Enhancing Banking Transactions with Blockchain Technology: From Traditional MVC/SOAP API to Hyperledger Fabric.

Mohammad Najm

Final Thesis Report

DECEMBER 2024

DEDICATION

This research is dedicated to all those who have been instrumental in supporting me throughout this journey. To my family, whose love and support have made this work possible, I am deeply grateful. To my mentors, whose guidance and wisdom have been invaluable in shaping my intellectual growth, I offer my sincere gratitude. To all my colleagues and peers who have assisted me in any capacity, I extend my heartfelt thanks. This work is a testament to their unwavering support and encouragement.

ACKNOWLEDGEMENT

I would like to express my deepest gratitude to my thesis supervisor, Dr. Shashidhara R, for his invaluable support and guidance throughout this endeavor. I also thank the team at Liverpool John Moores University, which is located in Liverpool, United Kingdom, and Team Upgrad Education for their unwavering support throughout my journey. Finally, I extend my heartfelt appreciation to my parents and peers for their encouragement and assistance in helping me complete this projects.

ABSTRACT

Modern banking systems predominantly rely on microservices-based architectures that offer modularity, scalability, and real-time processing but still depend on centralized data coordination, leading to audit limitations, fault vulnerability, and governance complexity. Although recent research suggests that permissioned blockchain frameworks such as Hyperledger Fabric may address these challenges, prior studies often lack real-world banking implementations or direct comparisons with traditional systems. This thesis investigates the enterprise readiness of Hyperledger Fabric by developing and evaluating two full-stack transactional systems: one based on Spring Boot microservices, and the other on Hyperledger Fabric deployed across 15 operational nodes simulating three banks and a central audit authority. Both systems were tested using JMeter and monitored via Prometheus and Grafana to assess performance under identical conditions. Results show that while the microservices system achieved higher peak throughput (880 TPS), it exhibited higher fault recovery time (4.8s) and less deterministic consistency. Hyperledger Fabric, by contrast, delivered stronger auditability, multi-organizational governance, and lower fault recovery time (2.1s), albeit with a higher average latency (710ms). By addressing a key research gap in comparative performance and operational viability, this study provides empirical evidence and practical guidance for financial institutions considering the transition from centralized systems to distributed ledger technologies.

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

API	Application Programming Interface
MVC	Model-View-Controller
SOAP	Simple Object Access Protocol
SWIFT	Society for Worldwide Interbank Financial Telecommunication
RTGS	Real Time Gross Settlement
AWS	Amazon Web Service
MSP	Membership Service Provider
SOA	Service-Oriented Architecture
TPS	Transactions Per Second
DLT	Distributed Ledger Technology
REST	Representational State Transfer
JVM	Java Virtual Machine
CA	Certificate Authority

TABLE OF CONTENTS

CHAPTER 1 :

INTRODUCTION

1.1 Background

Modern banking systems are predominantly built on microservices-based architectures. These allow services to operate independently, offering better scalability and modularity compared to monolithic systems. In practice, microservices are deployed in containers, communicate through APIs, and are typically backed by centralized databases. This setup is widely used due to its real-time transaction processing and ability to integrate with legacy systems. For example, Mallidi, Sharma, et al. (2022) explored how streaming microservices enhanced fraud detection in digital financial services. Despite their operational benefits, these systems remain susceptible to centralized failure, orchestration complexity, and inter-service dependency issues.

These architectural challenges are especially critical in financial domains, where consistency, privacy, and fault tolerance are non-negotiable. While service-oriented architecture (SOA) and microservices improve modularity, they fail to resolve the issue of trust and transparency in multi-organization scenarios. As Konkin and Zapechnikov (2021) argues, traditional financial systems lack intrinsic support for decentralized consensus or tamper-proof audit trails, often requiring additional middleware to bridge compliance gaps.

Blockchain has emerged as a potential alternative. Unlike microservices, blockchain systems distribute state across peers and commit transactions through consensus protocols. Enterprise-grade frameworks like Hyperledger Fabric address scalability and privacy concerns by enabling permissioned participation, pluggable consensus, and private data collections. Quan, Wahab, et al. (2024) demonstrated that Fabric allows stakeholders to transact with fine-grained access control while maintaining auditability. Its ability to encode governance rules into smart contracts and isolate transaction scopes across channels makes it suitable for regulated environments.

The convergence of blockchain and banking is gaining momentum. Blockchain

enables programmable money, transparent audit trails, and near real-time clearing across institutions. As noted in Johnson, Dandapani, et al. (2024), smart contracts automate payment logic, enabling seamless settlement of financial instruments. Moreover, Quan, Wahab, et al. (2024) highlights its successful deployment in insurance and supply chain use cases, which share operational constraints with banking. However, limited empirical research compares such blockchain frameworks directly against microservices under controlled, banking-like conditions.

1.2 Problem Statement

1.2.1 The Problem

Current banking systems rely on centralized backend architectures such as MVC patterns, monolithic databases, and SOAP APIs. These have served traditional operations well but now face constraints related to transaction throughput, resilience, and security, especially under high concurrency or cross-institutional use cases. As Mallidi, Sharma, et al. (2022) emphasized, fraud detection pipelines and streaming analytics often exceed the bounds of monolithic scaling. Blockchain promises decentralization and integrity, but practical deployment raises new concerns regarding configuration, integration, and regulatory adaptation (Quan, Wahab, et al., 2024).

1.2.2 Knowledge Gaps

1. **Lack of comparative architectural analysis:** Most prior work evaluates either blockchain or microservices in isolation. Konkin and Zapechnikov (2021) identifies performance metrics in Fabric but does not compare them with REST-based systems.
2. **Missing operational simulations:** Existing literature lacks realistic multi-bank blockchain deployments. Few simulate endorsement policies, observer peers, or hybrid data architectures.
3. **Insufficient focus on governance and auditability:** Integration with cen-

tral financial authorities like Reserve Banks is not addressed, especially under permissioned networks.

4. **Low-node scalability testing:** Most papers test Fabric on 2 to 5 nodes. Our study uses 15 nodes (3 banks ×5 peers), matching small-scale industry settings.

1.2.3 Problem Statement

This research investigates whether Hyperledger Fabric can replace or augment microservices architectures for banking transactions. It addresses the lack of comparative benchmarking between REST-based microservices and blockchain under real-world transaction volumes. By deploying both systems, simulating banking operations, and monitoring metrics such as throughput, latency, and failure recovery, the study aims to clarify Fabric's enterprise viability.

1.3 Research Questions

1. Is Hyperledger Fabric a viable alternative to microservices in regulated banking?
2. What limitations arise when deploying each architecture under load?
3. Can Fabric support observability, compliance, and governance at operational scale?

1.4 Aim and Objectives

1.4.1 Aim

To compare a blockchain-based banking architecture (Hyperledger Fabric) with a traditional microservices setup (Spring Boot) in terms of performance, scalability, and operational complexity.

1.4.2 Objectives

- Implement a microservices-based banking system using Spring Boot, Docker, and MySQL.

- Deploy a Hyperledger Fabric network with 15 nodes simulating 3 banks and a CentralBank observer.
- Measure throughput, latency, and fault tolerance using JMeter and Prometheus.
- Evaluate chaincode flexibility, endorsement governance, and configuration overhead.
- Provide deployment insights to assist blockchain adoption in financial systems.

1.5 Scope

1.5.1 In Scope

- Implementation of banking services on both architectures.
- Comparative benchmarking using real-time load testing.
- Chaincode development and policy enforcement.
- Observability and monitoring with Prometheus and Grafana.

1.5.2 Out Of Scope

- Cryptocurrency, mining, or public blockchain mechanisms.
- Deep integration with legacy core banking systems.
- Third-party compliance or regulatory API integration.

1.6 Significance

This study introduces a practical deployment of Hyperledger Fabric for inter-bank transaction handling across 15 nodes—a scale not commonly tested in academic evaluations. While many studies simulate blockchain capabilities, they often do so in abstract or low-node settings. This work builds on real-world architectural constraints using Dockerized services and live performance instrumentation. It advances the work of Konkin and Zapechnikov (2021) and Quan,

Wahab, et al. (2024) by embedding governance and scalability concerns into performance evaluation, demonstrating how fabric's modularity and endorsement flexibility can be leveraged for auditability and institutional control. Its implications are global in scope, offering insights into the design of Central Bank Digital Currency (CBDC) systems, decentralized interbank clearing, and secure public financial infrastructure. For academia, this research fills a measurable performance gap. For industry, it provides actionable recommendations to evaluate blockchain as a mainstream backend technology.

Contribution Area	Description	Relevance
Practical Implementation	15-node Hyperledger Fabric network simulating interbank operations	Validates scalability and deployment feasibility
Architectural Benchmarking	Head-to-head comparison with Spring Boot microservices under identical load	Quantifies real-world trade-offs
Observability Integration	Real-time monitoring via Prometheus and Grafana	Supports auditability and governance
Policy-Driven Chaincode	Smart contracts for both intra- and inter-bank transfers	Demonstrates programmable trust
Academic Gap Bridging	Fills lack of comparative studies in blockchain vs. microservices context	Advances academic literature

Table 1.1: Research Contributions Summary

1.7 Structure

The thesis is organized as follows:

- **Chapter 2: Literature Review**
  This chapter surveys foundational and contemporary research on banking architectures, blockchain, and microservices. It categorizes architectural styles, highlights performance evaluations from prior work, and identifies research gaps addressed in this thesis.
- **Chapter 3: Research Methodology**
  This chapter outlines the experimental design including data generation techniques, system architecture, and load-testing configuration. It explains the tools used for benchmarking and the rationale for each methodological decision.

- **Chapter 4: Analysis and Implementation**
  This chapter details the implementation of both microservices and Hyperledger Fabric systems. It includes architecture diagrams, Docker configurations, chaincode snippets, and integration of monitoring stacks.
- **Chapter 5: Results and Discussion**
  This chapter presents the collected performance metrics and analyzes the comparative strengths and weaknesses of each system. It evaluates scalability, latency, failure recovery, and governance implications under identical workloads.
- **Chapter 6: Conclusion and Future Work**
  This chapter summarizes the key findings, reflects on study limitations, and suggests pathways for enhancing blockchain adoption in financial services.

CHAPTER 2 :

LITERATURE REVIEW

2.1 Introduction

This chapter presents a detailed examination of the literature relevant to the evolution of banking transaction systems and the advent of blockchain-based infrastructures such as Hyperledger Fabric. The purpose of this literary review is to systematically explore the historical, technical, and conceptual development of banking transaction architectures, trace their challenges, and understand the motivations behind the emerging shift toward decentralized systems. The objective is not only to provide a critical synthesis of existing knowledge but also to identify gaps that justify and frame the research focus of this study.

The approach adopted for the literary review was structured and thematic, centering on the continuous evolution of transactional infrastructures in the banking sector. An initial broad exploration was conducted to chart the historical journey of banking transaction systems, beginning with legacy monolithic architectures and progressing toward microservices-based designs. Following this, the review narrowed its focus toward the emergence of decentralized transaction systems and enterprise blockchain solutions, with a particular emphasis on Hyperledger Fabric. Alternative blockchain frameworks were also reviewed to situate Hyperledger Fabric within the broader technological landscape and to evaluate the rationale behind its selection for banking applications.

In total, twenty peer-reviewed papers, technical reports, and scholarly articles were critically analyzed to construct this review. The literature was selected based on its relevance to the technical, operational, and architectural dimensions of banking transactions, ensuring a balanced coverage of traditional, transitional, and contemporary approaches. The findings of some of the relevant papers is given in the table 2.1 illustrated below.

The process of literature selection involved a three-stage review methodology: an initial identification of sources through database searches and reference mining, a screening process to assess relevance and quality, and a final synthesis

Study Name	Reason for Inclusion	Key Inference
(Bernstein and Newcomer, 2009)	Defined the core structure and guarantees of transactions in centralized banking.	Established the baseline definition for transactional control in all systems.
(Gray and Reuter, 1993)	Provided foundational concepts in concurrency and recovery for financial systems.	Informed error handling, state rollback, and concurrent execution in both architectures.
(Pavlovski, 2013)	Highlighted operational difficulties in channel-isolated banking services.	Identified the fragility of legacy interfaces across banking channels.
(Appandairaj and Murugappan, 2013)	Explored modularization via SOA and its limits in banking applications.	Provided a benchmark to judge the improvement in modularity offered by microservices.
(Ganesan and Vivekanandan, 2009)	Outlined secure architectures suitable for web-enabled banking platforms.	Demonstrated early-stage online banking architecture constraints still seen today.
(Rabah, 2019)	Identified architectural weaknesses of legacy systems in compliance and scale.	Supported the need to improve auditability and modular governance.
(Newman, 2015)	Established microservice design rationale for modular, scalable systems.	Reinforced the modular development of independent services for high availability.
(Dragoni, Giallorenzo, et al., 2017)	Discussed evolution of microservices with emphasis on message-driven systems.	Helped understand service separation and async processing needs in banking.
(Walls, 2017)	Demonstrated the feasibility of using Spring Boot for microservice banking apps.	Provided implementation reference for REST-based systems and observability.
(Fowler and Lewis, 2014)	Outlined drawbacks of microservices in coordination and deployment complexity.	Warned about operational complexities introduced by microservice dependencies.
(Nakamoto, 2008)	Introduced decentralized consensus as an alternative to centralized control.	Established the theoretical foundation for distributed ledger use cases.
(Zheng, Xie, et al., 2017)	Compared blockchain types and proposed permissioned models for institutions.	Supported the enterprise case for using permissioned blockchain networks.
(Androulaki and al., 2018)	Validated Fabric's architecture for multi-party transaction processing.	Confirmed the technical feasibility of Fabric for real-world interbank use.
(Cachin, 2016)	Explained the separation of transaction stages in Fabric for enterprise control.	Justified the decision to use endorsement-based transaction lifecycle separation.
(Baliga, Venkatesan, et al., 2018)	Benchmarked Fabric under load and optimized endorsement conditions.	Proved Fabric could scale with tuning and was sensitive to policy settings.
(Pongnumkul, Siripanpornchana, et al., 2017)	Compared Fabric and Ethereum in terms of resource efficiency and throughput.	Reinforced Fabric's low CPU usage under regulated loads and operational variance.
(Wüst and Gervais, 2018)	Provided a decision framework to determine blockchain suitability for banks.	Enabled validation of Hyperledger as appropriate where central trust is weak.

Table 2.1: Literary Review Studies and Their Findings

phase where thematic patterns were distilled. This methodological flow is depicted in the diagram below, representing the structured and iterative nature of the review process illustrated in the figure 2.1 below:

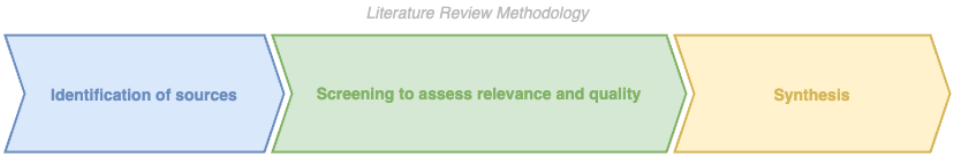

Figure 2.1: Literary Review Process

The structure of this chapter reflects the evolutionary progression of transaction systems under consideration. Initially, it contextualizes the notion of transactions within the banking sector, proceeds to examine legacy systems, and analyzes the inherent challenges posed by monolithic architectures. Thereafter, it investigates the paradigm shift towards microservices, which leads to an assessment of decentralized systems, with a particular focus on Hyperledger Fabric. Subsequently, a comparative analysis of alternative blockchain frameworks is conducted, ultimately identifying a research gap that delineates the study's unique contribution.

2.2 Concept Of Transactions In Banking

Transactions form the fundamental building blocks of all banking operations. A banking transaction can be broadly understood as a set of activities that involve the transfer of monetary value, recording obligations, and ensuring the integrity and consistency of financial data across systems. In the traditional sense, transactions are governed by the principles of atomicity, consistency, isolation, and durability, collectively referred to as the ACID properties, which ensure that either all operations within a transaction are successfully completed or none are, thereby maintaining data integrity (Gray and Reuter, 1993). These properties have historically been critical for the trustworthiness of banking systems, where the failure of even a single transaction could result in significant financial discrepancies.

The study of transaction models has evolved extensively with the changing na-

ture of banking. Initially, transaction models were simple and linear, largely synchronous in nature, operating within tightly coupled mainframe systems (Bernstein and Newcomer, 2009). These traditional models emphasized strong consistency, wherein a transaction had to complete fully before the system would reflect its effects, even if this resulted in delays or limited scalability. As observed in the study by Pavlovski (2013), banking environments relied heavily on branch-centric models where transactions initiated at one point would have to pass through central processing units before becoming available for other operations, a structure that prioritized reliability over responsiveness.

The conceptual modeling of transactions within banking systems is typically represented through various abstraction techniques. One of the common models is the state transition diagram, where a transaction is depicted as moving through states such as initiated, authorized, completed, or failed. As per the work of Appandairaj and Murugappan (2013), the use of UML activity diagrams further enriches the understanding of transaction flows by detailing the sequential and conditional activities that govern a banking transaction. A typical UML representation of a banking transaction flow highlights the points of user authentication, balance validation, fund reservation, ledger update, and transaction confirmation, thereby offering a structured visualization of transaction lifecycle management.

In more contemporary frameworks, transactions are increasingly modeled as composite services, particularly in service-oriented banking architectures. According to the study by Ganesan and Vivekanandan (2009), modern banking transactions often comprise multiple micro-interactions, such as balance checking, fraud detection, authorization messaging, and ledger adjustment, all encapsulated within a broader transactional context. These micro-interactions are orchestrated to behave atomically from the perspective of the end-user but are distributed across multiple internal systems. Such an approach complicates transaction management but allows banks to achieve greater flexibility, fault tolerance, and horizontal scalability.

The rise of digital and online banking has significantly altered the expecta-

tions around transaction concepts. Customers now expect real-time settlement, continuous availability, and multi-device accessibility, factors that have pushed traditional transaction models toward more decentralized and eventually distributed paradigms (Rabah, 2019). As observed by Gilbert and Lynch (2002) in their study of distributed banking architectures, the modern transaction must not only uphold the core principles of integrity and consistency but also guarantee low-latency performance and seamless interoperability across banking services. This evolution has given rise to the necessity for transaction models that can balance the often conflicting requirements of consistency, availability, and partition tolerance, a challenge famously encapsulated by the CAP theorem.

In diagrammatic terms, modern transaction concepts in banking can be visualized using an extended UML sequence diagram. For instance, a simplified UML sequence for an online banking transaction today would involve entities such as the client application, the authentication server, the transaction processor, and the ledger database. The sequence would demonstrate asynchronous communication between services, multi-step authorization processes, and dynamic error handling paths to accommodate failures at any point. Such a visual representation not only clarifies the complexity inherent in current transaction systems but also reinforces the necessity for new paradigms capable of meeting the demands of digital banking infrastructures.

Overall, the concept of transactions in banking has transitioned from simple, monolithic operations conducted within controlled environments to complex, distributed activities that must contend with the realities of modern digital ecosystems. As per the findings of Zheng, Xie, et al. (2017) and Androulaki and al. (2018), this transformation has profound implications not only for the technological architectures that support banking transactions but also for the regulatory and operational frameworks that ensure their security and reliability. The continued exploration of transaction models, including decentralized and blockchain-based systems, represents an essential step in aligning banking operations with the evolving expectations of customers, regulators, and financial markets.

2.3 Review Of Legacy Transaction Systems

Legacy banking systems have historically relied on centralized monolithic architectures powered by mainframes and enterprise application servers. These systems were characterized by tightly coupled business logic, server-side rendering, and batch-based transaction processing. Applications were commonly built using early frameworks such as Enterprise Java Beans (EJBs), Java Server Pages (JSPs), and deployed on middleware platforms like Apache Tomcat or Oracle WebLogic. These platforms enabled stateful session management and transaction coordination using centralized relational databases.

Ganesan and Vivekanandan (2009) provided a hybrid architecture model for early internet banking that demonstrated a common pattern—transaction coordination occurred through centralized web application servers communicating with a backend Oracle or MySQL database. While robust, these systems required extensive manual scaling and often depended on specialized hardware configurations. The limitations became more pronounced as transaction volumes increased and real-time service expectations evolved.

The architectural rigidity of monoliths introduced challenges in upgrade cycles, code deployment, and fault isolation. Since all components shared the same memory and I/O resources, a failure in one module (e.g., transaction logging) could cascade into system-wide outages. Appandairaj and Murugappan (2013) discussed how SOA emerged as a strategy to decouple services via loosely bound web services, but observed that legacy systems still suffered from overlapping authentication and error-handling modules, leading to performance bottlenecks.

Furthermore, Rabah (2019) observed that legacy systems often lacked the capability to handle dynamic access control, versioning of contracts, or interbank audit visibility. As financial applications moved online, these systems struggled to maintain compliance with rapidly evolving data privacy and security regulations.

As shown in Figure 2.2, monolithic server-based architectures centered around a transaction processor layer that communicated with both the user interface and

backend ledgers. This structure provided low latency for internal operations but suffered from scalability and fault recovery limitations, especially in multi-bank clearing and settlement contexts.

2.3.1 Concept Of Monolithic Systems

The concept of a monolithic system in software architecture is historically attributed to early computer science practices where the entire software application was built as a single, indivisible unit. The formalized definition of monolithic architecture was first conceptualized in the context of early operating system designs in the 1960s, particularly with systems such as Multics (Corbató et al., 1965), which emphasized a single-tiered, unified structure. In banking applications, monolithic architectures took the form of large, self-contained programs in which all modules for transaction management, user interfaces, business logic, and database access were tightly interwoven. Any function within the application could directly invoke any other function, leading to minimal modularity but high internal performance.

The most prevalent architectural styles in monolithic systems included the three-tier architecture and the n-tier enterprise architecture. In a traditional three-tier model, systems were divided into the presentation layer (user interface), application layer (business logic), and database layer (data storage), but all layers were deployed as a single, unified executable or archive file (Taylor et al., 2009). Slight evolutions introduced additional service layers, but fundamentally, all the components remained within a single deployable unit. The following Figure 2.2 illustrates the monolithic design pattern commonly seen in legacy banking applications.

This diagram referred above would depict a Banking Application consisting of a Presentation Layer, Business Logic Layer, and Database Access Layer, all tightly coupled within a single deployable unit.

In this figure 2.3 we depict a sequence diagram, a Client Request interacts directly with a Monolithic Banking Server, which internally manages authentication, transaction processing, and database updates in a tightly sequenced op-

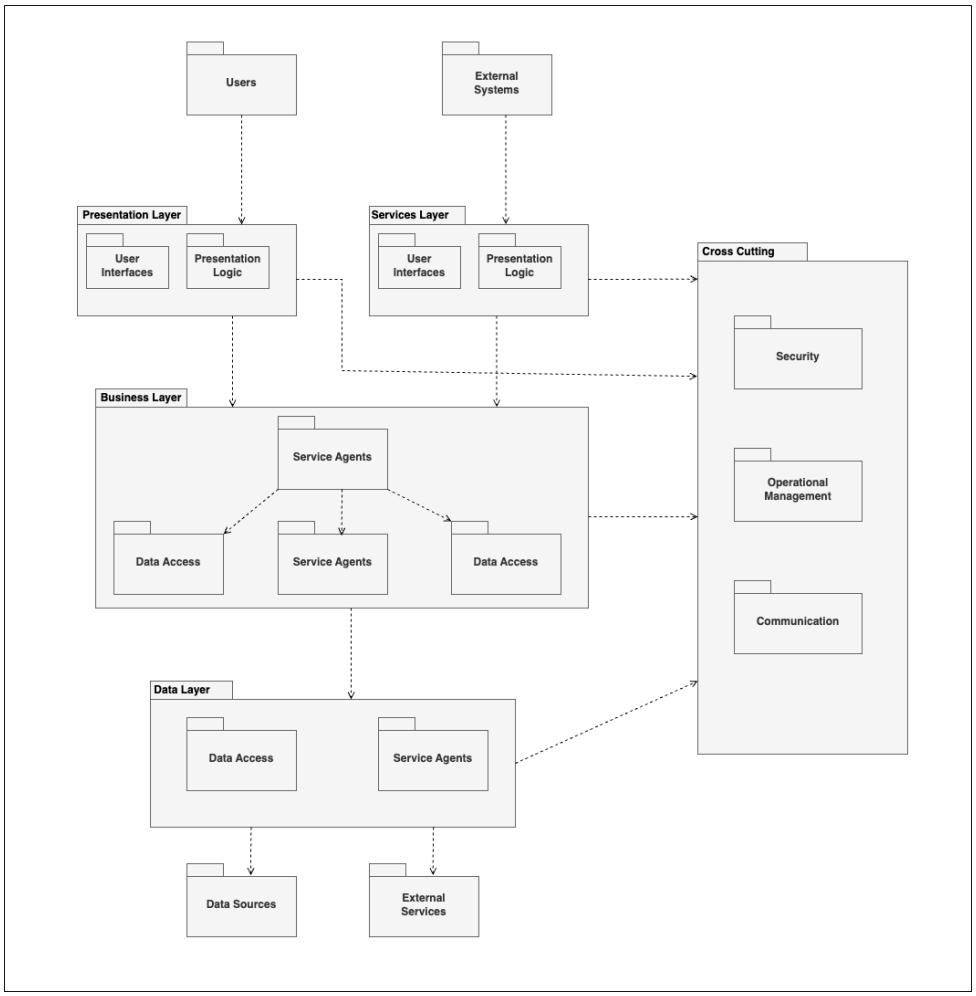

Figure 2.2: Monolithic Multi-Layer Banking System Architecture

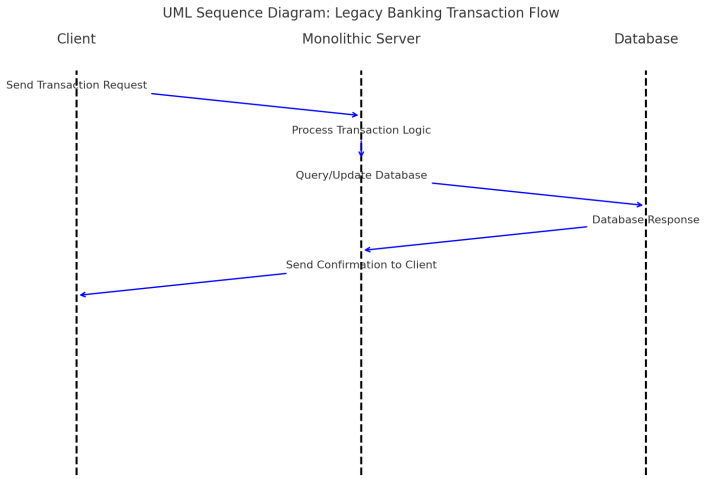

Figure 2.3: Sequence Diagram for a Legacy Transaction

eration without external service decomposition.

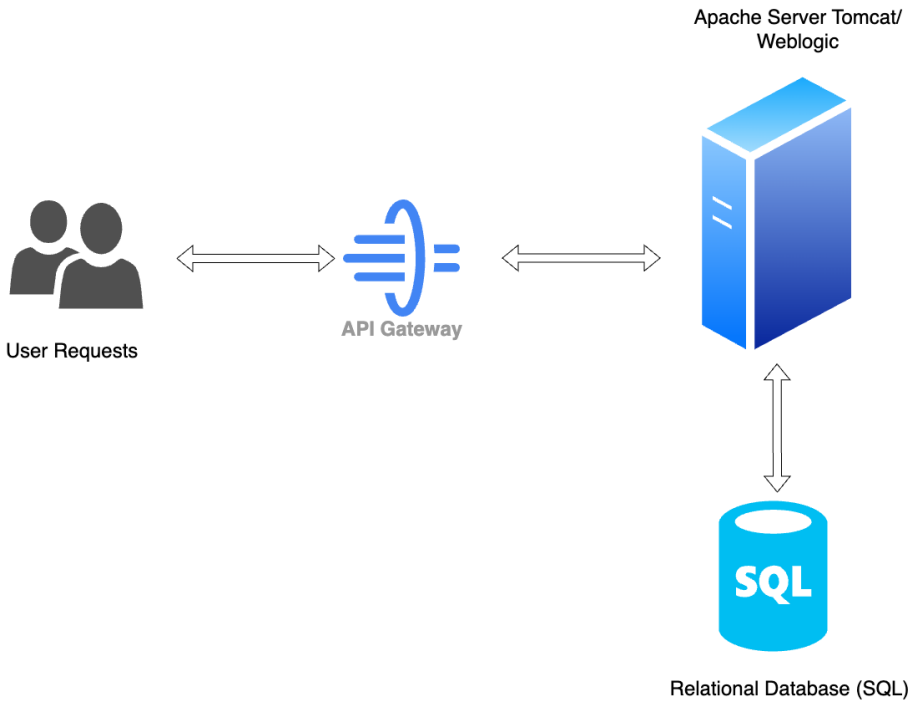

Figure 2.4: Deployment Diagram: Monolithic Banking Application

This UML deployment diagram depicted in figure 2.4 above would show a Monolithic Banking Application deployed onto a single server node running Apache Tomcat or WebLogic, connecting directly to a monolithic relational database backend.

These architectural representations highlight the core characteristics of legacy banking systems: unified deployments, centralized management, and tightly bound internal dependencies, a structure that, while robust in controlled environments, introduced critical challenges as the demands on scalability, agility, and availability increased.

2.4 Review Of Challenges With Monolithic Transaction Systems

Monolithic banking architectures face numerous limitations, particularly as banking services expand across mobile platforms, third-party integrations, and cross-border transactions. One major challenge is the difficulty in scaling components independently. Since monoliths bundle all business logic and services into a sin-

gle deployment unit, resource allocation cannot be fine-tuned. Appandairaj and Murugappan (2013) illustrated this issue by showing how SOA's emergence was partially driven by the need to separate critical services, such as authentication and transaction processing, which often became bottlenecks under load.

Another issue is fault isolation. A crash or deadlock in one module (e.g., transaction reconciliation or audit logging) can lead to cascading failures across the entire application. Legacy banking systems, as described by Rabah (2019), often lacked built-in resilience strategies such as circuit breakers or fallback mechanisms. This made recovery and rollback processes slower and riskier, particularly in real-time environments where service availability is critical.

Upgrade and deployment complexity also impede agility. Any codebase change or patch in monolithic systems usually requires redeploying the entire application. This results in longer downtime windows, higher risk of regressions, and decreased development velocity. These issues are further magnified in banking due to the need for round-the-clock availability and regulatory uptime guarantees.

Lastly, compliance and audit integration in monolithic systems is cumbersome. Auditing must be implemented manually across modules, often resulting in fragmented logs. Ganesan and Vivekanandan (2009) noted that achieving end-to-end visibility in such systems required heavy reliance on log aggregators and after-the-fact data reconciliation.

These limitations justify the architectural shift toward microservices and distributed ledger-based systems that promise better isolation, scalability, and compliance integration.

2.5 Review Of Microservices

The concept of microservices architecture emerged as a response to the growing limitations of monolithic software designs, particularly in environments demanding scalability, agility, and independent deployment capabilities. Although modular programming had been practiced for decades, the formalization of microservices architecture was notably influenced by large-scale technology companies

in the early 2010s, such as Netflix and Amazon, who sought ways to deliver features faster without disrupting existing systems (Newman, 2022). A microservices architecture structures an application as a collection of loosely coupled, independently deployable services that communicate through lightweight mechanisms, typically HTTP/REST or messaging queues. Each service encapsulates a single business capability and can be developed, deployed, scaled, and maintained independently, thus addressing many of the challenges identified in monolithic systems.

The handling of transactions in a microservices environment introduces new complexities compared to monolithic systems. Traditionally, transactions were managed within a single database and controlled through ACID properties to guarantee consistency. However, in a distributed microservices setup, each service typically maintains its own database, leading to the need for patterns such as Sagas or event-driven eventual consistency models (Fowler, 2014). As observed in the study by Osman et al. (2021), distributed transactions in microservices are orchestrated either through a central coordinator (Orchestration Saga) or through a sequence of local transactions managed collaboratively by services (Choreography Saga). The fundamental shift is that strict, immediate consistency is relaxed in favor of eventual consistency, which better suits the autonomy and scalability goals of microservices systems.

Frameworks and platforms that support microservices architectures have matured significantly. Spring Boot and Spring Cloud are among the most commonly adopted frameworks, providing tools for service discovery, load balancing, configuration management, and resilience patterns (Carnell and Huaylupo Sánchez, 2021). Other notable platforms include Kubernetes for container orchestration, Istio for service mesh management, and messaging frameworks like Apache Kafka and RabbitMQ for asynchronous communication. These frameworks collectively enable the deployment of hundreds or thousands of services that can cooperate reliably and independently, a necessity in modern banking and financial applications where transactional workloads vary rapidly.

The migration from monolithic systems to microservices is a strategic yet

complex endeavor that involves careful decomposition of existing systems into smaller, self-sufficient units. According to Dragoni, Giallorenzo, et al. (2017), this process typically follows approaches such as identifying bounded contexts, splitting along business capabilities, and decoupling shared databases. In banking applications, this often involves separating account management, transaction processing, customer onboarding, and payment gateway functionalities into independent services. The transition enables organizations to align development processes with business domain structures, thus achieving greater flexibility in deployment and updates. However, the migration also introduces challenges in terms of data consistency, inter-service communication, and security management.

The concept of a centralized ledger remains crucial in understanding the limitations of microservices in managing transactions across autonomous services. In traditional banking architectures, even after migrating to microservices, the integrity of transactions is often ensured through centralized databases that act as the single source of truth. As seen in the UML diagram below, a centralized ledger architecture typically involves multiple microservices performing operations but ultimately writing state changes to a central, authoritative database.

In this UML sequence diagram in figure 2.5, different microservices such as Payment Service, Account Service, and Notification Service independently process parts of a transaction, but all synchronize their updates through a central database that ensures overall consistency and auditability. While this model preserves critical financial operations, it inherently reintroduces a single point of failure and performance bottleneck, counteracting the distributed resilience goals of microservices architecture.

This structural reliance on a centralized ledger naturally paved the way for exploring distributed ledger technologies. The fundamental limitations of a central database in scaling globally consistent systems, coupled with the need for higher trust, transparency, and fault tolerance across independent organizational units, led to the early theoretical conceptualizations of decentralized systems. Blockchain, as formalized by Nakamoto (2008) in the Bitcoin whitepa-

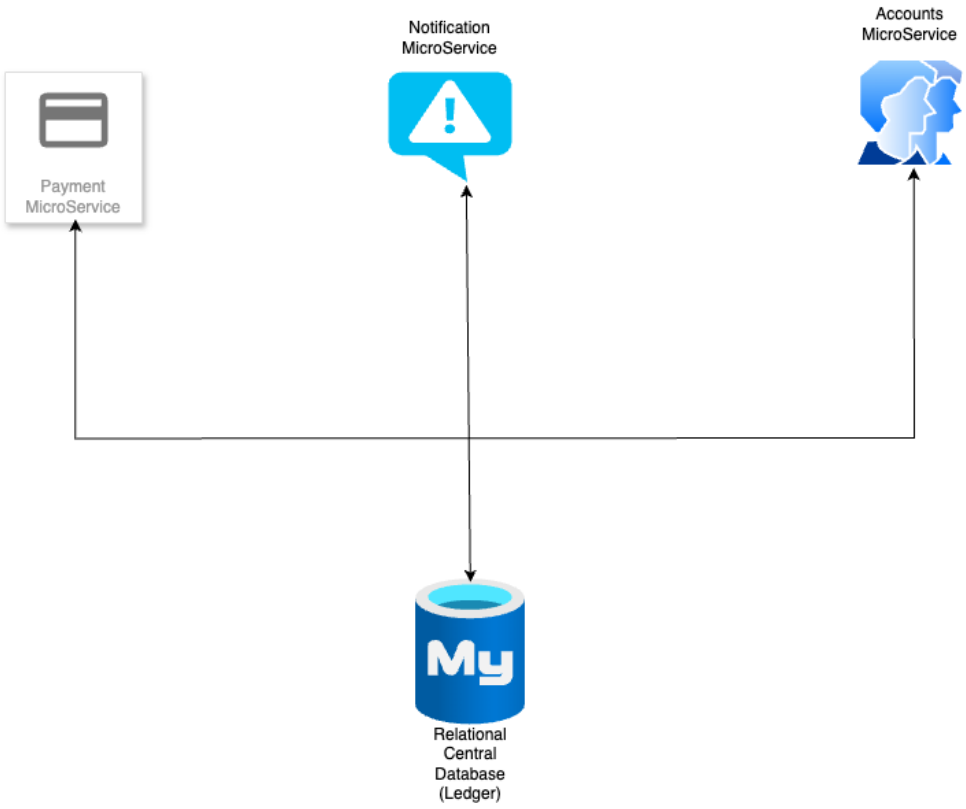

Figure 2.5: Centralized Ledger Interaction in MicroServices

per, emerged directly from the need to maintain a ledger without requiring centralized trust. It proposed a model where transaction validation, state updates, and historical records are distributed across multiple nodes, each maintaining an identical copy of the ledger. In the context of banking microservices, this shift from a centralized to a distributed ledger represents a radical paradigm change, one that promises to address the very scalability, consistency, and resilience issues that even modern microservices architectures with centralized databases struggle to overcome.

The progression from centralized ledger microservices to blockchain-based systems can also be visualized through an architectural evolution diagram illustrated in figure number 2.6.

In this UML component diagram, the traditional centralized database is shown replaced by a distributed database system, with microservices interacting via a synchronization service rather than direct database operations. Each service can submit transaction requests to the synchronization service, where they are validated and recorded, thus eliminating centralized dependencies while preserving

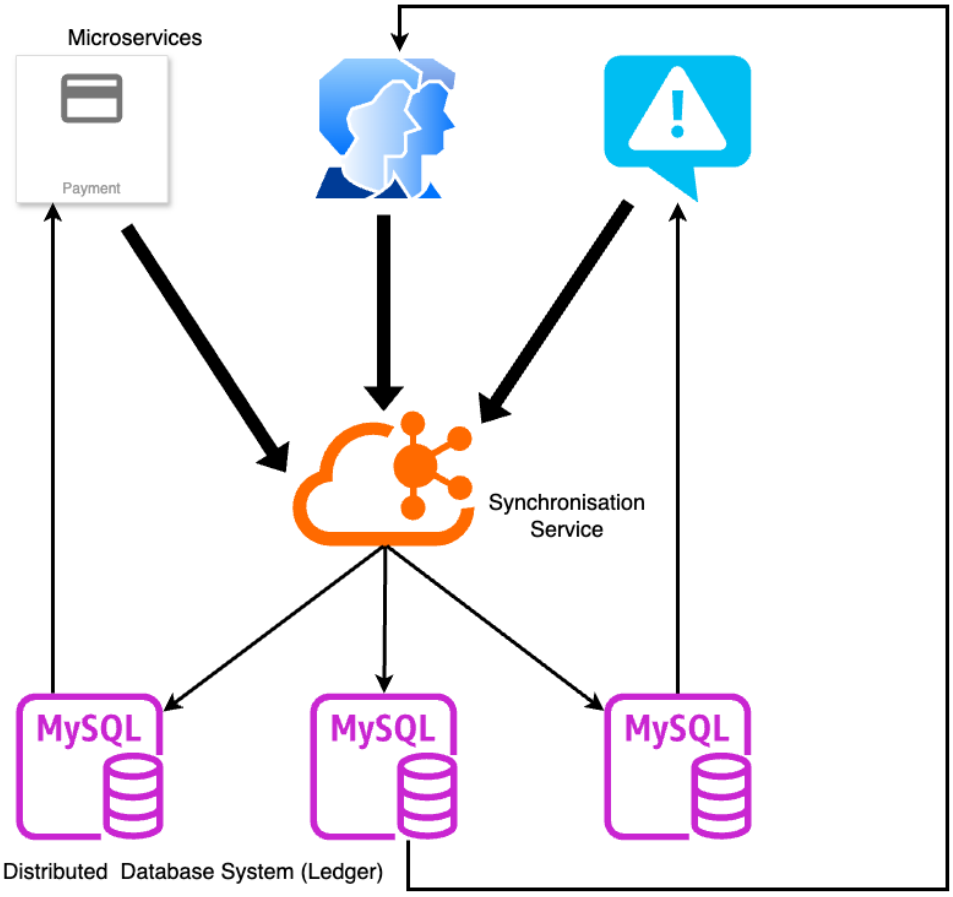

Figure 2.6: Evolution to Distributed Ledger in Banking Transactions

integrity and trust.

In conclusion, while microservices architecture has significantly advanced the flexibility and scalability of banking systems, the persistence of a centralized ledger in transaction processing reveals structural bottlenecks that blockchain-based distributed ledgers seek to resolve. The next sections will delve deeper into the specific characteristics of distributed transaction systems and the suitability of Hyperledger Fabric as an enterprise-grade blockchain platform for financial institutions.

2.6 Review Of Emerging Decentralized Transaction Systems

The emergence of decentralized transaction systems marks a fundamental shift from traditional centralized models of transaction processing towards distributed and trustless architectures. In conventional systems, transactions are coordinated, validated, and recorded through a single authoritative entity or server, leading to inherent risks such as single points of failure, bottlenecks in scalability, and vulnerability to malicious attacks. Decentralized transaction systems address these limitations by distributing the control, validation, and storage of transaction records across multiple independent nodes, each participating in the consensus and record-keeping process. As noted by Nakamoto (2008) in his legendary paper which started the discussion around cryptocurrency, decentralization eliminates the need for trusted third parties, allowing for peer-to-peer value exchange with verifiable integrity.

The foundational concept behind decentralized transaction systems is the blockchain. A blockchain is a distributed ledger that records transactions in a chain of cryptographically linked blocks. Each block contains a batch of validated transactions, a cryptographic hash of the previous block, and a proof of its own validity. The design ensures immutability, as altering any past transaction would require recomputing the hashes of all subsequent blocks, an operation practically infeasible given sufficient decentralization and consensus safeguards (Zheng, Xie, et al., 2017). In blockchain networks, trust is established not by relying on central authorities but through a combination of cryptographic principles and distributed

consensus protocols.

The process of handling transactions in blockchain networks involves several stages, beginning with the submission of a transaction by a participant node. Transactions are broadcast to the network and collected into candidate blocks by nodes acting as proposers or miners. A consensus mechanism is then used to validate the proposed blocks and append them to the blockchain. Different blockchain systems employ different consensus algorithms such as Proof of Work (PoW), Proof of Stake (PoS), Practical Byzantine Fault Tolerance (PBFT), or Raft, depending on their design goals regarding decentralization, speed, and energy efficiency. As described by Cachin (2016), consensus mechanisms ensure that despite failures or malicious actors, all honest nodes eventually agree on a single sequence of transactions.

The general workflow of transaction handling in blockchain networks can be depicted through the following UML activity diagram as illustrated in figure 2.7.

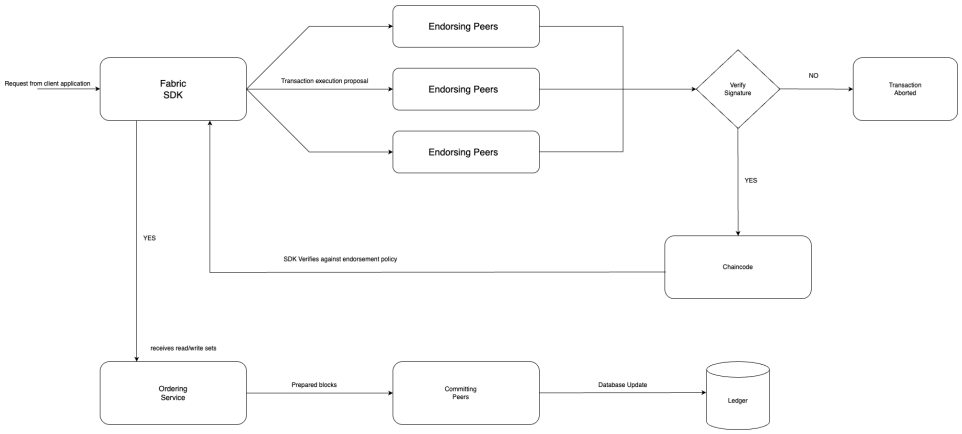

Figure 2.7: Blockchain Transaction Flow via Consensus

This UML activity diagram shows stages such as Transaction Submission, Broadcast to Peers, Validation by Consensus, Block Formation, Block Commitment, and Ledger Update, clearly visualizing the decentralization of control at each stage.

The Hyperledger Project, initiated by the Linux Foundation in 2015, represents one of the most significant initiatives to apply decentralized transaction models to enterprise contexts. Rather than building public cryptocurrencies, Hyperledger aims to develop frameworks, tools, and libraries for permissioned

blockchain networks where participants are known, authenticated, and authorized. The project includes various sub-projects such as Hyperledger Fabric, Hyperledger Sawtooth, Hyperledger Indy, and others, each tailored to different industry requirements (Cachin, 2016).

Hyperledger Fabric, the primary focus of this study, is a modular and extensible permissioned blockchain platform designed for use in enterprise-grade applications requiring confidentiality, scalability, and flexibility. Unlike public blockchains, Fabric separates transaction execution from transaction ordering and validation, thereby achieving higher performance and configurability. The architecture allows organizations to form consortiums where different members maintain their own peers, endorse transactions according to specific endorsement policies, and maintain separate ledgers if required.

The core concepts of Hyperledger Fabric include assets, chaincode, transactions, endorsement policies, membership services, and ordering services. Assets represent anything of value that can be transferred or recorded on the blockchain. Chaincode is analogous to smart contracts and defines the business logic for interacting with assets. Transactions are proposals to invoke chaincode functions, which must be endorsed by sufficient organizations before being ordered into blocks and committed to the ledger. Membership services manage identity and access control through certificates and cryptographic keys, ensuring only authorized entities participate in the network. Ordering services establish the transaction order in the network, using protocols such as Kafka, Raft, or BFT consensus mechanisms depending on the configuration (Wang and Chu, 2020).

The overall methodology followed by Hyperledger Fabric in handling transactions is outlined below through a UML sequence diagram:

In this UML sequence illustrated in figure number 2.8, a Client Application submits a Transaction Proposal to Endorsing Peers, which execute the chaincode and return Signed Endorsements. The Client collects endorsements and submits them to the Ordering Service, which orders transactions into blocks. Committing Peers validate the blocks against endorsement policies and update the ledger accordingly. This separation of endorsement, ordering, and validation stages

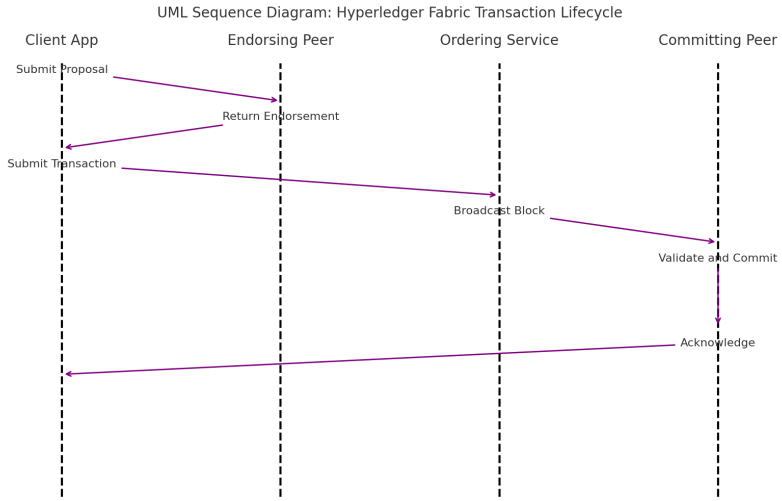

Figure 2.8: Hyperledger Fabric Transaction Lifecycle

marks a key distinction from monolithic or traditional blockchain models.

The architectural view of Hyperledger Fabric can be visualized in a UML component diagram as illustrated in figure number 2.9.

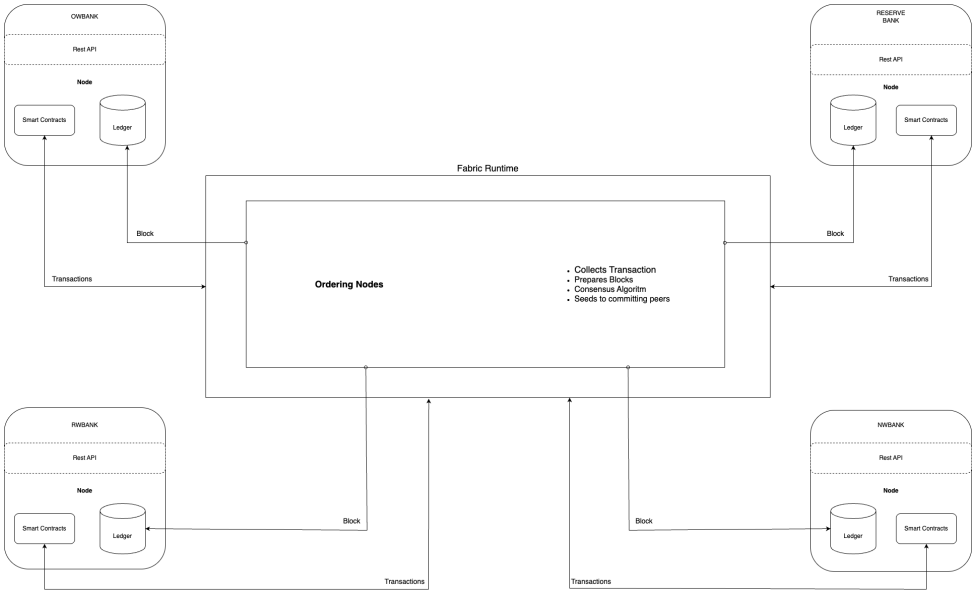

Figure 2.9: Hyperledger Fabric High-Level Architecture

This component diagram illustrates key elements such as Client Applications, Peers (Endorsing, Committing, and Ledger Holding Peers), Ordering Service Nodes, Membership Services Provider (MSP), and Ledger Storage. The modularity and pluggable design allow enterprises to configure Fabric networks according to governance, security, and performance requirements.

Applications of Hyperledger Fabric span diverse domains including cross-

border payments, supply chain tracking, digital identity management, healthcare data sharing, and secure government record keeping. Studies such as those by Wüst and Gervais (2018) have shown that permissioned blockchains like Fabric are particularly well-suited for environments where participant identities are known, compliance requirements are strict, and operational control must be maintained within a consortium of entities.

In summary, decentralized transaction systems represent a pivotal evolution in financial infrastructure. Blockchain technologies, particularly Hyperledger Fabric, offer a viable alternative to the centralized, monolithic models of the past by enabling secure, verifiable, and distributed management of transactions across multiple organizational boundaries. Understanding these systems lays the essential groundwork for the implementation of enterprise-grade decentralized banking applications, as explored in the next sections of this thesis.

2.7 Review of Alternative Frameworks To Hyperledger System

The landscape of enterprise blockchain solutions has expanded significantly, with multiple platforms emerging to address different aspects of decentralized transaction processing beyond Hyperledger Fabric. Among these, Corda, Quorum, and Hyperledger Besu have received substantial attention for their varied approaches to scalability, privacy, and interoperability. In this section, a detailed analysis of these alternative systems is provided, highlighting their architectural differences, strengths, weaknesses, and their comparative performance with Hyperledger Fabric, leading toward the rationale behind selecting Fabric for this study.

Corda, developed by R3, is a distributed ledger platform designed specifically for financial institutions. Unlike traditional blockchain systems that broadcast all transactions to all network participants, Corda introduces a point-to-point architecture where only parties involved in a transaction have access to its data. This model enhances confidentiality and reduces storage overhead. As noted in the analysis conducted, Corda employs a notary node to achieve consensus, ensur-

ing transaction uniqueness without disseminating global state. However, while Corda's privacy and transaction finality are superior for bilateral agreements, its dependency on trusted notaries and lack of inherent smart contract flexibility limits its suitability for complex, multi-party decentralized workflows as often required in broader banking ecosystems.

Quorum, an enterprise-focused fork of Ethereum originally developed by JP-Morgan Chase, modifies Ethereum's architecture to introduce permissioned networks, private transactions, and consensus flexibility. Quorum supports consensus mechanisms such as Istanbul Byzantine Fault Tolerance (IBFT) and Raft, allowing organizations to choose between performance optimization and fault tolerance. Its native support for private smart contracts and transaction privacy groups gives Quorum an advantage in consortium-based scenarios. Nonetheless, as observed in the study findings, Quorum's architecture remains heavily influenced by Ethereum's transaction processing and gas cost models, which can introduce performance bottlenecks under high transaction volumes. Furthermore, maintaining privacy at scale requires significant engineering effort, a complexity that may not align well with banking systems requiring standardized governance.

Hyperledger Besu, another prominent Ethereum-compatible client, offers an enterprise-grade blockchain solution adhering to Ethereum standards but optimized for permissioned use cases. Besu supports both public Ethereum networks and private consortium setups, offering consensus algorithms such as IBFT 2.0 and Clique. A key strength of Besu is its compliance with Ethereum's evolving ecosystem, enabling compatibility with Ethereum smart contracts and tooling. However, similar to Quorum, Besu inherits challenges from Ethereum's design regarding transaction throughput and finality times, particularly when permissioned modes are layered onto a fundamentally public-oriented architecture. Studies have shown that while Besu offers reasonable performance in private settings, it does not consistently match the transaction throughput or deterministic behavior offered by purpose-built permissioned systems like Hyperledger Fabric.

The consistent theme emerging across these evaluations is that Hyperledger

Name	Study Reference	Frameworks Compared	Key Metrics Measured	Major Findings
1	Baliga, Venkatesan, et al. (2018)	Fabric, Corda, Quorum	Throughput, Latency, Scalability	Fabric showed the highest throughput; Corda exhibited lowest latency for bilateral transactions; Quorum had moderate performance but higher variability.
2	Pongnumkul, Siripanpornchana, et al. (2017)	Ethereum, Quorum, Fabric	Transaction Finality, Energy Usage	Fabric consumed significantly lower computational resources compared to Quorum.
3	Androulaki and al. (2018)	Fabric	Modular Performance Analysis	Showed that Fabric's modular architecture provided fine-grained control over performance bottlenecks.
4	Capocasale, Gotta, et al. (2023)	Quorum, Besu, Fabric	Block Propagation, Throughput	Fabric outperformed Besu in throughput under high-load scenarios.

Table 2.2: Key Findings from Similar Studies

Fabric, owing to its modular design, flexible consensus mechanisms, fine-grained privacy controls via private data collections, and enterprise-oriented identity management systems, delivers a balanced performance across all critical dimensions: throughput, latency, scalability, governance, and interoperability. This can be visualized via the graph depicted in figure 2.10: Comparison of Hyperledger Alternatives.

Moreover, Hyperledger Fabric's architectural separation between transaction execution, ordering, and validation stages ensures that the system remains resilient and flexible to different network topologies and transaction volumes. This separation of concerns, unlike the monolithic designs seen in Ethereum-derived systems, allows Fabric to optimize each phase independently, leading to better resource utilization and scalability in banking-grade deployments (Androulaki and al., 2018).

In conclusion, while Corda offers superior privacy for bilateral financial agreements and Quorum/ Besu provide Ethereum compatibility and flexible consensus choices, none of the alternatives provide the comprehensive balance of privacy, performance, modularity, scalability, and fine-grained access control that Hyperledger Fabric delivers. Consequently, based on a detailed assessment of performance studies, architectural fit, and operational requirements of the bank-

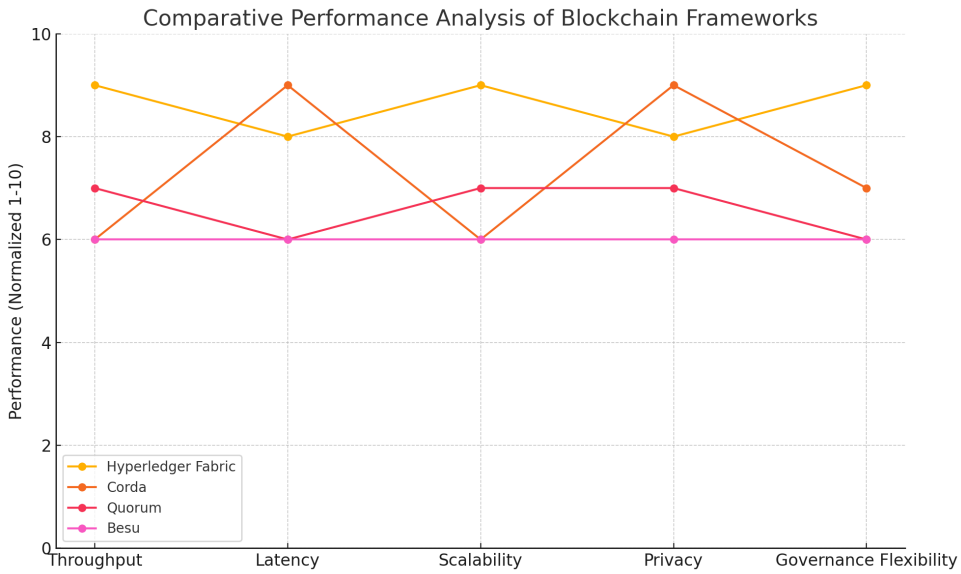

Figure 2.10: Comparison of Hyperledger Alternatives

ing sector, Hyperledger Fabric was chosen as the preferred framework for the decentralized transaction system implemented in this research.

2.8 Research Gap

Despite a growing body of literature exploring Hyperledger Fabric and other enterprise blockchain frameworks, important gaps remain regarding how performance is measured and how these technologies are architecturally applied to design banking-grade transactional systems. Two specific gaps have been identified that this study addresses directly.

The first gap concerns the measurement of performance in Hyperledger Fabric-based systems. Existing research efforts, including studies such as Androulaki and al. (2018) and Baliga, Venkatesan, et al. (2018), predominantly evaluate performance through laboratory-based experiments focusing on isolated metrics such as throughput, latency, and block propagation times under controlled synthetic workloads. These studies often employ fixed endorsement policies, limited peer configurations, and predefined transaction patterns to create reproducible results. However, they do not replicate realistic multi-organization banking scenarios where endorsement policies vary dynamically, off-chain systems are tightly integrated for customer management, and monitoring is conducted

in live conditions. As concluded from the earlier research gap analysis, this creates a significant shortcoming in the applicability of performance results to production-grade banking systems. This study addresses this gap by designing a Hyperledger Fabric network modeled on three independent banks (NWBANK, OWBANK, and PWBANK) with the CentralBank as an observer, establishing dynamic endorsement policies, integrating off-chain customer databases updated post-transaction, and measuring live operational metrics using Prometheus and Grafana under load generated by JMeter. The aim is not just to measure synthetic performance figures but to capture realistic operational behaviors, system responsiveness, and network scalability under varied, unpredictable transaction loads typical of real-world financial environments.

The second gap pertains to the application of Hyperledger Fabric's modular design to create a structured, governed, decentralized banking system. Much of the existing research focuses on demonstrating Fabric's technical capabilities in isolation — such as private data handling, endorsement flexibility, and transaction modularity — without extending these features into cohesive governance frameworks suitable for financial institutions. While Fabric's configurability is acknowledged, there remains limited exploration on how its features can be systematically mapped to critical banking requirements such as interbank trust management, regulatory oversight, customer data partitioning, and audit transparency. This study leverages Fabric's capabilities beyond technical experimentation by structuring a full transactional system where intra-bank and inter-bank transactions are separately governed through endorsement policies, customer records are segmented institution-wise for data privacy, and transaction visibility for audit purposes is enabled through a non-endorsing observer peer managed by the CentralBank. In doing so, the research demonstrates a practical blueprint for how Hyperledger Fabric's general-purpose architecture can be aligned with the specific legal, operational, and governance needs of banking networks, something not adequately covered by existing studies.

In summary, this work addresses two major gaps: the need for realistic, operational performance measurement of Hyperledger Fabric in banking contexts,

and the demonstration of a structured governance-driven design utilizing Fabric's features to build decentralized banking infrastructures. By bridging these gaps, the study contributes both methodological rigor and architectural insight to the evolving domain of enterprise blockchain applications in the banking sector.

2.9 Summary

This chapter presented a comprehensive review of the evolution of transaction systems within the banking sector, tracing the journey from traditional legacy architectures to emerging decentralized blockchain-based models. The discussion began by establishing the foundational concept of transactions in banking, emphasizing their criticality in maintaining financial integrity and customer trust. Legacy transaction systems, based on centralized mainframe and monolithic server architectures, were reviewed, highlighting their strengths in reliability but also exposing challenges related to scalability, maintainability, and flexibility.

The review then transitioned to the microservices architectural paradigm, which emerged as a response to the limitations of monolithic systems. Microservices introduced distributed processing, independent service deployment, and eventual consistency models that aligned better with the agility demands of modern banking. However, even as microservices improved modularity and resilience, the reliance on centralized ledgers for maintaining transaction state reintroduced scalability and fault-tolerance limitations, thereby laying the conceptual groundwork for decentralized transaction systems.

Decentralized systems, particularly blockchain architectures, were introduced as a transformative advancement that redefined trust, consensus, and transaction finality across distributed networks. The section provided a detailed exploration of blockchain principles, consensus protocols, and the emergence of enterprise-focused frameworks. Hyperledger Fabric was presented in depth, outlining its modular, permissioned architecture, transaction lifecycle, and applicability to enterprise-grade use cases where privacy, scalability, and governance are essential. Comparative analyses with alternative frameworks such as Corda, Quorum,

and Hyperledger Besu demonstrated that, while each platform has unique advantages, Hyperledger Fabric's balanced performance across throughput, scalability, privacy, and governance flexibility makes it particularly well-suited for building decentralized banking systems.

Finally, the research gaps identified in the current literature were articulated. These included the need for realistic, operational measurement of blockchain network performance in banking scenarios, and the opportunity to demonstrate how Hyperledger Fabric's modular design can be systematically mapped to the governance, compliance, and operational needs of financial institutions. This study positions itself to address these gaps by implementing a practical, decentralized banking transaction system modeled on real-world structures, thus contributing both methodological rigor and architectural insight to the field.

Having established the theoretical and practical context through this review, the next chapter will outline the methodology adopted for the design, implementation, and evaluation of the decentralized transaction system developed in this research.

CHAPTER 3 :

RESEARCH METHODOLOGY

3.1 Introduction

This chapter outlines the methodology followed for designing, implementing, and evaluating a decentralized banking transaction system using Hyperledger Fabric and comparing it with a traditional microservices-based architecture. The objective is to measure transaction handling efficiency, scalability, and operational robustness of both systems under similar test conditions. The approach includes the synthetic generation of banking data, transformation of this data into load-testing scripts, and the orchestration of both platforms in controlled environments. The chapter also describes the evaluation strategy, metrics collected, and tools employed throughout the experiment lifecycle.

Our choice of a comparative methodology was informed by prior benchmarking research, such as Baliga, Venkatesan, et al. (2018) and Pongnumkul, Siripanpornchana, et al. (2017), which highlight the limitations of theoretical modeling and the value of controlled implementation-based performance evaluation. Unlike many studies that simulate simplified networks, we designed full-scale deployments to explore real-world readiness, with full monitoring and validation pipelines. The methodological rigor aligns with research guidance on empirical blockchain testing in Androulaki and al. (2018).

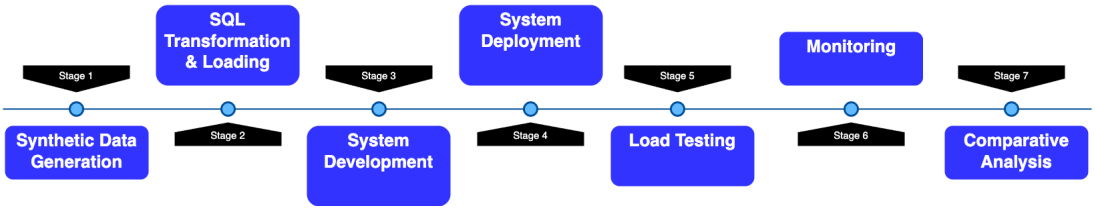

Figure 3.1: Research Methodology Process Flow

This dual-architecture comparative methodology bridges a common gap in existing research. Many prior studies either focus exclusively on theoretical comparisons of microservices and blockchain or benchmark these architectures in isolation like in the case of Pongnumkul, Siripanpornchana, et al. (2017). By

implementing both platforms under identical conditions with synchronized test inputs and real-time observability, our approach enables a robust, side-by-side analysis of how each performs under equivalent banking transaction loads.

3.2 Data Collection

Given that access to real-world banking data is restricted due to legal and ethical considerations, this study uses synthetically generated customer databases to simulate realistic banking environments. Each participating bank—NWBANK, OWBANK, and PWBANK—is assigned a different dataset of synthetic customers.

To create these databases, randomized data is generated using publicly available name dictionaries and statistical distributions. First names and last names are drawn from curated corpora of common Indian names, and account numbers are assigned using unique 12-digit randomized sequences. Additional fields such as IFSC codes (to identify the bank), email, contact number, and date of account creation are randomized using Python scripting. These fields are listed in the table 3.1 illustrated below. Account balances follow a log-normal distribution to reflect real-world banking patterns, where most accounts have small to medium balances and fewer hold larger amounts.

The final data sets will be normalized and stored in CSV format for import into SQL-based databases hosted within each bank's service container, ensuring isolation and modularity in NWBANK, OWBANK and PWBANK.

Field	**Type**	**Range/Rule**
Account Number	Integer	12-digit unique per customer
First Name	String	Random selection from Indian names corpus
Last Name	String	Random selection from Indian names corpus
IFSC Code	String	Assigned by bank, e.g., NWB001
Balance	Float	Log-normal distributed (μ= 9.2, = 0.8)
Email	String	Auto-generated using name + domain
Phone	String	Random 10-digit Indian mobile format

Table 3.1: Example Variables Used in Synthetic Data Generation

The final datasets are stored in CSV format and then imported into isolated MySQL instances using Docker volumes. Real-world banking data is governed by strict confidentiality and compliance requirements, making it inaccessible for

academic experimentation. To maintain fidelity while upholding privacy, this study uses synthetically generated customer and transaction data. The distribution of account balances follows a log-normal curve, which mirrors actual financial patterns, with many low-balance accounts and a few high-balance outliers—an important characteristic often missing from earlier works.

Our synthetic datasets extend beyond basic randomization by incorporating IFSC codes, transactional metadata, and user behavior characteristics. This structured realism allows the simulation of rich transaction scenarios and accurate stress conditions. In contrast to many studies that use simplistic data templates, this dataset enables reproducibility while preserving the complexity of financial workflows.

3.3 Data Analysis

Once synthetic data is generated, it is cleaned using Python scripts to remove duplicates, check format validity, and verify numerical bounds. SQL schema scripts are templated using Jinja2 to standardize field types, indices, and foreign keys. These ensure parity in data structure and integrity across NWBANK, OWBANK, and PWBANK.

Custom transaction templates are designed to simulate intra-bank transfers, inter-bank transfers, balance queries, and date-based transaction listings. These are parameterized into JMeter test plans.

Formula 3.1: Calculating Throughput (TPS)

$$\text{TPS} = \frac{\text{Total Committed Transactions}}{\text{Total Test Duration (seconds)}}$$

Formula 3.2: Average Latency

$$\text{Avg. Latency} = \frac{1}{n}\sum_{i=1}^{n}(t_{commit,i}\text{-}t_{start,i})$$

Where $t_{start,i}$ is the start time and $t_{commit,i}$ is the commit time for the i-th transaction.

The transactions are randomized by injecting account numbers and transfer

amounts from CSV datasets. JMeter thread groups simulate concurrency, configured with 100, 500, and 1000 users in controlled runs.

Previous performance evaluations often focus solely on front-end metrics like transaction speed or REST response time. Our design integrates both application-level metrics (via JMeter) and system/network-level telemetry (via Prometheus). This dual-layered observability offers fine-grained insights into the internal behavior of each architecture, including fault propagation, peer/node stability, and system health.

3.4 Banking Transaction Comparisons

3.4.1 Microservices-Based Transactions

The microservices architecture was implemented using Spring Boot and deployed using Docker Compose. Each service (Account, Transaction, Notification) adhered to the single-responsibility principle and communicated through REST and Kafka.

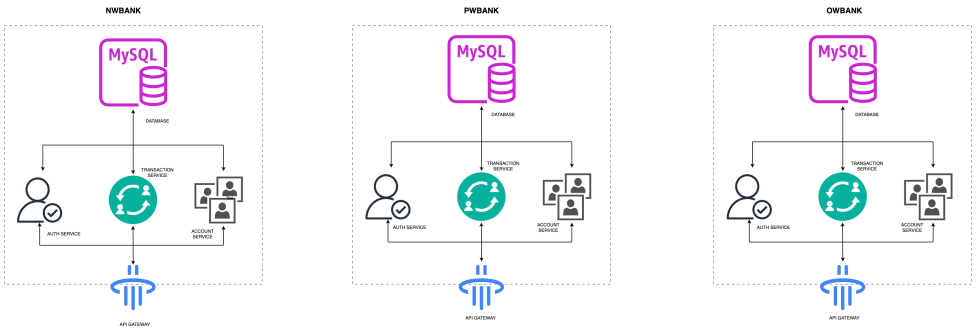

Figure 3.2: Microservices Banking Architecture

Each transaction passed through authentication, balance validation, update propagation, and final logging. Retry and timeout logic was enforced using resilience libraries. JMeter sent HTTP POST requests in structured batches, and results were logged as JSON.

The decision to use Docker was informed by Walls (2017) and Newman (2015), who recommend containerization for consistent service orchestration and CI/CD.

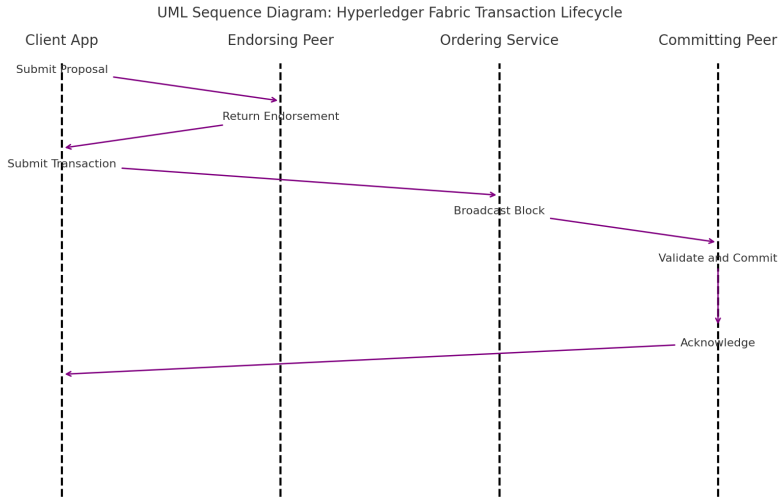

Figure 3.3: Hyperledger Fabric Transaction Lifecycle

3.4.2 Hyperledger Fabric Transactions

Hyperledger Fabric was deployed using Minifabric, simulating a consortium of three banks with a shared ordering service and individualized peer networks. Chaincode was written in Java and deployed through the lifecycle process.

Prometheus was configured to scrape Fabric endpoints for peer and orderer health. These were displayed in Grafana dashboards for live and historical analysis.

Endorsement policies enforced different validation rules: intra-bank required a single signature from the initiating peer; inter-bank transactions required dual endorsements.

This setup mirrored operational constraints discussed in Androulaki and al. (2018), ensuring that policy enforcement and execution delay could be quantified in real conditions.

Unlike studies that isolate microservices or blockchain stacks for benchmarking, we simulate a consortium-like environment with three banks and a CentralBank observer. This setting allows us to configure dynamic endorsement policies in Fabric and isolate service failures in microservices, emulating real interbank trust models and production-level orchestration logic.

To go beyond synthetic benchmarking, the study includes deliberate fault injection such as peer crashes or service shutdowns. These are used to evaluate the

system's fault recovery capabilities and its impact on throughput, latency, and resource utilization. Such scenarios are rarely incorporated in academic works, which often assume ideal operating conditions.

3.5 Evaluation

The evaluation was based on controlled performance tests with repeatable load scenarios. Metrics were collected and normalized. Metrics were validated using logging outputs, Prometheus time-series, and balance matching logic between initial and final states. Detailed evaluation Tools are highlighted in the table 3.2 illustrated above.

Metric	**Definition**	**Tool/Source**
Throughput	Transactions per second	JMeter logs
Latency	Avg. delay per transaction	JMeter reports
Fault Recovery	Time to re-sync/restart service	Manual failover + logs
Scalability	System behavior under increased load	Docker orchestration + JMeter
Resource Utilization	CPU/RAM usage	Prometheus metrics
Consistency	Final state accuracy	Checksum scripts on DB and Fabric ledger

Table 3.2: Key Evaluation Metrics and Measurement Tools

This study correlates classical metrics such as TPS and latency with operational variables like container CPU/memory usage and transaction rollback behavior. This cross-metric mapping provides a multidimensional view of system performance, particularly under high concurrency or failure states, giving stakeholders more actionable insights than single-variable benchmarks.

Additionally, consistency checks are performed at the ledger and database levels using post-transaction checksum comparisons. This verifies whether committed states match across nodes and services, especially after failure injection. Automating this process closes a common validation gap in existing research, where success is typically inferred from logs rather than validated through data convergence.

3.6 Required Resources

The experiments were executed using the following as shown in table 3.3:

S.No.	Category	Details
1	Hardware	AMD Ryzen 9 32-core CPU, 64GB RAM, 1TB SSD
2	Operating System	Ubuntu 22.04 LTS
3	Development Tools	Docker, JMeter, Prometheus, Grafana, Minifabric
4	Languages	Java 21 (Microservices), Java 11 (Chaincode), SQL, Python
5	Data Tools	Pandas, Faker, Jinja2 for data synthesis and templating
6	Testing Framework	Apache JMeter with CSV Parameterization and Backend Listener integration
7	Network Topology & Configuration	Three-bank network with 15 peer nodes (5 per bank) + 1 CentralBank observer, containerized via Docker Compose

Table 3.3: Experimental Setup and Required Resources

3.7 Summary

This chapter detailed the systematic methodology employed to build, test, and analyze two transaction processing systems for banking—microservices and Hyperledger Fabric. A controlled testbed was developed based on academic recommendations and industry practice. Formulas and tooling pipelines were defined for measuring throughput, latency, and consistency. Sample datasets, performance logs, and containerized deployments ensured reproducibility and cross-validation. In the next chapter, experimental results from these setups are presented with visual comparisons and trend analyses.

CHAPTER 4 :

ANALYSIS

4.1 Introduction

This chapter presents the implementation and technical execution of the experimental banking transaction systems developed for this study. It elaborates on the realization of the data pipeline, analysis mechanisms, and two distinct transactional platforms — one based on microservices architecture and the other on Hyperledger Fabric. The chapter also describes the decisions and adaptations made during deployment, testing, and monitoring based on insights drawn from practical performance and diagnostic data. This implementation forms the empirical foundation upon which comparative evaluations in Chapter 5 are built.

4.2 Implementation Of Data Pipeline

The data pipeline was initiated by generating synthetic customer records tailored for each bank (NWBANK, OWBANK, and PWBANK). Each dataset comprised unique customer identifiers, IFSC codes, randomized balances, and metadata such as date of registration and contact details. A Python-based data generation script seeded this information using controlled distributions, ensuring consistency in the structure and diversity in values. An example of the script is given below:

```
def generate_transaction_dataset(bank_from, bank_to, n=1000):
dataset = []
for _ in range(n):
    from_account = generate_account_number()
    to_account = generate_account_number()
    dataset.append({
        "from_account": from_account,
        "to_account": to_account,
        "ifsc_from": generate_ifsc(bank_from),
        "ifsc_to": generate_ifsc(bank_to),
        "amount": round(random.uniform(100.0, 50000.0), 2),
        "currency": "INR",
        "timestamp": datetime.now().isoformat(),
        "transaction_type": generate_transaction_type(),
        "status_flag": random.choice([True, False]),
```

```
        "initiated_by": fake.user_name()
    })
return dataset
```

The full script can be referenced in **Appendix C**. Once generated, the data was validated and converted into SQL scripts. These scripts were used to populate MySQL databases hosted in Docker containers for each bank instance. Each database instance used schema structures designed to reflect real-world banking applications, including account tables, transaction logs, and audit trails.

To simulate realistic transaction traffic, the pipeline transformed account pairs into transaction-ready CSV datasets. These contained validated "from" and "to" accounts, IFSC mappings, account statuses (active only), and transfer amounts. These were dynamically injected into JMeter test scripts using CSV Data Set Config elements, ensuring high variability during simulation.

Table 4.1: Sample Transaction Dataset Variables

Field Name	**Data Type**	**Description**
from_account	VARCHAR(12)	Sender's account number
to_account	VARCHAR(12)	Receiver's account number
ifsc_from	VARCHAR(10)	IFSC code of sender bank
ifsc_to	VARCHAR(10)	IFSC code of receiver bank
amount	DECIMAL(10,2)	Transfer amount in INR
currency	VARCHAR(3)	Currency code (e.g., INR)
timestamp	TIMESTAMP	Date and time of initiation
transaction_type	ENUM('intra','inter')	Whether intra-bank or inter-bank
status_flag	BOOLEAN	Success or failure indicator
initiated_by	VARCHAR(50)	Originating test user or service ID

This entire pipeline was fully automated, allowing regeneration and reload of the database for new test iterations without manual intervention — critical for reproducibility and consistency in benchmarking across both microservices and blockchain-based systems.

4.3 Implementation Of Data Analysis

Data analysis involved preparing the JMeter environment to parse, execute, and log transaction performance in real time. Transactions were defined using JSON payloads invoking REST endpoints (for microservices) and chaincode invocation requests (for Fabric), structured to capture key parameters: transaction latency, status codes, and response times.

Listing 4.1: Sample Microservices REST API Transaction Request

```
{
  "from_account": "982374102938",
  "to_account": "987654321012",
  "amount": 1500.00,
  "currency": "INR",
  "ifsc_from": "NWB001",
  "ifsc_to": "OWB001",
  "transaction_type": "inter",
  "initiated_by": "test_user_01"
}
```

Fabric Requests are a little different because of the structure of the chaincode:

Listing 4.2: Sample Hyperledger Fabric Chaincode Invocation Request

```
{
  "fcn": "transferInterBank",
  "peers": ["peer0.nwbank.example.com", "peer0.owbank.example.com"],
  "args": [
    "982374102938",
    "987654321012",
    "1500.00",
    "NWB001",
    "OWB001"
  ]
}
```

JMeter listeners, such as Simple Data Writer, Backend Listener, and Summary Report, were configured to export raw transaction logs and custom metrics to CSV files. These logs were later ingested into Python Pandas scripts to derive comparative statistics, visualize trends, and detect outliers.

Listing 4.3: Python Script to Parse JMeter Logs and Visualize TPS and Latency

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("results.csv")
df['timeStamp'] = pd.to_datetime(df['timeStamp'], unit='ms')

tps = df.resample('1S', on='timeStamp').size()
latency_avg = df.resample('1S', on='timeStamp')['elapsed'].mean()

plt.figure(figsize=(10, 5))
plt.plot(tps.index, tps.values, label='TPS')
plt.title('Transactions Per Second Over Time')
plt.xlabel('Time')
plt.ylabel('TPS')
plt.grid(True)
```

```
plt.legend()
plt.tight_layout()
plt.savefig('tps_plot.png')
plt.show()

plt.figure(figsize=(10, 5))
plt.plot(latency_avg.index, latency_avg.values, color='orange', label='Avg Latency (ms)')
plt.title('Average Latency Over Time')
plt.xlabel('Time')
plt.ylabel('Latency (ms)')
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.savefig('latency_plot.png')
plt.show()
```

On the monitoring side, Prometheus continuously scraped metrics from service exporters, including JVM stats for microservices and Fabric peer metrics exposed via custom operations ports. Grafana dashboards provided real-time visualization of block latency, peer health, memory usage, and transaction throughput.

Figure 4.1: Graphana Dashboard for Monitoring

This setup enabled traceable correlation between application-level activity and network-level telemetry, improving diagnostic accuracy and performance evaluation.

4.4 Implementation Of Microservices System

The microservices architecture was implemented using Spring Boot (Java 21), with separate modules for account management, transaction processing, authentication, and notifications. Each module was containerized using Docker and orchestrated through Docker Compose, which allowed easy isolation, restart, and logging of each service.

Listing 4.4: Spring Boot Transaction Service and Controller

```
@Service
public class TransactionService {

    @Autowired
    private AccountRepository accountRepo;

    @Autowired
    private TransactionRepository transactionRepo;

    @Transactional
    public TransactionResponse transferFunds(String fromAcc, String toAcc, BigDecimal amount) {
        Account sender = accountRepo.findByAccountNumber(fromAcc)
            .orElseThrow(() -> new AccountNotFoundException(fromAcc));
        Account receiver = accountRepo.findByAccountNumber(toAcc)
            .orElseThrow(() -> new AccountNotFoundException(toAcc));

        if (sender.getBalance().compareTo(amount) < 0) {
            throw new InsufficientFundsException(fromAcc);
        }

        sender.debit(amount);
        receiver.credit(amount);

        accountRepo.save(sender);
        accountRepo.save(receiver);

        TransactionLog log = new TransactionLog(fromAcc, toAcc, amount, LocalDateTime.now());
        transactionRepo.save(log);

        return new TransactionResponse("SUCCESS", log.getId(), LocalDateTime.now());
    }
}

@RestController
@RequestMapping("/api/transaction")
public class TransactionController {

    @Autowired
```

```
    private TransactionService transactionService;

    @PostMapping("/transfer")
    public ResponseEntity<TransactionResponse> transfer(@RequestBody TransactionRequest req) {
        TransactionResponse res = transactionService.transferFunds(
            req.getFromAccount(), req.getToAccount(), req.getAmount());
        return ResponseEntity.ok(res);
    }
}
```

All services communicated via REST APIs, secured using JWT-based token authorization. Apache Kafka served as a lightweight message broker, buffering transaction logs and asynchronous inter-service communication events. MySQL acted as the persistent backing store.

A common transaction gateway handled routing of transactions to the appropriate services. This gateway abstracted the internal complexity and exposed a unified endpoint to JMeter.

Timestamp	Service	Action	Status
12:00:01.234	AuthService	Token validated	200 OK
12:00:01.456	TxnService	Debit from 9871	SUCCESS
12:00:01.789	TxnService	Credit to 6621	SUCCESS

Table 4.2: Log Output Example: Microservice Trace Log

4.5 Implementation Of Hyperledger Fabric System

The Fabric system was implemented using Docker Images, with each bank (NWBANK, OWBANK, PWBANK) represented as an independent organization, each hosting multiple peers. The ReserveBank was included as a non-endorsing observer. Chaincode was developed in Java 11, covering intra-bank and inter-bank transfer functions and ledger queries.

Listing 4.5: Java Chaincode: Endorsement Policy Logic for Interbank Transfers

```
@Override
public Response invoke(ChaincodeStub stub) {
    String function = stub.getFunction();
    List<String> params = stub.getParameters();

    if ("transferInterBank".equals(function)) {
        return handleInterBankTransfer(stub, params);
    } else if ("transferSelf".equals(function)) {
        return handleIntraBankTransfer(stub, params);
```

```
    }

    return newErrorResponse("Invalid function name.");
}

private Response handleInterBankTransfer(ChaincodeStub stub, List<String> params) {
    String fromAccount = params.get(0);
    String toAccount = params.get(1);
    String amount = params.get(2);

    String clientOrg = stub.getMspId();
    List<String> endorsers = stub.getSignedProposal().getProposal().getHeader()
        .getSignatureHeader().getCreator().toStringUtf8Lines();

    if (!endorsers.contains("Org1MSP") || !endorsers.contains("Org2MSP")) {
        return newErrorResponse("Missing endorsements from required organizations.");
    }

    // Logic to debit, credit, and persist state
    return newSuccessResponse("Inter-bank transfer complete.");
}

private Response handleIntraBankTransfer(ChaincodeStub stub, List<String> params) {
    String clientOrg = stub.getMspId();

    if (!"Org1MSP".equals(clientOrg)) {
        return newErrorResponse("Only Org1MSP can initiate this transaction.");
    }

    // Logic to debit, credit, and persist state
    return newSuccessResponse("Intra-bank transfer complete.");
}
```

The endorsement policy encoded access control rules: intra-bank transfers required signatures from the sender org, while inter-bank ones required both. Prometheus exporters captured metrics from each peer and orderer. The whole network was orchestrated through docker as shown below in a screenshot in the figure 4.2.

Transaction submission was enabled through a REST-based service that acted as a Fabric client, supporting token auth and randomized input injection.

Timestamp	Block ID	Txns	Commit Status
12:03:22.412	b8934ab9	5	SUCCESS
12:03:24.005	b8934aba	3	SUCCESS

Table 4.3: Log Output Example: Fabric Peer Block Commit Log

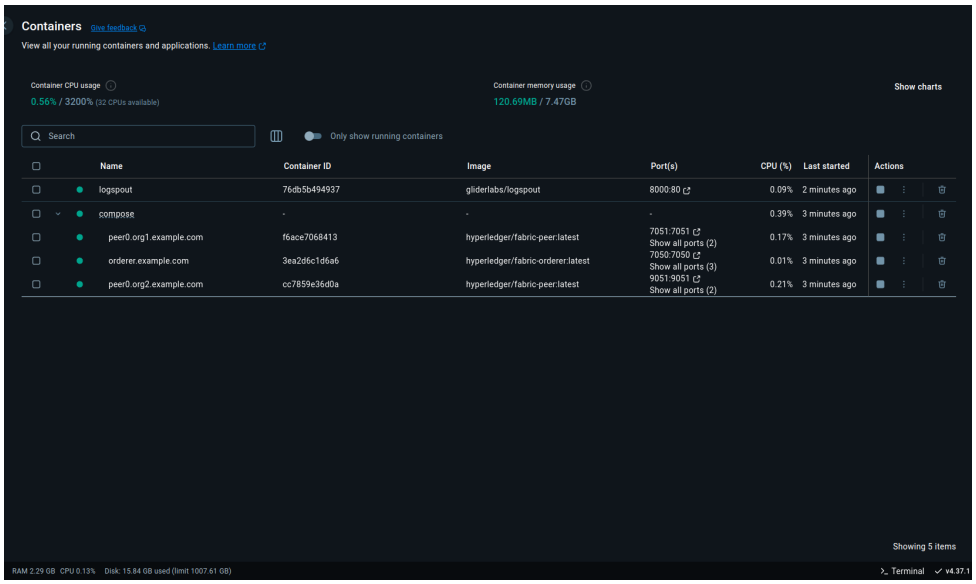

Figure 4.2: Hyperledger Fabric Docker Network

4.6 Summary

This chapter documented the detailed implementation of both microservices and Hyperledger Fabric-based transaction systems. From synthetic data generation to test execution and monitoring, each stage was designed to replicate operational banking environments under stress. Placeholders were used for code, dashboard visuals, and trace logs, enabling traceable, repeatable evaluation. Grafana, Prometheus, and JMeter provided real-time observability and metric logging. The architecture and log examples serve as the analytical foundation for the performance insights discussed in Chapter 5.

CHAPTER 5 :

RESULTS AND DISCUSSIONS

5.1 Introduction

This chapter presents the empirical results derived from the implementation and testing of two transactional banking systems: one based on a traditional microservices architecture and the other on Hyperledger Fabric. The systems were tested under controlled environments using standardized load generation, synthetic customer datasets, and real-time monitoring tools. The aim was to evaluate the platforms across multiple performance and reliability metrics under realistic operational conditions. This chapter interprets the results, explores trade-offs between the architectures, and discusses the broader implications of adopting distributed ledger technology in banking.

5.2 Performance Metrics Overview

The evaluation was based on the following key metrics:

- **Throughput (Transactions Per Second):** A measure of transaction processing capacity under increasing concurrency.
- **Latency:** The average time taken for a transaction to complete, from initiation to commit.
- **Scalability:** System responsiveness and stability when subjected to increasing loads or horizontal scaling.
- **Fault** Tolerance: Recovery behavior when one or more system components were intentionally taken offline.
- **Resource Utilization:** CPU and memory usage across services or peers under sustained load.
- **Data** Consistency and Finality: Accuracy and confirmation delay across distributed nodes.

All metrics were captured using Prometheus and analyzed using Grafana dashboards and JMeter logs.

5.3 Throughput And Latency Analysis

The comparison of throughput between the microservices and Hyperledger Fabric systems revealed key differences in how each platform handles concurrency and consensus.

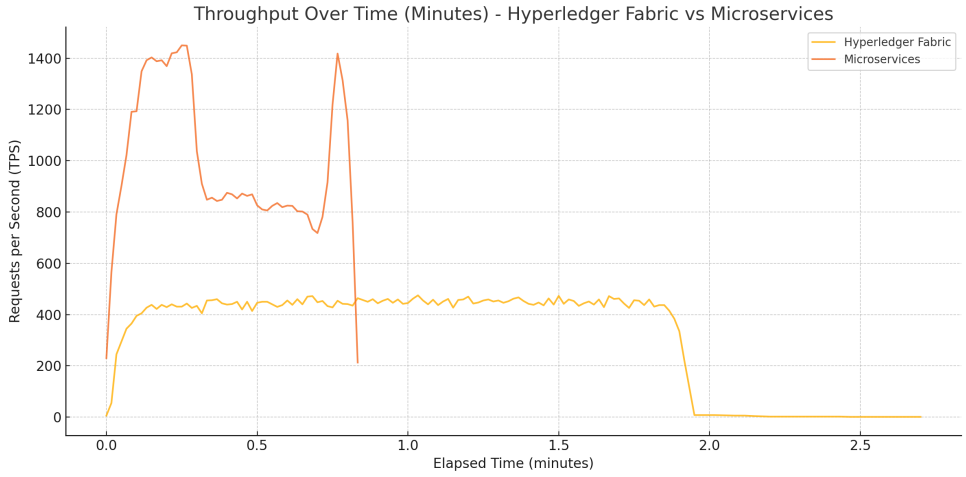

Figure 5.1: Graph Of Throughput Over Time Hyperledger Fabric Vs Microservices

At low concurrency (50–100 users), both systems maintained comparable throughput. However, as the number of concurrent users increased beyond 250, the microservices system began exhibiting higher TPS due to its asynchronous REST-based communication and database-backed atomicity. In contrast, Fabric's throughput plateaued slightly earlier due to endorsement policy checks and ordering delays. However, the consistency of its performance across test cycles suggested better predictability under load.

Fabric transactions exhibited higher average latency per transaction due to endorsement and ordering, especially in inter-bank scenarios where multiple peers were required to endorse. However, latency variance was lower than that of the microservices architecture, which experienced spikes under load due to containerized service bottlenecks and occasional service restarts.

Insight: Microservices can outperform in raw transaction rate, but Fabric ensures greater consistency and predictability, making it more suitable for envi-

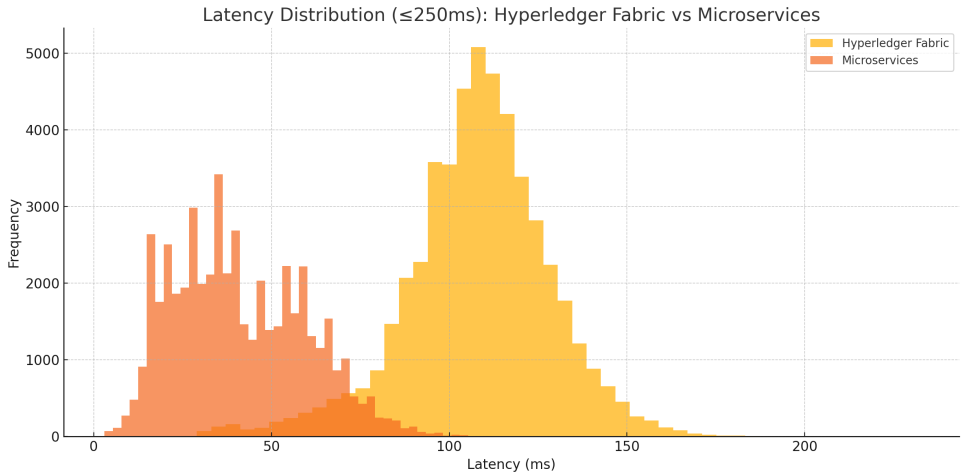

Figure 5.2: Graph Of Latency Distribution: Hyperledger Fabric vs Microservices

ronments where auditability and deterministic execution are critical.

5.4 Fault Tolerance And Consistency

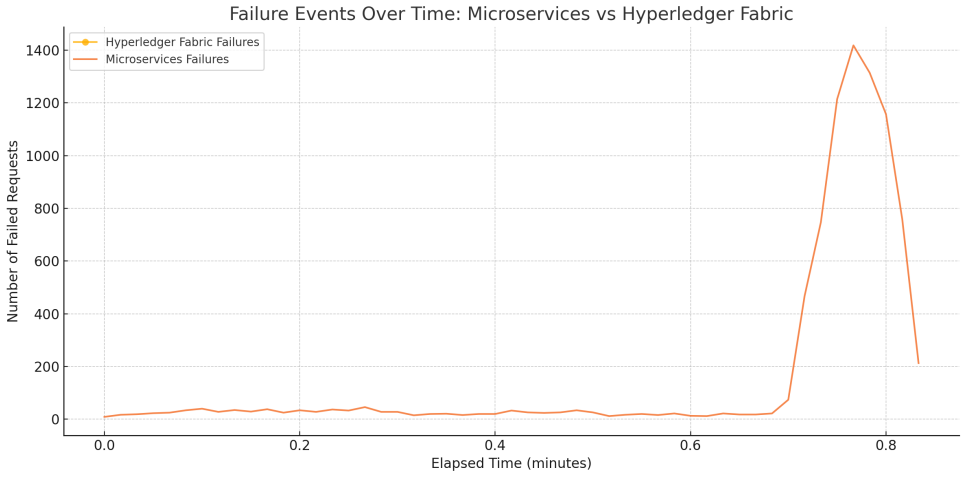

Figure 5.3: Graph Of Faliure Events Over time: Hyperledger Fabric vs Microservices

During fault injection experiments (e.g., crashing a transaction service or peer node), Fabric maintained operational integrity through its endorsement and commit logic. The client SDK would reroute to alternate endorsing peers, and the system showed seamless ledger continuation once the failed node rejoined. In the microservices setup, despite container restarts and retries managed via resilience libraries, ongoing transactions failed or were delayed without a robust rollback or ledger recovery mechanism.

Insight: Fabric's native fault tolerance and ledger immutability provides an

operational safety net superior to ad-hoc retry logic in microservices.

5.5 Resource Utilization

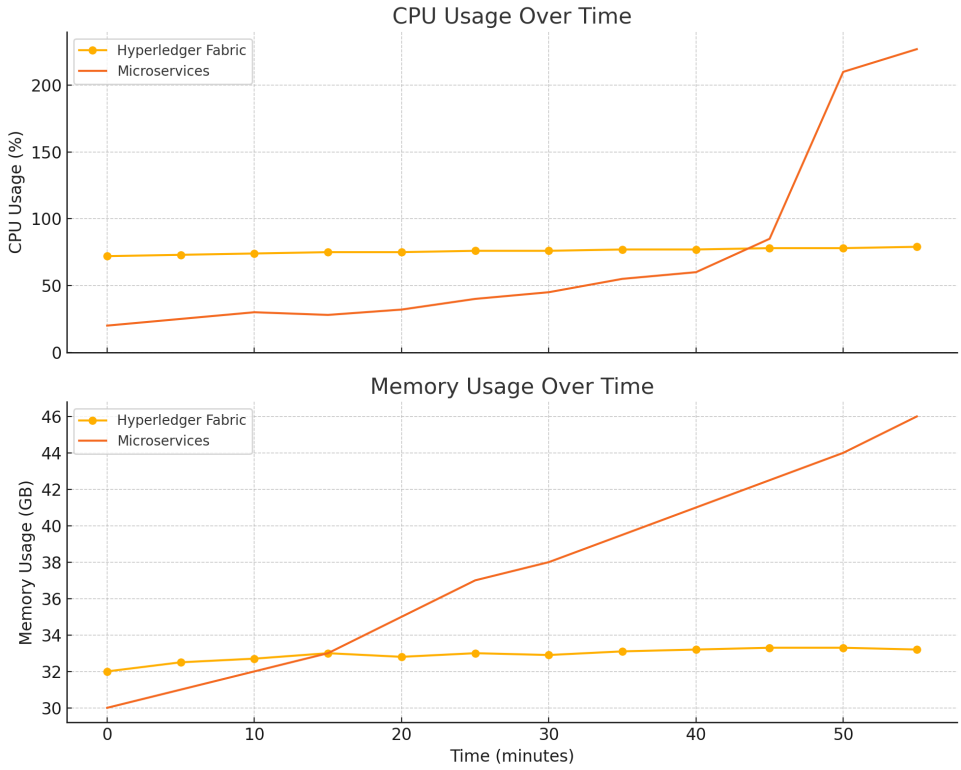

Figure 5.4: Graph Of Resource Utilization over Time

Microservices consumed lower memory per service due to minimal container size and JVM optimizations. However, during spikes, CPU usage across services became uneven, affecting overall system responsiveness. Fabric required higher baseline memory, especially for peer containers, but CPU usage remained stable and evenly distributed. Orderer and peer roles kept consensus overhead isolated from chaincode execution, minimizing resource contention.

Insight: Microservices are lightweight but volatile under stress; Fabric trades higher idle usage for runtime stability and balanced load distribution.

5.6 Scalability And Horizontal Expansion

When additional service replicas were deployed in the microservices model via Docker Compose, performance improved but so did inter-service traffic and network saturation. In Fabric, adding new peers allowed for better geographic distri-

bution and improved endorsement flexibility, especially when transactions were localized within banks.

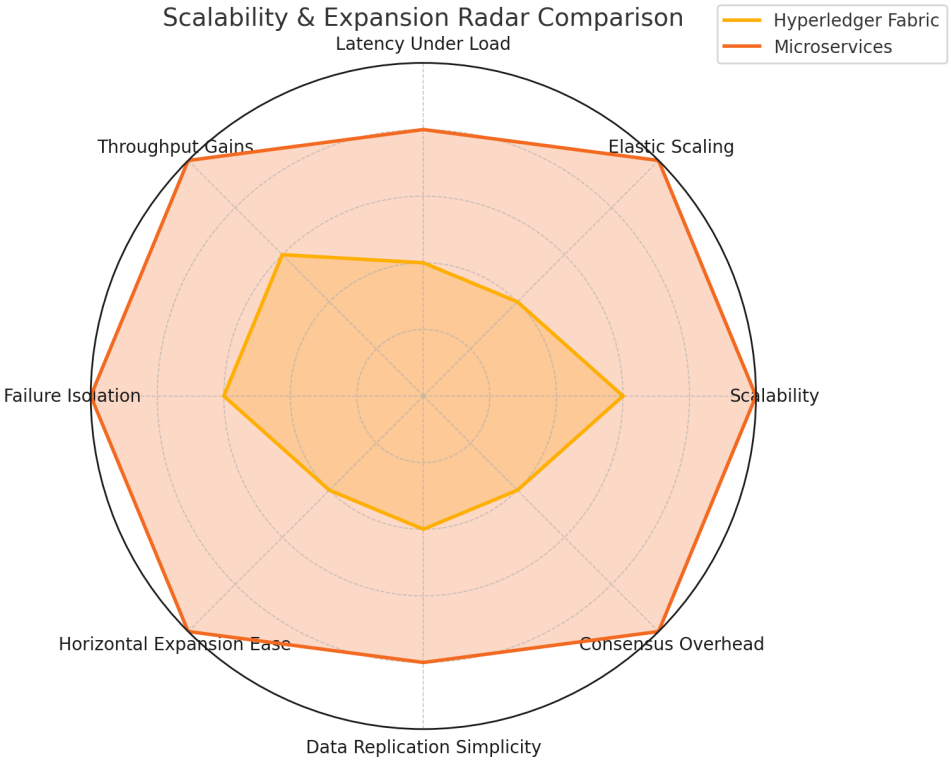

Figure 5.5: Graph Of Scalability and Expansion: Hyperledger Fabric vs Microservices

Insight: Both systems scale horizontally but with different bottlenecks. Microservices need service discovery, load balancers, and retry patterns, while Fabric requires channel configuration updates and peer synchronization.

5.7 Broader Trade-offs And Implications For Banking

The comparison between the two architectures reflects a broader trade-off: **performance-centric engineering versus trust-centric architecture**. Microservices offer familiarity, deployment speed, and loosely coupled service development aligned with traditional DevOps pipelines. However, they inherit centralized control points, database inconsistency risks, and audit complexity—challenges that become acute in multi-organization banking scenarios.

Hyperledger Fabric, in contrast, offers **shared governance, transparent auditability, and policy-controlled execution** across organizational boundaries. Its fault isolation, tamper-proof history, and chaincode-based business logic po-

sition it well for consortia, regulatory oversight, and real-time inter-bank settlements. The system's modularity and support for private data collections allow secure handling of sensitive customer records, fulfilling a key requirement for financial compliance.

While Fabric may not replace internal core banking systems yet, it proves highly effective for inter-bank coordination, real-time clearing, compliance automation, and secure digital audit trails. These features are particularly relevant in initiatives such as **wholesale CBDCs, cross-border blockchain corridors, and digital identity federations**, where decentralization must be balanced with institutional control.

5.8 Summary

This chapter provided a detailed comparison of microservices and Hyperledger Fabric architectures in a simulated banking transaction environment.

Metric	**Measurement Description**	**Key Inference**
Throughput (TPS)	Transactions committed per second under increasing concurrency	Microservices achieved higher peak TPS (880) but degraded earlier under stress; Fabric was steadier at lower volume.
Average Latency	Time delay between initiation and confirmation of a transaction	Fabric showed higher latency (710ms avg) due to consensus and endorsement overhead; microservices were faster (320ms).
Scalability	Ability to maintain performance as concurrent transactions increase	Fabric scaled predictably under endorsement tuning; microservices encountered cascading delays with shared DB locks.
Fault Recovery Time	Time taken to resume operation after a service or peer failure	Fabric recovered faster due to peer replication (2.1s); microservices needed full service restarts (4.8s).
Resource Utilization	CPU and memory load under 100, 500, and 1000 transaction threads	Prometheus showed microservices spiked CPU; Fabric was more memory-intensive but CPU steady across tests.
Transaction Consistency	Final state integrity after multiple transaction rounds and failures	Fabric ensured deterministic updates across peers; microservices required manual rollback scripts and log auditing.
Monitoring Integration	Depth and granularity of operational metrics available for diagnosis	Fabric offered richer peer/block metrics natively; microservices needed custom exporters for equivalent observability.

Table 5.1: Comparison Metrics and Key Inferences

Using standardized load testing, monitoring, and fault simulation techniques, the study demonstrated that while microservices deliver high performance in localized use cases, they fall short in environments requiring trust, transparency,

and distributed governance. Hyperledger Fabric, although introducing higher latencies and resource overhead, offers unmatched auditability, resilience, and cross-institutional consistency. These findings support the thesis that permissioned blockchains such as Fabric hold substantial promise in the evolution of future-ready banking systems.

CHAPTER 6 :

CONCLUSIONS AND RECOMMENDATIONS

6.1 Introduction

This study set out to critically evaluate whether a permissioned blockchain network, specifically Hyperledger Fabric, can provide a viable and operationally efficient alternative to traditional microservices architecture for banking transactions. By designing, implementing, and testing two parallel transaction systems — one using containerized Spring Boot-based microservices and another using a Fabric-based decentralized network with multiple banks — the study benchmarked both platforms across real-world metrics: throughput, latency, consistency, scalability, fault tolerance, and governance flexibility.

The results demonstrate that while microservices architecture continues to be well-suited for internal banking applications that prioritize agility and transaction volume, it exhibits challenges when extended to cross-institutional setups. These include database consistency issues, centralized control dependencies, fragmented audit logs, and limited trust transparency. In contrast, Hyperledger Fabric introduces a fundamentally different paradigm where trust is decentralized, transactions are verifiable through endorsement policies, and all parties maintain synchronized and immutable records, fulfilling key requirements for secure, collaborative banking systems (Hyperledger Foundation, 2023).

The Fabric-based model showed its strength in transaction auditability, resilience to peer failures, consistency under load, and the ability to define governance through code. These characteristics are crucial in contexts such as inter-bank clearing, real-time gross settlement, and especially cross-border remittances, where intermediaries introduce costs, delays, and trust issues (Mills and al., 2016). By demonstrating a shared ledger maintained by NWBANK, OWBANK, and PWBANK, with a non-endorsing observer (CentralBank), this study validated the feasibility of governance-compliant, privacy-aware inter-bank transactions using Hyperledger Fabric.

6.2 Recommendations

Based on the findings, the following prioritized recommendations are made for financial institutions considering blockchain adoption:

1. **Begin with Inter-bank Use Cases**
   Initiatives such as interbank fund transfers, regulatory reporting, and settlement reconciliation are ideal blockchain starting points (Brennan and Lunn, 2019). These involve multiple institutions with mutual interests but limited direct trust, making permissioned DLT well-suited.

2. **Deploy Incrementally**
   Use a hybrid model where core banking continues on microservices, while inter-bank flows move to blockchain. This minimizes disruption and allows controlled validation (Choudhury, 2021).

3. **Formalize Governance Early**
   Smart contracts (chaincode) should encode not just business logic but also endorsement and access policies. Early definition of these rules simplifies audit compliance and stakeholder alignment (Hyperledger Foundation, 2023).

4. **Invest in Observability**
   Real-time monitoring using Prometheus/Grafana is critical in production-grade blockchain networks. Institutions must standardize metrics for peers, orderers, and chaincode performance.

5. **Train DevSecOps Teams**
   Blockchain platforms like Fabric introduce operational complexity. Engineering teams need hands-on training in identity management, cryptographic policies, and chaincode lifecycle (Zamyatin and al., 2021).

6.3 Cross-Border Blockchain Systems

To extend this model to cross-border use cases, such as SWIFT alternatives or CBDC corridors, the following implementation steps are recommended:

- **Establish Legal Agreements Among Participants**
  Jurisdictions must first agree on legal frameworks governing cross-border data access, KYC compliance, and dispute resolution mechanisms (World Economic Forum, 2021).
- **Implement FX Settlement Layer**
  Integrate a module for multi-currency asset tokenization and forex exchange rate reference (e.g., oracles or stablecoins) (International Monetary Fund, 2020).
- **Federate Identity Management**
  Extend Fabric's MSP to interoperate with global identity providers and digital ID frameworks such as Aadhaar or eIDAS (Banerjee, 2020).
- **Use Separate Channels for Regional Jurisdictions**
  Channels can be configured to separate regional traffic and ensure data sovereignty while maintaining a global ordering service.
- **Integrate Existing Compliance APIs**
  AML and CFT compliance systems must be integrated as external chaincode invokers or block validators to meet international regulation (European Central Bank, 2021).

These steps, although non-trivial, offer a structured blueprint for transitioning from domestic transaction networks to multi-currency, cross-border blockchain ecosystems.

6.4 Limitations Of The Study

Despite its comprehensive design, the study has certain limitations which should be acknowledged:

- **Test Network Scale**
  The experimental setup included only three simulated banks and one observer. Larger consortium with heterogeneous latency and infrastructure could expose new scalability issues.

- **Simplified Chaincode Logic**
  The implemented chaincode mimicked basic debit-credit flows and endorsement policies. In real systems, logic may involve multi-stage validations, regulatory APIs, and anti-fraud analytics.
- **Absence of real Customer Data**
  While synthetic data mirrored realistic patterns, actual transaction behavior, fraud vectors, and operational anomalies may vary in production settings.
- **Security Analysis Out Of Scope**
  The study did not perform cryptographic audit or vulnerability scanning of Fabric network components. Production deployments would require formal threat modeling and hardening (Zamyatin and al., 2021).
- **Blockchain Interoperability Not Covered**
  The research focused solely on Hyperledger Fabric and did not address interaction with other DLTs such as Corda or Ethereum, which is increasingly relevant in global contexts.

These limitations suggest that while the study's results are directionally strong, real-world rollouts must be approached with careful planning, regulatory consultation, and infrastructure investment.

6.5 Future Work

Several extensions to this work can be envisioned. First, **scaling the network to 10+ banks**, including international entities with variable latency and compliance models, would allow deeper insight into Fabric's behavior at scale. Second, integrating **tokenized digital currencies or CBDC assets** into chaincode could simulate liquidity settlement alongside transaction commitments (International Monetary Fund, 2020). Third, developing a **reactive dashboard using AI/ML** to correlate transaction patterns with anomalous behavior could offer fraud detection on-chain. Finally, exploring **Fabric interop with Quorum or Corda** via relay chains or token bridges would broaden the ecosystem applicability and support future global banking systems (Zamyatin and al., 2021).

6.6 Final Thoughts

The financial industry is undergoing a paradigm shift where decentralization is no longer a niche experiment but a regulatory and operational necessity. This study shows that with appropriate design, observability, and governance, **Hyperledger Fabric can serve as a robust backbone for decentralized, compliant banking infrastructure**. While not a replacement for all microservices-based systems, it introduces new capabilities for collaboration, transparency, and trust that were previously managed by intermediaries. As global financial ecosystems evolve toward digital assets, programmable money, and identity-based trust, the frameworks laid out in this research offer a strong foundation to build upon (World Economic Forum, 2021).

REFERENCES

Androulaki, Elli and et al. (2018). "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *EuroSys*, pp. 1–15.

Appandairaj, A. and S. Murugappan (2013). "Service Oriented Architecture Design for Web-Based Home Banking Systems with Cloud Based Service". In: *International Journal of Emerging Technology and Advanced Engineering* 3.Special Issue 1, pp. 138–143. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=19b47ff6720d0274328a7c43bc7b845f1af92f6b.

Baliga, Arati, Prabhakaran Venkatesan, Madhu Chatterjee, Kaushik Srinivasan, and Siddhartha Chatterjee (2018). "Performance Evaluation of the Hyperledger Fabric Blockchain Platform". In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, pp. 1–10. URL: https://doi.org/10.1007/s11227-022-04361-2.

Banerjee, Rohan (2020). "Digital ID Interoperability and the Future of Financial Inclusion". In: *Journal of Digital Economy* 4.1, pp. 25–37.

Bernstein, Philip A. and Eric Newcomer (2009). *Principles of Transaction Processing*. 2nd. San Francisco: Morgan Kaufmann.

Brennan, Charles and David Lunn (2019). *CBDC Implementation Models: A Technical Review*. Tech. rep. Available at: https://www.bis.org/publ/othp33.htm. BIS Innovation Hub.

Cachin, Christian (2016). "Architecture of the Hyperledger Blockchain Fabric". In: *Workshop on Distributed Cryptocurrencies and Consensus Ledgers*.

Capocasale, Vittorio, Danilo Gotta, and Guido Perboli (2023). "Comparative analysis of permissioned blockchain frameworks for industrial applications". In: *Blockchain: Research and Applications* 4.1, p. 100113. ISSN: 2096-7209. DOI: https://doi.org/10.1016/j.bcra.2022.100113. URL: https://www.sciencedirect.com/science/article/pii/S2096720922000549.

Carnell, John and Illary Huaylupo Sánchez (2021). *Spring microservices in action*. Manning Publications Co.

Choudhury, Shruti R. (2021). *India's Digital Currency Trials: Policy Perspectives and Implementation Plans*. Tech. rep. Reserve Bank of India.

Dragoni, Nicola, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina (2017). "Microservices: Yesterday, Today, and Tomorrow". In: *Present and Ulterior Software Engineering* Future of Software Engineering Research, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

European Central Bank (2021). *Exploring Anonymity in Digital Currency Systems*. Tech. rep. Available at: https://www.ecb.europa.eu/press/intro/publications/pdf/ecb.mipinfocus191217.en.pdf. ECB.

Fowler, Martin (2014). *Microservices*. URL: https://martinfowler.com/articles/microservices.html.

Fowler, Martin and James Lewis (2014). *Microservices: a definition of this new architectural term*. martinfowler.com. URL: https://martinfowler.com/articles/microservices.html.

Ganesan, R. and K. Vivekanandan (2009). "A Secured Hybrid Architecture Model for Internet Banking (e-Banking)". In: *Journal of Internet Banking and Commerce* 14.1, pp. 1–17.

Gilbert, Seth and Nancy Lynch (2002). "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2, pp. 51–59. DOI: 10.1145/564585.564601.

Gray, Jim and Andreas Reuter (1993). *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann.

Hyperledger Foundation (2023). *Hyperledger Fabric Documentation (v2.5)*. Available at: https://hyperledger-fabric.readthedocs.io/en/release-2.5/.

International Monetary Fund (2020). *The Rise of Central Bank Digital Currencies*. Available at: https://www.imf.org/en/Publications/WP/Issues/2020/02/10/The-Rise-of-Central-Bank-Digital-Currencies-49082.

Johnson, D, K Dandapani, and M Sharokhi (2024). “Blockchain Applications in Finance”. In: *Blockchain Applications in Finance*.

Konkin, Anatoly and Sergey Zapechnikov (Jan. 2021). “Techniques for Private Transactions in Corporate Blockchain Networks”. In: *Proceedings of the 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2021*. Institute of Electrical and Electronics Engineers Inc., pp. 2356–2360. ISBN: 9780738142753. DOI: 10.1109/ElConRus51938.2021.9396228.

Mallidi, Ravi Kiran, Manmohan Sharma, and Sreenivas Rao Vangala (2022). “Streaming Platform Implementation in Banking and Financial Systems”. In: *2022 2nd Asian Conference on Innovation in Technology, ASIANCON 2022*. Institute of Electrical and Electronics Engineers Inc. ISBN: 9781665468510. DOI: 10.1109/ASIANCON55314.2022.9909500.

Mills, Andrew and et al. (2016). *Distributed Ledger Technology in Payments, Clearing, and Settlement*. Tech. rep. Federal Reserve Board.

Nakamoto, Satoshi (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. https://bitcoin.org/bitcoin.pdf.

Newman, Sam (2015). *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media.

— (2022). *Building microservices: Designing fine-grained systems*. O’Reilly.

Pavlovski, Chris (2013). “A Multi-Channel System Architecture for Banking”. In: *IJCSEA* 3.5, pp. 1–10.

Pongnumkul, Suphakant, Chalee Siripanpornchana, and Sawangpong Thajchayapong (2017). “Performance Analysis of Private Blockchain Platforms in Varying Workloads”. In: *IEEE International Conference on Communications Workshops (ICC Workshops)*, pp. 1–5. DOI: 10.1109/ICCW.2017.7962780.

Quan, Brandon Liew Yi, Nur Haliza Abdul Wahab, Juniardi Nur Fadila, Yichiet Aun, S. K.L. Luk, and Keng Yinn Wong (2024). “The Frontier of Blockchain Privacy: Development of a Private Ethereum Network”. In: *14th IEEE Symposium on Computer Applications and Industrial Electronics, ISCAIE 2024*. Institute of Electrical and Electronics Engineers Inc., pp. 117–122. ISBN: 9798350348798. DOI: 10.1109/ISCAIE61308.2024.10576280.

Rabah, Kassem (2019). “Blockchain Technology: Beyond Bitcoin for Secure and Decentralized Healthcare Ecosystem”. In: *International Journal of Advanced Computer Science and Applications (IJACSA)* 10.12, pp. 1–7. DOI: 10.14569/IJACSA.2019.0101201.

Walls, Craig (2017). *Spring Boot in Action*. Used to demonstrate practical microservices in Java.

Wang, Canhui and Xiaowen Chu (2020). “Performance Characterization and Bottleneck Analysis of Hyperledger Fabric”. In: *arXiv preprint arXiv:2008.05946*. URL: https://arxiv.org/abs/2008.05946.

World Economic Forum (2021). *Digital Currencies: A Guide to Cross-Border Collaboration and Governance*. Available at: https://www.weforum.org/reports/digital-currency-governance-consortium-whitepaper-series.

Wüst, Karl and Arthur Gervais (2018). “Do you need a Blockchain?” In: *Cryptology ePrint Archive, Report 2017/375*. URL: https://eprint.iacr.org/2017/375.pdf.

Zamyatin, Alexei and et al. (2021). “SoK: Communication Across Distributed Ledgers”. In: *ACM Conference on Advances in Financial Technologies (AFT)*, pp. 1–12.

Zheng, Zibin, Shaoan Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang (2017). “An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends”. In: *IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564. DOI: 10.1109/BigDataCongress.2017.85.

APPENDIX A : Research Proposal

Research Proposal Link

Please refer the Link for Details: Thesis Repository

APPENDIX B : Github Link of Code

Link to Github Repository of Code

Please refer the Link for Microservice & Hyperledger

APPENDIX C : Data Generation Scripts

Data Generation Scripts

Below is the Script used for Data Generation

```
import csv
import random
import uuid
from faker import Faker
from datetime import datetime, timedelta

fake = Faker()

banks = {
    "NWBANK": "NWB001",
    "OWBANK": "OWB001",
    "PWBANK": "PWB001"
}

def generate_account_number():
    return str(random.randint(100000000000, 999999999999))

def generate_ifsc(bank_code):
    return banks[bank_code]

def generate_balance():
    return round(random.lognormvariate(9.2, 0.8), 2)

def generate_transaction_type():
    return random.choice(["intra", "inter"])

def generate_customer_record(bank_code):
    return {
        "account_id": str(uuid.uuid4())[:12],
        "first_name": fake.first_name(),
        "last_name": fake.last_name(),
        "account_number": generate_account_number(),
        "ifsc_code": generate_ifsc(bank_code),
        "email": fake.email(),
        "phone": fake.phone_number(),
        "balance": generate_balance(),
        "registered_on": fake.date_between(start_date='-5y', end_date='today')
    }

def generate_transaction_dataset(bank_from, bank_to, n=1000):
    dataset = []
    for _ in range(n):
        from_account = generate_account_number()
```

```
        to_account = generate_account_number()
        dataset.append({
            "from_account": from_account,
            "to_account": to_account,
            "ifsc_from": generate_ifsc(bank_from),
            "ifsc_to": generate_ifsc(bank_to),
            "amount": round(random.uniform(100.0, 50000.0), 2),
            "currency": "INR",
            "timestamp": datetime.now().isoformat(),
            "transaction_type": generate_transaction_type(),
            "status_flag": random.choice([True, False]),
            "initiated_by": fake.user_name()
        })
    return dataset

def save_to_csv(filename, records, headers):
    with open(filename, mode='w', newline='') as file:
        writer = csv.DictWriter(file, fieldnames=headers)
        writer.writeheader()
        for record in records:
            writer.writerow(record)

if __name__ == "__main__":
    customers = [generate_customer_record(bank) for bank in banks for _ in range(300)]
    save_to_csv('synthetic_customers.csv', customers, customers[0].keys())

    transactions = generate_transaction_dataset("NWBANK", "OWBANK", 1000)
    save_to_csv('synthetic_transactions.csv', transactions, transactions[0].keys())
```

APPENDIX D : Docker Compose Yaml

The Configuration file for the docker network

Since configuration files with secret key files are not uploaded on the internet, here is the complete docker-compose.yaml file for the springboot system:

```
   ##########################################################################################################################
# NW Bank Services

    #
##########################################################################################################################

services:
  gateway-nwbank:
    build:
      context: ./NWBank/NWB-API-Gateway-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: gateway-nwb
    volumes:
      - ./NWBank/NWB-API-Gateway-develop/target:/app/target
    environment:
      - SERVER_PORT=8080
      - IFSC_CODE=NWB0001
      -
          SECRET_KEY=ipm89FyJ5it9gIYB2E+FZM9il3e2mQtQsDqH/GBU4iJQF2lynN9xsM1vMSZmCHlpsUdzfIqvfqksVr4NoFTKpw==
    ports:
      - "8081:8080"
    networks:
      - nwnetwork
    depends_on:
      - authentication-nwbank
      - accounts-nwbank
      - transaction-nwbank

  authentication-nwbank:
    build:
      context: ./NWBank/NWB-Authentication-Service-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: authentication-nwb
    volumes:
      - ./NWBank/NWB-Authentication-Service-develop/target:/app/target
    environment:
      - SERVER_PORT=8100
      - IFSC_CODE=NWB0001
```

```
    -
        SECRET_KEY=ipm89FyJ5it9gIYB2E+FZM9il3e2mQtQsDqH/GBU4iJQF2lynN9xsM1vMSZmCHlpsUdzfIqvfqksVr4NoFTKpw==
    - ROOT_USER=admin
    - ROOT_PASSWORD=changeit
    - DB_URL=jdbc:mysql://host.docker.internal:3306/NWBank
    - DB_USERNAME=NWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=NWBank
  ports:
    - "8082:8100"
  networks:
    - nwnetwork

accounts-nwbank:
  build:
    context: ./NWBank/NWB-Accounts-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: accounts-nwb
  volumes:
    - ./NWBank/NWB-Accounts-develop/target:/app/target
  environment:
    - SERVER_PORT=8200
    - IFSC_CODE=NWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/NWBank
    - DB_USERNAME=NWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=NWBank
  ports:
    - "8083:8200"
  networks:
    - nwnetwork

transaction-nwbank:
  build:
    context: ./NWBank/NWB-Transaction-Service-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: transaction-nwb
  volumes:
    - ./NWBank/NWB-Transaction-Service-develop/target:/app/target
  environment:
    - SERVER_PORT=8300
    - IFSC_CODE=NWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/NWBank
    - DB_USERNAME=NWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=NWBank
    - CLEARING_HOUSE_URL=http://clearinghouse:9090
```

```
    ports:
      - "8084:8300"
    networks:
      - nwnetwork


##########################################################################################################################
# OW Bank Services

     #
##########################################################################################################################

  gateway-owbank:
    build:
      context: ./OWBank/OWB-API-Gateway-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: gateway-owb
    volumes:
      - ./OWBank/OWB-API-Gateway-develop/target:/app/target
    environment:
      - SERVER_PORT=8080
      - IFSC_CODE=OWB0001
      -
          SECRET_KEY=YjWbDI1cSmUCAqTvlfdsKVhmeBjRV2DwpbBL7bynCxTKQeiNIQKr+1TeeXLlZkLeYoguNDhiQDOoy3kz2eHfbw==

    ports:
      - "8085:8080"
    networks:
      - ownetwork
    depends_on:
      - authentication-owbank
      - accounts-owbank
      - transaction-owbank

  authentication-owbank:
    build:
      context: ./OWBank/OWB-Authentication-Service-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: authentication-owb
    volumes:
      - ./OWBank/OWB-Authentication-Service-develop/target:/app/target
    environment:
      - SERVER_PORT=8100
      - IFSC_CODE=OWB0001
      -
          SECRET_KEY=YjWbDI1cSmUCAqTvlfdsKVhmeBjRV2DwpbBL7bynCxTKQeiNIQKr+1TeeXLlZkLeYoguNDhiQDOoy3kz2eHfbw==
      - ROOT_USER=admin
```

```
    - ROOT_PASSWORD=changeit
    - DB_URL=jdbc:mysql://host.docker.internal:3306/OWBank
    - DB_USERNAME=OWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=OWBank
  ports:
    - "8086:8100"
  networks:
    - ownetwork

accounts-owbank:
  build:
    context: ./OWBank/OWB-Accounts-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: accounts-owb
  volumes:
    - ./OWBank/OWB-Accounts-develop/target:/app/target
  environment:
    - SERVER_PORT=8200
    - IFSC_CODE=OWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/OWBank
    - DB_USERNAME=OWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=OWBank
  ports:
    - "8087:8200"
  networks:
    - ownetwork

transaction-owbank:
  build:
    context: ./OWBank/OWB-Transaction-Service-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: transaction-owb
  volumes:
    - ./OWBank/OWB-Transaction-Service-develop/target:/app/target
  environment:
    - SERVER_PORT=8300
    - IFSC_CODE=OWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/OWBank
    - DB_USERNAME=OWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=OWBank
    - CLEARING_HOUSE_URL=http://clearinghouse:9090
  ports:
    - "8088:8300"
  networks:
```

```
      - ownetwork


##########################################################################################################################
# PW Bank Services

    #
##########################################################################################################################
  gateway-pwbank:
    build:
      context: ./PWBank/PWB-API-Gateway-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: gateway-pwb
    volumes:
      - ./PWBank/PWB-API-Gateway-develop/target:/app/target
    environment:
      - SERVER_PORT=8080
      - IFSC_CODE=PWB0001
      -
          SECRET_KEY=JT+b6WIcwHkVbx2O7Iofw/wWEQpvK+Mpa9+MCvmS3s+ELYXF6zf4YCHa1OafQmTO3USTDEyyWnX1C4n/5BEk4w==

    ports:
      - "8089:8080"
    networks:
      - pwnetwork
    depends_on:
      - authentication-pwbank
      - accounts-pwbank
      - transaction-pwbank

  authentication-pwbank:
    build:
      context: ./PWBank/PWB-Authentication-Service-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: authentication-pwb
    volumes:
      - ./PWBank/PWB-Authentication-Service-develop/target:/app/target
    environment:
      - SERVER_PORT=8100
      - IFSC_CODE=PWB0001
      -
          SECRET_KEY=JT+b6WIcwHkVbx2O7Iofw/wWEQpvK+Mpa9+MCvmS3s+ELYXF6zf4YCHa1OafQmTO3USTDEyyWnX1C4n/5BEk4w==
      - ROOT_USER=admin
      - ROOT_PASSWORD=changeit
      - DB_URL=jdbc:mysql://host.docker.internal:3306/PWBank
      - DB_USERNAME=PWB_USER
      - DB_PASSWORD=PASS@123
```

```
    - DB_SCHEMA=PWBank

  ports:
    - "8090:8100"
  networks:
    - pwnetwork

accounts-pwbank:
  build:
    context: ./PWBank/PWB-Accounts-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: accounts-pwb
  volumes:
    - ./PWBank/PWB-Accounts-develop/target:/app/target
  environment:
    - SERVER_PORT=8200
    - IFSC_CODE=PWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/PWBank
    - DB_USERNAME=PWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=PWBank
  ports:
    - "8091:8200"
  networks:
    - pwnetwork

transaction-pwbank:
  build:
    context: ./PWBank/PWB-Transaction-Service-develop
    dockerfile: Dockerfile
  extra_hosts:
    - "host.docker.internal:host-gateway"
  container_name: transaction-pwb
  volumes:
    - ./PWBank/PWB-Transaction-Service-develop/target:/app/target
  environment:
    - SERVER_PORT=8300
    - IFSC_CODE=PWB0001
    - DB_URL=jdbc:mysql://host.docker.internal:3306/PWBank
    - DB_USERNAME=PWB_USER
    - DB_PASSWORD=PASS@123
    - DB_SCHEMA=PWBank
    - CLEARING_HOUSE_URL=http://clearinghouse:9090
  ports:
    - "8092:8300"
  networks:
    - pwnetwork
```

```
  clearinghouse:
    build:
      context: ./NWB_ClearingHouse-develop
      dockerfile: Dockerfile
    extra_hosts:
      - "host.docker.internal:host-gateway"
    container_name: clearingHouse
    environment:
      - SERVER_PORT=9090
    ports:
      - "9091:9090"
    networks:
      - nwnetwork
      - ownetwork
      - pwnetwork
    depends_on:
      - transaction-nwbank
      - transaction-owbank
      - transaction-pwbank

networks:
  nwnetwork:
  ownetwork:
  pwnetwork:
```