Frontiers in Emerging Computer Science and Information Technology (Open Access)

Volume 02, Issue 07, July 2025,
Publish Date: 31-07-2025
PageNo.13-22

Architecting Transaction-Intensive Microservices: Saga-Based Distributed Transactions, Clean Boundaries, and Reactive Data Infrastructure

Elora Handerson
Department of Computer Science, University of Milan, Milan, Italy

ABSTRACT

Background:
Microservice architectures have become the dominant paradigm for high-scale, API-driven systems such as e-commerce platforms and digital banking backends, promising independent deployment, fine-grained scalability, and organizational alignment with business domains (Nadareishvili et al., 2016; Sharma, 2017). However, as monolithic applications are decomposed and data ownership is distributed across services, transaction management becomes significantly more complex. Workflows such as online shopping checkouts or banking fund transfers, which previously relied on local ACID transactions, now span many autonomous services, databases, and infrastructure components, including message brokers and caches (Stonebraker & Cattell, 2011; Narkhede et al., 2017; Hebbar, 2025).
Objective:
This article develops a comprehensive, theoretically grounded account of how to design transaction-intensive microservice architectures that reconcile distributed workflows, domain purity, and reactive data infrastructure. It focuses on the Saga pattern and its framework implementations, such as SagaMAS, in combination with clean architectural boundaries, Enterprise Integration Patterns (EIP), and streaming technologies like Kafka, Redis, RxJava, and PostgreSQL (Garcia-Molina & Salem, 1987; Limón et al., 2018; Petrasch, 2017; Narkhede et al., 2017; Nurkiewicz & Christensen, 2016; Douglas & Douglas, 2003). Banking APIs and e-commerce shopping flows are used as primary application contexts (Shopping Process, 2022; Hebbar, 2025).
Methods:
A qualitative, literature-based methodology is applied, structured in the spirit of systematic reviews in software engineering while remaining narrative and theory-building (Keele et al., 2007; Sedeño et al., 2019). Foundational texts on microservices, architecture, and simple-operation datastores are combined with systematic reviews on microservice deployment, security, and data management to form the architectural baseline (Nadareishvili et al., 2016; Martin, 2018; Stonebraker & Cattell, 2011; Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022). Saga literature, including Hebbar's banking-focused work and SagaMAS, is then interpreted in the context of real-time data pipelines and reactive programming (Limón et al., 2018; Stefanko, 2017; Hebbar, 2025; Narkhede et al., 2017; Nurkiewicz & Christensen, 2016). Practices from meta-surveys and systematic reviews in neighboring domains such as explainable AI, DDoS detection, and knowledge graphs inform the rigor of synthesis and the structuring of research gaps (Saeed & Omlin, 2023; Mittal et al., 2022; Tiwari et al., 2021).
Results:
The synthesis shows that Saga-based approaches are structurally better aligned with microservice principles than classical distributed transactions, particularly in domains where workflows cross bounded contexts and infrastructure tiers (Nadareishvili et al., 2016; Garcia-Molina & Salem, 1987; Hebbar, 2025). Clean architecture principles, when combined with REST rulebooks and EIP-based integration, provide a way to isolate Saga orchestration from domain logic and infrastructure concerns, enhancing maintainability and testability (Martin, 2018; Mass, 2012; Petrasch, 2017; Walls, 2016). Kafka, Redis, RxJava, and PostgreSQL form a powerful yet non-trivial substrate for implementing Saga coordination, outbox and inbox patterns, and idempotent processing in transactional flows (Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016; Douglas & Douglas, 2003). However, the analysis also reveals significant challenges in security, deployment topology, multi-datastore consistency, and observability, underscoring the need for domain-sensitive patterns and framework support (Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022; Limón et al., 2018).
Conclusion:

For transaction-intensive microservice systems in banking and e-commerce, Saga-based distributed transactions, implemented via specialized frameworks and embedded in clean, domain-centric architectures, offer a robust path toward balancing consistency, scalability, and organizational autonomy. Yet this path is not purely technical; it demands disciplined modeling of compensations, cross-cutting security controls, deployment-aware data strategies, and a culture capable of reasoning about eventual consistency and asynchronous workflows (Vogels, 2009; Hebbar, 2025; Laigner et al., 2021). The article outlines a research agenda calling for deeper empirical studies on Saga frameworks, cross-datastore strategies, and explainable observability for highly distributed transactional flows.

KEYWORDS: Microservice architecture; Saga pattern; distributed transactions; reactive data pipelines; Kafka and Redis; clean architecture; banking and e-commerce APIs.

INTRODUCTION

The shift from monolithic systems to microservice architectures has reshaped how organizations conceive, build, deploy, and operate software-intensive systems, particularly those involving customer-facing APIs in domains such as e-commerce and digital banking (Nadareishvili et al., 2016; Sharma, 2017). Early monolithic applications typically relied on a single relational database and local ACID transactions to ensure consistent state transitions for operations such as “place order,” “charge credit card,” or “transfer funds.” These operations, though sometimes complex in business semantics, were technically encapsulated within a single process and a single database transaction, allowing engineers to reason about correctness largely within a narrowly bounded technical context (Stonebraker & Cattell, 2011; Douglas & Douglas, 2003).

Microservices disrupt this comfort zone. By decomposing systems into smaller, independently deployable services, microservice architectures distribute ownership of data and behavior across bounded contexts that map to business capabilities (Nadareishvili et al., 2016; Martin, 2018). A typical shopping checkout in an e-commerce platform becomes a cross-service interaction involving catalog, cart, pricing, inventory, payment, shipping, and notification services, each with its own datastore and its own operational and scaling concerns (Sharma, 2017; Shopping Process, 2022). Similarly, banking operations such as “initiate funds transfer” involve customer, accounts, limits, fraud, compliance, settlement, and statement services, potentially spanning internal and external systems (Hebbar, 2025).

In this distributed landscape, traditional transaction mechanisms struggle. Stonebraker and Cattell’s rules for scalable simple-operation datastores emphasize sharding, avoiding distributed joins, and minimizing coordination, which fundamentally conflicts with heavyweight global transactions (Stonebraker & Cattell, 2011). At the same time, the growing prevalence of geo-distributed deployments, cloud-native elastic scaling, and polyglot data stores intensifies the challenges of managing consistency across microservices (Karabey Aksakalli et al., 2021; Laigner et al., 2021).

The Saga pattern emerges as a response to these pressures. Initially formalized as a way to break long-lived transactions into a sequence of local transactions with compensating actions, Sagas fit naturally into an environment where each microservice manages its own data but participates in larger business workflows (Garcia-Molina & Salem, 1987; Stefanko, 2017; Limón et al., 2018). Sagas trade global atomicity for eventual consistency, relying on defined compensations to restore invariants when sub-steps fail. This fits the “eventually consistent” perspective championed by Vogels, which argues that distributed systems must embrace temporal windows of inconsistency while maintaining overall integrity and user satisfaction (Vogels, 2009).

Yet Saga is not merely a drop-in replacement for ACID transactions. Its use forces architects to make explicit the semantics of partial failures, compensation, and time, shifting the burden of correctness from database-level mechanisms to domain modeling and coordination logic. In financial systems, Hebbar’s comparison of Saga and two-phase commit in banking APIs illustrates how these trade-offs become particularly sharp: while Saga aligns with microservice autonomy and failure tolerance, it requires rigorous design of compensating operations and careful integration with regulatory and risk controls (Hebbar, 2025).

The surrounding ecosystem of technologies further complicates design. Microservice systems increasingly rely on message brokers and log-based streaming platforms such as Kafka to propagate domain events and coordinate workflows at scale (Narkhede et al., 2017). In-memory data stores like Redis are used as caches, queues, or transient stores for idempotency and rate limiting (Redis, 2022). Reactive libraries such as RxJava and frameworks like Spring Boot are employed to build non-blocking, event-driven services capable of handling large volumes of concurrent I/O (Nurkiewicz & Christensen, 2016; Walls, 2016). PostgreSQL continues to serve as a robust relational backend in many architectures, interacting with this reactive and streaming fabric through patterns like outbox/inbox and change data capture (Douglas & Douglas, 2003; Narkhede et al., 2017).

These ingredients must be composed within architectural styles that prioritize testability, evolvability, and clear separation of concerns. Clean Architecture positions domain models and use cases at the center, insulating them from frameworks, interface technologies, and persistence mechanisms (Martin, 2018). REST API design rulebooks and Enterprise Integration Patterns offer guidance on API design and inter-service communication, framing how Sagas and events should be exposed, consumed, and orchestrated (Mass, 2012; Petrasch, 2017).

At the same time, the microservice landscape is accompanied by a fast-growing body of systematic reviews on deployment and communication patterns, data management, and security, which collectively highlight the complexity and diversity of real-world designs (Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022). Insights from systematic methodologies in software engineering and related fields, such as explainable AI, DDoS detection, and knowledge graphs, demonstrate the value of disciplined synthesis when grappling with complex technology ecosystems (Keele et al., 2007; Sedeño et al., 2019; Saeed & Omlin, 2023; Mittal et al., 2022; Tiwari et al., 2021).

Despite this rich context, there remains a gap: a consolidated, deeply elaborated account that integrates Saga-based transactional patterns, microservice architectural principles, and reactive data infrastructure with specific attention to transaction-intensive domains like shopping and banking. This article addresses that gap by constructing a conceptual architecture and analytical framework, grounded strictly in the provided literature, that illuminates how these elements interact and what trade-offs they imply.

METHODOLOGY

The methodological approach adopted in this work is qualitative and integrative, taking inspiration from established guidelines for systematic literature reviews while intentionally orienting toward theory building and architectural synthesis (Keele et al., 2007; Sedeño et al., 2019). The process can be described along four intertwined dimensions: corpus construction, conceptual clustering, cross-domain methodological borrowing, and architectural modeling.

First, corpus construction relies strictly on the provided references, which form a heterogeneous body of sources including technical books, conference and journal articles, systematic literature reviews, blog posts, and domain-specific web resources. Foundational microservice texts by Nadareishvili and co-authors and Sharma establish the general principles of microservice architecture, domain modeling, and pragmatic implementation concerns (Nadareishvili et al., 2016; Sharma, 2017). Stonebraker and Cattell's "10 rules" provide a principled view of datastore behavior in high-throughput, simple-operation scenarios, which is crucial for reasoning about data-tier design under distributed transactional loads (Stonebraker & Cattell, 2011).

Saga-specific and transaction-oriented works form the second cornerstone of the corpus. This includes the original Saga formulation and subsequent implementation comparisons, the SagaMAS framework, and Hebbar's analysis of Saga versus two-phase commit in banking APIs (Garcia-Molina & Salem, 1987; Stefanko, 2017; Limón et al., 2018; Hebbar, 2025). The inclusion of SAGA BigJob contributes insight into generalizable abstractions for distributed jobs, which, though primarily targeted at high-performance computing scenarios, share conceptual ground with distributed orchestration patterns (Luckow et al., 2010).

The third pillar encompasses architectural and integration-focused sources. Clean Architecture and Spring Boot in Action inform structural and implementation choices at the service level (Martin, 2018; Walls, 2016). REST API Design Rulebook provides constraints and patterns for resource-oriented interfaces, status codes, and hypermedia considerations that are relevant for Saga orchestration endpoints (Mass, 2012). Petrasch's work on model-based engineering for microservices using Enterprise Integration Patterns (EIP) contributes a vocabulary and set of abstractions for asynchronous messaging, routing, aggregation, and error handling between services (Petrasch, 2017).

The fourth pillar addresses microservice deployment, communication, data management, and security, derived from systematic reviews by Karabey Aksakalli and colleagues, Laigner and co-authors, and Berardi and collaborators (Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022). These works offer structured mappings of the design spaces that distributed transactional architectures must inhabit.

A fifth component relates to methodologies and meta-level studies. Guidelines for systematic literature reviews in software engineering provide a template for search, inclusion, and synthesis strategies (Keele et al., 2007). Sedeño and colleagues' review of systematic service discovery, Saeed and Omlin's meta-survey on explainable AI, Mittal and co-authors' review of deep learning for DDoS detection, and Tiwari and collaborators' survey on knowledge graphs are used not for their content domains but for their methodological structures, especially in how they categorize solutions, map research gaps, and articulate future work (Sedeño et al., 2019; Saeed & Omlin, 2023; Mittal et al., 2022; Tiwari et al., 2021).

On the technology side, references to Kafka, Redis, RxJava, and PostgreSQL are used to anchor the concrete infrastructural context (Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016; Douglas & Douglas, 2003). These technologies serve as exemplars of log-based streaming, in-memory key–value storage, reactive

programming, and relational persistence in contemporary microservice systems.

The conceptual clustering step involves grouping references into thematic clusters:

- Architectural principles and service design: microservices, clean architecture, REST rulebooks, and integration patterns (Nadareishvili et al., 2016; Sharma, 2017; Martin, 2018; Mass, 2012; Petrasch, 2017).

- Transactional patterns and frameworks: Sagas, SagaMAS, Saga implementation comparisons, and banking-focused Saga versus 2PC analyses (Garcia-Molina & Salem, 1987; Stefanko, 2017; Limón et al., 2018; Hebbar, 2025).

- Deployment, data, and security: systematic literature reviews on deployment and communication, microservice data management, and security patterns (Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022).

- Infrastructure technologies: Kafka, Redis, RxJava, PostgreSQL, and their roles in building scalable, reactive microservices (Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016; Douglas & Douglas, 2003).

- Methodological exemplars: systematic review guidelines and meta-surveys guiding how knowledge is structured and research gaps are articulated (Keele et al., 2007; Sedeño et al., 2019; Saeed & Omlin, 2023; Mittal et al., 2022; Tiwari et al., 2021).

Cross-domain methodological borrowing is then used to structure the argument. In particular, the multi-step processes described by systematic reviews—problem formulation, search and selection, categorization, synthesis, and gap analysis—are mirrored in the way this article constructs its conceptual model of Saga-based architectural design (Keele et al., 2007; Sedeño et al., 2019; Saeed & Omlin, 2023). The aim is not to perform a new systematic review but to adopt the rigorous habits of these methodologies in a narrative synthesis.

Finally, architectural modeling is performed by conceptually applying the clustered insights to two archetypal transaction-intensive scenarios: an e-commerce shopping process and a banking fund-transfer API (Shopping Process, 2022; Hebbar, 2025). These scenarios serve as lenses through which the interplay of Sagas, architecture, and infrastructure is examined. The modeling is descriptive, not executable: it proposes plausible compositions of patterns, frameworks, and technologies consistent with the cited literature, and it explores the implications of these compositions for consistency, scalability, resilience, and security.

RESULTS

Microservice Foundations for Transactional Workflows

Microservice architecture is built on the idea that complex systems are better constructed as collections of small, autonomous services each responsible for a cohesive business capability (Nadareishvili et al., 2016). These services can be developed, deployed, and scaled independently by small teams, enabling faster iteration and more targeted optimizations (Sharma, 2017). From a transactional perspective, this autonomy reshapes the boundaries within which consistency and invariants must be managed.

Nadareishvili and co-authors emphasize the alignment of microservices with organizational structures and culture, arguing that technical boundaries should reflect domain-driven design and team ownership (Nadareishvili et al., 2016). This alignment is critical when designing transactional workflows: the services that take part in a multi-step process like shopping checkout or funds transfer are not arbitrary; they correspond to bounded contexts such as “orders,” “payments,” “inventory,” and “accounts,” each with its own ubiquitous language and domain constraints (Nadareishvili et al., 2016; Martin, 2018).

Sharma’s practical perspective on microservices reinforces this structural view, highlighting concerns such as service granularity, shared libraries versus shared services, and the trade-offs in database ownership models (Sharma, 2017). Transaction-intense flows tend to traverse multiple aggregates, and microservice boundaries must carefully balance cohesion against the overhead of cross-service communication. Too fine a granularity can lead to excessive cross-service transactional coordination; too coarse a granularity reintroduces monolithic characteristics (Sharma, 2017; Stonebraker & Cattell, 2011).

Clean Architecture adds another layer of structure by insisting that domain entities and use cases remain independent of frameworks, UI layers, and external data sources (Martin, 2018). In a Saga-oriented microservice, this implies that the core business rule “reserve inventory and charge payment atomically at business level” should be expressed as a use case independent of whether the orchestration is implemented using a SagaMAS-like framework, Kafka-based event routing, or REST callbacks (Limón et al., 2018; Narkhede et al., 2017). This separation allows architects to swap infrastructural mechanisms without contaminating domain models with low-level concerns (Martin, 2018; Walls, 2016).

REST API design further shapes transactional workflows by imposing constraints on resource naming, HTTP status codes, idempotency semantics, and hypermedia design (Mass, 2012). For example, representing a checkout Saga as a resource with its own lifecycle, identified by a URI such as /checkouts/{id}, allows clients to observe its progress, re-submit idempotent commands, and understand failure states through standardized response codes and representations (Mass, 2012; Hebbar, 2025). Such design

choices are not purely stylistic; they encode expectations about retries, eventual consistency, and compensations that underpin the user experience in distributed transactions (Vogels, 2009; Mass, 2012).

Petrasch's model-based engineering for microservices using Enterprise Integration Patterns provides a systematic vocabulary for the fabric that connects services: message channels, message routers, aggregators, compensating transactions, dead-letter queues, and more (Petrasch, 2017). These patterns are directly applicable to Saga orchestration. For instance, a Saga orchestrator may act as an aggregator, collecting a series of events from underlying services to determine the outcome of a checkout or transfer; an error channel may route failed messages to a compensating workflow; and correlation identifiers become essential for reconstructing the history of each Saga instance (Petrasch, 2017; Limón et al., 2018).

Together, these architectural foundations outline a landscape in which transaction-intensive workflows must be expressed as compositions of domain-centered microservices, integrated via patterns that explicitly acknowledge asynchrony, partial failure, and eventual visibility to users (Nadareishvili et al., 2016; Martin, 2018; Petrasch, 2017; Vogels, 2009).

Saga as a First-Class Pattern for Microservice Transactions

The Saga pattern proposes a structured approach to long-lived, distributed transactions by decomposing them into a sequence of local transactions with well-defined compensations (Garcia-Molina & Salem, 1987). In microservice architectures, these local transactions map naturally onto operations within individual services, each using its preferred datastore and transaction mechanisms.

Stefanko's comparison of Saga implementations reveals a spectrum of options, from simple, library-based approaches to integrated orchestration engines (Stefanko, 2017). Library-based solutions embed Saga state and coordination logic directly into service code, granting flexibility at the cost of duplication and potential inconsistencies in handling corner cases. Orchestration engines, by contrast, centralize Saga definitions, state management, and retry policies, offering stronger guarantees and better observability but introducing a new infrastructural dependency (Stefanko, 2017; Limón et al., 2018).

SagaMAS exemplifies a framework-level approach. Limón and co-authors design SagaMAS as a software framework for distributed transactions in microservices, providing abstractions for defining Saga steps, compensation actions, and coordination logic (Limón et al., 2018). The framework supports service invocation, state persistence, and rollback handling, enabling developers to declare Sagas declaratively and rely on the framework for execution and monitoring (Limón et al., 2018). This moves some of the complexity inherent in Sagas from application code into a reusable infrastructure component, aligning with the general trend in microservices to encapsulate cross-cutting concerns in dedicated platforms.

Hebbar's study of Saga versus two-phase commit in banking APIs adds domain-specific nuance (Hebbar, 2025). Banking operations often require multi-step workflows, such as verifying identities, checking limits, reserving funds, sending payment instructions, and updating statements. Hebbar shows that Saga is well-suited to these workflows because it can express intermediate success and failure, compensations for partial progress, and integration with external systems that cannot join strict distributed transactions (Hebbar, 2025). He notes that while two-phase commit promises atomicity, its operational complexity and blocking behavior in distributed, heterogeneous environments make it harder to align with the principles of microservices and the realities of banking ecosystems that include third-party providers and regulators (Hebbar, 2025).

Saga-based design, however, forces explicit modeling of compensations, which in banking is not trivial. For instance, if funds have been reserved in a ledger but the payment network rejects the transfer, a compensating transaction must release the reservation. In contrast, if the payment has already been settled externally, compensation may require initiating a new transfer in the opposite direction or issuing a credit to the customer's account, leading to more complex financial and legal semantics (Hebbar, 2025; Garcia-Molina & Salem, 1987). SagaMAS and similar frameworks can mechanize the mechanics of compensation but cannot define the semantics; this remains a domain modeling task.

In e-commerce shopping, Sagas often orchestrate inventory reservation, payment authorization, shipping label allocation, and notification (Shopping Process, 2022; Sharma, 2017). If any step fails, compensations might include releasing reserved inventory, canceling payment authorizations, or retracting shipping labels. This domain tends to tolerate eventual consistency more readily than banking; customers may accept brief delays in seeing order status updates as long as the ultimate outcome is correct and well communicated (Vogels, 2009; Mass, 2012).

SAGA BigJob extends Saga-like abstraction into high-performance, distributed computing, illustrating the generality of coordinated distributed tasks over heterogeneous resources (Luckow et al., 2010). While its context is cluster and grid computing rather than business transactions, the abstraction of a "pilot job" that encapsulates multiple tasks, scheduling, and resource management resonates with Saga's approach to grouping multiple operations under a single logical umbrella (Luckow et al., 2010).

Across these contexts, the core insight is that Saga repositions transactional responsibility from a single, monolithic transaction manager to a distributed model where each service participates through local transactions and compensations, while coordination frameworks and

messaging systems manage ordering, retries, and failure detection (Garcia-Molina & Salem, 1987; Limón et al., 2018; Stefanko, 2017).

Reactive Data Infrastructure: Kafka, Redis, RxJava, and PostgreSQL

Modern microservice systems are increasingly built on reactive and streaming infrastructure that supports real-time data propagation, non-blocking I/O, and elastic consumption of events (Narkhede et al., 2017; Nurkiewicz & Christensen, 2016). Transaction-intensive workflows are no longer confined to synchronous request–response paths; instead, they unfold across a graph of events, commands, and projections that connect services asynchronously.

Kafka provides a distributed, append-only log that scales horizontally and supports high-throughput, low-latency event streaming (Narkhede et al., 2017). In a Saga-oriented architecture, Kafka topics can host domain events such as OrderPlaced, PaymentAuthorized, InventoryReserved, and ShipmentCreated. Saga orchestrators or participant services consume these events to trigger subsequent steps or compensations, achieving loose coupling and temporal decoupling between services (Narkhede et al., 2017; Petrasch, 2017). Kafka's partitioning and consumer group mechanisms allow Saga-related events to be processed in parallel while maintaining per-key ordering, which is critical for correctness when multiple events relate to the same order or transaction (Narkhede et al., 2017; Stonebraker & Cattell, 2011).

Redis, an in-memory key–value store supporting rich data structures and atomic operations, often complements Kafka as a low-latency store for session state, idempotency keys, rate limits, and transient Saga state (Redis, 2022). For instance, a Saga orchestrator might use Redis to track the current step of each Saga instance, store correlation identifiers, and record whether a given event has already been processed, ensuring that at-least-once delivery semantics do not lead to duplicate side effects (Redis, 2022; Petrasch, 2017). Redis's support for TTLs can help expire stale Saga instances and reclaim resources associated with abandoned workflows (Redis, 2022).

RxJava provides a compositional model for asynchronous, event-based programming through observable streams and operators such as map, flatMap, retry, and onErrorResumeNext (Nurkiewicz & Christensen, 2016). In microservice applications, RxJava can serve as the programming model for building reactive Saga orchestrators and reactive client interactions. For example, events from Kafka can be wrapped as observables, and business logic can be expressed as transformations and side effects on these streams, with error handling operators implementing compensations or fallback paths (Nurkiewicz & Christensen, 2016; Limón et al., 2018).

PostgreSQL remains a robust anchor for relational data storage, transactions, and advanced features such as JSON support, stored procedures, and triggers (Douglas & Douglas, 2003). In Saga-oriented microservices, PostgreSQL can implement local transactions and maintain authoritative state for aggregates such as orders, accounts, and payments. The outbox pattern can be implemented by writing outgoing events into a dedicated table as part of the local transaction, then using a process that reads these outbox records and publishes them to Kafka, ensuring that state changes and event emissions are atomic at the service level (Douglas & Douglas, 2003; Narkhede et al., 2017).

Stonebraker and Cattell's rules for scalable simple-operation datastores emphasize the importance of avoiding distributed transactions, maximizing independence between partitions, and using asynchronous replication and partitioning strategies (Stonebraker & Cattell, 2011). Kafka, Redis, RxJava, and PostgreSQL collectively embody this philosophy: they enable local, simple operations at scale, with coordination achieved through logs and messages rather than synchronous, cross-node locking (Stonebraker & Cattell, 2011; Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016).

In a shopping process, for example, the customer's action of "placing order" may synchronously create an order in a PostgreSQL-backed order service and synchronously reserve inventory in a separate service, while producing events that other services consume to perform further actions such as charging payment and arranging shipment (Shopping Process, 2022; Douglas & Douglas, 2003; Narkhede et al., 2017). In a banking transfer API, an initial debit operation may be executed locally in an account service with corresponding events driving downstream settlement and notification services (Hebbar, 2025; Narkhede et al., 2017).

Reactive infrastructure thus provides the substrate on which Sagas can be executed, monitored, and recovered. However, integrating these technologies into coherent, secure, and maintainable architectures requires systematic understanding of deployment, data management, and security patterns.

Deployment, Communication, Data, and Security Considerations

Microservice architectures are not only logical decompositions; they also manifest as concrete deployment topologies and communication patterns that affect latency, failure propagation, and security. Karabey Aksakalli and co-authors' systematic review of deployment and communication patterns in microservices reveals a wide variety of options, including single-host and multi-host deployments, container-orchestrated clusters, service meshes, synchronous and asynchronous communication, and various forms of API gateways and message brokers (Karabey Aksakalli et al., 2021).

Saga-oriented transactional architectures must be aware of these patterns. For instance, Saga orchestrators may be

deployed as dedicated services within a cluster, relying on service discovery and load balancing to communicate with participant services (Limón et al., 2018; Karabey Aksakalli et al., 2021). The chosen communication patterns—REST over HTTP, gRPC, messaging via Kafka, or hybrid approaches—affect reliability and observability. Synchronous REST calls may simplify error handling but risk cascading failures; asynchronous messaging decouples services but complicates debugging and latency control (Petrasch, 2017; Narkhede et al., 2017).

Laigner and colleagues highlight that microservice data management is characterized by polyglot persistence, decentralized data ownership, and challenges in maintaining consistency and transactionality across services (Laigner et al., 2021). They describe patterns such as database-per-service, event sourcing, and CQRS as prevalent strategies to manage these challenges. In Saga-based architectures, database-per-service aligns naturally with the notion that each Saga step operates on a single local datastore, while event sourcing and CQRS can provide detailed auditability and flexible read models for transactional histories (Laigner et al., 2021; Narkhede et al., 2017; Martin, 2018).

However, Laigner and co-authors also emphasize that transactional guarantees become weaker as more services and data stores are involved, making compensations, idempotency, and careful schema evolution mandatory (Laigner et al., 2021; Vogels, 2009). In banking and e-commerce, where data integrity and customer trust are paramount, this implies that Saga-based designs must incorporate robust mechanisms for conflict detection, reconciliation, and manual override in exceptional cases (Hebbar, 2025; Shopping Process, 2022).

Berardi and collaborators' systematic review on microservice security underscores that decomposition multiplies the attack surface, requiring consistent application of authentication, authorization, encryption, and auditing across services (Berardi et al., 2022). For Saga-oriented transaction flows, this has several implications. Orchestrators and participants must authenticate mutually; tokens representing customer or system identities must be propagated securely; and compensation actions must be subject to the same authorization rules as forward actions (Berardi et al., 2022; Mass, 2012; Hebbar, 2025). Logging and monitoring required for Saga observability must be designed to avoid leaking sensitive data while providing enough detail for incident response and compliance reporting (Berardi et al., 2022; Laigner et al., 2021).

Furthermore, Saga-based architectures must be hardened against denial-of-service and abuse. Mittal and co-authors' systematic review of deep learning approaches for DDoS detection, although focused on network-level defenses, signals the importance of proactive strategies to detect and mitigate overload and malicious traffic in distributed systems (Mittal et al., 2022). In microservice environments, DDoS can target API gateways, orchestrators, or backing datastores; rate limiting, backpressure, and load shedding must be built into the architecture (Karabey Aksakalli et al., 2021; Narkhede et al., 2017; Redis, 2022). Redis can be used as a central rate-limiting store, while Kafka-based buffering can absorb spikes and decouple ingestion from processing (Redis, 2022; Narkhede et al., 2017).

Knowledge graphs, as surveyed by Tiwari and colleagues, provide a complementary perspective on how complex relationships in systems—between microservices, APIs, data stores, and domain entities—can be modeled and queried (Tiwari et al., 2021). For transaction-intensive systems, knowledge graphs could be used to represent Saga structures, service dependencies, and security policies, enabling more advanced reasoning and tooling for impact analysis, compliance checking, and anomaly detection (Tiwari et al., 2021; Berardi et al., 2022).

Systematic meta-surveys such as that by Saeed and Omlin on explainable AI highlight the growing expectation that complex systems should be explainable to humans, not only in their models but also in their operational behaviors (Saeed & Omlin, 2023). Saga workflows, with their cross-service steps, compensations, and events, require similar explainability: operators, auditors, and even end users may need to understand why a transaction took a particular path, why a compensation was triggered, or why a temporary inconsistency occurred (Vogels, 2009; Hebbar, 2025). Designing such explainability into Saga frameworks and monitoring tooling becomes a crucial research and practice area.

DISCUSSION

Integrating Saga, Clean Architecture, and Reactive Infrastructure in Practice

The conceptual synthesis presented above suggests that successful transaction-intensive microservice systems are those that deliberately integrate Saga patterns, clean architectural boundaries, and reactive data infrastructure into coherent designs. Achieving this integration requires careful attention to the separation of concerns, the alignment of technology choices with domain semantics, and the management of inevitable trade-offs between consistency, availability, and complexity.

From an architectural standpoint, Clean Architecture provides a powerful mental model: domain entities and use cases form the inner layers, surrounded by application services, interface adapters, and external interfaces such as Kafka, Redis, HTTP APIs, and databases (Martin, 2018; Walls, 2016). Within this structure, Saga definitions should not live at the domain core; rather, they belong in a layer that coordinates use cases across bounded contexts, leveraging frameworks like SagaMAS and infrastructure components such as Kafka and Redis (Limón et al., 2018; Narkhede et al., 2017; Redis, 2022). This keeps domain models focused on core business rules, such as how to compute an account

balance or determine inventory availability, while Saga orchestration handles the ordering and compensation of cross-context interactions.

REST APIs, guided by rulebooks for resource naming and status codes, provide the external contract through which clients initiate Sagas and query their progress (Mass, 2012; Hebbar, 2025). For example, starting a shopping checkout Saga could involve POSTing to /checkouts, receiving a representation that includes a unique identifier and links to track status or cancel the operation (Mass, 2012; Shopping Process, 2022). Internally, the Saga orchestrator might publish a CheckoutInitiated event to Kafka, triggering inventory reservation and payment authorization workflows (Narkhede et al., 2017; Petrasch, 2017).

Reactive programming with RxJava allows such workflows to be expressed as compositional pipelines over streams of events and commands, enabling sophisticated behaviors such as retry with exponential backoff, circuit breaking, and dynamic adaptation to load (Nurkiewicz & Christensen, 2016). When combined with Kafka's backpressure-friendly consumption model and Redis-based state tracking, RxJava-based orchestrators can maintain responsiveness even under varying load and partial failures (Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016).

PostgreSQL's role remains central but localized. Each microservice uses local ACID transactions to maintain strong consistency within its bounded context—for example, ensuring that an order's line items and totals are always consistent, or that an account's balance and ledger entries are reconciled (Douglas & Douglas, 2003; Stonebraker & Cattell, 2011). By using patterns such as outbox to emit events as part of these transactions, services ensure that Sagas operate on a consistent stream of domain events (Narkhede et al., 2017; Douglas & Douglas, 2003).

In banking, Hebbar's findings strongly suggest that such a Saga-oriented, reactive architecture is more robust and adaptable than attempting to extend two-phase commit across microservices and external services (Hebbar, 2025). The microservice literature supports this conclusion by emphasizing the negative impact of tight coupling and synchronous coordination on scalability and agility (Nadareishvili et al., 2016; Karabey Aksakalli et al., 2021; Laigner et al., 2021).

Theoretical and Practical Trade-Offs

Despite its appeal, Saga is not without drawbacks. One theoretical trade-off is the shift from strong, system-enforced atomicity to weaker, model-dependent consistency, which must be carefully reasoned about and tested (Garcia-Molina & Salem, 1987; Vogels, 2009). Unlike a single database transaction, where invariants are often enforced by constraints and short-lived locks, a Saga may span seconds or minutes, during which external events, concurrent Sagas, and partial system outages can alter the environment (Limón et al., 2018; Laigner et al., 2021).

This raises subtle questions: How should conflicting Sagas be handled when they touch the same domain entities? What happens if a compensation fails or partially succeeds? How are legal and regulatory requirements translated into compensation semantics in domains like banking? The literature provides partial answers—emphasizing idempotency, commutative updates, and carefully designed compensation policies—but much of the burden remains on domain experts and architects (Garcia-Molina & Salem, 1987; Hebbar, 2025; Laigner et al., 2021).

A practical trade-off concerns observability and explainability. While Saga frameworks can maintain state machines and logs of transitions, reconstructing the full narrative of a complex transaction that traverses many services and events remains challenging (Limón et al., 2018; Karabey Aksakalli et al., 2021). Here, methodological insights from knowledge graphs and explainable AI become relevant: representing Saga instances, steps, events, and compensations as nodes and edges in a graph could enable more intuitive queries and visualizations of transactional histories (Tiwari et al., 2021; Saeed & Omlin, 2023). Yet implementing such capabilities introduces additional infrastructure and modeling complexity.

Security trade-offs also arise. Each additional service in a Saga increases the number of points where authentication, authorization, and input validation must be correctly enforced (Berardi et al., 2022). Compensations can inadvertently violate security assumptions if they bypass normal validation paths or introduce new side effects not considered in original policies (Berardi et al., 2022; Hebbar, 2025). Designing and verifying security properties across forward and compensating flows demands systematic approaches, possibly informed by model-based engineering and security-oriented service discovery (Petrasch, 2017; Sedeño et al., 2019).

Deployment trade-offs, described in the systematic review of deployment and communication patterns, also play a role (Karabey Aksakalli et al., 2021). Co-locating Saga orchestrators and participating services in the same cluster can reduce latency but may increase the scope of failure domains; distributing them across zones or regions improves resilience but accentuates consistency and latency challenges. Hybrid patterns, using service meshes to provide uniform observability and fault injection capabilities, have potential but require production-grade operational maturity (Karabey Aksakalli et al., 2021; Laigner et al., 2021).

Methodological Reflections and Research Gaps

Drawing on systematic review methodologies, it is possible to use the existing literature to outline a research agenda and identify gaps that are particularly relevant to Saga-based transactional architectures (Keele et al., 2007; Saeed & Omlin, 2023; Mittal et al., 2022).

First, there is a lack of large-scale, empirical evaluations of Saga frameworks like SagaMAS in real-world transactional

domains such as banking and e-commerce. While conceptual and small-scale case studies exist, comprehensive benchmarks that measure throughput, latency, failure recovery time, and developer productivity compared with alternative patterns are scarce (Limón et al., 2018; Hebbar, 2025; Laigner et al., 2021). Adopting methodologies akin to those used in DDoS detection evaluations could yield more robust evidence, systematically exploring the parameter space of loads, failure scenarios, and topologies (Mittal et al., 2022; Karabey Aksakalli et al., 2021).

Second, there is limited work on combining Saga with advanced data management strategies such as multi-model databases, cross-datastore transactions, and zero-trust security architectures (Laigner et al., 2021; Berardi et al., 2022). As microservice systems increasingly integrate relational, NoSQL, and streaming stores, understanding how Saga semantics interact with heterogeneous consistency guarantees becomes critical, especially in regulated domains.

Third, the intersection of Saga orchestration with explainability and knowledge representation is under-explored. While knowledge graphs have been proposed for representing relationships in complex domains, applying them to microservice topologies and Saga flows could open powerful new capabilities for auditing, impact analysis, and automated reasoning about transaction safety (Tiwari et al., 2021; Saeed & Omlin, 2023).

Fourth, there is a need for improved methodological support for practitioners facing service decomposition decisions. Sedeño and colleagues highlight the challenge of systematic service discovery in early agile development stages, underscoring that the way services are identified and bounded has long-term implications for transactional patterns (Sedeño et al., 2019). Integrating Saga considerations and data management constraints into service discovery methodologies could prevent architectural anti-patterns like transactional "distributed monoliths."

Finally, security remains both crucial and delicate. Berardi and co-authors' review illustrates the variety of microservice security patterns, but their specific interactions with Saga compensation paths are not well documented (Berardi et al., 2022). Given that compensations can alter system state in ways not originally foreseen, formal methods and static analysis tools tailored to Saga definitions might be an important area of research.

CONCLUSION

This article has provided a deeply elaborated, literature-grounded exploration of how to architect transaction-intensive microservice systems using Saga-based distributed transactions, clean architectural boundaries, and reactive, streaming-oriented data infrastructure. Operating strictly within the constraints of the referenced works, it has synthesized insights from microservice architecture, database scalability rules, transactional pattern frameworks, deployment and communication patterns, data management strategies, and security reviews, situating them in the context of e-commerce shopping flows and banking APIs (Nadareishvili et al., 2016; Stonebraker & Cattell, 2011; Limón et al., 2018; Karabey Aksakalli et al., 2021; Laigner et al., 2021; Berardi et al., 2022; Hebbar, 2025).

The main conclusion is that Saga, supported by frameworks such as SagaMAS and powered by reactive technologies like Kafka, Redis, RxJava, and PostgreSQL, offers a conceptually coherent and practically adaptable mechanism for coordinating distributed workflows across microservices while respecting domain boundaries and infrastructure heterogeneity (Limón et al., 2018; Narkhede et al., 2017; Redis, 2022; Nurkiewicz & Christensen, 2016; Douglas & Douglas, 2003). Clean Architecture and REST design principles ensure that domain logic remains insulated from infrastructure concerns, allowing systems to evolve technologically without compromising core transactional correctness (Martin, 2018; Mass, 2012; Walls, 2016).

At the same time, the analysis underscores that Saga is not a universal remedy. It introduces complexity in compensations and observability, requires disciplined modeling and cross-team collaboration, and must be embedded within robust security and data management practices (Garcia-Molina & Salem, 1987; Berardi et al., 2022; Laigner et al., 2021). Hebbar's banking-specific findings suggest that while Saga is usually preferable to two-phase commit in cross-service transactional contexts, localized use of strong ACID transactions within individual services remains essential (Hebbar, 2025).

Looking forward, the article identifies gaps and future directions that mirror the methodological concerns of systematic reviews in software engineering and related fields: the need for more empirical studies, better tool support, and cross-disciplinary approaches that incorporate knowledge representation and explainable observability (Keele et al., 2007; Saeed & Omlin, 2023; Tiwari et al., 2021). Distributed transactions in microservice systems are as much about organizational culture and domain understanding as they are about technology; microservice architecture itself reminds us that principles, practices, and culture must align for systems to be truly resilient and evolvable (Nadareishvili et al., 2016; Sharma, 2017).

In transaction-intensive domains like banking and e-commerce, embracing Saga-based patterns, while maintaining a critical awareness of their trade-offs and a commitment to ongoing empirical validation, offers a promising path toward architectures that are not only scalable and responsive but also trustworthy, comprehensible, and aligned with the realities of distributed computation.

REFERENCES

1. Berardi, D., Giallorenzo, S., Mauro, J., Melis, A., Montesi, F., & Prandini, M. (2022). Microservice security: A

systematic literature review. PeerJ Computer Science, 7, e779.

2. Douglas, K., & Douglas, S. (2003). PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgreSQL Databases (1st ed.). Sams.
3. Garcia-Molina, H., & Salem, K. (1987). Sagas. ACM SIGMOD Record, 16(3), 249–259.
4. 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
5. Karabey Aksakalli, I., Çelik, T., Can, A. B., & Tekinerdogan, B. (2021). Deployment and communication patterns in microservice architectures: A systematic literature review. Journal of Systems and Software, 180, 111014.
6. Keele, S., et al. (2007). Guidelines for Performing Systematic Literature Reviews in Software Engineering.
7. Laigner, R., Zhou, Y., Salles, M. A. V., Liu, Y., & Kalinowski, M. (2021). Data management in microservices. Proceedings of the VLDB Endowment, 14, 3348–3361.
8. Limón, X., Guerra-Hernández, A., Sánchez-García, Á. 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.
9. Luckow, A., Lacinski, L., & Jha, S. (2010). SAGA BigJob: An extensible and interoperable pilot-job abstraction for distributed applications and systems. In 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (pp. 135–144). IEEE.
10. Martin, R. C. (2018). Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall.
11. Mass, M. (2012). REST API Design Rulebook. O'Reilly.
12. Mittal, M., Kumar, K., & Behal, S. (2022). Deep learning approaches for detecting DDoS attacks: A systematic review. Soft Computing, 27, 13039–13075.
13. Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly Media.
14. Narkhede, N., Shapira, G., & Palino, T. (2017). Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale (1st ed.). O'Reilly Media.
15. Nurkiewicz, T., & Christensen, B. (2016). Reactive Programming with RxJava: Creating Asynchronous, Event-Based Applications (1st ed.). O'Reilly Media.
16. Petrasch, R. (2017). Model-based engineering for microservice architectures using Enterprise Integration Patterns for inter-service communication. In 2017 14th International Joint Conference on Computer Science and Software Engineering (JCSSE) (pp. 1–4). IEEE.
17. Redis. (2022). Redis. Retrieved from https://redis.io/
18. Saeed, W., & Omlin, C. (2023). Explainable AI (XAI): A systematic meta-survey of current challenges and future opportunities. Knowledge-Based Systems, 263, 110273.
19. Sedeño, J., Vázquez, G., Escalona, M. J., & Mejías, M. (2019). The systematic discovery of services in early stages of agile developments: A systematic literature review. Journal of Computer and Communications, 7, 114–134.
20. Sharma, U. R. (2017). Practical Microservices (1st ed.). Packt Publishing.
21. Shopping Process. (2022). Shopping process. Retrieved from https://www.books.com.tw/web/sys_qalist/qa_1_2/0?loc=000_002
22. Stefanko, M. (2017). Saga implementations comparison. Retrieved from http://jbossts.blogspot.cz/2017/12/sagaimplementations-comparison.html
23. Stonebraker, M., & Cattell, R. (2011). 10 rules for scalable performance in "simple operation" datastores. Communications of the ACM, 54(6), 72–80.
24. Tiwari, S., Al-Aswadi, F. N., & Gaurav, D. (2021). Recent trends in knowledge graphs: Theory and practice. Soft Computing, 25, 8337–8355.
25. Vogels, W. (2009). Eventually consistent. Communications of the ACM, 52(1), 40–44.
26. Walls, C. (2016). Spring Boot in Action. Manning Publications.