International Journal for Multidisciplinary Research (IJFMR)
E-ISSN: 2582-2160 ● Website: www.ijfmr.com ● Email: editor@ijfmr.com

A Review of the Saga Pattern for Distributed Transactions in Microservices Architecture

Arun Neelan

Independent Researcher
PA, USA
arunneelan@yahoo.co.in

Abstract:
Microservices architecture has revolutionized software development by decomposing applications into small, independent services that can be developed, deployed, and scaled independently. However, managing transactions that span multiple microservices remains a significant challenge due to decentralized data ownership and autonomous service deployment. Traditional approaches, such as Two-Phase Commit (2PC), often introduce performance bottlenecks and reduce reliability in distributed environments. The Saga pattern addresses these issues by coordinating a sequence of local transactions with compensating actions to achieve eventual consistency.
This paper presents a comprehensive review of the Saga pattern, outlining its core principles and two primary implementation styles: orchestration-based and choreography-based. It compares the Saga pattern with other distributed transaction models, including 2PC and the Outbox pattern, highlighting trade-offs in performance, consistency, and complexity. Real-world use cases and available tooling are examined to demonstrate practical adoption. Key challenges, such as compensation logic, idempotency, observability, and fault tolerance, are explored in depth. Finally, emerging research areas are discussed, including advanced orchestration strategies, formal verification methods, and integration with novel computing paradigms.
By evaluating the strengths and limitations of the Saga pattern, this review offers practical insights for developers and system architects, while also identifying opportunities for future research aimed at improving the resilience and scalability of microservices-based applications.

Keywords: Saga Pattern, Distributed Transactions, Microservices, Orchestration-based Saga, Choreography-based Saga, Two-Phase Commit (2PC), Outbox Pattern, Transaction Management, Compensation Logic, Event-drive architecture.

I. INTRODUCTION

Microservices have become a widely adopted architectural style for building scalable and maintainable systems. By splitting applications into independently deployable services, they offer clear benefits in modularity and agility. However, managing data consistency and reliability across distributed services presents complex challenges. Traditional transaction protocols like Two-Phase Commit are often unsuitable for these environments, which has led to the rise of alternative approaches such as the Saga pattern [1][2].

A. Context: Rise of Microservices and the Complexity of Distributed Transactions

The adoption of microservices architecture has grown rapidly in the software industry due to its support for scalability, flexibility, and faster release cycles [2]. By decomposing large, monolithic applications into smaller, independently deployable services, systems can scale components on demand, release updates more frequently, and isolate failures more effectively.

However, this architectural style introduces significant challenges, particularly in maintaining data consistency across distributed systems. Each microservice typically owns its own database to preserve autonomy and reduce coupling [2]. When business processes span multiple services—for example, order management, payment processing, and inventory updates, ensuring consistency and correctness across these distributed components becomes complex. Strict transactional guarantees, as traditionally defined, are often difficult to maintain in such environments [4]. Instead, eventual consistency models are frequently adopted as a more practical way to manage data across microservices [3][5].

B. Limitations of Traditional Distributed Transactions

Traditional protocols such as Two-Phase Commit (2PC) enforce consistency by coordinating a global commit among all participating services. However, 2PC presents several challenges in microservices environments, including:

- **Tight coupling between services:** This conflicts with the microservices principle of independent deployability.
- **Synchronous, blocking communication:** This can increase latency and degrade overall system performance.
- **Long-held locks:** Holding locks for extended periods raises the risk of deadlocks and resource contention.
- **Single point of failure:** The central coordinator can compromise system reliability if it fails.

Due to these drawbacks, 2PC and similar protocols are often considered impractical for modern, scalable, cloud-native microservices architectures [2][4].

C. Saga Pattern: A Practical Solution

The Saga pattern has emerged as a practical and popular approach for managing distributed transactions in microservices. Instead of relying on a global transaction, a saga decomposes the process into a sequence of local transactions executed across different services. If any step fails, compensating transactions are triggered to undo prior changes, enabling eventual consistency without requiring distributed locks or global coordination [1][2].

There are two primary implementation styles for sagas. The first, choreography-based, involves each service listening for events and publishing its own to coordinate the overall process. The second, orchestration-based, uses a centralized controller to direct the transaction flow [1][2]. Both approaches promote asynchronous communication, loose coupling, and improved system resilience. However, they introduce new challenges such as managing compensation logic, ensuring idempotency, and effectively monitoring complex asynchronous workflows.

D. Aim and Scope of this Review

This paper provides a comprehensive review of the Saga pattern as a solution for distributed transactions in microservices. The main objectives are to:

- Explore the architectural principles and variations of the Saga pattern.
- Analyze practical challenges including failure handling, state management, and observability.
- Survey current tools, frameworks, and industry best practices.
- Highlight ongoing research trends and open problems in the field.

By synthesizing insights from academic research and industry practice, this review aims to support architects and developers in designing microservices systems that are both resilient and consistent.

II. BACKGROUND AND MOTIVATION

A. Overview of Distributed Systems and Consistency Models

Distributed systems consist of multiple independent components that communicate over a network to accomplish shared goals. Ensuring data consistency in these systems is inherently challenging due to

factors such as network delays, failures, and concurrent operations. According to the CAP theorem, when a network partition occurs, a distributed system can guarantee either consistency or availability, but not both simultaneously [3].

To navigate these trade-offs, two main consistency models are commonly used:

- **ACID (Atomicity, Consistency, Isolation, Durability):** This model ensures strong consistency and transactional integrity and has traditionally been employed in monolithic database systems [4].
- **BASE (Basically Available, Soft state, Eventual consistency):** This approach prioritizes availability and partition tolerance by relaxing consistency guarantees, making it more suitable for distributed microservices environments [5].

B. Challenges of Distributed Transactions in Microservices

Microservices architecture emphasizes service autonomy, where each service owns and manages its own database. This design improves scalability and fault isolation but introduces significant complexity when business processes span multiple services with separate data stores. Ensuring atomicity and consistency across these distributed databases remains a difficult challenge.

C. Limitations of Two-Phase Commit (2PC)

The Two-Phase Commit (2PC) protocol is a traditional approach used to ensure atomic transactions across distributed systems by coordinating a global commit process. It operates in two stages:

- **Prepare phase:** All participants vote on whether they are ready to commit.
- **Commit phase:** The coordinator decides to either commit or roll back the transaction based on these votes.

Although 2PC offers strong theoretical guarantees, it faces several practical limitations when applied to microservices architectures:

- **Blocking behavior:** Participants must hold locks until the global commit completes, increasing latency and lowering system throughput.
- **Tight coupling:** The coordinator acts as a single point of failure and can become a bottleneck.
- **Reduced availability:** Network partitions may cause transactions to block indefinitely.

Due to these issues, 2PC is generally considered unsuitable for the asynchronous and loosely coupled nature of microservices [2].

D. Motivation for Using the Saga Pattern

To address the limitations of Two-Phase Commit, the Saga pattern decomposes a distributed transaction into a series of local ACID transactions, each handled by an individual service. If any step fails, compensating transactions are triggered to reverse the effects of previous steps, enabling the system to maintain eventual consistency without relying on global locks or blocking [1].

This approach aligns well with microservices principles by:

- Enabling services to maintain autonomy and be deployed independently [2].
- Using asynchronous communication to improve performance and system availability [2].
- Providing fault tolerance through compensating actions [1].

Although managing compensation logic and monitoring can be complex, the Saga pattern presents a scalable and practical solution for handling distributed transactions in microservices architectures [1].

III. THE SAGA PATTERN: OVERVIEW

The Saga pattern is a design strategy used to maintain data consistency in distributed systems, especially microservices, without relying on traditional protocols such as two-phase commit. Instead of treating an entire business process as a single atomic transaction, the Saga breaks it into a sequence of smaller, independent transactions managed by different services. If any transaction fails, compensating actions are triggered to undo the effects of previous steps, achieving eventual consistency rather than immediate atomicity.

In summary, a Saga can be characterized as:

- A long-running process composed of multiple independent, self-contained operations.
- A method that prioritizes eventual consistency when enforcing strict ACID properties across distributed services is difficult.

A. How the SAGA Pattern Works

A Saga coordinates a sequence of local transactions where each service executes and commits its transaction independently. Upon completing its task, a service notifies the next participant, typically by sending events or messages, to continue the workflow [1].

Key components include:

- **Sequence of Local Transactions:** Each microservice manages its own database and performs local transactions autonomously, communicating progress asynchronously via events or messages [2].
- **Compensating Transactions:** Instead of conventional rollbacks, compensating transactions are explicitly defined business operations designed to reverse previously completed steps when failures occur, thereby preserving data integrity [1].

Example: Online Retail System

1. The Order Service creates a new order and initiates the transaction.
2. The Payment Service processes the customer's payment.
3. The Shipping Service generates a shipping label and schedules delivery.

If the Payment Service fails during the payment step, a compensating transaction is triggered. For example, the Order Service cancels the order to prevent subsequent actions, such as shipping.

B. The Role of Event-Driven Architecture (EDA) in Saga

Sagas are often implemented using event-driven architecture, enabling services to communicate by producing and consuming events. This approach supports asynchronous communication and loose coupling, both critical for creating scalable distributed systems [2].

A typical flow includes:

- The Order Service emits an "OrderCreated" event after creating a new order.
- The Payment Service listens for the "OrderCreated" event and attempts payment processing. Upon success, it emits a "PaymentSuccessful" event.
- The Shipping Service reacts to the "PaymentSuccessful" event by generating the shipping label and scheduling delivery.
- If the payment fails, the Payment Service emits a "PaymentFailed" event, prompting the Order Service to execute compensation logic, such as canceling the order.

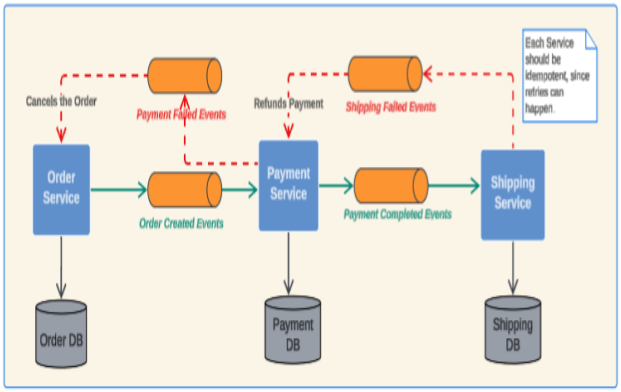

Fig. 1. Choreography Based Saga Flow Example

By decomposing complex transactions into smaller, manageable steps coordinated through events, the Saga pattern enables resilient, scalable, and maintainable management of distributed transactions in microservices environments [1][2].

IV. COORDINATION STYLES IN SAGA

The Saga pattern is a popular approach for managing distributed transactions in microservices-based systems. It is commonly implemented using two coordination styles: choreography and orchestration. Both aim to ensure eventual consistency and resilience but differ in how control flows between services. In choreography, each service reacts to events and makes decisions independently, while orchestration relies on a central coordinator to manage the transaction flow [1][2].

A. Choreography-Based Saga

1) Overview: The choreography-based Saga pattern offers a decentralized way to manage long-running business processes spanning multiple microservices. Unlike orchestration, which depends on a central coordinator, choreography lets each service react independently to the events it receives. This promotes loose coupling and improves scalability in distributed systems [1][2].

In a typical event-driven choreography setup, services subscribe to relevant events, perform their local transactions, then publish new events to trigger the next step. For example, after a customer places an order, the Order Service emits an "OrderCreated" event. The Payment Service listens for this event, processes the payment, and publishes a "PaymentCompleted" event. The Shipping Service then picks up this event to start the delivery process. This continues until the entire business process is complete [1].

While this model improves modularity and fault isolation, it can make coordination more challenging. Monitoring, debugging, and maintaining a clear picture of the overall transaction flow can become difficult, especially as the number of services grows [5].

When failures occur, services emit events that trigger compensating actions—business-specific operations designed to undo the effects of previously completed steps. These compensating transactions help maintain eventual consistency but require each service to be designed carefully for resilience [1].

2) Handling Failures in Event-Driven Choreography: Managing failures is one of the trickiest parts of the choreography-based Saga pattern, mainly because there is no central coordinator to oversee recovery. Each service must detect failures on its own and initiate compensating actions when needed.

The key recovery mechanism is the compensating transaction, which reverses the effect of a previously completed local operation. For example, if the Shipping Service encounters an error after the Payment Service has completed its task, it might emit a "ShippingFailed" event. When the Payment Service receives this, it can trigger a refund and publish a "PaymentReversed" event. This chain of compensation can continue backwards through services until the system reaches a consistent state again [1].

To ensure reliable failure handling in this decentralized model, services should be designed to:

- Perform idempotent operations so retries don't cause inconsistencies.
- Use correlation identifiers to trace the full Saga execution across services.
- Handle event ordering, retries, and duplicate messages gracefully.
- Log and persist key state changes for auditing and recovery.

Embedding failure recovery logic within each service makes choreography resilient, but it also adds complexity to service design and coordination [1][5].

3) Timeout & Incomplete Saga Handling: Not all failures are obvious. Sometimes, an expected event simply never arrives, causing incomplete or stalled Sagas. This can happen due to service crashes, lost messages, or delays. To detect and recover, systems typically use scheduled background processes known as Saga monitors or watchdogs [1].

These monitoring components commonly perform the following tasks:

- Regularly scan a shared state store (e.g., a database tracking Saga progress) to spot stuck transactions—like an order created without payment confirmation within a set timeframe.
- Trigger retries or compensating transactions for stalled Sagas.
- Flag failed Sagas, log incidents, and escalate issues for manual review if automatic recovery fails.

If all services update a shared Saga store, a single monitor can track the entire transaction's status. But in fully decoupled systems, each service may need its own monitoring logic. This boosts autonomy but increases coordination complexity and operational overhead.

4) Alternative Approach – REST Based Choreography: While choreography usually relies on asynchronous, event-driven messaging, it can also be done using RESTful calls. Here, each service directly calls the next after completing its task, keeping decentralization by avoiding a central orchestrator. For example, the Order Service calls the Payment Service after order creation, which then calls the Shipping Service after successful payment.

However, this approach has drawbacks: it creates tighter coupling between services, raises the risk of timeouts and network failures, and forces each service to manage retries and errors. Because of these challenges, REST-based choreography tends to be less resilient and scalable than event-driven implementations. It's mostly used in smaller or transitional systems, and generally not recommended for large-scale microservices architectures [1][2].

B. Orchestration-Based Saga

1) Overview: The orchestration Saga pattern uses a centralized controller to manage distributed, long-running transactions across multiple microservices. Unlike choreography, where services independently react to events, orchestration assigns a dedicated orchestrator to oversee and direct the entire transaction [2].

The orchestrator calls each service in sequence, manages intermediate results, handles failures, and coordinates compensating actions when needed. For example, an Order Orchestrator might start by instructing the Order Service to create an order. After successful completion, it calls the Payment Service to process payment, and once payment is confirmed, it triggers the Shipping Service to begin delivery. This continues until the Saga either completes successfully or rolls back to compensate for failures [1].

There are two common ways to implement orchestration.

- In complex systems, the orchestrator is a standalone service coordinating workflows across microservices like Order, Payment, and Shipping. This provides clear separation of concerns, centralized monitoring, and better reusability [2].
- In simpler systems, orchestration logic might be embedded within a domain service, such as the Order Service managing the Saga internally. While this simplifies design, it reduces flexibility and makes scaling or maintaining cross-domain workflows more difficult [5].

While offers greater visibility and control but also introduces a central dependency. The orchestrator must be highly resilient and available to avoid becoming a single point of failure [1][2].

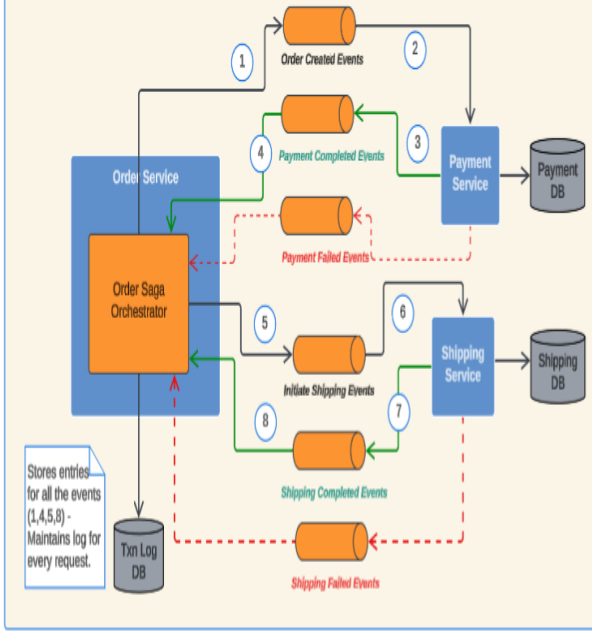

Fig. 2. Orchestration Based Saga Flow Example (Event Driven)

2) Handling Failures in Orchestration: In orchestration-based Sagas, the orchestrator centrally manages failure detection and recovery. If a service fails to complete a step or returns an error, the orchestrator triggers compensating actions to keep the system consistent [1].
For example, if the Shipping Service fails after payment is processed, the orchestrator may instruct the Payment Service to issue a refund. To support this:

- Each service should provide idempotent operations to allow safe retries.
- Saga state must be saved in durable storage for recovery and auditing.
- Timeouts, retries, and circuit breakers should handle transient issues.
- Correlation IDs are crucial to track Saga execution across services.

Additionally, orchestration must also handle less obvious failures like unresponsive services or lost messages, emphasizing the need for managing timeouts and incomplete Sagas [1].

3) Timeout & Incomplete Saga Management: Distributed systems can experience silent failures, such as a service becoming unresponsive. In orchestration-based Sagas, the orchestrator uses timeout mechanisms and progress monitoring to detect and handle these issues [1].
If a service does not respond within the expected time, the orchestrator can:

- Mark the saga as failed and trigger compensating transactions.
- Retry the operation a set number of times.
- Escalate the problem for manual resolution if automated recovery fails.

To manage this effectively, the orchestrator should:

- Maintain a durable state or event log tracking each Saga's status.
- Use scheduled background processes or monitoring tools to spot stalled or incomplete Sagas and initiate recovery.

This prevents transactions from stalling indefinitely and supports system consistency and fault tolerance [1].

4) REST-Based vs. Messaging-Based Orchestration: Orchestration can be implemented with synchronous REST calls or asynchronous messaging, each with pros and cons.

Synchronous REST-Oriented Orchestration:

- Uses direct HTTP calls to invoke services step-by-step.
- Offers clear, predictable control flow since the orchestrator waits for immediate responses.
- Works well for low-latency, short workflows.

However, this approach can create tighter coupling and is vulnerable to network failures or outages, requiring careful handling to avoid cascading failures.

In synchronous workflows, special care is needed for final steps that may not support compensation. For instance, the Shipping Service often triggers irreversible actions. If a timeout occurs but the operation completes later, prematurely rolling back (e.g., refunding payment) can cause inconsistencies. The orchestrator should verify the final step's status before executing compensations. This is especially important for irreversible operations [5].

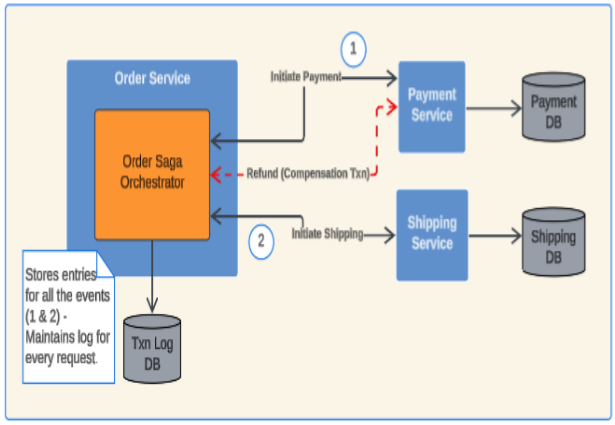

Fig. 3. Orchestration Based Saga Flow Example (Synchronous)

Asynchronous Messaging-Based Orchestration:

- Communicates via message brokers, sending commands and receiving event-based responses.
- Offers higher fault tolerance since messages can be persisted, retried, or delayed.
- Improves scalability and resilience by decoupling service execution.

For example, an orchestrator might send a "CreateOrderCommand" to a message queue and wait for an "OrderCreatedEvent" before proceeding. This approach is more robust for long-running tasks and intermittent failures.

While both have their place, messaging-based orchestration usually provides stronger support for fault-tolerant distributed workflows and is preferred in highly reliable, loosely coupled systems [5].

Some systems combine both methods, using REST for latency-sensitive immediate calls and messaging for durable asynchronous operations. This hybrid approach balances responsiveness and reliability, allowing orchestration to fit different transaction phases [1]. By choosing the right mix, architects can better align orchestration with system requirements, enhancing user experience and operational stability [2].

V. COMPARISON WITH OTHER PATTERNS

This section explores how the Saga pattern compares to other common strategies for handling transactions and messaging in distributed systems. The focus is on how each approach manages coordination, fault tolerance, and system compatibility, particularly within modern microservices architectures.

A. Saga vs. Two-Phase Commit (2PC)

Two-Phase Commit (2PC) is a long-standing protocol used to ensure atomicity across distributed systems. While both 2PC and the Saga pattern deal with multi-step operations, they differ significantly in how they coordinate actions, handle failures, and affect system performance. This comparison highlights key differences in architecture, efficiency, and reliability [1][4].

*1) **Overview of Two-Phase Commit (2PC):*** The 2PC protocol relies on a central component called the Transaction Manager, which coordinates operations across multiple participating services. The process is divided into two distinct steps:

- **Phase 1 – Prepare :** The coordinator sends a "prepare" request to all participants. Each service performs its local transaction, locks necessary resources, and responds with either "Yes" (ready to commit) or "No" (abort).
- **Phase 2 - Commit or Rollback** : If all participants respond with "Yes," the coordinator tells them to commit. If any participant responds with "No," the coordinator instructs all services to roll back.

This protocol ensures that either all services commit successfully or none of them do, which preserves consistency across the system. The typical process looks like this:

1. The coordinator begins the transaction.
2. Participants execute their local operations and acquire locks.
3. Each service returns a vote ("Yes" or "No").
4. The coordinator decides the outcome and sends commit or rollback instructions.
5. Participants complete the transaction and acknowledge the result.

Although 2PC provides strong consistency, it has a major drawback: it can block progress. If the coordinator crashes after collecting votes but before sending the final decision, participants remain stuck in a "prepared" state, holding locks and waiting for resolution. This situation can reduce system responsiveness and increase the risk of deadlocks or bottlenecks. As a result, 2PC is often considered a poor fit for modern, loosely coupled microservices architectures [1][4].

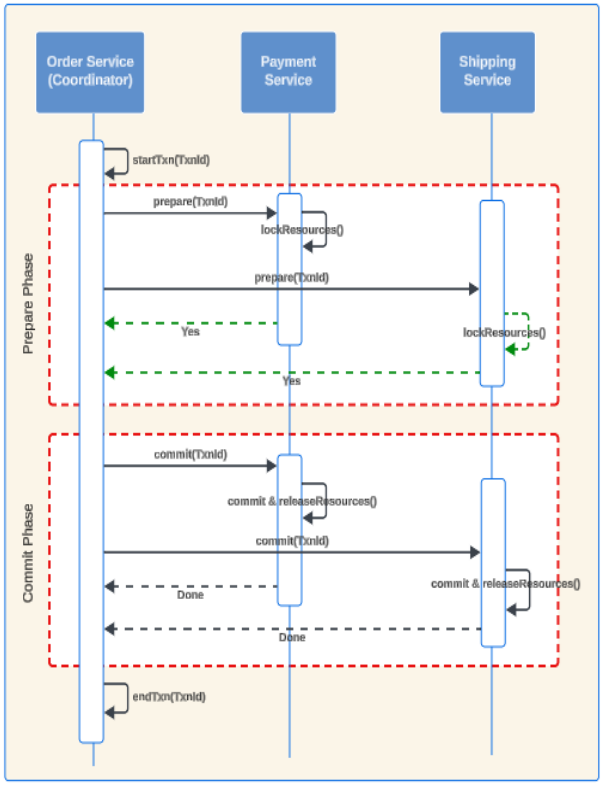

Fig. 4. 2PC - Commit Flow

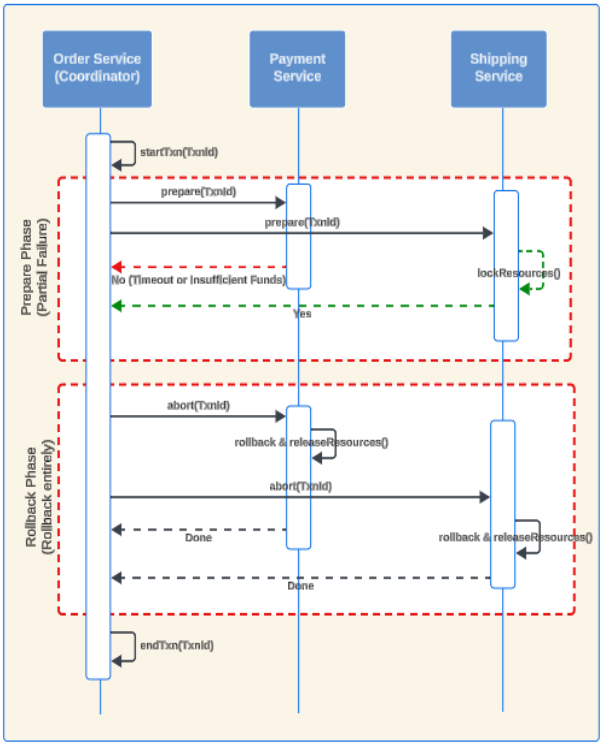

Fig. 5. 2PC - Rollback/Abort Flow

*2) **Limitations and Failure Scenarios:*** Despite its consistency guarantees, 2PC introduces several real-world challenges. One of the most serious is the coordinator acting as a single point of failure. If it goes down after receiving all commit votes but before issuing the final instruction, participants are left in an

uncertain state. This classic blocking issue affects availability until the coordinator recovers and can check its transaction logs [4].

Failures on the participant side can also halt progress. If even one service crashes during the process, the entire transaction is delayed until it comes back online. At the database level, operations like INSERT without an explicit COMMIT or ROLLBACK can leave transactions open, consuming resources and locking data. Some databases automatically roll back such transactions when sessions end, but this behavior is not guaranteed across all systems [4].

Together with its reliance on synchronous communication and high latency, 2PC proves increasingly unsuitable for microservices environments, which prioritize independence, scalability, and asynchronous interaction.

3) The Saga Pattern in Comparison: The Saga pattern offers a more flexible and decentralized alternative to the 2PC protocol. Instead of relying on a central coordinator that locks participants into a single global transaction, a Saga breaks the operation into a sequence of smaller, independent local transactions. Each service handles its own transaction and commits it independently. If a failure occurs at any step, the system triggers compensating transactions to undo previously completed actions. This approach maintains data consistency while allowing services to remain autonomous [1].

Sagas are typically implemented in one of two ways:

- **Choreography:** Services communicate by emitting and reacting to domain events. Each service listens for specific events and triggers the next step in the process.
- **Orchestration:** A central orchestrator service explicitly manages the flow, calling each participant in a defined sequence.

While Sagas do not provide strict atomicity, they support eventual consistency without the need for global locks or tight coupling. This makes them ideal for distributed, cloud-native systems where resilience and scalability are more important than immediate consistency. In summary, the Saga pattern aligns better with modern architecture principles. It encourages asynchronous workflows, improves system resilience, and reduces the risk of system-wide failures. As a result, it is often the preferred approach in microservices environments [1][2].

4) Summary Comparison:

Aspect	*Two-Phase Commit (2PC)*	*Saga Pattern*
Atomicity Guarantee	Strong atomicity (all-or-nothing).	Eventual consistency through compensation.
Locking & Blocking	Yes, participants block during protocol.	No, each local transaction commits independently.
Failure Recovery	Dependent on coordinator logs and recovery.	Handled via compensating transactions.
Single Point of Failure	Coordinator is critical.	No central dependency (choreography) or optional (orchestration).
Scalability	Poor in distributed environments.	High; promotes loose coupling.
Suitability for Microservices	Limited; tightly coupled and synchronous.	High; asynchronous, fault-tolerant.
Complexity	Low compensation logic but recovery risk.	Requires compensation logic per operation.

Table 1. Saga vs. Two-Phase Commit (2PC)

B. Saga vs. Outbox Pattern

The Outbox pattern is primarily used to ensure reliable delivery of messages or events from a single service, maintaining consistency between database changes and event publication. In contrast, the Saga

pattern focuses on coordinating business transactions that span across multiple services. This section outlines the core differences between the two patterns in terms of their scope, reliability mechanisms, and impact on architecture.

1) Overview of the Outbox Pattern: The Outbox pattern is designed to ensure that updates made by a service are consistently propagated to other systems, especially when using asynchronous messaging. It addresses the dual-write problem, which occurs when a service needs to update its database and publish an event as part of the same operation.

To avoid inconsistencies, this pattern introduces an Outbox table inside the same database used for the service's primary data. Rather than publishing the event immediately, the service writes it to the Outbox table as part of the same database transaction. A separate background process, often referred to as an Outbox Poller, then reads the table and publishes the event to a message broker. This guarantees that the event is not published unless the database changes have been committed successfully [1][6].

A typical Outbox implementation follows these steps:

1. The service processes a request and writes both domain data and the event to the Outbox table as part of a single transaction.
2. The transaction is committed, ensuring both changes are persisted together.
3. A poller or a Change Data Capture (CDC) tool scans the Outbox table for new entries.
4. Detected events are published to the message broker and consumed by downstream services.
5. After successful delivery, entries are either marked as sent or deleted from the Outbox.

This pattern ensures atomic visibility: an event will not be observed or consumed unless the associated database operation has been fully committed. This eliminates problems such as orphaned events or premature processing by other services.

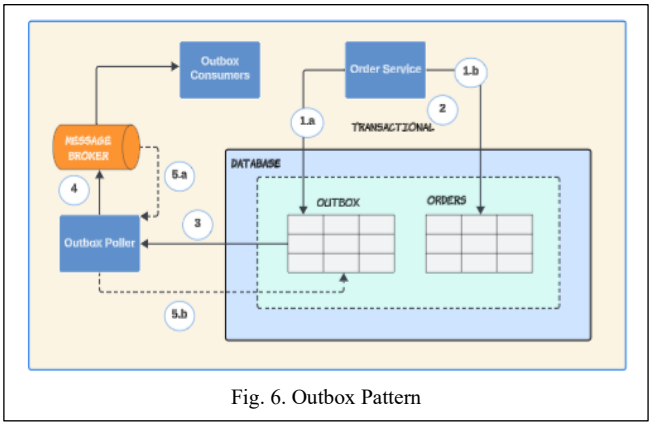

Fig. 6. Outbox Pattern

2) Limitations and Failure Scenarios: While the Outbox pattern strengthens messaging reliability, it introduces certain challenges in both design and operations. Managing the lifecycle of Outbox entries—including idempotency, duplicate suppression, and failure recovery—requires careful engineering. If the poller process fails or slows down, event delivery may be delayed, resulting in eventual consistency instead of real-time updates [5][6].

There are also operational concerns:

- Database management overhead grows as Outbox tables accumulate unprocessed or old entries.
- Monitoring and retry logic must be built to handle failed message deliveries.
- In high-traffic systems, the Outbox table can become a performance bottleneck, especially when using polling rather than CDC [6][7].

In addition to design complexity, the Outbox pattern introduces notable operational overhead such as managing and maintaining database schemas for Outbox tables, implementing monitoring and retry

mechanisms for failed event deliveries, and controlling storage growth, particularly when outdated Outbox entries are not purged efficiently [6][7].

Perhaps most importantly, while the Outbox pattern ensures reliable event publication within a single service boundary, it does not manage cross-service coordination or transaction consistency. Its scope is limited to ensuring safe and consistent event propagation, not end-to-end business process management [1] [6].

3) The Saga Pattern in Comparison: The Saga pattern addresses the challenge of managing distributed, long-running transactions across multiple services. It works by sequencing local transactions and invoking compensating actions when one of the steps fails. This is fundamentally different from the Outbox pattern, which focuses only on reliable message emission from a single service [1][2].

In practice:

- The Outbox pattern is often used within each service participating in a Saga to ensure that the events they emit after completing local transactions are delivered reliably [6].
- The Saga pattern, on the other hand, manages the overall coordination, whether through event choreography or an orchestrator, and defines what happens when something goes wrong in a downstream service [1][7].

In essence, the Saga pattern solves a broader and more complex problem. While the Outbox ensures that a single service can safely emit events, Sagas coordinate how multiple services interact and recover across a complete business workflow. The two patterns often complement each other in practice, with Outbox enabling reliable messaging and Sagas ensuring transactional consistency across services [1][6][7].

4) Summary Comparison:

Aspect	***Saga Pattern***	***Outbox Pattern***
Primary Purpose	Coordinate multi-step transactions across services.	Ensure reliable event publication from a single service.
Scope	Business-level workflow.	Infrastructure-level messaging.
Failure Handling	Compensating transactions triggered on failure.	Retry mechanism + idempotency for message delivery.
Consistency Model	Eventual consistency via compensation.	Local atomicity + eventual delivery.
Coordination	Choreography or Orchestration.	No coordination; only message emission.
Integration	May use Outbox to emit events from each step.	Often used inside Saga implementations.
Best Use Case	Order processing, travel bookings, financial workflows.	Event-driven systems, decoupling service state from messaging.

Both patterns complement each other rather than compete. The Saga pattern provides the overall coordination logic for distributed transactions, while the Outbox pattern ensures reliable message delivery. This reliability supports Sagas, particularly when using choreography, by increasing confidence in message propagation.

C. Saga vs. Event Source Integration

Event Source Integration and the Saga pattern are both used in distributed systems to enable service decoupling. However, they serve different purposes. This section compares their goals, how they handle data, and how they manage failures.

1) Overview of Event Source Integration: Event Source Integration is an architectural approach where services communicate by reacting to domain events published by other services. These events, which are often stored in an event store or written via an Outbox mechanism, act as the primary means of sharing state changes across service boundaries [6][8].

Unlike request-driven architectures, Event Source Integration follows a reactive, event-driven model. Services emit events after key business actions (such as OrderCreated or PaymentReceived), and other services consume these events to update their local state or trigger follow-up processes. This model promotes loose coupling, scalability, and extensibility in distributed systems [2][8].

Key characteristics include:

- Producers emit immutable domain events after completing local transactions.
- Consumers subscribe to and process these events to update state or trigger additional workflows.
- Events are transmitted through message brokers like Apache Kafka or RabbitMQ.
- Services may may use events to build materialized views or perform additional business logic.

This model often complements architectural patterns like CQRS (Command Query Responsibility Segregation), where write and read models are decoupled for better performance and scalability [8].

2) Limitations and Failure Scenarios: While Event Source Integration offers flexibility and scalability, it also introduces several limitations.

- **No built-in coordination:** There is no central mechanism to manage multi-step workflows. Consumers process events independently, which means they must handle out-of-order, duplicate, or missing events [6][8].
- **Schema evolution challenges:** Changes in event structure or semantics can break downstream consumers if versioning is not handled carefully [8].
- **Increased operational complexity:** Services must maintain durable and replayable event logs, manage idempotency, and ensure reliable delivery, which adds infrastructure overhead [11].
- **No rollback support:** Since each consumer operates independently, reversing changes when a downstream process fails is difficult.
- **Eventually consistent by design:** Consumers may temporarily hold stale data, especially when there are delays or retries in message processing. This aligns with eventual consistency principles as described by Vogels [5].

In short, while Event Source Integration excels at broadcasting state changes, it lacks the ability to coordinate business logic or recover from failures across multiple services.

3) The Saga Pattern in Comparison: The Saga pattern is purpose-built for modeling long-running, distributed business transactions. Unlike Event Source Integration, which simply propagates state changes, the Saga pattern enforces a specific execution sequence and includes failure recovery mechanisms [1][6].

Although both patterns rely on events, their use cases differ:

- Event Source Integration treats events as notifications with no coordinated follow-up or acknowledgment.
- The Saga pattern, in contrast, uses events (in choreography) or commands (in orchestration) to coordinate progress across multiple services. It defines the start, sequence, and rollback logic for a complete transaction [1][2].

A key feature of the Saga pattern is its support for compensating actions. These are logical undo operations that reverse earlier steps when a failure occurs. This level of transactional coordination is not natively supported in Event Source Integration.

In practice:

- Event Source Integration is commonly used for service synchronization, state replication, or reactive processing [6][8].

- The Saga pattern is employed when business consistency, step coordination, and failure handling across services are required [1][14].

4) Summary Comparison:

Aspect	*Saga Pattern*	*Event Source Integration*
Primary Purpose	Coordinate multi-step business processes.	Propagate state changes across services via events.
Scope	Transactional workflow across services.	Data synchronization and service decoupling.
Failure Handling	Compensating transactions.	Retry, idempotency, but no built-in rollback.
Consistency Model	Eventual consistency with compensation logic.	Eventual consistency without coordination.
Coordination	Orchestration or choreography of service actions.	No coordination; services react independently.
Integration	May emit and consume events as part of process steps.	Emits events as primary interface for state propagation.
Best Use Case	Order fulfillment, financial workflows, booking systems.	Data replication, search index updates, view projections.

Event Source Integration and the Saga pattern serve different but complementary roles in distributed systems. Event Source Integration focuses on broadcasting domain events to propagate state changes across services in a reactive and decoupled manner [6][8]. The Saga pattern, in contrast, provides process-level coordination and failure recovery for managing long-running business transactions across service boundaries [1][2]. These approaches can be used together in scenarios such as event-driven Sagas that use choreography, allowing systems to achieve scalability, fault tolerance, and consistency without relying on centralized control [1][6][14].

VI. IMPLEMENTATION IN PRACTICE

The Saga pattern is widely implemented in real-world systems that require resilient, fault-tolerant, and distributed transaction coordination. This section explores how the pattern is applied across major platforms, programming languages, and frameworks that support Saga-based workflows.

A. Usecases from Industry

1) Netflix Conductor – Orchestration-Based Workflow Engine: Netflix Conductor is a workflow orchestration engine designed to coordinate microservice interactions. It models business logic as a sequence of tasks, where a central controller governs the order of execution, handles retries, and invokes compensating actions in case of failures. As an orchestration-style implementation of the Saga pattern, it provides strong control over distributed workflows [9].

2) Camunda – BPMN-Oriented Orchestration Platform: Camunda is a business process management platform that supports long-running workflows using BPMN (Business Process Model and Notation). It enables both developers and business analysts to model Saga logic visually, including service calls and compensation paths. Camunda's execution engine manages the flow and recovery logic, making it well-suited for enterprise systems that require collaboration between technical and non-technical stakeholders [10].

3) Apache Kafka – Event-Driven Choreography: Apache Kafka is commonly used to implement Saga choreography. In this model, services communicate by emitting and consuming domain events. After completing a local transaction, a service publishes an event that triggers the next action in the workflow.

Kafka's event-streaming capabilities support loose coupling and eventual consistency, although they introduce challenges in monitoring, tracing, and managing complex workflows [11].

B. Programming Lanugage Support

The Saga pattern is supported across multiple programming languages and ecosystems:

- **Java:** Developers often use Spring Boot along with Spring Cloud, integrating with Kafka or RabbitMQ to implement sagas. The Spring State Machine library is commonly used to manage workflow states, while compensation logic is embedded in the service code itself [6][12].
- **Node.js:** Asynchronous and event-driven programming in Node.js aligns well with Saga principles. Developers can build sagas using message brokers and orchestration engines like Temporal, enabling non-blocking, efficient microservices [13].

C. Tools and Frameworks

Several mature frameworks offer built-in support for implementing the Saga pattern:

- **Axon Framework (Java) :** Provides support for CQRS and Event Sourcing, and includes Saga components that coordinate distributed transactions by subscribing to events and managing compensating actions [14].
- **Temporal.io :** Offers a durable orchestration engine that allows sagas to be defined programmatically. It simplifies failure handling, retries, timeouts, and compensation logic, while abstracting away state management [13].
- **NserviceBus (.NET) :** Includes robust abstractions for managing sagas across long-running, stateful processes. Using message handlers and timeouts, it supports both orchestration and choreography, depending on the desired service interaction model [15][16].

VII. CHALLENGES AND LIMITATIONS

The Saga pattern provides a robust way to manage distributed transactions across microservices by embracing eventual consistency and decoupled workflows. However, this flexibility comes with a range of non-trivial challenges that impact correctness, resilience, and operational overhead [1][2][5].

Unlike traditional ACID transactions, Sagas push the responsibility for coordination, rollback, and consistency to the application layer [1][4]. This architectural shift introduces trade-offs that increase system complexity, place greater demands on developers, and complicate observability and fault diagnosis.

1) Compensation Logic Complexity: One of the most difficult aspects of Saga implementation is defining compensating transactions. For every business operation that could fail downstream, a corresponding rollback step must be written. This is especially challenging when dealing with irreversible actions such as sending emails, interacting with third-party APIs, or transferring funds. Without robust compensation logic, inconsistencies may persist, leading to data divergence or unintended business outcomes [1][6].

2) Lack of Strong Atomicity: Sagas relax the atomicity guarantee of traditional transactions. Since each local transaction commits independently, failures can leave the system in a partially updated state. Although compensations attempt to restore consistency, they cannot always reverse all effects (e.g., a package already shipped or a partial refund issued). As a result, it becomes harder to enforce strict correctness across workflows [1][4][5].

3) Idempotency and Retry Semantics: Sagas often rely on asynchronous communication and retries. Therefore, all participating services must implement idempotent operations to safely handle repeated requests. Without idempotency, retries or duplicate messages caused by network failures can result in double processing, data corruption, or financial discrepancies. Implementing safe retry logic typically requires tracking message identifiers and storing request histories, adding extra complexity to service design [2][6].

4) Debugging Distributed Workflows: Troubleshooting Sagas is more complex than debugging traditional, monolithic transactions. Failures may occur in isolated services, without a shared execution

context or traceable flow. If centralized logging, correlation IDs, or distributed tracing are not in place, it becomes difficult to determine which step failed or how to reproduce the issue. This challenge is amplified in autoscaled or containerized environments where service instances are ephemeral [2][13].

5) Observability and Tracing Limitations: Since Saga execution spans multiple services and often involves asynchronous events, strong observability is essential. Distributed tracing tools such as OpenTelemetry, Jaeger, or Zipkin help visualize the flow, but integrating and maintaining them across diverse services and teams remains a significant effort. Even when such tools are available, understanding and troubleshooting asynchronous flows can be difficult for operations teams [10][13].

6) Operational Overhead and Maintenance Cost: Implementing the Saga pattern typically requires additional infrastructure such as orchestration engines, message brokers, state stores, and monitoring systems. This expands the operational footprint and raises the cost of maintaining the system. It demands careful DevOps practices, robust CI/CD pipelines, and shared ownership across service teams. Without ongoing maintenance and governance, this complexity can lead to brittle workflows, outdated configurations, and hidden failure modes that erode system reliability over time [2][10][11].

VIII. FUTURE DIRECTIONS AND OPEN RESEARCH QUESTIONS

As the adoption of the Saga pattern continues to grow in cloud-native and microservices architectures, new challenges and opportunities emerge. The following research directions highlight ongoing efforts to improve Saga reliability, flexibility, and integration with evolving technologies.

1) Formal Verification of Saga Workflows: Formal verification involves applying mathematical and logical techniques to prove that a Saga workflow behaves correctly under all execution scenarios. In distributed systems, issues such as partial completion, failed compensations, or inconsistent final states can occur [1][4]. To address these risks, formal models are used to check properties like safety (ensuring undesirable outcomes never happen), liveness (ensuring the workflow eventually completes), and compensation correctness (verifying that rollback steps restore a valid state). This level of rigor is particularly important in domains like finance and healthcare, where system correctness is critical [1]. Researchers are developing formal models and using tools such as model checkers to validate Saga workflows, even when compensation logic is only partially specified at design time [13][16].

2) Intelligent Orchestration Using AI: Traditional Saga coordination typically relies on static rules that may not adapt well to dynamic system conditions. Incorporating artificial intelligence can introduce flexibility and improve fault tolerance. Current research investigates the use of:

- Reinforcement learning to predict and avoid transaction failures.
- Machine learning models to detect patterns in workflow failures and recommend recovery strategies.
- Context-aware compensation planning, where AI selects appropriate rollback actions based on failure type, historical outcomes, and service health.

3) Observability Tools and Standards: Monitoring and debugging Sagas is challenging due to their distributed, asynchronous nature. Failures may not be immediately visible, and workflow state may be difficult to trace. To improve transparency and reliability.

- Standardized observability schemas are being developed, often by extending frameworks like OpenTelemetry and OpenTracing.
- Advanced visualization tools are under development to depict the full lifecycle of Saga executions across services.
- Domain-specific languages (DSLs) are being proposed to express expected Saga behaviors, supporting automated diagnostics and validation rules [5][13].

These efforts aim to create shared standards and reusable tools that reduce cognitive overhead and operational risk.

4) Combining Saga with Blockchain or Serverless Computing: Emerging computing paradigms introduce both challenges and opportunities for the Saga pattern.

- **Blockchain integration:** Researchers are exploring how to use smart contracts to execute and record Saga steps, providing tamper-proof audit trails and decentralized coordination. This is particularly relevant in multi-party workflows where trust and transparency are essential [2].
- **Serverless computing:** Implementing Sagas in serverless environments introduces coordination challenges due to the stateless and ephemeral nature of serverless functions. Maintaining state, handling retries, and executing compensations require new patterns or external orchestration mechanisms [2][10].
- **Hybrid models:** Combinations of centralized orchestration with blockchain-backed state recording or serverless execution environments are being investigated. These hybrid approaches aim to balance scalability, resilience, and auditability.

IX. CONCLUSION

The Saga pattern has emerged as a practical solution for managing distributed transactions in microservice architectures. By decomposing long-lived transactions into smaller, compensatable local operations, it fits naturally within the decentralized and loosely coupled structure of modern systems. This review examined the theoretical foundations of the pattern, including both orchestration and choreography styles, and compared it with traditional approaches like the Two-Phase Commit (2PC) protocol, as well as related patterns such as Outbox and Event Sourcing.

Although Sagas offer improved scalability, resilience, and flexibility, especially in asynchronous environments, they also introduce trade-offs. The lack of atomicity, the complexity of implementing compensation logic, and challenges related to debugging and observability all add to the system's operational burden. These issues highlight the need for careful architectural planning and a deep understanding of domain requirements.

Sagas are not a universal solution. They are best applied in scenarios where eventual consistency is acceptable and where compensating actions can be reliably defined. Choosing to adopt the Saga pattern should be based on a thorough evaluation of the system’s consistency needs, fault tolerance, and business rules.

Looking forward, the evolution of the Saga pattern will be influenced by better tooling, improved observability standards, and integration with emerging technologies such as artificial intelligence, formal verification techniques, and blockchain-based coordination. Ongoing research and development in these areas will help make Sagas more robust, transparent, and easier to manage in real-world applications.

In conclusion, the Saga pattern provides a pragmatic approach to distributed transaction management. With an understanding of both its benefits and limitations, software architects and developers can apply it effectively to build systems that are resilient, scalable, and maintainable.

REFERENCES:

[1] H. Garcia-Molina and H. F. Korth, "Sagas," ACM SIGMOD Record, vol. 16, no. 3, pp. 249–259, 1987.
[2] M. Fowler, "Microservices," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html
[3] E. Brewer, "CAP twelve years later: How the ‘rules’ have changed," Computer, vol. 45, no. 2, pp. 23–29, Feb. 2012.
[4] J. Gray and A. Reuter, Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1992.
[5] W. Vogels, “Eventually Consistent,” Communications of the ACM, vol. 52, no. 1, pp. 40-44, Jan. 2009.
[6] C. Richardson, "Reliable messaging with the transactional outbox pattern," 2020. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html
[7] Red Hat, "Implementing the outbox pattern for event-driven architectures," 2021. [Online]. Available: https://developers.redhat.com/articles/2021/09/22

[8] M. Fowler, "Event sourcing," 2005. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html
[9] N. Sharma, "Workflow orchestration using Netflix Conductor," Medium, Mar. 1, 2025. [Online]. Available: https://nitish1503.medium.com/workflow-orchestration-using-netflix-conductor-1047cff467d9
[10] Camunda, "Agentic Orchestration & Automation Platform | Camunda," Camunda, Jun. 26, 2025. https://camunda.com/platform/
[11] Apache Kafka, "Apache Kafka." [Online]. Available: https://kafka.apache.org/
[12] Jacky, "Implementing the saga pattern with Spring Boot and ActiveMQ in microservice," DEV Community, Oct. 18, 2023. [Online]. Available: https://dev.to/jackynote/implementing-the-saga-pattern-with-spring-boot-and-activemq-in-microservice-14me
[13] Temporal, "Temporal TypeScript SDK API reference." [Online]. Available: https://nodejs.temporal.io/
[14] AxonIQ, "Sagas." [Online]. Available: https://docs.axoniq.io/axon-framework-reference/4.11/sagas/
[15] Particular Software, Sagas - NServiceBus. [Online]. Available: https://docs.particular.net/nservicebus/sagas/
[16] Ujedinya I. Kalu, "C# Saga Pattern with NServiceBus," Code Maze, 2023. [Online]. Available: https://code-maze.com/csharp-saga-pattern-with-nservicebus/