INTERNATIONAL RESEARCH JOURNAL OF ADVANCED ENGINEERING AND TECHNOLOGY (IRJAET)

eISSN: 3087-4068

Volume. 02, Issue. 08, pp. 07-19, August 2025"

Transactional Resilience in Banking Microservices: A Comparative Study of Saga and Two-Phase Commit for Distributed APIs

Lucas Meyer
Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

Article Received: 08/07/2025, Article Accepted: 06/08/2025, Article Published: 31/08/2025

ABSTRACT

Background:
Banking platforms are undergoing rapid transformation as monolithic core systems are decomposed into microservices and exposed via API ecosystems to partners, fintechs, and customer-facing channels. This architectural evolution exposes a long-standing tension in distributed systems: how to maintain transactional correctness across multiple services and databases without compromising availability, scalability, and responsiveness (Bucchiarone et al., 2019; Hasselbring & Steinacker, 2017). Classical distributed transaction protocols such as two-phase commit (2PC) were designed for tightly controlled database clusters, not for heterogeneous, failure-prone microservice landscapes; yet financial institutions still require strong guarantees for money movements and ledger consistency (Mohan et al., 1986; Gray, 1981; Hebbar, 2025).
Objective:
This article develops an in-depth, publication-ready conceptual analysis of distributed transaction management in banking APIs, comparing the Saga pattern and 2PC across architectural, operational, and business dimensions. Building strictly on the given references, it aims to synthesize a theoretically grounded view of when Saga-based orchestration or choreography should be preferred over 2PC, how each approach behaves under microservice failures, and how their trade-offs align with the constraints of the CAP theorem, reactive systems thinking, and modern high-availability design patterns (Garcia-Molina & Salem, 1987; Boner et al., 2018; Gilbert & Lynch, 2012; Ahluwalia & Jain, 2006; Hebbar, 2025).
Methods:
A qualitative methodology is used, consisting of a structured narrative review and conceptual comparison. Foundational work on transactions, recovery, and distributed databases is combined with contemporary literature on microservices, reactive architectures, and distributed transactions in microservice ecosystems (Haerder & Reuter, 1983; Mohan et al., 1986; Thomson et al., 2012; Bucchiarone et al., 2019; Salah et al., 2016; Lungu & Nyirenda, 2024). This is complemented by focused analyses of seminal and recent Saga-related contributions, including the original Saga formulation, enhanced Saga variants, and Saga frameworks such as SagaMAS, as well as refinements of 2PC for replicated state machines (Garcia-Molina & Salem, 1987; Limón et al., 2018; Daraghmi et al., 2022; Uyanik & Ovatman, 2020). Hebbar's banking-specific comparison of Saga and 2PC serves as an anchor for financial-domain interpretation (Hebbar, 2025).
Results:
The synthesis shows that 2PC offers simple, strong atomicity semantics in relatively synchronous and tightly coupled environments but becomes increasingly brittle in microservice ecosystems characterized by partial failures, independent deployability, and cross-organizational boundaries (Mohan et al., 1986; Gray, 1981; Helland, 2007; Helland & Campbell, 2009). Enhancements like those proposed for replicated state machines mitigate some blocking behavior but do not fundamentally resolve the tension between global coordination and availability (Uyanik & Ovatman, 2020; Gilbert & Lynch, 2012). In contrast, Saga-based approaches trade strict atomicity for long-lived, semantically compensating transactions, aligning more naturally with microservice autonomy, reactive principles, and high availability but introducing complexity in the design of compensations, partial-failure handling, and observability (Garcia-Molina & Salem, 1987; Richardson, 2019; Štefanko et al., 2019; Lungu & Nyirenda, 2024; Hebbar, 2025).

Conclusion:
For modern banking APIs that must support openness, elasticity, and resilience while managing business-critical money flows, the evidence favors Saga-based designs as the default strategy for cross-service transactions, with 2PC reserved for bounded domains with strong coordination capabilities and tight control over participants (Hebbar, 2025; Helland, 2007; Lungu & Nyirenda, 2024). The article argues that practitioners should frame Saga versus 2PC not as a binary choice but as part of a broader design space governed by domain semantics, consistency requirements, and architectural boundaries, and calls for further empirical research on performance, fault behavior, and developer productivity under each paradigm.

KEYWORDS

Distributed transactions; Saga pattern; two-phase commit; banking APIs; microservices architecture; reactive systems; consistency and availability.

INTRODUCTION

Banking systems have historically been built around large, centralized transaction processing engines designed to provide strong guarantees about the durability and atomicity of financial operations (Gray, 1981; Haerder & Reuter, 1983; Little et al., 2004). These engines often operate on tightly integrated databases, with transaction managers enforcing ACID properties over carefully engineered data schemas and transaction logs (Mohan et al., 1986; Haerder & Reuter, 1983). For decades, this architectural approach aligned well with the organizational and technological environment of financial institutions: batch-based interfaces to external parties were relatively limited, latency constraints were primarily internal, and regulatory requirements were satisfied through strong internal consistency and periodic reconciliation (Gray, 1981; Haerder & Reuter, 1983).

Over the last decade, however, the context for banking software has changed dramatically. The rise of open banking, API-first business models, and fintech ecosystems has transformed banks from relatively closed, monolithic operators into participants in a broader digital ecosystem (Hebbar, 2025). Banking APIs now mediate interactions with mobile apps, partner platforms, payment gateways, and embedded finance providers, all expecting low latency, high availability, and resilient behavior even in the presence of network partitions and partial system failures (Hasselbring & Steinacker, 2017; Bucchiarone et al., 2019). At the same time, internal systems have shifted toward microservices architectures to improve agility, scalability, and fault isolation, decomposing previously monolithic core banking systems into independently deployable services that own their data and evolve at different speeds (Bucchiarone et al., 2019; Richardson, 2019; Salah et al., 2016).

This shift introduces a fundamental tension in transactional semantics. On the one hand, the classical transaction concept, as articulated by Gray, emphasizes atomicity, consistency, isolation, and durability as non-negotiable virtues for financial correctness, while acknowledging their limitations in the face of distributed failures and long-running operations (Gray, 1981; Haerder & Reuter, 1983). On the other hand, the CAP theorem demonstrates that in the presence of network partitions, systems must trade off between consistency and availability, making it impossible to provide strong, synchronous, globally consistent transactions across arbitrarily distributed services without sacrificing availability (Gilbert & Lynch, 2012).

Traditional distributed transaction protocols such as two-phase commit were designed in an era where distributed database nodes were often within the same administrative domain and network environment, making it reasonable to coordinate transactions across partitions using global transaction managers and synchronous logs (Mohan et al., 1986; Haerder & Reuter, 1983). These protocols provide strong atomicity and consistency by coordinating commit decisions across participants, but they do so by introducing blocking behavior and tight coupling, which are increasingly at odds with the goals of microservices architectures and cloud-native deployments (Uyanik & Ovatman, 2020; Bucchiarone et al., 2019).

Microservices, by contrast, emphasize decentralized data management, independent deployment, fault isolation, and autonomous scaling (Bucchiarone et al., 2019; Richardson, 2019; Salah et al., 2016). They are often deployed across heterogeneous infrastructures, potentially spanning on-premise data centers and multiple clouds, and may include third-party services outside the bank's direct control (Hasselbring & Steinacker, 2017; Hebbar, 2025). In such an environment, coordinating a global two-phase commit across many services becomes not only operationally difficult, but also conceptually misaligned with the principle that each service should own its data and remain independently evolvable (Richardson, 2019; Helland, 2007; Helland & Campbell, 2009).

The Saga pattern emerged as an alternative approach to long-running and distributed transactions, originally proposed as a way to break large transactions into sequences of smaller, individually committed sub-transactions, each with a defined compensating operation in case later steps fail (Garcia-Molina & Salem, 1987;

Campbell & Richards, 1981). Instead of relying on a single atomic commit across all participants, Sagas accept that global atomicity is relaxed in favor of eventual consistency managed through semantic compensations. This model maps naturally to microservices where each service can expose local transactions and compensating actions through APIs, orchestrated or choreographed to achieve business-level consistency (Richardson, 2019; Bucchiarone et al., 2019; Lungu & Nyirenda, 2024).

In the context of banking, this architectural choice is far from trivial. Money movements, ledger updates, and risk controls often appear to require strong atomicity; yet the practical realities of distributed systems and business processes suggest that compensating semantics, idempotent operations, and multi-step business workflows are already ubiquitous (Helland, 2007; Helland & Campbell, 2009; Hebbar, 2025). Hebbar's detailed comparison of Saga and 2PC in banking APIs provides a contemporary perspective on how these patterns behave in production-grade financial landscapes, highlighting trade-offs in latency, availability, operational complexity, and business semantics (Hebbar, 2025).

At the same time, the emergence of reactive systems thinking emphasizes responsiveness, resilience, elasticity, and message-driven interaction as core principles for systems that must remain responsive under varying loads and failure conditions (Boner et al., 2018). Saga implementations in reactive microservices, as explored by Štefanko and colleagues, demonstrate how reactive tools can be used to structure long-running distributed workflows while maintaining non-blocking behavior and backpressure awareness (Štefanko et al., 2019). Saga frameworks such as SagaMAS further attempt to generalize these patterns into reusable middleware, showing how Saga concepts can be embedded into microservice platforms (Limón et al., 2018).

Despite this growing body of work, there remains a lack of integrated, domain-specific analysis that brings together foundational transaction theory, modern distributed systems perspectives, microservice architecture patterns, and banking-domain requirements into a coherent evaluation of Saga versus 2PC. Lungu and Nyirenda's systematic literature review on distributed transactions in microservices identifies a broad range of approaches and challenges, but emphasizes that the field still lacks consolidated guidance for practitioners facing real-world design decisions (Lungu & Nyirenda, 2024). Hebbar's banking-focused article provides critical domain grounding but does not aim to exhaust the theoretical landscape (Hebbar, 2025).

This article addresses that gap by providing an extensive, theory-informed synthesis that compares Saga and 2PC across architecture, consistency, availability, failure recovery, and operational dimensions, with banking APIs as the focal domain. It situates these patterns in the broader context of microservice architectures, CAP constraints, reactive principles, and high-availability design patterns, offering a nuanced account of where each approach excels, where it fails, and how hybrid strategies can be constructed (Bucchiarone et al., 2019; Ahluwalia & Jain, 2006; Boner et al., 2018; Gilbert & Lynch, 2012; Hebbar, 2025).

METHODOLOGY

The methodological approach adopted in this study is qualitative, integrative, and theory-building rather than empirical. It is grounded strictly in the supplied reference corpus, which spans foundational database transaction research, distributed systems theory, microservice and reactive architecture literature, and contemporary work on Saga and 2PC in microservice and banking contexts (Gray, 1981; Haerder & Reuter, 1983; Mohan et al., 1986; Bucchiarone et al., 2019; Hebbar, 2025; Lungu & Nyirenda, 2024).

The first phase of the methodology consists of a structured narrative review. References are categorized according to their primary focus:

Works such as those by Mohan and colleagues, Gray, and Haerder and Reuter are treated as foundational sources on transaction models, logging, and recovery, providing formal properties and design principles for ACID transactions and distributed transaction management (Gray, 1981; Haerder & Reuter, 1983; Mohan et al., 1986). Calvin's design for fast distributed transactions, as described by Thomson and co-authors, is examined as an example of an alternative architecture that uses deterministic ordering to enable distributed transactions at scale, illustrating how different assumptions lead to different protocol designs (Thomson et al., 2012).

Work by Garcia-Molina and Salem, Campbell and Richards, and the later Saga literature is categorized as Saga-focused contributions. Garcia-Molina and Salem's original SAGAS paper formalizes the Saga concept and compensation semantics, while Campbell and Richards demonstrate an early system-level application that automates software production processes using Saga-like mechanisms (Garcia-Molina & Salem, 1987; Campbell & Richards, 1981). Later work by Limón and colleagues, Daraghmi and co-authors, and Štefanko and collaborators explores Saga frameworks, enhancements, and adaptations for microservices and reactive environments (Limón et al., 2018; Daraghmi et al., 2022; Štefanko et al., 2019).

Literature on microservices architecture and distributed systems evolution, including Bucchiarone and colleagues, Hasselbring and Steinacker, Salah and co-

authors, and Richardson, is treated as the primary source for architectural context and design patterns (Bucchiarone et al., 2019; Hasselbring & Steinacker, 2017; Salah et al., 2016; Richardson, 2019). This is complemented by Messina and colleagues' work on the Database-is-the-Service pattern, which highlights the impact of data ownership and database-as-a-service concepts on transaction design (Messina et al., 2016).

Research on distributed systems theory, including Gilbert and Lynch's perspectives on the CAP theorem and Helland's essays on life beyond distributed transactions and building on quicksand, is used to interpret architectural patterns in light of fundamental impossibility results and practical experiences with distributed state and failure management (Gilbert & Lynch, 2012; Helland, 2007; Helland & Campbell, 2009).

The systematic review by Lungu and Nyirenda is used as a meta-level source summarizing contemporary approaches and challenges in managing distributed transactions in microservice architectures, providing a broad landscape into which Saga and 2PC must be situated (Lungu & Nyirenda, 2024).

Finally, domain-specific and pattern-focused contributions such as Hebbar's banking API article, Ahluwalia and Jain's high-availability design patterns, the Reactive Manifesto by Boner and co-authors, Fowler's CQRS description, and Little and colleagues' Java transaction processing book provide concrete design options, domain constraints, and practical implications that ground the theoretical analysis in realistic system design (Ahluwalia & Jain, 2006; Little et al., 2004; Boner et al., 2018; Fowler, 2018; Hebbar, 2025).

In the second phase, a conceptual comparison framework is constructed. This framework identifies key dimensions along which Saga and 2PC can be compared in the context of banking microservices:

1. Consistency semantics and isolation behavior.

2. Availability under network partitions and node failures.

3. Latency and impact on end-to-end response times.

4. Coupling between services and implications for independent deployability.

5. Failure handling, recovery, and business-level compensation.

6. Alignment with microservice and reactive design principles.

7. Operational complexity, observability, and tooling.

Each dimension is informed by relevant references. For example, consistency semantics and isolation behavior are grounded in Gray's transaction concept, Haerder and Reuter's recovery principles, and Mohan et al.'s R* transaction management (Gray, 1981; Haerder & Reuter, 1983; Mohan et al., 1986). Availability under partitions is interpreted through CAP perspectives (Gilbert & Lynch, 2012). Coupling and independent deployability are analyzed using microservice design principles and patterns (Bucchiarone et al., 2019; Richardson, 2019).

The third phase consists of an interpretive synthesis in which Hebbar's comparison of Saga and 2PC in banking APIs is read in light of the conceptual framework, connecting observed trade-offs and design recommendations to foundational theory and broader microservice practice (Hebbar, 2025). Lungu and Nyirenda's systematic literature review is used to cross-check whether the conceptual picture aligns with trends and challenges observed across multiple studies (Lungu & Nyirenda, 2024).

Throughout, the analysis remains descriptive and argumentative rather than computational; it does not introduce new experimental data or performance measurements but instead elaborates on the theoretical implications and practical interpretations of the cited work. This methodology is suited to providing deep explanatory insights into why particular patterns behave as they do in modern banking architectures, and to articulating nuanced guidance for practitioners and researchers.

RESULTS

Foundations of Distributed Transactions in Banking Systems

The classical foundation of transaction management in banking lies in the ACID properties and the transaction concept as formulated by Gray and elaborated by Haerder and Reuter (Gray, 1981; Haerder & Reuter, 1983). In this model, a transaction encapsulates a sequence of read and write operations on a database that must appear as an indivisible unit: either all its effects are visible or none are, ensuring atomicity; it moves the database from one consistent state to another, preserving systemic invariants; it is isolated from concurrent transactions so that intermediate states are not observed; and its effects survive failures, ensuring durability (Gray, 1981; Haerder & Reuter, 1983).

In centralized systems, guaranteeing these properties relies on careful coordination between the transaction manager, the concurrency control subsystem, and the recovery manager, which uses logging and checkpointing to rebuild a consistent state after failures (Haerder &

Reuter, 1983; Little et al., 2004). Banking ledgers, account balances, and payment records have traditionally been managed within this framework, with local transactions executed inside a single core banking database or tightly coupled cluster under the control of one administrative domain.

As banking systems evolved to distributed databases, protocols for coordinating transactions across nodes became necessary. Mohan and colleagues' work on the R* distributed database system illustrates how subsystems at different sites can participate in a global transaction while maintaining local log and recovery mechanisms (Mohan et al., 1986). In such systems, the two-phase commit protocol emerged as the dominant mechanism for ensuring that distributed transactions either commit on all involved sites or abort everywhere, thereby maintaining global atomicity (Mohan et al., 1986; Little et al., 2004).

Two-phase commit operates through a coordinator that first sends a prepare request to all participants and waits for confirmation that each is ready to commit; if all vote yes, it then sends a commit decision, otherwise it sends an abort (Mohan et al., 1986; Uyanik & Ovatman, 2020). Logging ensures that if failures occur, participants can determine whether to commit or abort by consulting their logs. This protocol guarantees atomicity but introduces blocking: if the coordinator fails at certain points in the protocol, participants may be left in uncertain states until it recovers, and if communication is disrupted, the system must wait or employ complex failure detection strategies (Mohan et al., 1986; Gray, 1981).

For decades, such protocols were deemed acceptable within controlled environments where nodes shared similar failure characteristics and communication was relatively reliable. However, even early researchers recognized limitations; Gray explicitly noted that global transactions can become long-running and contentious, arguing for careful use of locking and timeout policies to avoid undue blocking (Gray, 1981). Haerder and Reuter similarly emphasized that recovery mechanisms must balance robustness with performance and complexity (Haerder & Reuter, 1983).

With the rise of partitioned, high-throughput distributed systems, alternative designs such as Calvin emerged. Thomson and colleagues describe Calvin as a transactional system that performs deterministic ordering of transactions across partitions, enabling high-performance distributed transactions by predefining execution order and replicating logs (Thomson et al., 2012). While Calvin is not specifically targeted at banking APIs, it underscores that the design space for distributed transactions is broader than 2PC alone; however, it assumes a fairly homogeneous, tightly integrated environment, not the heterogeneous microservice ecosystems now common in banking (Thomson et al., 2012; Bucchiarone et al., 2019).

Gilbert and Lynch's perspective on the CAP theorem provides a theoretical lens through which to interpret the behavior of distributed transaction protocols in the presence of network partitions (Gilbert & Lynch, 2012). They clarify that in partitioned systems, one cannot simultaneously guarantee strong consistency and availability; choices must be made about which properties are prioritized in the face of failures, and these choices are often reflected in protocol design. Two-phase commit, by requiring synchronous agreement among participants, tilts toward consistency at the potential expense of availability when partitions occur.

In parallel, practitioners such as Helland began to argue that reliance on distributed transactions in large-scale systems is akin to building on quicksand, pointing out that coordination across autonomous services is fragile and can severely restrict scalability and evolution (Helland, 2007; Helland & Campbell, 2009). Helland's "Life Beyond Distributed Transactions" articulates an alternative philosophy, one in which services operate with local transactions and communicate through messages, employing idempotency, at-least-once delivery, and semantic compensation to achieve business-level reliability rather than strict global atomicity (Helland, 2007; Helland & Campbell, 2009).

These foundational insights set the stage for understanding why the Saga pattern has gained prominence as microservices and distributed APIs have become the norm in banking. They also explain why financial institutions cannot simply transplant classical distributed transaction protocols into their new architectures without encountering severe issues in availability, complexity, and organizational alignment (Bucchiarone et al., 2019; Hebbar, 2025; Lungu & Nyirenda, 2024).

Microservices, Data Ownership, and the Transactional Challenge

Microservices architectures reconfigure the boundaries of systems, services, and databases in ways that fundamentally affect transaction management. Bucchiarone and colleagues define microservices as independently deployable, loosely coupled services that implement business capabilities, stressing the importance of autonomy, fine-grained decomposition, and decentralized data management (Bucchiarone et al., 2019). Hasselbring and Steinacker, analyzing e-commerce systems, illustrate how microservices can improve scalability, agility, and reliability by avoiding the tight coupling and shared database bottlenecks typical of monolithic architectures (Hasselbring & Steinacker, 2017). Salah and co-authors trace the evolution from traditional distributed systems to microservices, noting that this evolution is accompanied by new challenges in

service coordination, fault tolerance, and data consistency (Salah et al., 2016).

In microservices, it is common for each service to own its database, applying patterns like Database-per-Service to avoid shared schemas and cross-service joins (Richardson, 2019; Messina et al., 2016). Messina and colleagues propose the Database-is-the-Service pattern, in which databases are themselves treated as services, accessed through API-like interfaces rather than direct cross-service table access (Messina et al., 2016). This pattern reinforces the autonomy of services but also makes it more difficult to enforce cross-service transactional invariants via traditional means, because no single transaction manager controls all data.

Richardson's catalog of microservice patterns explicitly positions distributed transactions as a problematic fit for microservices, arguing that they violate local autonomy, hinder independent deployment, and increase coupling (Richardson, 2019). Instead, he suggests patterns such as Saga for managing cross-service workflows and CQRS for separating read and write models, thereby reducing the need for strongly consistent cross-service transactions (Richardson, 2019; Fowler, 2018).

The Reactive Manifesto further emphasizes that modern systems, including those built on microservices, should be responsive, resilient, elastic, and message-driven, advocating event-driven communication and backpressure-aware design (Boner et al., 2018). These principles align more closely with messaging and Saga-based coordination than with synchronous, blocking two-phase commit protocols.

In the context of banking APIs, this architectural reorientation is particularly challenging because financial operations often cross multiple services: customer identity services, risk scoring, account ledgers, payment gateways, notification systems, and fraud detection all participate in workflows such as "initiate payment" or "open account" (Hebbar, 2025). If these services own their data and must remain independently deployable, imposing a global two-phase commit across them for every cross-service operation can quickly become impractical, especially when some participants may be external third parties not under the bank's operational control (Hasselbring & Steinacker, 2017; Helland, 2007; Hebbar, 2025).

High-availability design patterns described by Ahluwalia and Jain highlight that systems aiming for continuous operation must avoid single points of failure and minimize blocking interactions that can cascade into outages (Ahluwalia & Jain, 2006). Two-phase commit, with its centralized coordinator and potential blocking behavior, can conflict with these patterns, whereas message-driven, asynchronous coordination is more compatible with failover, replication, and graceful degradation strategies (Gilbert & Lynch, 2012; Helland, 2007; Boner et al., 2018).

Lungu and Nyirenda's systematic literature review on distributed transactions in microservices confirms that the prevalence of microservice architectures has led to a shift away from classical distributed transactions toward more decentralized patterns, with Saga emerging as a dominant approach for orchestrating distributed workflows in a way that respects service autonomy (Lungu & Nyirenda, 2024). They note that while Saga is far from a silver bullet, it aligns better with the predominant architecture of microservices and addresses many of the practical concerns raised by Helland and others about traditional distributed transactions (Helland, 2007; Helland & Campbell, 2009; Lungu & Nyirenda, 2024).

Against this background, the question for banking APIs is not simply "Saga or 2PC?" but "Under what conditions and architectural constraints should Saga or 2PC be employed, and what hybrid strategies are viable?" Hebbar's banking-specific analysis provides domain illustrations of this question, comparing Saga and 2PC in the context of distributed banking APIs and evaluating their trade-offs from a financial services perspective (Hebbar, 2025).

The Saga Pattern: Origins, Semantics, and Evolution

The Saga pattern, as formulated by Garcia-Molina and Salem, conceptualizes a long-lived transaction as a sequence of smaller, atomic sub-transactions, each operating on a single database and committing independently (Garcia-Molina & Salem, 1987). A Saga is defined as an ordered set of sub-transactions, each with an associated compensating transaction that semantically reverses its effects if a later step fails. The overall consistency of the system is preserved not by a global atomic commit but by ensuring that if the Saga cannot complete successfully, compensations are executed backward to bring the system to a state that is semantically equivalent to having never started the Saga (Garcia-Molina & Salem, 1987).

This model was originally proposed for databases but quickly found application domains beyond pure data management. Campbell and Richards' SAGA system for automating software production management demonstrates how Saga concepts can be used to orchestrate complex, multi-step processes involving compilation, testing, packaging, and deployment, each potentially requiring compensation if later steps fail (Campbell & Richards, 1981). This early system illustrates that Sagas are not limited to short-lived database transactions but can accommodate long-running processes with heterogeneous side effects, a property highly relevant to banking workflows such as opening accounts or processing multi-leg payments.

In microservice architectures, Sagas are often implemented using either orchestration or choreography (Richardson, 2019; Bucchiarone et al., 2019; Lungu & Nyirenda, 2024). In orchestration, a central Saga orchestrator coordinates the steps, invoking each service's API in turn and triggering compensations as needed. In choreography, services emit and listen to domain events, reacting to events representing Saga progression or failure and taking compensating actions autonomously. Both approaches have advantages and drawbacks: orchestration centralizes control and simplifies reasoning but can become a single point of failure and a conceptual "mini-monolith," while choreography preserves service autonomy but can lead to complex, emergent behavior that is harder to understand and debug (Richardson, 2019; Lungu & Nyirenda, 2024).

Daraghmi and co-authors propose enhancements to the Saga pattern for distributed transactions within microservices, aiming to improve its ability to handle failures and maintain consistency across services (Daraghmi et al., 2022). Their work refines compensation strategies, explores ordering and dependency management among Saga steps, and highlights the importance of idempotency and compensation semantics for maintaining correctness. These enhancements are particularly relevant in banking, where compensating transactions must be carefully designed to avoid violations of regulatory constraints or unintended financial side effects.

Limón and colleagues' SagaMAS framework exemplifies how Saga can be packaged into a software framework tailored for microservices, providing abstractions, APIs, and runtime support for defining, executing, and monitoring Sagas (Limón et al., 2018). SagaMAS emphasizes flexibility and integration, illustrating how Saga logic can be encapsulated, instrumented, and reused across different microservice-based applications.

Štefanko, Chaloupka, and Rossi focus on the Saga pattern in a reactive microservices environment, exploring how reactive programming techniques can be used to implement Sagas in systems that use non-blocking, event-driven communication (Štefanko et al., 2019). Their work demonstrates how Sagas can be modeled as reactive streams, integrating Saga steps and compensations into reactive flows that can leverage backpressure, asynchronous scheduling, and compositional operators. This reactive interpretation aligns Saga with the principles of the Reactive Manifesto, enabling Sagas to operate in systems where responsiveness, elasticity, and resilience are primary goals (Boner et al., 2018; Štefanko et al., 2019).

From a theoretical perspective, Saga can be seen as embracing the limitations articulated by the CAP theorem and the practical concerns of large-scale distributed systems. By decomposing global transactions into local ones and relying on compensation rather than atomic commit, Sagas avoid the need for synchronous agreement across all participants in the face of partitions, favoring availability and partition tolerance at the cost of complex, domain-specific compensation logic (Gilbert & Lynch, 2012; Helland, 2007; Garcia-Molina & Salem, 1987).

However, this does not mean Saga is a universal solution. Lungu and Nyirenda's systematic review notes that Saga implementations face challenges in defining correct compensations, managing orchestration complexity, dealing with concurrent Sagas that touch overlapping data, and providing strong guarantees about invariants that span services (Lungu & Nyirenda, 2024). In banking, these challenges are especially acute, because domain rules are stringent and compensations may have legal and financial implications if not correctly defined and executed (Hebbar, 2025).

Two-Phase Commit: Strengths, Enhancements, and Limitations

Two-phase commit remains the canonical protocol for distributed atomic commit in traditional distributed databases. Its core strength lies in its simple correctness property: either all participants commit or none do, ensuring global atomicity for transactions that span multiple resources (Mohan et al., 1986; Little et al., 2004). In banking environments where all participating systems are under centralized control, network conditions are stable, and low availability during rare failures is acceptable, 2PC can serve as an effective mechanism for guaranteeing strong consistency (Mohan et al., 1986; Gray, 1981).

Yet researchers and practitioners have long recognized limitations of 2PC in more complex environments. The protocol can block when the coordinator fails or when communication is disrupted at particular stages, leading to "in-doubt" transactions that prevent resources from making progress (Gray, 1981; Haerder & Reuter, 1983). Adding timeouts and heuristics can mitigate some blocking, but at the cost of potentially violating strict atomicity and requiring manual resolution, which is especially problematic in safety-critical domains such as banking (Little et al., 2004; Uyanik & Ovatman, 2020).

To address some of these limitations, Uyanik and Ovatman propose enhancements to the two-phase commit protocol tailored for replicated state machines, a common model in distributed systems where services maintain copies of state via consensus algorithms (Uyanik & Ovatman, 2020). Their work aims to reduce blocking and improve performance by integrating 2PC with replication mechanisms, but the resulting system remains complex and bounded to environments where such replication strategies are applicable and tightly controlled (Uyanik & Ovatman, 2020).

Despite these enhancements, 2PC retains core characteristics that make it difficult to apply in microservice-based banking APIs. It assumes a well-defined set of participants, a coordinator with authority, and a shared willingness to block during coordination; yet in microservice systems, services may appear and disappear dynamically, belong to different teams or organizations, and have differing availability requirements and deployment schedules (Bucchiarone et al., 2019; Hasselbring & Steinacker, 2017; Salah et al., 2016).

Helland's critiques of distributed transactions emphasize this mismatch. He argues that, in many real-world systems, cross-service operations must be designed in such a way that each service maintains its own durability and correctness, and that global consistency is achieved through business-level mechanisms, messaging, and reconciliation rather than through strict atomic commit (Helland, 2007; Helland & Campbell, 2009). His characterization of distributed transactions as "quicksand" highlights that while they appear to provide firm ground for developers, they can lead to subtle entrapment as systems scale and evolve (Helland & Campbell, 2009).

Gilbert and Lynch's CAP perspective further clarifies that in the presence of partitions, protocols like 2PC that require synchronous coordination across participants cannot guarantee availability, and will either block or require engineers to break atomicity guarantees by introducing heuristic decisions (Gilbert & Lynch, 2012; Uyanik & Ovatman, 2020). For public banking APIs that must remain responsive even during partial outages, this trade-off is often unacceptable; customers and partners expect graceful degradation rather than complete inability to process operations for extended periods (Hasselbring & Steinacker, 2017; Hebbar, 2025).

Nonetheless, Hebbar notes that 2PC may still have a role in specific subdomains of banking systems, particularly within tightly controlled internal clusters where strong atomicity is critical and where service boundaries align with database partitions that can be coordinated effectively (Hebbar, 2025). For example, internal ledger systems that operate within a single data center or controlled cluster may still rely on distributed transactions among a small number of databases, especially when regulatory obligations demand strong internal consistency and when higher availability can be achieved through redundancy rather than by relaxing consistency (Mohan et al., 1986; Gray, 1981; Hebbar, 2025).

Comparative Insights from Banking APIs and Microservice Literature

Hebbar's comparison of Saga and 2PC in banking APIs brings these theoretical and architectural considerations into a concrete domain (Hebbar, 2025). In his analysis, Saga-based approaches are shown to align better with the realities of distributed banking APIs that span multiple services and potentially external partners. Saga allows each participating service to own its local transactions and expose compensable operations through APIs, with the overall workflow being managed either by a central orchestrator or by participants reacting to events (Hebbar, 2025; Garcia-Molina & Salem, 1987; Richardson, 2019).

This alignment manifests in several ways. First, Saga respects service autonomy: each microservice in a banking platform can implement its own transaction boundaries, concurrency control, and durability mechanisms, exposing only those operations that form part of the Saga steps (Bucchiarone et al., 2019; Richardson, 2019; Messina et al., 2016). Second, Saga tolerates partial failures by design: if a later step in a payment or account-opening workflow fails, compensations can be executed to revert earlier steps, preserving business consistency without requiring synchronous agreement across all participants at the time of failure (Garcia-Molina & Salem, 1987; Hebbar, 2025). Third, Saga can be implemented using asynchronous messaging and event-driven communication, facilitating high availability and responsiveness even under increased loads or partial outages, in line with reactive and high-availability design principles (Boner et al., 2018; Ahluwalia & Jain, 2006; Štefanko et al., 2019).

By contrast, Hebbar finds that 2PC becomes cumbersome and fragile when applied across heterogeneous banking microservices. The need for a global coordinator, the complexity of enrolling services in distributed transactions, and the potential for blocking in the presence of network or node failures pose significant operational risks (Hebbar, 2025; Mohan et al., 1986; Uyanik & Ovatman, 2020). Furthermore, externally facing APIs cannot easily participate in internal 2PC because external services may not conform to the protocol or may reside in different administrative domains, making global atomicity infeasible (Hasselbring & Steinacker, 2017; Helland, 2007; Lungu & Nyirenda, 2024).

Lungu and Nyirenda's literature review corroborates these findings from a broader perspective, noting that most microservice-based systems avoid traditional distributed transactions and instead adopt Saga and related patterns, often combining them with CQRS and event sourcing to handle read and write workloads differently and to provide traceability for compensations (Lungu & Nyirenda, 2024; Fowler, 2018; Richardson, 2019). They emphasize that while Saga introduces its own challenges, particularly in designing correct compensations and ensuring idempotency, it has become the de facto standard for distributed transactions in microservice architectures across domains, including finance (Lungu & Nyirenda, 2024).

The microservice pattern literature thus positions Saga as a more natural fit for cross-service workflows, while leaving room for 2PC to continue serving intra-service or intra-cluster coordination needs where architecture permits and demands strong atomicity (Richardson, 2019; Bucchiarone et al., 2019; Messina et al., 2016; Hebbar, 2025).

DISCUSSION

Theoretical Interpretation of Saga versus 2PC in the Light of CAP and ACID

From a theoretical standpoint, the choice between Saga and 2PC can be understood as a choice between different positions in the design space defined by ACID properties and the CAP theorem. Two-phase commit is tightly aligned with ACID's atomicity and consistency requirements, ensuring that distributed transactions appear as single, indivisible units across multiple databases (Gray, 1981; Haerder & Reuter, 1983; Mohan et al., 1986). It does so by enforcing synchronous coordination among participants, making it a consistency-favoring protocol under the CAP perspective (Gilbert & Lynch, 2012).

However, CAP implies that in the presence of partitions, systems cannot simultaneously provide strong consistency and availability; they must choose whether to remain available during partitions or to preserve consistency by rejecting or blocking operations (Gilbert & Lynch, 2012). Two-phase commit, by design, rejects or delays operations when necessary to preserve atomicity, leading to blocked or failed transactions in cases where participants cannot communicate or where the coordinator is unavailable (Mohan et al., 1986; Uyanik & Ovatman, 2020). In closed, controlled environments with limited partitions, such as well-managed internal clusters, this trade-off may be acceptable, especially for critical operations (Little et al., 2004; Hebbar, 2025).

Saga, by contrast, deliberately relaxes global atomicity in favor of availability and partition tolerance. Instead of requiring all participants to commit or abort synchronously, it allows sub-transactions to commit individually and relies on compensating transactions to restore business-level invariants when later steps fail (Garcia-Molina & Salem, 1987). From a CAP perspective, Saga-based systems often operate closer to the AP side: they remain available and tolerant of partitions by allowing local progress and deferring global consistency, provided that compensations and eventual convergence are correctly handled (Gilbert & Lynch, 2012; Helland, 2007).

This does not mean that Saga forgoes consistency entirely; rather, it shifts from strict, immediate consistency to eventual consistency with domain-specific semantics (Garcia-Molina & Salem, 1987; Daraghmi et al., 2022). For many banking operations, especially those involving user-visible artifacts such as transaction histories and balances, eventual consistency is acceptable as long as invariants such as "no money is created or destroyed" and "balances eventually reflect all committed operations" are maintained and discrepancies are resolved within reasonable time frames (Helland, 2007; Hebbar, 2025).

The theoretical implication is that Saga and 2PC embody different philosophies of correctness. Two-phase commit offers a straightforward, system-level notion of atomicity, whereas Saga provides a more nuanced, application-level notion of consistency grounded in compensation semantics. Choosing between them requires careful consideration of which invariants must be strictly enforced at the system boundary and which can be managed through business processes and compensation (Gray, 1981; Garcia-Molina & Salem, 1987; Helland, 2007; Hebbar, 2025).

Architectural Implications for Banking APIs

Architecturally, Saga and 2PC lead to different shapes of banking systems. When 2PC is used across services, the system tends to centralize transactional control in a coordinator or transaction manager that interfaces with all participating systems (Mohan et al., 1986; Little et al., 2004). This centralization reduces some complexity for application developers, who can rely on the transaction manager to enforce atomicity, but it introduces tight coupling between services and the coordinator, making it harder to evolve service boundaries, adopt new persistence technologies, or integrate external services (Bucchiarone et al., 2019; Hasselbring & Steinacker, 2017).

In banking APIs that span multiple internal and external services, such tight coupling conflicts with the microservices principle of independent deployment and heterogeneous technology stacks (Bucchiarone et al., 2019; Richardson, 2019; Salah et al., 2016). It also conflicts with high-availability patterns that emphasize redundancy and fault isolation; if the coordinator or a critical participant fails, the entire system's ability to process cross-service operations may be compromised (Ahluwalia & Jain, 2006; Gilbert & Lynch, 2012; Uyanik & Ovatman, 2020).

Saga-based architectures, on the other hand, distribute responsibility for transactional behavior across services. Each microservice in a banking system exposes operations that form part of Saga steps, together with compensations; orchestration or choreography logic manages the progression of Sagas across services, logging Saga state and decisions (Garcia-Molina & Salem, 1987; Limón et al., 2018; Richardson, 2019). This distribution aligns with microservice autonomy and

enables services to evolve their internal implementation without breaking the overall workflow, as long as the contract of step and compensation operations remains stable (Bucchiarone et al., 2019; Messina et al., 2016).

Hebbar describes how banking APIs can implement operations such as "transfer funds," "apply loan," or "execute payment" as Sagas that involve ledger services, risk assessment services, notification services, and external payment networks, each contributing a step and a compensation (Hebbar, 2025). For example, if a payment fails after funds have been reserved in the account ledger, a compensating transaction may release the reservation; if a notification fails, the system may log a follow-up task or attempt alternative communication channels (Hebbar, 2025).

This architecture enables high availability: if one service is temporarily unavailable, the Saga can use retries, timeouts, or fallback paths, and compensations can be executed to keep business invariants intact without requiring synchronous agreement from all services at the moment of failure (Štefanko et al., 2019; Boner et al., 2018; Ahluwalia & Jain, 2006). It also facilitates integration with external services that cannot participate in 2PC; instead, they can be treated as Saga steps with compensations that attempt to reverse or mitigate external effects, acknowledging that perfect rollback may not always be possible (Helland, 2007; Helland & Campbell, 2009; Daraghmi et al., 2022).

At the same time, Saga-based architectures require careful design of compensations. In banking, some operations are not easily reversible, or reversals may have non-trivial financial and regulatory implications. For instance, reversing a completed funds transfer may require consent from the recipient or may be legally impossible after a certain point; in such cases, compensations might need to be modeled as separate business operations, such as issuing a refund or a chargeback, which carry their own risks and processes (Garcia-Molina & Salem, 1987; Daraghmi et al., 2022; Hebbar, 2025).

This points to an important architectural insight: Saga shifts the burden of correctness from a protocol-level guarantee to domain modeling. Where 2PC hides complexity behind strong atomicity, Saga makes this complexity explicit and requires domain experts and architects to reason about what it means to "undo" an operation in a specific business context. In banking, this can be an advantage, as it aligns transaction management with legal and business processes, but it also demands greater discipline and collaboration between technical and business stakeholders (Helland, 2007; Lungu & Nyirenda, 2024; Hebbar, 2025).

Operational and Organizational Considerations

Beyond theoretical and architectural concerns, practical adoption of Saga or 2PC in banking APIs is strongly influenced by operational and organizational factors. Two-phase commit centralizes a critical function in the transaction coordinator, which must be highly available, secure, and carefully monitored (Mohan et al., 1986; Little et al., 2004). Operational errors or misconfigurations in the coordinator can have system-wide effects, causing widespread blocking, inconsistent states, or data loss if recovery procedures fail (Haerder & Reuter, 1983; Uyanik & Ovatman, 2020).

Saga-based systems distribute responsibility, which can reduce single points of failure but increase the complexity of monitoring, tracing, and incident management. Limón and colleagues emphasize the importance of framework support for defining, tracking, and compensating Sagas, noting that without such support, developers may implement ad hoc mechanisms that are hard to reason about and audit (Limón et al., 2018).

Štefanko and co-authors, by exploring Sagas in reactive microservices, highlight the value of reactive programming in handling the concurrency and asynchrony inherent in Saga workflows, but they also point out that complexity can grow as Sagas interact with other reactive streams (Štefanko et al., 2019). Operators must be able to understand the end-to-end flow of Sagas across services, observe where failures occur, and trigger manual or automated remediation when compensations fail or behave unexpectedly.

Lungu and Nyirenda's review notes that tooling and observability are ongoing challenges for distributed transaction management in microservices; they advocate for systematic logging, correlation identifiers, distributed tracing, and dashboarding that makes Sagas visible to operators (Lungu & Nyirenda, 2024). In banking, this need is even more acute because regulatory compliance often requires detailed audit trails for financial transactions, including who initiated them, what steps were executed, and how failures were resolved (Hebbar, 2025).

Organizationally, Saga and 2PC embody different models of responsibility and coordination. Two-phase commit centralizes coordination in a technical component that may be owned by a specific infrastructure or database team, whereas Saga diffuses coordination across service teams that must collectively define and maintain Saga steps and compensations (Bucchiarone et al., 2019; Richardson, 2019; Lungu & Nyirenda, 2024). This diffusion matches the socio-technical structure of microservice organizations, where teams own services end-to-end, but it also requires strong cross-team collaboration and governance to ensure that Sagas are implemented consistently and safely, particularly when business rules change (Hasselbring &

Steinacker, 2017; Salah et al., 2016; Hebbar, 2025).

Limitations of the Current Evidence and the Need for Further Work

The evidence base synthesized in this article, while rich and multi-faceted, has limitations. Many of the contributions are conceptual, design-oriented, or based on specific case studies rather than large-scale, comparative experiments that systematically measure performance, availability, and correctness properties of Saga and 2PC across diverse banking workloads (Bucchiarone et al., 2019; Hasselbring & Steinacker, 2017; Hebbar, 2025; Lungu & Nyirenda, 2024). While experimental systems like Calvin demonstrate that distributed transactions can be made fast in specific architectures, they do not directly answer questions about integrating such systems into heterogeneous microservice landscapes (Thomson et al., 2012).

Saga-related work, including SagaMAS, enhanced Saga variants, and reactive Saga implementations, provides frameworks and patterns but often lacks extensive empirical evaluation in production-like settings, particularly in banking environments with strict regulatory requirements and complex failure modes (Limón et al., 2018; Daraghmi et al., 2022; Štefanko et al., 2019; Hebbar, 2025).

Moreover, the references include relatively little on developer productivity and error rates under different transaction management paradigms. While Helland's essays and anecdotal evidence suggest that distributed transactions and complex compensations can be hard to reason about, systematic measurements of cognitive load, defect density, and maintenance burden are missing (Helland, 2007; Helland & Campbell, 2009; Lungu & Nyirenda, 2024).

These limitations point to a clear agenda for further research. Empirical studies that compare Saga-based and 2PC-based designs for representative banking workflows, under realistic workloads and failure conditions, would significantly enrich the evidence base and allow for more precise guidance (Hebbar, 2025; Lungu & Nyirenda, 2024). Similarly, human-centered studies investigating how architects and developers understand and work with Saga and 2PC could inform tooling, education, and pattern documentation.

Future Scope: Hybrid Strategies and Advanced Frameworks

Looking ahead, it is unlikely that either Saga or 2PC will completely dominate the design of banking APIs. Instead, hybrid strategies that combine the strengths of each approach, and that exploit advanced frameworks and patterns, are likely to emerge as the most pragmatic path.

One natural hybrid is to use 2PC or similar atomic commit protocols within narrowly scoped, tightly controlled domains—such as within a single account ledger cluster or a core treasury system—while using Saga to coordinate cross-domain workflows that involve multiple services and external parties (Mohan et al., 1986; Richardson, 2019; Hebbar, 2025). This hybrid respects the autonomy of services and the realities of distributed systems while still leveraging strong atomicity where it is most valuable and feasible.

Another promising direction is the combination of Saga with CQRS and event sourcing, as suggested by Fowler and Richardson (Fowler, 2018; Richardson, 2019). In such designs, write operations form Sagas that emit events captured in append-only logs, while read models are built by projecting events into query-optimized views. This separation can simplify compensations and improve auditability—both crucial in banking—and align with reactive principles by enabling streaming updates to read models (Boner et al., 2018; Štefanko et al., 2019).

Frameworks like SagaMAS and enhanced Saga patterns proposed by Daraghmi and co-authors indicate that there is room for more sophisticated middleware that makes Saga semantics easier to express, verify, and monitor (Limón et al., 2018; Daraghmi et al., 2022). For banking, such frameworks would need to integrate with existing core banking systems, identity and access management solutions, and regulatory reporting tools, providing strong guarantees about audit trails, access control, and separation of duties.

Finally, as banking systems continue to embrace reactive architectures and message-driven communication, Saga implementations that leverage reactive streams and backpressure-aware messaging will become increasingly important (Boner et al., 2018; Štefanko et al., 2019). These implementations must address not only correctness and availability but also performance under high load, which may require integration with deterministic transaction ordering or sharding techniques in the spirit of Calvin, adapted to microservice contexts (Thomson et al., 2012; Bucchiarone et al., 2019).

CONCLUSION

The transition of banking systems toward microservices and API-centric architectures has reopened fundamental questions about how to manage distributed transactions across heterogeneous, failure-prone services. This article has provided a deeply elaborated, theory-informed comparison of the Saga pattern and two-phase commit within this context, drawing on foundational transaction and distributed-systems research, microservice and reactive architecture literature, Saga and 2PC refinements, and banking-specific analyses (Gray, 1981; Haerder & Reuter, 1983; Mohan et al., 1986;

Bucchiarone et al., 2019; Garcia-Molina & Salem, 1987; Hebbar, 2025; Lungu & Nyirenda, 2024).

Two-phase commit offers strong, intuitive atomicity guarantees, making it attractive for tightly controlled domains where all participants are under unified administration, network partitions are rare, and blocking is an acceptable trade-off for consistency (Mohan et al., 1986; Little et al., 2004). Yet its reliance on synchronous coordination and a central coordinator renders it ill-suited to the open, loosely coupled, and evolving landscapes typical of modern banking APIs and microservices (Gilbert & Lynch, 2012; Helland, 2007; Bucchiarone et al., 2019).

Saga-based approaches, by decomposing global transactions into sequences of local transactions with compensations, align more naturally with microservice autonomy, high availability, and reactive, message-driven architectures (Garcia-Molina & Salem, 1987; Boner et al., 2018; Richardson, 2019; Štefanko et al., 2019). In banking, Saga enables robust workflows across internal and external services, tolerating partial failures and allowing systems to remain responsive under load, provided that compensations are carefully designed to reflect domain semantics and regulatory constraints (Hebbar, 2025; Daraghmi et al., 2022; Lungu & Nyirenda, 2024).

The theoretical lens of CAP clarifies that Saga and 2PC occupy different positions in the space of consistency and availability, and that microservice architectures and open banking ecosystems often favor the trade-offs embodied by Saga (Gilbert & Lynch, 2012; Helland, 2007). However, Saga is not a panacea; it shifts complexity into domain modeling, compensation design, and operational observability, demanding strong collaboration between technical and business stakeholders and robust framework and tooling support (Limón et al., 2018; Lungu & Nyirenda, 2024).

The synthesis suggests that practitioners should adopt a nuanced, portfolio-based approach: using Saga as the default strategy for cross-service banking workflows that span microservices and external parties, while reserving 2PC or similar atomic protocols for narrowly scoped internal domains where their assumptions hold and their strengths can be fully exploited (Mohan et al., 1986; Richardson, 2019; Hebbar, 2025). Future work should focus on empirical evaluations of Saga and 2PC in realistic banking scenarios, on advanced Saga frameworks that integrate with core banking infrastructure, and on human-centered studies of how developers and operators work with these patterns in practice (Lungu & Nyirenda, 2024; Hebbar, 2025).

In conclusion, the choice between Saga and 2PC for banking APIs is not merely a technical one; it is a reflection of deeper decisions about how to distribute responsibility, how to align system behavior with business processes and regulatory obligations, and how to embrace the realities of modern distributed systems. By grounding these choices in the rich body of research and practice synthesized here, banking architects and researchers can make more informed, resilient, and future-proof decisions in the design of transactional infrastructures.

REFERENCES

1. Ahluwalia, K. S., & Jain, A. (2006). High availability design patterns. In Proceedings of the 2006 Conference on Pattern Languages of Programs (PLoP ’06) (p. 1). ACM Press.
2. Boner, J., Farley, D., Kuhn, R., & Thompson, M. (2018). The Reactive Manifesto. Retrieved from https://www.reactivemanifesto.org
3. Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., & Sadovykh, A. (2019). Microservices: Science and Engineering. Springer International Publishing.
4. Campbell, R. H., & Richards, P. G. (1981). SAGA: A system to automate the management of software production. In Proceedings of the May 4–7, 1981, National Computer Conference (AFIPS ’81) (p. 231). ACM Press.
5. Daraghmi, E., Zhang, C. P., & Yuan, S. M. (2022). Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences, 12(12), 6242.
6. Garcia-Molina, H., & Salem, K. (1987). Sagas. ACM SIGMOD Record, 16(3), 249–259.
7. Gilbert, S., & Lynch, N. (2012). Perspectives on the CAP theorem. Computer, 45(2), 30–36.
8. Gray, J. (1981). The transaction concept: Virtues and limitations. In Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB ’81) (pp. 144–154). VLDB Endowment.
9. Haerder, T., & Reuter, A. (1983). Principles of transaction-oriented database recovery. ACM Computing Surveys, 15(4), 287–317.
10. Hasselbring, W., & Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW) (pp. 243–246). IEEE.
11. Hebbar, K. S. (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

12. Helland, P. (2007). Life beyond distributed transactions: An apostate's opinion. In CIDR 2007 (pp. 132–141).

13. Helland, P., & Campbell, D. (2009). Building on quicksand. CoRR, abs/0909.1788.

14. Limón, X., Guerra-Hernández, A., Sánchez-García, A. J., & Arriaga, J. C. P. (2018). SagaMAS: A software framework for distributed transactions in the microservice architecture. In 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT) (pp. 50–58). IEEE.

15. Little, M., Maron, J., & Pavlik, G. (2004). Java Transaction Processing. Prentice Hall.

16. Lungu, S., & Nyirenda, M. (2024). Current trends in the management of distributed transactions in micro-services architectures: A systematic literature review. Open Journal of Applied Sciences, 14(9), 2519–2543.

17. Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., & Urso, A. (2016). The Database-is-the-Service pattern for microservice architectures. In M. E. Renda, M. Bursa, A. Holzinger, & S. Khuri (Eds.), Information Technology in Bio- and Medical Informatics (Vol. 9832, pp. 223–233). Springer International Publishing.

18. Mohan, C., Lindsay, B., & Obermarck, R. (1986). Transaction management in the R* distributed database management system. ACM Transactions on Database Systems, 11, 378–396.

19. Richardson, C. (2019). Microservices Patterns: With Examples in Java. Manning Publications.

20. Salah, T., Zemerly, M. J., Yeun, C. Y., AlQutayri, M., & Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST) (pp. 318–325). IEEE.

21. Štefanko, M., Chaloupka, O., & Rossi, B. (2019). The saga pattern in a reactive microservices environment. In Proceedings of the 14th International Conference on Software Technologies (pp. 483–490). SCITEPRESS.

22. Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., & Abadi, D. J. (2012). Calvin: Fast distributed transactions for partitioned database systems. In Proceedings of the 2012 International Conference on Management of Data (SIGMOD '12) (p. 1). ACM Press.

23. Uyanik, H., & Ovatman, T. (2020). Enhancing two-phase commit protocol for replicated state machines. In Proceedings of the 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (pp. 118–121). IEEE.

24. Bucchiarone, A., Dragoni, N., Dustdar, S., Lago, P., Mazzara, M., Rivera, V., & Sadovykh, A. (2019). Microservices: Science and Engineering. Springer International Publishing.

25. Boner, J., Farley, D., Kuhn, R., & Thompson, M. (2018). Reactive manifesto. https://www.reactivemanifesto.org

26. Fowler, M. (2018). CQRS. https://martinfowler.com/bliki/CQRS.html