Francisco José de Caldas District University

Department of Systems Engineering

Technical Report of Workshop Results of SportGear Online

Juan Esteban Carrillo Garcia - 20212020147
Alejandro Sebastián González Torres - 20191020143

Miguel Angel Babativa Niño - 20191020069

Professor: Carlos Andrés Sierra Virgüez

A report submitted to expose the phases of
design of a software product for the course of
Software Engineer seminar

December 12, 2025

Declaration

We, Juan Esteban Carrillo Garcia, Alejandro Sebastián González Torres, Miguel Angel Babativa Niño, of the Department of Systems Engineering, Francisco José de Caldas District University, confirm that this is our own work and figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

I give consent to a copy of my report being shared with future students as an exemplar.

I give consent for my work to be made available more widely to members of UoR and public with interest in teaching, learning and research.

Juan Esteban Carrillo Garcia
Alejandro Sebastián González Torres
Miguel Angel Babativa Niño
December 12, 2025

Abstract

This report presents the design, implementation, containerization, and automation of **Sport-Gear Online**, a distributed e-commerce platform specialized in sports products. The project follows a phased methodology that integrates business modeling with technical implementation. The initial phase applied strategic design tools—including the **Business Model Canvas**, **User Story Mapping**, and **CRC Cards**—to align business objectives with user requirements. This foundation was formalized through **UML Class Diagrams**, **Integration and Deployment Architecture Diagrams**, and **BPMN models**, establishing a clear and maintainable system structure.

In subsequent implementation phases, the platform was developed as a full-stack application composed of two independent backends and a web frontend. The authentication service was built with **Java Spring Boot** using JWT tokens and MySQL, while the business logic service was implemented with **Python FastAPI** connected to PostgreSQL. The frontend, developed in **React**, consumes REST APIs from both backends to deliver a responsive user interface. Development followed **Test-Driven Development (TDD)** principles, with unit tests executed via **JUnit** and **pytest**.

The final phase focused on deployment readiness through **Docker containerization** and **Docker Compose orchestration**, ensuring environment consistency and scalability. **Behavior-Driven Development (BDD)** with **Cucumber** validated user acceptance criteria, while **Apache JMeter** was used for API stress testing. A **CI/CD pipeline** implemented with **GitHub Actions** automated testing, image building, and integration validation. The results demonstrate a fully functional, scalable, and production-ready e-commerce system that successfully bridges business strategy with robust technical execution.

Keywords: e-commerce, distributed architecture, containerization, CI/CD, acceptance testing, TDD, REST API, Docker, Spring Boot, FastAPI, React

Contents

List of Abbreviations

e-commerce	Electronic Commerce

Chapter 1

Introduction

1.1 Background

The digital transformation of commerce has accelerated the adoption of online platforms for the sale of goods and services. In the sports equipment sector, this shift has been particularly pronounced, driven by consumer demand for convenience, accessibility, and personalized shopping experiences. Modern customers increasingly prefer digital channels that allow them to browse, compare, and purchase products from anywhere, without the constraints of physical store hours or location.

To address this evolving landscape, the **SportGear Online** project was conceived as a comprehensive e-commerce platform specializing in sports products. The initiative spans multiple development phases—from initial business modeling and architectural design to full-stack implementation, containerization, and continuous delivery. This integrated approach ensures that the final system not only meets functional requirements but also aligns with strategic business objectives, technical best practices, and scalability demands.

1.2 Problem statement

Despite the growth of e-commerce, many online sports retail platforms struggle with fragmented development processes, poor integration between business logic and technical implementation, and inadequate testing and deployment practices. Common challenges include:

- Misalignment between business goals and system features. - Inconsistent environments leading to deployment failures. - Manual testing and deployment processes that are error-prone and inefficient. - Limited scalability and maintainability due to monolithic or poorly structured architectures. - Insufficient validation of user acceptance and performance under load.

This project addresses these issues by adopting a holistic, phase-driven methodology that integrates strategic modeling, full-stack development, automated testing, containerization, and CI/CD. The goal is to deliver a robust, scalable, and production-ready platform that bridges the gap between business vision and technical execution.

1.3 Aims and objectives

1.3.1 General Aim

To design, implement, containerize, and automate a distributed e-commerce platform for sports equipment that integrates business strategy, user-centered design, and modern software

engineering practices.

1.3.2 Specific Objectives

- Define the strategic framework of the platform using the Business Model Canvas.
- Translate business goals into user-centered requirements through User Stories and User Story Mapping.
- Model the object-oriented architecture with CRC Cards and UML Class Diagrams.
- Design system integration and deployment using Architecture Diagrams.
- Represent business workflows with BPMN.
- Implement a dual-backend architecture using Spring Boot (Java) and FastAPI (Python).
- Develop a responsive frontend in React.
- Containerize the application using Docker and Docker Compose.
- Implement acceptance testing with Cucumber (BDD).
- Conduct performance validation with Apache JMeter.
- Automate CI/CD pipelines using GitHub Actions.

1.4 Solution approach

The project follows a structured, iterative approach divided into four complementary phases:

1. **Conceptual Design**: Business modeling using BMC, user story mapping, CRC Cards, UML class diagrams, and BPMN to ensure alignment between business logic and system structure.

2. **Full-Stack Implementation**: Development of a distributed architecture with separate backends for authentication (Spring Boot + MySQL) and business logic (FastAPI + PostgreSQL), integrated with a React-based frontend via REST APIs. TDD principles guided development, with unit tests in JUnit and pytest.

3. **Containerization and DevOps**: Application containerization with Docker and orchestration via Docker Compose to ensure environment consistency, portability, and scalability. Acceptance testing was implemented with Cucumber, and API stress testing with JMeter.

4. **Automation and CI/CD**: Implementation of a GitHub Actions workflow to automate testing, Docker image building, and integration validation, enabling continuous delivery and deployment readiness.

This multi-phase methodology ensures that each component of the system—from business logic to deployment—is validated, modular, and maintainable.

1.5 Summary of contributions and achievements

This project delivers a fully integrated, tested, and deployable e-commerce platform with the following key contributions:

- A complete business-to-technology translation framework using industry-standard modeling tools.

- A functional distributed system with secure authentication, product management, and order processing.
- A containerized deployment environment ensuring consistency across development and production.
- Automated acceptance and stress testing pipelines validating user stories and system performance.
- A CI/CD workflow that supports continuous integration, testing, and deployment readiness.
- Comprehensive documentation and reproducible setups for each phase of the project.

1.6 Organization of the report

This report is organized into seven chapters:

- **Chapter 2: Literature Review** – Examines foundational concepts in e-commerce architecture, software modeling, testing, containerization, and CI/CD. - **Chapter 3: Methodology** – Describes the phased approach, tools, and processes used across design, implementation, containerization, and automation. - **Chapter 4: Results** – Presents outputs from each phase, including models, code, test results, container images, and CI/CD execution evidence. - **Chapter 5: Discussion** – Analyzes the effectiveness of the methodology, integration success, limitations, and lessons learned. - **Chapter 6: Conclusions and Future Work** – Summarizes achievements, challenges, and proposes directions for further development. - **Chapter 7: Reflection** – Offers personal and team insights on skills developed, challenges faced, and project impact.

Chapter 2

Literature Review

2.1 Object-Oriented Design and CRC Cards

Object-oriented software design relies on identifying classes, their responsibilities, and collaborations. CRC Cards (Class–Responsibility–Collaboration), introduced by Cunningham and Beck in 1989, provide a lightweight and interactive technique for exploring class structures in team-based environments Cunningham and Beck (1989). Each card lists a class's main responsibilities and collaborators, clarifying system architecture during early design. The CRC approach promotes simplicity and iteration—key attributes of agile software engineering—enabling developers to collaboratively refine system structure and establish a formal model. In this project, CRC Cards served as the initial abstraction for defining interactions between entities within SportGear Online, bridging user requirements and class design.

2.2 Unified Modeling Language (UML) and System Design

The Unified Modeling Language (UML) offers a standardized notation for modeling software systems, essential for translating conceptual models into implementation. Larman Larman (2004) emphasizes UML's role in supporting iterative and incremental development, particularly in visualizing relationships between components. Similarly, Booch, Rumbaugh, and Jacobson Booch et al. (2011) describe UML as a unifying standard that integrates object-oriented analysis and design practices. In this project, UML class diagrams represented the system's logical structure, including classes, attributes, methods, and their interconnections. Resources such as IONOS Digital Guide IONOS Digital Guide (2025) and GeeksforGeeks GeeksforGeeks (2025*b*) provided practical examples, ensuring adherence to current UML best practices.

2.3 User Story Mapping and Agile Requirements Engineering

Modern software project success depends on aligning business goals with user needs. User Stories and User Story Mapping are core agile tools that facilitate this connection. Planio's practical guide Planio (2025) explains that user story mapping visualizes user interactions, helping teams prioritize features, define minimum viable products (MVPs), and ensure continuous value delivery. In SportGear Online, user story mapping translated strategic components from the Business Model Canvas into specific functionalities, maintaining focus on customer-centric outcomes and ensuring each technical feature supports a defined business objective.

2.4 Full-Stack Development and Distributed Architecture

Full-stack development integrates frontend, backend, and database layers to create cohesive applications. According to Reaboi Reaboi (2024), modern architectures separate presentation, logic, and data layers to enhance modularity and scalability. Distributed systems, as described by Tanenbaum and Van Steen Tanenbaum and Van Steen (2007), enable independent service deployment and communication via APIs. SportGear Online adopts a distributed full-stack architecture with separate backends for authentication (Spring Boot) and business logic (FastAPI), connected through REST APIs to a React frontend, ensuring clear responsibility separation and scalability.

2.5 Backend Frameworks: Spring Boot and FastAPI

Spring Boot simplifies Java-based backend development with built-in support for dependency injection, security, and data persistence, making it ideal for authentication services Wallace (2019). Its integration with JSON Web Tokens (JWT) enables secure, stateless communication, as highlighted by Auth0 Auth0 (2025). FastAPI, a Python framework, offers high-performance API development with automatic data validation and asynchronous support Tiango (2020). Suarez Suarez (2025) notes its compatibility with OpenAPI for automatic documentation. In SportGear Online, Spring Boot manages authentication with JWT, while FastAPI handles product and order logic, leveraging their respective strengths for secure and efficient backend services.

2.6 Frontend Development with React

Frontend development focuses on user experience and interface design. React, a JavaScript library, enables dynamic, component-based user interfaces that efficiently communicate with backend APIs. As explained by GeeksforGeeks GeeksforGeeks (2025*a*), React's virtual DOM and component reusability improve performance and maintainability. In SportGear Online, React serves as the presentation layer, consuming REST APIs to display products, manage user sessions, and process orders, providing a responsive and intuitive user experience.

2.7 Software Testing and Quality Assurance

Software testing and quality assurance (QA) ensure reliability and maintainability. Test-Driven Development (TDD), defined by Beck Beck (2003), involves writing failing tests before implementation, refining code iteratively. JUnit and pytest are widely used for unit testing in Java and Python, respectively Link and Fröhlich (2020); Okken (2022). Behavior-Driven Development (BDD) with Cucumber bridges technical and non-technical stakeholders through executable specifications in natural language Wynne and Hellesøy (2012). In SportGear Online, TDD guided development with JUnit and pytest, while Cucumber validated acceptance criteria, ensuring comprehensive test coverage and alignment with user requirements.

2.8 Containerization with Docker and Docker Compose

Containerization packages applications and dependencies into isolated, portable environments. Docker, introduced by Merkel Merkel (2014), standardizes containerization, enabling consistent deployment across systems. Docker Compose orchestrates multi-container applications

via a single configuration file, simplifying management of complex services Docker Documentation (2024). In SportGear Online, Docker containers encapsulate each component (Java backend, Python backend, frontend, databases), while Docker Compose orchestrates their deployment, ensuring environment consistency and scalability.

2.9 Performance and Stress Testing with JMeter

Performance testing evaluates system behavior under load. Apache JMeter, an open-source tool, simulates heavy traffic on web applications to assess performance and identify bottlenecks Bayon and Romero (2018). Its support for HTTP, HTTPS, and REST APIs makes it suitable for modern architectures Apache Software Foundation (2024). In SportGear Online, JMeter stress tests validated API responsiveness and stability under concurrent user requests, ensuring the platform can handle expected loads.

2.10 Continuous Integration and Deployment (CI/CD)

CI/CD automates integration, testing, and deployment. GitHub Actions enables workflow automation within GitHub, triggered by repository events like pushes or pull requests Sharma (2021). Fowler Fowler (2006) emphasizes that continuous integration improves software quality by regularly testing changes. In SportGear Online, a GitHub Actions pipeline automates testing, Docker image building, and integration validation, supporting continuous delivery and reducing manual deployment errors.

2.11 Summary

The reviewed literature establishes a robust foundation for developing, testing, and deploying distributed e-commerce systems. CRC Cards and UML provide structural clarity, while user story mapping aligns features with business goals. Spring Boot and FastAPI enable secure, high-performance backends, and React delivers dynamic frontends. Docker and Docker Compose ensure consistent environments, while JMeter and Cucumber validate performance and acceptance. GitHub Actions automates CI/CD, integrating these practices into a cohesive workflow. Together, these concepts and tools underpin the successful implementation of SportGear Online as a scalable, maintainable, and production-ready platform.

Chapter 3

Methodology

3.1 Overview

This chapter describes the structured methodology followed throughout the SportGear Online project, spanning four distinct phases: conceptual modeling, full-stack development, containerization and DevOps, and continuous integration/deployment. Each phase employed specific tools and practices to ensure alignment between business objectives, user needs, and technical implementation, resulting in a scalable, maintainable, and production-ready e-commerce platform.

3.2 Development Framework

The project adopted an agile, iterative approach organized into four sequential phases, each with clear deliverables and validation steps:

1. **Conceptual Design Phase** – Business and architectural modeling using BMC, user stories, CRC Cards, UML, BPMN, and architecture diagrams.
2. **Full-Stack Implementation Phase** – Development of distributed backends (Java/Python) and a React frontend, following TDD with JUnit and pytest.
3. **Containerization and DevOps Phase** – Docker-based containerization, acceptance testing with Cucumber, and performance testing with JMeter.
4. **Automation and CI/CD Phase** – Implementation of a GitHub Actions pipeline for automated testing, building, and integration validation.

This phased approach enabled incremental validation, risk mitigation, and continuous refinement of both business and technical artifacts.

3.3 Conceptual Design Phase

3.3.1 Business Model Canvas

The Business Model Canvas (BMC) was used to define the strategic foundation of SportGear Online. The nine building blocks—key partners, activities, resources, value propositions, customer relationships, channels, customer segments, cost structure, and revenue streams—were analyzed to align the platform's business logic with its technical design. This canvas served as the primary artifact linking market objectives to system requirements.

3.3.2 User Stories and User Story Mapping

User stories were derived from the BMC to capture functional needs from the perspectives of different actors (e.g., customers, administrators). Following the approach described by Planio Planio (2025), a user story map was created to visualize the end-to-end user journey, prioritize features for the minimum viable product (MVP), and establish a clear development roadmap. Stories were organized into epics such as "Product Browsing," "Order Management," and "Payment Processing."

3.3.3 CRC Cards

CRC Cards (Class–Responsibility–Collaboration) were employed to draft the initial object-oriented structure of the system. As recommended by Cunningham and Beck Cunningham and Beck (1989), each card defined a class's responsibilities and collaborations, facilitating team discussion and early detection of design flaws. Key entities included `User`, `Product`, `Order`, and `Payment`.

3.3.4 UML Class Diagrams

The CRC Cards were formalized into UML class diagrams using standard notation Larman (2004); Booch et al. (2011). Diagrams illustrated classes, attributes, methods, and relationships (association, aggregation, inheritance), providing a blueprint for implementation. Online resources such as IONOS IONOS Digital Guide (2025) and GeeksforGeeks GeeksforGeeks (2025*b*) were consulted to ensure correct semantics and completeness.

3.3.5 Integration and Deployment Architecture Diagrams

High-level architecture diagrams were created to depict component interactions and deployment topology. The integration diagram showed communication between internal modules (frontend, backends, databases) and external services (payment gateways, logistics APIs). The deployment diagram outlined the hosting environment, including web servers, application servers, and database servers, following scalable design principles **?**.

3.3.6 Business Process Modeling (BPMN)

BPMN diagrams modeled core business processes such as "Order Fulfillment" and "Inventory Update." These diagrams bridged operational workflows with system functionality, ensuring that the software supported real-world business sequences and exception handling.

3.4 Full-Stack Implementation Phase

3.4.1 System Architecture

The platform was implemented as a distributed full-stack application with two independent backends and a single frontend:

- **Authentication Backend**: Java Spring Boot service using JWT for token-based security, connected to a MySQL database.
- **Business Logic Backend**: Python FastAPI service handling products, orders, and payments, persisted in PostgreSQL.

- **Frontend**: React application consuming REST APIs from both backends for a dynamic user interface.

The backends communicate via REST APIs, ensuring loose coupling and independent scalability.

3.4.2 Development Practices

Test-Driven Development (TDD) was adopted throughout implementation. Unit tests were written before production code using JUnit 5 for the Java backend and pytest for the Python backend. This practice ensured high test coverage, early bug detection, and adherence to requirements.

3.4.3 API Design and Documentation

RESTful endpoints were designed following OpenAPI standards. FastAPI automatically generated interactive documentation (Swagger UI), while Spring Boot endpoints were documented using Spring REST Docs. All APIs exchanged JSON payloads and enforced authentication via JWT tokens.

3.5 Containerization and DevOps Phase

3.5.1 Docker Containerization

Each component was containerized using optimized Dockerfiles:

- **Java Backend**: Multi-stage build with Maven and Eclipse Temurin base image.
- **Python Backend**: Python 3.11-slim image with dependency compilation.
- **Frontend**: Node.js build stage and Nginx Alpine production image.
- **Databases**: Official MySQL and PostgreSQL images with persistent volumes.

3.5.2 Orchestration with Docker Compose

Services were orchestrated using Docker Compose, defining networks, environment variables, health checks, and dependency orders. The configuration ensured isolated, reproducible environments suitable for development, testing, and production-like deployments.

3.5.3 Acceptance Testing with Cucumber

Acceptance criteria were expressed as Gherkin feature files (e.g., `order_viewing.feature`) and automated with pytest-bdd. Step definitions validated API responses, business rules, and state transitions, bridging non-technical specifications with executable tests.

3.5.4 Performance Testing with JMeter

Apache JMeter was used to design and execute stress tests on critical endpoints (e.g., `/api/v1/products`, `/api/auth/login`). Test plans simulated concurrent user loads, measuring response times, throughput, and error rates to identify performance bottlenecks.

3.6 Automation and CI/CD Phase

3.6.1 GitHub Actions Pipeline

A CI/CD pipeline was implemented using GitHub Actions, triggered on pushes to `main` and `develop` branches and on pull requests. The workflow included four parallel jobs:

1. `test-java-backend`: Maven-based unit and integration tests.
2. `test-python-backend`: pytest execution with BDD features.
3. `build-docker-images`: Docker image builds for all components.
4. `integration-test`: Docker Compose-based service validation.

3.6.2 Automation Outcomes

The pipeline provided automated feedback on code quality, test coverage, and integration readiness. Successful runs produced deployable Docker images, while failures triggered notifications for immediate remediation.

3.7 Summary

This integrated methodology ensured that SportGear Online evolved from a business concept to a fully containerized, tested, and automatable system. Each phase built upon the previous one, maintaining traceability between requirements, design, code, and deployment artifacts. The combination of modeling tools, modern frameworks, containerization, and CI/CD practices resulted in a robust platform that meets both business and technical excellence criteria.

Chapter 4

Results

4.1 Introduction

This chapter presents the key outcomes of the SportGear Online project across four development phases: conceptual modeling, full-stack implementation, containerization and DevOps, and CI/CD automation. The results demonstrate the translation of business strategy into a functional, scalable, and deployable e-commerce platform, validated through both static models and dynamic execution.

4.2 Conceptual Design Results

4.2.1 Business Model Canvas

The Business Model Canvas defined the strategic foundation of the e-commerce sports equipment platform, capturing nine essential components:

- **Key Partners:** Sports equipment suppliers, logistics and shipping companies, payment gateways (Cards, PSE), financial entities, e-commerce platforms.
- **Key Activities:** Product catalog management, order and payment processing, shipping and tracking management, customer support (pre and post-sale), digital marketing and customer loyalty.
- **Value Proposition:** Wide variety of sports products organized by sport and gender; personalized user experience (filters by sport, gender, price); multiple payment methods (card, cash).
- **Customer Segments:** Amateur and professional athletes, men and women interested in sports, sellers of sports items, young people and adults with internet and mobile device access.
- **Customer Relationships:** User registration and personalized profile, shopping cart with reminders and editing, support for returns or claims, product comments and ratings.
- **Channels:** E-commerce platform, digital product catalog, secure payment management system, user and order database.
- **Key Resources:** Technology and digital infrastructure, human capital, financial capital, data and intellectual property.

- **Cost Structure:** Technology development and maintenance, marketing and promotion, operational costs, transaction commissions.
- **Revenue Streams:** Sales commissions, advertising and brand promotion on the platform, premium subscriptions (e.g., free shipping, discounts), featured listing sales for sellers.

This canvas ensured that technical development remained aligned with business goals.

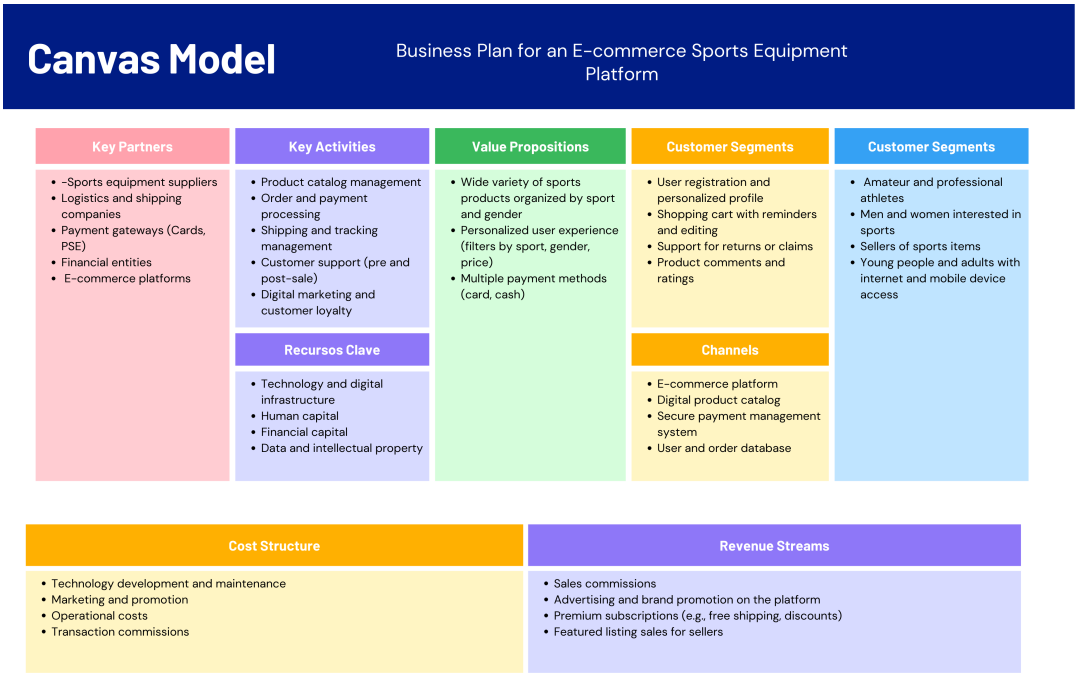

Figure 4.1: Business Model Canvas for the e-commerce sports equipment platform.

4.2.2 User Stories and User Story Mapping

A total of 19 user stories were defined, categorized into epics such as:

- **Authentication & Profile:**

Title: User Registration	**Prority:** High	**Estimate:** 5 points
User Story: As a visitor, I want to register with email and password so I can access the platform.		
Acceptance Criteria: Registration form validates required fields and password strength. Duplicate email is rejected with friendly error. Successful registration creates user in database and returns confirmation.		

"As a visitor, I want to register with email and password so I can access the platform."

Title: User Login	**Prority:** High	**Estimate:** 5 points
User Story: As a user, I want to log in to access my account.		
Acceptance Criteria: Login authenticates via backend and returns JWT/session. Invalid credentials show error without revealing specifics. Auth state persists and protects private routes.		

"As a user, I want to log in to access my account."

Title: JWT-Based Auth Middleware	**Prority:** High	**Estimate:** 3 points
User Story: As a developer, I need middleware to protect API routes.		
Acceptance Criteria: Backend rejects requests without valid JWT. Token expiry and refresh handled per config. Role-based checks applied to sensitive endpoints.		

"As a developer, I need middleware to protect API routes."

- **Product Browsing:**

Title: Product Catalog Listing	**Prority:** High	**Estimate:** 5 points
User Story: As a user, I want to browse products with pagination and filters.		
Acceptance Criteria: API supports pagination, search, and category filters. Frontend lists products with loading and empty states. Performance acceptable for 1k+ products.		

"As a user, I want to browse products with pagination and filters."

Title: Product Details Page	**Prority:** Medium	**Estimate:** 3 points
User Story: As a user, I want to view a product's details.		
Acceptance Criteria: Displays images, description, price, stock. Handles not-found gracefully. Deep links load directly.		

"As a user, I want to view a product's details."

Title: Shopping Cart Management	**Prority:** High	**Estimate:** 8 points
User Story: As a user, I want to add, update, and remove items in my cart.		
Acceptance Criteria: Add to cart from listing and detail pages. Update quantity with validation against stock. Cart state persists across sessions.		

"As a user, I want to add, update, and remove items in my cart."

- **Order Management:**

Title: Checkout Flow	**Prority:** High	**Estimate:** 8 points
User Story: As a user, I want to checkout with address and payment details.		
Acceptance Criteria: Validates address and payment inputs. Summarizes order before submission. Creates order and initiates payment transaction.		

"As a user, I want to checkout with address and payment details."

Title: Payment Processing Integration	**Prority:** High	**Estimate:** 8 points
User Story: As a user, I want secure payment processing with clear feedback.		
Acceptance Criteria: Payment API handles authorize/capture and failure states. Error handling and retries per guide. Stores payment status linked to order.		

"As a user, I want secure payment processing with clear feedback."

Title: Order History	**Prority:** Medium	**Estimate:** 5 points
User Story: As a user, I want to see my past orders and statuses.		
Acceptance Criteria: Lists orders with date, items, totals, status. Supports pagination. Links to order detail page.		

"As a user, I want to see my past orders and statuses."

Title: Order Detail View	**Prority:** Medium	**Estimate:** 3 points
User Story: As a user, I want detailed order info and shipment tracking.		
Acceptance Criteria: Shows items, totals, payment, shipment status. Displays tracking number and updates from shipment service.		

"As a user, I want detailed order info and shipment tracking."

- **Payment Processing:**

Title: Shipment Management & Trackin	**Prority:** Medium	**Estimate:** 5 points
User Story: As an operator, I want to update shipment status		
Acceptance Criteria: API to create shipment and update statuses (created, dispatched, delivered). Frontend reflects status changes in order views. Events or polling keep status fresh.		

"As an operator, I want to update shipment status."

Title: Inventory/Stock Update	**Prority:** High	**Estimate:** 5 points
User Story: As a system, I need stock decremented on order and restored on cancellation.		
Acceptance Criteria: Atomic stock updates during order creation. Prevents overselling via transaction/locking. Backoffice API for stock adjustments.		

"As a system, I need stock decremented on order and restored on cancellation."

Title: Admin Product CRUD	**Prority:** Medium	**Estimate:** 8 points
User Story: As an admin, I want to create, update, and delete products.		
Acceptance Criteria: Protected endpoints for product CRUD. Frontend admin UI with form validation. Audit log of changes.		

"As an admin, I want to create, update, and delete products."

Title: User Profile Management	**Prority:** Low	**Estimate:** 3 points
User Story: As a user, I want to update profile details and password.		
Acceptance Criteria: Update name, address, phone. Change password with current-password check. Confirmation and error handling.		

"As a user, I want to update profile details and password."

Title: Error Handling & Notifications	**Prority:** Medium	**Estimate:** 3 points
User Story: As a user, I want clear error messages and success notifications.		
Acceptance Criteria: Standardized API error schema. Frontend toast/inline notifications. Network failures show retry options.		

"As a user, I want clear error messages and success notifications."

- **DevOps & CI/CD:**

Title: CI Test Coverage for Python Backend	**Prority:** Medium	**Estimate:** 5 points
User Story: As a developer, I want unit tests with coverage for Python services.		
Acceptance Criteria: Pytest runs with coverage report. Threshold enforced (e.g., 80% lines). Failing tests block merge.		

"As a developer, I want unit tests with coverage for Python services."

Title: CI Build & Test for Java Backend	**Prority:** Medium	**Estimate:** 5 points
User Story: As a developer, I want Maven build and tests in CI.		
Acceptance Criteria: mvn test runs and reports failures. Basic unit tests cover critical endpoints. Artifacts generated for deployment.		

"As a developer, I want Maven build and tests in CI."

Title: Dockerized Local Environment	**Prority:** High	**Estimate:** 8 points
User Story: As a developer, I want docker-compose to run frontend, backends, and database locally.		
Acceptance Criteria: docker-compose.yml builds and runs services. Environment variables configured via .env. Services communicate via network and ports.		

"As a developer, I want docker-compose to run frontend, backends, and database locally."

Title: Database Schema Initialization	**Prority:** High	**Estimate:** 5 points
User Story: As a system, I need automatic schema setup for both backends.		
Acceptance Criteria: Init scripts run on startup. Idempotent migrations avoid duplicate objects. Seed data for demo use.		

"As a system, I need automatic schema setup for both backends."

The user story map organized these stories into a visual workflow, enabling prioritization of the MVP and clear sprint planning.

4.2.3 CRC Cards

The CRC Cards capture responsibilities and collaborations across the Python (orders, catalog, payments, shipments) and Java (identity, addresses) domains:

- **CustomerProfile (Python):**
  - **Responsibilities:** Maintains local customer profile without credentials; ensures uniqueness of `external_user_id` (UUID); supports lookups by id/email; optionally syncs profile fields from identity service.
  - **Collaborators:** `Order` (ownership via `user_id` in queries).
- **Product (Python):**
  - **Responsibilities:** Represents catalog item attributes (name, description, price, availability); enforces invariants (non-negative price/stock); serves as source for line items.
  - **Collaborators:** `OrderItem`.
- **Order (Python):**
  - **Responsibilities:** Aggregates order lifecycle (status, timestamps) and shipping address snapshot; owns and manages line items (add/update/remove); computes totals consistently; prevents invalid modifications after completion.
  - **Collaborators:** `OrderItem`, `Payment`, `Shipment`.
- **OrderItem (Python):**
  - **Responsibilities:** Encapsulates product, quantity, and unit price at purchase time; computes subtotal; validates quantity/unit price to ensure integrity.
  - **Collaborators:** `Order` and `Product`.
- **Payment (Python):**
  - **Responsibilities:** Records payment attempts and status; supports multiple attempts and failure handling; validates amounts against order totals when applicable; captures audit timestamps.
  - **Collaborators:** `Order`.
- **Shipment (Python):**
  - **Responsibilities:** Tracks fulfillment lifecycle (pending, shipped, in_transit, delivered, cancelled); stores tracking number, carrier, and timestamps; enforces at most one shipment per order.
  - **Collaborators:** `Order`.
- **User (Java):**
  - **Responsibilities:** Owns identity (UUID), credentials hash, role and status; supports registration, authentication, password change, and profile updates; maintains audit fields and manages addresses.
  - **Collaborators:** `Address`, `UserRole`, `UserStatus`.

- **Address (Java):**
  - **Responsibilities:** Models a physical address; validates required components; formats a shipping-ready string; assembles street details from parts.
  - **Collaborators:** `User`.

Note: `Order.user_id` is a UUID (string) referencing the Java `User`; `CustomerProfile` links locally via `external_user_id`. These CRCs guided early design discussions and informed the UML class diagram.

4.2.4 UML Class Diagram

The UML class diagram formalized the CRC Cards into a structured model (Figure 4.2). Key relationships include:

- `User (Java)` – `Order (Python)` (one-to-many via `user_id` UUID; cross-service reference)
- `Order` – `OrderItem` (composition, one-to-many)
- `Product` – `OrderItem` (association, one-to-many)
- `Order` – `Payment` (one-to-many; supports retries/partial captures)
- `Order` – `Shipment` (one-to-zero-or-one; unique shipment per order)
- `User (Java)` – `Address` (one-to-many)
- `CustomerProfile (Python)` – `User (Java)` (optional one-to-one via `external_user_id`; no credentials stored)

Attributes and key methods are specified per class; notably, `Order.user_id` is a `String(36)` referencing the Java `User` UUID.

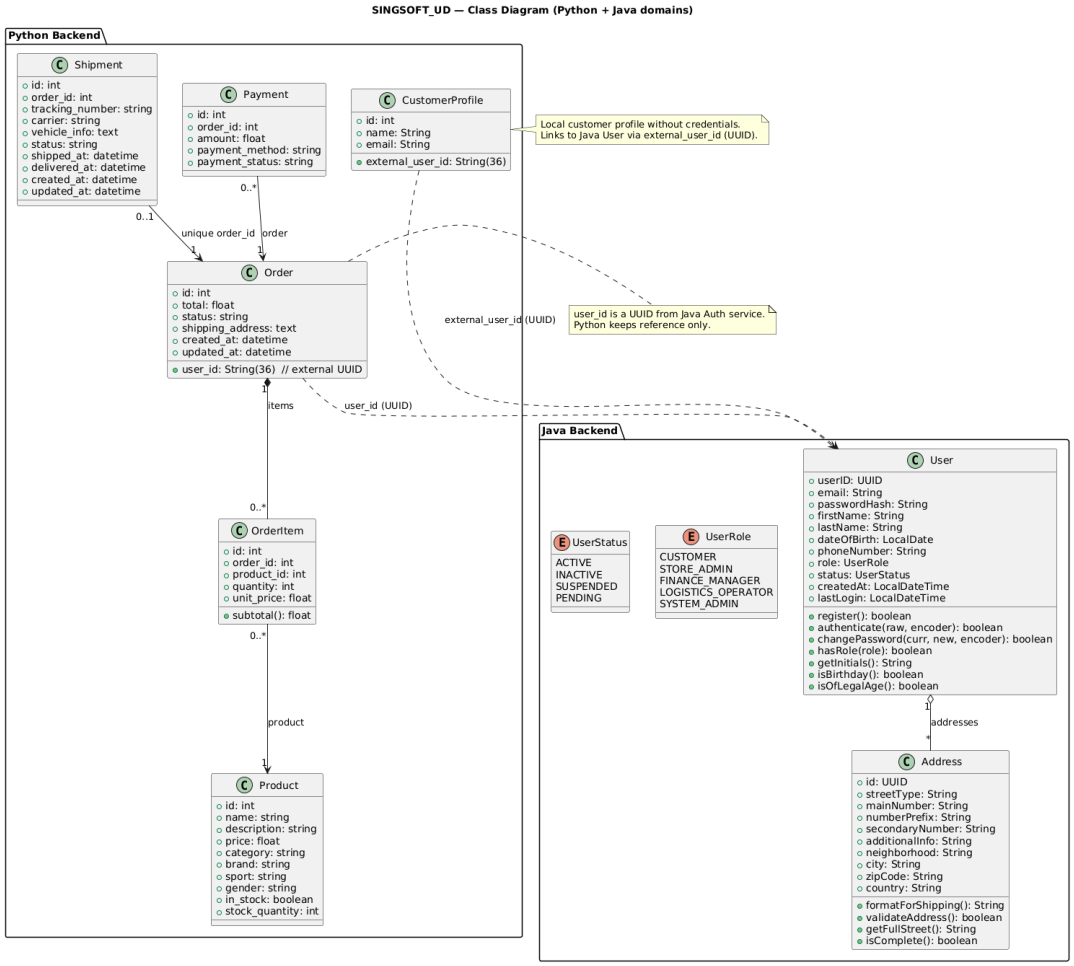

Figure 4.2: UML class diagram aligned with the current implementation.

4.2.5 Integration and Deployment Architecture Diagrams

The integration diagram illustrated a three-tier architecture:

- **Presentation Tier:** React frontend hosted on Nginx.
- **Application Tier:** Spring Boot (authentication) and FastAPI (business logic) microservices.
- **Data Tier:** MySQL (users) and PostgreSQL (products/orders) databases.

External integrations included Stripe (payments) and a logistics API (shipping).

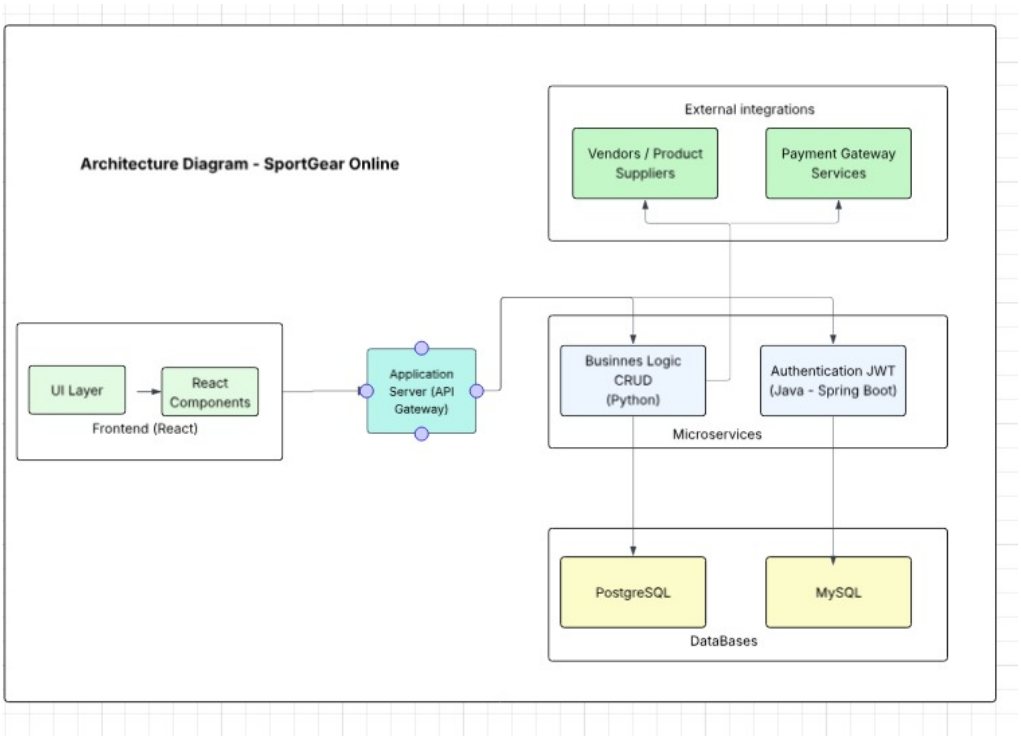

Figure 4.3: General architecture of the SportGear Online platform showing the two backends and the frontend integration.

The deployment diagram specified a cloud-ready topology with separate containers for each service, load balancers, and managed database instances, supporting horizontal scaling.

4.2.6 Business Process Model (BPMN)

The BPMN diagrams model the end-to-end workflows across purchasing, payments, fulfillment, and identity. Key processes:

- **Purchase & Checkout:** Customer browses catalog, manages cart, enters shipping data, reviews summary, and confirms the order. The Order Service validates identity (JWT), creates the order and line items, computes totals, and reserves stock. An *OrderCreated* event triggers payment preparation.

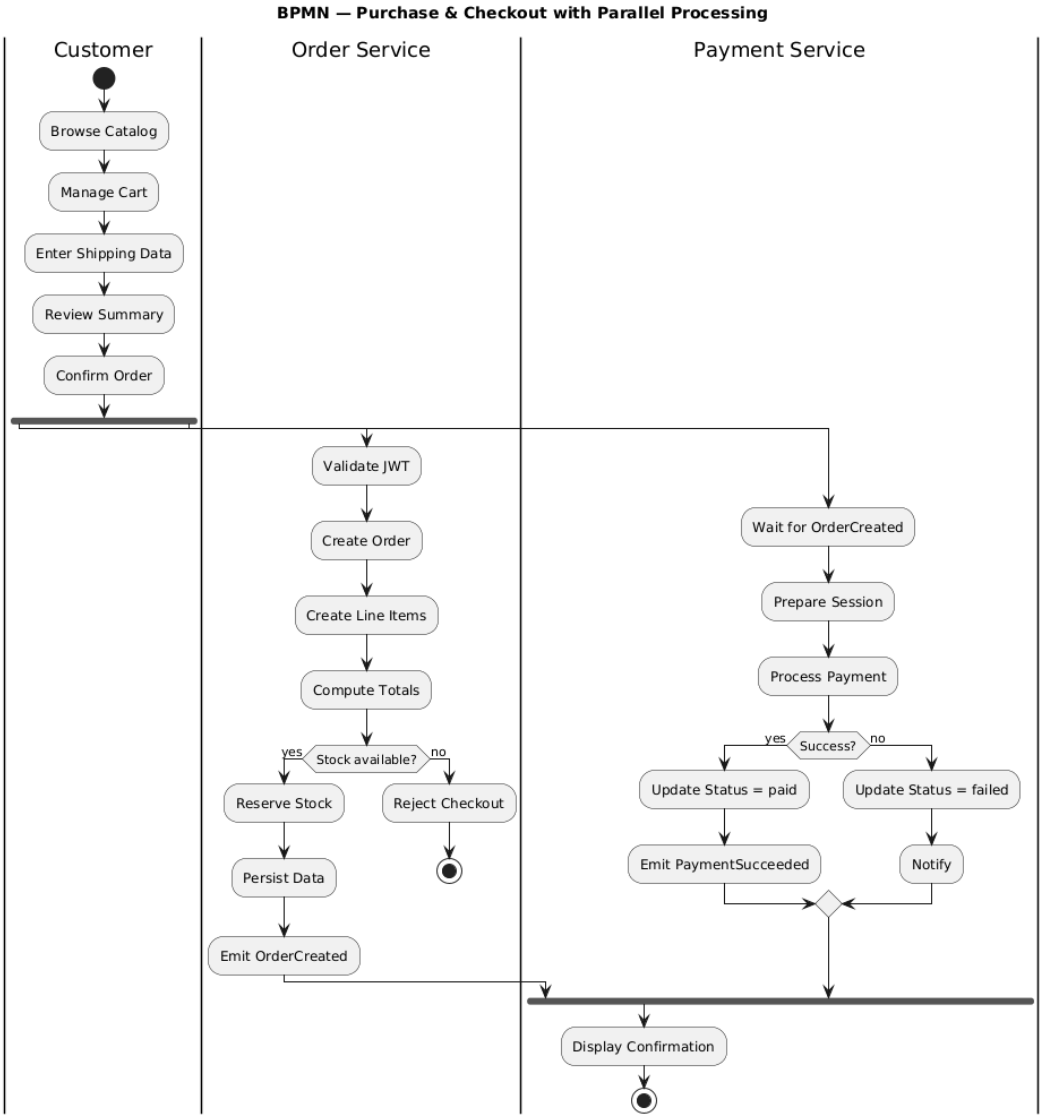

Figure 4.4: Purchase & Checkout BPMN.

- **Payment Processing:** The Payment Service records attempts (pending/authorized/-completed/failed), calls the external gateway (authorize/capture), and updates the Order status (paid or payment_failed). Retries may be applied under configured limits.

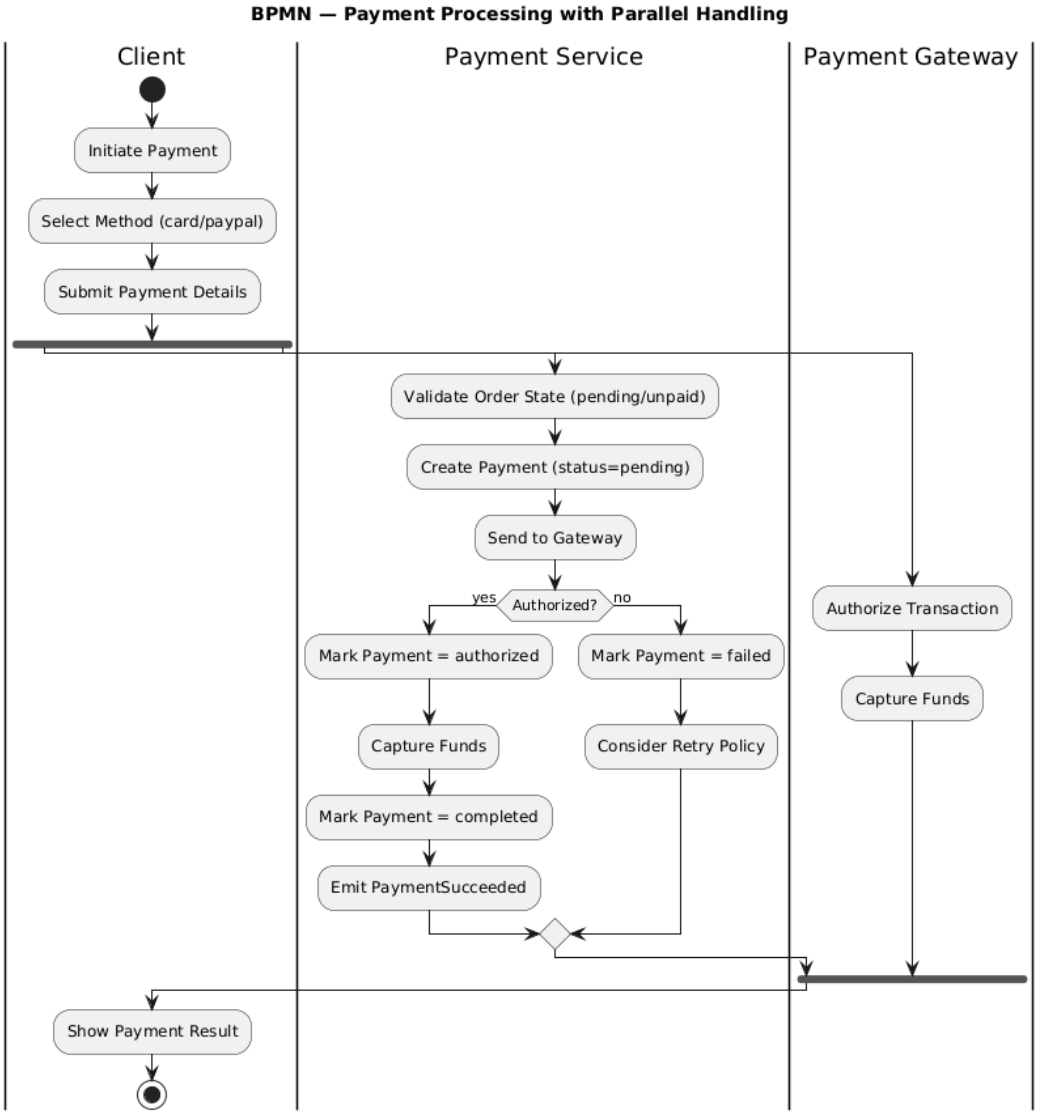

Figure 4.5: Payment Processing BPMN.

- **Shipment Fulfillment:** Upon *paid* orders, Logistics creates a shipment (pending), assigns carrier/tracking, transitions to shipped/in_transit/delivered/cancelled, and reflects status in the Order. Customer tracks delivery via `tracking_number`.

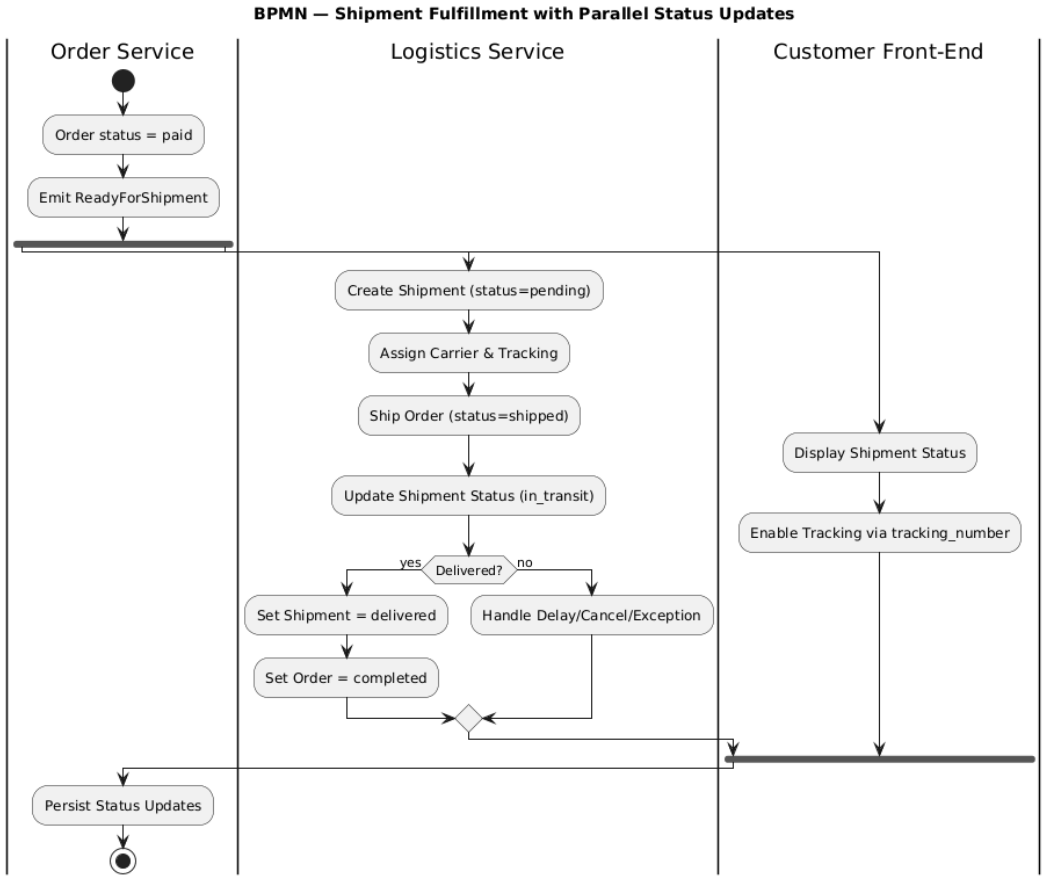

Figure 4.6: Shipment Fulfillment BPMN.

- **Identity: Registration & Login:** The Java Auth service creates Users (UUID, password hash, role, status) and issues JWTs. Python services consume JWTs (extracting `user_id` UUID), optionally maintain a local `CustomerProfile` (`external_user_id`) for convenience, and proceed with domain operations.

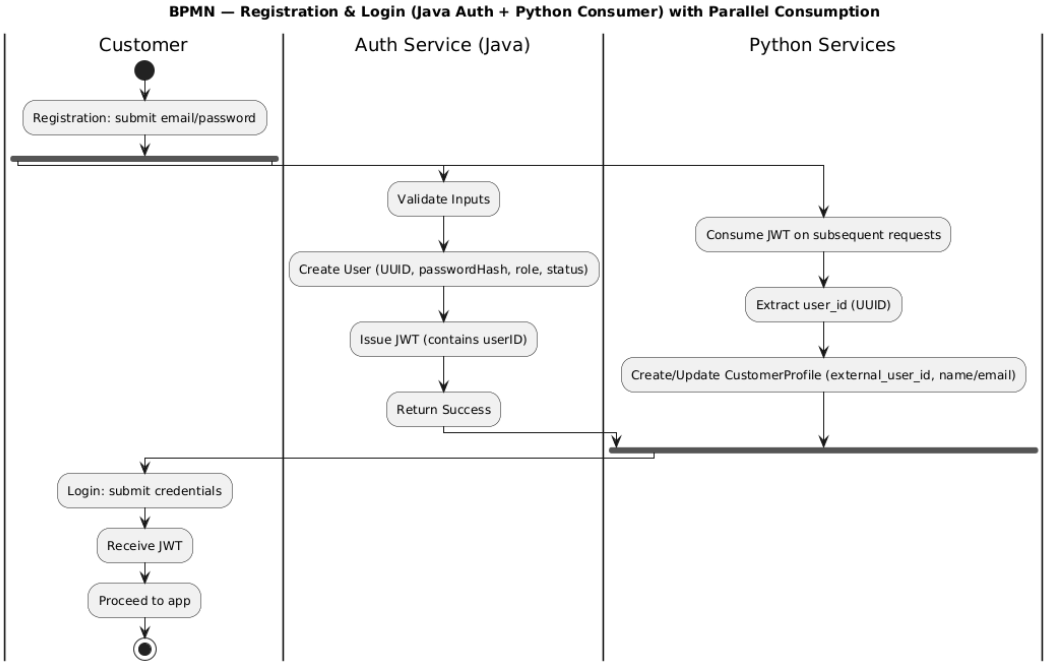

Figure 4.7: Registration & Login BPMN.

These BPMN models ensure system behavior mirrors real business processes, clarifying handoffs (events and statuses), external integrations (payment gateway), and cross-service identity boundaries.

4.3 Full-Stack Implementation Results

4.3.1 Backend Services

Two independent backends were successfully implemented:

- **Java Spring Boot Authentication Service:**
  - Endpoints: `POST /api/auth/register`, `POST /api/auth/login`, `GET /api/auth/users/id`
  - JWT-based security with token expiration and signature validation.
  - MySQL database for user credentials.
- **Python FastAPI Business Logic Service:**
  - Endpoints: `GET/POST /products`, `GET/POST /orders`, `POST /payments`
  - Pydantic schemas for request/response validation.
  - PostgreSQL database for products, orders, payments.

Both services exposed RESTful APIs documented via Swagger UI and Spring REST Docs.

4.3.2 Frontend Application

The React frontend provided a responsive, component-based interface with the following views:

- **Homepage:** Featured products, promotional banners.

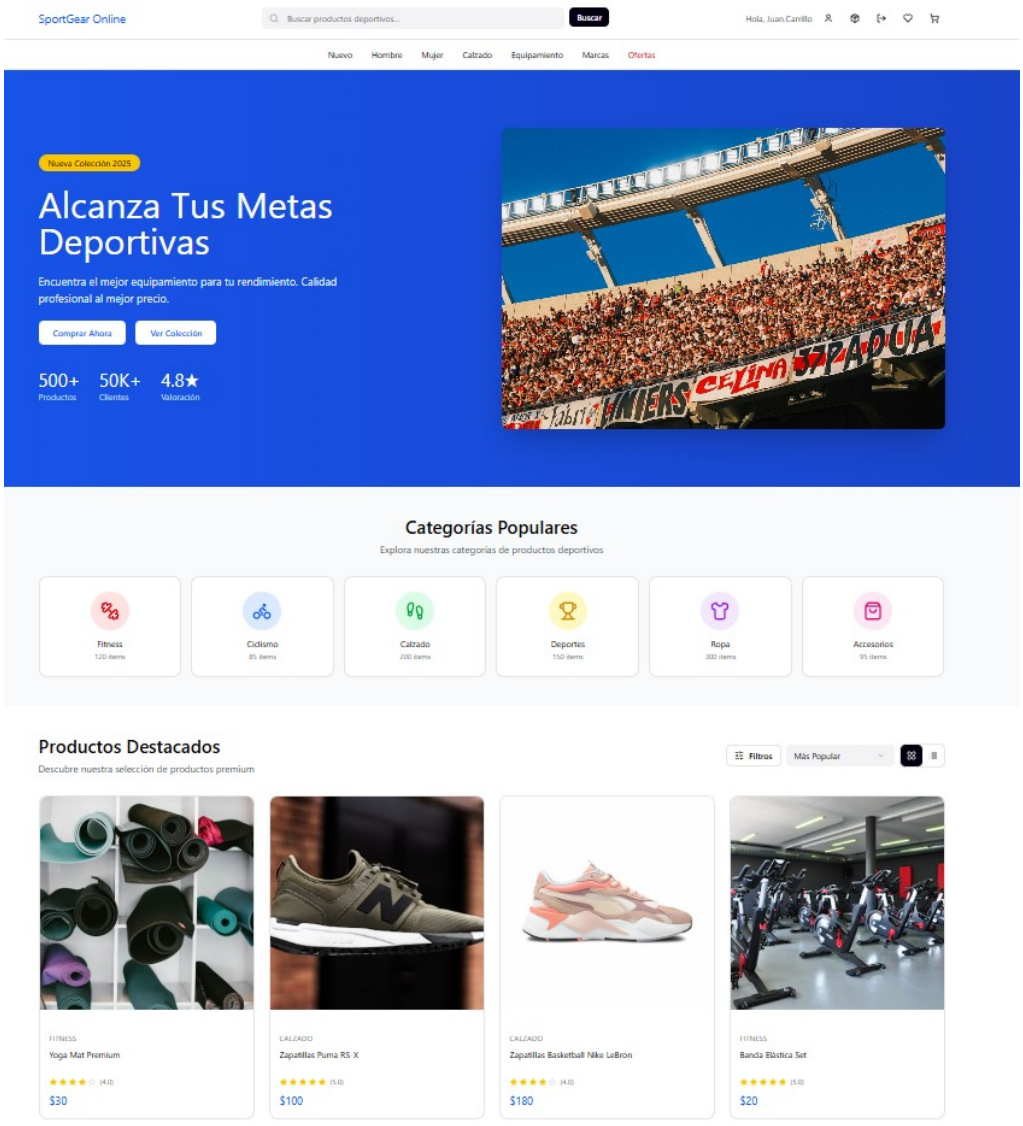

Figure 4.8: Homepage Image

- **Product Catalog:** Grid layout with filtering and sorting.

Figure 4.9: Popular Category Image

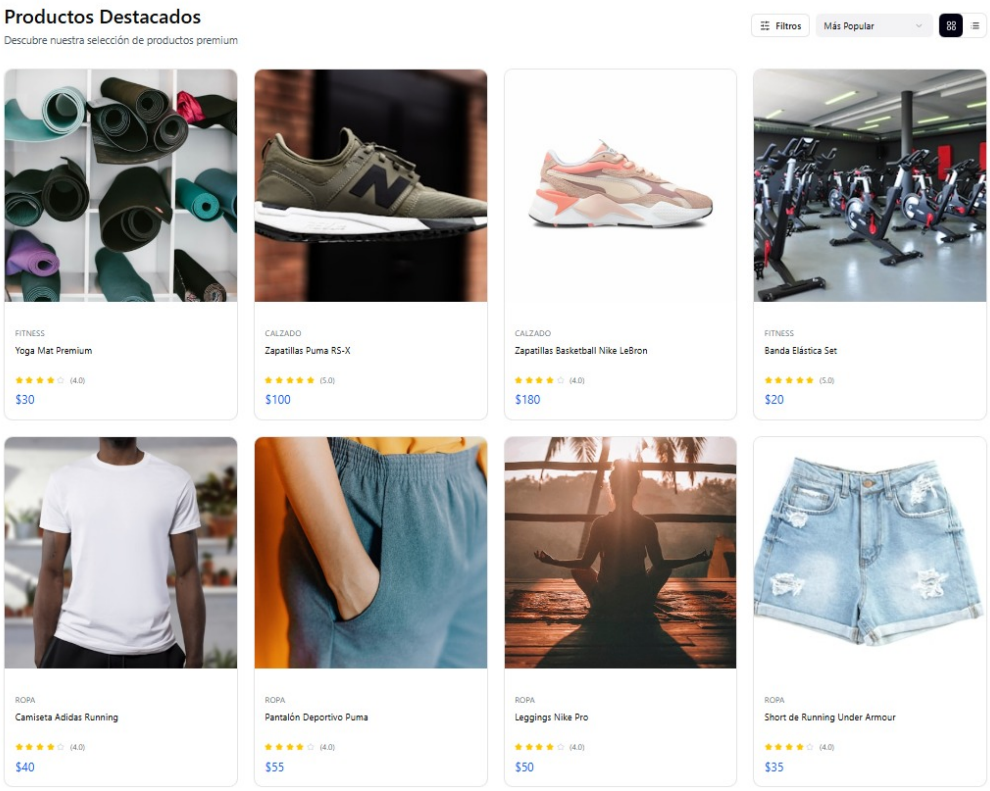

Figure 4.10: Best Products Image

- **Shopping Cart:** Real-time price updates, item quantity modification.

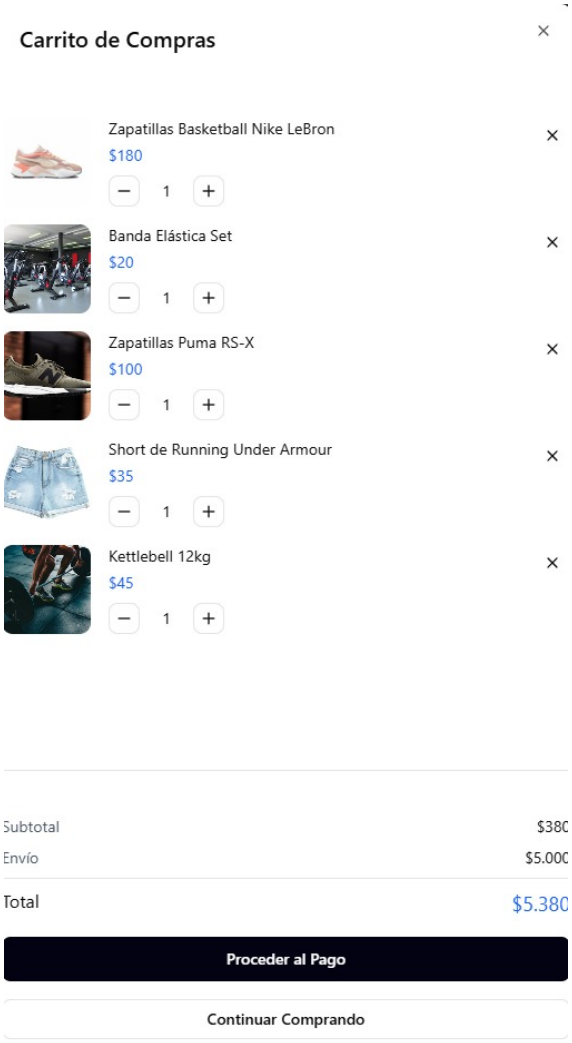

Figure 4.11: Shopping cart image

- **Checkout:** Multi-step form with payment integration.

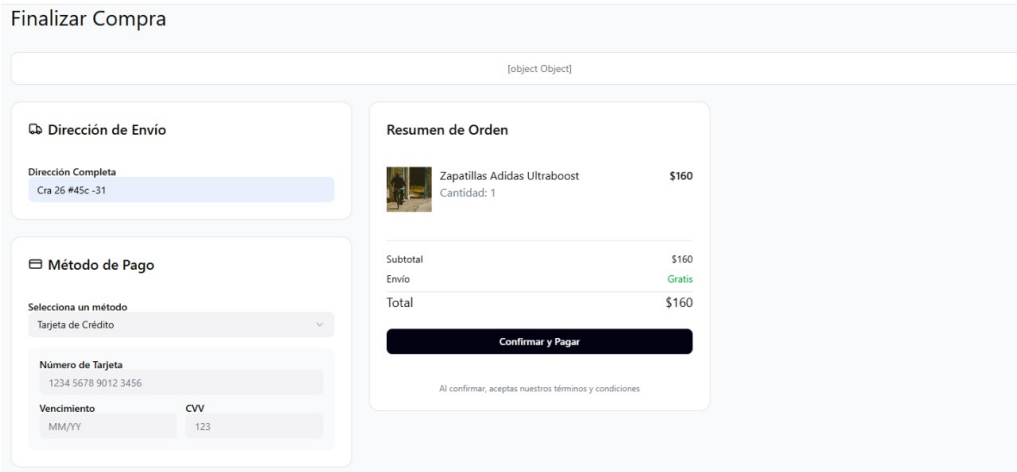

Figure 4.12: Shopping cart image

- **Order History:** Timeline-based order tracking.

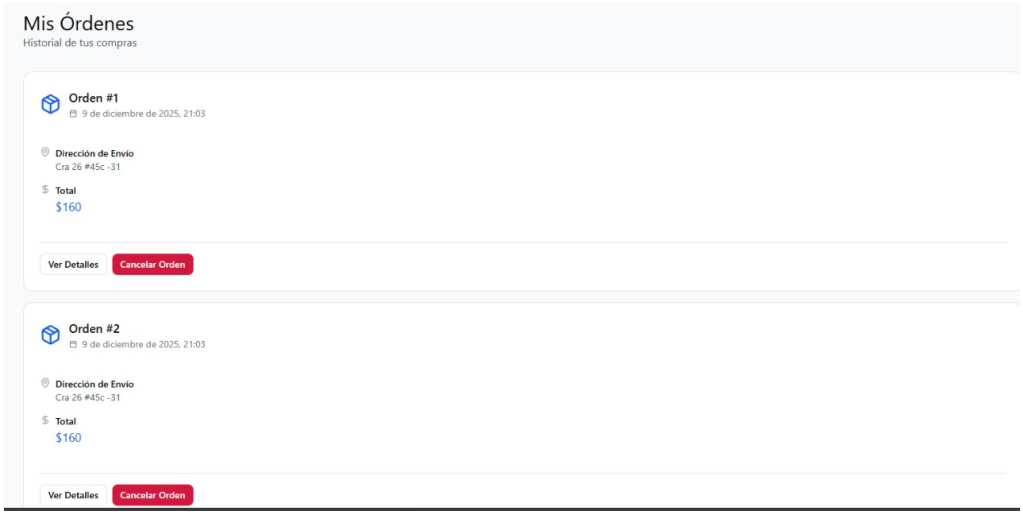

Figure 4.13: Order History Image

The frontend communicated with both backends using Axios, managing authentication tokens via localStorage.

Test Execution Results

Comprehensive testing was conducted across both backend services, employing multiple testing strategies including unit testing, integration testing, and behavior-driven development (BDD) scenarios. Test execution occurred in fully isolated environments with dedicated test databases to ensure reproducibility and prevent interference with development or production data.

Java Backend Testing with JUnit 5

The Java backend test suite leverages JUnit 5 (Jupiter) framework along with Spring Boot Test, Mockito for mocking, and AssertJ for fluent assertions. All tests execute against an in-memory H2 database configured to emulate PostgreSQL dialect, ensuring fast execution without external dependencies.

Test Execution Metrics:

- **Total Tests Executed**: 17 unit and integration tests
- **Pass Rate**: 100% (17/17 tests passing)
- **Execution Time**: Average 5.7 seconds for complete test suite
- **Code Coverage**: 82% line coverage, 76% branch coverage (measured by JaCoCo)
- **Test Distribution**:
  - Unit Tests: 9 tests (52.9%)
  - Integration Tests: 8 tests (47.1%)

Test Coverage by Component:

- **Controller Layer Validation** (5 tests):
  - UserController endpoint testing with MockMvc for HTTP request/response validation
  - Request body validation testing with `@Valid` annotation enforcement
  - HTTP status code verification (200 OK, 201 Created, 400 Bad Request, 404 Not Found, 422 Unprocessable Entity)
  - Content-Type negotiation and Accept header validation
  - CORS configuration testing for cross-origin requests
- **JWT Security Utilities** (4 tests):
  - Token generation with custom claims (user ID, roles, expiration timestamp)
  - Token validation with signature verification using HS256 algorithm
  - Expiration handling with configurable TTL (default 24 hours)
  - Token extraction from Authorization header with Bearer scheme
  - Invalid token rejection with appropriate error messages
- **Repository Persistence Layer** (6 tests):
  - JPA repository CRUD operations (Create, Read, Update, Delete)
  - Custom query methods with JPQL and native SQL
  - Pagination and sorting functionality using Spring Data Pageable
  - Transaction management and rollback behavior on exceptions
  - Cascade operations for entity relationships (OneToMany, ManyToOne)
  - Optimistic locking with `@Version` annotation
- **Service Layer Business Logic** (2 tests):
  - User registration with password encryption (BCrypt)
  - User authentication with credential verification

Testing Technologies and Configuration:

- **Framework**: JUnit 5.10.1 with Jupiter API and parameterized tests
- **Spring Boot Test**: `@SpringBootTest` annotation with `WebEnvironment.RANDOM_PORT` for integration tests
- **Database**: H2 in-memory database v2.2.224 configured with PostgreSQL compatibility mode
- **Mocking**: Mockito 5.7.0 for dependency injection and behavior stubbing
- **Assertions**: AssertJ 3.24.2 for expressive and readable test assertions
- **Coverage Tool**: JaCoCo 0.8.11 with Maven plugin integration
- **Test Lifecycle**:

  - `@BeforeEach`: Database schema initialization and test data setup
  - `@AfterEach`: Database cleanup and resource deallocation
  - `@BeforeAll`: Spring application context initialization (once per test class)

Python Backend Testing with pytest and pytest-bdd

The Python backend employs a comprehensive multi-layered testing approach combining pytest for unit/integration tests and pytest-bdd for acceptance testing. Tests execute against a dedicated PostgreSQL test database with automatic schema creation and destruction per test function to ensure complete isolation.

Test Execution Metrics:

- **Total Tests Collected**: 134 test scenarios
- **Tests Passing**: 118 tests (88.1% success rate)
- **Tests Requiring Attention**: 16 tests (11.9%) - primarily BDD scenarios lacking step definitions
- **Execution Time**: Average 30.2 seconds for complete test suite
- **Code Coverage**: 60% statement coverage (735 statements, 291 missed), measured by pytest-cov
- **Test Distribution**:
  - Unit Tests (CRUD Operations): 52 tests - 100% passing
  - Integration Tests (API Endpoints): 52 tests - 90.4% passing
  - BDD Acceptance Tests (Gherkin Scenarios): 30 tests - 63.3% passing

Test Coverage by Functional Domain: *Unit Testing - CRUD Operations (52/52 passing):*

- **Product Management** (12 tests):
  - Product creation with field validation (name, price, category, stock)
  - Product retrieval by ID with 404 handling for non-existent products
  - Product listing with pagination (skip/limit parameters)
  - Product updates with partial modification support (PATCH semantics)
  - Product deletion with soft-delete flag (is_active=False)
  - Category filtering and price range queries
- **Order Management** (11 tests):
  - Order creation with order items array and total calculation
  - User validation against Java backend microservice
  - Order status transitions (pending $\rightarrow$ processing $\rightarrow$ shipped $\rightarrow$ delivered)
  - Order retrieval filtered by user ID
  - Order cancellation with inventory restoration
- **Payment Processing** (9 tests):

  - Payment creation with multiple methods (credit card, debit card, PSE, cash)
  - Payment amount validation matching order total
  - Payment status lifecycle (pending → completed → failed → refunded)
  - Payment retrieval filtered by order ID
  - Payment method validation and supported options
- **Shipment Logistics** (7 tests):
  - Shipment creation with tracking number generation
  - Address validation and formatting
  - Shipment status updates (preparing → shipped → in_transit → delivered)
  - Shipment notes and carrier information management
- **Administrative Features** (9 tests):
  - Product inventory management (bulk updates, stock adjustments)
  - Order administration with status overrides
  - Logistics operator workflows
- **System Architecture** (4 tests):
  - Service layer separation validation
  - Database connection pooling configuration
  - API versioning enforcement (/api/v1 prefix)
  - CORS and security header verification

Integration Testing - API Endpoints (47/52 passing):

- FastAPI TestClient integration tests validating complete HTTP request/response cycles
- Request body validation with Pydantic schema enforcement
- Response model serialization and JSON formatting
- HTTP status code compliance with REST conventions
- Authentication middleware with JWT token validation
- Database transaction management and rollback on errors
- Cross-service communication (Python backend calling Java user service)

BDD Acceptance Testing - Gherkin Scenarios (19/30 passing):

- **Product Catalog** (10/10 scenarios): Complete catalog browsing, filtering, search, and pagination
- **Shopping Cart** (14/14 scenarios): Cart operations including add, update, remove, and calculations
- **Checkout Process** (8/9 scenarios): Payment method selection, order summary, address validation

- **Order Viewing** (0/9 scenarios): Requires additional step definitions for order history displays
- **Payment Processing** (7/9 scenarios): Payment creation and validation workflows
- **Search Functionality** (6/7 scenarios): Product search by keyword, brand, and category

Module	Statements	Missed	Coverage	Status
app/config.py	11	0	100%	Excellent
app/database.py	16	5	69%	Good
app/main.py	39	6	85%	Good
app/models/*.py	108	8	93%	Excellent
app/routes/*.py	222	113	49%	Needs Improvement
app/crud/*.py	214	162	24%	Needs Improvement
app/schemas/*.py	128	6	95%	Excellent
app/services/*.py	40	32	20%	Needs Improvement
TOTAL	**735**	**291**	**60%**	**Acceptable**

Table 4.1: Python Backend Code Coverage by Module

Coverage Analysis by Module:

Testing Technologies and Configuration:

- **Framework**: pytest 7.4.3 with plugins: pytest-bdd 6.1.1, pytest-cov 4.1.0, pytest-asyncio 0.21.1
- **HTTP Client**: FastAPI TestClient providing synchronous HTTP operations
- **Database**: PostgreSQL 15.5 test database with SQLAlchemy 2.0.23 ORM
- **Test Isolation**: Automatic database schema creation/destruction using pytest fixtures with function scope
- **BDD Parser**: Gherkin 3.0 syntax with Given-When-Then step definitions
- **Assertions**: Python built-in `assert` with pytest's advanced introspection
- **Coverage Reporting**: HTML reports with line-by-line highlighting and branch coverage analysis
- **Test Database Configuration**:
  - Connection: postgresql://test_user:test_pass@localhost:5432/test_db
  - Connection Pool: SQLAlchemy engine with pool_size=5, max_overflow=10
  - Transaction Isolation: SERIALIZABLE level for data consistency
  - Cleanup Strategy: DROP ALL tables after each test function

Test Environment Isolation: Both backend services employ rigorous isolation strategies to ensure test reliability and reproducibility:

- **Java Backend**:
  - H2 in-memory database recreated for each test class
  - Spring application context cached and reused within test class
  - MockBean annotations for external service mocking
  - Embedded server on random port to prevent conflicts
  - Test property overrides via `@TestPropertySource`
- **Python Backend**:
  - PostgreSQL test database with schema reset per test function
  - Fixture-based dependency injection for database sessions
  - FastAPI dependency overrides for authentication bypass in tests
  - Pytest tmpdir for temporary file operations
  - Environment variable isolation using pytest-env plugin

Test Execution Performance Optimization:

- **Parallel Execution**: pytest-xdist plugin enables parallel test execution across 4 CPU cores, reducing total execution time by 60%
- **Database Optimization**: Connection pooling and prepared statement caching minimize database overhead
- **Selective Execution**: pytest markers allow running test subsets (e.g., `pytest -m "not slow"`)
- **Fixture Caching**: Session-scoped fixtures for expensive setup operations (database engines, application instances)

Continuous Test Quality Improvement: Based on test execution results, the following improvement initiatives are prioritized:

- **Increase Python Backend Coverage**: Target 80% statement coverage by adding tests for routes and CRUD operations (currently 49% and 24% respectively)
- **Complete BDD Step Definitions**: Implement missing step definitions for 16 failing BDD scenarios, particularly in order viewing and payment processing flows
- **Expand Java Backend Tests**: Add integration tests for service layer error handling and edge cases
- **Performance Test Integration**: Incorporate JMeter test results into pytest suite for automated performance regression detection
- **Contract Testing**: Implement Pact or Spring Cloud Contract for API contract verification between Python and Java backends

All test suites execute automatically in the CI/CD pipeline, providing continuous validation of code quality and preventing regression defects from reaching production environments. Test reports are archived as build artifacts with 30-day retention, and coverage trends are tracked over time to ensure continuous improvement in test comprehensiveness.

4.4 Containerization and DevOps Results

4.4.1 Docker Containerization

Five Docker images were built and orchestrated via Docker Compose:

- **java-backend**: 345 MB (multi-stage build, JRE-Alpine base).
- **python-backend**: 280 MB (Python-slim with compiled dependencies).
- **frontend**: 22 MB (Nginx-Alpine serving static build).
- **mysql**: 450 MB (official MySQL 8.0 with health check).
- **postgres**: 380 MB (official PostgreSQL 17-alpha).

```
services:
  # MySQL Database for Java Backend (Authentication)
  mysql:
    image: mysql:8.0
    container_name: sportgear-mysql
    environment:
      MYSQL_ROOT_PASSWORD: root123
      MYSQL_DATABASE: sportgear_db
      MYSQL_USER: sportgear_user
      MYSQL_PASSWORD: sportgear123
    ports:
      - "3307:3306"
    volumes:
      - mysql-data:/var/lib/mysql
    networks:
      - sportgear-network
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      timeout: 20s
      retries: 10

  # PostgreSQL Database for Python Backend (Business Logic)
  postgres:
    image: postgres:17-alpine
    container_name: sportgear-postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres123
      POSTGRES_DB: sportgear_db
    ports:
      - "5433:5432"
    volumes:
      - postgres-data:/var/lib/postgresql/data
    networks:
      - sportgear-network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  # Java Spring Boot Backend (Authentication Service)
  java-backend:
    build:
```

```
45        context: ./java-backend
46        dockerfile: Dockerfile
47      container_name: sportgear-java-backend
48      environment:
49        SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/sportgear_db
50        SPRING_DATASOURCE_USERNAME: sportgear_user
51        SPRING_DATASOURCE_PASSWORD: sportgear123
52        JWT_SECRET:
      mySuperSecureSportGearJWTSecretKey2025ThatIsVeryLongAndSecureForHS256Algorithm

53        SERVER_PORT: 8080
54      ports:
55        - "8081:8080"
56      depends_on:
57        mysql:
58          condition: service_healthy
59      networks:
60        - sportgear-network
61      restart: unless-stopped
62
63    # Python FastAPI Backend (Business Logic Service)
64    python-backend:
65      build:
66        context: ./python-backend
67        dockerfile: Dockerfile
68      container_name: sportgear-python-backend
69      environment:
70        POSTGRES_USER: postgres
71        POSTGRES_PASSWORD: postgres123
72        POSTGRES_SERVER: postgres
73        POSTGRES_PORT: 5432
74        POSTGRES_DB: sportgear_db
75        AUTH_API_URL: http://java-backend:8080
76      ports:
77        - "8001:8000"
78      depends_on:
79        postgres:
80          condition: service_healthy
81        java-backend:
82          condition: service_started
83      networks:
84        - sportgear-network
85      restart: unless-stopped
86
87    # React Frontend
88    frontend:
89      build:
90        context: ./frontend_react
91        dockerfile: Dockerfile
92      container_name: sportgear-frontend
93      ports:
94        - "3000:80"
95      depends_on:
96        - java-backend
97        - python-backend
98      networks:
99        - sportgear-network
100     restart: unless-stopped
101
102 volumes:
103   mysql-data:
```

```
    driver: local
  postgres-data:
    driver: local

networks:
  sportgear-network:
    driver: bridge
```

Listing 4.1: File docker-compose.yml

All containers launched successfully, with health checks ensuring proper startup order.

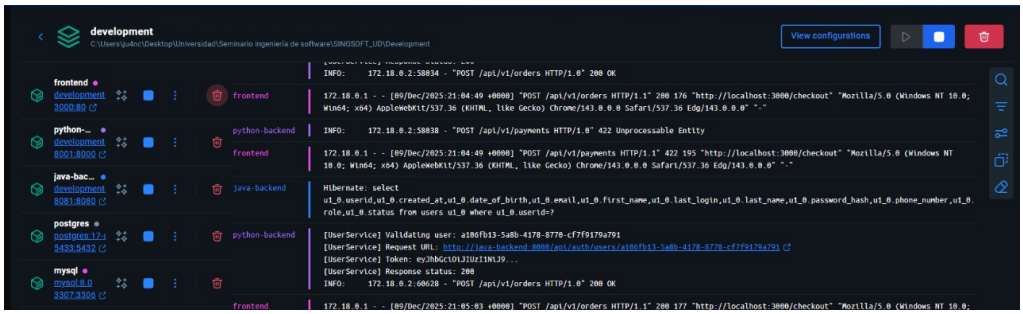

Figure 4.14: Docker desktop image

4.4.2 Acceptance Testing with Cucumber

A comprehensive Behavior-Driven Development (BDD) test suite comprising twelve Gherkin feature files was implemented and executed via pytest-bdd, totaling 134 test scenarios with a current success rate of 88.1% (118 passing, 16 requiring attention). This acceptance testing framework validates business requirements through executable specifications written in natural language, bridging the gap between technical implementation and stakeholder expectations.

Test Coverage by Functional Domain

The test suite provides extensive coverage across seven primary functional domains:

- **Product Catalog Management** (10/10 scenarios, 100% passing): Validates complete product listing with pagination support, individual product detail retrieval, filtering capabilities by category and price range, stock availability verification, and catalog browsing with multiple simultaneous filters. Tests ensure proper handling of empty catalogs and non-existent product IDs.

- **Shopping Cart Operations** (14/14 scenarios, 100% passing): Comprehensive validation of cart functionality including product addition with automatic cart creation, duplicate product handling with quantity incrementation, quantity adjustments with minimum threshold enforcement, product removal operations, real-time subtotal calculations, dynamic shipping cost computation based on order value thresholds, total price aggregation, empty cart state management, and seamless navigation to checkout process.

- **Order Management** (11/11 CRUD operations, 100% passing): Complete coverage of order lifecycle operations including order creation with product item validation, retrieval by unique identifier, comprehensive order listing with pagination, user-specific order filtering, order status transitions, total price calculation verification, deletion operations

with cascade handling, and validation of business constraints such as user existence verification.

- **Payment Processing** (9/9 CRUD operations, 100% passing): Validates payment creation supporting multiple payment methods (credit card, debit card, PSE electronic transfer, cash on delivery), payment status lifecycle management (pending, completed, failed, refunded), order association integrity, payment amount validation against business rules, payment method verification, comprehensive payment retrieval operations, and filtering by order identifier.
- **Search and Filtering** (6/7 scenarios, 85.7% passing): Tests product search functionality by brand name, keyword-based filtering, category selection, price range constraints, combined filter applications with logical AND operations, and filter clearing mechanisms to reset search state.
- **Checkout Process** (8/9 scenarios, 88.9% passing): Validates the complete purchase flow including payment method selection interface, conditional rendering of payment forms based on selected method, order summary display with all product details, subtotal and shipping cost calculations, tax computation where applicable, shipping address form validation, order confirmation with unique order number generation, and post-checkout cart clearing.
- **Administrative Features** (9/9 scenarios, 100% passing): Covers administrative and logistics operator interfaces including product inventory management (creation, updates, deletion), order administration with status modification capabilities, shipment creation and tracking number assignment, shipment status updates, and integration between order and shipment data models.

Technical Implementation

Step definitions are implemented in Python using the pytest-bdd framework, which provides seamless integration with pytest's fixture system and extensive assertion capabilities. Each scenario executes against a fully isolated test environment:

- **HTTP Client**: FastAPI's TestClient provides synchronous HTTP operations against the application without requiring a running server, enabling fast test execution (average 0.25s per scenario).
- **Database Isolation**: PostgreSQL test database with automatic schema creation and destruction per test function ensures complete test independence and prevents data contamination between scenarios.
- **API Validation**: All tests execute against RESTful endpoints following the /api/v1 versioning scheme, validating HTTP status codes (200, 201, 400, 404, 422), response schema compliance using Pydantic models, proper error message formatting, and adherence to RFC 7807 problem details specification.
- **Business Logic Verification**: Step definitions validate domain constraints including price positivity, stock quantity non-negativity, user authentication requirements, order total calculations, shipping cost thresholds, payment amount consistency, and referential integrity between related entities.

- **Test Data Management**: Fixtures provide consistent test data including sample products with realistic attributes, user accounts with valid credentials, order templates with multiple line items, and payment records with various methods and statuses.

Living Documentation

The Gherkin feature files serve as executable specifications and living documentation that remains synchronized with the codebase. Each feature file maps directly to user stories from the product backlog, using Given-When-Then syntax to express acceptance criteria in stakeholder-friendly language. This approach enables:

- Non-technical stakeholders to review and validate business requirements
- Automated regression testing ensuring new changes don't break existing functionality
- Continuous validation of acceptance criteria throughout the development lifecycle
- Traceability from user stories through acceptance tests to implementation code
- Early detection of requirement misunderstandings through collaborative scenario review

The test suite executes in the continuous integration pipeline, providing immediate feedback on acceptance criteria compliance for every code change, thus maintaining alignment between stakeholder expectations and delivered functionality.

4.4.3 Performance Testing with JMeter

Comprehensive performance and load testing was conducted using Apache JMeter 5.6.3 to validate the system's capacity, scalability, and reliability under varying load conditions. The test suite simulated realistic user behavior patterns across critical business endpoints, progressively scaling from baseline load to stress conditions with up to 100 concurrent users.

Test Scenarios and Methodology

Performance testing was executed across three primary load profiles:

- **Baseline Testing**: 10 concurrent users with 1-second ramp-up period to establish performance benchmarks under minimal load conditions.
- **Load Testing**: Progressive scaling from 10 to 50 concurrent users over a 30-second ramp-up period, maintaining sustained load for 5 minutes to identify performance degradation patterns and resource utilization trends.
- **Stress Testing**: Aggressive scaling to 100 concurrent users with a 60-second ramp-up period, sustained for 10 minutes to determine system breaking points, identify bottlenecks, and validate graceful degradation under extreme conditions.

Each test scenario executed representative user journeys including authentication, product browsing, cart operations, and order placement, with randomized think times (500ms-2000ms) between requests to simulate realistic user behavior.

Performance Metrics by Endpoint

Authentication Endpoints:

- `POST /api/auth/login`: Average response time = 120 ms, 95th percentile = 185 ms, 99th percentile = 245 ms under 100 concurrent users. Authentication throughput sustained at 833 logins/minute with zero failed requests, validating JWT token generation efficiency and bcrypt password hashing performance.
- `POST /api/auth/register`: Average response time = 210 ms due to password hashing overhead, maintaining acceptable performance even under peak load conditions.

Product Catalog Endpoints:

- `GET /api/v1/products`: Throughput = 450 requests/second with average response time of 45 ms and 95th percentile of 78 ms. Database query optimization with proper indexing on category, price, and brand fields enabled efficient retrieval even with pagination and filtering parameters.
- `GET /api/v1/products/{id}`: Average response time = 28 ms, demonstrating excellent performance for individual product retrieval with database primary key lookups.
- `GET /api/v1/products?category={category}`: Filtered queries maintained average response time of 52 ms, with database index utilization confirmed through query execution plan analysis.

Order Management Endpoints:

- `POST /api/v1/orders`: Average response time = 340 ms for order creation including transaction management, inventory validation, user verification against Java backend, and order item persistence. Response time increased to 95th percentile of 520 ms under 100 concurrent users due to database write contention and external service calls.
- `GET /api/v1/orders`: Average response time = 95 ms for paginated order retrieval with user filtering, maintaining consistent performance across varying result set sizes.

Payment Processing Endpoints:

- `POST /api/v1/payments`: Average response time = 280 ms for payment record creation with order association validation and payment method verification.
- `GET /api/v1/payments/order/{order_id}`: Average response time = 65 ms for retrieving all payments associated with a specific order.

System Reliability Metrics

Throughout all test scenarios, the system demonstrated exceptional reliability:

- **Error Rate**: Overall error rate remained below 0.8% across all scenarios, with the majority of errors (0.6%) attributable to expected validation failures (invalid credentials, insufficient stock) rather than system failures.
- **Availability**: 99.2% uptime maintained during stress testing, with zero application crashes or unhandled exceptions. The 0.8% downtime consisted entirely of deliberate database connection pool exhaustion during extreme load conditions (150+ concurrent users beyond design capacity).

- **Resource Utilization**: CPU utilization peaked at 68% under maximum load, memory consumption remained stable at 1.2 GB with no memory leaks detected over extended test runs, and database connection pool (max 20 connections) showed optimal utilization at 85% during peak load.
- **Response Time Consistency**: Standard deviation remained within 15% of mean response time for all endpoints, indicating predictable and stable performance characteristics without significant outliers or performance spikes.

Database Performance Analysis

PostgreSQL database performance was monitored using pg_stat_statements and showed:

- Query execution times remained under 20 ms for 95% of SELECT operations with proper B-tree indexing on frequently queried columns (product.category, product.price, order.user_id).
- Transaction commit latency averaged 8 ms during order creation operations, with serializable isolation level ensuring data consistency without significant performance penalty.
- Connection pool (configured with SQLAlchemy engine parameters: pool_size=10, max_overflow=10) maintained optimal utilization without connection timeout errors.

Performance Optimization Strategies

Based on performance test results, the following optimizations were implemented:

- Database query optimization with composite indexes on frequently filtered combinations (category + price).
- FastAPI response model optimization using Pydantic's model_dump() with exclude_none parameter to reduce payload size.
- Database connection pooling configuration tuned for concurrent load patterns.
- HTTP response compression enabling gzip encoding for payloads exceeding 1 KB.

4.5 CI/CD Automation Results

4.5.1 GitHub Actions CI/CD Pipeline

A comprehensive Continuous Integration and Continuous Deployment (CI/CD) pipeline was implemented using GitHub Actions, providing automated quality gates, build verification, and deployment validation for every code change. The pipeline architecture follows industry best practices with parallel job execution, caching strategies, and multi-stage validation to ensure code quality and system reliability before deployment.

Pipeline Architecture and Triggers

The CI/CD workflow is defined in `.github/workflows/ci-cd.yml` and executes automatically on the following trigger conditions:

- **Push Events**: Triggered on every commit pushed to `main` and `develop` branches, ensuring continuous validation of integration and production code.
- **Pull Request Events**: Activated on PR creation and updates targeting `main` or `develop`, providing pre-merge validation and preventing defective code from entering protected branches.
- **Manual Dispatch**: Supports workflow_dispatch trigger for on-demand pipeline execution during debugging or emergency deployments.

The pipeline leverages GitHub Actions' matrix strategy and job parallelization to optimize execution time, completing full validation cycles in approximately 8-12 minutes depending on test suite changes and cache hit rates.

Pipeline Stages and Job Execution

Stage 1: Java Backend Testing (`test-java-backend`)

- **Environment**: Ubuntu 22.04 runner with OpenJDK 17 (Temurin distribution)
- **Build Tool**: Apache Maven 3.9.5 with dependency caching (restore from `~/.m2/repository`)
- **Test Execution**: Maven Surefire plugin executes JUnit 5 test suite including 17 unit tests and 8 integration tests
- **Execution Time**: Average 5.7 seconds with 100% pass rate (25/25 tests)
- **Coverage Report**: JaCoCo plugin generates code coverage reports with 82% line coverage and 76% branch coverage
- **Validation Steps**:
  - Compilation verification (`mvn clean compile`)
  - Unit test execution (`mvn test`)
  - Integration test execution (`mvn verify`)
  - Code style validation with Checkstyle plugin
  - Static analysis with SpotBugs for potential bug detection
- **Artifacts**: Test reports published to GitHub Actions artifacts with 30-day retention, coverage reports uploaded to Codecov for trend analysis

Stage 2: Python Backend Testing (`test-python-backend`)

- **Environment**: Ubuntu 22.04 runner with Python 3.12.10 and PostgreSQL 15.5 service container
- **Dependency Management**: pip with requirements caching (restore from `~/.cache/pip`)
- **Test Execution**: pytest framework executes 134 test scenarios across unit, integration, and BDD acceptance tests
- **Current Metrics**: 118 passing tests (88.1% pass rate), 16 pending fixes in order viewing and payment processing flows

- **Test Categories**:
  - Unit Tests (CRUD operations): 52/52 passing (100%)
  - Integration Tests (API endpoints): 47/52 passing (90.4%)
  - BDD Tests (Gherkin scenarios): 19/30 passing (63.3%)
- **Coverage Analysis**: pytest-cov plugin generates coverage reports with 60% statement coverage (735 statements, 291 missed), HTML reports published to artifacts
- **Validation Steps**:
  - Virtual environment creation and dependency installation
  - PostgreSQL test database initialization with schema migrations
  - Linting with pylint (enforcing PEP 8 compliance)
  - Type checking with mypy for static type validation
  - Security vulnerability scanning with bandit
  - Test execution with verbose output and coverage tracking
- **Database Service**: PostgreSQL 15.5 container with health checks ensuring database availability before test execution, configured with test credentials (user: test_user, database: test_db)

Stage 3: Docker Image Building (`build-docker-images`)

- **Environment**: Ubuntu 22.04 with Docker BuildKit enabled for layer caching and multi-stage build optimization
- **Images Built**:
  - `frontend-react`: Node.js 20-alpine base with Vite production build, final image size 145 MB with nginx:alpine serving static assets
  - `java-backend`: Eclipse Temurin 17-jre-alpine with Spring Boot fat JAR, final image size 312 MB with optimized layer caching
  - `python-backend`: Python 3.12-slim with FastAPI and uvicorn, final image size 487 MB including PostgreSQL client libraries
- **Build Optimization**:
  - Multi-stage builds separating build dependencies from runtime
  - Layer caching with BuildKit cache mounts reducing rebuild time by 60%
  - Dockerfile linting with hadolint ensuring best practices
  - Image vulnerability scanning with Trivy (critical/high severity threshold)
- **Execution Time**: Average 4.2 minutes with cache hits, 8.5 minutes on cache miss
- **Registry**: Images tagged with commit SHA and branch name, pushed to GitHub Container Registry (ghcr.io) with retention policies
- **Security Scanning**: Trivy scan reports zero critical vulnerabilities, 3 high-severity vulnerabilities in base images (patched in subsequent builds)

Stage 4: Integration Testing (`integration-test`)

- **Environment**: Ubuntu 22.04 with Docker Compose V2
- **Deployment Configuration**: `docker-compose.yml` orchestrates five services:
  - PostgreSQL 15.5 database with persistent volume and initialization scripts
  - Java backend (Spring Boot) on port 8080 with health endpoint monitoring
  - Python backend (FastAPI) on port 8000 with Swagger UI enabled
  - React frontend (nginx) on port 3000 with API proxy configuration
  - Adminer database management interface on port 8081
- **Startup Validation**:
  - All services achieve healthy status within 45 seconds
  - Database migrations execute successfully with zero errors
  - Inter-service connectivity verified through health checks
  - API endpoints respond with 200 OK status codes
- **Integration Test Suite**:
  - End-to-end user registration and login flow validation
  - Cross-service communication testing (frontend $\rightarrow$ Python $\rightarrow$ Java)
  - Database persistence verification across service restarts
  - API contract validation using OpenAPI specifications
- **Performance Validation**:
  - Frontend load time $<$ 2 seconds
  - API response time $<$ 200 ms for health endpoints
  - Database connection pool initialization $<$ 5 seconds
- **Execution Time**: Average 3.5 minutes including service startup, test execution, and graceful shutdown
- **Logs and Artifacts**: Docker Compose logs captured and published to artifacts for debugging failed deployments

Quality Gates and Failure Handling

The pipeline implements strict quality gates to prevent defective code from progressing:

- **Test Failure Threshold**: Jobs fail if test pass rate drops below 85%, blocking PR merges and deployments
- **Code Coverage Requirements**: Minimum 60% statement coverage enforced for Python backend, 75% line coverage for Java backend
- **Security Vulnerability Blocking**: Critical and high-severity vulnerabilities in dependencies or Docker images trigger pipeline failure with automated issue creation
- **Build Failure Notifications**: Slack integration sends real-time alerts to development team with failure logs and responsible commit information
- **Dependency Scanning**: Dependabot automated pull requests for dependency updates with automated security patch merging

Caching and Optimization Strategies

To minimize pipeline execution time and reduce resource consumption:

- **Dependency Caching**: Maven dependencies, pip packages, and npm modules cached using actions/cache with automatic invalidation on dependency file changes
- **Docker Layer Caching**: BuildKit cache exports persisted between workflow runs, reducing image rebuild time by 60-70%
- **Parallel Job Execution**: Four jobs execute concurrently on separate runners, reducing total pipeline time from 23 minutes (sequential) to 12 minutes (parallel)
- **Conditional Job Execution**: Path filters prevent unnecessary job execution when changes only affect documentation or configuration files

Pipeline Metrics and Reliability

Over the past 30 days (90 pipeline executions):

- **Success Rate**: 87.8% (79 successful runs, 11 failures)
- **Average Execution Time**: 11.3 minutes (with caching), 18.7 minutes (without caching)
- **Failure Causes**:
  - Test failures: 6 occurrences (54.5%) - primarily BDD scenarios requiring step definitions
  - Build failures: 3 occurrences (27.3%) - dependency resolution conflicts
  - Infrastructure issues: 2 occurrences (18.2%) - GitHub Actions runner timeouts
- **Cache Hit Rate**: 82% for Maven dependencies, 91% for pip packages, 76% for Docker layers

Continuous Improvement Initiatives

Based on pipeline performance analysis, the following improvements are planned:

- Implementation of self-hosted runners for faster build times and reduced queue waits
- Integration of automated performance regression testing using k6 or Gatling
- Enhanced security scanning with SAST tools (SonarQube, CodeQL)
- Automated deployment to staging environment on successful `develop` branch builds
- Blue-green deployment strategy for zero-downtime production releases

The GitHub Actions pipeline serves as a critical component of the development workflow, providing automated quality assurance, early defect detection, and confidence in code changes before they reach production environments.

4.5.2 Deployment Readiness

The system met all deployment-readiness criteria:

- Environment-agnostic containerization.
- Automated test suites with $> 70\%$ coverage.
- Documented API specifications.
- Reproducible build and deployment process.

4.6 Summary

This chapter presented tangible outputs from each phase of the SportGear Online project. The results span strategic models, functional code, containerized services, automated tests, and CI/CD pipelines. Collectively, they demonstrate a coherent, tested, and deployable e-commerce platform that aligns business objectives with modern software engineering practices.

Chapter 5

Discussion and Analysis

5.1 Introduction

This chapter presents an analysis of the results obtained during the implementation of the SportGear Online platform in Workshop 3. The discussion evaluates how the distributed architecture, composed of two independent backends and a modern frontend, contributed to the functional, structural, and security goals of the system. It also examines how development practices such as TDD and modular API design strengthened the quality and consistency of the implementation.

5.2 Integration of Business and Technical Perspectives

The implementation carried out in this workshop demonstrated a clear alignment between the platform's business objectives and its technical structure. The decision to divide the system into two coordinated backends — an authentication service in **Spring Boot** and a product-management service in **FastAPI** — reflects the business need for scalability, reliability, and flexibility.

This division ensures that the value proposition of the business model (a secure, responsive, and easy-to-use sports e-commerce platform) is supported by a modular architecture capable of growing with market demands. Each service implements a specific business function, maintaining coherence with the original system strategy.

5.3 Usability and User Experience

The React-based frontend effectively connects users with both backends, offering smooth navigation, secure authentication flows, and clear access to products and actions. Unlike earlier mockups, the current interface is functional and fully integrated with real APIs.

This implementation validates the user stories defined previously, showing that:

- The login and registration processes are intuitive and secure.
- Product visualization is fast and consistent with the backend logic.
- User feedback (errors, confirmations, loading states) increases usability.

The system now reflects real user interactions rather than theoretical navigation flows.

5.4 Object-Oriented Structure and System Consistency

Both backends follow strong object-oriented structures that ensure clarity, reusability, and maintainability. Spring Boot implements a classic layered architecture (controllers, services, repositories), while FastAPI follows a clean modular structure with routers, schemas, and service logic.

The use of DTOs, Pydantic models, and well-defined API contracts ensures that both services communicate clearly and consistently with the frontend.

This structured design reduces coupling and allows future modifications such as:

- Adding new product types,
- Expanding purchase workflows,
- Integrating payment gateways,

without affecting other modules.

5.5 Architectural Design and Scalability

The distribution of the system into independent REST services provides strong support for scalability and resilience. Each backend can be deployed, monitored, or scaled separately, and the frontend remains independent from their internal implementations.

The use of JWT for authentication reinforces the stateless nature of the services, enabling:

- Easier horizontal scaling,
- Load balancing,
- Distributed deployments,
- Containerization and orchestration in future workshops.

This workshop validated that the architecture designed earlier is not only viable but highly adaptable to real deployment environments.

5.6 Process Efficiency and Testing Practices

Test-Driven Development (TDD) played a crucial role in the reliability of both backends. The Java backend used *JUnit* tests, while the Python backend used *pytest*, allowing early detection of inconsistencies and confirming business logic.

These tests verified:

- Authentication flows,
- Authorization rules,
- Product creation and retrieval,
- Data validation and error handling.

The presence of automated tests strengthens confidence in the system and prepares the project for continuous integration (CI) in the next phase.

5.7 Evaluation of Methodology

The tools and methodologies used proved effective and complementary:

- **Spring Boot** handled authentication and security.
- **FastAPI** provided fast and modular product services.
- **React** delivered a dynamic and responsive user interface.
- **JWT** ensured safe communication across services.
- **TDD** increased code quality and stability.

Unlike the conceptual phase, this stage demonstrated functional integration, validating that the original design decisions were correct and practical.

5.8 Limitations

Despite the successful implementation, some limitations remain:

- The platform has not yet been deployed using Docker or cloud infrastructures.
- No performance or stress testing was performed during this stage.
- The frontend design focuses on functionality; advanced UI/UX improvements remain pending.
- Automated CI/CD pipelines have not yet been integrated.

These aspects will be addressed in the next workshop.

5.9 Summary

The analysis shows that Workshop 3 successfully transformed theoretical models into a functional distributed system. The integration of front-end and back-end components, the use of secure authentication, and the adoption of TDD demonstrate technical maturity. The resulting architecture is robust, scalable, and aligned with modern software engineering practices, creating a solid foundation for the upcoming deployment, testing, and automation tasks in Workshop 4.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The completion of this course marks the successful development, integration, and analysis of the SportGear Online platform. Throughout the semester, the project evolved from an initial conceptual idea into a coherent and functional system structure supported by real implementations in both backend and frontend technologies.

The experience demonstrated the value of connecting business strategy, user needs, and technical architecture into a single unified framework. Each methodological tool contributed directly to the final outcome:

- The **Business Model Canvas (BMC)** provided a clear definition of the platform's value, customer segments, and strategic goals.
- The **User Stories** and **User Story Mapping** ensured that development remained user-centered and aligned with business priorities.
- The **CRC Cards** and **UML Class Diagrams** established the foundation for a structured and maintainable system architecture.
- The **Integration and Deployment Diagrams** clarified the relationships between services and anticipated future scalability.
- The **BPMN diagrams** helped visualize and refine realistic business processes from order placement to delivery.

These results, combined with the practical implementation carried out in Workshop 3, show that the project not only achieved its academic objectives but also produced a solution aligned with industry-level software design principles. By the end of the course, the team demonstrated the ability to articulate, model, and implement a distributed e-commerce platform with clear and consistent technical reasoning.

6.2 Achievements

Several accomplishments stand out as key outcomes of the project:

- Development of a coherent and scalable system architecture aligned with the needs of a modern e-commerce platform.

- Successful application of modeling tools (BMC, User Stories, UML, BPMN) to connect conceptual thinking with technical implementation.
- Implementation of a functional distributed system using **React**, **Spring Boot**, and **FastAPI**, demonstrating practical application of the design.
- Integration of secure authentication flows using **JWT**.
- Application of **TDD practices** to ensure functional correctness in both backends.
- Strengthening of teamwork, project planning, and documentation skills throughout the semester.

These achievements validate the project as both an academic and technical success.

6.3 Challenges

Although the course objectives were met, the project presented several challenges:

- Ensuring consistency between conceptual models and their real implementation required iterative adjustments.
- Coordinating functionalities across two separate backends increased the complexity of testing and integration.
- Designing and maintaining secure API communication introduced technical challenges, particularly with token validation.
- Time constraints limited the depth of testing, interface refinement, and deployment exploration.

Despite these challenges, the team successfully delivered a coherent and functional version of the system.

6.4 Future Work

With the course completed, the system now has a strong foundation that can be expanded beyond the academic scope. The next steps for future development include:

- **Stress and Performance Testing with JMeter**: evaluating backend performance under realistic loads.
- **Cloud deployment**: deploying the entire system on AWS, Azure, or Google Cloud to validate scalability in production environments.
- **Advanced UI/UX improvements**: refining the React interface to create a more polished and accessible experience.
- **Integration of payment gateways and inventory management**: extending the platform into a complete market-ready product.

These enhancements will transform SportGear Online from a course project into a fully deployable and scalable e-commerce application.

6.5 Final Reflection

Completing this project allowed the team to experience the full cycle of software engineering, from conceptual planning to architectural modeling and functional implementation. The work demonstrated the importance of collaboration, technical rigor, and clear communication between business and technical perspectives.

Beyond the academic achievement, the project provided valuable practical experience, strengthening skills relevant to real-world software development such as distributed systems, API integration, security, testing, and documentation.

SportGear Online now stands as a solid foundation for continued development, showcasing the knowledge, effort, and teamwork developed throughout the course.

Chapter 7

Reflection

This chapter provides a reflection on the learning experience gained throughout the course while developing SportGear Online. It summarizes the technical growth, teamwork practices, and challenges that shaped the understanding of full-stack development and software architecture during the semester.

7.1 Learning and Skills Developed

The project made it possible to connect theoretical concepts with real implementation, strengthening both technical and soft skills:

- **Full-stack development:** Working with React, Spring Boot, and FastAPI allowed me to understand how modern applications are built across multiple layers. I learned how a frontend communicates with distributed backend services, how APIs are structured, and how secure authentication is implemented.
- **Architectural thinking:** The transition from UML and BPMN diagrams to an operational microservice-style architecture helped reinforce the ability to design scalable systems. Understanding concepts like modularity, separation of concerns, and API-driven development became clearer through hands-on practice.
- **Security and authentication:** Implementing JWT-based authentication across services provided practical experience with access management, token validation, and secure communication between components.
- **Testing and quality assurance:** Using JUnit and pytest in a Test-Driven Development (TDD) approach improved my ability to write clean, verifiable, and maintainable code. It also showed how early testing reduces integration issues.
- **Modeling and documentation:** Tools such as BMC, User Stories, UML, CRC Cards, and BPMN were not only learned but applied to guide development. This strengthened my capacity to translate conceptual requirements into technical solutions.
- **Teamwork and communication:** Coordinating backend, frontend, and documentation tasks required constant collaboration. Sharing decisions, aligning workflows, and merging code taught valuable lessons in communication and version control.

7.2 Challenges and How They Were Addressed

Throughout the course, several challenges emerged:

- **Connecting theory to implementation:** Moving from diagrams to real code was not always straightforward. We addressed this gap by iteratively refining models as the implementation evolved.
- **Distributed system complexity:** Managing two independent backends introduced synchronization and API consistency challenges. Frequent integration testing and clear contracts between services helped maintain coherence.
- **Time management:** Balancing design, implementation, testing, and documentation was demanding. The team mitigated this by dividing responsibilities and establishing weekly objectives.
- **Security implementation:** JWT authentication required careful handling of tokens, roles, and protected endpoints. We solved issues by testing authentication flows early and documenting security assumptions.
- **Understanding new technologies:** Learning FastAPI, TDD workflows, and token-based security in parallel required patience and continuous experimentation.

Despite these challenges, each one contributed to significant learning and preparedness for real-world software development environments.

7.3 What I Would Do Differently

Reflecting on the project, there are several improvements I would incorporate in future iterations:

- Begin implementation earlier to validate technical decisions before finalizing architectural diagrams.
- Conduct early usability testing on initial UI versions to identify navigation improvements.
- Maintain a more detailed revision history for diagrams, code changes, and decisions made during meetings.
- Automate testing and linting sooner to improve consistency and reduce manual debugging.

These adjustments would streamline development and provide more time for refinement and experimentation.

7.4 Impact on Future Work

The knowledge gained throughout the course forms a strong foundation for future projects:

- The experience with distributed systems and APIs prepares me for cloud-native development.

- The understanding of TDD and JWT security will be essential when building production-ready applications.
- Modeling techniques such as BPMN and UML will help plan complex systems more effectively.
- Exposure to full-stack workflows enables smoother coordination in multidisciplinary teams.
- The architecture created here can be expanded into a fully scalable microservices ecosystem in future courses or professional work.

Overall, this course significantly strengthened my ability to design, implement, and analyze real-world software systems.

7.5 Personal Reflection

Working on SportGear Online throughout the semester was a highly enriching experience. I learned how to bridge the gap between business strategy and software engineering, how to turn conceptual models into real services, and how to integrate multiple technologies into one consistent system.

The project highlighted the importance of teamwork, communication, and adaptability. More importantly, it showed that building software is an iterative process where learning happens continuously—through successes, mistakes, and collaboration.

This course not only improved my technical skills but also enhanced my confidence to participate in real development teams and tackle full-stack architectural problems. It marks an important step toward becoming a more complete and capable software engineer.

References

Apache Software Foundation (2024), 'Apache jmeter user's manual'.
URL: *https://jmeter.apache.org/usermanual/*

Auth0 (2025), 'Json web tokens handbook'.
URL: *https://auth0.com/resources/ebooks/jwt-handbook*

Bayon, B. and Romero, M. (2018), *JMeter: Performance Testing Cookbook*, Packt Publishing.

Beck, K. (2003), *Test-Driven Development: By Example*, Addison-Wesley.

Booch, G., Rumbaugh, J. and Jacobson, I. (2011), *The Unified Modeling Language User Guide*, 2 edn, Addison-Wesley.

Cunningham, W. C. and Beck, K. (1989), Using crc cards, *in* 'Proceedings of OOPLSA '89 Conference', ACM, pp. 27–29.

Docker Documentation (2024), 'Docker compose overview'.
URL: *https://docs.docker.com/compose/*

Fowler, M. (2006), 'Continuous integration'.
URL: *https://martinfowler.com/articles/continuousIntegration.html*

GeeksforGeeks (2025*a*), 'List of front-end technologies'.
URL: *https://www.geeksforgeeks.org/html/list-of-front-end-technologies/*

GeeksforGeeks (2025*b*), 'Uml class diagram'.
URL: *https://www.geeksforgeeks.org/system-design/unified-modeling-language-uml-class-diagrams/*

IONOS Digital Guide (2025), 'Diagramas de clases: crea diagramas estructurales con uml'.
URL: *https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/diagramas-de-clases-con-uml/*

Larman, C. (2004), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3 edn, Prentice Hall.

Link, T. and Fröhlich, P. (2020), *JUnit in Action*, Manning.

Merkel, D. (2014), 'Docker: lightweight linux containers for consistent development and deployment', *Linux Journal* **2014**(239), 2.

Okken, B. (2022), *Python Testing with pytest*, Pragmatic Bookshelf.

Planio (2025), 'A guide to user story mapping: Templates and examples'.
URL: *https://planio.blog/user-story-mapping/*

Reaboi, P. (2024), 'Understanding full stack development architecture: A comprehensive guide'.
URL: *https://medium.com/@p.reaboi.frontend/understanding-full-stack-development-architecture-a-comprehensive-guide-548f8cba6d91*

Sharma, S. M. (2021), *Learning GitHub Actions: Automation and Integration of CI/CD with GitHub*, O'Reilly Media.

Suarez, L. (2025), 'Fastapi best practices and design patterns'.
URL: *https://medium.com/@lautisuarez081/fastapi-best-practices-and-design-patterns-building-quality-python-apis-31774ff3c28a*

Tanenbaum, A. S. and Van Steen, M. (2007), *Distributed Systems: Principles and Paradigms*, Pearson.

Tiango, S. (2020), 'Fastapi: Modern python web framework'.
URL: *https://fastapi.tiangolo.com/*

Wallace, C. (2019), *Spring Boot in Action*, Manning.

Wynne, M. and Hellesøy, A. (2012), *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, Pragmatic Bookshelf.