International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

A Review of Event-Driven Architecture Patterns Using Message Brokers in .NET

Serhii Yakhin*

Senior .NET Software Engineer at Growe
Email: serhii.yakhin@gmail.com

Abstract

This article describes different event-driven architecture (EDA) patterns that use message brokers to implement distributed computing in .NET. This article also describes data consistency and state management techniques in a distributed system architecture. In response to the industry trend of moving from monolithic architecture to microservice architecture for scalability, agility, and resilience, this project seeks to provide a thorough basis for decision-making for the adoption of EDA patterns and message brokers (like Apache Kafka, RabbitMQ, Azure Service Bus) based on non-functional requirements and organizational maturity levels. The methodological foundation is based on a Systematic Literature Review (SLR), ensuring reproducibility, completeness, and methodological rigor of the analysis. The scientific novelty of this work lies in integrating three analytical dimensions: theoretical architectural patterns, the technological implementation of message brokers, and the practical aspects of applying them within .NET systems. The main findings emphasize that the choice of EDA patterns and broker technologies is not a search for an optimal, universal solution but a deliberate balancing of technical and organizational factors. Successful use of EDA requires maturity and experience in DevOps and observability, as well as strict adherence to idempotency principles. Therefore, the target audience for this article is software architects, developers, and researchers concerned with microservices and event-driven systems of all kinds built on .NET, as well as technical decision-makers responsible for enterprise adoption of EDA.

Keywords: event-driven architecture; microservices; .NET; message brokers; Apache Kafka; RabbitMQ; Azure Service Bus; Saga pattern; Transactional Outbox pattern; data consistency; asynchronous communication.

Received: 9/26/2025
Accepted: 11/26/2025
Published: 12/5/2025

---

* *Corresponding author.*

1. Introduction

The shift from monolithic architectures to microservices has become the dominant trend in the design and development of contemporary enterprise systems [1]. This transformation is driven by a fundamental business need to accelerate time-to-market and to deliver high scalability and fault tolerance under continuously increasing workloads [2]. In this context, event-driven architecture (EDA) functions not merely as a methodological option but as a paradigmatic shift that enables the core microservice principles of loose coupling and asynchronous inter-component communication. The popularity of EDA in industry is rapidly increasing, as it allows for the creation of flexible, reactive systems that adapt to real-time changes [3].

Despite these evident advantages, the practical adoption of EDA entails significant technical and organizational challenges. Practitioners frequently confront a constellation of issues intrinsic to event management in distributed environments, including data consistency in the absence of traditional ACID transactions, failure handling, guarantees of message ordering, and operational idempotency [3]. The complexity of these concerns is sufficiently high that, according to published studies, many organizations reassess their decision to migrate to microservices when confronted with unforeseen difficulties. The central problem examined in this study is the absence of a systematized guide that tightly couples theoretical EDA patterns, concrete message-broker technologies, and the practicalities of their implementation within the .NET ecosystem. The selection of EDA is not merely a technical choice; it is a strategic move that requires revisiting development, testing, and operations practices and elevating the maturity of DevOps culture across the organization.

The purpose of this work is to develop a comprehensive analytical foundation for selecting and applying EDA patterns and message brokers in applications built on the .NET platform.

To achieve this objective, the following tasks were defined:

1. Systematize and classify key EDA patterns, focusing on resolving data-consistency concerns (Saga, Transactional Outbox) and state management (Event Sourcing, CQRS).
2. Conduct a comparative analysis of leading message-broker technologies (Apache Kafka, RabbitMQ, Azure Service Bus), identifying their architectural differences and performance trade-offs.
3. Analyze principal challenges associated with EDA adoption (eventual consistency, operational complexity, monitoring) and assess the role of specialized .NET libraries (MassTransit, NServiceBus) in addressing them.
4. Synthesize the results into a practice-oriented decision model for .NET architects.

The scientific novelty of this work lies in the integrated synthesis of three levels of analysis: (i) theoretical architectural patterns; (ii) concrete message-broker technologies with quantitative characteristics; and (iii) practical implementation aspects in .NET. Unlike existing works that consider these facets in isolation, this study proposes a holistic model that connects a project's non-functional requirements to specific architectural decisions. This synthesis constitutes an original contribution aimed at bridging the gap between theory and the practice of EDA adoption.

2. Materials and Methodology

The methodological basis of the study is a Systematic Literature Review (SLR). This method was selected to ensure completeness, objectivity, and reproducibility of the analysis by aggregating and synthesizing findings across a broad range of relevant scientific publications and technical sources. The primary search, following the PRISMA 2020 standard, was conducted across five bibliographic databases (Scopus, Web of Science, IEEE Xplore, ACM Digital Library, SpringerLink) and two additional discovery sources, as shown in Figure 1.

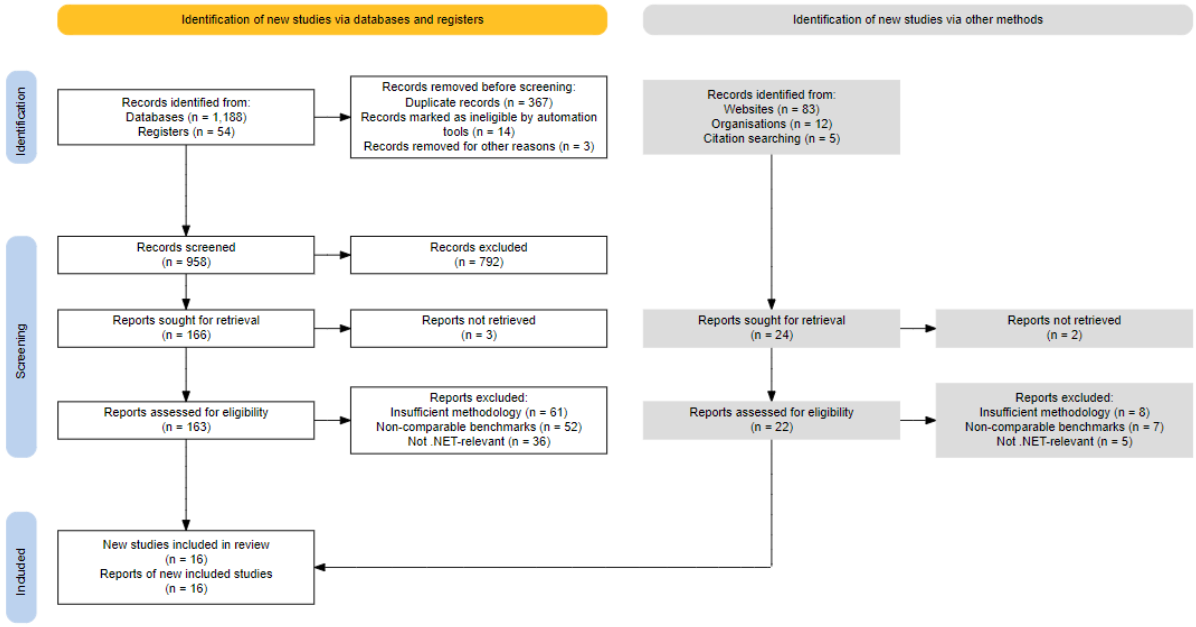


Figure 1: PRISMA Flow Diagram

In total, 1,342 records were identified: from databases, 1,188; registries, 54; websites, 83; organizational materials, 12; citation chaining, 5. Automated deduplication reduced the corpus by 367 items; a further 14 records were automatically excluded by date/language filters, and 3 by other technical reasons, resulting in 958 records proceeding to manual screening of titles and abstracts.

At the screening stage, 792 records were excluded according to pre-established criteria (non-EDA context; no .NET linkage; absence of data on consistency/state-management patterns or on brokers). Full-text assessments were requested for 166 reports; 3 could not be obtained. An additional 24 reports were located via alternative channels; 2 failed extraction. This left 163 items from the mainstream and 22 from other streams for full-text review.

Two reviewers screened full texts for eligibility determination and reliably rated titles and abstracts ($\kappa = 0.81$) and full texts ($\kappa = 0.78$). Grey literature searching identified 163 qualifiable full texts, and methodology inadequacies, benchmark incomparability, .NET data absences, or an editor's lack of empirics excluded 149. Of the 22 remaining texts, 20 were excluded based on AACODS' grey literature quality assessment. The final corpus comprised 16 sources aligned with the review objectives and used in the synthesis.

3. Results

3.1. Classification of EDA Patterns for Ensuring Data Consistency

One of the most challenging problems in distributed systems is guaranteeing atomicity for operations that update database state and emit a notification event. This dual-write problem arises because most contemporary technology stacks do not provide distributed transactions [4]. Within EDA, specialized patterns have been developed to address this issue.

3.1.1. Transactional Outbox Pattern: Guaranteeing Atomic Event Delivery

The Transactional Outbox pattern resolves the dual-write problem by coalescing two operations, updating the state of a business entity and persisting the outgoing event, within a single local database transaction [5]. As shown in Figure 2, instead of being sent directly to the message broker, the event is stored in a separate Outbox table inside the same database. It is then asynchronously polled via a background service (the dispatcher) that reads unsent events from the Outbox table and sends them to the message broker [6]. This ensures an event sends only after a business transaction completes successfully, ensuring atomicity across service boundaries.

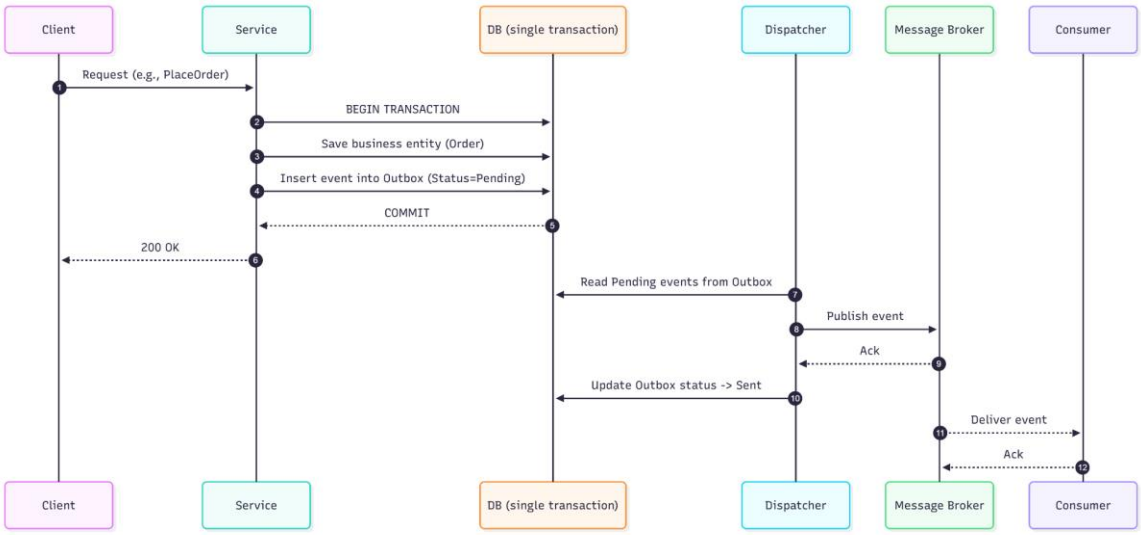


Figure 2: Transactional Outbox Pattern

In the .NET ecosystem, people commonly implement this pattern with the Entity Framework Core for transaction management and BackgroundService for the dispatcher. Industrial-grade libraries such as NServiceBus and MassTransit provide robust, production-hardened implementations that encapsulate the complexity of the pattern Reference [7]. For NoSQL databases like Azure Cosmos DB, transactional batches confined to a single logical partition are used to atomically persist both the business object and the event [8].

The principal advantage of the pattern is strict atomicity. Nevertheless, it introduces additional latency between commit and actual publication, necessitates management of Outbox table growth, and requires idempotent

consumers because delivery is guaranteed on an at-least-once basis [6].

In production settings, the Transactional Outbox confronts nontrivial challenges that extend well beyond the canonical “save and dispatch” pattern. High concurrency can cause contention over newly inserted outbox rows, leading to races, duplicate sends, and thundering herd effects. Effective designs combine optimistic concurrency (e.g., rowversion/ETag or compare-and-swap flags) with lease metadata (ClaimedBy/ClaimedAt/LeaseUntil) and bounded batch selection using skip-locked semantics; in relational stores this is commonly implemented via SELECT … FOR UPDATE SKIP LOCKED (often invoked from EF Core through FromSqlRaw). In Azure Cosmos DB, atomicity relies on ETags and partition-scoped TransactionalBatch, which in practice mandates co-partitioning the business aggregate and its outbox entry under the same partition key. Crucially, the Outbox guarantees atomic persistence of state and publication intent, not business-level exactly-once effects: transport delivery remains at-least-once, which necessitates idempotent consumers, a durable de-duplication store keyed by a stable message or business identifier with a bounded TTL window, and, where ordering matters, constrained parallelism per aggregate key. The dispatcher must page records, honor transport backpressure, apply exponential, jittered retries for transient faults, and quarantine poison records to a dead-letter facility; crashes between “send” and “mark-as-sent” require an idempotent publish protocol with broker acknowledgments and expiring leases. Unchecked outbox growth degrades latency and increases lock contention, thus time-based partitioning, TTL/archival, index hygiene, and carefully tuned batch sizes are integral to sustained performance.

Industrial-grade libraries encapsulate a substantial portion of this infrastructural complexity without claiming business-semantic exactly-once. In NServiceBus, a transactional session coordinates EF Core (or other persistence) with outgoing messages as a single logical commit, while inbox/outbox de-duplication at the handler boundary prevents reprocessing after retries or failover; the stack adds transport confirmations, immediate and delayed retry policies, DLQ routing, safe outbox cleanup, guidance for per-key ordering/partitioning, and saga persistence that coexists with the Outbox. MassTransit provides EF Core Outbox/Inbox middleware that captures outgoing events within the same database transaction for safe deferred dispatch, partitioners and concurrency limiters to ensure single-threaded processing per key at high aggregate throughput, configurable retry/redelivery pipelines with error queue handling, Quartz/delayed scheduling, utilities for outbox cleanup and re-send, and first-class transports for Kafka, RabbitMQ, and Azure Service Bus. These features provide package claiming and safe dispatch, idempotent publication, ordering, failure handling, and observability hooks, so teams can implement policies rather than reinvent plumbing.

Practically, robust deployments observe a small set of disciplines: persist business state and the outbox entry in a single transaction, run the dispatcher outside that transaction and make it strictly idempotent; use a stable sharding key per aggregate to preserve order without consumer-side reordering; instrument commit-to-publish latency, dispatcher lag (age of oldest unsent record), duplicate rate observed by consumers, and DLQ inflow; avoid invoking the broker within the database transaction and never mark records as “sent” before transport acknowledgment. Under these conditions, the Outbox delivers predictable, loss-free publication semantics without cascading duplicates.

3.1.2. Saga Pattern: Managing Distributed Transactions

The Saga pattern orchestrates long-running business transactions spanning multiple microservices. A saga is a sequence of local transactions; each step updates state within a single service and publishes an event to trigger the next step [9]. In the event of failure at any stage, the saga initiates compensating transactions in reverse order to undo prior actions semantically.

In practice, sagas can be implemented in one of two ways: orchestration or choreography. Both orchestrations and choreographies coordinate a series of local transactions and compensations, but differ in how control flow is expressed and where the business-process knowledge is stored.

A central orchestrator coordinates the saga in orchestration-based sagas and makes the necessary calls to the services involved. Based on the services' responses or domain events, the orchestrator determines the current state, maintains the saga state, triggers state transitions, sets up timers, and executes compensations. This approach centralizes process logic in a single component, simplifying change management, monitoring, and auditability: the lifecycle of each business instance is explicit and queryable. Typical trade-offs include the risk of a single logical point of failure, potential bottlenecks at very high throughput if not scaled, and tighter coupling of services to the orchestrator's command schema. In .NET, orchestration is commonly implemented with frameworks that provide saga state machines and durable timers (e.g., MassTransit or NServiceBus), where the orchestrator is a dedicated consumer/handler with persisted saga state and correlation identifiers [10]. An orchestration-based Saga approach flow diagram is shown in Figure 3.

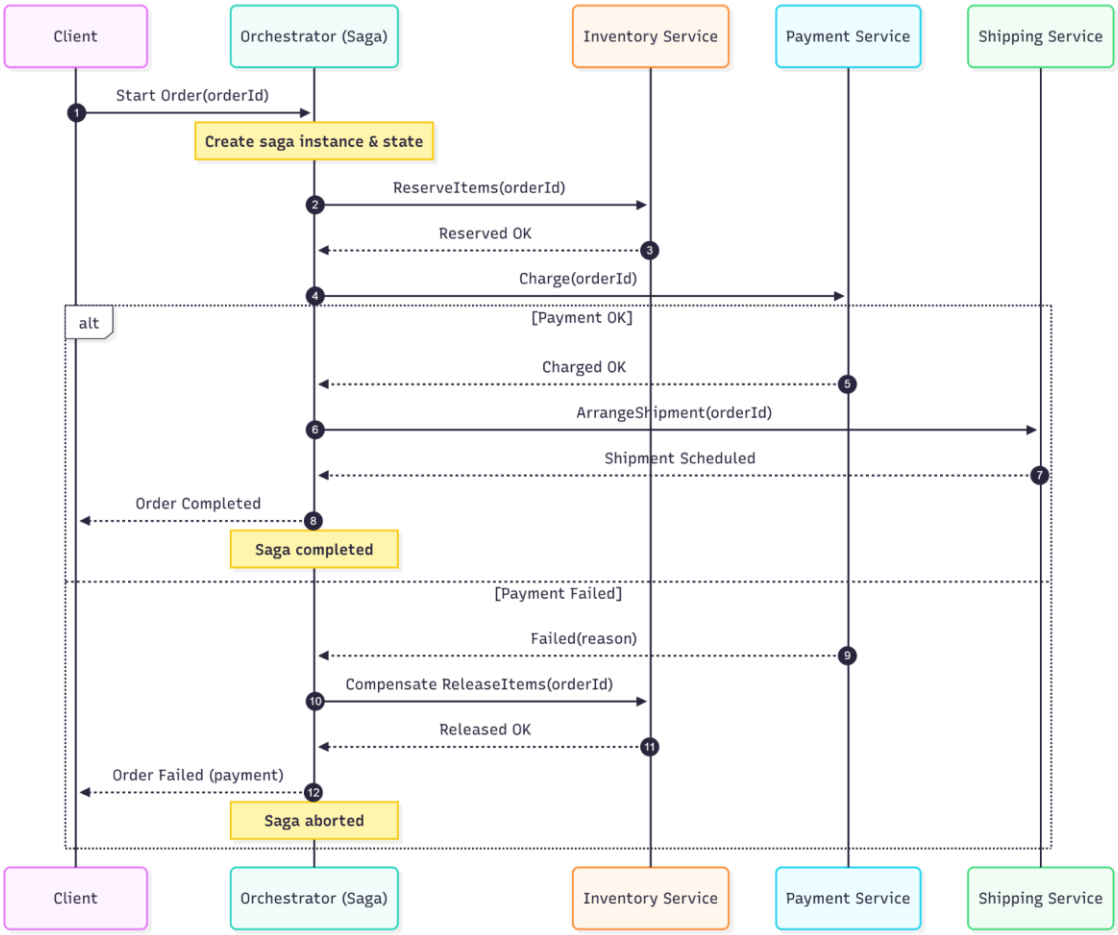


Figure 3: An Orchestration-based Saga Approach

A choreography-based saga relies on decentralized publish–subscribe interactions: each service reacts to domain events, performs its local transaction, and publishes the next event that other services may consume. No single component owns the overall control flow; the process emerges from event subscriptions and handlers. This gives loose coupling and horizontal scalability, minimizes the need for a central coordinator, and often minimizes end-to-end latency for simple flows. However, it introduces complexity at global visibility and governance, such as end-to-end tracing, compensation behavior across services, and the need to coordinate versioning across multiple consumers when the process changes (for example, a service is added to the process). Strongly observe with correlation IDs or trace in a distributed fashion. Handle events idempotently, and contract events clearly. .NET shops often choreograph by using a message broker like Kafka, RabbitMQ, or Azure Service Bus. Consumers subscribe to domain events and publish more events after the local transaction commits. A choreography-based Saga approach flow diagram is shown in Figure 4.

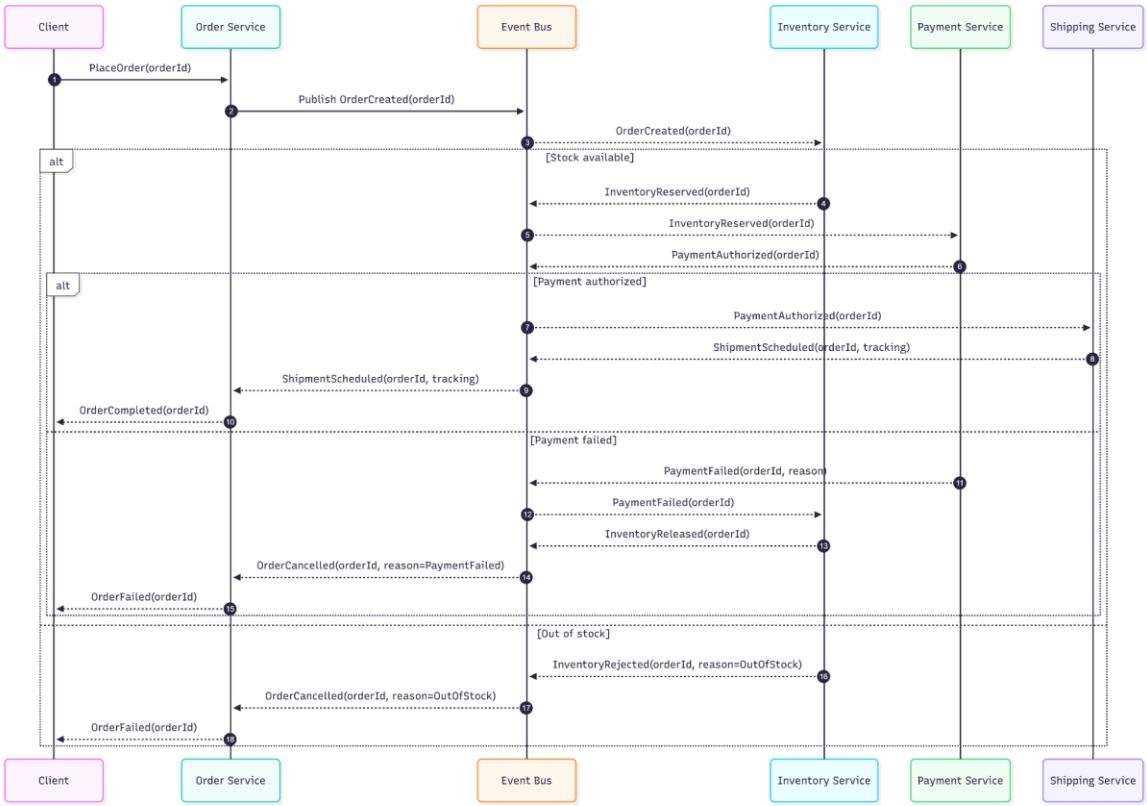


Figure 4: A Choreography-based Saga Approach

A key limitation of sagas is the absence of isolation, one of the ACID properties. Intermediate changes produced by successful local transactions may become visible to other processes before the saga completes [9]. Consequently, sagas provide Atomicity, Consistency, and Durability (ACD), but not Isolation [11]. A comparative description of the Saga and Transactional Outbox patterns is presented in Table 1.

Table 1: Comparative characteristics of the Saga and Transactional Outbox templates

Criterion	Transactional Outbox pattern	Saga pattern
Primary purpose	Guarantee atomic delivery of an event when a single service's state changes.	Manage long-running, distributed business transactions that span multiple services.
Consistency model	Atomicity (local). Ensures the update and the event are either both committed or both rolled back.	Eventual consistency. Ensures the system will eventually reach a consistent state.
Isolation (ACID)	Provided at the local database level.	Absent. Intermediate changes may be visible to other transactions.
Implementation complexity	Moderate (requires a background process and management of an Outbox table).	High (requires designing compensating transactions; debugging is more complex).
Performance impact	Minimal. Adds a row to a table and a slight publication delay.	Low (non-blocking). More performant than the two-phase commit (2PC) but incurs overhead for orchestration/choreography.
Typical .NET scenario	Send a notification (e.g., OrderCreated) after saving the order to the database.	Order processing: reserve inventory, process payment, create shipment.

Empirical studies indicate that, under normal conditions, sagas exhibit approximately 30% lower latency than 2PC [12].

3.2. State-Management and Dataflow Patterns

Beyond consistency concerns, EDA includes patterns that fundamentally reshape how system state and data streams are handled.

Event Sourcing (ES) is a pattern in which an application's state is defined not by current database values but by the complete sequence of immutable events that led to that state [13]. Instead of executing these update or delete commands, it generates events like OrderPlaced or ItemShipped. The state of the system can be recovered from the corresponding event stream at any time. Some additional advantages of the event-sourcing pattern include full

auditability and traceability, state recovery at any time, and easier debugging. Disadvantages include difficulties with event schema versioning and decreased performance as more events are replayed to recreate aggregate state Reference [14]. The flow diagram of this pattern is shown in Figure 5.

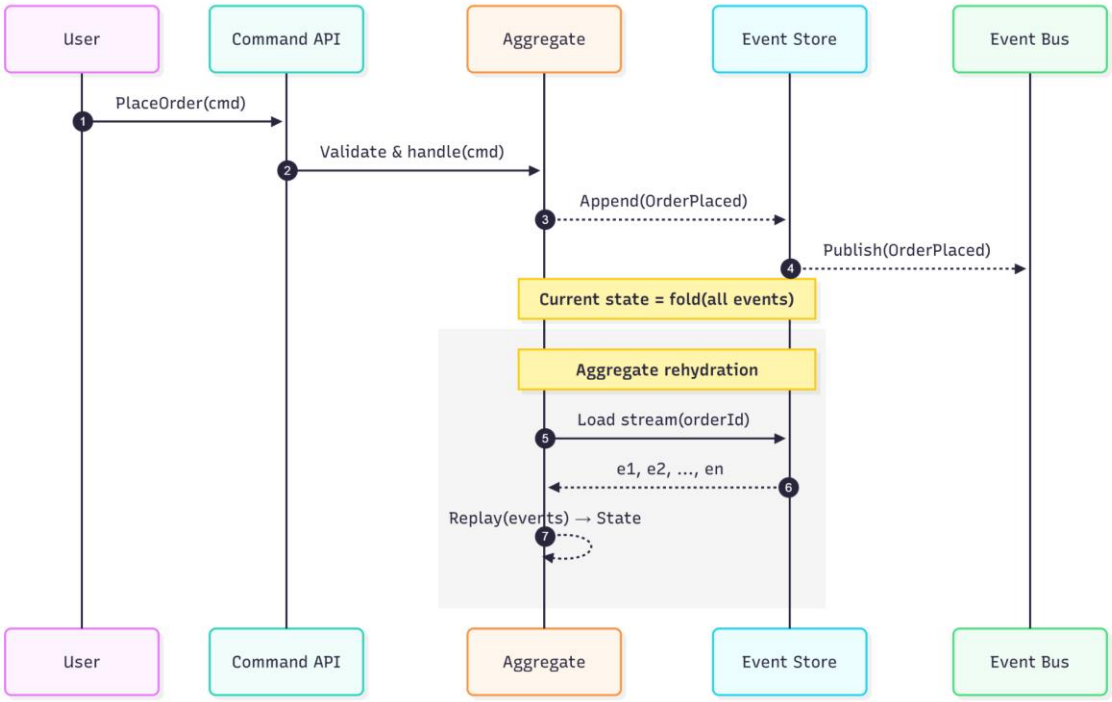


Figure 5: Event Sourcing Pattern

Command Query Responsibility Segregation (CQRS) uses separate models for Command and Query operations, potentially including separate data stores. Command operations change the application state, they do not return data. Query operations return data and do not mutate state. This enables optimizing each model for its purpose (e.g., a normalized write model and denormalized, query-optimized read models). Benefits include independent scaling of read and write paths, as well as improved performance and security. The principal drawback is increased architectural complexity and the need to synchronize models, which introduces eventual consistency [15]. The flow diagram of this pattern is shown in Figure 6.

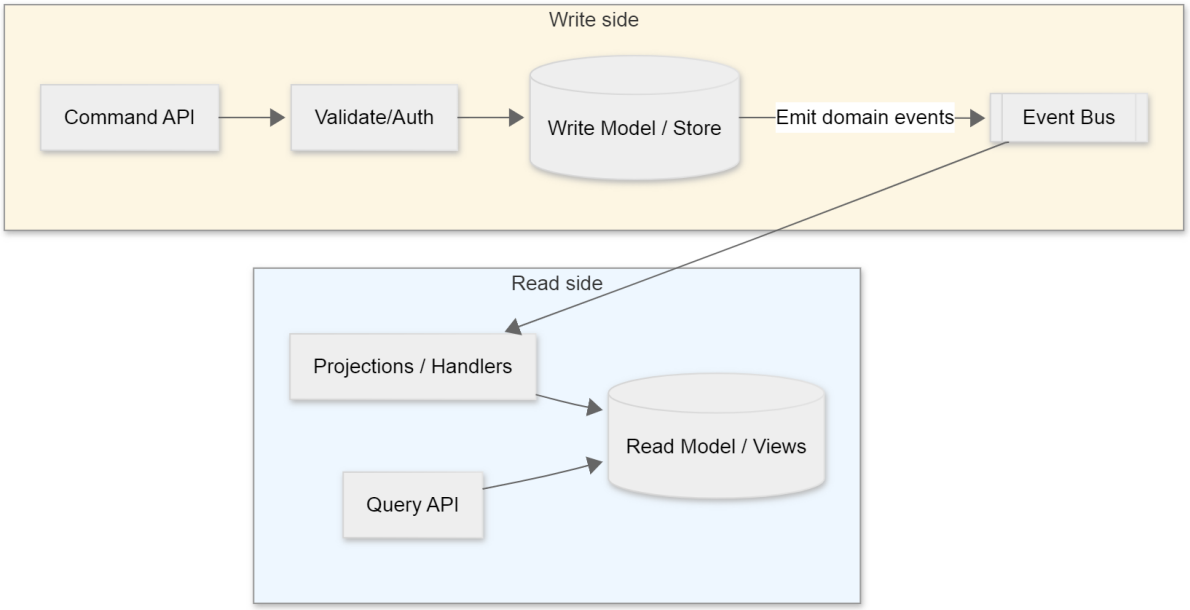


Figure 6: Command Query Responsibility Segregation Pattern

These two patterns often operate in synergy, as shown in Figure 7: the event stream from Event Sourcing (write model) feeds updates to the denormalized CQRS read models.

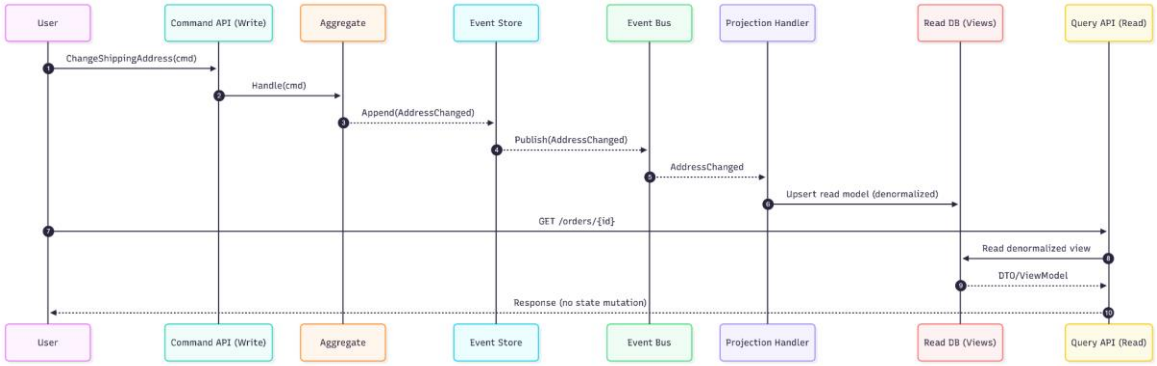


Figure 7: Synergy of Event Sourcing Pattern and Command Query Responsibility Segregation Pattern

3.3. Comparative Analysis of Message Brokers in the .NET Ecosystem

Selecting a message broker is among the most consequential architectural decisions in an EDA system. The leading technologies differ in their foundational architecture, which directly shapes performance, scalability, and usage scenarios. Figure 8 illustrates two primary architectural models.

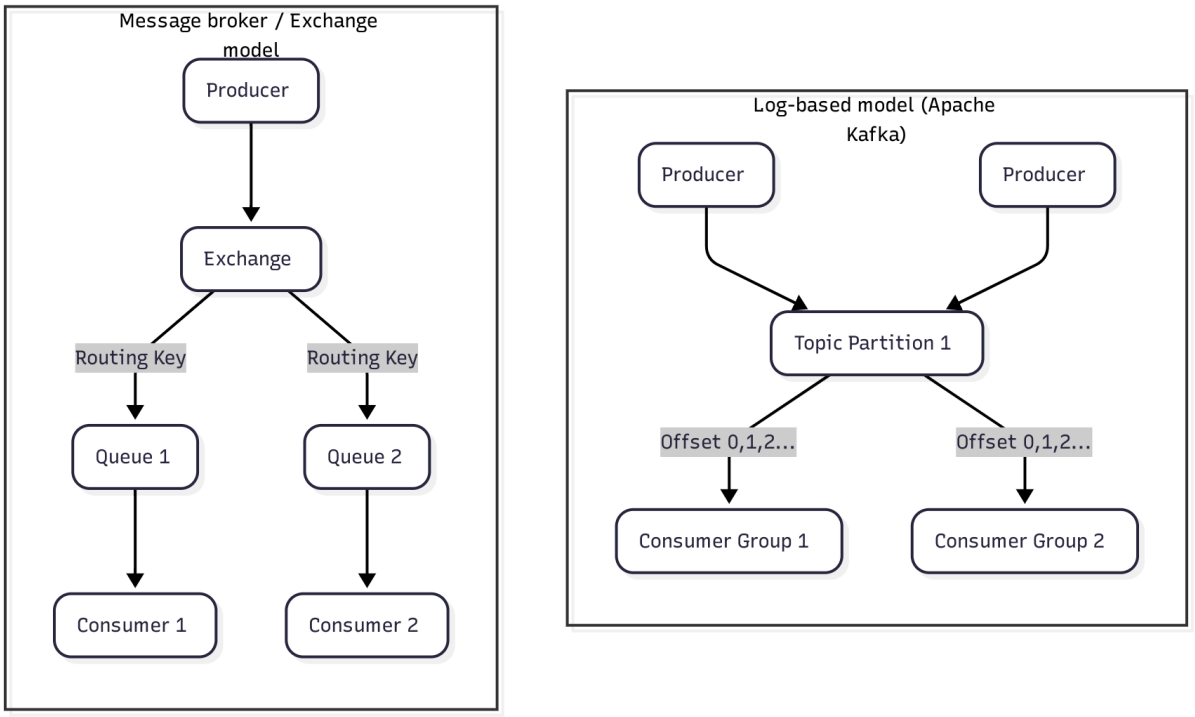


Figure 8: Architectural models of message brokers

The log-based model, as implemented by Apache Kafka, is an immutable, ordered event log. Consumers track their own position (offset), allowing multiple consumer groups to independently reread the same data. The queue-based model, used in RabbitMQ and Azure Service Bus, delegates delivery management to the broker; messages are consumed from queues and then removed. A detailed comparison of three leading brokers is provided in Table 2.

Table 2: Comparative analysis of message brokers

Characteristic	Apache Kafka	RabbitMQ	Azure Service Bus (Premium)
Architecture model	Distributed, replicated log (streaming platform).	Message broker (AMQP) with exchanges and queues.	Message broker with queues and topics (subscriptions).
Throughput	Very high (hundreds of thousands to >1.2M messages/sec). Optimized for streaming.	Medium/High, depending on configuration and routing complexity.	High scales via Messaging Units (MUs).
Latency	Low (p95 ≈ 18 ms), but can be higher than RabbitMQ in point-to-point scenarios.	Very low, especially for transactional messages.	Low, optimized for enterprise scenarios.
Scalability	Horizontal: add brokers and partitions. Excellent scalability.	Horizontal (clustering), but it is more complex to configure and has limitations.	Vertical and horizontal (via MUs); managed by Azure.
Delivery guarantees	At-least-once (default); Exactly-once with transactions.	At-least-once; At-most-once.	At-least-once; At-most-once. Supports duplicate detection.
Protocols	Proprietary binary protocol over TCP/IP.	AMQP 0-9-1, MQTT, STOMP.	AMQP 1.0, SBMP (proprietary).
Operational complexity	High, requires cluster management expertise (ZooKeeper / KRaft).	Moderate, cluster, and policy management required.	Low (PaaS), fully managed by Azure.

Overall, the table shows that the architectural and operational characteristics of the brokers can vary widely. Apache Kafka is a distributed replicated log broker with extreme throughput and horizontal scaling via brokers and partitioning, but meaningful operational overhead in cluster management. RabbitMQ is an AMQP broker with complex routing, low latency, and support for multiple protocols. It is suited to topological routing and transactional use cases but suffers from meaningful scaling overhead. Azure Service Bus (Premium) is a highly available PaaS broker with duplicate detection and hybrid scaling using Messaging Units. It is intended for enterprise message broker use cases with low operational overhead, but it incurs the cost of cloud provider lock-in. To summarize, select the broker based on the use case: Kafka for streaming and extreme throughput, RabbitMQ for topological routing and transactional messaging, and Azure Service Bus (Premium) for enterprise integration with minimal operational effort.

4. Discussion

4.1. Decision Model for Selecting Architectural Patterns in .NET

The results indicate that selecting patterns and technologies in EDA is not about identifying a universally best solution but about navigating a landscape of trade-offs. High performance and functional richness often accrue at the cost of increased operational and cognitive complexity. For example, Apache Kafka delivers top-tier throughput but typically requires a team of several specialists for reliable operations (on average, 2.3 FTE) [16]. Similarly, the Saga pattern enables coordination of complex distributed processes while shifting the burden of compensating logic design and debugging to developers [12].

This observation implies that the most critical failure mode in EDA adoption is underestimating complexity. Technology choices must be grounded not only in technical requirements but also in organizational capabilities and maturity, including team skills and the state of DevOps practices. Based on the analysis, an authorial decision model in the form of a flowchart (Figure 9) is proposed to guide architects systematically through key design questions.

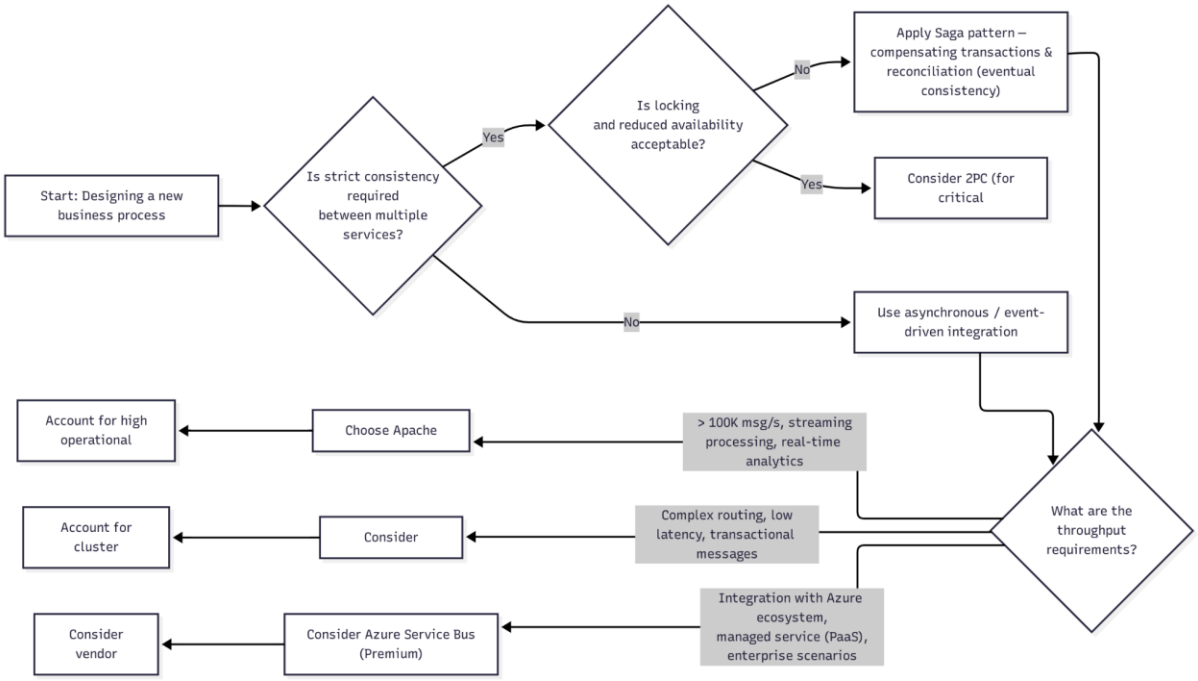


Figure 9: Flowchart of the decision-making model for .NET architects

The model begins with the fundamental question of transaction boundaries. If a business process requires consistent changes across multiple services, the choice lies between Saga and 2PC. The decision regarding the acceptability of eventual consistency (Saga) versus the necessity of strict consistency (2PC) is less a technical judgment than a business determination, to be made jointly with product owners based on risk analysis and user-experience requirements for the specific process. This often leads to hybrid architectures, wherein 2PC is used for mission-critical operations (e.g., balance updates in financial systems), while Saga governs ancillary flows. If

atomicity is required within a single service, the simpler and more reliable Transactional Outbox is preferred. Non-functional performance requirements and the operational model determine the subsequent broker choice.

4.2. Key Challenges and Mitigation Strategies

First, eventual consistency is not a defect but a constitutive property of most EDA systems [13]. Mitigation strategies include designing user interfaces that faithfully surface intermediate states (e.g., payment in progress) and implementing mechanisms to detect and resolve conflicts.

Second, idempotent consumers are mandatory. Because most brokers provide at least one delivery, event consumers must be idempotent, ensuring that duplicate processing does not yield adverse side effects such as record duplication or double charging. In .NET, this is commonly achieved by tracking processed message identifiers in persistent storage (e.g., Redis or a database table).

Third, ordering and versioning are critical. Strict event-ordering guarantees are essential for many business processes. Apache Kafka ensures order within a topic partition, whereas other systems may require additional consumer-side logic. An event-schema versioning strategy, which allows for performing changes to events in an evolutionary manner while remaining backward compatible.Fourth, asynchrony materially increases operational complexity and observability. Distributed tracing (e.g., OpenTelemetry) is needed to correlate the processing that a single business request undergoes across multiple services. However, high-throughput systems like Kafka require expertise and resources to operate reliably at scale [16].

Fifth, the role of .NET libraries (MassTransit, NServiceBus) is important. Such libraries can reduce the burden on developers by providing a high-level abstraction over brokers, encapsulating advanced design patterns (such as Sagas and Outbox) and providing abstractions over error-handling logic (such as Dead-Letter Queues). This allows a team to focus on business logic rather than low-level interactions with the broker. However, these abstractions can limit access to broker-specific capabilities.

Beyond reduced access to broker-specific capabilities, adopting high-level messaging libraries introduces several additional liabilities. Licensing and cost models may be nontrivial: commercial offerings can entail per-service, per-environment, or tiered fees, while open-source alternatives shift costs to internal maintenance, governance, and security response. Vendor support and project vitality are uneven; organizations assume risks related to the maintainer’s roadmap, release cadence, and responsiveness to CVEs, as well as the possibility of breaking changes or delayed compatibility with new .NET or broker versions. Abstractions also create a form of framework lock-in: core message contracts, saga state persistence, and pipeline behaviors become coupled to the library’s programming model, raising migration costs and constraining architectural evolution.

Operationally, additional layers create new points of failure and complicate debugging and performance tuning. Middleware pipelines, serializer choices, retry/redelivery policies, and background schedulers introduce performance and state complexities that must be tested and monitored. Diagnosable problems with these components often require low-level knowledge of the library's implementation and the underlying transport. Some features remain “leaky”: advanced routing, partitioning, transactions, or exactly-once semantics often degrade to

the "least common denominator," making it harder to exploit native broker strengths or cloud-managed capabilities. In regulated environments, delegating critical flow control to third-party components may raise compliance and auditability concerns, including provenance of binaries, SBOM requirements, and long-term support commitments.

Finally, the convenience of encapsulation can encourage centralization of cross-cutting logic in the bus layer, filters, policies, compensations, blurring domain boundaries and increasing cognitive load. Testability may suffer when behaviors are distributed across generated infrastructure or background agents, necessitating elaborate test harnesses and contract tests to attain confidence. Table 3 summarizes key challenges and mitigation strategies.

Table 3: Key challenges and mitigation strategies

Domain	Risks	Recommended Actions
Licensing and Cost	Non-trivial fee structures for commercial offerings (per service, per environment, tiered); for OSS, costs shift to internal maintenance, governance, and security response.	Perform total-cost-of-ownership analyses; budget explicitly for licensing or internal support; establish clear licensing compliance policies.
Vendor/Project Vitality	Dependence on the maintainer's roadmap and release cadence; uneven responsiveness to CVEs; lagging compatibility with new .NET or broker versions; potential breaking changes.	Assess project health and community signals; secure support SLAs; maintain upgrade plans and compatibility testing pipelines.
Framework Lock-in	Coupling of message contracts, saga state persistence, and pipeline behaviors to the library's programming model, raising migration costs and constraining architectural evolution.	Define explicit exit strategies; minimize coupling via adapter layers and bounded contexts; segregate domain contracts from transport concerns.
Operational Complexity	Additional failure points and harder debugging/performance tuning due to middleware, serializers, retry/redelivery policies, and background schedulers.	Standardize configurations; implement robust diagnostics and tracing; conduct load and chaos testing; document operational runbooks.
Leaky Abstractions	Advanced broker capabilities (routing, partitioning, transactions, exactly-once) may be reduced to a least-common-denominator feature set.	Where broker-specific strengths are critical, use native clients or targeted integrations; document criteria for bypassing the abstraction.

Compliance and Auditability	Delegation of flow control to third-party components can raise concerns regarding provenance, SBOM requirements, and long-term support commitments.	Enforce software supply-chain controls; require SBOMs; set minimum support horizons; integrate vulnerability management and attestations.
Centralization of Cross-Cutting Concerns	Bus-layer accumulation of filters, policies, and compensations can blur domain boundaries and increase cognitive load.	Keep cross-cutting logic minimal and well-scoped; preserve domain boundaries; prefer domain-level policies over bus-level orchestration.
Testability	Behaviors distributed across generated infrastructure and background agents require elaborate harnesses and contract tests to achieve confidence.	Invest in contract testing and reproducible fixtures for sagas/outbox; simplify pipelines; provide hermetic test environments.
Overall Implication	Libraries deliver value but shift trade-offs toward disciplined dependency management and explicit performance, cost, and supportability objectives.	Formalize SLOs/SLA; review alignment periodically; maintain documented migration paths and governance for dependency updates.

These factors do not negate the value of mature libraries. Still, they shift the trade-off toward disciplined dependency management, explicit exit strategies, and rigorous SLOs for performance, cost, and supportability.

5. Conclusion

In the conducted work, all objectives have been achieved. The research systemizes and classifies key patterns of EDA, compares the most dominant technologies in the message broker space, and finally describes the main problems encountered when adopting EDA practices in the .NET ecosystem.

The core conclusions thus are the following:

1. Saga and Transactional Outbox are appropriate to use on different types of boundaries. The Transactional Outbox is most appropriate for keeping data consistent within a single service. At the same time, the Saga is appropriate for coordinating a set of services for a business transaction with multiple steps.

2. Message-broker technology comprises a trade-off between streaming throughput (Apache Kafka), routing flexibility and low latency (RabbitMQ), and operational simplicity with cloud-native integration (Azure Service Bus).

3. EDA can be difficult to achieve without not only deep technical skills but also other organizational capabilities, such as the ability to deal with distributed-system complexity, investment in monitoring and observability, and a

willingness to accept eventual-consistency semantics.

The main practical contribution of this work is the decision model and the tables, which form the basis of a concrete decision-support method for .NET architects and developers that supports systematic, knowledge-based architectural decisions when designing and developing modern distributed applications.

References

[1]. Auer F, Lenarduzzi V, Felderer M, Taibi D. From monolithic systems to Microservices: An assessment framework. Information and Software Technology. 2021 Sep;137:106600.

[2]. Hassan H, Abdel-Fattah MA, Mohamed W. Migrating from Monolithic to Microservice Architectures: A Systematic Literature Review. International Journal of Advanced Computer Science and Applications. 2024;15(10).

[3]. Laigner R, Almeida AC, Zhou Y. An Empirical Study on Challenges of Event Management in Microservice Architectures. Arxiv. 2024 Aug 1.

[4]. AWS. Transactional outbox pattern [Internet]. AWS. 2025 [cited 2025 Oct 3]. Available from: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/transactional-outbox.html

[5]. Braun S, Deßloch S, Wolff E, Elberzhager F, Jedlitschka A. Tackling Consistency-related Design Challenges of Distributed Data-Intensive Systems - An Action Research Study. Arxiv. 2021 Aug 8.

[6]. Richardson C. Microservices Pattern: Transactional outbox [Internet]. Microservices. [cited 2025 Oct 5]. Available from: https://microservices.io/patterns/data/transactional-outbox.html

[7]. Microsoft Learn. Data platform for mission-critical workloads on Azure - Azure Architecture Center [Internet]. Microsoft Learn. 2025 [cited 2025 Oct 6]. Available from: https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/containers/aks-mission-critical/mission-critical-data-platform

[8]. Microsoft Learn. Transactional Outbox pattern with Azure Cosmos DB [Internet]. Microsoft Learn. [cited 2025 Oct 7]. Available from: https://learn.microsoft.com/en-us/azure/architecture/databases/guide/transactional-outbox-cosmos

[9]. Daraghmi E, Zhang CP, Yuan SM. Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture. Applied Sciences. 2022 Jun 19;12(12):6242.

[10]. Lungu S, Nyirenda M. Current Trends in the Management of Distributed Transactions in Micro-Services Architectures: A Systematic Literature Review. Open Journal of Applied Sciences. 2024;14(9):2519–43.

[11]. Fan P, Liu J, Yin W, Wang H, Chen X, Sun H. 2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform. Journal of Cloud Computing. 2020 Jul 23;9(40).

[12]. View of Optimizing Distributed Transactions in Banking APIs: Saga Pattern vs. Two-Phase commit (2PC). The American Journal of Engineering and Technology. 2025;7(06).

[13]. Özkan O, Babur Ö, van den Brand M. Domain-Driven Design in software development: A systematic literature review on implementation, challenges, and effectiveness. Journal of Systems and Software. 2025 Dec;230:112537.

[14]. Alshikh H. Evaluation and Use of Event-Sourcing for Audit Logging [Internet]. 2023 [cited 2025 Sep 13]. Available from: https://reposit.haw-hamburg.de/bitstream/20.500.12738/16034/1/BA_Evaluation_Use_of_Event-Sourcing.pdf

[15]. Caturbawa IGNB, Asri SA, Suasnawa IW, Sapteka AANG, Saptarini H, Yasa KA. Implementation of Command Query Responsibility Segregation (CQRS) in the Competency Test Information System. Proceedings of the 4th International Conference on Applied Science and Technology on Engineering Science. 2021 Jan 1;1322–6.

[16]. Arafat J, Tasmin F, Poudel S. Next-Generation Event-Driven Architectures: Performance, Scalability, and Intelligent Orchestration Across Messaging Frameworks. Arxiv. 2025.