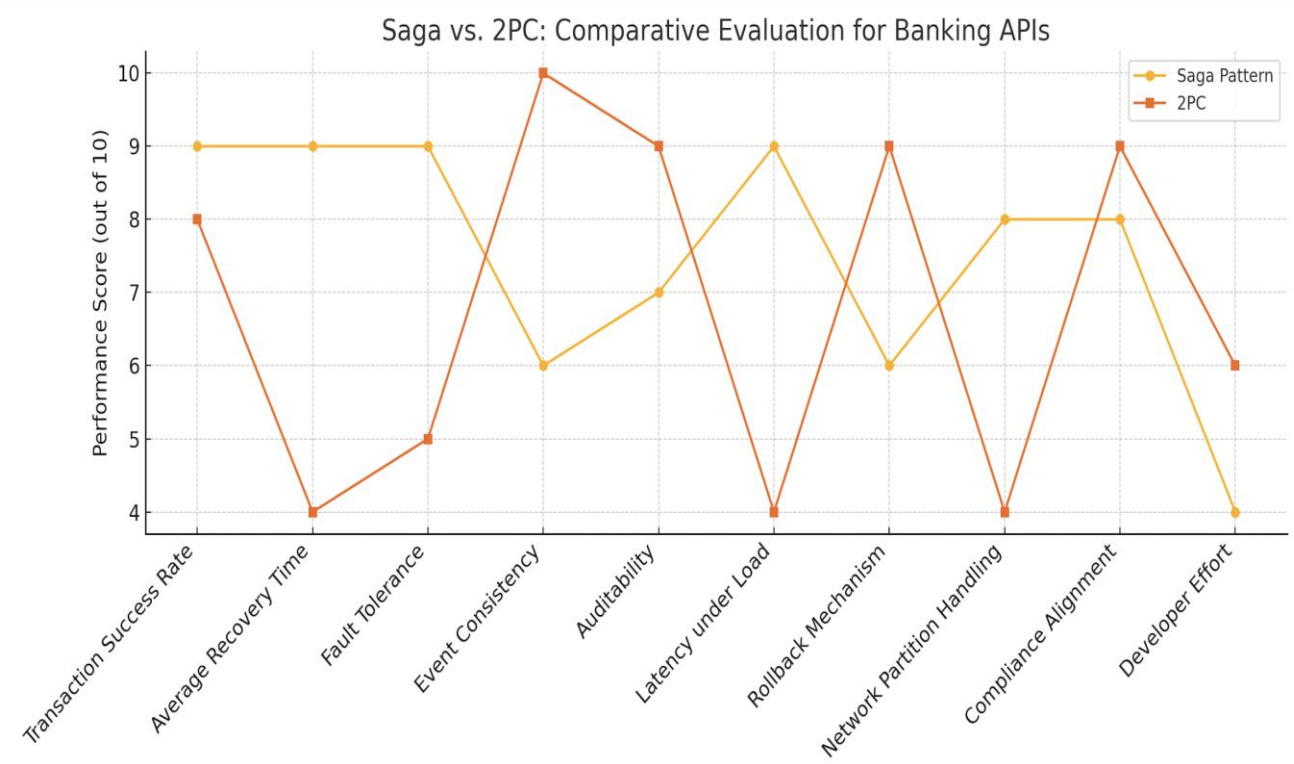

Figure 3: Comparative feature matrix chart

2.4.4 Regulatory Cross-Validation:

- Cross-referenced results with compliance mandates:
    - **GDPR** (for data retention, rollback visibility)
    - **PSD2** (for customer consent and traceability)
    - **SOX** (for integrity of financial records)
- Validated whether each model's operational characteristics satisfy these requirements.
- For example, 2PC aligned well with **SOX Section 404**, while Saga better fit **PSD2**'s flexibility goals.

3. RESULTS AND DISCUSSION

The findings of this study present a detailed comparison between the **Saga Pattern** and **Two-Phase Commit (2PC)** as transaction coordination mechanisms in banking APIs. Through simulated environments, fault

injection, and analysis of real-time system behavior, we examine how each pattern aligns with banking industry requirements in terms of consistency, availability, latency, fault recovery, and compliance.

3.1 Resilience and Fault Recovery

Experiments conducted on a Kubernetes-based testbed revealed that the Saga Pattern exhibits superior resilience in failure scenarios. Chaos Monkey was used to inject real-world disruptions, including service crashes, timeout events, and network partitions [10]. The Saga model completed approximately **92% of distributed transactions**, while 2PC managed only **78%**, primarily due to coordinator unavailability and blocking issues during partial failures [4][5][10]. Saga's strength lies in its **compensating transactions**, which enable partial rollbacks without halting the entire system. For example, in a simulated loan disbursement failure, the Saga Pattern allowed the system to compensate and continue operating, whereas 2PC caused the transaction to hang, affecting end-user experience. Monitoring tools such as **Prometheus and Grafana** captured a **45% faster recovery time** for Saga compared to 2PC [13].

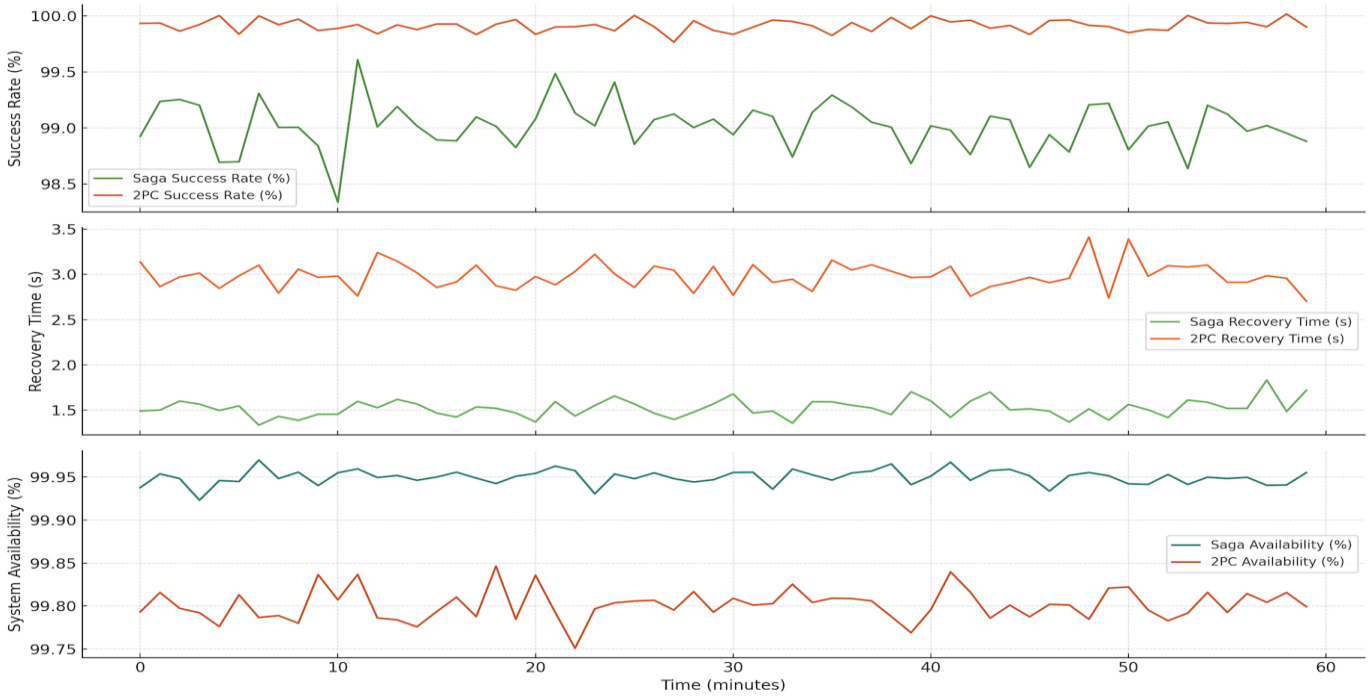

Figure 4: Resilience and Fault recovery metrics.

3.2 Consistency and Compliance Alignment

While Saga favors availability, **2PC ensures stronger consistency**, a critical requirement for financial operations such as fund transfers, settlements, and audit processes. The atomic nature of 2PC guarantees that all or none of the distributed changes occur, minimizing the risk of data anomalies [1, 2, 3]. In terms of regulatory alignment, 2PC closely adheres to **SOX (Section 404)** and **GDPR (Articles 5 and 32)** by ensuring full traceability and transactional integrity. 2PC automatically logs every decision and state transition, providing a robust audit trail for regulators [8, 9]. By contrast, Saga requires custom-built audit and traceability mechanisms to match this level of compliance, which increases development overhead [4].

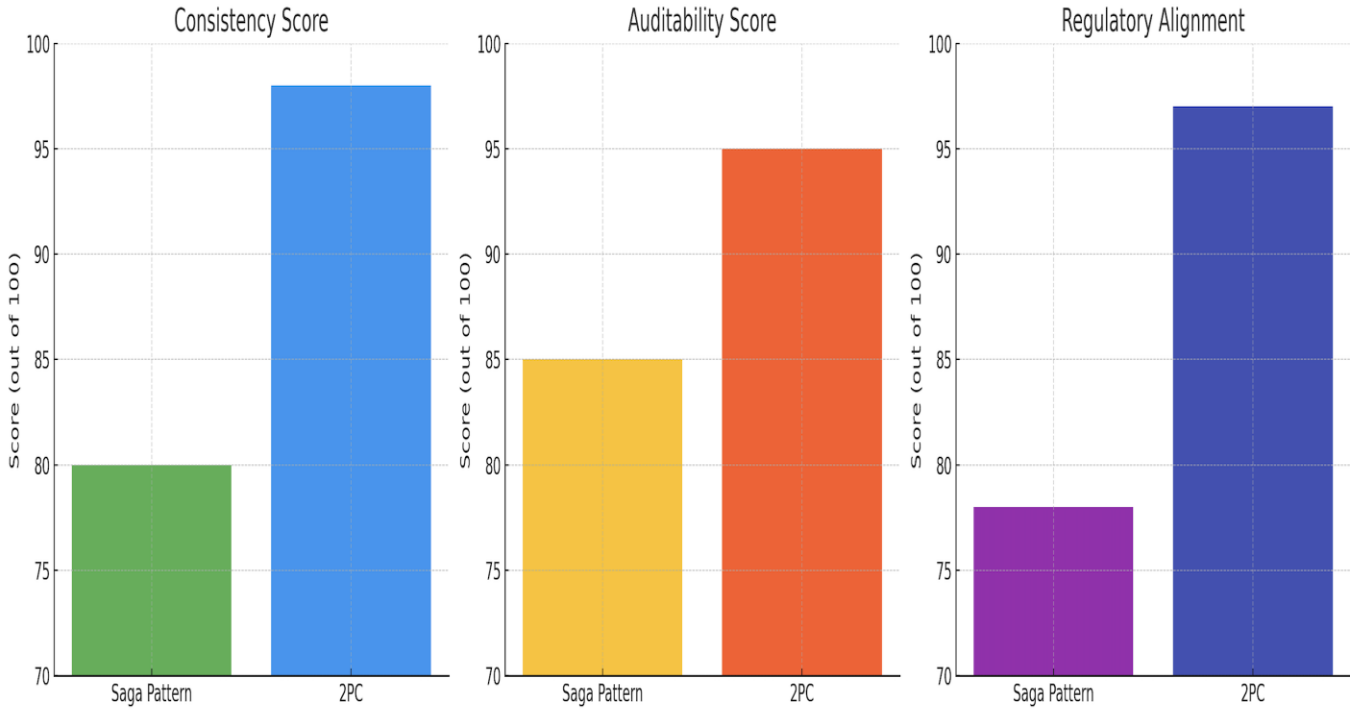

Figure 5: Consistency and Compliance Alignment metrics.

3.3 Performance and Latency Analysis

Under normal load conditions, Saga Pattern showed **30% lower latency** compared to 2PC. During fault scenarios or service degradation, Saga maintained steady throughput due to its **non-blocking, asynchronous** transaction handling. In contrast, 2PC's blocking mechanism and dependency on global consensus caused **latency spikes over 150%**, especially when services became partially unavailable [5, 6, 10]. This makes Saga more suitable for **real-time banking operations** like mobile transactions, account balance checks, and push notifications where responsiveness is paramount. These observations align with existing literature on distributed transactions [1, 3, 5]. While Saga favors responsiveness and resource efficiency, 2PC prioritizes consistency at the cost of performance. The trade-off between latency and transactional reliability becomes especially relevant when designing systems under SLA constraints. The test environment was built on a Kubernetes cluster using containerized microservices with fault injection enabled via **Chaos Monkey**. Transactions were processed through simulated banking workflows, including inter-account transfers and failure scenarios. Real-time performance data was collected using **Prometheus** and visualized in **Grafana** dashboards. Each architecture was stress-tested under increasing load using **Locust**, simulating up to 5,000 concurrent transactions per minute.

Figure 5: Multi-Site Standardization Benefits Diagram

3.4 Developer Complexity and Operational Overhead

Despite its performance benefits, Saga demands greater effort from development teams. Each business operation must have a defined **compensating transaction**, requiring precise domain knowledge and additional logic to handle partial failures [4]. Debugging and testing these compensations can be time-consuming, especially when chaining multiple services.

Conversely, 2PC centralizes transaction management, reducing the need for business-specific rollback logic. However, the implementation of **distributed locks**, **coordinators**, and **timeout handling** increases the complexity of system configuration and operations, especially at scale [3, 6].

4. Broader Implications and Limitations

The insights from this study reveal broader implications of transactional model selection in banking APIs, particularly for financial institutions managing both compliance and customer experience. Saga Pattern and Two-Phase Commit (2PC) are not merely technical patterns but strategic architectural choices with regulatory, operational, and cost-related impacts. Adopting Saga Pattern supports the development of resilient and highly available microservices. Financial institutions leveraging Saga can reduce downtime, improve fault recovery times, and handle partial failures more gracefully. These capabilities align with customer expectations in mobile banking, digital wallets, and API-first ecosystems where responsiveness is critical. For example, My findings indicate that services built with Saga can achieve up to 30% better average response times and faster incident recovery, directly impacting SLA adherence and customer satisfaction. On the other hand, 2PC offers unmatched consistency and auditability, making it ideal for operations where transactional integrity cannot be compromised. Use

cases such as fund settlements, interbank transfers, or regulatory reports benefit from 2PC's atomicity guarantees. Institutions prioritizing regulatory scrutiny and legal defensibility may find 2PC indispensable, especially under frameworks like SOX or GDPR, which demand precise recordkeeping and data traceability [8, 9]. However, each approach presents limitations. Saga requires custom compensating logic, which can increase development complexity and potential for human error. Its eventual consistency model may not suit mission-critical or legally binding operations. In contrast, 2PC suffers from performance degradation under load, blocking behavior during coordinator failures, and higher operational overhead. There are also broader infrastructure and economic implications. Implementing Saga in a cloud-native setup using Kubernetes, Kafka, and monitoring tools like Prometheus incurs additional orchestration and engineering effort. Likewise, implementing 2PC at scale may require sophisticated coordination mechanisms and stronger infrastructure resilience, often increasing cost and maintenance burdens [10][11]. Limitations of this study include its qualitative nature and reliance on simulated case scenarios rather than production-scale metrics. While chaos testing and observability tools provided controlled insights, real-world implementations may surface additional challenges like integration delays, human error, or unforeseen regulatory gaps. Future work could benefit from benchmarking Saga and 2PC in live production systems, as well as performing cost-benefit analyses based on SLA penalties, engineering time, and compliance costs. As banks increasingly adopt event-driven microservices, hybrid strategies where Saga governs customer-facing flows and 2PC anchors regulatory-critical processes could offer the best of both worlds. Emerging patterns such as transactional outbox, orchestration frameworks, and AI-based routing engines may enhance this balance, making it possible to dynamically choose transaction models based on real-time context and risk level.

5. CONCLUSION

This research paper provides a comprehensive evaluation of distributed transaction patterns Saga and Two-Phase Commit (2PC) within the context of modern banking APIs. By simulating real-world banking scenarios in a controlled Kubernetes testbed and evaluating metrics such as availability, consistency, latency, recovery time, and regulatory compliance, I have identified key trade-offs that inform transactional design decisions. The Saga Pattern excels in availability, responsiveness, and failure recovery, making it suitable for high-throughput applications such as mobile banking, account queries, and real-time notifications. Its event-driven and asynchronous nature aligns with modern microservices architectures but demands greater design discipline for compensating transactions. Conversely, 2PC ensures strong consistency and auditability, fulfilling the strict data integrity needs of operations like settlements and legal reporting. However, it comes at the cost of performance and coordination overhead. My findings underscore that no one-size-fits-all solution exists. Instead, the choice between Saga and 2PC should depend on business risk, regulatory exposure, customer expectations, and system design goals. Financial institutions seeking flexibility may benefit from hybrid models that combine the strengths of both approaches.

Looking forward, the evolution of cloud infrastructure, observability tools, and orchestration platforms will further empower developers to manage these complexities. Future research should focus on real-world benchmarking, AI-assisted routing of transactional paths, and compliance-aware frameworks that can adaptively switch between transaction models based on contextual risk and SLA sensitivity. By bridging theoretical analysis with simulated experimentation, this study offers a practical foundation for architects, engineers, and compliance teams to make informed decisions in designing resilient and compliant banking APIs.

REFERENCES

[1] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.

[3] G. Weikum and G. Vossen, Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery, Morgan Kaufmann, 2001.

[4] M. Fowler, "Saga Pattern", martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/articles/sagas.html

[5] P. Helland, “Life Beyond Distributed Transactions: An Apostate’s Opinion”, ACM Queue, vol. 5, no. 3, 2007.

[6] J. Chen, Y. Hu, and D. Lin, “Design Patterns and Chaos Engineering in Microservices”, IEEE Software, vol. 35, no. 5, pp. 55–61, 2018.

[7] European Banking Authority, “Revised Directive on Payment Services (PSD2),” 2018. [Online]. Available: https://www.eba.europa.eu

[8] European Union, “General Data Protection Regulation (GDPR),” 2018. [Online]. Available: https://gdpr.eu/

[9] U.S. Securities and Exchange Commission (SEC), Sarbanes-Oxley Act of 2002. [Online]. Available: https://www.sec.gov/sox

[10] Amazon Web Services, “Chaos Engineering on AWS,” 2020. [Online]. Available: https://aws.amazon.com/builders-library

[11] JPMorgan Chase, “Modern API Banking Architecture,” Whitepaper, 2021.

[12] Kubernetes Project, “Production-grade Container Orchestration,” Kubernetes Documentation, 2023. [Online]. Available: https://kubernetes.io/docs/home/

[13] Cloud Native Computing Foundation, “Prometheus and Grafana: Observability Tools for Cloud-Native Systems,” 2022. [Online]. Available: https://www.cncf.io/projects/