Mahmud FARZALI
Master's student at School of High Technologies and Innovative Engineering, Department of Information Technologies, Western Caspian University, Baku, Azerbaijan
E-mail: mahmudfrzli@gmail.com

DESIGN OF A FAULT-TOLERANT PAYMENT SYSTEM USING MICROSERVICE ARCHITECTURE AND EVENT-DRIVEN COMMUNICATION

Abstract

The purpose of this article is to propose a practical architecture for a fault-tolerant digital payment system built on microservices and event-driven communication. The study uses a design-science approach based on architectural modelling, analysis of microservice patterns and implementation-oriented decomposition of a payment flow into gateway, order, payment, processing, adapter, ledger and notification services. The proposed model shows that reliability in payment systems should not depend on synchronous calls alone. Idempotency keys, transactional outbox, saga-based state transitions, retry policies, dead-letter queues, circuit breakers and audit-oriented ledger entries together reduce data loss, duplicate charging and inconsistent transaction states.

Keywords: microservices, payment processing, event-driven architecture, transactional outbox, idempotency.

UOT: 004.75:004.414:336.71
JEL: L86, O33, G20, D85
DOI: https://doi.org/10.54414/IDOJ8538

Introduction

Digital payment platforms operate in an environment where technical failure directly affects financial trust. A delayed callback, duplicated request, unavailable bank adapter or inconsistent ledger record may create customer complaints, operational disputes and reputational risk. For this reason, the architecture of a payment platform must be designed not only for functional correctness but also for failure tolerance, traceability and controlled recovery. Traditional monolithic architectures may be easier to deploy at the beginning, but as transaction volume and integration complexity increase, tightly coupled modules become difficult to scale, test and recover independently.

Microservice architecture addresses part of this problem by decomposing a large system into independently deployable services with clear responsibilities [1]. In a payment domain, this decomposition usually separates authentication, order management, payment orchestration, external provider adapters, ledger accounting, notification and reporting. However, this separation introduces a new challenge: a business transaction is no longer executed inside one database transaction. Each service owns its own data and communicates with other services through APIs or messages. Therefore, distributed consistency must be achieved through architectural patterns rather than a single global transaction [2].

The research problem of this article is the following: how can a payment system be structured so that it continues processing safely when one of its components fails, without charging the customer twice or losing the financial audit trail? The objective is to design a microservice-based and event-driven payment architecture that combines asynchronous communication, local transactions and explicit status management. The article focuses on practical mechanisms such as the transactional outbox, idempotent consumers, saga-style transaction flow, double-entry ledger records and resilience patterns. These mechanisms are especially important in fintech systems where correctness, availability and auditability must be balanced.

Theoretical foundations of the proposed architecture. A microservice is not simply a small application; it is a service boundary around a business capability. In payment systems, service boundaries should be chosen according to domain responsibilities. For example, the payment service accepts and validates a payment request, the processing service coordinates transaction execution, the adapter service communicates with a bank or payment provider, and the ledger service records financial movements. This separation reduces coupling and allows each service to evolve independently, but it requires strong integration discipline [1, 2].

Event-driven architecture provides such integration discipline by allowing services to publish and consume events through a broker. Instead of forcing all services to wait for each other synchronously, the system records important business facts such as PaymentCreated, PaymentAuthorized, ProviderCallbackReceived or LedgerEntryPosted. Apache Kafka and similar log-based brokers are often used for this purpose because they provide durable ordered records, consumer groups and scalable event processing [7,8]. Nevertheless, brokers usually provide delivery guarantees that must be handled carefully by applications. A message may be retried, consumed more than once or processed after a temporary delay. Therefore, idempotency and state validation are mandatory in financial workflows.

Distributed transaction management is another theoretical foundation of the design. The classical saga pattern divides a large business transaction into a sequence of local transactions. If one step fails, compensating actions or explicit failure states are executed instead of rolling back a global transaction [5, 10]. Modern research also confirms that sagas are suitable for maintaining consistency in microservice systems, although isolation problems must be carefully managed [6]. In a payment platform, the saga is represented through controlled statuses such as NEW, PENDING, AUTHORIZED, CAPTURED, FAILED and REVERSED. Each status transition must be validated so that a late callback or retry cannot move the transaction into an invalid state.

The transactional outbox pattern is used to close the reliability gap between database updates and event publication. A service writes the business change and the outgoing event into the same local database transaction. A separate publisher later reads the outbox table and sends the event to the broker [9]. This means that if a payment row is committed, the corresponding event is also durably stored. The pattern does not remove the need for idempotent consumers, but it prevents the dangerous situation where the database is updated while the message is lost.

Research methodology and architectural design. The article follows a design-science methodology. First, the main architectural requirements of a payment platform are identified: high availability, consistency of transaction status, prevention of duplicate financial operations, auditability, provider integration flexibility, observability and security. Second, these requirements are mapped to microservice and event-driven patterns. Third, the proposed solution is expressed as a target architecture that can be implemented with Spring Boot services, PostgreSQL databases, Kafka topics and monitoring tools.

The proposed architecture contains eight main components. The API Gateway receives external requests, applies routing and central cross-cutting policies. The Auth Service issues and validates user or merchant credentials. The Order Service stores merchant order data. The Payment Service creates payment intents, validates idempotency keys and publishes payment events. The Processing Service orchestrates the payment state machine. The Adapter Service isolates provider-specific APIs. The Ledger Service records debit and credit entries. Finally, the Notification Service informs merchants and customers about payment results. Each service owns its database schema, and integration is performed through synchronous calls only when immediate response is necessary; otherwise, events are preferred. The composition and fault-tolerance role of these services are summarized in Table 1.

Table 1. Main components of the proposed architecture.

Component	Responsibility	Fault-tolerance role
API Gateway	Routes requests and applies common entry policies	Reduces direct exposure of internal services
Payment Service	Creates payment intent and checks idempotency	Prevents duplicate payment creation
Processing Service	Coordinates transaction state transitions	Controls saga flow and recovery states
Adapter Service	Integrates with external banks/providers	Isolates provider failures and timeouts
Ledger Service	Stores debit/credit records	Provides auditable financial history
Kafka and Outbox	Transfers events reliably	Prevents message loss and supports retries

The normal payment flow starts when a merchant creates an order and sends a payment request with a unique idempotency key. The Payment Service checks whether this key was already processed. If it is new, the service stores the payment intent and an outbox event in one database transaction. The Processing Service consumes the event and calls the appropriate Adapter Service. If the provider returns a synchronous response, the transaction status is updated immediately. If the provider uses delayed callbacks, the system keeps the transaction in an intermediate state and later reconciles it through callback events. This approach makes the payment lifecycle explicit and observable.

A key design decision is that external provider callbacks are not treated as simple HTTP notifications. They are converted into internal domain events. This prevents provider-specific logic from spreading across the platform and allows replay, reconciliation and dead-letter analysis. For example, if a bank confirms that money was debited but the merchant callback fails, the platform can keep the payment in a technically successful but merchant-notification-pending state. Support teams can then diagnose the exact stage of failure rather than guessing whether the customer was charged.

Implementation mechanisms for fault tolerance. The first implementation mechanism is idempotency. In payment systems, clients and network intermediaries may retry the same request after a timeout. Without idempotency, a retry may create a second transaction or trigger a second provider authorization. The proposed design stores an idempotency key together with the request hash, merchant identifier and final response. If the same key is received again, the system returns the stored result instead of executing the financial operation again. Idempotency must also be applied to event consumers by storing processed event identifiers or by validating state transitions before update.

The second mechanism is the transactional outbox and inbox combination. The outbox guarantees that local database changes and outgoing events are stored atomically [9]. The inbox or processed-message table protects consumers against duplicate delivery. In practice, the publisher marks outbox records as sent only after successful broker acknowledgment, while consumers process an event only if its unique identifier has not already been recorded. This model is compatible with at-least-once delivery and avoids relying on unrealistic assumptions about perfect networks.

The third mechanism is controlled retries with dead-letter queues. Temporary errors such as provider timeout, network interruption or database deadlock should be retried with backoff. Permanent errors such as invalid card data, blocked merchant or malformed callback should not be retried indefinitely. A dead-letter queue stores failed events after the retry limit is reached, allowing operational review and manual or automated recovery. This separation reduces noisy failures and prevents one bad message from blocking the processing of valid payments.

The fourth mechanism is the circuit breaker. When an external provider becomes unstable, continuous calls from the platform can increase latency and exhaust connection pools. A circuit breaker tracks recent failures and temporarily stops calls to the failing provider, returning a controlled response or

switching to a fallback path [12]. In payment processing, circuit breakers must be combined with clear business states. A provider timeout does not automatically mean payment failure; it may mean that reconciliation is required. Therefore, the architecture distinguishes technical failure from financial failure.

The fifth mechanism is ledger-based auditability. A payment platform should not only store the current payment status; it must also preserve a financial trail. The ledger service records debit and credit style entries for each financial movement. Double-entry logic improves internal consistency because every debit must have a corresponding credit. It also helps in reconciliation, reporting and dispute investigation. Even when a transaction is reversed, the system should create compensating ledger entries instead of deleting previous records. This preserves historical truth and supports audit requirements.

Practical recommendations for deployment and testing. For implementation in a real fintech environment, the proposed architecture should be introduced incrementally. The first phase should separate the payment lifecycle from provider-specific code by creating an adapter layer. This reduces the risk of changing core payment logic whenever a new bank or payment provider is integrated. The second phase should introduce idempotency and the outbox table in the Payment Service because these two mechanisms directly protect against duplicate requests and lost events. The third phase should move non-critical steps such as notification, reporting and reconciliation into asynchronous consumers. This staged approach allows the organization to gain reliability benefits without rewriting the whole platform at once.

Testing should also reflect the distributed nature of the architecture. Unit tests are not sufficient for payment workflows because many failures appear only during integration. Contract tests should verify that adapters send and receive provider messages in the expected format. Integration tests should check that an outbox record is published after the database transaction commits. Failure-injection tests should simulate provider timeout, Kafka unavailability, duplicated callbacks, database deadlocks and delayed merchant notification. The expected result of each test should be a valid business state, not only a successful HTTP status.

Deployment must be supported by operational safeguards. Each service should define readiness and liveness probes, database connection limits, timeout values and rollback procedures. Kafka consumers should expose consumer lag metrics; outbox publishers should expose backlog metrics; and provider adapters should expose latency and error-rate metrics. Alerting rules should be connected to business risk, such as an increasing number of provider timeouts or ledger posting failures [4, 8, 12].

Discussion

The proposed design improves reliability but does not remove all complexity. Event-driven systems introduce eventual consistency, which means that different services may temporarily observe different states. This is acceptable only when the business process is designed for it. For example, a merchant screen may show a payment as processing while the platform waits for a provider callback. The benefit is that the system remains available and recoverable instead of blocking all participants on a distributed transaction. This trade-off is consistent with the idea that scalable distributed systems should prefer local transactions and explicit coordination over global locks [3].

Another important issue is observability. A microservice payment platform must include correlation identifiers, structured logs, metrics and dashboards. Each request should carry a request identifier across gateway, payment, processing, adapter and ledger services. This makes it possible to find the exact point where a transaction stopped. Metrics such as payment success rate, provider latency, Kafka consumer lag, outbox backlog, dead-letter count and database connection pool usage should be monitored continuously. Without observability, event-driven architecture may become difficult to operate despite being theoretically resilient.

Security is also an architectural requirement, not a separate module. Authentication and authorization must protect merchant APIs; sensitive data must be encrypted or tokenized; and operational access must be controlled. Digital identity guidance emphasizes the need for appropriate authentication assurance and secure lifecycle management [11]. In a payment system, this principle applies both to users and to service-to-service communication. Therefore, API keys, OAuth2 tokens, mutual TLS or signed callback validation should be considered depending on the integration risk level.

Compared with a simple synchronous architecture, the proposed model requires more infrastructure and stronger engineering discipline. Developers must understand message ordering, duplicate delivery, transaction boundaries and operational recovery. However, for fintech systems the additional complexity is justified because it reduces the probability of data loss, duplicate charging and untraceable failures. The main scientific and practical contribution of this article is the combination of known distributed-system patterns into one payment-oriented architecture that can be implemented incrementally.

Conclusion

This article presented a fault-tolerant payment system architecture based on microservices and event-driven communication. The proposed design decomposes the platform into gateway, authentication, order, payment, processing, adapter, ledger and notification services. It uses Kafka-style event communication, local databases, saga-based status transitions, transactional outbox, idempotent request handling, retry policies, dead-letter queues, circuit breakers and double-entry ledger records.

The analysis shows that payment reliability cannot be achieved by one pattern alone. Idempotency protects against duplicate requests, the outbox protects against lost events, sagas control distributed state, circuit breakers protect the platform from cascading external failures, and ledger entries preserve auditability. The proposed architecture differs from earlier generic microservice descriptions by focusing specifically on payment lifecycle risks: duplicate charging, delayed provider callback, merchant notification failure, reconciliation and financial traceability.

The practical value of the work is that it can be used as a reference model for fintech platforms that need to modernize monolithic payment processing or design a new payment gateway. Future research may extend the model with multi-region deployment, formal verification of payment state transitions, automated reconciliation algorithms and performance testing under high transaction volume.

REFERENCES

1. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media; 2021.
2. Richardson C. Microservices patterns: with examples in Java. Shelter Island: Manning Publications; 2018.
3. Kleppmann M. Designing data-intensive applications. Sebastopol: O'Reilly Media; 2017.
4. Nygard M.T. Release It!: Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf; 2018.
5. Garcia-Molina H., Salem K. Sagas. ACM SIGMOD Record. 1987;16(3):249-259. Doi:10.1145/38713.38742
6. Daraghmi E.Y., Daraghmi Y.A., Yuan S.M. Enhancing saga pattern for distributed transactions within a microservices architecture. Applied Sciences. 2022;12(12):6242. Doi:10.3390/app12126242
7. Kreps J., Narkhede N., Rao J. Kafka: a Distributed Messaging System for Log Processing. Proceedings of the NetDB Workshop; 2011; Athens, Greece.
8. Apache Software Foundation. Apache Kafka Documentation. Available at: https://kafka.apache.org/documentation/ Accessed May 14, 2026.
9. Richardson C. Pattern: Transactional Outbox. Available at: https://microservices.io/patterns/data/tra

nsactional-outbox.html Accessed May 14, 2026.
10. Richardson C. Pattern: Saga. Available at: https://microservices.io/patterns/data/saga.html . Accessed May 14, 2026.
11. Grassi P.A., Fenton J.L., Newton E.M., Perlner R.A., Regenscheid A.R., Burr W.E., et al. Digital Identity Guidelines: Authentication and Lifecycle Management. NIST Special Publication 800-63B. Gaithersburg: National Institute of Standards and Technology; 2020. Doi:10.6028/NIST.SP.800-63b
12. Resilience Circuit Breaker Documentation. Available at: https://resilience4j.readme.io/docs/circuitbreaker . Accessed May 14, 2026.

Mahmud FƏRZƏLİ
Qərbi Kaspi Universitetinin magistrantı, Yüksək Texnologiyalar və İnnovativ Mühəndislik Məktəbi, İnformasiya Texnologiyaları kafedrası, Bakı, Azərbaycan

MİKROSERVİS ARXİTEKTURASI VƏ HADİSƏ-YÖNÜMLÜ KOMMUNİKASİYA ƏSASINDA DAYANIQLI ÖDƏNİŞ SİSTEMİNİN LAYİHƏLƏNDİRİLMƏSİ

Xülasə

Məqalənin məqsədi mikroservis və hadisə-yönümlü yanaşma əsasında dayanıqlı rəqəmsal ödəniş sistemi üçün praktik arxitektura modelinin təklif edilməsidir. Tədqiqatda arxitektur modelləşdirmə, dizayn-elm yanaşması və ödəniş axınının funksional servis sərhədlərinə bölünməsi metodlarından istifadə edilmişdir. Təklif edilən model göstərir ki, idempotentlik, transactional outbox, saga statusları, retry mexanizmləri, dead-letter queue, circuit breaker və ledger qeydləri birlikdə sistemin etibarlılığını və audit imkanlarını artırır.

Açar sözlər: mikroservis, ödəniş sistemi, hadisə-yönümlü arxitektura, transactional outbox, idempotentlik.

Махмуд ФАРЗАЛИ
Магистрант Западно-Каспийского университета, Школа высоких технологий и инновационной инженерии, кафедра информационных технологий, Баку, Азербайджан

ПРОЕКТИРОВАНИЕ ОТКАЗОУСТОЙЧИВОЙ ПЛАТЕЖНОЙ СИСТЕМЫ НА ОСНОВЕ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ И СОБЫТИЙНОЙ КОММУНИКАЦИИ

Резюме

Целью статьи является разработка практической архитектурной модели отказоустойчивой цифровой платежной системы на основе микросервисов и событийного взаимодействия. Использованы архитектурное моделирование, дизайн-научный подход и декомпозиция платежного процесса на функциональные сервисы. Модель показывает, что идемпотентность, transactional outbox, saga-состояния, повторные попытки, dead-letter queue, circuit breaker и бухгалтерские ledger-записи совместно повышают надежность, восстановимость и аудируемость системы.

Ключевые слова: микросервисы, платежная система, событийная архитектура, transactional outbox, идемпотентность.

Daxil olub: 14.05.2026