Frontiers in Emerging Multidisciplinary Sciences (Open Access)

Volume 02, Issue 08, August 2025,
Publish Date: 12-08-2025
PageNo.06-11

Optimizing Distributed Transaction Strategies for Microservice-Based Banking APIs: A Comparative Theoretical and Practical Analysis of Saga and Two-Phase Commit Patterns

Rajiv K. Deshmukh
Global Institute of Distributed Systems, United Kingdom

ABSTRACT

Background: Modern banking platforms increasingly adopt microservice architectures and RESTful APIs to achieve scalability, agility, and independent deployability, yet they confront fundamental challenges in preserving transactional integrity across polyglot services and distributed data stores. Two prominent approaches—coordinated, blocking protocols typified by Two-Phase Commit (2PC), and decentralized, compensation-based workflows typified by the Saga pattern—offer divergent trade-offs in consistency, availability, latency, and operational complexity. This paper synthesizes the theoretical foundations and practical implications of these strategies specifically for banking API ecosystems, integrating perspectives from transaction theory, distributed systems design, and contemporary microservice engineering.

Methods: Relying exclusively on the provided literature, this research conducts a rigorous conceptual synthesis and comparative analysis. We reconstruct the problem space by aligning classical transaction theory with microservice characteristics, elucidate the mechanics and failure modes of 2PC and Saga workflows, and propose a decision framework and hybrid design patterns for banking scenarios. Our methodological approach foregrounds architectural constraints found in RESTful API design, event-driven orchestration, and multi-database deployment common in banking domains.

Results: The analysis reveals that while 2PC provides strong atomicity guarantees suitable for tightly coupled systems and guaranteed commit semantics, it imposes coordination latency, resource locking, and limited fault-tolerance at scale (Haerder & Reuter, 1983; Fan et al., 2020). Conversely, Saga affords greater availability and resilience in loosely coupled microservices and supports eventual business consistency through compensating transactions, yet it demands rigorous compensation design, complex recovery logic, and can produce transient anomalies that must be addressed at the domain level (Christudas & Christudas, 2019; Kleppmann, 2017). Hybrid approaches—combining local 2PC within bounded contexts, Saga orchestration across services, and compensation-aware idempotency controls—emerge as pragmatic for banking APIs that require both strict monetary correctness and scalable operations (Hebbar, 2025; Zhang et al., 2019).

Discussion: We provide an extended theoretical elaboration on failure semantics, concurrency control, idempotence, and observability necessary for operationalizing either pattern, explore counter-arguments regarding correctness vs. performance trade-offs, and discuss limitations of each approach. We synthesize a prescriptive, textual methodology for architects: classifying transactions by criticality, mapping failure and latency budgets, and selecting patterns or hybrids accordingly. Proposed future research directions include formal verification of compensation logic, adaptive middleware for mixed-mode transactions, and empirical validation in production banking environments.

Conclusion: No single pattern universally dominates; the optimal choice for banking APIs depends on explicit domain requirements, tolerance for eventual consistency, and operational maturity. Banks benefit from adopting an evidence-based hybrid strategy: local strong coordination where monetary correctness cannot be compromised, distributed Saga orchestration for cross-cutting business flows, and enhanced tooling for monitoring and recovery. This synthesis offers architects an in-depth conceptual toolkit and stepwise design guidelines informed by foundational and contemporary research.

KEYWORDS: Distributed transactions, microservices, Saga pattern, two-phase commit, banking APIs, compensation, transactional consistency.

INTRODUCTION

The emergence of microservice architectures has precipitated a fundamental re-evaluation of how transactional integrity is achieved in distributed systems. Historically, transactional semantics in monolithic systems

rested on centralized database management systems capable of providing ACID (Atomicity, Consistency, Isolation, Durability) guarantees (Haerder & Reuter, 1983). Banking systems—by virtue of handling monetary transfers, balances, and regulatory compliance—demand correctness and robust recovery semantics. As enterprises have deconstructed monoliths into independently deployable services, each owning its own data store, the classical centralized model fractures: a single global transaction that spans multiple services and databases becomes impractical and, in many cases, impossible without introducing operational brittleness and performance penalties (Newman, 2021; Kleppmann, 2017).

This fragmentation raises a core design question for banking API architects: how should distributed transactions be implemented so that they harmonize the business requirement for monetary correctness with the architectural goals of scalability and resilience intrinsic to microservices and RESTful services (Fielding, 2000; Lewis & Fowler, 2014)? The literature offers two broad families of approaches. First, coordinated commit protocols—exemplified by Two-Phase Commit (2PC) and its variants—attempt to preserve ACID-like semantics through a centralized coordinator that orchestrates the commit or abort of participants (Haerder & Reuter, 1983; Fan et al., 2020). Second, compensation-based designs—exemplified by the Saga pattern—eschew global locks and instead decompose a global transaction into sequences of local transactions interleaved with compensating transactions to restore invariants when needed (Christudas & Christudas, 2019; Newman, 2021).

While general principles are well-studied, the banking domain presents unique constraints: regulatory visibility requirements, strict correctness for monetary transfers, real-time responses for customer-facing APIs, and interactions with legacy backend systems and external clearing networks. These constraints create a rich design space in which the trade-offs between 2PC and Saga become nuanced and context-dependent (Hebbar, 2025; Navarro, 2022). This paper aims to produce an exhaustive, publication-quality synthesis of these patterns, including their theoretical underpinnings, operational behaviors, failure modes, and practical prescriptions tailored for banking APIs. The work fills a literature gap by integrating transaction theory, recent microservice research, and domain-specific banking constraints into a unified decision framework, accompanied by detailed methodological guidance for architects.

METHODOLOGY

This research adopts a conceptual-analytic methodology rooted in an exhaustive reading and critical synthesis of the provided references. The objective is to derive logically consistent conclusions and prescriptive design guidance without introducing empirical data beyond what the cited works supply. The methodology unfolds in the following stages: (1) conceptual alignment—mapping classical transaction theory to microservice realities; (2) pattern deconstruction—detailing the mechanisms, guarantees, and failure semantics of 2PC and Saga; (3) scenario-driven analysis—applying the deconstructed patterns to canonical banking API workflows to elucidate trade-offs; (4) hybridization and decision framework—proposing combinational strategies and selection criteria; and (5) robustness considerations—discussing idempotency, concurrency control, monitoring, and compensation design.

Conceptual alignment synthesizes insights from classical database recovery and transaction principles (Haerder & Reuter, 1983), modern data-intensive system thinking (Kleppmann, 2017), and RESTful architectural constraints (Fielding, 2000). Pattern deconstruction builds on domain-specific microservices research, which analyzes distributed transaction strategies and introduces new protocol variants and frameworks (Bashtovyi & Fechan, 2024; Fan et al., 2020; Zhang et al., 2019). Scenario-driven analysis draws on applied research in banking and enterprise transaction platforms to model transfer operations, balance updates, and reserve accounting across microservices (Hebbar, 2025; Navarro, 2022; González-Aparicio et al., 2023). The decision framework combines theoretical metrics (consistency models, latency, availability) with practical operational constraints (observability, rollback windows, compensatability) to produce prescriptive guidance (Godage et al., 2023; Nylund, 2023).

Throughout the analysis, every major claim is explicitly supported by the literature. Where the literature presents competing viewpoints, both are discussed and evaluated. Importantly, the methodology intentionally avoids empirical simulation and instead focuses on theoretical articulation and design rationale—a choice appropriate given the instruction to base content strictly on the provided references.

RESULTS

This section presents a descriptive analysis of the principal findings of the conceptual synthesis: a deep articulation of the behaviors, trade-offs, and pragmatic considerations associated with 2PC and Saga when applied to banking APIs. The results are organized into several thematic subsections: transactional guarantees and semantics; latency and scalability; failure modes and recovery; operational complexity and maintainability; and hybrid strategies.

Transactional Guarantees and Semantics

Two-Phase Commit (2PC) seeks to provide atomic commit semantics across multiple participants by employing a coordinator which first asks participants to prepare and then to commit or abort. When all participants respond positively in the prepare phase, the coordinator issues a commit; otherwise, it instructs abort. The principal advantage is preservation of atomicity: either all

participants commit or none do, thus maintaining strong consistency across multiple datastores (Haerder & Reuter, 1983; Fan et al., 2020). In banking contexts, this can theoretically guarantee the preservation of critical monetary invariants—no net creation or loss of funds across services—provided proper isolation and serializability are enforced locally.

The Saga pattern, by contrast, models a long-running transaction as a sequence of local transactions where each local transaction publishes an event or effect. If a subsequent step fails, the system executes compensating transactions to revert the effects of preceding steps, thereby attempting to restore a consistent state at the business level (Christudas & Christudas, 2019; Newman, 2021). Sagas shift the burden from centralized coordination to distributed orchestration or choreography, favoring availability and resilience but delivering eventual consistency rather than strict atomicity.

Applying these semantics to banking APIs yields clear distinctions. For atomic monetary transfers involving ledger updates across multiple bounded contexts (e.g., account service, fraud service, ledger service), 2PC appears to map directly to the atomicity requirement—if a coordinated commit can be achieved, the system ensures invariant preservation. However, 2PC's reliance on blocking locks and global coordination creates availability and latency issues, particularly under partial failures or network partitions (Haerder & Reuter, 1983; Fan et al., 2020). Saga approaches can achieve high availability and responsiveness but require careful compensation logic and domain-level reconciliation to ensure that the eventual state is correct and that transient inconsistencies are tolerable given business rules (Christudas & Christudas, 2019; Kleppmann, 2017).

Latency and Scalability

Coordination in 2PC introduces round-trip communication and often requires participants to hold resources in a prepared state until the commit/abort decision is made, resulting in increased latency and reduced throughput as the number of participants grows (Fan et al., 2020). For banking APIs that must respond to customer requests within strict latency budgets, this can degrade user experience and jeopardize SLAs (Navarro, 2022). Furthermore, under high concurrency, the global locking behavior of 2PC can create contention and cascading delays.

Saga achieves better scalability by avoiding global locks: local transactions commit independently, and compensations are invoked if necessary. This non-blocking behavior means that services remain responsive and can scale horizontally. However, Sagas may introduce extended periods of eventual consistency during which business views diverge, necessitating compensating activities or reconciliation jobs. API-level design must therefore expose appropriate semantics to clients—e.g., indicating "pending" states for transfers—so consumers understand that finality may be delayed (Christudas & Christudas, 2019; Newman, 2021).

Failure Modes and Recovery Semantics

2PC exhibits well-characterized failure modes: coordinator failure during commit/abort decision points can leave participants in uncertain (prepared) states, necessitating manual or automated recovery mechanisms that rely on persistent logs and timeouts (Haerder & Reuter, 1983). In banking systems, unresolved prepared states can lock accounts, freeze funds, and trigger customer-facing outages if not resolved. Moreover, network partitions exacerbate these problems since participants cannot reach the coordinator, and the blocking nature of 2PC may prolong outages.

Saga's failure modes commonly relate to compensating transaction design failures, partial application of side effects (e.g., notification sent before compensation), and difficulty in guaranteeing global invariants during error windows. Compensation logic may be non-trivial, especially for operations that are not naturally reversible (e.g., sending notifications, interacting with external settlement networks). Moreover, race conditions can produce anomalies where compensations are applied concurrently with subsequent operations. Robust Saga implementations thus require idempotent operations, careful ordering, and compensating semantics designed from the outset (Christudas & Christudas, 2019; Kleppmann, 2017; Zhang et al., 2019).

Operational Complexity and Maintainability

2PC centralizes complexity within the coordinator and participant protocol implementations, which can simplify reasonability but increases operational burden for database and infrastructure engineers. The need to manage coordinator failover, persistent logs, and participant timeouts introduces additional operational complexity (Fan et al., 2020).

Saga shifts complexity to the application domain: business logic must include compensating transactions, and orchestration or choreography engines must handle sequencing, retries, and partial failures. From a maintainability standpoint, code becomes entangled with compensation semantics, demanding rigorous testing, simulation, and documentation. However, the microservice model—where teams own services and their data—aligns well with Saga's decomposition, allowing team-level autonomy and localized reasoning about failure handling (Lewis & Fowler, 2014; Newman, 2021).

Hybrid Strategies and Pattern Selection

The literature and domain analysis suggest that hybrid strategies can combine the strengths of both approaches. For example, within a bounded context where multiple operations touch a tightly coupled subset of data, localized 2PC or even database-level transactions may be acceptable and less risky since participant count is small and latency

implications contained. Across bounded contexts, Saga orchestration can manage long-running, cross-service workflows. Further hybridization includes using compensation for business-level rollback while employing lightweight consensus protocols for critical primitive state (Zhang et al., 2019; Hebbar, 2025). The design of such hybrids should be guided by transaction criticality classification, latency budgets, and recovery windows.

DISCUSSION

This section elaborates interpretively on the implications of the results, discusses limitations, and articulates detailed practical recommendations and future research directions. The discussion traverses theoretical foundations, counterarguments, and prescriptive design fluency for architects of banking APIs.

Theoretical Interpretations and Trade-offs

At a theoretical level, the trade-off between 2PC and Saga is an instantiation of the broader CAP and FLP-like tensions in distributed computing: coordination enables stronger consistency but at the cost of availability and performance; decentralization trades immediate consistency for availability and partition tolerance (Kleppmann, 2017). In banking, where correctness is non-negotiable, the temptation is to favor coordination protocols. However, practical systems cannot ignore performance and resilience; customer expectations and regulatory requirements demand both correctness and high availability. Thus, an exclusive reliance on 2PC in a microservice architecture may be impractical.

An important nuance arises in distinguishing correctness at the technical level (e.g., atomic commit) from correctness at the business level (e.g., maintaining ledger invariants and regulatory compliance). Saga, when implemented with well-designed compensations, can preserve business correctness even without providing strict atomicity at the storage level. Compensations can be designed to ensure that, after recovery, the system satisfies its business invariants, albeit with transient states. This reframing suggests that transactional semantics should be recast in terms of business invariants amenable to compensation-based recovery, not solely database-level atomicity (Christudas & Christudas, 2019; Kleppmann, 2017).

Counter-Arguments and Rebuttals

A common counter-argument is that Saga's eventual consistency is fundamentally inadequate for banking due to the potential for transient double-spend or inconsistent balance presentation. Proponents of this view argue that only strong atomicity prevents such anomalies. In response, several points mitigate this concern. First, many banking operations can be decomposed into critically atomic primitives and less-critical follow-on steps; atomicity can be enforced for the primitives (via localized strong transactions), while longer workflows are managed as Sagas. Second, engineering practices—such as provisional holds, reservation semantics, and compensations—can prevent double-spend exposure to end-users. For example, a provisional hold on funds (a local atomic operation) can be used to guarantee a reserve while subsequent operations complete asynchronously—this design pattern leverages a hybrid of atomic primitives and Saga orchestration to balance correctness and responsiveness (Hebbar, 2025; Newman, 2021).

Another rebuttal posits that 2PC, when augmented with new consensus protocols or non-blocking commit variants (e.g., 2PC*), can be made performant and non-blocking. Research into protocol variants and optimized concurrency control demonstrates promising directions (Fan et al., 2020). Yet, such protocols often entail significant complexity and require specialized infrastructure support, limiting their practical adoption across polyglot microservice ecosystems. Thus, while advanced 2PC variants may reduce some drawbacks, they do not universally eliminate the fundamental cost of centralized coordination.

Design Prescriptions and Best Practices for Banking APIs

Drawing on the theoretical synthesis and scenario analysis, a set of concrete design prescriptions is offered for banking architects:

1. Classify transaction criticality: Establish clear labels for operations: (a) critical atomic primitives (e.g., ledger updates that must be atomic at the account level), (b) medium-critical workflows (e.g., cross-service reconciliation, external settlements), and (c) non-critical processes (e.g., notifications, analytics ingestion). Use strong coordination only for (a); employ Saga orchestration and idempotent compensations for (b) and (c) (Hebbar, 2025; Navarro, 2022).
2. Use local strong transactions inside bounded contexts: Preserve atomicity at the service/database boundary using local DB transactions or lightweight coordinator protocols where the participant set is small and co-located. This reduces complexity while guaranteeing primitives (Christudas & Christudas, 2019; Newman, 2021).
3. Design compensations as first-class artifacts: Compensation logic must be specified, tested, and versioned alongside forward business logic. Consider that not all operations are easily reversible—design for safe compensations by introducing reversible primitives (e.g., holds and releases) wherever possible (Christudas & Christudas, 2019; Kleppmann, 2017).
4. Adopt orchestration engines for complex workflows: Explicit orchestrators (e.g., state machines or workflow engines) centralize Saga control and simplify recovery logic compared to ad-hoc choreography. Orchestrators provide clearer failure semantics, centralized observability, and easier reasoning about long-running flows (Bashtovyi & Fechan, 2024; Toffetti et al., 2015).
5. Invest in idempotency and retry semantics: Endpoints should be idempotent and designed for safe

retries; compensating transactions should also be idempotent. Explicit idempotency keys and deduplication logic prevent double application under retries and network duplication (Godage et al., 2023; Newman, 2021).

6. Surface eventuality semantics in APIs: Banking APIs should explicitly communicate the finality semantics of operations (e.g., "tentative" vs. "final") so clients can adapt UI/UX and downstream reconciliation. This reduces confusion and supports better user experience during eventual consistency windows (Christudas & Christudas, 2019).

7. Implement robust observability and reconciliation: End-to-end tracing, audit logs, and reconciliation jobs are necessary to detect and remediate anomalies induced by distributed failure. Recovery tooling should integrate with operational runbooks to handle prepared states in 2PC or unresolved compensations in Sagas (González-Aparicio et al., 2023; Godage et al., 2023).

8. Use hybrid transactional middleware where appropriate: Emerging platforms that support polyglot transactions and reconciliation (e.g., GRIT-like systems) provide new options for combining local transactional guarantees with cross-service coordination. Evaluate these platforms carefully for integration cost and operational maturity (Zhang et al., 2019; González-Aparicio et al., 2023).

Limitations and Future Work

This research is constrained by its theoretical nature and exclusive reliance on the provided references; empirical validation in active banking environments would strengthen and quantify the trade-offs described. Future research should pursue formal verification of compensation logic and recovery protocols, develop middleware that dynamically chooses coordination strategies based on runtime conditions, and carry out controlled experiments comparing throughput, latency, and correctness incidents under both patterns across realistic bank-like workloads. Additionally, development of domain-specific languages for compensation specification and model checking would materially reduce the risk inherent in Saga design.

CONCLUSION

This paper has provided an exhaustive theoretical analysis and prescriptive synthesis of distributed transaction strategies—Two-Phase Commit and Saga—applied to microservice-based banking APIs. Both patterns present compelling advantages and salient drawbacks: 2PC's strong atomicity versus Saga's scalability and resilience. Banking architectures should not adopt either pattern wholesale but instead apply a nuanced hybrid approach: enforcing atomic primitives locally, orchestrating cross-service workflows via Saga or orchestrators, and employing carefully designed compensations and observability practices to ensure business correctness.

The decision framework and practical prescriptions distilled herein aim to empower architects to make design choices that reflect the nuanced risk, performance, and operational constraints endemic to banking. Furthermore, the field would benefit from tooling and formal methods that reduce the difficulty of specifying and verifying compensating transactions, alongside empirical studies that quantify operational trade-offs in live banking systems. As banks evolve toward increasingly decentralized architectures, such research and tooling will be critical to preserving monetary correctness while delivering scalable, resilient services to customers.

REFERENCES

1. Bashtovyi, A., & Fechan, A. (2024). DISTRIBUTED TRANSACTIONS IN MICROSERVICE ARCHITECTURE: INFORMED DECISION-MAKING STRATEGIES.
2. Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015, April). An architecture for self-managing microservices. In Proceedings of the 1st international workshop on automated incident management in cloud (pp. 19-24).
3. Yadav, P. S. DESIGN AND EVALUATION OF EVENT-DRIVEN ARCHITECTURES FOR TRANSACTION MANAGEMENT IN MICROSERVICES.
4. Navarro, A. (2022). Fundamentals of Transaction Management in Enterprise Application Architectures. IEEE Access, 10, 124305-124332.
5. Nylund, W. (2023). Comparing Transaction Management Methods in Microservice Architecture.
6. Zhang, G., Ren, K., Ahn, J. S., & BenRomdhane, S. (2019, April). GRIT: consistent distributed transactions across polyglot microservices with multiple databases. In 2019 IEEE 35th International Conference on Data Engineering (ICDE) (pp. 2024-2027). IEEE.
7. González-Aparicio, M. T., Younas, M., Tuya, J., & Casado, R. (2023). A transaction platform for microservices-based big data systems. Simulation Modelling Practice and Theory, 123, 102709.
8. Godage, S., Kumar, T. R., Pandya, H., Bhosale, S., & Patil, R. (2023). Web Interface for Distributed Transaction System. Computer Integrated Manufacturing Systems, 29(6), 214-227.
9. 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
10. Christudas, B., & Christudas, B. (2019). Distributed Transactions. Practical Microservices Architectural Patterns: EventBased Java Microservices with Spring Boot and Spring Cloud, 385-481.
11. Fan, P., Liu, J., Yin, W., Wang, H., Chen, X., & Sun, H. (2020). 2PC*: a distributed transaction concurrency control protocol of multimicroservice based on cloud computing platform. Journal of Cloud Computing, 9, 1-22.

12. Newman, S. (2021). Building microservices. O'Reilly Media, Inc.

13. Christudas, B., & Christudas, B. (2019). Transactions and Microservices. Practical Microservices Architectural Patterns: EventBased Java Microservices with Spring Boot and Spring Cloud, 483-541.

14. Fielding, R.T. (2000) Rest: Architectural Styles and the Design of Network-Based Software Architectures. Doctoral Dissertation, University of California.

15. Taibi, D., Lenarduzzi, V., Pahl, C. and Janes, A. (2017) Microservices in Agile Software Development. Proceedings of the XP2017 Scientific Workshops, Cologne, 22-26 May 2017, 1-5.

16. Kleppmann, M. (2017) Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems. O'Reilly Media, Inc.

17. Haerder, T. and Reuter, A. (1983) Principles of Transaction-Oriented Database Recovery. ACM Computing Surveys, 15, 287-317.

18. Carrera-Rivera, A., Ochoa, W., Larrinaga, F. and Lasa, G. (2022) How-to Conduct a Systematic Literature Review: A Quick Guide for Computer Science Research. MethodsX, 9, Article 101895.

19. Lewis, J. and Fowler, M. (2014) Microservices, a Definition of This New Architectural Term.

20. Pahl, C. and Jamshidi, P. (2016) Microservices: A Systematic Mapping Study. Proceedings of the 6th International Conference on Cloud Computing and Services Science, Rome, 23-25 April 2016, 137-146.