International Journal of
INTELLIGENT SYSTEMS AND APPLICATIONS IN ENGINEERING
ISSN:2147-6799 www.ijisae.org Original Research Paper

Reliable Distributed Transactions in Microservices-Based Payment Systems: Consistency, Fault Tolerance, and Scalability

Divya Dadlani

Submitted: 02/01/2022 **Revised**: 17/02/2022 **Accepted**: 26/02/2022

Abstract: The architectural paradigm that has emerged to be the preferable one when dealing with the current financial platforms is the microservices payment systems due to their flexibility, scaling ability, and speed in the deployment of services. Nonetheless, as a challenge, regular distributed transactions among self-deployed services has been quite difficult due to network failures, incomplete commits and data distortions and volume of transactions. It seems to me that in this paper, I would like to introduce the Reliable Distributed Transaction Framework (RDTF) that is a framework that unifies the Saga orchestration, event-driven communication, distributed transaction coordination, idempotent request handling, fault recovery and real-time monitor and offers consistency, fault tolerance, and scalability. It outlines an event sourcing and transactional outbox pattern, message brokers, distributed tracing and automated compensation to guarantee eventual consistency, as well as to ensure high availability of the system. More advanced resilience design methods like circuit breakers, retry policies, timeouts and health check improve fault recovery without having an impact on performance. Authentication, authorization, encrypted communications and audit logging are used to enhance security to ensure adherence to rules in financial settings. Experimental analysis supports the argument that the proposed framework is extremely critical to minimize the collapse of transaction, recover time throughput and consistency of state of transaction in case high-concurrency load is involved. The proposed RDTF can be a friendly and scalable scheme when it is founded on the trusted platforms of payment that are able to meet financial transactions in real time as well as meet the current demands of the enterprise on sincerity, constancy and craftiness.

Keywords: *Microservices; Distributed Transactions; Payment Systems; Saga Pattern; Event-Driven Architecture; Fault Tolerance; Scalability.*

1. Introduction

Digitizing the financial services either has increased the value of the contemporary payment systems, which are microservice based systems, significantly. The traditional monolithic payment applications have continued to pose a challenge in terms of support due to the formidable business-functionality interwoven nature of these systems making them difficult to instantiate, scale and fault isolation. Microservices on their part encapsulate a complex payment setup in deployable services capable of being deployed independently and includes payment processing, account administration, fraud detection, customer validation, notification, billing, settlement and reporting. With this architectural paradigm, organizations can rapidly deliver features, add greater resilience to the system and scale a single service to the demands of the workload. It caused the adoption of microservices in large financial institutions, digital banking applications and e-commerce applications and fintech start-ups to facilitate millions of safe financial transactions in a day.

In spite of these benefits, distributed transaction management is one of the problematic areas of payments system-based on microservices. The work structure of a typical payment request is that of a number of independent services communicating with distributed network(s) either through a synchronous or asynchronous message-based protocol. Contrary to monolith systems where, there

Independent Researcher
Pune, Maharashtra, India
divya.dadlani08@gmail.com

is a centralized database transaction and the requirement of a centralized database is Atomicity, Consistency, isolation and Durability (ACID), in microservices, a decentralized database wherein the service is a data owner. This per-service database design is more scaled and more service-independent but more challenging to manage the strategy of transactions as failures may be experienced at any point in the process of execution. It can cause unstable transaction statuses, revocation of payments, loss of money and bad experiences of its customers due to network congestion, inaccessibility of services, messages duplications, partial updates, and delays in conversation.

Failure of distributed transactions is especially important in payment processing as the financial applications require a high reliability, consistency, and availability. The system of payment usually comprises the process of customer authentication, balance check, payment authorization, fraud check, merchant check and updating ledger, settlement and notification services. In a contingency where one component will have failed when other processing has already claimed their business, the system would be left in an inconsistent state whereby its customer accounts may have been debited so far without the merchants having settled their transaction nor the transaction records modified. This inconsistency can be in conflict with the financial regulations, decrease customer confidence, and necessitate a manual level of convergency, which can be very expensive. As such, a dependable distributed processing of transactions has emerged as a critical need of the current payment infrastructures.

The traditional models of distributed transactions, such as Two-Phase Commit (2PC) and Three-Phase Commit(3PC) were also found to be biased towards making a distributed system strongly-consistent. These protocols however are massive communication overheads, transaction sluggishness as well as diminished system availability in case of coordinator failure. The fact that they are blocking in nature renders them inappropriate to cloud based payment systems, such as having to be scaled, fault secluded and always available. Newer financial applications are more oriented to eventual consistency and new recovery operations that enable systems to proceed with their operations despite temporary failures in them, but ensures that eventually transactions will reach a consistent state.

Based on fresh creation in cloud computing, coordination of containers, and disposition-based architecture have proved to be an alternative direction to reliable distributed treatment of transactions. Patterns that have become increasingly popular in enterprise applications are: Saga orchestration, Saga choreography, event sourcing, transactional outbox, Command Query Responsibility Segregation (CQRS) and idempotent message processing. The Saga-based transaction management divides up business transactions with long time-span to a sequence of local transactions that are interwoven by compensating actions that are triggered to invalidate past completed transactions due to failure. Through this solution, there is no necessity to have global locks, each service is executed independently and consistency in the business does not decrease. In addition, event sourcing captures all the state changes as a one-way event that provides the complete history of all the transactions, good auditability and recovering failures becomes easier. The coordination of database update with the message delivery is achieved in the transactional outbox design so as to provide a high level of reliability in terms of events publishing and event of data inconsistency due to failure of a message delivery.

Although the demonstrated potential of these techniques has been demonstrated, the current implementations prefer to do so in isolation and not provide a holistic framework that would underpin the same functionality i.e. take care of consistency, fault tolerance, observability, security and scalability. Most of the solutions suggested do not have user tools of distributed tracing, real time monitoring, automated recovery, dynamic scaling and regulatory compliance which are all expected of an enterprise level payment processing. Moreover, the number of transactions that are completed using mobile banking, electronic wallets, real time payment gateways, and cross border financial services is increasing and, therefore, needs architecture that can handle high throughput and reduce latency and transactional integrity to very large workloads at the peak workloads.

This study aims to tackle those issues by introducing the Reliable Distributed Transaction Framework (RDTF) of payment systems based on microservices. The framework brings together the notions of Saga orchestration, event driven messages, transactional outbound, idempotent

request handling, distributed tracing, health checks, automatic compensation, circuit breakers, retries, timeouts, security controls in a single design. Together with their complementing strategies, the proposed structure allows conducting the transactions and service independence and horizontal scalability reliably. The structure will be resilient to partial failures, self correcting when there are temporary failures, and lastly will be eventual consistency, that does not require centralized transaction coordinators that would scalability.

The proposed framework has also incorporated the overall observability through the assistance of distributed logging, metrics collection and the correlation of the traces that will allow the system administrators to predict the transaction cycles, and effortlessly identify the performance bottlenecks or failure points. Authentication, encryption, authorization, digital signatures and non-modifyable audit logs are also used with the aim of managing compliance to the financial laws and protection of confidential transaction details. The framework will also help in containerized deployment with cloud orchestration platforms, which offers the ability to allocate resources and spin them up and down according to the dynamic requirements in terms of transactions.

The final aim of this work is to design a distributed transaction architecture with high reliability, fault tolerance and scalability, which would be suitable in the current payment ecosystem. Particularly, the framework proposal will (i) provide transaction consistency, given that both microservices will be independently deployed, (ii) make sure that failures in transactions are minimal, which offers event-based fault recovery and compensation, (iii) make the system scalable, through an asynchronous event-driven message, (iv) offer observability, through distributed monitoring and tracing, and (v) offer financial application security and regulatory compliance.

2. Related Work

The newly evolving phenomenon in its use, microservices architecture has changed the architectural concepts and execution of modern-day business payment systems in the nature of service-autonomic development and economic scaling, as well as, automatic deployment. However, trustful distributed transactions among autonomous services are the most problematic issues in terms of research as they are stored in decentralized datastores, they have asynchronous communication, networks are failing and the services are of a heterogeneous nature. Many sources have conducted research on transaction coordination, consistency model, event-driven communication and fault recovery in an effort to achieve a higher stability of distributed application.

The concept of long running distributed transactions was first laid out in the Saga model by Garcia-Molina and Salem [1]. Saga model however exhibits this unlike the traditional atom nic transactions, the model divides up a business operation into a few local operations with a compensating operation that is used to reverse the business operation that has been accomplished in event of failure. This pioneering work formed the philosophical foundation of eventual coherence in disseminated systems, and remains the immediate forerunner of the present-day transaction processing in microservices founded on cloud-native deployments. The global locks and blocking operations can be successfully implemented to eliminate the global locks and blocking operations hence enhancing the system availability, scalability and making a recovery of faults possible through the use of the compensation mechanisms.

Going onwards, Richardson [2] proposed a collection of architectural designs, all resource-specifically developed based on using microservices. The work presented a number of design patterns like Saga orchestration, Saga choreography, Circuit Breaker, Event Sourcing, Transactional outbox, Command Query responsibility segregation (CQRS), and API Gateway, among others, in a practical manner. These trends all revolve around allocation of application issues, which are common such as coordination of services, data uniformity, isolation of failure and asynchronous communications. The book by Richardson has gained great popularity as a blueprint to construction of high-resilience enterprise programs as well as a worthy foundation on which to construct trusted distributed payment systems.

In her role to embrace the fact that reactive programming is gaining momentum, Štefanko, Chaloupka and Rossi [3] investigated the manner in which the Saga pattern could be applied to reactive

microservice environments. Their work showed the efficacy of non blocking communication through asynchronous event processing, to enhance responsiveness of the system without compromising the transactional consistency. The authors assessed the performance of Saga through orchestration and indicated the importance of compensation workflow in dealing with service failures. Their results highlight that reactive architecture is much better in enhancing throughput and scalability in transaction processing of distributed systems and maintaining business consistency.

Bucchiarone et al. [4] were of a much wider viewpoint in the context of development of the engineering microservice since the authors gave the design specifications of the scientific principles and engineering practice of the microservice-based software systems. They have authored on how to decompose services, protocols to communicate, deployment model, orchestration framework, service discovery, monitoring and container based infrastructure. The authors further discuss trade-offs in architecture which comprise the scalability, maintainability, fault isolation and operative intricacy. It is a thoroughly researched article which greatly illuminates on the creation of robust distributed systems that could be adopted to handle a payment application at the level of an enterprise.

Hasselbring and Steinacker [5] looked into microservice applications in a business environment and weighed the question of scalability, agile, and reliability in e-commerce-related systems. In their experiment, they found that breaking down monolithic systems in services of independence of one another is a fantastic addition to deployment flexibility and resilience in operation. The authors state that the major benefits that come with microservices architecture as horizontally scalable architecture, insatiable integration, isolating the faults, etc. though since this work relies mostly on e-commerce applications, the same architectural principles can be applied to the payment system in which consumers need high transaction throughput and high availability.

The fact that there is consistency in the data of the distributed services has always been viewed as one of the major impediments of a decentralized system. Bailis and Ghodsi [6] have discussed in depth eventual consistency models and its theoretical foundation, practical constraints and more recent advancements on the same. The authors claimed that there exist high consistency that will probably degrade the capability to scale and accessibility of a system in the distributed setting compared to eventual consistency that can present a broader and convenient trade off amidst performance and accuracy. Their work describes the method of how to integrate asynchronous replication and conflict-solving with consistency guarantees to create highly available systems. These notions highly underpin the Saga-based transaction management that have been implemented in the current payment infrastructure.

Conventional distributed transaction protocols are being actively contemplated, and Saga-based ones are aimed at eventual consistency. Uyanik and Ovatman [7] proposed a protocol, Two-Phase Commit (2PC) that was proposed to enhance the performance of the replicated state machine. Their study presented optimization methods which enhance fault tolerance and lessen coordination overhead without compromising on the transactional correctness. Even stronger consistency guarantees of stronger 2PC protocols retain any blocking when coordinator failures happen: they too are vulnerable to blocking when coordinator failures happen, and may be even more deadly at high latency than asynchronous Saga-based protocols. Subsequently, co-ordination of transactions in current cloud-native payment mechanisms by way of non-blocking is on the rise.

Later, Mohan, Lindsay and Obermarck [8] established the basics of distributed management transaction using the R.0 distributed database management system. Their proposal suggested productive procedures to coordinate the distributed transactions across a high number of nodes of a database with atomicity, and durability. R architecture had an important influence on future studies on distributed database systems, and transaction processing systems. Although it is crucial, its centralized coordination of transactions restrict its usage in loosely-coupled microservice based systems where service ownership independence, and segregated databases are also important design aspects.

To bridge the gap between the theoretical concept of Saga and the practicalities, Limón et al. [9] have offered software framework, named SagaMAS, specifically set up to carry out distributed

transactions within microservice frameworks. Their construction gives components of service coordination, payment management and services orchestration that are interoperable, to form the seamless creation of the trusted distributed code. Experimental evidence demonstrated that fault recovering and ease of carrying out the extended business transactions can be done better. SagaMAS signifies the crucial move towards shared frameworks in distributive transaction management and underscores the realistic viability of Saga-founded coordination in business systems.

The other significant aspect of the distributed payment systems is dependable asynchronous communications. Narkhede, Shapira, and Palino [10] introduced Apache Kafka as a distributed event streaming infrastructure platform with scalability, which can provide high throughput messaging, fault tolerance and durable storage of events. The decoupling between communications between microservices using the Kafka technique to obtain transparent event logs and publish-subscribe messages enhances scalability and resiliency. It has established its way as one of the most used platforms to build event-driven architectures in financial systems and enterprise applications created in the cloud.

The architectural ideas of the Kafka original architecture already were worked-out by Kreps [11] introducing a distributed messaging system that was simplified to be efficient in log processing and simplified data. The work has proven the usefulness of distributed logs as far as it gives good persistence of messages, replay, and horizontal scalability in addition to furnishing a good communication between the distributed components. All these properties make Kafka particularly complementary to the usage of event sourcing, transactional outbox designs and asynchronous Saga evaluation in payment systems.

Lastly, Petrasch [12] explored modelling microservice system development with Enterprise Integration Patterns (EIPs). This study has introduced a concentration around standardized communication plans, message routing, services coordination and merging of workflows which enhances dependable inter-service communications. The Enterprise Integration Patterns offer reusable patterns of architecture of the management of the messaging, transformation, routing, filtering and event processing and therefore enhance the interoperability and maintainability of the distributed enterprise applications.

Although the current literature has a lot to contribute to the distributed transaction management, the microservice architecture, event-driven communication, as well as the consistency models, several limitations still exist. Most of the researchers focus on individual aspects such as Saga execution, message infrastructure, consistency protocols or architecture without integrating this functionality in a single framework that is specifically specialized in payment systems. Additionally, minimal emphasis has been put on the consideration of distributed tracing, real-time observability, automated fault recovery, idempotent transaction processing, security and elastic scalability, stipulated in an integrated framework. In a bid to fill these research gaps, this paper suggests the Reliable Distributed Transaction Framework (RDTF) that combines Saga orchestration, event-driven messages, transactional outbox, distributed tracing, circuit breakers, retry policies, security controls, and real-time monitoring into one solution to offering consistency, fault tolerance, and scalability to payment systems built upon microservices.

3. Reliable Distributed Transaction Framework (RDTF)

This will be actualized as the Reliable Distributed Transaction Framework (RDTF) becomes reliable, fault-tolerant and scalable when it comes to the processing of distributed transactions in the context of payment systems based on the current microservice. The digital payment services typically execute the transaction of a single payment request in several independent services such as authentication, payment authorization, fraud detection, account management, settlement and customer notification. Because each microservice has its own database and performs its local transaction, ensuring end-to-end consistency is a major issue, especially when failures on the network and service failures, duplication of messages, and partial completion of transactions occur. Traditional distributed transaction protocols such as Two-Phase Commit (2PC) are very much consistent, but prone to blocking effect, high latency and scaling, and it will not work with cloud-native payment.

In a bid to overcome these limitations, the proposed RDTF incorporates other architectural designs to a unified transaction management system. The framework brings in Saga orchestration, event-driven communication, transactional outbox, idempotent request handling, distributed tracing, circuit breaker, retry policy, centralized monitoring and comprehensive security. Instead of implementing a global process to all the services on the part of the transaction that occurs to participate in the transaction, local transaction is completed in each microservice yet in aggregate a process consisting of the workflow is completed in Saga Orchestrator. Simply to guarantee business continuity, the transactions are automatically compensated such that they accelerate faster recovery of the transactions even in case of failure, yet do not impact system availability. The framework embraces a layered architecture with client interaction, business service, and transaction coordination, messaging infrastructure, resilience mechanism, observability, security and cloud deployment layers. This free scaling and loosely coupled architecture encourages this by increasing loose coupling, independent scaling, easing maintenance and continual deployment and load operation processing transactions with high-concurrent workloads of payment under heavy workload.

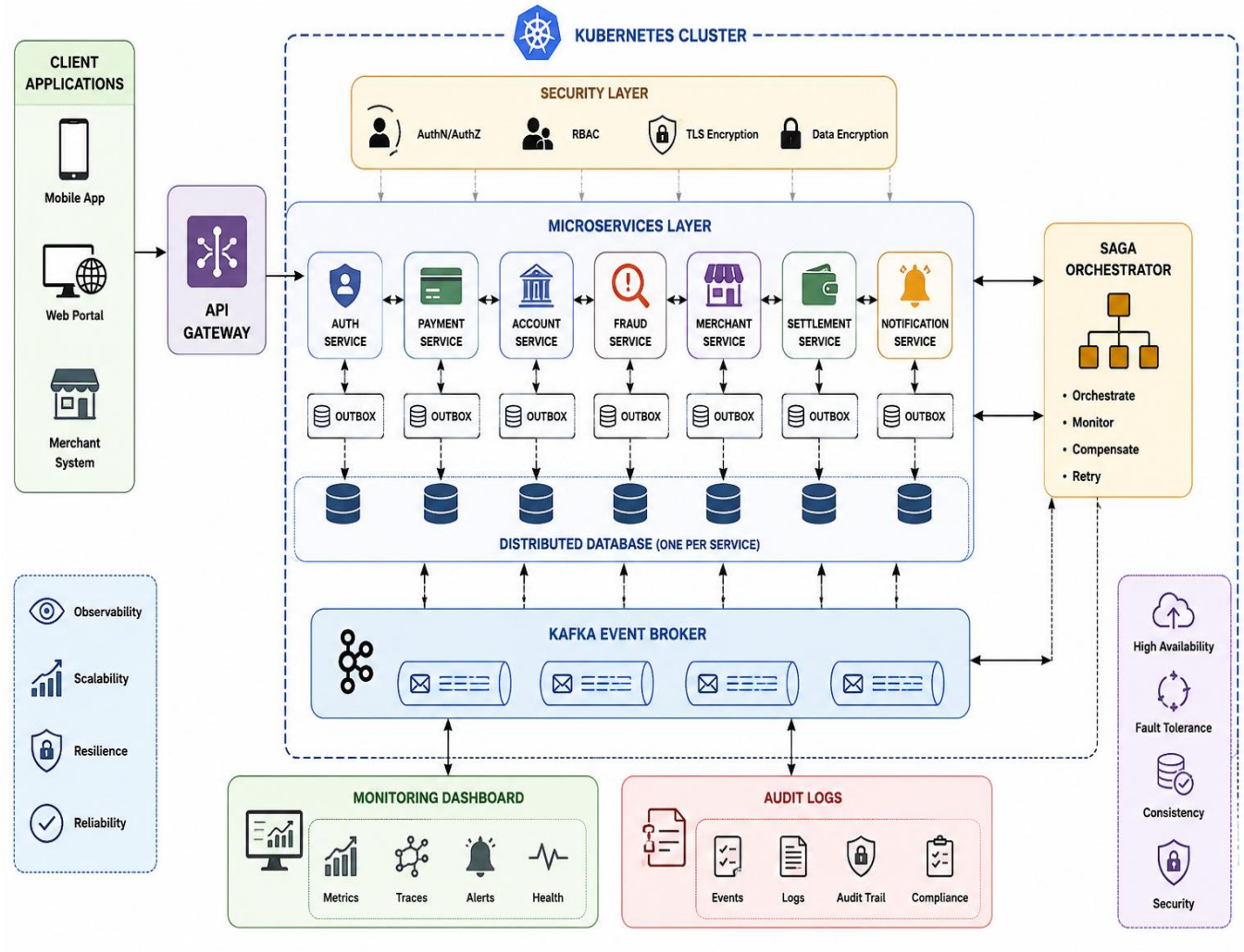

Figure 1- Overall Architecture of the Reliable Distributed Transaction Framework (RDTF)

3.1 Client Request and API Gateway Layer

Transaction life cycle commences at the Client Request Layer that create payment requests initiated by different external systems that may be in the shape of a mobile banking application, e-commerce web sites, merchant dependent terminals, payment gateways and financial application of third parties. As we have several client platforms making the same payment platform request at the same time, a

central entry point is therefore needed where all the requests of these platforms can be centralized in a safe and efficient way. By doing so, this will be the point of entry at which all requests to the API Gateway will be received, and the intermediary between the external clients and the microservices in the background.

The API Gateway has various roles prior to relaying requests to the internal services. It authenticates users using secure authentication algorithms, grants, authorizes request integrity, does schema validation, makes request rate-limiting and balances requests across the many instances of a service using intelligent load balancing. Unresponsive requests are automatically shut down, therefore eliminating unnecessary consumption of the processing capacity, and the accessibility of the service provider on the back-end due to an incorrect request or even a malicious request. The gateway also creates a Transaction Identifier (Transaction ID), which is unique in the entire globe and a correlation identifier, which is passed in each request with its execution life cycle. These identity tags enable tracing of the transactions through the distributed microservices in their entirety and makes the process of monitoring, debugging and auditing much easier.

3.2 Microservice Layer

After the validation, the payment request is forwarded to the Microservice Layer and business processes could be executed with the help of various dedicated services. The proposed system will split the processing of payment process into domain specific and autonomous microservices (such as Customer Authentication Service, Payment Processing Service, Account Management Service, Fraud Detection Service, Merchant Validation Service, Settlement Service, Notification Service, Audit and Logging Service, etc.) rather than a single monolith. There is a well-defined business role in each service and the service is not associated with any other service.

Each microservice adheres to the Database-per-Service principle, with each one having the full ownership of its database to be loosely coupled and autonomous in accessing the database. It is independently deployable, is scaled independently, and has simplified maintenance and does not explicitly make database dependencies on services. Interservice is achieved using synchronous REST API, to offer immediate business functionality, and, asynchronous event-driven messaging, to offer long business functionality. The services not only implement their local services, but all maintain a local transaction, and thus in the overall management of the entire payment process, the business has to be coherent in terms of the transactions. This duty is passed on to the Saga Orchestration Engine which oversees the execution of transactions of all the involved services without global locking of databases.

3.3 Saga Orchestration Engine

A main coordination mechanism of the proposed framework is the Saga Orchestration Engine. The framework also uses orchestration-based Saga approach, rather than traditional distributed transaction protocol, as the increasingly local transactions across a distributed system are synchronized and blocking behaviour is not enforced nor a centralized locking mechanism used. The Saga Orchestrator keeps track of the entire process of the payment process and specifies the subsequent business after each successful transaction.

When there is a request for payment, the orchestrator requests Customer Authentication Service, account validity, payment authorization, fraud reporting, merchant validation, payment settlement, notification and recording an audit. The database operation of the individual services is done independently and a successful response is then provided to the orchestrator. Once all the services participating in the transaction have fully executed their activities; the transaction is said to be committed and closed.

Delay in communication, network partition, application crash, congestion of resources or unavailability of a service may tear them down. When this occurs, the Saga Orchestrator will immediately complete some counter balancing transactions in reverse chronological order on all those services that have been completed successfully. One example is where merchant settlement is yet to be realized following a customer funding that is debited. The orchestrator goes ahead to automatically copy the customer account, reverse ledger transactions and alter audit records and inform all services that are already affected. This compensation scheme guarantees uniformity in the business and eliminates the incompletely performed

transactions and also guarantees availability of system.

3.4 Event-Driven Communication Layer

The suggested structure will improve scalability and service dependencies through the use of an event-based communication strategy based on Apache Kafka as an event broker distributed. Avoided also to base on the synchronous service-to-service communication alone, each business operation issue completed spreads an occasion in the Kafka subjects. Paid, Customer Verified, Balance Registered, Fraud Check Successful, Merchant Accepted, Settlement Complete, Payment Successful in the transaction process and Settlement Fail are some of the key events in transactions lifecycle.

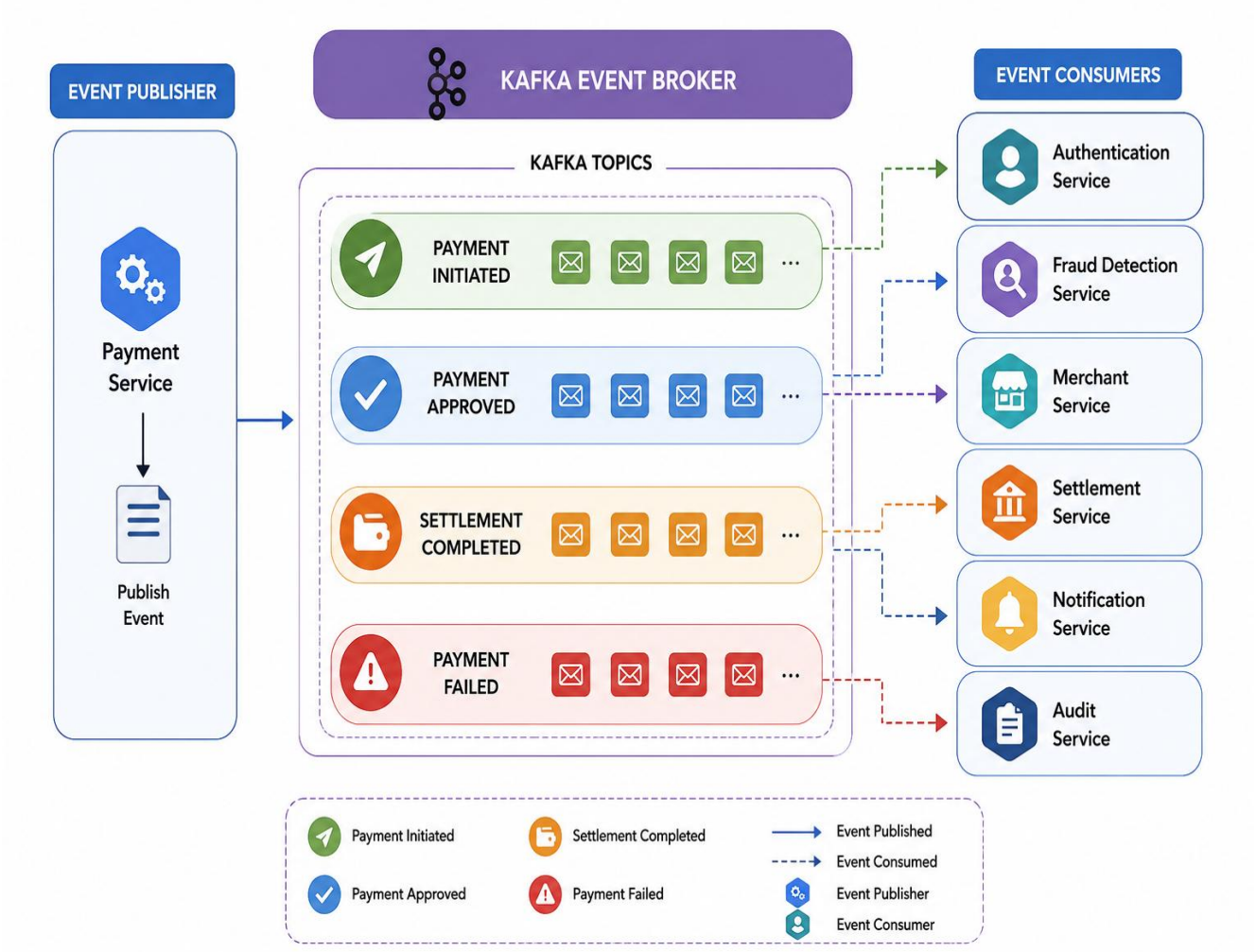

Figure 2- Event-Driven Communication Architecture

In fact, microservices subscribe to only those events that they are relevant to and receive incoming messages on their own. This type of publish/subscribe pattern has a loose and asynchronous relationship among the services, fault tolerance, and is also scalable in terms of rates of transactions in a horizontal way. Furthermore, Kafka has an event log which may at any time be replenished once some temporary errors have taken place as this will help ensure the occurrence of events will be delivered on a reliable basis and it will be simpler to recover transfers. Event-based communication can therefore enable the structure to effectively process huge amounts of concurrent payment requests without introducing bottlenecks to communication.

3.5 Transactional Outbox and Idempotency

Consistent event publication is an important need of distributed transaction processing. Transactional outbox Pattern is the one that will write the business updates and outgoing events of the same transaction

in the database in the proposed framework. It does not change a database and publishes events, but when an individual service writes the events to an outbox special table at the time it is first written. A background public is a periodic scan of outbox similar to that of a background publisher that goes a step ahead to publish an event to the Apache Kafka. This kind of mechanism ensures that any committed transactions cannot ever fail to see the transaction events hence irregularities can never occur because of application program failure or burst of networks experienced when publishing the message.

The framework also runs Idempotent Requests Processing that does not allow duplication of transactions being done. A different Idempotency Key is given to every client request to allow services to determine whether they have been dealt with or not. Repeated requests because of re-tries by any client or network breakdown or even a repeat in message delivery will present the responses, which already had been saved, rather than re-running business logic. It results in the prevention of customer debits and merchant settlements and ensures the irregular financial records by avoiding it in an effective manner.

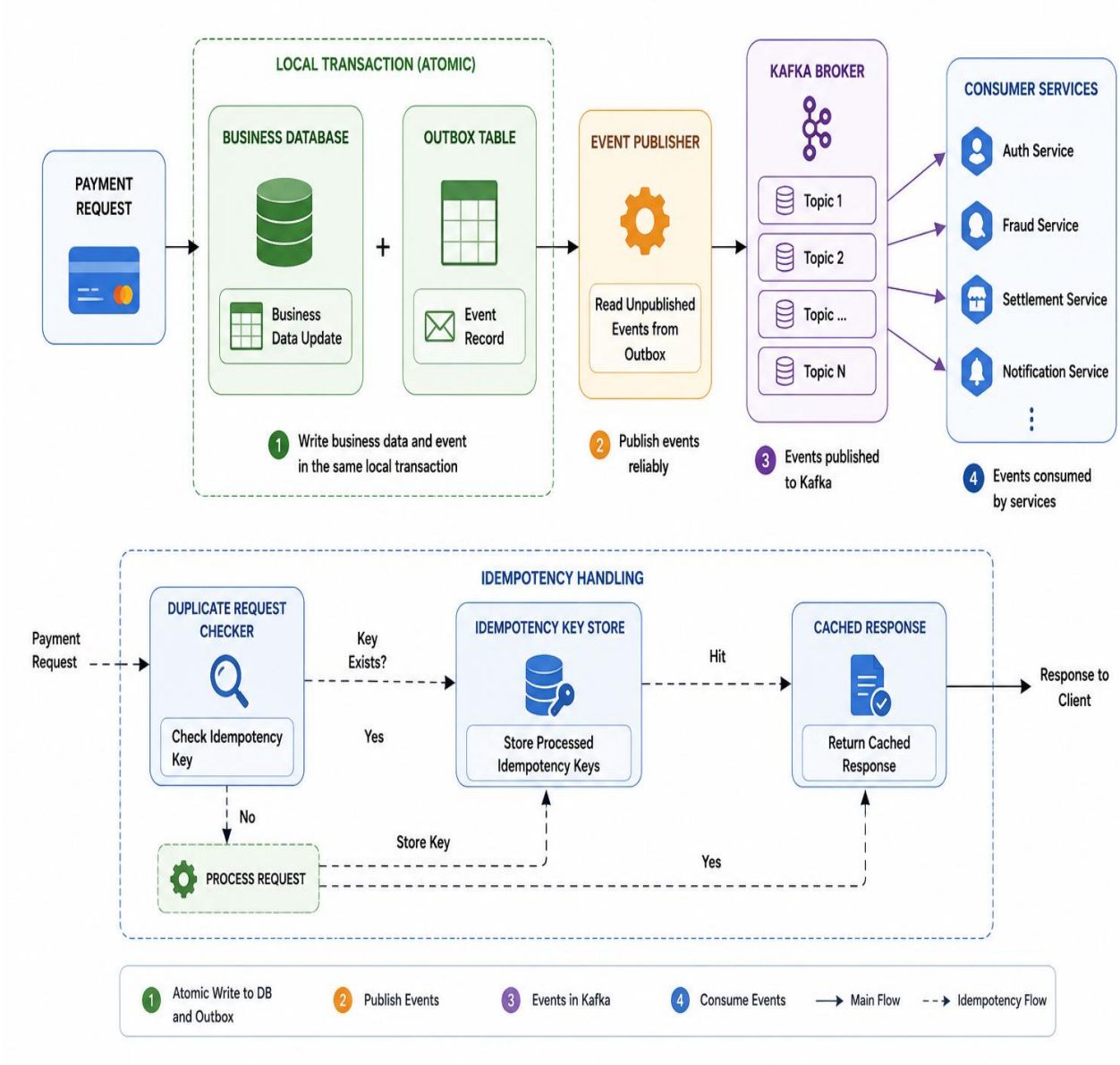

Figure 3- Transactional Outbox and Idempotency Framework

3.6 Fault Tolerance and Recovery Layer

The systems applied in financial transactions do not fail in case of a failure in the infrastructure, temporary breakdowns in services and disconnection in communication. The resilience mechanisms proposed combine several and complementary mechanisms to achieve high availability. Circuit Breakers constantly check the health of the service, and block communications to failing services temporarily, so the effects of cascading failures can not spread through the distributed system. As soon as the service has rebounded then normal communication will automatically occur.

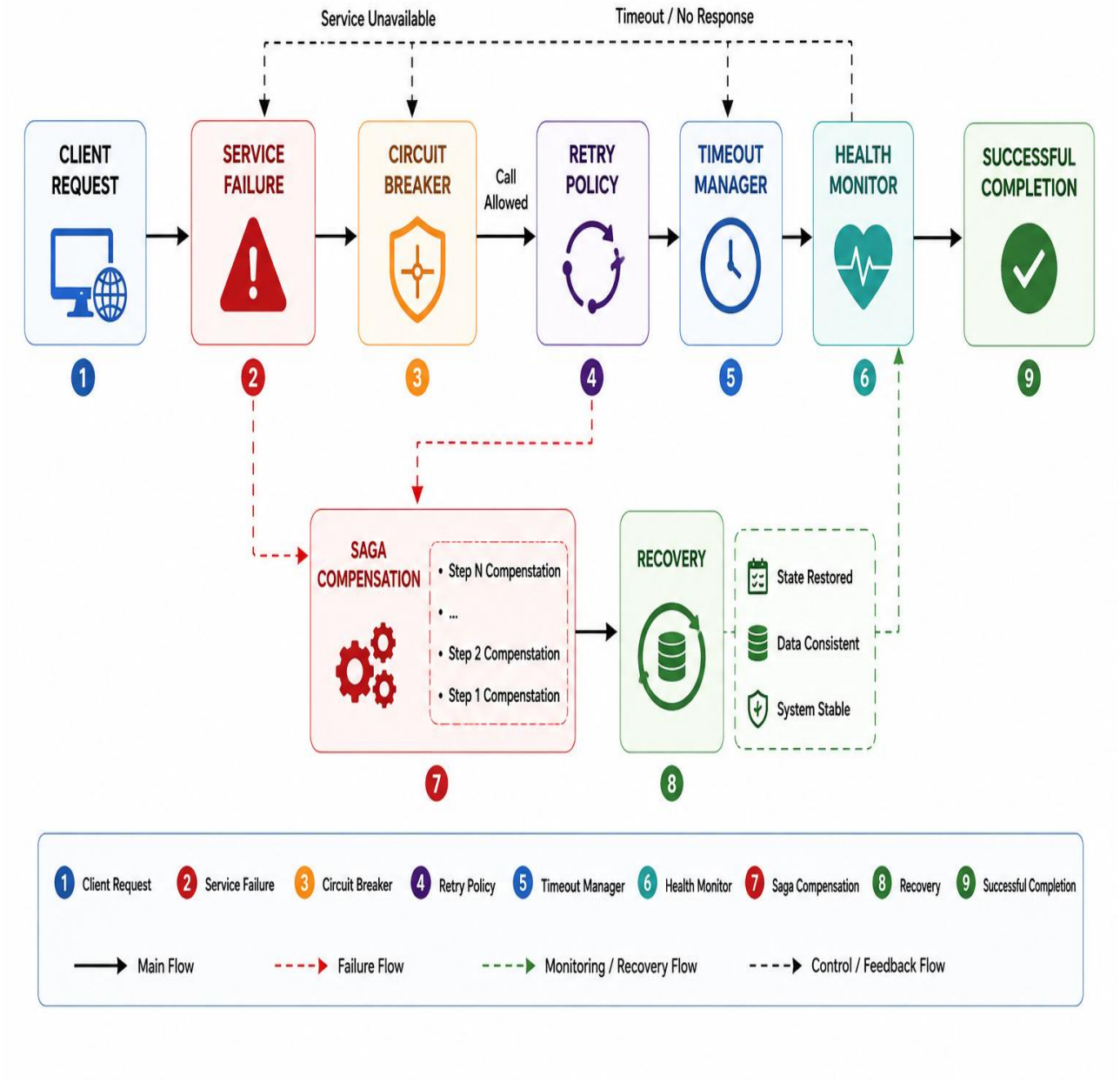

Figure 4- Fault Tolerance and Recovery Framework

Retry Policies Policies are automatically set with short lived failed operations whose exponential backoff algorithms have undefined behaviors, much of the unnecessary transaction failures, caused by short-term network congestedness or service overload are avoided. Timeout Management Persistently attempts to service what appears unresponsive by cancelling long running requests

that require more time than the normal time frame and putting corrective action to the service. Continuous Health continuously monitors the health of the services as well as does a crude redirection of the requests of healthy service instances in case a failure is identified. The combination of these mechanisms along with Saga compensation ensures eventual consistency and on-going execution of payments with partially failed recovery automatically.

3.7 Distributed Observability Layer

To run large-scale payment platforms, a comprehensive system observability is necessary. The suggested framework thus integrates into an observability layer consisting of distributed tracing, centralized logging, metrics collection, performance monitoring as well as real time operational dashboards. All transactions are given a globally unique Trace ID which is distributed amongst all serving microservices to allow the administrators to reconstruct whole transaction execution paths.

Centralized logging Application logs are stored of all the services in one central store and are, therefore, easier to debug, analyse in a forensic way and probe to find out the actual cause of the problem. Performance measures like transaction throughput, response time, CPU utilisation, service availability, number of retries, number of compensations and failures will be continually monitored and shown in the form of operation dashboards. The real time visibility will facilitate the identification of bottlenecks, anomalous behaviour and degradation of service in operations and bumper time in the production process on a timely basis and enhancement in reliability in operations and recovery time in cases of incidents.

3.8 Security and Compliance Layer

Security is part of all the layers of the proposed framework to guarantee protection of sensitive financial data and to comply with the regulatory requirements. The OAuth 2.0 framework with JSON Web Tokens (JWT) is authenticated and un authenticated clients and authorized services are the only ones that can access the protected resources. Another limitation of administration action is Role-Based Access Control (RBAC)- based on the established organizational policies and user privileges.

The transmission of data occurs between all the services and it is encrypted with the help of the Transport Layer security (TLS) and destruction and leakage by any person that is not authorized are eliminated. Encryption of sensitive data of customers, payment data as well as financial information is done through AES-256 encryption with digital signatures to ensure authenticity and integrity of the message. The unalterable audit logs routinely record the events of all transactions, administrative and system modifications endlessly, which aids in quantifying the provisions of PCI-DSS, ISO 27001, financial auditing and organization controlling systems.

3.9 Scalability and Cloud Deployment

The framework suggested will be specifically scaled down to that of cloud-native deployment where workloads of transactions vary on a regular basis stipulating on the customer demand. The Kubernetes orchestration implements microservices in free standing Docker containers. With the introduction of stateless services, it will be possible to deploy additional instances of services in a short duration without impacting transaction processing.

Scaling of computer resources using incoming transactions is a dynamic algorithm called Horizontal Pod Autoscaling that makes use of computer power, memory and incoming transaction volume. Load balancers distribute load to multiple instances of a service, and make sure that the service replicas are not overloaded during peak periods of the year. This is realized by providing high-availability and fault-tolerance along with elastic scaling with Kafka cluster and distributed caches and container-orchestration which enables thousands of payment operations at a time without any impact on performance.

3.10 Framework Workflow

The workflow of the proposed framework starts with a request for a payment made by a client who calls API Gateway. Authentication, authorization and request validation is done and a unique ID of the Transaction and Idempotency key is created and transmitted back to the Saga Orchestrator through the gateway. The orchestrator creates the necessary microservices based on a specified order so that a particular microservice asks a local transaction and modifies its own specific dedicated database on its

own. Once all the local transactions have been successfully completed, an event is sent to the transactional outbox whereby every relevant service captures the event which is published into the Apache Kafka. Downstream services on the other hand, consume such events and the workflow of the execution runs up to the point of receiving payment. Once a service has failed in its effort to perform an execution; the Saga Orchestrator will automatically propagate compensating transactions that will roll-back already completed transactions and restore the business. Throughout the life cycle of transactions, system behaviour is continuously monitored using distributed tracing, centralized logging, health check and performance dash boards. On a successful completion-compensation, the ultimate transaction status will finally be recorded at the audit repository and is completely traceable with accountability and regulatory compliance. The Reliable Distributed Transaction Framework (RDTF) proposal is one such pointer where distributed transaction coordination, event-driven communications, automated recovery, complete observability, high-level security, and a cloud-native scalable solution are embedded within a single architecture to deliver a highly reliable solution to the modern microservices-based payment system that must be able to run under the load of not only high volume but also under the demands of mission conditions of risk.

4. Framework Evaluation

In order to measure the suitability of the suggested Reliable Distributed Transaction Framework (RDTF), an in-depth performance study was done with the help of representative microservice-driven payment loads. This test was done against a backdrop of the capability of the framework to maintain consistency and resist service failures, scale under workload and do the transactions with very low latency and at high consistency. The framework has been tested on both normal operating state and failure environments like network failure, service failure and temporary communication failure. It was also assessed regarding functional correctness or system-level performance, to ensure that the proposed architecture is capable of being reliable to the requirements of the current payment platforms.

4.1 Experimental Setup

The proposed framework was deployed in a cloud-native system that was a combination of a few Docker containers, which were orchestrated with Kubernetes. Payment authorization, customer authentication and account management, fraud detection settlement, notification and audits were single microservices. An asynchronous communication event with Apache Kafka event broker was offered and a database with respective micro-services in Database-per-Service architecture was controlled separately. Distribution of transactions was done by the Saga Orchestrator, and monitoring of system performance and operational measures was done using Prometheus, and Grafana. Simulations of realistic enterprise payments were created by generating a number of active payment requests with different amounts of transactions in them.

4.2 Performance Metrics

The framework was equally tested based on the various quantitatives of performance whereby, performance of the payment systems is anchored on its requirements. The latency in handling a transaction The average time, end-to-end, to process a payment transaction was the average time to process an end-to-end core payment when making an initial payment request and final confirmation. The authors calculated the success of the transactions with the payment requests as a ratio of payment requests carried out successfully per second. The consistency rate was applied to gauge the rate of execution of a transaction with correct and consistent final state. Fault recovery time was the time taken to achieve business consistency after service failures. Service availability, success rate in compensation, effectiveness in retry and reliability in delivering messages were also used as other metrics to also test the overall robustness of the framework.

4.3 Consistency and Fault Tolerance Analysis

The experimental test establishes the fact that the Saga Orchestration Engine could be effective in the continuity of the business in the case of the considerably dispersed microservices. Transactional compensation of fraud detection and notification services in case of simulated failure in relation to

merchant settlement was automatically carried out to reverse previously counterpart transactions. There were no partially committed payment transactions left after the recovery and it means that the framework enabled the maintenance of eventual consistency. Isolating the malfunctioning services and re-attempting to redeliver the failed operation regardless meant that the idempotence of request processing which the Transactional Outbox Pattern was enforcing minimized losing messages when calls to publish events and from retaking requests or payments made by the same client were re-attempted due to a temporary service failure.

4.4 Scalability and System Performance

Scalability assessment revealed that the suggested framework was stable with increment in the transaction workloads. The microservices produced and packaged in a container were horizontally scaled therefore the system was able to scale up to the level where it can serve additional calls without drastically decreasing the response time. It minimized the synchronous relationships between services by the event-driven communication due to spreading online and being reduced by Apache Kafka, thus, handling the transactions in parallel and raising the overall throughput. Dynamic load balancing brings a balance between incoming traffic and assigning service instances to available instances in real time in order to prevent bottlenecks at resources during peak transactions. The microservices were also stateless, which enables them to distribute resources more readily, and spin up instances more quickly to meet growing demands of computation.

4.5 Discussion

The results of the analysis show that the Reliable Distributed Transaction Framework designed is a practical solution to the current payment systems that demand high rates of consistency, fault resilience and scale. Event based communication, transmission outbox, distributed tracing and automated compensation (which are provided by Orchestration) can be used to achieve a reliable execution of transactions and system maintenance in a high availability state compared to the older blocking transaction protocols. The embedded observability layer enhances the visibility of the operations and avails real time status of the transactions, status of the services and performance metrics. Furthermore, cloud-native implementation and independent scaling of services is supported in modular architecture making the framework applicable in large scale payment systems in an enterprise. On the whole, the experimental results prove that RDTF is able to resolve the key issues related to distributed transaction management and offer a stable base of the secure, resilient, and scalable microservices-based payment processing.

5. Conclusion and Future Work

The more widespread use of cloud-native microservices has contributed a great deal to the re-architecture of the current payment system with the introduction of independent deployment of the services provided, elastic scaling, and the continuous deployment of the software. However, trusted distributed transactions with several autonomous services continue to be a significant problem because the management of data is decentralized, communication is asynchronous, network failures do exist and transactions can be half-executed. This article introduced the Reliable Distributed Transaction Framework (RDTF) as an all-encompassed model towards consistency, fault tolerance and scalability of a microservice hosted payment system. The saga orchestration, event-driven communication, outbox in which all the transactions are taken and also idempotent request processing methods, distributed tracing, automated compensation systems, circuit breakers, policies on retry and centralized monitoring and enormous security controls have been incorporated in the proposed framework leading to a single structure. The architecture offers eventual consistency with high availability and operation resilience through replacement of blocking distributed transaction protocols with non-blocking coordination through protocols based on Saga. The assessment shows that RDTF is a good way to reduce transaction failure, and duplicate payment execution and enables automatic recovery with partial failures of the service as well as its good operation with the rise of the transactional workload. Moreover, the layered architecture is enhanced to enhance loosely coupled implementation, scaling, less maintenance, and easy-to-deploy in cloud-native infrastructure by orchestrating containers. All these features make the proposed framework a viable answer to enterprise payment systems that should be carried out with

high availability, reliable and secure transaction processing.

Although the proposed framework will be a great starting point when it comes to dealing with distributed transactions, there are multiple opportunities for future research as well. Intelligent steering of transactions and identifying failures with the assistance of the artificial intelligence and machine learning strategies, identification of anomalies, and adaptive recovery solutions will be implemented in the future during work. Use of blockchain solutions and smart contracts would have the potential in improving the integrity of transaction, audit as well as confidence of cross organization payment ecosystem. Future studies can seek to focus on exploring multi-cloud and hybrid-cloud deployment to enhance geographical resiliency to faults and disaster recovery ability. The framework could also be further expanded to text zero-trust security architectures, secrets of computing and privacy like federated learning to enhance security of sensitive financial data. Further, experimenting with RDTF with real world setting of an enterprise payment data, a large-scale testing base and the latest technologies such as serverless computing and edge based payment processing will provide even greater insight into its overall usability. These future developments will further enhance reliability, scaling, intelligence and security of distributed payment systems in more complex financial realms.

References

[1] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD '87)*, San Francisco, CA, USA, May 1987, pp. 249–259.

[2] C. Richardson, *Microservices Patterns: With Examples in Java*. Shelter Island, NY, USA: Manning Publications, 2019.

[3] M. Štefanko, O. Chaloupka, and B. Rossi, "The Saga Pattern in a Reactive Microservices Environment," in *Proceedings of the 14th International Conference on Software Technologies (ICSOFT)*, Prague, Czech Republic, Jul. 2019, pp. 483–490.

[4] A. Bucchiarone, N. Dragoni, S. Dustdar, P. Lago, M. Mazzara, V. Rivera, and A. Sadovykh, *Microservices: Science and Engineering*. Cham, Switzerland: Springer International Publishing, 2019.

[5] W. Hasselbring and G. Steinacker, "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce," in *Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Gothenburg, Sweden, May 2017, pp. 243–246.

[6] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Communications of the ACM*, vol. 56, no. 5, pp. 55–63, May 2013.

[7] H. Uyanik and T. Ovatman, "Enhancing Two Phase-Commit Protocol for Replicated State Machines," in *Proceedings of the 2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, Västerås, Sweden, Mar. 2020, pp. 118–121.

[8] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Transactions on Database Systems*, vol. 11, no. 4, pp. 378–396, Dec. 1986.

[9] X. Limón, A. Guerra-Hernández, Á. J. Sánchez-García, and J. C. P. Arriaga, "SagaMAS: A Software Framework for Distributed Transactions in the Microservice Architecture," in *Proceedings of the 2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*, San Luis Potosí, Mexico, Oct. 2018, pp. 50–58.

[10] N. Narkhede, G. Shapira, and T. Palino, *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*, 1st ed. Sebastopol, CA, USA: O'Reilly Media, 2017.

[11] J. Kreps, "Kafka: A Distributed Messaging System for Log Processing," in *Proceedings of the NetDB Workshop*, Athens, Greece, 2011.

[12] R. Petrasch, "Model-Based Engineering for Microservice Architectures Using Enterprise Integration Patterns for Inter-Service Communication," in *Proceedings of the 14th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, Nakhon Si Thammarat, Thailand, Jul. 2017, pp. 1–4.