Journal of Computer Science and Technology Studies
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

JCSTS
AL-KINDI CENTER FOR RESEARCH AND DEVELOPMENT

| RESEARCH ARTICLE

Event-Driven Architecture in Financial Systems: Performance Metrics and Resilience Patterns from the Real Wallet Platform

Srikanth Pulicherla
Independent Researcher, USA
Corresponding Author: Srikanth Pulicherla, **E-mail**: dev.srikanthp@gmail.com

| ABSTRACT

This article demystifies Event-Driven Architecture (EDA), a paradigm where software components communicate through event production and consumption rather than direct requests. The introduction explores EDA's conceptual foundations, historical evolution, and value proposition compared to traditional request-response models. Core components and patterns are examined, including event producers/consumers/brokers, event streams/stores, Command Query Responsibility Segregation, event sourcing, and messaging patterns. Implementation considerations cover technology stack selection, schema design, consistency challenges, error handling, and observability requirements. A detailed case study of the Real Wallet platform illustrates EDA principles in action, demonstrating how a commission payment system orchestrates complex financial workflows across specialized subsystems while maintaining loose coupling, scalability, and resilience. The article presents EDA as an architectural methods particularly suited for modern distributed systems requiring adaptability, fault tolerance, and independent evolution of components.

| KEYWORDS

Event-Driven Architecture, Distributed Systems, Message Patterns, Microservices, System Resilience.

| ARTICLE INFORMATION

ACCEPTED: 03 October 2025 **PUBLISHED:** 06 October 2025 **DOI:** 10.32996/jcsts.2025.7.10.17

1. Introduction to Event-Driven Architecture

The software engineering landscape has witnessed a fundamental paradigm transformation with the emergence of Event-Driven Architecture (EDA), which radically alters interaction patterns among components in contemporary distributed frameworks. EDA introduces a novel communication framework wherein system elements interact through discrete event notifications—specific signals indicating meaningful occurrences or state alterations within the application domain. These notification mechanisms eliminate the necessity for direct component awareness, instead facilitating state transmission and significant activity propagation across the entire application ecosystem. The architectural foundation supporting EDA comprises critical messaging structures, including subscription-publication models, communication channels, and event-responsive processors, collectively establishing the infrastructure necessary for creating systems characterized by minimal coupling and autonomous evolutionary capabilities [1].

Tracing the conceptual origins of EDA reveals roots extending to early distributed computing innovations during the 1970s-1980s period, though widespread implementation remained limited until the proliferation of sophisticated web architectures and microservice paradigms in the early 2000s. The progressive decomposition of monolithic software structures into discrete, independently deployable functional units exposed the inherent limitations of tightly integrated request-response interaction models. This architectural evolution catalyzed the development of specialized message-oriented intermediary technologies designed to support non-blocking communication between distributed system elements. The event-centered approach manifested as an elegant solution addressing persistent challenges in distributed computing: maintaining consistent data states, orchestrating multifaceted operational sequences, and ensuring system durability within increasingly fragmented computational environments where conventional synchronous methodologies demonstrated significant inadequacies [2].

Conventional request-response architectural models typically implement blocking communication patterns requiring clients to directly engage specific services while awaiting processing completion. Despite offering straightforward implementation advantages, this methodology creates substantial inter-component dependencies, necessitating comprehensive knowledge regarding service endpoints and expected response structures. Conversely, EDA establishes separation between event generators and processors through specialized intermediary routing mechanisms. This architectural distinction significantly enhances system adaptability, as individual components require familiarity only with standardized event formats rather than comprehensive knowledge of the complete system architecture. The isolation mechanisms inherent in event channel implementation provide enhanced failure containment and graceful performance degradation during partial system disruptions—characteristics proving especially valuable within extensive distributed environments where component failures represent anticipated operational conditions rather than exceptional circumstances [1].

The strategic advantages of EDA transcend purely architectural considerations, delivering substantive benefits across multiple operational dimensions. The reduced coupling between system elements allows differential evolutionary pacing among services without necessitating coordinated modifications throughout the entire application landscape. Using an asynchronous processing model leads to better utilization of resources and allows independent scaling of each component according to its processing requirements. Lightweight event contracts and an overall reduced coupling between services for asynchronous communication greatly improve long-term maintainability by allowing it to modify, expand, and swap out any one part of the system without jeopardizing the system's overall integrity. This architecture complements applications that are required to adapt constantly based on changing business needs, while providing operational and performance stability according to varying processing needs [2].

Component	Key Characteristics	Implementation Considerations
Event Producers	Generate discrete notifications in response to state changes; remain unaware of consumers; maintain separation of concerns	Domain-specific event generation; event schema design; validation and enrichment; idempotent event production
Event Brokers	Route events between producers and consumers; guarantee ordered delivery; provide persistence; manage subscription patterns	Technology selection (Kafka, RabbitMQ, EventBridge); partition strategy; throughput capacity; disaster recovery mechanisms
Event Consumers	Subscribe to specific event types; process events independently; implement domain-specific business logic	Idempotent processing; scaling based on event volume; replay capabilities; dead-letter handling
Event Streams/Stores	Maintain time-ordered sequences; support multiple concurrent consumers; provide immutable event history; enable system reconstruction	Retention policies; partitioning strategies; schema evolution; snapshot mechanisms; data governance
Observability Infrastructure	Trace event flows across service boundaries; monitor processing backlogs; track latency and throughput; correlate distributed logs	Correlation ID propagation; custom dashboards; anomaly detection; business-level metrics; performance alerting

Table 1: Core Components of Event-Driven Architecture: Implementation Characteristics. [1, 2]

2. Essential Elements and Patterns of Event-Driven Systems

The structure of event-driven architectures describes several dependent elements that combine to provide non-blocking interaction and processing. The three main players in the event-driven interaction architecture are producers, consumers, and intermediaries. Originators create notifications following status modifications or notable occurrences within their operational domain, translating business-significant changes into discrete notification records. These notification sources function without awareness of potential interested parties, thereby maintaining strict operational separation. Recipients, by contrast, register interest in specific notification categories and implement domain-specific processing logic upon receiving relevant notifications, handling them according to individualized requirements. Positioned between these endpoints exists the intermediary mechanism—specialized middleware infrastructure receiving notifications from originators and directing them toward appropriate recipients. This intermediary assumes responsibility for dependable message transmission, frequently offering assurances regarding sequential delivery, persistence capabilities, and precisely-once processing guarantees essential for maintaining system coherence. Contemporary notification-based service architectures leverage these fundamental building blocks to establish frameworks wherein services interact predominantly through notifications rather than explicit instructions, substantially enhancing resilience characteristics and evolutionary adaptability [3].

Notification sequences and repositories represent complementary methodologies for managing temporal notification data. Notification sequences provide uninterrupted, chronologically arranged notification series that recipients process with negligible delay. These sequences accommodate numerous independent recipients processing at individualized rates, enabling concurrent handling and specialized processing of identical notification content across diverse system elements. Notification repositories, alternatively, maintain notifications as unalterable records documenting all system modifications. This preservation establishes comprehensive operational histories and enables complete system reconstruction from any historical reference point. The integration of streaming capabilities with persistent storage creates the foundation for sophisticated information processing patterns while guaranteeing complete information preservation despite transient system disruptions or processing interruptions. The temporal characteristics inherent in these notification sequences naturally correspond with real-world business process execution, rendering notification-oriented systems exceptionally suitable for modeling intricate domains with extensive historical context. Notification sequences frequently implement segmentation strategies preserving sequential integrity within specific aggregations while facilitating horizontal scalability throughout the broader system landscape [3].

Operation Distinction represents a sophisticated architectural methodology complementing notification-oriented design through explicit separation between information retrieval and modification operations. Within this pattern, modification instructions and information requests operate against distinct data representations optimized for their specific functions. This separation acknowledges the inherent asymmetry between modification and retrieval operations across most applications—modifications typically incorporate complex validation mechanisms and business rules, while retrievals frequently require specialized views optimized for particular usage scenarios. When integrated with notification-oriented principles, Operation Distinction enables clear separation between authoritative notification sources and various specialized retrieval models derived from these notifications. Each retrieval model undergoes independent optimization for specific query requirements, enhancing performance characteristics and simplifying retrieval mechanisms while preserving singular authority regarding system state modifications. This architectural approach delivers particular value within complex domains where structures optimized for modification differ substantially from those required for efficient information retrieval, allowing each aspect to evolve according to specific requirements without compromising counterpart functionality [3].

Historical preservation extends the immutability concept to its logical conclusion by maintaining comprehensive chronological notification sequences as primary system records rather than merely current state representations. Within this pattern, any entity's current state derives from replaying all historical notifications affecting that entity since inception. This methodology yields substantial advantages: exhaustive audit capabilities, historical state reconstruction, and temporal query functionality. The unalterable notification log serves as the definitive record resistant to retrospective modification, providing robust guarantees regarding information integrity and auditability requirements. When applications require entity modifications, they append new notifications to the repository rather than altering existing records. This additive-only model naturally aligns with distributed system constraints while providing clear mechanisms for maintaining consistency across eventually-consistent boundaries. Historical preservation additionally enables powerful diagnostic capabilities, allowing developers to recreate precise notification sequences leading to specific system states, substantially simplifying the diagnosis of complex issues that might otherwise remain difficult to reproduce [4].

Message sequencing and distribution patterns constitute the communication infrastructure within notification-oriented systems. Message sequences establish point-to-point communication channels, ensuring reliable notification delivery between components, even when originators and recipients operate at differing rates or experience temporary unavailability. These sequences typically incorporate features including message persistence, delivery assurances, and specialized handling for unprocessable messages. Distribution patterns extend this model by enabling one-to-many communication, where notifications published to specific topics reach all interested subscribers. This pattern facilitates highly decoupled systems where additional recipients can be introduced without modifying originators. The distinction between domain notifications and integration notifications assumes particular importance within these messaging contexts—domain notifications represent meaningful state changes within contained operational boundaries, while integration notifications facilitate communication across contextual boundaries. Understanding these distinctions helps architects develop appropriate serialization, versioning, and routing strategies addressing the divergent requirements between internal and cross-boundary communication patterns [3].

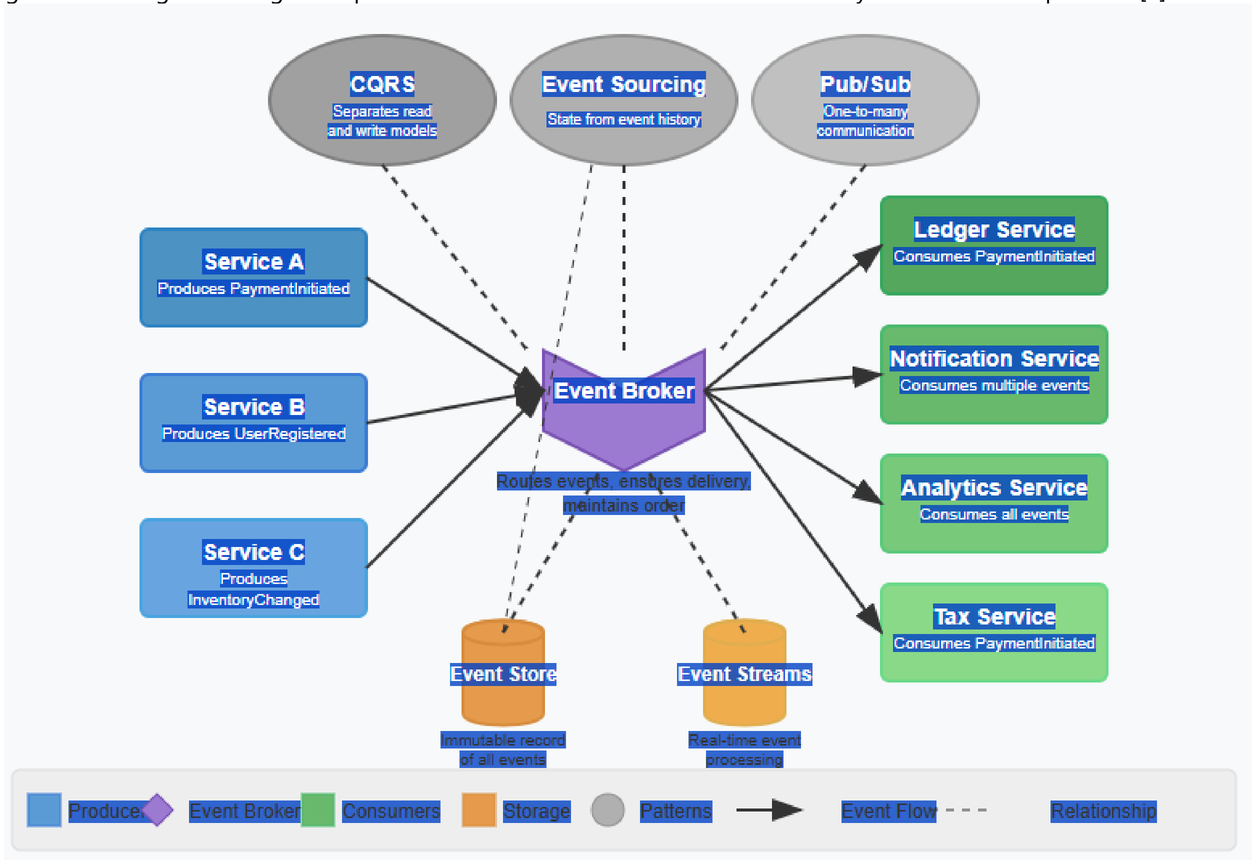

Fig. 1: Event-Driven Architecture: Components, Flows, and Patterns. [3, 4]

3. Implementation Considerations in Modern Backend Systems

The deliberate selection of technological infrastructure constitutes a pivotal determination when constructing notification-oriented system architectures. The diverse ecosystem of notification streaming and messaging frameworks presents varied alternatives with distinctive compromises regarding information throughput, response timing, coherence assurances, and administrative intricacy. Substantial-capacity, elevated-throughput contexts frequently gravitate towards decentralized recording platforms prioritizing lateral expandability and disruption resistance. Such frameworks characteristically deploy segmented notification sequences with adjustable preservation directives and recipient collective semantics, facilitating concurrent processing while preserving sequential guarantees within segments. Contemporary notification handling platforms seamlessly integrate with computational intelligence frameworks, enabling instantaneous deduction on streaming information without necessitating intermediate batch handling mechanisms. This convergence between streaming capabilities and computational intelligence establishes powerful functionalities for instantaneous analytics, irregularity identification, and anticipatory operations. For circumstances with modest throughput demands but sophisticated routing requirements, message coordination systems supporting intricate routing arrangements and message transformation capabilities may prove advantageous. Cloud-optimized notification routing services abstract substantial administrative complexity associated with coordinator infrastructure management while delivering effortless integration with complementary managed services. The technology determination process necessitates careful evaluation of considerations, including anticipated notification volume, response time sensitivity, persistence requirements, and organizational technical proficiency to identify the most appropriate foundation for notification-driven system architecture [5].

Notification structure design and progression introduce distinctive challenges within notification-driven systems, where notifications frequently outlast the services generating them. Effectively designed notification structures balance expressiveness with adaptability, capturing essential domain information while maintaining resilience against evolving business requirements. Structure registries have emerged as critical infrastructure components, providing centralized administration of notification formats and facilitating compatibility verification between generators and recipients. Compatible structure evolution typically adheres to established patterns: incorporating optional attributes, expanding categorizations, and introducing new notification types while preserving backward compatibility with existing recipients. More substantial changes may necessitate versioning approaches, such as maintaining multiple structure versions during transition periods or implementing notification transformations converting between versions. The distinction between notifications and directives assumes particular importance within structure design—notifications represent accomplished occurrences appropriately named using past tense expressions, while directives represent intentions suitably named using imperative expressions. Structure design should emphasize domain semantics rather than technical implementation particulars, ensuring notifications capture the business significance of state modifications rather than data storage mechanisms. This semantic methodology establishes more durable contracts between services and diminishes coupling with implementation-specific details potentially subject to modification. Notification structures should additionally consider diverse recipient requirements, potentially incorporating attributes irrelevant to certain recipients but enabling significant functionalities for specific downstream services [6].

Coherence challenges naturally manifest within distributed notification-driven systems, where conventional atomic transactions potentially span multiple independent services maintaining separate persistence mechanisms. These systems frequently adopt eventual coherence models, where temporary inconsistencies between components remain acceptable provided the system progressively converges toward a coherent state within reasonable timeframes. Various patterns have emerged addressing coherence concerns, including the externalization pattern, which atomically preserves domain modifications alongside outbound notifications within local transaction boundaries before asynchronously publishing notifications to external coordinators. Progress tracking patterns monitor notification processing advancement across distributed components, enabling consistent recovery following disruptions. Primary-secondary replication ensures consistent notification sequencing across distributed nodes while providing failover capabilities. Preliminary recording logs provide durability assurances by persistently documenting operations before application to primary data storage, establishing foundations for reliable historical preservation implementations. Multi-step orchestration patterns coordinate complex processes across service boundaries, utilizing compensatory transactions, maintaining business-level coherence despite technical transactions being unable to span multiple services. Notification-transported state transfer reduces inter-service dependencies by incorporating sufficient context within notifications, enabling processing without additional service interactions. These patterns acknowledge fundamental constraints of distributed systems while providing practical approaches to maintaining business coherence within environments where network partitions and partial failures represent expected rather than exceptional circumstances [7].

Malfunction handling and reattempt strategies assume heightened significance within notification-driven systems, where processing failures require management without distributed transaction safeguards. Unprocessable notification repositories capture notifications unsuccessfully processed after multiple reattempt efforts, preventing information loss while isolating problematic messages for subsequent analysis and remediation. Reattempt strategies typically implement progressive delay mechanisms with randomization, preventing concurrent recovery attempts during downstream service recovery. Processing idempotence ensures repeated processing of identical notifications produces consistent outcomes, representing a critical property when implementing reattempt logic or recovering from malfunctions. This idempotence may be achieved through various techniques, including natural idempotence (where operations inherently permit repetition), idempotence identifiers tracking previously processed notifications, or transactional filtering preventing duplicate processing. Circuit interruption mechanisms prevent cascading malfunctions by temporarily suspending notification processing when downstream systems demonstrate sustained failures, allowing recovery time before resuming normal operation. Problematic notification identification recognizes notifications consistently triggering processing failures, routing them toward specialized handling procedures rather than consuming system resources through ineffective reattempt efforts. These patterns require complementary operational practices, including comprehensive alerting regarding processing backlogs, automated recovery mechanisms, and utilities for replaying notifications from specific temporal points when necessary for recovering from failures or processing errors [5].

Supervision and perceptibility present distinctive challenges within notification-driven architectures, where request flows span multiple asynchronous processing stages without direct causal connections within traditional request documentation. Distributed pathway tracing emerges as an essential capability, propagating correlation identifiers across service boundaries, reconstructing complete notification paths through the system. Notification flow supervision tracks crucial metrics, including throughput, response timing, error frequencies, and processing backlogs throughout the entire notification pipeline, providing advanced warning regarding developing issues. Recipient delay metrics highlight processing postponements, potentially indicating performance problems or resource limitations. Unprocessable notification queue supervision identifies patterns within failed processing attempts, potentially signaling systemic issues requiring attention. Documentation correlation aggregates related log entries across multiple services, reconstructing operation sequences triggered by individual initiating notifications. Specialized visualization instruments for notification flows help administrators understand complex relationships between generators, coordinators, and recipients, providing insights regarding bottlenecks and processing anomalies. These perceptibility practices enable administrators to comprehend system behavior, diagnose issues, and validate notifications flowing through the system as expected, despite processing spanning multiple independent services with asynchronous communication patterns. The most sophisticated notification-driven systems implement comprehensive measurement systems enabling both technical and business-level insights, connecting technical metrics with supported business processes, providing context-aware operational information displays [6].

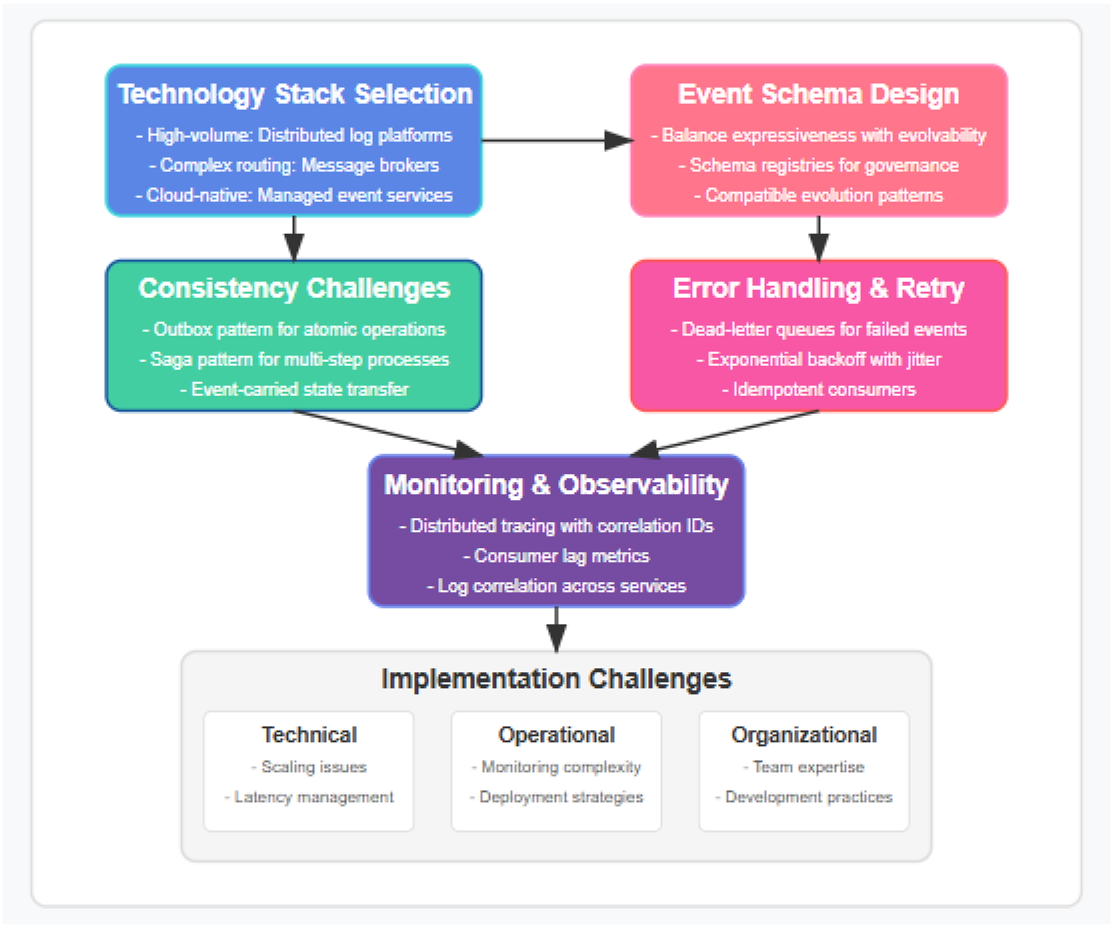
Technology Stack Selection
- High-volume: Distributed log platforms
- Complex routing: Message brokers
- Cloud-native: Managed event services
Event Schema Design
- Balance expressiveness with evolvability
- Schema registries for governance
- Compatible evolution patterns
Consistency Challenges
- Outbox pattern for atomic operations
- Saga pattern for multi-step processes
- Event-carried state transfer
Error Handling & Retry
- Dead-letter queues for failed events
- Exponential backoff with jitter
- Idempotent consumers
Monitoring & Observability
- Distributed tracing with correlation IDs
- Consumer lag metrics
- Log correlation across services
Implementation Challenges
Technical
- Scaling issues
- Latency management
Operational
- Monitoring complexity
- Deployment strategies
Organizational
- Team expertise
- Development practices

Fig. 2: Implementation Considerations in Modern Backend Systems. [6, 7]

4. Case Study: Real Wallet Platform

The commissioning structure within the Real Wallet ecosystem exemplifies advanced notification-centric architectural methodologies through its monetary compensation framework, orchestrating sophisticated financial procedures across various specialized operational components. The architectural foundation incorporates domain-focused construction principles, segmenting the payment handling domain into distinct contextual boundaries encompassing transaction acquisition, financial recording, taxation computation, regulatory adherence, and participant communication. Each operational boundary functions as an autonomous service with dedicated information persistence mechanisms, interacting with alternative boundaries predominantly through a unified notification infrastructure. This infrastructure incorporates segmented notification distribution technologies, ensuring sequential delivery within transactional constraints while accommodating concurrent processing across disparate user financial profiles. The framework leverages procedural coordination to administer extended operational sequences spanning multiple services, guaranteeing complex monetary transactions advance through necessary procedural stages despite potential service disruptions or processing interruptions. State alterations within these procedures generate notifications driving subsequent processing activities, establishing a composite architecture merging procedural coordination with notification-driven communication. This structural methodology provides essential disruption management capabilities for monetary transactions, incorporating automated reattempt mechanisms, temporal limitation handling, and compensatory transactions, preserving information consistency despite partial system disruptions. The platform's notification-driven foundation enables implementation of currency movement restrictions, regulatory verification, and fraudulent activity identification as independent considerations integrating through shared notification infrastructure rather than rigid service-to-service connections that would restrict independent evolution and capacity expansion [8].

The notification pathway for payment handling demonstrates how intricate organizational procedures can be divided into discrete operational stages coordinated through notifications. The sequence initiates when external mechanisms generate TransactionInitiated notifications containing commission payment specifications. The transaction acquisition service processes these notifications, performs validation, augmentation, and deceptive activity identification, subsequently publishing TransactionAuthorized notifications. These notifications trigger simultaneous processing across multiple downstream services: the financial recording service documents the monetary transaction, the taxation service calculates and reserves appropriate tax quantities, and the communication service prepares notifications for payment recipients. Each service publishes completion notifications—FinancialEntryDocumented, TaxReservationCalculated, and CommunicationQueued—consumed by a coordination service tracking overall payment progression. The notification-centric payment architecture fundamentally transforms traditional payment processing approaches, replacing conventional monolithic batch processing with responsive, instantaneous procedures that adapt to fluctuating transaction volumes. Cloud-optimized deployment frameworks support this transformation, with containerized services expanding elastically according to current processing requirements. The notification infrastructure implements precise processing semantics for financial transactions, ensuring commission payments avoid duplication or loss despite infrastructure disruptions or deployments. This architecture enables consolidated payment status visualization across multiple processing stages while permitting specialized service independent evolution, accelerating innovation while preserving stringent reliability requirements essential within financial ecosystems [9].

Integration mechanisms connecting financial recording, communication, and taxation components demonstrate how notification-centric architectures facilitate minimal coupling between specialized components. The financial recording component maintains authoritative financial records, documenting credits and debits across various account categories while ensuring balance accuracy. When processing commission payments, the financial system consumes TransactionAuthorized notifications, creates necessary accounting entries, and publishes FinancialEntryDocumented notifications referencing created financial records. The taxation component similarly consumes TransactionAuthorized notifications but concentrates on taxation implications, calculating withholding amounts according to jurisdiction-specific regulations and publishing TaxReservationCalculated notifications. The communication component operates distinctively, consuming notifications from both financial and taxation systems to generate comprehensive payment communications incorporating both net payment amounts and taxation withholding details. This notification-transported state transfer pattern reduces direct dependencies between services, as each service incorporates sufficient contextual information within published notifications, enabling subsequent processing without requiring additional service-to-service interface calls. This loosely coupled integration strategy establishes system-wide resilience, as temporary unavailability within individual subsystems doesn't prevent others from continuing operations. The notification streams connecting these subsystems implement persistent messaging with delivery assurances, ensuring processing ultimately completes despite services experiencing temporary unavailability or scaling operations. This reliability foundation enables the platform to satisfy strict regulatory requirements regarding financial recordkeeping while maintaining the adaptability necessary for accommodating evolving business requirements [10].

Performance characteristics from the Real Wallet platform demonstrate the scalability advantages of a notification-centric architecture within financial systems. By decomposing monolithic payment processes into independent, notification-driven services, the platform achieved substantial improvements regarding information throughput, response timing, and resource utilization. The asynchronous processing model enables each subsystem to expand independently according to specific resource requirements and processing characteristics. The platform implements segmentation strategies, distributing processing workloads across multiple service instances while preserving transactional boundaries, enabling horizontal expansion without compromising information consistency. Temporal separation between workflow stages prevents resource competition between different processing phases, enabling more efficient computing resource utilization compared with synchronous processing models. The notification-centric architecture additionally facilitates a gradual transition from established systems, with notification adapters bridging between conventional batch processes and contemporary real-time processing models. This incremental modernization approach enabled platform transition regarding critical financial functions without disrupting ongoing operations. Performance evaluation under various capacity conditions demonstrated the architecture's capability to handle unpredictable transaction volumes through elastic expansion, with automatic resource allocation during peak periods and contraction during reduced activity periods to optimize infrastructure expenses. The platform's cloud-optimized deployment model complements this elasticity, utilizing containerized services rapidly deployable across multiple availability regions, ensuring continued operation despite regional infrastructure disruptions [8].

Resilience methodologies implemented within the Real Wallet platform demonstrate how notification-centric architectures maintain system reliability despite individual component disruptions. The platform implements circuit protection mechanisms that monitor error frequencies within downstream service communications, temporarily suspending new requests when error thresholds exceed acceptable limits to prevent cascading disruptions. The architecture acknowledges that distributed system disruptions represent inevitable rather than exceptional circumstances, designing accordingly with patterns preserving information consistency and business continuity despite partial system unavailability. Each financial transaction implements persistent execution contexts surviving service restarts, ensuring in-progress transactions resume from recent consistent states rather than requiring manual intervention or creating duplicate processing. Historical preservation provides additional resilience capabilities, with notification logs serving as authoritative records reconstructing service states following disruptions. The platform implements comprehensive observability through distributed pathway tracing following transactions across service boundaries, custom measurements tracking, processing timing, and error frequencies, and structured documentation facilitating troubleshooting across distributed components. Regional duplication regarding both services and notification streams enables disaster recovery with minimal information loss, while active-active deployment models allow continued operation despite significant infrastructure disruptions. These resilience methodologies collectively establish payment systems maintaining financial integrity despite adverse conditions, progressively reducing non-essential functions during partial disruptions while preserving core payment processing capabilities [10].

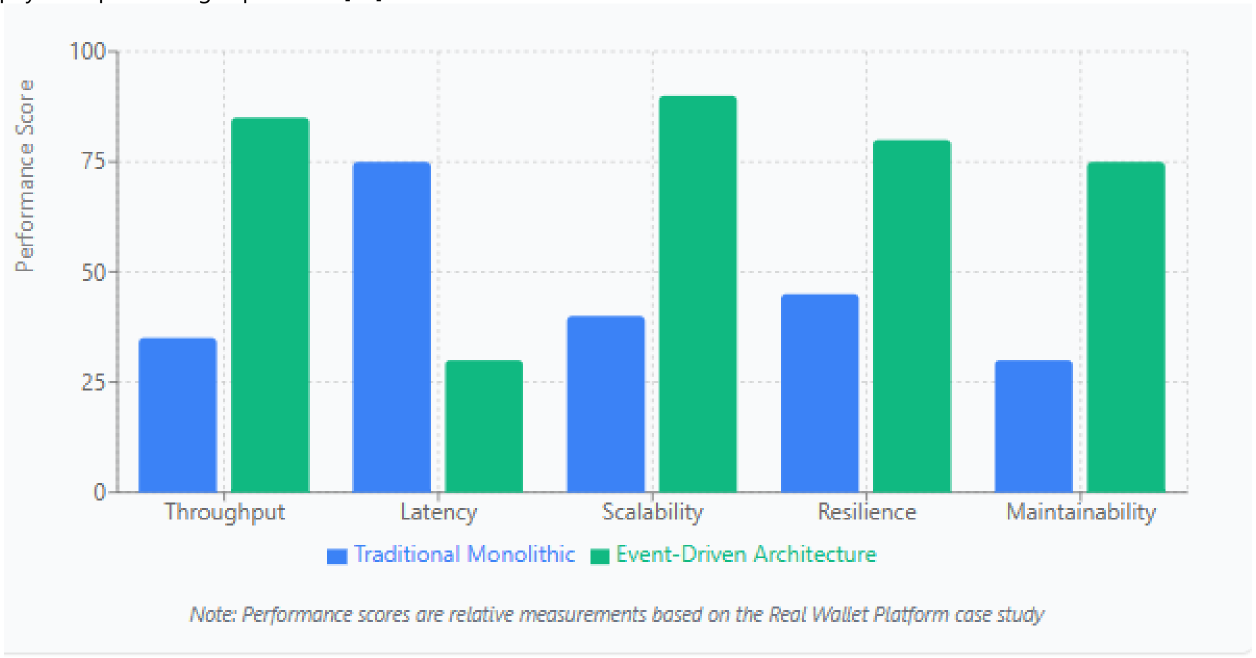

Fig. 3: Real Wallet Platform: Performance Benefits of Event-Driven Architecture. [9, 10]

5. Conclusion

Event-Driven Architecture offers transformative benefits for modern backend systems, fundamentally changing how components interact by replacing direct coupling with event-based communication. The decoupling of producers from consumers creates systems that can evolve independently, scale efficiently, and gracefully handle partial failures. As demonstrated through the Real Wallet platform case study, EDA enables complex domain processes to be decomposed into discrete, specialized services that communicate through well-defined events, creating systems that are both more flexible and more resilient than traditional architectures. Looking forward, the convergence of event streaming with technologies like machine learning, edge computing, and serverless platforms will further expand EDA's capabilities and application domains. Organizations adopting EDA should emphasize strong event design practices, invest in robust observability solutions, and implement appropriate consistency patterns that acknowledge the fundamental constraints of distributed systems. Future advancements in schema evolution, real-time analytics integration, and standardized patterns will continue to enhance the maturity and accessibility of event-driven methods, making them increasingly central to the architecture of responsive, scalable, and adaptable systems.

Funding: This research received no external funding.
Conflicts of Interest: The authors declare no conflict of interest.
Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers.

References

[1] Adam B, (2020) Building Event-Driven Microservices, O'Reilly Media, Inc., 2020. https://www.oreilly.com/library/view/building-event-driven-microservices/9781492057888/

[2] Ben S, (2018) Designing Event-Driven Systems, O'Reilly Media, 2018. https://www.oreilly.com/library/view/designing-event-driven-systems/9781492038252/

[3] Chris R, (2023) Book: Microservices patterns," Microservices.io. https://microservices.io/book

[4] Gregor H, Bobby W, (2003) Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions, Addison-Wesley Professional, 2003. https://www.oreilly.com/library/view/enterprise-integration-patterns/0321200683/

[5] Mark R, (2015) Software Architecture Patterns, O'Reilly Media, 2015. https://theswissbay.ch/pdf/Books/Computer%20science/O'Reilly/software-architecture-patterns.pdf

[6] Naresh K, (n.d) Harnessing cloud-based microservices for payments revolution, TCS BaNCS. https://www.tcs.com/what-we-do/products-platforms/tcs-bancs/articles/transforming-payments-with-cloud-microservices-architecture

[7] Rajesh K P, and Steef-Jan W, (2025) Designing Resilient Event-Driven Systems at Scale, InfoQ, 2025. https://www.infoq.com/articles/scalable-resilient-event-systems/

[8] TensorFlow, (2023) Robust machine learning on streaming data using Kafka and Tensorflow-IO, TensorFlow Documentation, 2023. https://www.tensorflow.org/io/tutorials/kafka

[9] Tim I, (2025) Building Resilient Event-Driven Architecture for Finance with Temporal, Temporal Technologies, 2025. https://temporal.io/blog/building-resilient-event-driven-architecture-for-finserv-with-temporal

[10] Unmesh J (2023) Catalog of Patterns of Distributed Systems, Martin Fowler's Website, 2023. https://martinfowler.com/articles/patterns-of-distributed-systems/