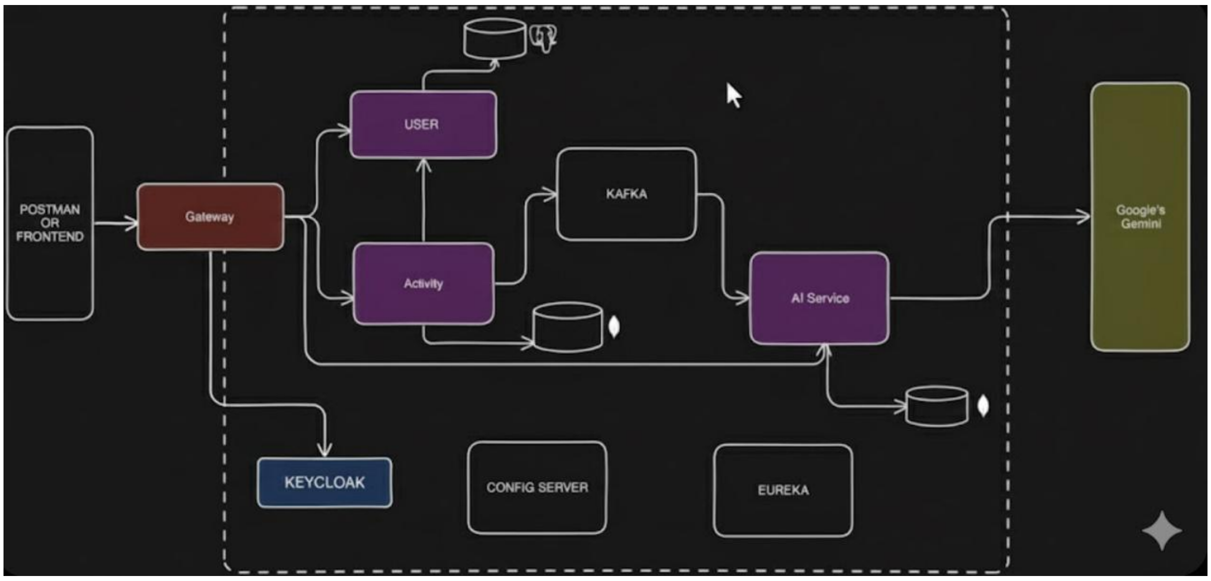
1. Introduction

In today's world, fitness applications are becoming increasingly intelligent and data-driven. Users expect personalized insights, real-time tracking, and AI-powered recommendations. To meet these expectations, our project implements a fully functional **AI-powered Fitness Recommendation System** using **Spring Boot Microservices**, **React**, **Kafka**, and **Google Gemini API**.

The backend of the system is designed using **six independent microservices**, each with a specific responsibility. This architecture improves scalability, security, independent deployment, and fault isolation. The system takes user activity logs (for example: walking, running, cycling), processes them through the AI microservice, and delivers personalized feedback, improvement areas, safety guidelines, and suggested workouts.

This report details the **design, implementation, technologies, and workflows** used in building the backend of this application.

2. System Architecture



The project follows a modern microservices architecture with the following core components:

2.1 List of All Backend Services

1. **User Service** – Manages user accounts, data storage, authentication integration.

2. **Activity Service** – Handles user fitness activities and publishes Kafka messages.
3. **AI Service** – Processes Kafka events and generates AI-driven recommendations.
4. **API Gateway** – Single entry point for all client requests, routing and token validation.
5. **Eureka Server** – Service discovery and registration.
6. **Config Server** – Centralized configuration storage for all services.

The system uses both **synchronous (REST)** and **asynchronous (Kafka)** communication. User → API Gateway → Microservice calls are synchronous. Activity → Kafka → AI Service flow is asynchronous for scalability and performance.

2.2 Architecture Features

- **Loose coupling**: Each service runs independently on different ports.
- **Decentralized data management**:
    - PostgreSQL for User Service
    - MongoDB for Activity & AI Service
- **Event-driven processing** using Kafka.
- **Security** implemented using **Keycloak OAuth2 + JWT**.
- **Automatic service registration** handled by Eureka.
- **Central configuration** managed by Config Server.

This architecture reflects real industry standards used in banking, fitness, e-commerce, and enterprise applications.

3. Technology Stack

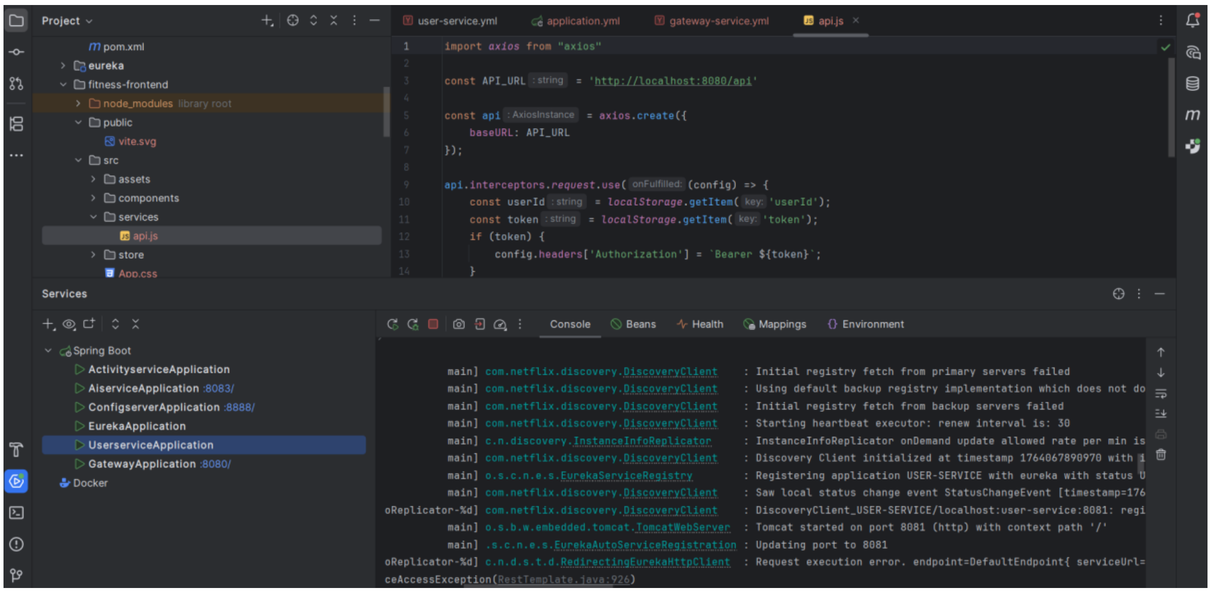
3.1 Backend Technologies

Component	**Technology**
Microservices	Spring Boot
Routing	Spring Cloud Gateway

Authentication	Keycloak OAuth2 / OpenID Connect
Service Discovery	Eureka (Spring Cloud Netflix)
Asynchronous Messaging	Apache Kafka
Databases	PostgreSQL, MongoDB
Build Tool	Maven
AI Integration	Google Gemini API

3.2 Reason for Technology Choices

- **Spring Boot** simplifies microservice creation with built-in support for REST, JPA, and cloud tools.
- **PostgreSQL** is robust for relational data like users.
- **MongoDB** is ideal for flexible activity logs and AI responses.
- **Kafka** enables high-speed asynchronous processing.
- **Keycloak** provides enterprise-level authentication and authorization.
- **Gemini API** allows free AI model usage for generating recommendations.



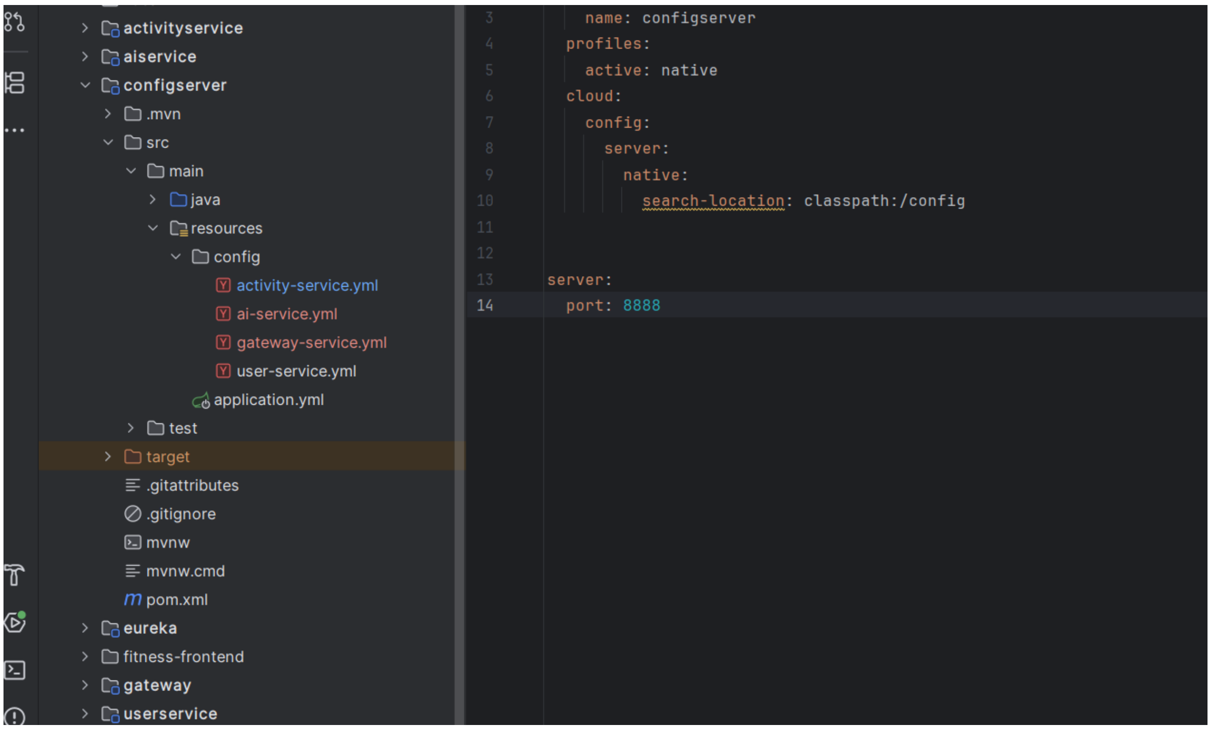
```
import axios from "axios"

const API_URL = 'http://localhost:8080/api'

const api = axios.create({
    baseURL: API_URL
});

api.interceptors.request.use( (config) => {
    const userId = localStorage.getItem( 'userId');
    const token = localStorage.getItem( 'token');
    if (token) {
        config.headers['Authorization'] = `Bearer ${token}`;
    }
```

```
         main] com.netflix.discovery.DiscoveryClient    : Initial registry fetch from primary servers failed
         main] com.netflix.discovery.DiscoveryClient    : Using default backup registry implementation which does not do
         main] com.netflix.discovery.DiscoveryClient    : Initial registry fetch from backup servers failed
         main] com.netflix.discovery.DiscoveryClient    : Starting heartbeat executor: renew interval is: 30
         main] c.n.discovery.InstanceInfoReplicator     : InstanceInfoReplicator onDemand update allowed rate per min is
         main] com.netflix.discovery.DiscoveryClient    : Discovery Client initialized at timestamp 1764067890970 with i
         main] o.s.c.n.e.s.EurekaServiceRegistry        : Registering application USER-SERVICE with eureka with status U
         main] com.netflix.discovery.DiscoveryClient    : Saw local status change event StatusChangeEvent [timestamp=176
oReplicator-%d] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_USER-SERVICE/localhost:user-service:8081: regi
         main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8081 (http) with context path '/'
         main] .s.c.n.e.s.EurekaAutoServiceRegistration : Updating port to 8081
oReplicator-%d] c.n.d.s.t.d.RedirectingEurekaHttpClient  : Request execution error. endpoint=DefaultEndpoint{ serviceUrl=
ceAccessException(RestTemplate.java:926)
```

4. Detailed Implementation

This section describes the backend implementation of each component in detail.

4.1 Config Server



```
activityservice
aiservice
configserver
  .mvn
  src
    main
      java
      resources
        config
          activity-service.yml
          ai-service.yml
          gateway-service.yml
          user-service.yml
        application.yml
    test
  target
  .gitattributes
  .gitignore
  mvnw
  mvnw.cmd
  pom.xml
eureka
fitness-frontend
gateway
userservice
```

```yaml
    name: configserver
  profiles:
    active: native
  cloud:
    config:
      server:
        native:
          search-location: classpath:/config


server:
  port: 8888
```

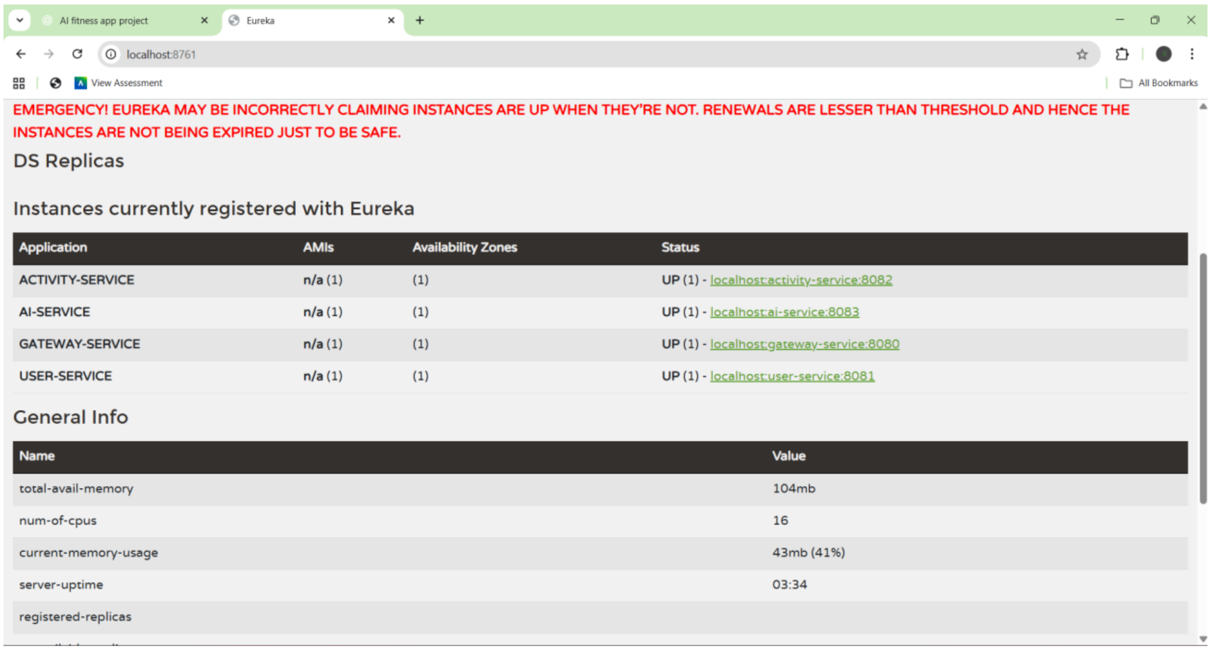
The Config Server centralizes all configuration files (application.yml).

All microservices fetch their configurations at startup.

This avoids duplication and improves maintainability.

The Config Server was created using Spring Cloud Config starter and linked to a configuration repository.

4.2 Eureka Server



EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACTIVITY-SERVICE	n/a (1)	(1)	UP (1) - localhost:activity-service:8082
AI-SERVICE	n/a (1)	(1)	UP (1) - localhost:ai-service:8083
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - localhost:gateway-service:8080
USER-SERVICE	n/a (1)	(1)	UP (1) - localhost:user-service:8081

General Info

Name	Value
total-avail-memory	104mb
num-of-cpus	16
current-memory-usage	43mb (41%)
server-uptime	03:34
registered-replicas	

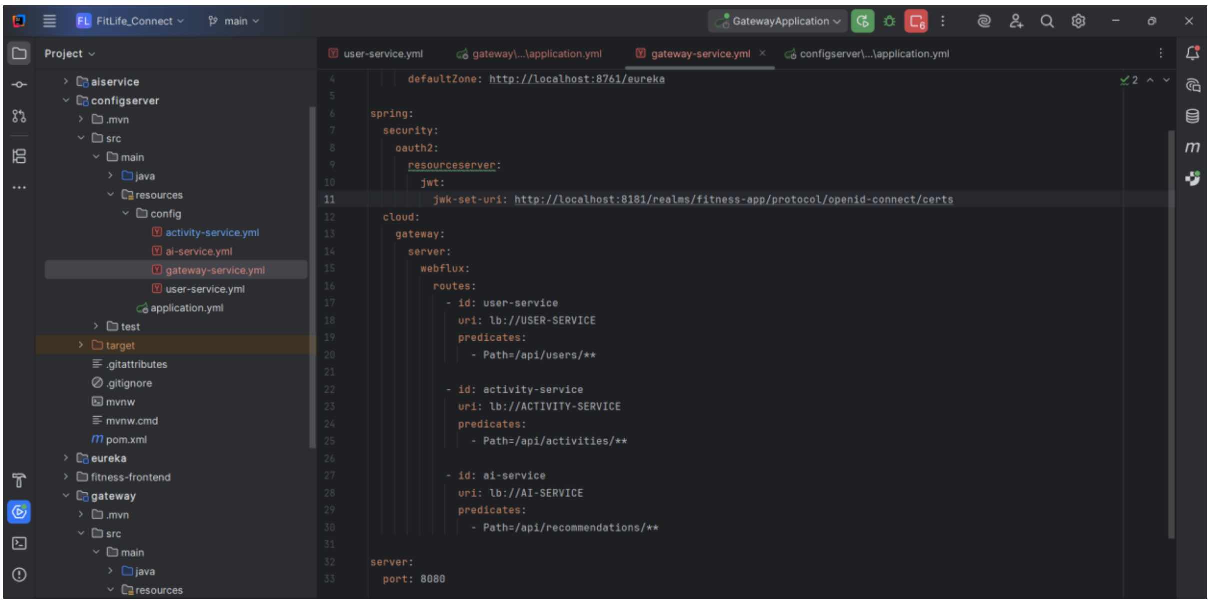
The Eureka Server acts as the service registry.

Each microservice automatically registers itself using its service name.

The Gateway queries Eureka to discover where each service is running.

This removes hard-coded URLs and allows dynamic routing even if services restart.

4.3 API Gateway



```yaml
      defaultZone: http://localhost:8761/eureka

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: http://localhost:8181/realms/fitness-app/protocol/openid-connect/certs
  cloud:
    gateway:
      server:
        webflux:
          routes:
            - id: user-service
              uri: lb://USER-SERVICE
              predicates:
                - Path=/api/users/**

            - id: activity-service
              uri: lb://ACTIVITY-SERVICE
              predicates:
                - Path=/api/activities/**

            - id: ai-service
              uri: lb://AI-SERVICE
              predicates:
                - Path=/api/recommendations/**

server:
  port: 8080
```

The API Gateway acts as the secure edge layer of the system.

Features implemented:

- All client calls pass through the gateway.
- Keycloak token validation using OAuth2.
- Routing to internal services based on URL patterns.
- Filters for security and request logging.

Routing example:

- /api/user/** → User Service
- /api/activity/** → Activity Service
- /api/ai/** → AI Service

This architecture hides internal microservices from direct exposure.

4.4 User Service (PostgreSQL)

The User Service stores and manages user profiles.

Technologies used:

- Spring Boot Web
- Spring Data JPA
- PostgreSQL
- Lombok
- Validation annotations

Key Features:

- UUID-based user ID generation
- Password & email validation
- Automatic timestamps using @CreationTimestamp
- Data stored using PostgreSQL table users

The service also integrates with Keycloak for secure login/logout.

4.5 Activity Service (MongoDB)

The Activity Service handles all user activity logs.

Functions implemented:

- Add new activity
- Retrieve all activities for a user
- Publish activity events to Kafka

MongoDB was used because activity logs may vary in format and grow continuously.

Kafka Publishing

Whenever a user logs an activity, a message is sent to Kafka topic:

activity-topic

The message contains:

- userId

- activity type
- calories burned
- duration
- timestamp

This triggers asynchronous processing in the AI Service.

4.6 Kafka Integration

Kafka is deployed using Docker.

Two components are used:

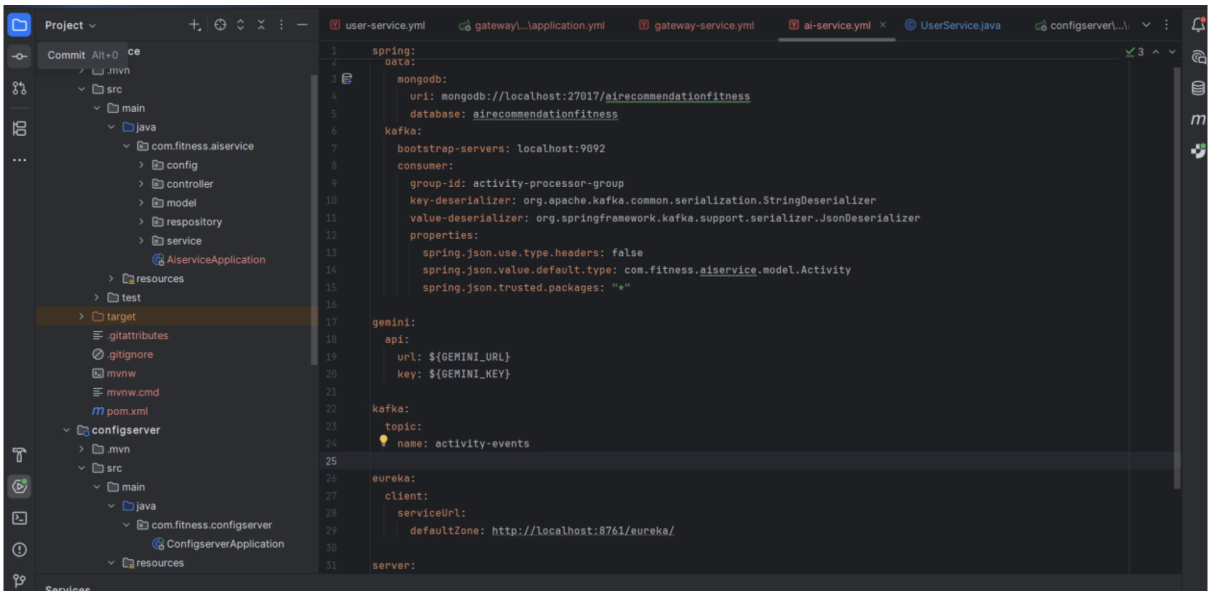
- Kafka Broker
- Zookeeper

Why Kafka?

- High throughput
- Asynchronous event processing
- Services remain loosely coupled
- AI Service can process activities without delaying the user

This design ensures the system stays responsive and scalable.

4.7 AI Service (Gemini API + MongoDB)



```yaml
spring:
  data:
    mongodb:
      uri: mongodb://localhost:27017/airecommendationfitness
      database: airecommendationfitness
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      group-id: activity-processor-group
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.use.type.headers: false
        spring.json.value.default.type: com.fitness.aiservice.model.Activity
        spring.json.trusted.packages: "*"

gemini:
  api:
    url: ${GEMINI_URL}
    key: ${GEMINI_KEY}

kafka:
  topic:
    name: activity-events

eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/

server:
```

The AI Service is the intelligent component of the system.

Workflow:

1. AI Service consumes activity events from Kafka.
2. Extracts required activity information.
3. Calls **Google Gemini AI API**.
4. Receives AI-generated recommendations.
5. Saves data to MongoDB.
6. Sends response to frontend via REST API.

Gemini Output Includes:

- Overall analysis
- Improvement suggestions
- Safety guidelines
- Activity summary

The AI Service transforms raw activity logs into personalized fitness insights.

5. Results & Conclusion

The backend system is fully developed with all six services functioning independently.

The microservices interact using modern enterprise-level patterns such as service discovery, API gateway routing, JWT-based authentication, and Kafka event processing.

The AI Service successfully generates personalized recommendations using Google Gemini API.

5.1 Achievements

- Built a scalable, maintainable microservices architecture.
- Implemented event-driven processing using Kafka.
- Integrated AI using a free and powerful model.
- Designed two different databases for optimal data handling.

- Created a secure ecosystem with Keycloak authentication.

5.2 Future Enhancements

- Deployment using Docker Compose or Kubernetes.
- Adding caching using Redis.
- Enhancing AI recommendations with custom model training.
- Creating an analytics dashboard in the UI.