Scalability Management in a Microservice System: A Case Study of Cameroon Banking System - An Experimental Approach

Djam Xaveria Youh
University of Yaounde 1, Cameroon

Tapamo Kenfack Hyppolyte Michel
University of Yaounde 1, Cameroon

Aminou Halidou
University of Yaounde 1, Cameroon

Atsa Roger Etoundi
University of Yaounde 1, Cameroon

ABSTRACT
The advent of microservice architecture has brought unprecedented success in the banking sector. In this present era, it is imperative for banking sector to possess adequate preparedness to effectively manage fluctuations in transaction volumes during peak periods due to scalability changes. This paper presents a methodological approach to solving the problem of scalability in banking operations using Cameroon context. Microservice architectures, with their modularity and scalability, have proven to be a relevant answer to these challenges. This research explores common problems encountered in microservices, proposes a systematic methodology to address them, and illustrates these concepts through the development of a banking platform integrating modern technologies such as Docker, Docker Compose, and Kubernetes. Furthermore, load balancing techniques were examined, which were essential to optimize the performance of the banking application, and their impact on the efficiency of microservices.

Crossover and mutation operators of the Adaptive Genetic Algorithm(AGA) were adopted to avoid premature convergence and to minimize plateau, which enhanced the diversity of population evolution and effectively reduced data transmission time between banking services.

The development environment was properly set up to support the research goals. This includes ensuring the availability of essential tools such as Visual Studio Code, Eclipse, Java Development Kit(JDK), Apache Maven, Docker Desktop, Rester, Hey, RabbitMQ, Spring Boot, Actuator, Spring Cloud Gateway, Spring Cloud Config, Eureka, H2, MariaDB. The Hey tool was used for request tracing among microservices.

To deploy this solution, modern technologies such as Docker, Docker Compose, and Kubernetes were employed which allowed efficient container management and service orchestration. Finally, performance and scalability tests were performed using the Hey tool, in order to evaluate the efficiency of our architecture and interpret the results for possible improvements.

For performance testing analysis, four metrics were used that indicated satisfactory performance. The total response time averaged 0.6156 seconds, with the fastest response at 0.0056 seconds and the slowest at 0.2388 seconds. The average response time was 0.0545 seconds, achieving a throughput of 1624.5330 requests per second. A histogram analysis indicated that the majority of requests (406) fell within a response time of 0.052 to 0.076 seconds, confirming the system's overall efficiency. Latency distribution analysis showed that 50% of requests had latency under 0.0499 seconds, while 90% were below 0.0999 seconds, and 99% 0.1999 in seconds. No significant bottlenecks were identified in the various process steps, including Domain Name System (DNS) resolution and response handling.

General Terms
Software Engineering, Search-based Software Engineering

Keywords
Scalability, Optimization, Orchestration, Microservice architecture, Event-Driven-Controller, Adaptive Genetic Algorithm

1. INTRODUCTION
Microservice architecture has become a standard in modern application development, especially in critical industries like banking. This architecture breaks applications into independent services, enabling unprecedented flexibility and scalability. However, this approach also presents unique challenges, including communication between services, data management, deployment, and performance monitoring. In the era of digitalization, financial institutions must quickly adapt to the growing expectations of customers for secure, efficient, and flexible solutions by adopting modern microservice approach.

In today's digitalization landscape, financial institutions are facing increased pressure to deliver solutions that meet their customers' growing expectations. These expectations include not only the speed and security of transactions, but also the ability to quickly adapt to a constantly changing environment. Microservice architectures are emerging as an effective response to these challenges, enabling modularity that fosters innovation while ensuring enhanced security.

In the current context of digitalization of Cameroon banking system, financial institutions must quickly adapt to their customers' growing expectations for secure, efficient and flexible solutions. Microservice architecture emerges as a relevant response to these challenges, enabling unprecedented modularity and scalability in the development of modern banking platforms.

This article aims to design a banking platform that not only facilitates financial transactions but also meets scalability and security challenges requirements. Customers will be able to make deposits and withdrawals seamlessly, while having the ability to transfer funds between different operators, such as UBA BANK and NFC BANK in Cameroon context. To ensure an optimal user experience, real-time notifications will inform customers of the status of their transactions.

This paper proposes a systematic methodology to identify and solve common problems associated with microservice architectures, focusing on a case study of a banking operations platform. This research examines functional and non-functional

requir, modeling techniques, as well as modern technologies such as Docker and Kubernetes, which play a crucial role in implementing and managing these architectures. By integrating concepts such as event-driven architecture and asynchronous communication, this research demonstrates how these approaches can improve the resilience, performance, and maintainability of banking systems.

A crucial aspect of this platform is the validation of account creation requests by bank staff, thus ensuring the security of operations. In addition, an Event-Driven Architecture (EDA) will be implemented to facilitate communication between the different microservices, thus strengthening the responsiveness and resilience of the system.

2. RESEARCH BACKGROUND

2.1 Common Problems of Microservices

The challenges encountered in microservice architectures are varied and can have a significant impact on system performance and reliability:

- **Communication between Services** : Microservices often need to communicate, which can lead to latency and reliability issues. Solutions include using lightweight protocols like HTTP/REST or gRPC, and integrating message brokers for asynchronous communication.
- **Data Management** : Decentralization of data makes it difficult to ensure consistency and integrity. Using appropriate data models and synchronization strategies is essential.
- **Deployment and Orchestration** : Managing frequent updates and deployments can become complex. Using tools like Kubernetes makes container orchestration and management easier.
- **Performance Monitoring** : System-wide visibility is crucial to detect issues early. Integrating monitoring tools like Prometheus or Grafana is recommended.

2.2 Load Balancing Mechanism in Microservice Applications

In recent years, there has been a lot of researches into modern microservice technologies. Many organizations are adopting these technologies to keep up with modern software development demands. This has led to a growing interest in understanding the challenges, best practices, and emerging trends in this area. This section aims to provide an overview of existing literature and research studies on microservice load balancing mechanism.

Load balancing is essential to optimize the performance of microservice-based applications. According to Sharma et al. [3], several load balancing techniques can be applied to efficiently manage workloads.

Alexander Sundberg [7] addresses load balancing algorithms for networked systems in a microservice application. He concluded that there is a lack of proposed load balancing algorithms for microservices, and it is not obvious how to adapt such algorithms to the architecture under consideration.

Also Shitole and Abishek Sanjay [6] proposed a technique that uses service-mesh to inject sidecar proxies into every microservice and dynamically balances the load among services by applying service-specific routing. The experimental results proved that the proposed design outperforms the traditional approach by maintaining stability and consistency in response rate and consumes fewer resources.

Microservice architecture aims to decompose a monolithic application into a set of independent services which communicate with each other through open APIs or highly scalable messaging [8] - [14].

The above mentioned researches did not explicitly explained how scalability can be effectively manage in a microservice application to reduce overload with effective authentication and monitoring.

3. PROPOSED APPROACH FOR MICROSERVICE BANKING APPLICATION

3.1 Design Solutions

Designing solutions for a microservice architecture must take into account several concepts and tools to ensure resilience, performance, and maintainability. The main features in a microservice application is to produce a scalable highly cohesive and loosely coupled distributed services. In view of the above, in this research, a three-phased approach was used to manage the problems of scalability in a banking application. The phases are:

Phase one: This phase consists of identifying the different services used in the banking application. Domain Driven Design (DDD) method was used to identify all the various services.

Phase two: This phase consists of representing the microservice architectural framework elements from user's view point. This architecture is presented in Figure 3.1 to Figure 3.6

Phase three: This phase consists of designing the architectural solution using the template in phase two. This architecture is presented in Figure 3.7

The architecture is presented in Figure 3.7 and comprises of the following services

3.1.1 Domain Driven Design

In this article a domain driven approach was used to identify the services in the microservice banking system. This approach to software development was initially explained in 2003 by Eric Evans in his book titled "Domain-Driven Design: Tackling Complexity in the Heart of Software," [11] centering the structuring and design of the system around the domain of knowledge and activity of the system. Its aim is to create a detailed domain model that represents the problem space the software must address, ensuring that the final software product is suitable for solving the problems of the business domain. This technique has two main components:

- **The Omnipresent Language**: This refers to establishing a common language between stakeholders and developers for domain description and solution design. It aims to ensure a shared understanding of the system among stakeholders.
- **Bounded Context**: A central element of Domain-Driven Design (DDD), where the "context" defines the boundary within which a model applies. This approach corresponds to the model defined according to the knowledge of experts and is essential for defining the structure of all models or sub-models, as well as for

limiting complexity. Given that the size of the microservice is relative, the domain is divided into subdomains, and each subdomain will be used to determine the microservices.

3.1.2 System Architecture

The system architecture of this research comprises of two main system services which includes functional system services and non-functional system services as depicted in Figure 3.1. The architecture shows the overall idea of the process of scheduling and load balancing of services in the banking system. The scheduler and load balancer are the components that use the load balancing optimization algorithm known as adaptive genetic algorithm (AGA) to distribute the loads to resources with low waited response time. The API Gateway (Bank_Proxy_Service) is a component that helps in deciding the appropriate Visual Machine (VM) to which an arrived task should be allocated. It ensures that the task is assigned to a VM that takes less time to complete the task by considering the amount of load in that specific VM and the total time needed to complete the entire load.

Representing the microservice architectural framework elements from user's view

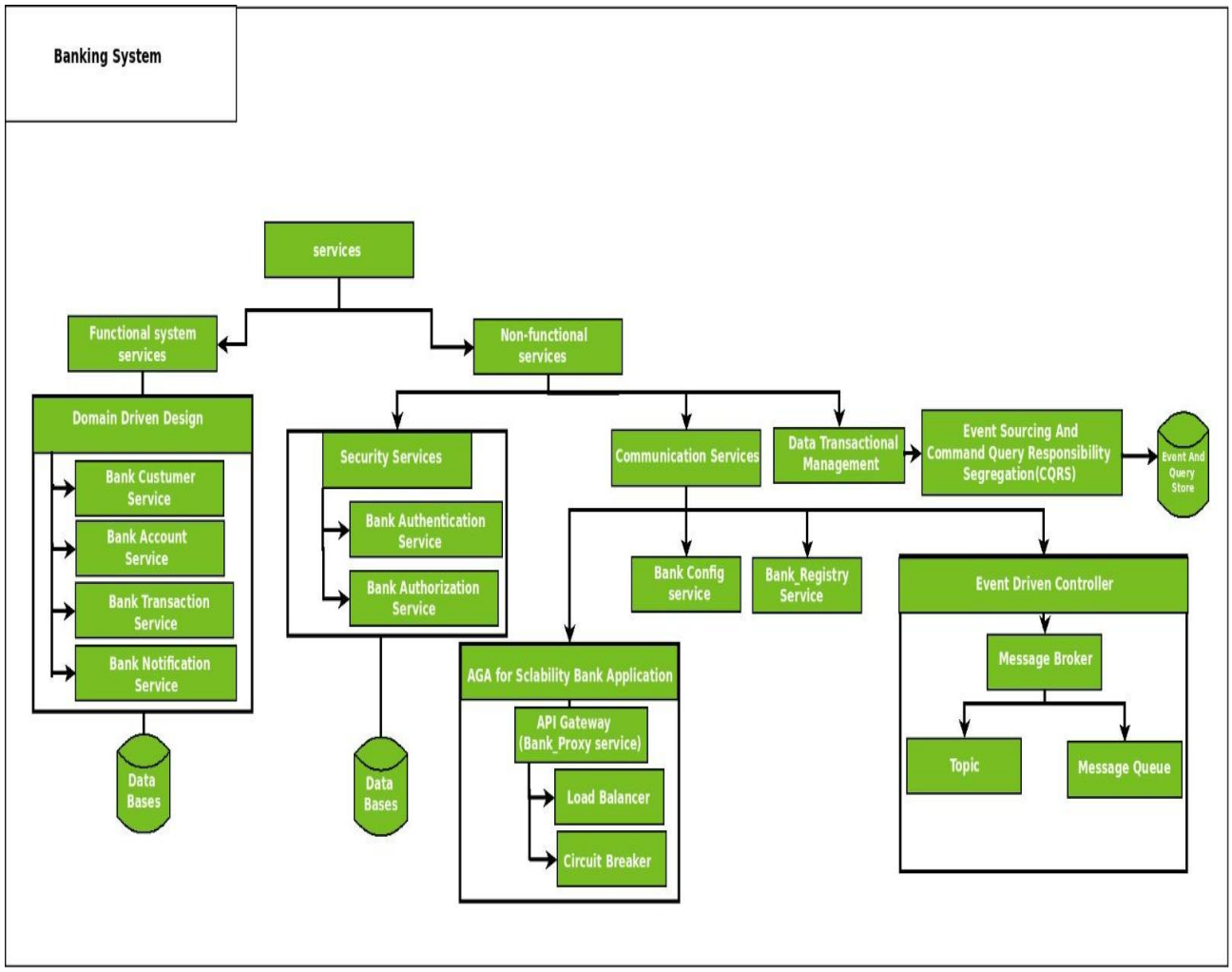

Figure 3.1: Microservice Architectural Framework elements from user's view

In this article, the microservice banking system is designed according to two types of services functional and non-functional services Figure 3.1:

- **Functional Services**: These are services from the user's perspective. These were obtained after applying Domain-Driven Design (DDD) as seen in Figure 3.2.

- **Non-Functional Services**: These are services from the developer's perspective. These services are essential for implementing the microservice architecture and are generally reused across microservice software systems. In the revolutionary context, this part defines a foundational model within a microservice architecture.

3.1.3 Functional Services

Applying Domain-Driven Design (DDD), the domain and subdomains of the microservice bank management system were derived such as: customer management which include User Service and Company Service, transaction management, account management, and notification management, which are then considered as the core microservices in the research.

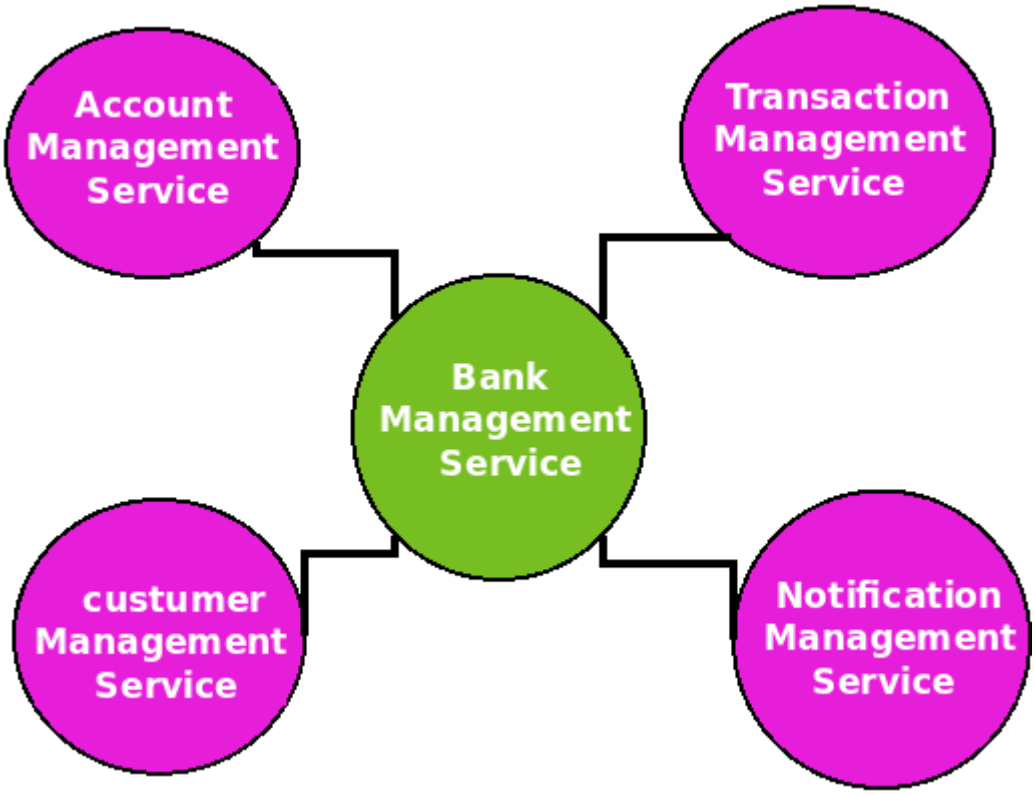

Figure 3.2: Context Domain Driven Design of a banking system

3.1.4 Non-Functional Services

There are three core non-functional services of the microservice banking application : services that manage system security, services that manage communication, and the service for managing transactional data (Figure 3.1).

Security Services: The security service manages authentication and authorization within the system.

Communication Services: Communication services ensure communication and maintain interactions between the system's services (functional and non-functional services). These services include: (Bank_configuration Service, Bank_Registry Service, Event-Driven Controller)

Configuration Service: Centralize configuration management for all services. Service configuration is about managing the configuration of services. In a microservice architecture, it is essential to manage configurations centrally to avoid inconsistencies. This pattern uses configuration servers, such as Spring Cloud configuration, which allow service configuration to be stored and distributed centrally, allowing updates without restarting services.

Registry Service : Implement a registry service to facilitate service discovery. The service registry is a component that maintains a list of available services in a microservice system. With service registry in place, each service registers itself with the registry when it starts and de-registers itself when it stops. This allows other services to dynamically discover available service instances, facilitating communication between them. Eureka server was used to implement this pattern.

Event Bus Handler: This module integrates a service called 'message broker' acting as an event bus. Its main objective is to ensure asynchronous communication between the system's services. This service operates on the basis of an Event Driven Controller (EDC) implementing the Publish And Subscribe (PUBSUB) model (Figure3.3).

EDC: It is an architectural style that solves communication problems in the microservice banking system. EDC on producing, detecting, and reacting to events.

Event: An event is a notification that a state change has occurred in a system. This can be a user action, a data update, or a service state change. In this model, system components communicate primarily through events, which allows for asynchronous and decentralized interaction. The main components in the event-driven architecture designed in this work are:

Event Producers (Producers in functional services): The services or applications that generate events.

Event Consumers (Consumers in functional services): Services or applications that react to events by performing specific actions.

Event mediators: Message Brokers (RabbitMQ) was used to handle the transmission of events between producers and consumers.

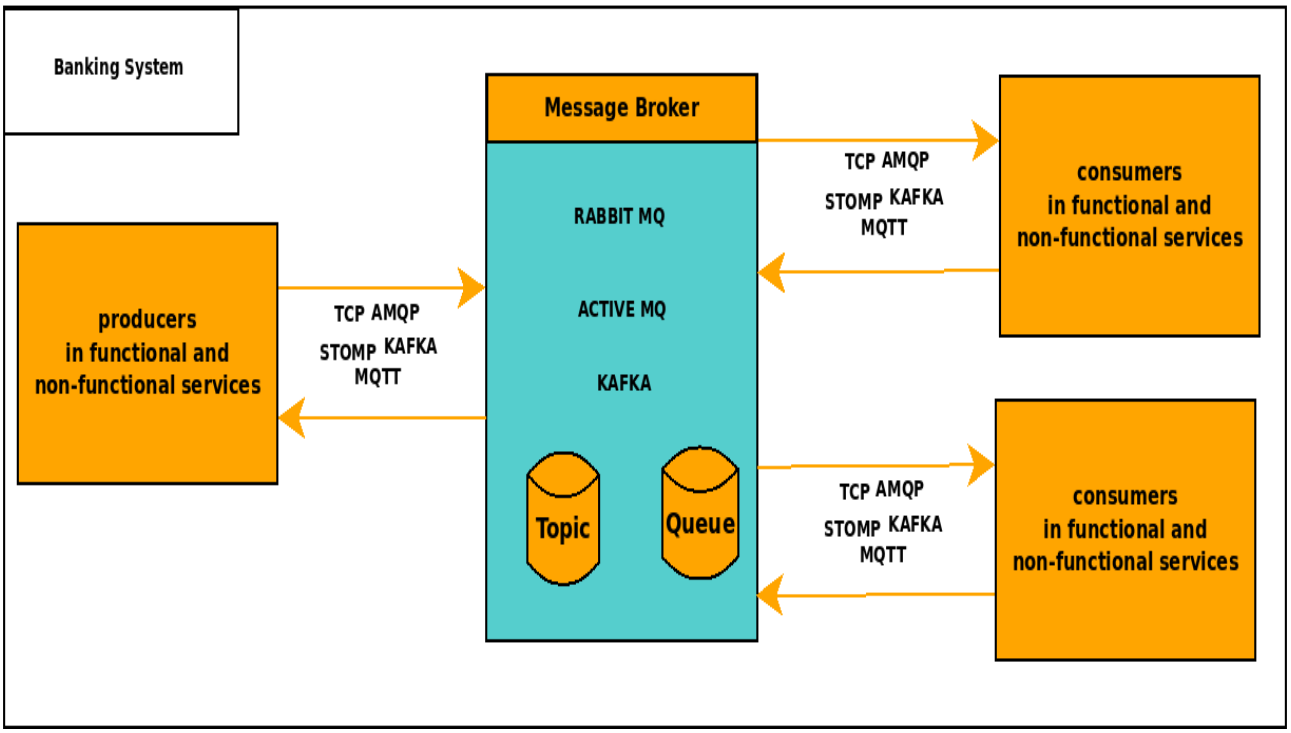

Figure 3.3: Event-driven Controller for Asynchronous Communication:PUBSUB Model

API Gateway :

It is a single entry point to handle client requests. It handles requests, redirects them to the appropriate services, and can also perform tasks such as authentication, rate limiting, and error handling. This concept simplifies the architecture for customers and improves security. Spring cloud Gateway was used to established API gateway. This service integrates two concepts: the load balancer implementing the Parallel Genetic Algorithm (PGA) for scalability in bank applications and the circuit breaker.

3.1.5 Adaptive Genetic Algorithm (AGA) for Microservice Banking System

This algorithm takes two inputs: a set of BankInstanceService I and a set of Requests R for an initial population. It initializes the variable CurrentIndex (the index specifying the position of the service instance that must process the request) to 0 and the Population with the set of incoming Requests R. If the BankServiceInstance set is empty, the algorithm returns null. Then, the algorithm calls the FitnessController function, which evaluates the quality of each request in the Population and creates a NewPopulation (a new set of requests), generated using genetic operations such as selection, crossover, and mutation. For each request in the NewPopulation, the algorithm selects the service instance from the BankServiceInstances array using the value of CurrentIndex, sends the request to the selected service instance, and then updates CurrentIndex (resetting it to 0 if it is equal to the length of the BankServiceInstances array, otherwise incrementing it by 1) for the service instance to execute the next request. Finally, the algorithm returns to Step 8 unless a stopping criterion is met. The flowchart of this algorithm is shown in Figure 3.4.

Stopping conditions: BankInstanceService is empty, the FitnessController function returns null or an empty array.

Adaptive Genetic Algorithm

Step1: **Step1:** Begin:
Step2: Input: Set of BankInstanceService I, Set of Requests R
Step3: Initialize CurrentIndex=0;
Step4: Initial Population R,
Step5: If BankServiceInstances.length =0
Step6: return null
Step7: End If
Step8: NewPopulation=FitnessController(R)
Step9: genetic operations (selection, crossover and mutation)
Step10: For each request r to NewPopulation do
Step11: Select the service instance SI=BankServiceInstances[CurrentIndex]
Step12: Send request r to SI
Step13: If CurrentIndex=BankServiceInstances.length
Step14: CurrentIndex=0;
Step15: Else
Step16: Update CurrentIndex=CurrentIndex+1
Step17: End If
Step18: End For
Step19: Return to Step 8 unless a termination criterion is satisfied.

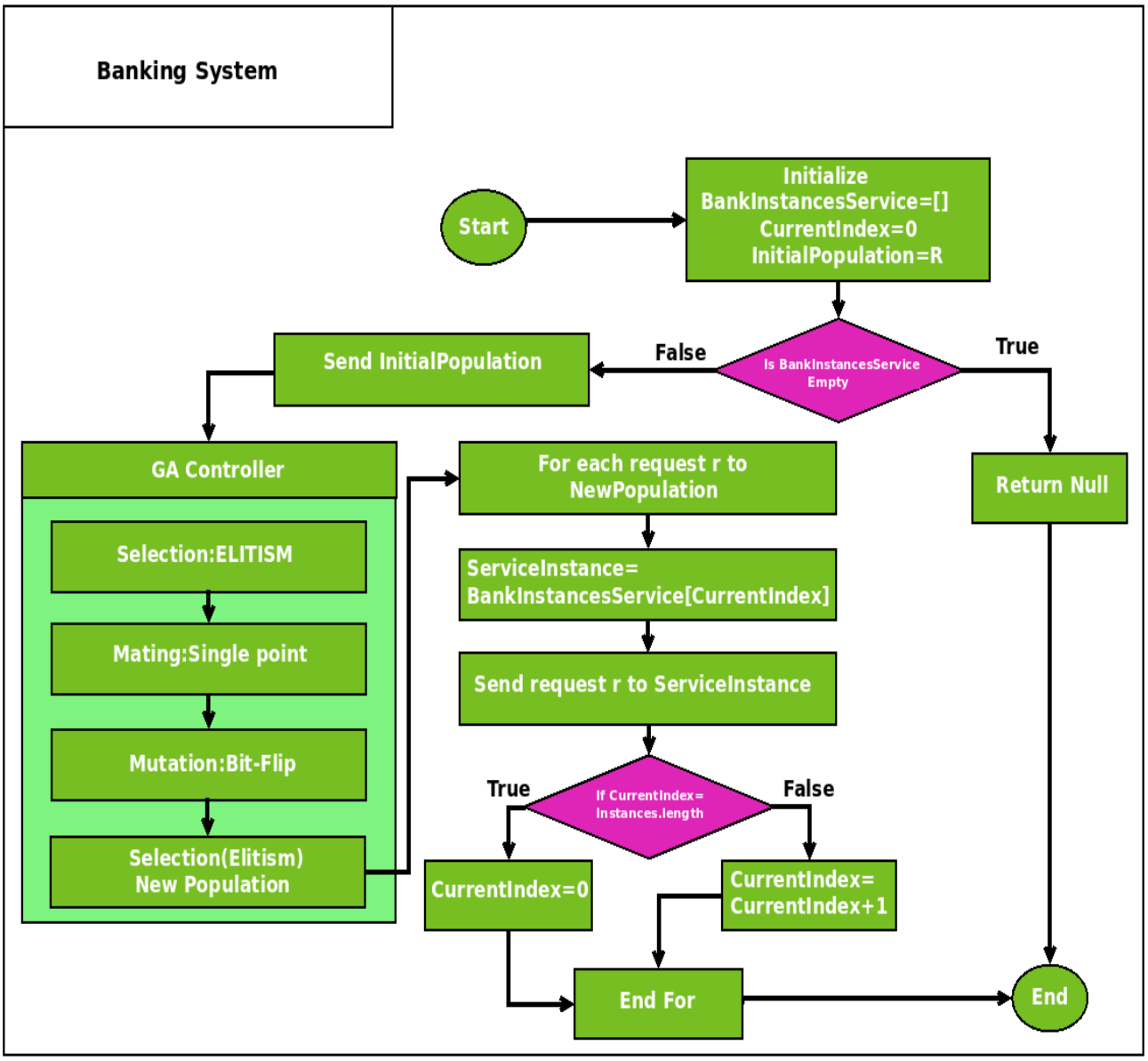

Figure 3.4: Flow diagram for Adaptive Genetic Algorithm (AGA) for Microservice Banking System

3.2 Circuit Breaker in Microservices

The Circuit Breaker pattern is essential for handling failures in microservices. It helps detect failures in service calls and prevents repeated calls to a service that is likely to fail. This reduces latency and improves system resilience. When a service fails multiple times, the circuit breaker "opens", preventing new requests for a set period of time. This allows the service to recover without being overwhelmed by requests. The circuit breaker pattern is a design technique used in the microservice system to handle failures. It consists of three main states:

- HALF OPEN : This state indicates that the circuit breaker has been triggered due to a failure, but is trying to retry a connection. During this phase, requests are allowed to pass with a delay.

- OPEN : When the failure threshold is exceeded, the circuit breaker goes into OPEN state. In this state, all requests are rejected without being sent to the failing service, in order to avoid the propagation of errors.

- CLOSED : When the circuit breaker detects that the service is available again, it goes into CLOSED state, allowing normal traffic again.

This pattern is presented in Figure3.4

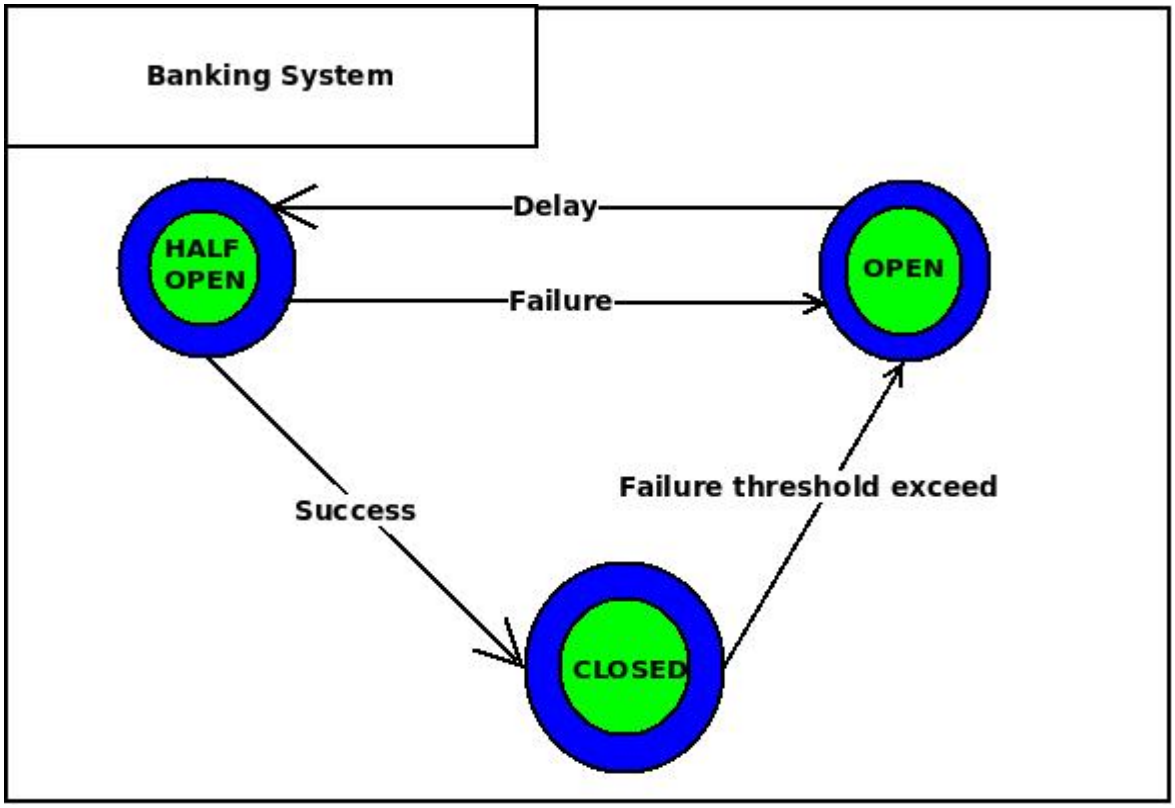

Figure 3.5: Circuit Breaker Pattern

3.3 Data Transactional Management Service

This service primarily aims to manage the history of banking operations. To achieve this, it integrates two concepts: Event Sourcing and Command Query Responsibility Segregation(CQRS). These are concepts that provide patterns for storing data in the form of events.

Event Sourcing: Event Sourcing **enables** full auditability and the ability to roll back to a previous state by replaying events. This is particularly useful in the banking sector to ensure regulatory compliance.

CQRS: The separation of responsibilities between command operations (writes) and query operations (reads) is an architectural pattern known as CQRS. In the context of the banking microservice application, this approach improves data consistency and performance. Write operations, such as account updates or transactions, are managed by one set of services and databases (Bank Event Store), while read operations, such as balance inquiries or transaction histories, are handled by another set, potentially using different data models or replication strategies.

This banking microservice platform utilizes CQRS to differentiate between the management of transactions (commands) and the retrieval of account information (queries). This allows the system to optimize each type of operation independently, thereby improving performance and scalability. Figure 3.6 illustrates the workings of Event Sourcing and CQRS.

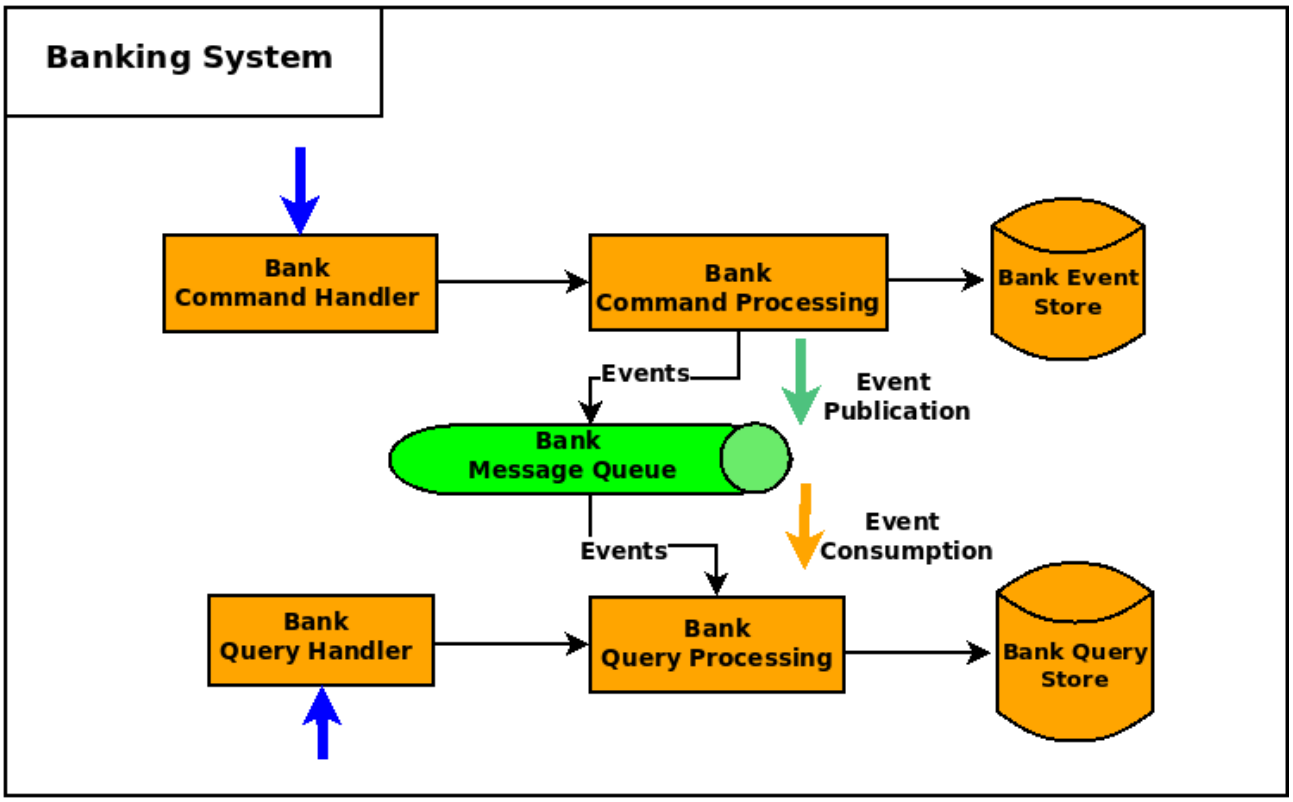

Figure 3.6: Event Sourcing and CQRS

Order handler , which is responsible for receiving all order requests was incorporated. The section dedicated to order processing takes care of handling these orders and generating the corresponding events. Before saving these events in the event store, validations and business rules are applied. Once the events are created, they are published to a message queue.

3.4 Design the architectural solution using the template from phrase two and implement the solution

These queues can be managed by brokers such as RabbitMQ or Kafka. The query processing application monitors these events. It typically extracts the payload from the event and saves the data to the query store, based on the required read patterns. The query handler section is responsible for processing incoming read requests, retrieving data from the query store to serve to users.

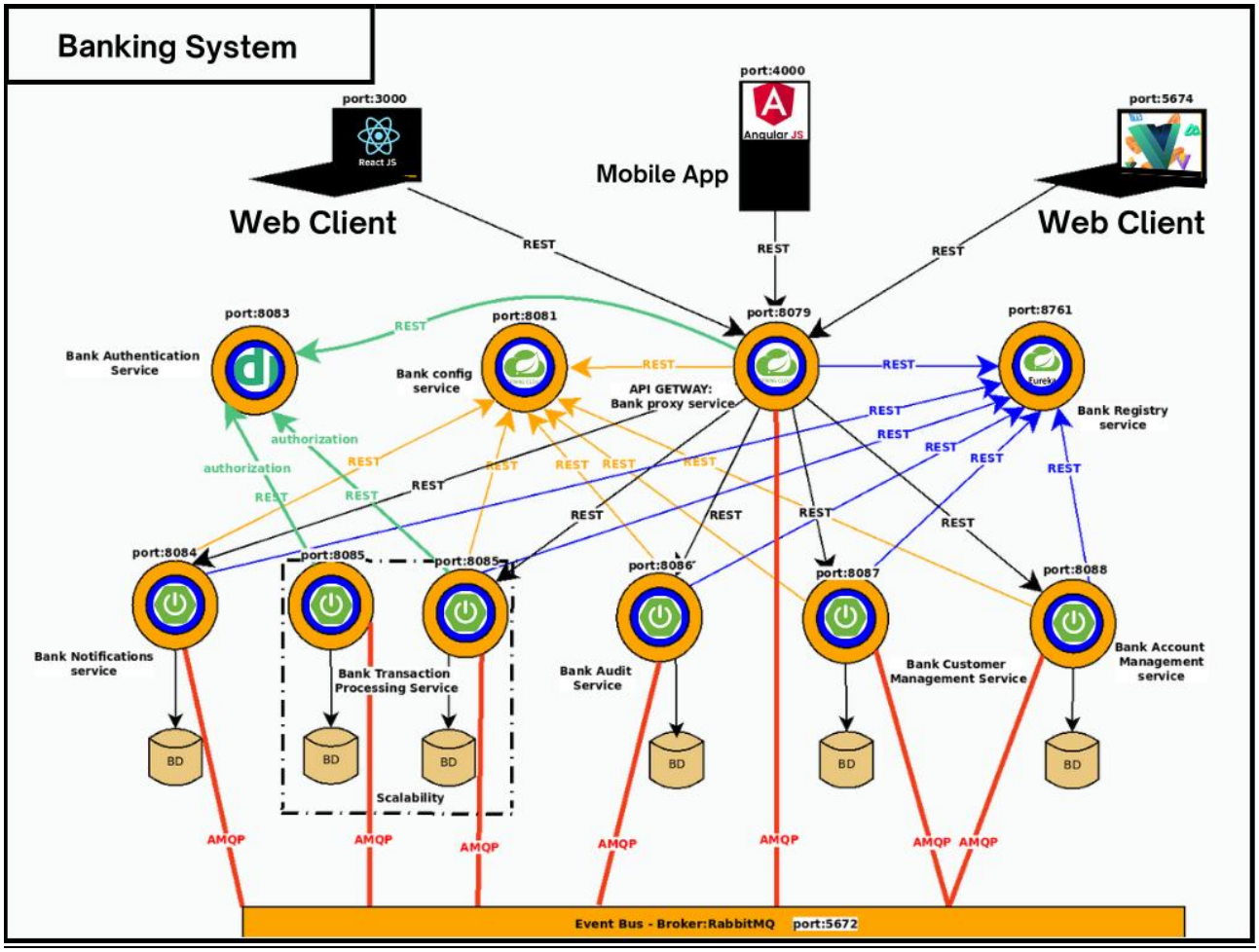

Figure 3.7: Microservice Banking architecture: Cameroon Context

In this microservice banking architecture with respect to Cameroon context, each service initiates a configuration request to the configuration server before starting. Once up and running, the service sends its name, address, and port to the registration server, and unregistered itself when it stops. The registration server therefore plays a crucial role in maintaining information about the various services available.

When a request is received by the API Gateway, it consults the registration server to determine the appropriate destination based on the information of the service concerned. Thus, the communication between a client and a microservice is not direct, which strengthens the architecture.

To facilitate communication between two microservices, an event bus is typically employed. While communication can also be done via REST APIs, this method is synchronous and can introduce latencies, resulting in a slowdown in overall system performance. By opting for an event bus, asynchronous communication was established that allows for efficient decoupling between microservices.

RabbitMQ, which uses the AMQP protocol, was used for asynchronous messaging, providing a robust solution for handling event exchanges between services.

3.5 Case Studies: Design of a Microservice Banking System platform - Cameroon Perceptive

3.5.1 Account Management Service

When implementing a microservice architecture for a banking platform, it is essential to define a set of interconnected services that meet the functional and non-functional needs of the institution.

An account management service was set up, allowing not only the creation of accounts, but also the validation of requests by bank staff. This service facilitates the consultation of account details, thus ensuring a smooth user experience. At the same time, a transaction processing service will be set up, offering features such as deposits and withdrawals of money, as well as transfers between accounts, whether within the same operator or between different operators. This service equally included a transaction history to ensure complete traceability.

To improve customer interaction, a notification service was integrated (see Figure 3.1). This will allow real-time notifications about deposits and withdrawals, as well as security alerts in case of suspicious login attempts. In addition, a security service was implemented to manage user authentication, access authorizations, and monitor suspicious activities.

Customer information management was handled by a dedicated service called Customer Management Service (see Figure 3.2), which can track customer interactions with the bank and ensure

effective customer support. In addition, an audit department is essential to track account and transaction changes, logging activities for compliance purposes and producing reports for regulators.

The system incorporates an API Gateway, which acts as a single entry point for all client requests. This gateway will be responsible for routing requests to the appropriate services and handling responses. A registration service was be implemented to maintain information about the various services available, allowing each service to register with a configuration server before starting. Once up and running, each service will communicate its name, address, and port to the registration server, and will unregistered when it stops.

To facilitate communication between bank microservices, an event bus was used. While communication can also be done via REST APIs, this mode of communication can introduce latencies and slow down the overall performance of the system. By opting for an event bus, asynchronous communication was established, thus promoting efficient decoupling between microservices while encouraging high cohesion. RabbitMQ, which uses the AMQP protocol, was employed for asynchronous messaging, offering a robust solution to manage event exchanges between bank services.

Example of Communication for the Creation of an Account

Below are the following elements used to create an account through API Gateway in the research (see Figure 3.7).

1. Port Configuration:

- Account Management Service : Port 8081
- API Gateway : Port 8080
- Client Application : Port 3000

2. Scenario:

The client application wants to create a new bank account using the controller called /createAccount API of the **Account Management Service** .

3. Communication Steps:

Step 1: Sending the Request by the Client

- The user fills out a form in the client application (port 3000) to create a new account.
- The client application sends an HTTP POST request to the API Gateway at the following URL:

  POST http://localhost:8080/ACCOUNT-MANAGEMENT/createAccount

- The body of the request contains the client's information (e.g. name, email, account type, etc.):

  {
  "name": "DJAM Xaveria",
  "email": "xaviera.kimbi@facsciences-uy1.cm",
  "accountType": "savings"
  }

Step 2: Processing by API Gateway

- The API Gateway receives the request on port 8080.
- It consults the registration server to obtain information about the **Account Management Service** , including its address and port (port 8081).
- The API Gateway redirects the request to the **Account Management Service** using the following URL:

  POST http://localhost:8081/createAccount

Step 3: Processing by the Account Management Service

- The **Account Management Service** receives the request on port 8081 and processes the account creation.
- It validates the information provided and saves the new account in its database.
- Once the account is successfully created, the **Account Management Service** returns a response in JSON format to the API Gateway, for example:
  {
  "status": "success",
  "message": "Account created successfully",
  "accountId": "123456789"
  }

Step 4: API Gateway Response to Client

- The API Gateway receives the response from the **Account Management Service** and sends it back to the client application on port 3000.

- The client application can then display a success message to the user, for example: "Account created successfully with ID: 123456789"

In this example, the communication between the client and the **Account Management Service** is done indirectly via the API Gateway. This allows for centralized management of requests and responses, thus promoting scalability and decoupling of services. The architecture also facilitates the management of the various interactions between components, while ensuring the security and reliability of operations.

Example of Communication with Events

Below are the following elements used in message broker for the communication with Events:

(A) Port configuration:

- Account Management Service : Port 8081
- API Gateway : Port 8080
- Customer Management Service : Port 8082
- Message Broker : Port 5672

(B) Scenario:

After a bank account is created, the **Account Management Service** publishes an event to a message broker, and the **Customer Management Service** consumes it to record the customer's information.

(C) Communication Steps:

Step 1: Account Creation

- As before, the user fills out a form in the client application and sends a request to create an account.

- The **Account Management Service** processes the request and creates the account.

Step 2: Publish the Event

- Once the account is successfully created, the **Account Management Service** publishes an event to the broker message with the client's information:

  {
  "eventType": "AccountCreated",
  "data": {
  "name": "DJAM Xaveria",
  "email": "xaviera.kimbi@facsciences-uy1.cm",

"accountId": "123456789"
}
}

- The event is sent to a specific queue or topic on the message broker (e.g. account-events).

Step 3: Consumption of the Event by the Customer Management Service

- The **Customer Management Service** is configurationd to listen for events on the message broker.
- AccountCreated event , it extracts the customer data.
- The service processes the customer's registration in its database, using the information received.

Step 4: Confirmation of Registration

- Once the customer is successfully registered, the **Customer Management Service** can publish a confirmation event (optional) on the message broker:

{
"eventType": "CustomerRegistered",
"data": {
"name": "DJAM Xaveria",
"email": "xaviera.kimbi@facsciences-uy1.cm",
"customerId": "987654321"
}
}

In this process, after the account is created, the **Account Management Service** uses a message broker to publish an event. The **Customer Management Service** consumes this event to register the customer, which enables a reactive and decoupled architecture. This also promotes scalability, as each service can evolve independently while staying synchronized via events.

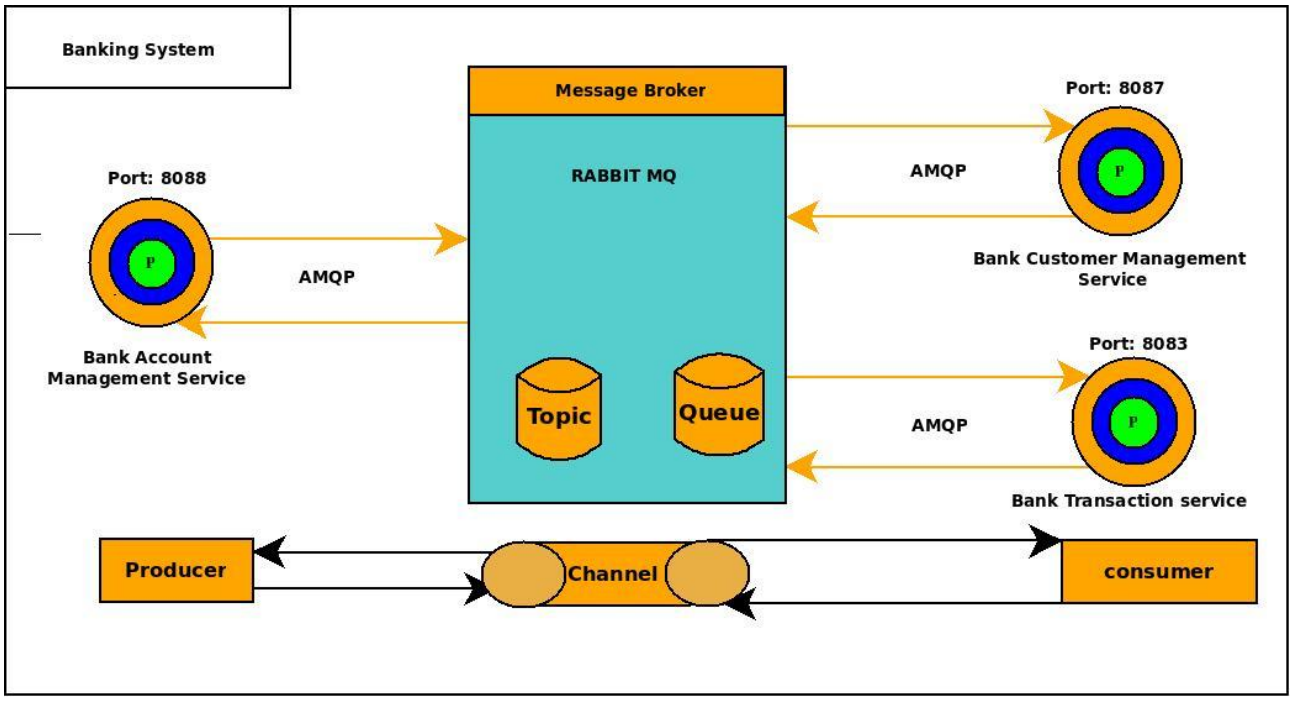

Figure3.8: Communication with Events

3.5.2 Scalability in Microservice Architecture

Scalability is the ability of a system to handle an increase in workload by adding resources, without compromising performance. In this microservice architecture, scalability is essential to meet increasing user demand while maintaining optimal performance (see Figure 3.9): **.**

Horizontal Scalability: This involves adding more instances of a microservice. For example, if a microservice is under heavy load, deploy multiple instances of that service can be deployed to distribute the load (see Figure 3.9).

Vertical Scalability: This involves increasing the resources of a single instance (e.g. adding more CPU or RAM). However, this method has its limitations and is often less flexible than horizontal scaling.

Scalability Scenario with example from the Banking System:

Given a microservice called **User Service** that needs to handle an increase in the number of requests due to the growth of the application. These service instances register after their startup in the registration service as shown in the Figure 3.9

Multiple instances of the **User Service** are deployed, for example, with referenced to two instances of **User Service** , each running on different ports or in separate containers (see Figure3.9):

- Instance 1: http://localhost:8084
- Instance 2: http://localhost:8084

Load Balancing to Show Scalability:

- To handle incoming traffic, a load balancing system is used. This system distributes client requests between different **User Service instances** .
- The API Gateway acts as a load balancer, redirecting requests to any of the available instances.

Sample request formats such as **POST, GET, DELETE or PUT** were used for request control API (Application Programming Interface) as seen in the following example:

Step 1: Customer Request

When a client sends a request to the API Gateway (for example, to add a user), the API Gateway receives the request on a specific port (for example, port 8084).

POST http://localhost:8084/USER-SERVICE/addUser

Step 2: Distribute Requests

- API Gateway uses the load balancing AGA to determine which **User Service instance** should handle the request.
- It can choose, for example, the first available instance (8081) for the first request, then move to the second (8082) for the next.

Step 3: Processing the Request

- The selected instance of the **User Service** processes the request and returns the response to the API Gateway.
- The API Gateway then forwards the response to the client.

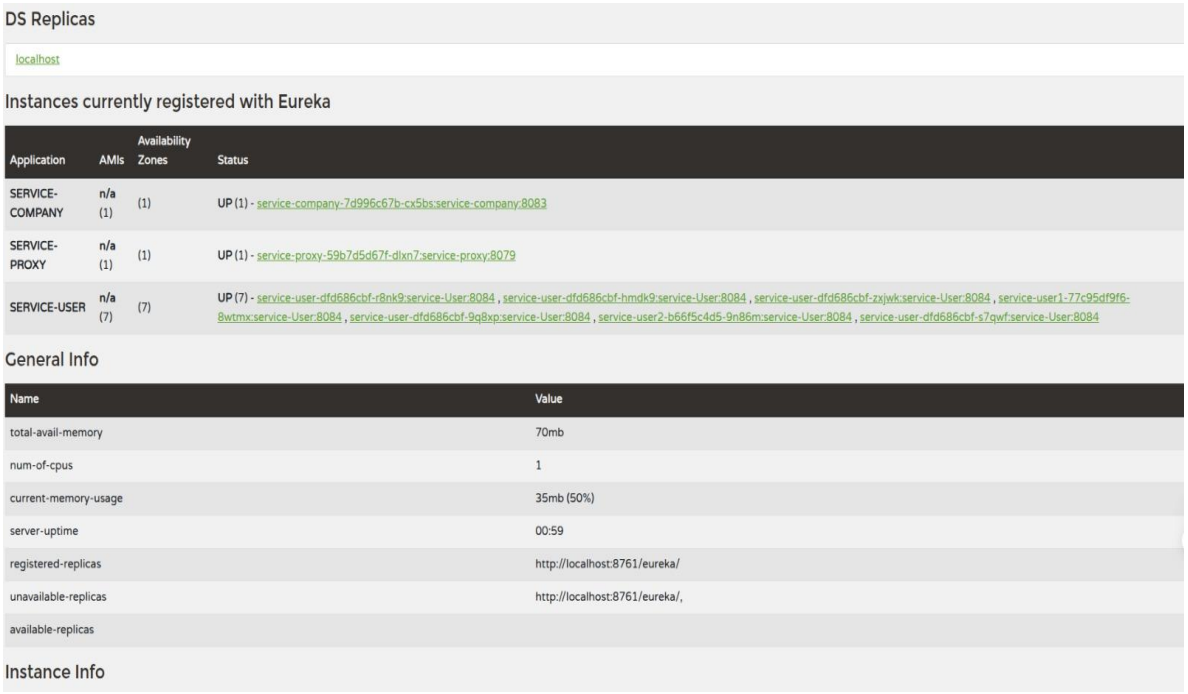

Figure 3.9: Scalability in Microservices Architecture

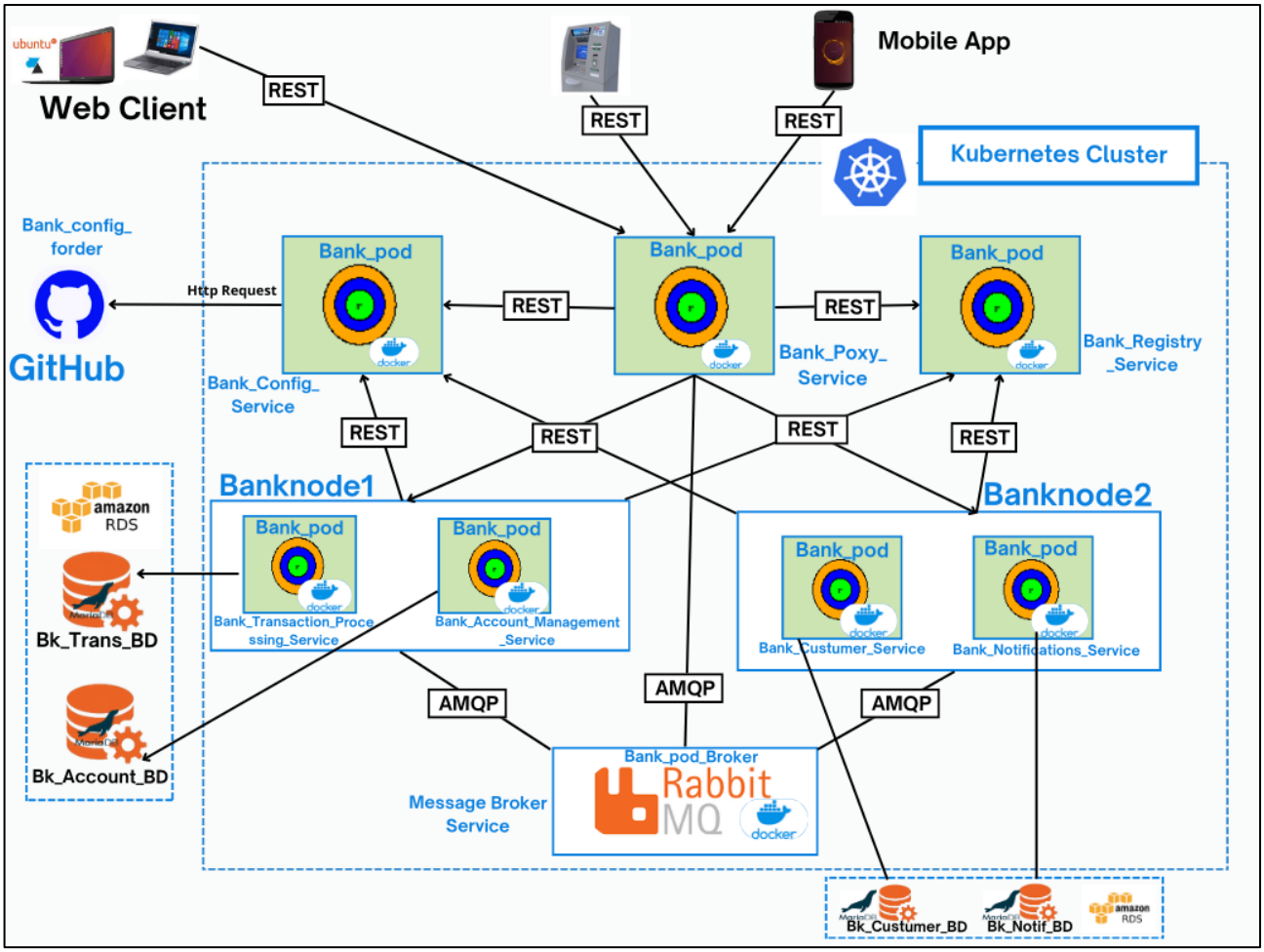

Figure 3.10: Deployment architecture of the banking system and orchestration in Cameroon context

Figure 3.10 illustrates the architecture of the banking system deployed on the Kubernetes Cluster. It shows the different components and their interactions via Rest and Advanced Message Queuing Protocol(AMQP) connection.

The key components are:

- Web Client: The web user of bank interface that interacts with the banking system.
- GitHub: Contains the services configuration files, accessible via a Rest service
- Kubernetes Cluster: The containerization infrastructure that host the various banking services.
- Bank_Pod: The different banking microservices deployed in the Kubernetes Cluster, communication via Rest APIs
- Banknode1 and Banknode2: The main nodes of the banking system,hosting Bank account management service ,Bank transaction service ,Bank customer service and Bank notification service.
- Bk_Trans_BD and Bk_Account_BD: The databases for bank transactions and bank account respectively
- Message broker service: The asynchronous messaging service used for communication between the different system components
- Bank_registry_service: A central service for registering and discovering other services
- Mobile App: The mobile application that interacts with the banking system via REST APIs.

4. RESULTS AND DISCUSSIONS

4.1. Environment Setup and Configuration

The development environment was properly set up to support the research objectives. This includes ensuring the availability of essential tools such as Visual Studio Code, Eclipse, Java Development Kit(JDK), Apache Maven , Docker Desktop, Rester, Hey, RabbitMQ, Spring Boot, Actuator, Spring Cloud Gateway, Spring Cloud Config, Eureka, H2, and MariaDB. The Hey tool was used for request tracing among microservices.
For the setup of the microservice banking system, the following parameter were set for AGA and the corresponding configurations were made (see Table 4.1):

Table 4.1: Parameter Setting for AGA

Parameter	Value
Size of population	20000
Selection operator	20
Crossover operator	Single-point

Crossover probability	1
Mutation operator	Bit-wise
Mutation probability	1/95

Configuration Folder: The microservice design for the banking system was configured in the following git hub address: "https://github.com/KIMBIXY/cloud-conf"

Bank Configuration Service:
Figure4.1 shows the Kubernetes deployment YAML configuration for the configuration service, responsible for centralizing the micro services configuration. The name space in which the service is deployed is "micro services". The deployment is named "service-configuration" and uses the Docker image "paulzk/seminar-service-configuration:latest". It exposes port 8080 of the container and defines two environment variables: "SPRING_APPLICATION_NAME" with the value "service-configuration" and "SPRING_CLOUD_CONFIG_SERVER_GIT_URI" with the value "https://github.com/KIMBIXY/cloud-conf", which points to the Git repository containing the micro services configuration. A "Load Balancer" type service is also defined to expose the micro services configuration service on port 8080, with a selector targeting pods with the "app: service-configuration" label

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: microservices
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-config
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-config
  template:
    metadata:
      labels:
        app: service-config
    spec:
      containers:
        - name: service-config
          image: paulzk/seminar-service-config:latest
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-config"
            - name: SPRING_CLOUD_CONFIG_SERVER_GIT_URI
              value: "https://github.com/KIMBIXY/cloud-conf"
---
apiVersion: v1
kind: Service
metadata:
  name: service-config
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8080
      targetPort: 8080
  selector:
    app: service-config
---
```

Figure 4.1: Kubernetes configuration for the bank configuration service

Bank Registry Service
The Kubernetes configuration for the bank registry service deploys a deployment in the "microservices" name space named "service-register", using the Docker image "paulzk/seminar-bank-registry:latest" and exposing port 8761; it defines three important environment variables to identify the Spring application, point to the centralized configuration service, and import the configuration from the configuration server, and a "Load Balancer" type service is also defined to expose the bank registry service on port 8761 by targeting pods with the "app: service-register" label, thus enabling the deployment and integration of the bank registry service into the microservice architecture by connecting it to the centralized configuration service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-register
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-register
  template:
    metadata:
      labels:
        app: service-register
    spec:
      containers:
        - name: service-register
          image: paulzk/seminar-service-register:latest
          ports:
            - containerPort: 8761
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-register"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: service-register
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8761
      targetPort: 8761
  selector:
    app: service-register
---

```

Figure4.2: Kubernetes configuration for the registry service

Bank User Service

In Figure 4.3, the Kubernetes configuration for the bank user service deploys a deployment in the "microservices" name space named "service-user", using the Docker image "paulzk/seminar-service-user:latest" and exposing port 8084; it defines three important environment variables to identify the Spring application, point to the centralized configuration service, and import the configuration from the configuration server, and a "Load Balancer" type service is also defined to expose the bank user service on port 8181 by targeting pods with the "app: service-user" label, thus enabling the deployment and integration of the bank user service into the microservice architecture by connecting it to the centralized configuration service. The initial number of replicas is set to 1, and Kubernetes can be configured to scale the number of replicas up or down based on the traffic.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-user
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-user
  template:
    metadata:
      labels:
        app: service-user
    spec:
      containers:
        - name: service-user
          image: paulzk/seminar-service-user:latest
          ports:
            - containerPort: 8084
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-User"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: service-user
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8181
      targetPort: 8084
  selector:
    app: service-user
---
```

Figure4.3: Kubernetes Configuration for the user service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-user1
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-user1
  template:
    metadata:
      labels:
        app: service-user1
    spec:
      containers:
        - name: service-user1
          image: paulzk/seminar-service-user1:latest
          ports:
            - containerPort: 8084
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-User"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: service-user1
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8182
      targetPort: 8084
  selector:
    app: service-user1

```

Figure4.4: Kubernetes configuration for the another instances of user service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-user2
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-user2
  template:
    metadata:
      labels:
        app: service-user2
    spec:
      containers:
        - name: service-user2
          image: paulzk/seminar-service-user2:latest
          ports:
            - containerPort: 8084
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-User"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: service-user2
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8183
      targetPort: 8084
  selector:
    app: service-user2
---

```

Figure4.5: Kubernetes configuration for the another instances of user service

Bank Proxy Service

The Kubernetes configuration for the service proxy deploys a deployment in the "microservices" name space named "service-

proxy", using the Docker image "paulzk/seminar-service-proxy:latest" and exposing port 8079; it defines three important environment variables to identify the Spring application, point to the centralized configuration service, and import the configuration from the configuration server, and a "Load Balancer" type service is also defined to expose the service proxy on port 8060 by targeting pods with the "app: service-proxy" label, thus enabling the deployment and integration of the service proxy into the microservice architecture by connecting it to the centralized configuration service(See Figure 4.6)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-proxy
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-proxy
  template:
    metadata:
      labels:
        app: service-proxy
    spec:
      containers:
        - name: service-proxy
          image: paulzk/seminar-service-proxy:latest
          ports:
            - containerPort: 8079
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-proxy"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
---
apiVersion: v1
kind: Service
metadata:
  name: service-proxy
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - port: 8060
      targetPort: 8079
  selector:
    app: service-proxy
---
```

Figure4.6: Kubernetes Configuration for the Another Instances of User Service

Message Broker Service

Figure 4.7 shows the configuration of the RabbitMQ message broker service in the Kubernetes cluster. It comprises two main resources: a Deployment and a Service.

The RabbitMQ Deployment creates a pod with a single container. This container uses the "rabbitmq:management" image, which provides both the RabbitMQ server and the management interface. The container exposes two ports: port 5672 for standard RabbitMQ traffic and port 15672 for the management interface.

The Load Balancer Service exposes these two ports to the outside world, allowing clients to access the RabbitMQ broker and its management interface. This configuration enables the simple and standardized deployment and exposure of the message broker service within the Kubernetes cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
        - name: rabbitmq
          image: rabbitmq:management
          ports:
            - containerPort: 5672
            - containerPort: 15672
---
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq
  namespace: microservices
spec:
  type: LoadBalancer
  ports:
    - name: rabbitmq
      port: 5672
      targetPort: 5672
    - name: rabbitmq-management
      port: 15672
      targetPort: 15672
  selector:
    app: rabbitmq
```

Figure4.7: Kubernetes configuration for the message broker service

Automatic Scalability of Bank Service User

Figure 4.8 shows the Kubernetes configuration of the Deployment for the banking user service with auto scaling parameters. The Deployment creates a pod with a single container using the "paulzk/seminar-service-user:latest" image. The container exposes port 8084 for the service traffic. The "resources" section defines the CPU and memory resource limits for the container, with a request of 20m CPU and 512Mi memory, and limits of 50m CPU and 300Mi memory. This configuration enables automatic scalability of the user service based on the workload, dynamically adjusting the number of replicas and allocated resources to ensure optimal performance and high availability of the banking user service.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: service-user
  namespace: microservices
spec:
  replicas: 1
  selector:
    matchLabels:
      app: service-user
  template:
    metadata:
      labels:
        app: service-user
    spec:
      containers:
        - name: service-user
          image: paulzk/seminar-service-user:latest
          ports:
            - containerPort: 8084
          env:
            - name: SPRING_APPLICATION_NAME
              value: "service-User"
            - name: SPRING_CLOUD_CONFIG_URI
              value: "http://service-config:8080"
            - name: SPRING_CONFIG_IMPORT
              value: "configserver:http://service-config:8080"
          resources:
            requests:
              cpu: "20m"
              memory: "512Mi"
            limits:
              cpu: "50m"
              memory: "300Mi"

```

Figure4.8: Kubernetes configuration for the automatic scalability of user service

4.2 Results of Microservices Deployed on Kubernetes Cluster

4.2.1 Synchronization between Instances of the Same Microservice

Given that each instance of a microservice is running and has its own database, replication between database was done through Message broker to solve the synchronization problem (See Figure 3.8)
RabbitMQ is a messaging system that uses the publish/subscribe (pubsub) model to enable asynchronous communication between microservices.

Orchestration and Containerization

Docker, Docker-Compose and Kubernetes were used for the management of containerization and orchestration of microservices.

Containerization

With Docker, services were put in an isolated environment that operates independently of other services. The use of Docker Compose allowed orchestration in development mode, but it is not suitable for production mode. All configurations are present in the reserach GitHub repository:

https://github.com/KIMBIXY/MicroserviceBankingSystem/blob/main/docker-compose.yml

Orchestration

Orchestration with Kubernetes has allowed us to manage all services in production. It enables scalability of the user service to seven instances. The number of replicas of a service can increase in the case of increased network traffic or decrease in the case of reduced network traffic. The following Figures show the deployments and services launched on Kubernetes used on Docker Desktop. All configurations are present in the research GitHub repository: https://github.com/KIMBIXY/MicroserviceBankingSystem/blob/main/kubernetes.yaml

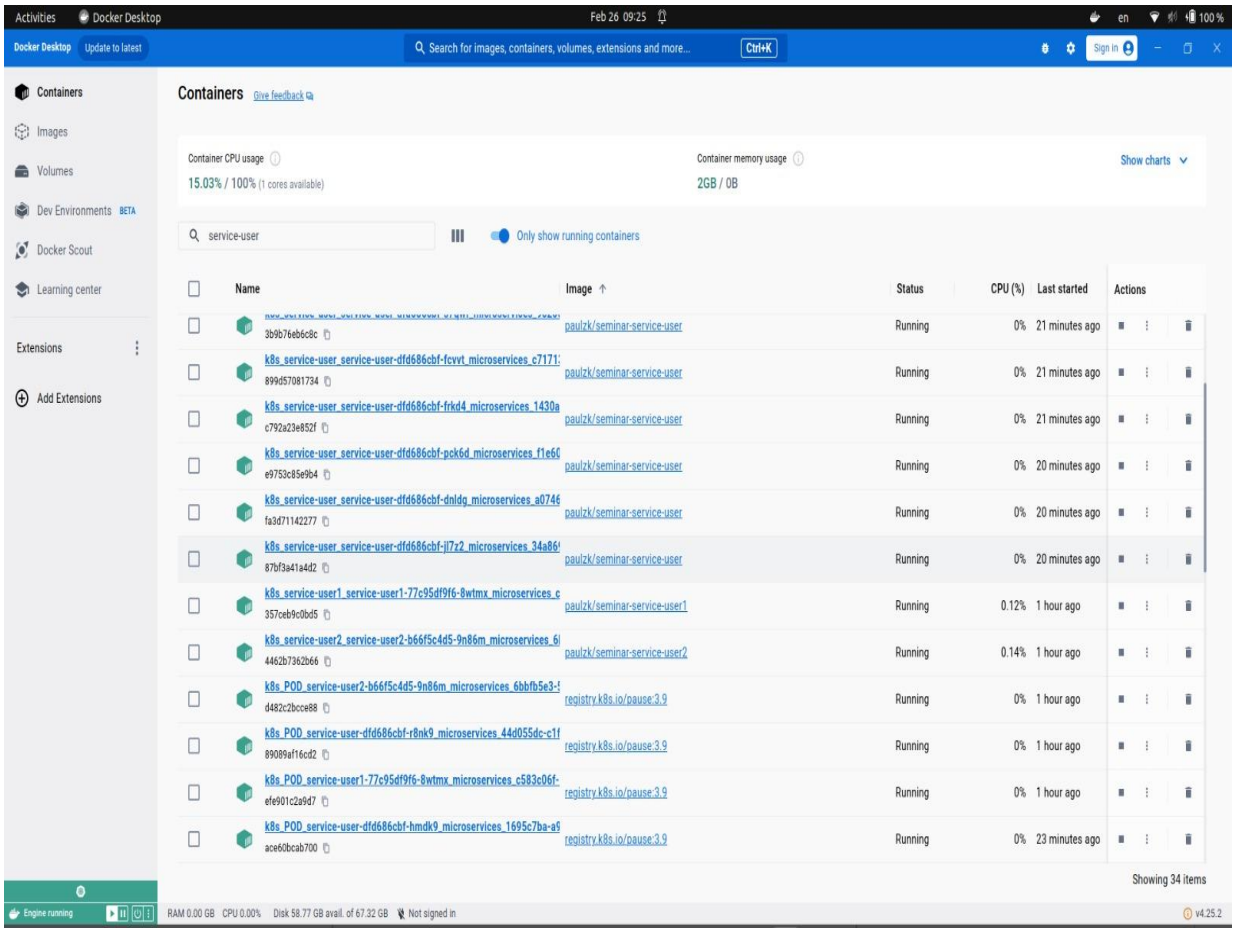

Figure4.9: Services launched on Kubernetes used on Docker Desktop

Figure4.10 depicts all the running microservices created and deployed on Kubernetes Cluster for the banking application. All the functional and non-functional services in Figure3.1 were deployed on Kubernetes Cluster on the research github address: https://github.com/KIMBIXY/MicroserviceBankingSystem

Figure 4.10 shows the Kubernetes cluster deploying the application composed of all the interconnected microservices, with pods, services, deployments, replica sets and nodes configured to ensure the operation of the application. It also shows the scalability of the user service to 8 instances, and the company service configuration for automatic horizontal scalability. The automatic scalability of the "service-company" service is configured. This can be seen in the "REFERENCE" section where it is indicated "horizontalpodautoscaler.autoscaling/service-company" (See Figure4.11). This means that the Kubernetes Horizontal Pod Autoscaler (HPA) is used to automatically adjust the number of replicas of the "service-company" service based on the workload metrics. The HPA monitors metrics such as CPU or memory utilization, and adjusts the number of pods accordingly to maintain the desired performance.

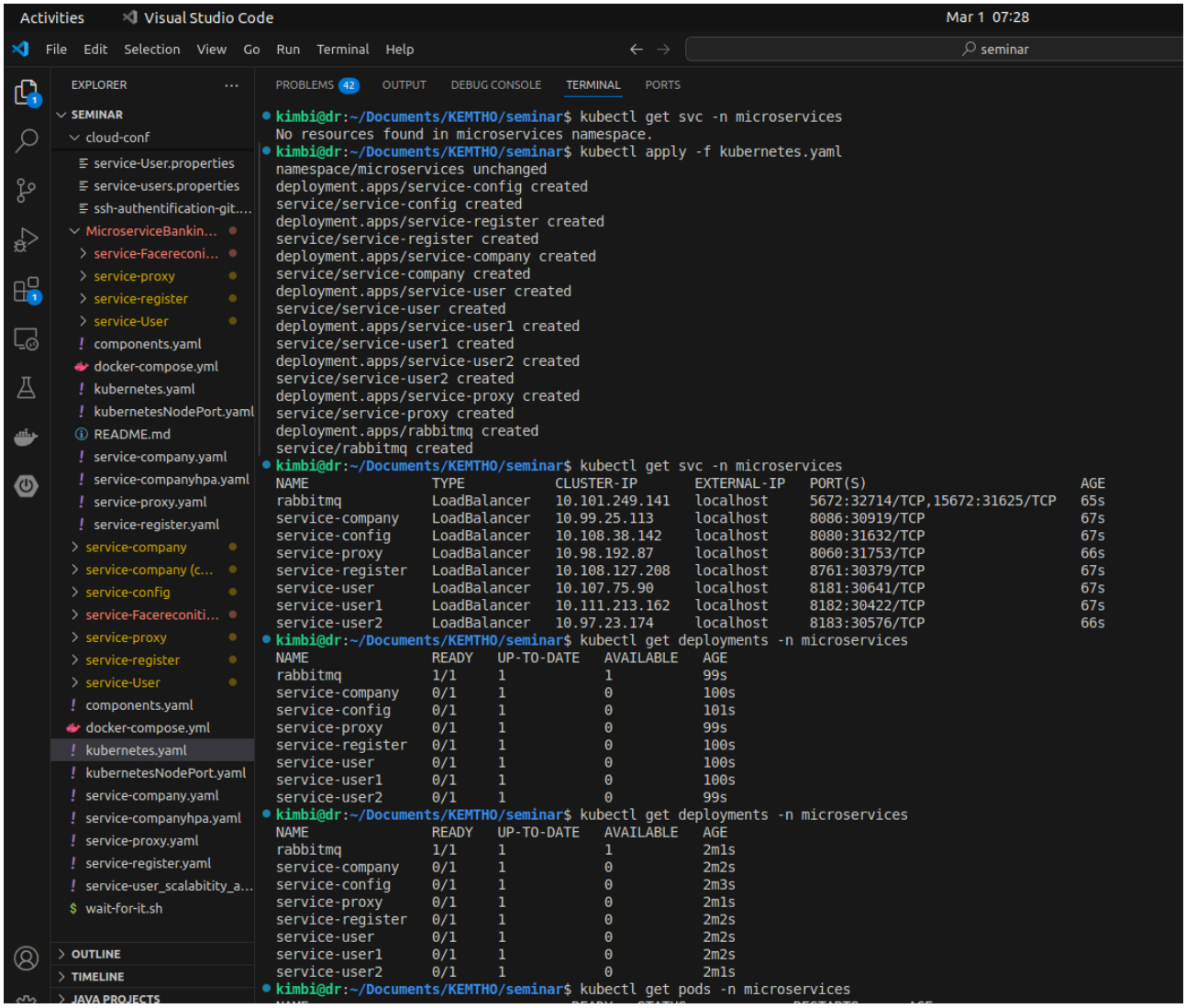

Figure4.10: Microservices Created and Deployed on Kubernetes Cluster

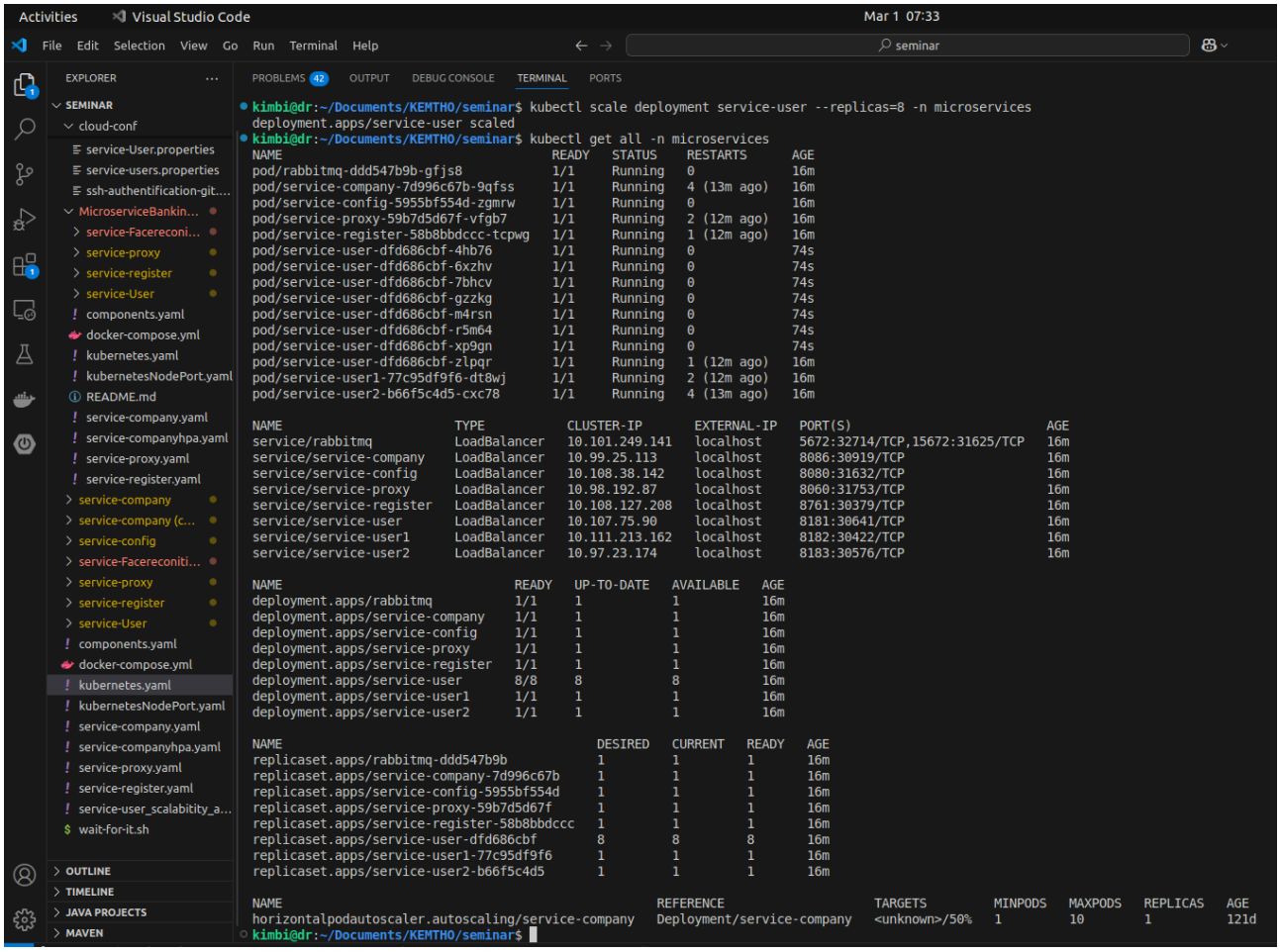

Figure 4.11: Results of Kubernetes Cluster Deployment with Automatic Scalability

4.2.2 Results of Implementing Event Sourcing and CQRS for Transaction Service Management

The Command Query Responsibility Segregation (CQRS) model has been implemented in the transaction service to improve the separation of responsibilities between read and write operations. This allows for optimized performance and better scalability in the management of banking transactions. The implementation of event sourcing gave the possibility to store all transactions in the form of events in order to keep track of the changes in the state of a bank account. An experimental approach was taken on the transaction service to highlight these concepts with some key results.

The key results are:

- Separation of read and write operations through CQRS improved performance and scalability of the transaction service.
- Event Sourcing provided a complete audit trail of all transactions, enabling better tracking of account state changes.
- The experimental implementation demonstrated the benefits of these architectural patterns for the transaction management use case.

Overall, the adoption of event sourcing and CQRS has been a successful strategy for enhancing the capabilities of the transaction service. These patterns have enabled the better manage the complexities of banking transaction processing at a scale. (see Figure 4.12 to Figure 4.18)
See git hub link for the corresponding results. https://github.com/KIMBIXY/MicroserviceBankingSystem/tree/main/transactionserviceapplication

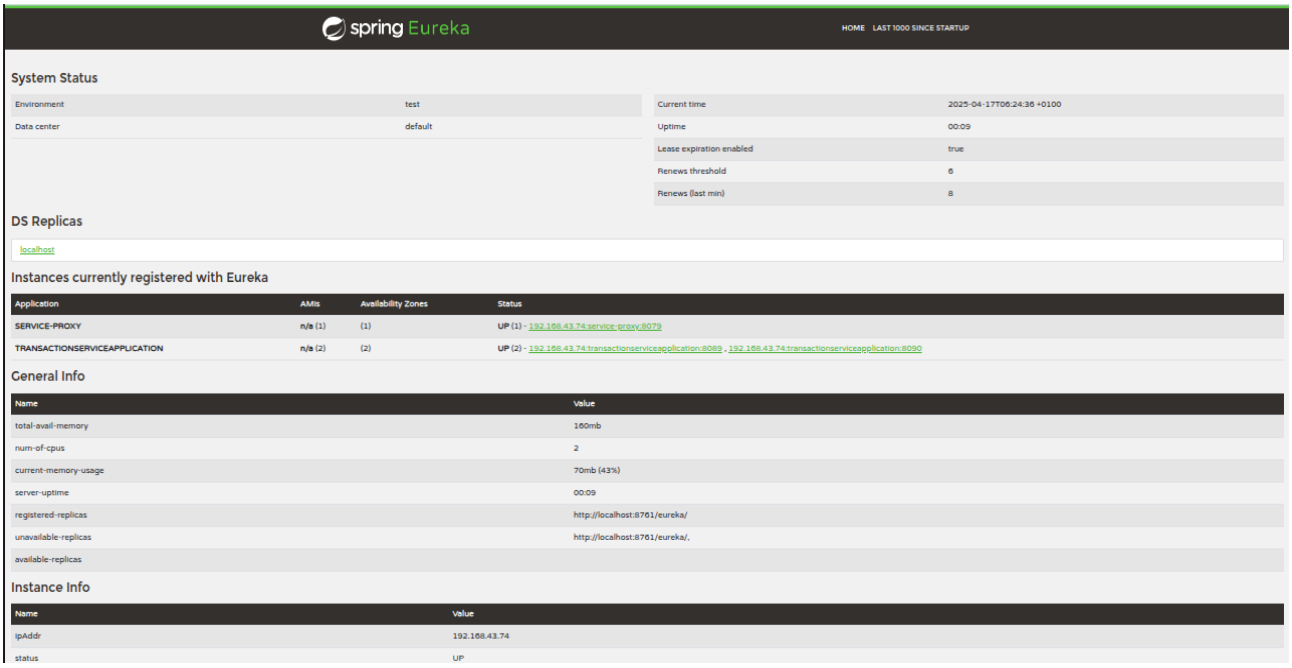

Figure 4.12: Transaction Recording in the Registration Service

When sending a request to make a deposit to a customer's account, the following result is obtained (Figure 4:13).

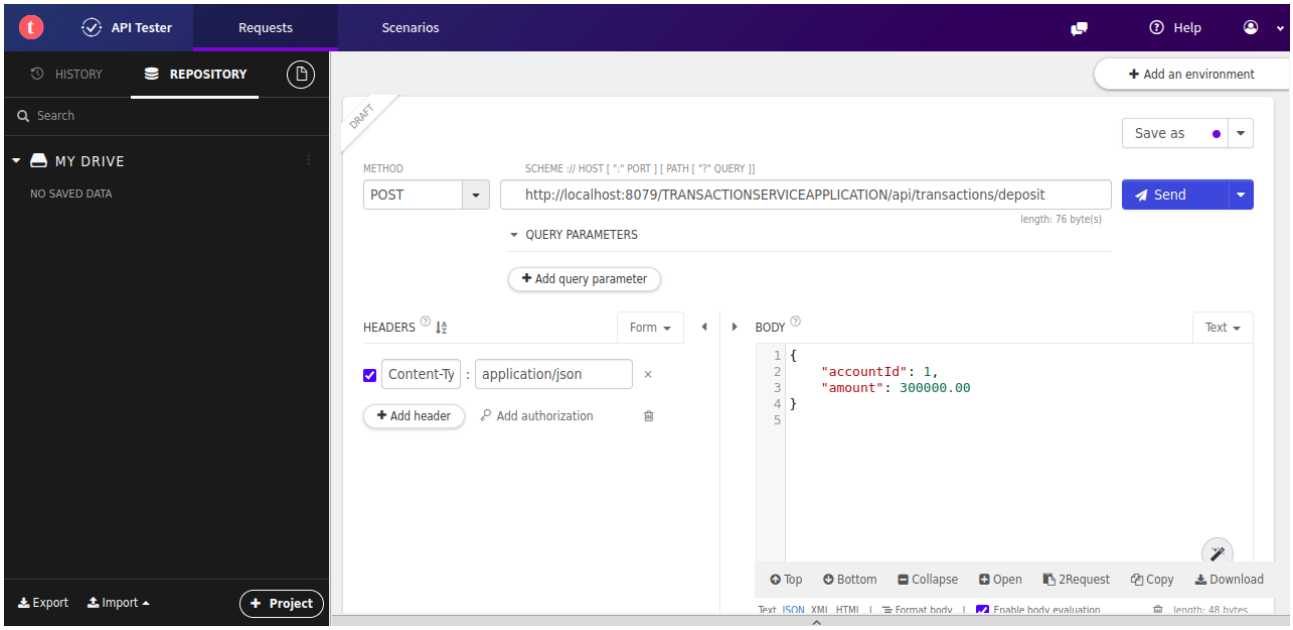

Figure 4.13 Depositing Money to an Account from the Proxy Service

Money is withdrawn from an account according to the transaction presented in Figure 4.14.

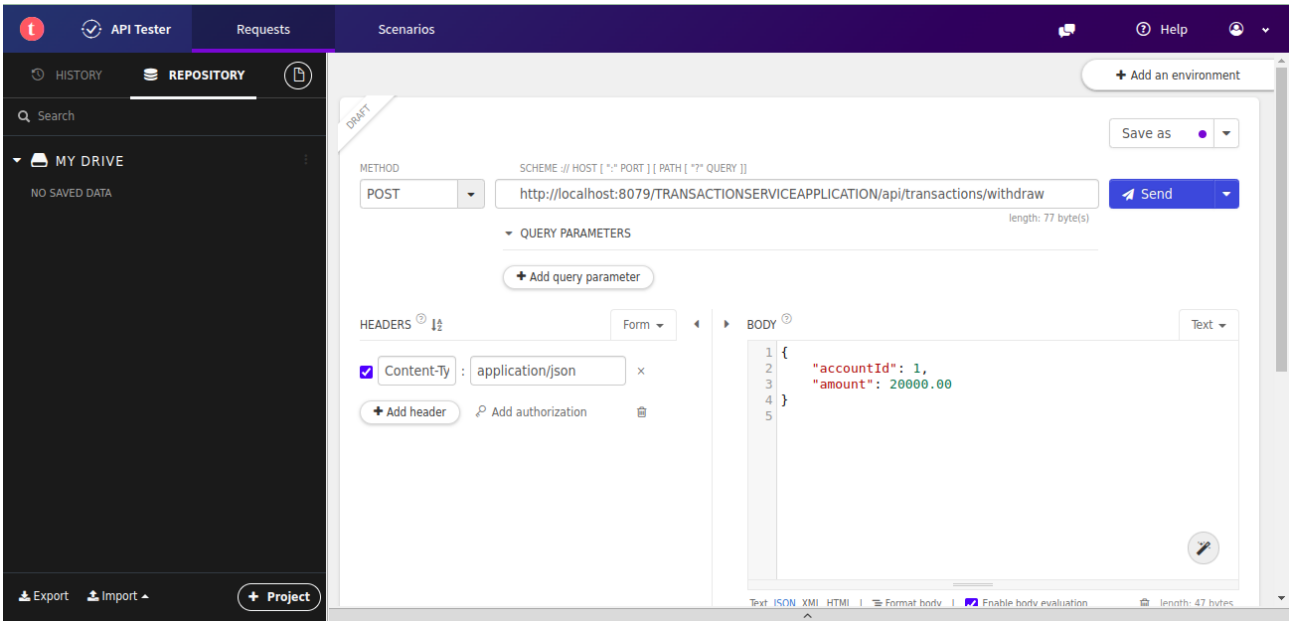

Figure 4.14 Withdrawing Money from an Account through the Proxy Service

The sequence of events performed on an account is captured according to Figure 4.15.

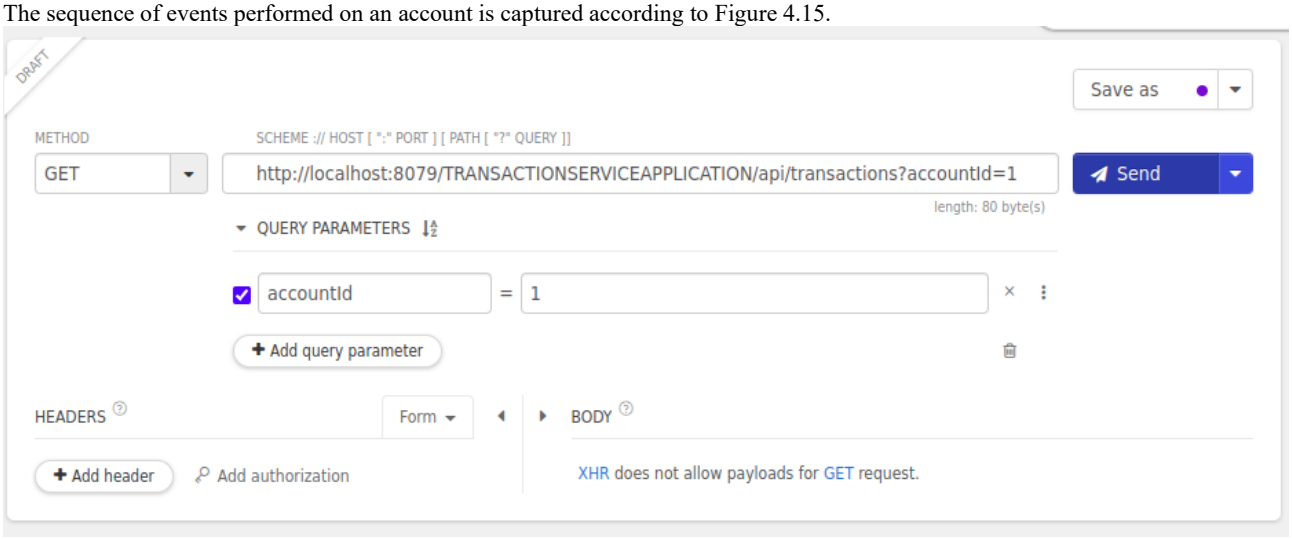

Figure 4.15 Retrieving All Events for an Account

The results of retrieving the transactional events performed in the system are presented (see Figure 4.15 and 4.16). However, the current account balance is not yet known. The events are stored, and to get the state of an account at a given time, preceding events was reconstituted up to that time. This concept allows banks to have the state of each account at any given moment.

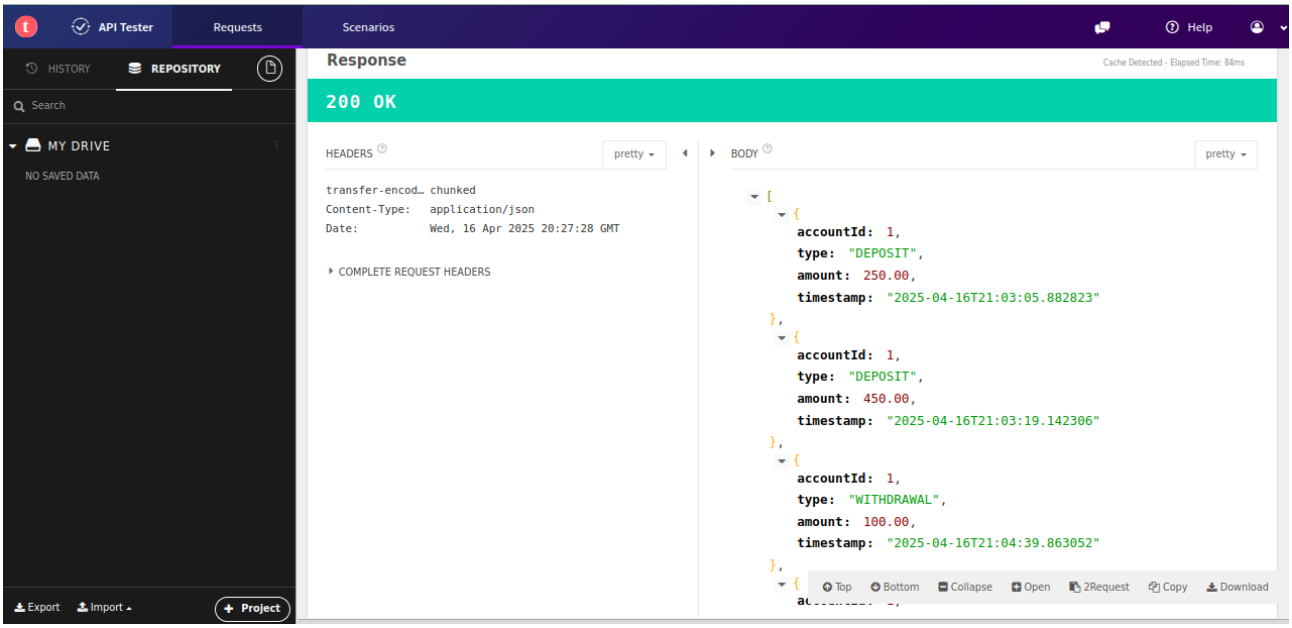

Figure 4.16 Result of Retrieving All Events for the Account

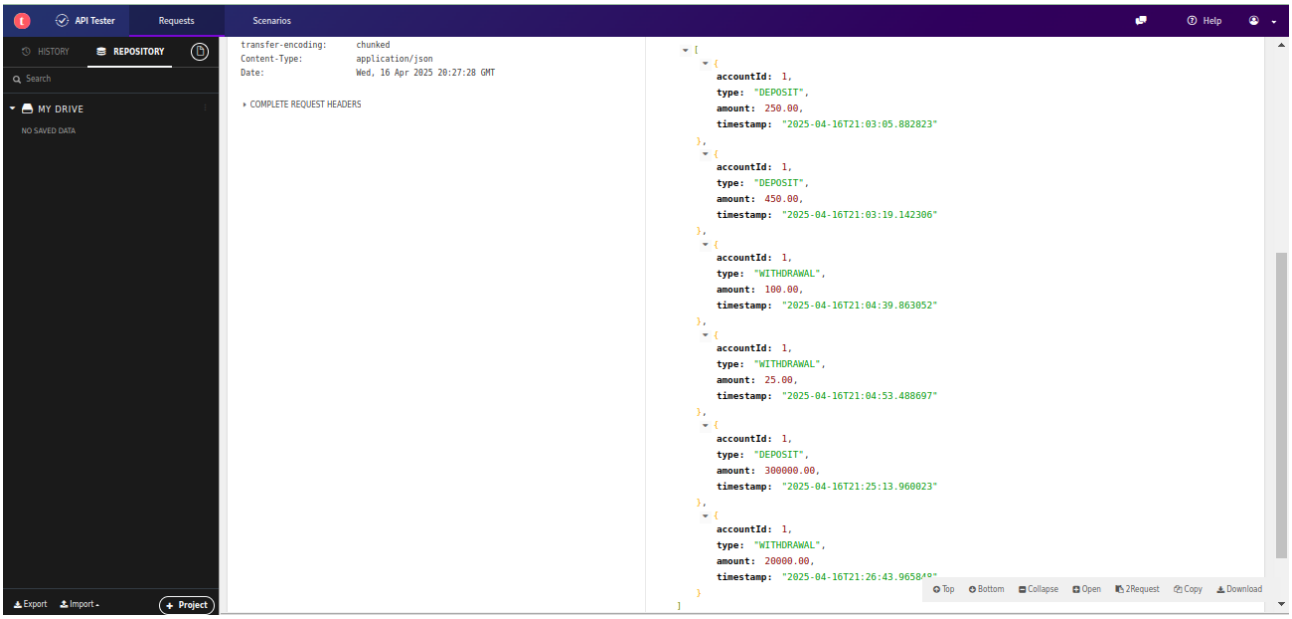

Figure 4.17 Result of Retrieving All Events for the Account

To know the balance of an account, reconstitution of the events was done (see Figure 4.17)

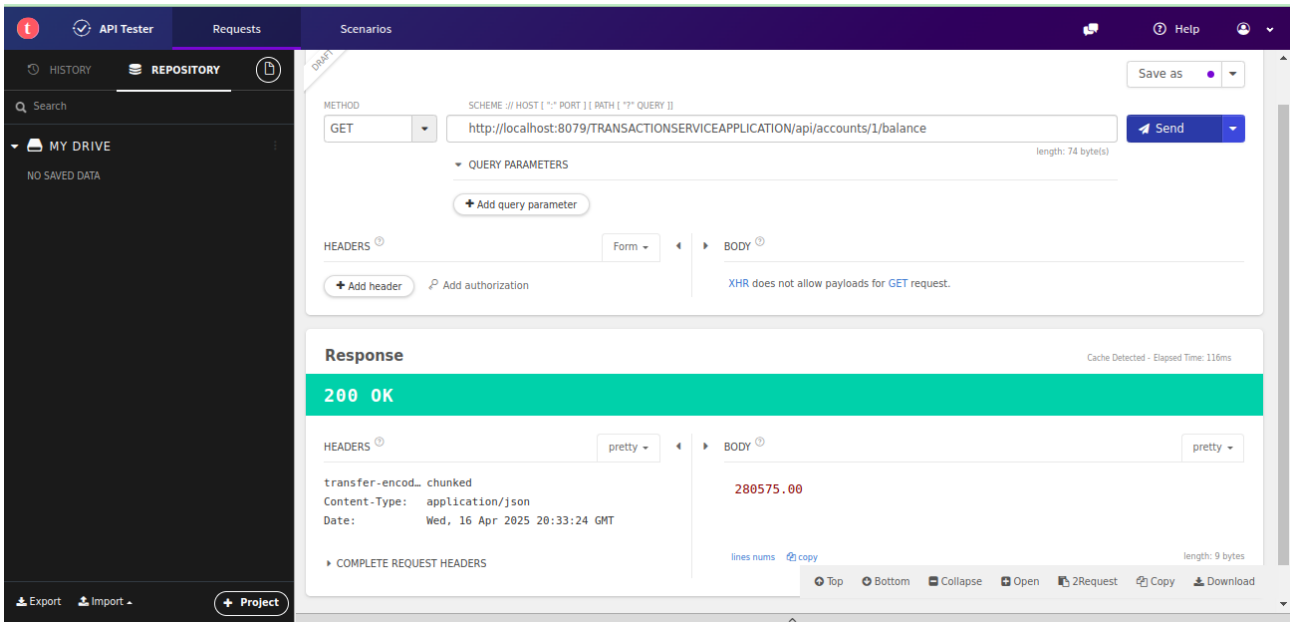

Figure 4.18: Obtaining the Account Balance by Reconstituting the Previous Events

4.3 Performance Testing Analysis

Performance testing of the microservice design was performed with Hey tool for tracing traffic management of user request across multiple microservices where the use case being tested is the retrieval of the list of users. The test involves sending 2,000 requests in two batches of 1,000 requests each and distributing them among 7 instances of the bank user service. Each 1,000 requests were distributed in batches of 100, the traffic was rapidly increased and the performance is as shown in Figure 4.19.

To evaluate the effectiveness of the developed load balancing algorithm, several performance evaluation criteria were taken into consideration:

Response Time : The amount of time it takes to process a request. A lower response time indicates better performance. In response time criterion the following results were established:

Total response time: 0.6156 seconds
Slowest response time: 0.2388 seconds
Fastest response time: 0.0056 seconds
Average response time: 0.0545 seconds

Throughput : Number of requests processed per unit of time. High throughput is crucial for applications that require high processing capacity. The performance test the banking system executed 1624.5330 requests per second

Resource Utilization : A measure of how efficiently resources are used. A good balance between resource utilization and performance is essential. The results indicate generally satisfactory performance,with a relatively short average response time and high throughput. However, the slowest response time raises areas for improvement.

Latency distribution: The histogram shows that the majority of requests 406 have a low response time between 0.052 and 0.076 seconds. This confirms the good overall performance of the system.

The latency distribution indicates that 50% of requests have latency less than 0.0499 seconds, 90% have latency less than 0.0999 seconds, and 99% have latency less than 0.1999 seconds. This latency distribution is consistent with the observed response times.

Scalability : The ability of the algorithm to handle an increase in workload without compromising performance. Analysis of the various steps of the process (DNS resolution, writing the query, waiting for the response, reading the response) did not reveal any major bottlenecks.

These performance results are very encouraging for this research. They demonstrate that the tested system is capable of handling a significant workload while maintaining acceptable response times.

Overall, these performance test results provide valuable information for further research. They allow us to better understand the system's capabilities and to direct improvement efforts in a more targeted manner.

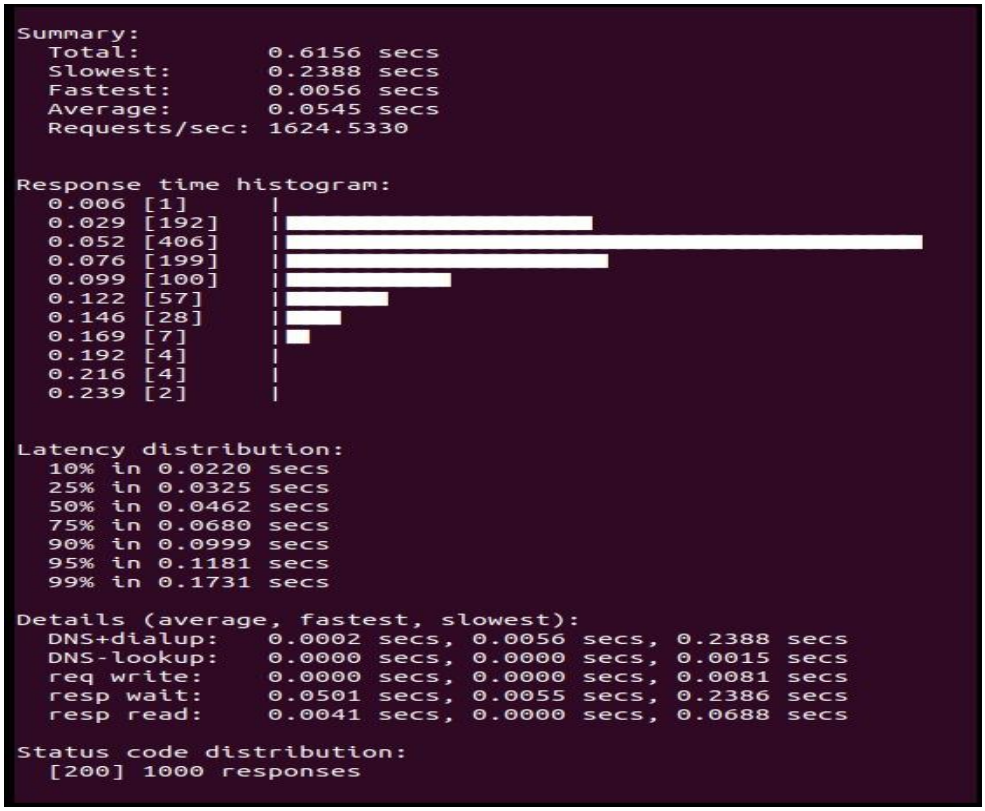

Figure 4.19: Performance Testing Analysis of the Microservice Design

5. CONCLUSION AND PERSPECTIVES

Microservice architecture represents a significant step forward in the development of modern banking solutions, meeting the increasing demands for scalability, security, efficiency and flexibility in a rapidly digitalizing environment. This paper presented a robust methodological approach to design a banking services platform, integrating essential elements such as account management, transaction processing, real-time notification.

The challenges associated with microservice architectures, including service communication, data management, and scalability were experimented. The use of modern technologies such as Docker, Docker Compose, Kubernetes, and RabbitMQ were employed to facilitate deployment, orchestration, and asynchronous communication between banking services. Implementing an event-driven architecture improved system responsiveness and resilience, while ensuring an optimal user experience.

The case studies presented demonstrate how a well-designed architecture can efficiently handle user requests, while ensuring regulatory compliance and secure operations. The service-to-service communication examples also illustrate the importance of a single point of entry via an API Gateway, as well as the use of load balancing mechanisms to ensure optimal performance.

In conclusion, this research highlights the importance of a systematic and integrated approach to developing micro services-based banking platforms. As the banking industry continues to evolve, adopting these innovative architectures will enable financial institutions to quickly adapt to customer needs and remain competitive in an ever-changing digital landscape.

As future work, it is worth researching further in to more advanced load balancing and optimization algorithms in Artificial Intelligence in order to manage the problems of scalability.

6. REFERENCES

[1] Sharma, S., et al. (2023). “Designing Scalable Microservices for Banking Applications.” *Journal of Banking Technology* , 12(3), 45-67.

[2] Lewis, J., & Fowler, M. (2014). “Microservices: A Definition of This New Architectural Term.” *MartinFowler.com* .

[3] Sharma, S., Bhavisha, Rupinder Singh, & Jaskaran Singh. (2023). “Analyzing Load Balancing Techniques for Cloud Computing: Pros, Cons, and Emerging Trends.” *5th International Conference on Communication and Information Processing (ICCIP-2023)* . Available on: SSRN. (SSRN is an open-access online preprint community, owned by Elsevier).

[4] Maestro, A., & Surantha, N. (2024). “Scalability Evaluation of Microservices Architecture for Banking Systems in Public Cloud.” In *Proceedings of the International Conference on P2P, Parallel, Grid, Cloud and Internet Computing* .

[5] Devi, D. and Uthariaraj, V. R. (2016). “Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks,” in The Scientific World Journal), vol. 6, pp. 214–229.

[6] Abishek S. S.(2022) “Dynamic Load Balancing of Microservices in Kubernetes Clusters using Service Mesh”. MA thesis. Dublin, National College of Ireland.

[7] Alexander S. (2019). “A study on load balancing within microservices architecture”. MA thesis. Halmstad University.

[8] Espejo, R. and Reyes, A. 2019. Organizational Systems: Managing Complexity with the Viable System Model. Springer Berlin Heidelberg.

[9] Gucer, V., Narain, S. and others. 2022. Creating Applications in Bluemix Using the Microservices Approach. IBM Redbooks.

[10] Gysel, M., Kölbener, L., Giersche, W. and Zimmermann, O. 2024. Service Cutter: A Systematic Approach to Service Decomposition. European Conference on Service-Oriented and Cloud Computing, p. 185–200.

[11] Levcovitz, A., Terra, R. and Valente, M. T. 2016. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems.

[12] Malavalli, D. and Sathappan, S., "Scalable Microservice-based Architecture for Enabling DMTF Profiles," in 2015 11th International Conference on Network and Service Management (CNSM), 2015, pp. 428-432: IEEE.

[13] Palihawadana, S., Wijeweera, C., Sanjitha, M., Liyanage, V., Perera, I., and Meedeniya, D., "Tool Support for Traceability Management of Software Artefacts with DevOps practices," in 2017 Moratuwa Engineering Research Conference (MERCon), 2017, pp. 129-134: IEEE.

[14] Sun, Y., Nanda, S., and Jaeger, T., "Security-as-a-Service for Microservices-based Cloud Applications," in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), 2015, pp. 50-57: IEEE.