Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

European Master in Software Engineering

Master Thesis

A Cloud-Native Microservices Architecture for the Coffee E-Commerce Market

Author: David Esteban Cediel Gómez

June, 2025

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis

European Master in Software Engineering

Title: A Cloud-Native Microservices Architecture for the Coffee E-Commerce Market

June / 2025

Author: David Esteban Cediel Gómez

Supervisor:

María Pilar Rodríguez González

Associate Professor

Universidad Politécnica de Madrid

ETSI Informáticos

Universidad Politécnica de Madrid

Co-supervisor:

Jaime Ramírez Rodríguez

Associate Professor

Universidad Politécnica de Madrid

ETSI Informáticos

Universidad Politécnica de Madrid

Abstract

This thesis presents the full development lifecycle of a cloud-native e-commerce platform specialized in coffee products, which connects local producers directly with consumers. The work includes requirements specification, high-level architectural design, and microservices interaction of a Minimum Viable Product, which was partially designed in detail, developed, and deployed using cloud-native technologies such as Docker and Kubernetes. The project focuses on the back-end system, showing an approach to building a reliable, scalable, and maintainable microservice architecture.

The requirements were captured through user stories and acceptance criteria. The system design was based on widely used software techniques, including Domain-Driven Design and Hexagonal Architecture.

To build a reliable, scalable, and high-performing system, multiple cloud patterns were implemented, such as event-driven communication, CQRS, Saga, and Circuit Breaker. Furthermore, a real-time monitoring application was included to identify performance bottlenecks and issues.

To confirm that the system can handle high loads while maintaining its responsiveness, performance tests were conducted on multiple functionalities of a prototype based on representative use cases.

Overall, the project demonstrates how to build a cloud-native software, from conception to deployment, using the best practices currently employed in the industry.

Resumen

Esta tesis presenta el ciclo de vida completo de desarrollo de una plataforma de comercio electrónico nativa de la nube, especializada en productos de café, que conecta directamente a productores locales con los consumidores. El trabajo incluye la especificación de requisitos, el diseño arquitectónico de alto nivel y la interacción de microservicios de un Producto Mínimo Viable (MVP), el cual fue parcialmente diseñado en detalle, desarrollado e implementado utilizando tecnologías nativas de la nube como Docker y Kubernetes. El proyecto se centra en los sistemas de back-end, mostrando un enfoque para construir una arquitectura de microservicios confiable, escalable y mantenible.

Los requisitos se capturaron mediante historias de usuario y criterios de aceptación. El diseño del sistema se basó en técnicas de software ampliamente utilizadas, incluyendo Diseño dirigido por el dominio y Arquitectura Hexagonal.

Para construir un sistema confiable, escalable y de alto rendimiento, se implementaron múltiples patrones de nube, como comunicación basada en eventos, CQRS, Saga y Circuit Breaker. Además, se incluyó una aplicación de monitoreo en tiempo real para identificar cuellos de botella y problemas de rendimiento.

Para confirmar que el sistema puede manejar cargas elevadas manteniendo su capacidad de respuesta, se realizaron pruebas de rendimiento en múltiples funcionalidades de un prototipo basado en casos de uso representativos.

En general, el proyecto demuestra cómo construir software nativo de la nube, desde su concepción hasta su implementación, utilizando las mejores prácticas empleadas actualmente en la industria.

Table of Contents

1 Introduction 12
1.1 Context and Motivation 12
1.2 Objectives and Scope 13
1.3 Design and Development Approach 14
1.4 Best Practices of Software Engineering 15
1.5 Structure of the document 16
2 Theoretical and Technological Foundations 17
2.1 Software Quality Attributes 17
2.1.1 Reliability 17
2.1.2 Maintainability 17
2.1.3 Scalability 18
2.1.4 Modifiability 19
2.1.5 Performance 19
2.2 Software Paradigms and Architectures 20
2.2.1 Domain-Driven Design 20
2.2.2 Monolithic Architecture 21
2.2.3 Microservices 23
2.2.4 Cloud Native 26
2.3 Data Management Technologies 27
2.3.1 Relational Databases 27
2.3.2 NoSQL Databases 27
2.3.3 Cache Databases 27
2.4 Communication in Distributed Systems 28
2.4.1 HTTPS 28
2.4.2 Asynchronous Messaging 29
2.5 Architectural Patterns 30
2.5.1 Hexagonal Architecture 30
2.5.2 Database per Service 32
2.5.3 CQRS 33
2.5.4 Saga 34
2.5.5 Circuit Breaker 35
2.5.6 Event-Driven Architecture 35
2.5.7 Transactional Outbox 36
2.5.8 Anti-corruption layer 37
2.5.9 Backends for Frontends 37

2.6 Testing Microservices 37
2.6.1 Testcontainers 37
2.6.2 Test-driven development 39
2.6.3 Contract testing 39
2.7 System Observability and Monitoring 41
2.7.1 Health Endpoint Monitoring 41
2.8 Cloud Computing Foundations 42
2.8.1 Introduction to Cloud Computing 42
2.8.2 Azure Ecosystem 42
2.8.3 Azure Services 42
3 Requirements Specification 44
3.1 Product Characteristics 45
3.1.1.1 Product Category 45
3.1.1.2 Tags 45
3.1.1.3 Product Variations 46
3.2 Issue Management 47
3.3 Marketing Campaigns 48
3.4 User Roles 48
3.5 User Stories 49
3.5.1 User Onboarding 49
3.5.2 Profile Management 52
3.5.3 Product Management 53
3.5.4 Product Browsing and Discovery 56
3.5.5 Shopping Cart Management 58
3.5.6 Product Purchase 59
3.5.7 Customer Order Management 61
3.5.8 Seller Order Management 63
3.5.9 Product Feedback 66
3.5.10 Personalized Recommendations 66
3.5.11 Customer Support 67
3.5.12 Administrator Tools 69
3.6 Non-Functional Requirements 72
4 Design 77
4.1 Design Roadmap 77
4.2 Domain Model 77
4.3 Context Model 79
4.4 Bounded Contexts and Domains 80
4.4.1 Identity 80
4.4.2 Profile Management 81

4.4.3 Product Management 82
4.4.4 Inventory Management 83
4.4.5 Pricing and Promotions 84
4.4.6 Customer Feedback 85
4.4.7 Shopping Cart Management 86
4.4.8 Checkout & Order Management 87
4.4.9 Payment Context 88
4.4.10 Notifications Context 89
4.4.11 User Interaction 90
4.4.12 Recommendations 91
4.4.13 Conversational AI 92
4.4.14 Analytics and Reporting 93
4.4.15 Visual Search 94
4.4.16 Customer Support 95
4.5 Microservices Decomposition 97
4.6 Microservices Interaction 98
4.6.1 Create Product 99
4.6.1.1 Upload Product Images 99
4.6.1.2 Suggest Tags 100
4.6.1.3 Product Creation 100
4.6.2 Filter Product 101
4.6.3 View Product Details 102
4.6.4 Add Product to Shopping Cart 103
4.6.5 Delete Product from the Shopping Cart 104
4.6.6 Purchase Products 105
4.6.6.1 Used Patterns 108
4.7 Microservices Infrastructure Decisions 110
4.7.1 Programming Language and Framework 110
4.7.2 Message Broker 110
4.7.3 Data Store 111
4.7.3.1 Product Management 111
4.7.3.2 Shopping Cart 113
4.7.3.3 Checkout 114
4.7.3.4 Order Management 115
4.7.3.5 Price and promotions 116
4.7.3.6 Inventory 117
5 Development and Detailed Design 119
5.1 Shared Architecture 122
5.2 Technologies 123

5.3 Package organization 124
5.4 Product Management 126
5.4.1 Generate Upload URLs 126
5.4.2 Save Product 131
5.5 Product Query 137
5.5.1 Create Product 137
5.5.2 Get Product Summary 139
5.6 Pricing and Promotions 142
5.6.1 Validate Prices 142
5.7 Shopping Cart 145
5.7.1 Validate Cart 145
5.8 Payment Adapter 148
5.8.1 Pay Shopping Cart 148
5.9 Inventory 151
5.9.1 Create Inventory 151
5.9.2 Reserve Inventory 153
5.10 Order Management 155
5.10.1 Create Order 155
5.11 Order Query 157
5.11.1 Create Order 157
5.12 Platform Notifications 158
5.12.1 Notify when an Order is Created 158
5.13 Checkout 161
5.13.1 Perform Checkout 161
6 Deployment 166
6.1 Initial Development Setup 166
6.2 Docker Integration 166
6.3 Cloud Resources Creation 167
6.4 Kubernetes Integration 168
6.5 Observability 176
7 Performance Tests and Non-functional Requirements Check 181
7.1 Tests for Purchasing Products 181
7.1.1 Success Rate for Purchasing Process 181
7.1.2 Performance for Purchasing Process 182
7.2 Test for Retrieving Products 184
7.2.1 Success Rate for Product Catalog 184
7.2.2 Performance of Product Catalog 185
7.3 Error Handling Non-functional requirement 186
7.4 Modularity Non-functional Requirement 187

7.5 Logging Non-functional Requirement 187
7.6 Security Non-functional Requirement.. 187
7.7 Availability Non-functional Requirement 188
8 Conclusions and Future Work 189
8.1 Conclusions 189
8.2 Future Work.. 190
9 Bibliography 192

Table of Figures

Figure 1.1 Increase in imported coffee in Colombia 12
Figure 1.2 Coffee Production, exports, and domestic consumption in Colombia 12
Figure 2.1 Monolith Architecture 21
Figure 2.2 Microservices architecture 24
Figure 2.3 Kafka Architecture 30
Figure 2.4 Hexagonal architecture 32
Figure 2.5 Database per service pattern 32
Figure 2.6 CQRS pattern 33
Figure 2.7 Saga pattern with choreography 34
Figure 2.8 Saga pattern with orchestration 34
Figure 2.9 Circuit breaker pattern 35
Figure 2.10 Transactional outbox pattern 36
Figure 2.11 Traditional way of mocking infrastructure 38
Figure 2.12 Perform tests using Testcontainers 38
Figure 2.13 TDD cycle 39
Figure 2.14 Traditional wat of testing with mocking communication 40
Figure 2.15 Testing using verified contracts 41
Figure 3.1 Categories Layout 45
Figure 3.2 Tags for a Product 46
Figure 3.3 Variants Example 46
Figure 3.4 Product with multiple dynamic attributes 47
Figure 3.5 Make a purchase quality scenario 73
Figure 3.6 List product quality scenario 73
Figure 3.7 Change phone number quality scenario 74
Figure 3.8 Prevent crashing quality scenario 74
Figure 3.9 Purchase product during peak traffic quality scenario 74
Figure 3.10 List products during peak traffic quality scenario 75
Figure 3.11 New payment integration quality scenario 75
Figure 3.12 Log failed payment quality scenario 76
Figure 3.13 Authorization quality scenario 76
Figure 3.14 Availability quality scenario 76
Figure 4.1 Domain Model 79
Figure 4.2 Context model 80
Figure 4.3 Identity Entities and Value Objects 81
Figure 4.4 Profile Management Entities and Value Objects 82
Figure 4.5 Product Management Entities and Value Objects 83
Figure 4.6 Inventory Management Entities and Value Objects 84
Figure 4.7 Pricing and Promotions Entities and Value Objects 85
Figure 4.8 Feedback Entities and Value Objects 86
Figure 4.9 Shopping Cart Management Entities and Value Objects 87
Figure 4.10 Checkout & Order Management Entities and Value Objects 88
Figure 4.11 Payment Entities and Value Objects 89
Figure 4.12 Notifications Entities and Value Objects 90
Figure 4.13 User Interaction Entities and Value Objects 91
Figure 4.14 Recommendations Entities and Value Objects 92
Figure 4.15 Conversational AI Entities and Value Objects 93
Figure 4.16 Analytics and Reporting Entities and Value Objects 94
Figure 4.17 Visual Search Entities and Value Objects 95
Figure 4.18 Customer Support Entities and Value Objects 96

Figure 4.19 Contexts Grouped by Domain Type 96
Figure 4.20 Microservice Decomposition 98
Figure 4.21 Notation to Describe Communication Between Services 99
Figure 4.22 Uploading Product Images.. 100
Figure 4.23 Retrieval of Tags Suggestion 100
Figure 4.24 Communication Diagram of Product Creation 101
Figure 4.25 Communication Diagram of Product Filtering.......................... 102
Figure 4.26 Communication Diagram to Obtain Product Details View......... 103
Figure 4.27 Communication Diagram to Add an Item to the Cart............... 104
Figure 4.28 Communication Diagram to Delete a Product from the Cart..... 104
Figure 4.29 Communication Diagram for the Product Purchase................. 106
Figure 4.30 Sequence Diagram for Product Purchase................................ 107
Figure 4.31 Saga Pattern if Purchase Payment Fails 108
Figure 4.32 Saga Pattern if Stock Reservation Fails 109
Figure 4.33 Data Stores for Product Management Services 112
Figure 4.34 Product Management Data Model... 112
Figure 4.35 Data Store for Shopping Cart 113
Figure 4.36 Shopping Cart Data Model 113
Figure 4.37 Checkout Data Model... 114
Figure 4.38 Data Store for Checkout... 114
Figure 4.39 Data Store for Order Management... 115
Figure 4.40 Order Management Data Model... 116
Figure 4.41 Data Store for Pricing and Promotions 116
Figure 4.42 Pricing and Promotions Data Model.. 117
Figure 4.43 Inventory Database 117
Figure 4.44 Inventory Data Model 118
Figure 5.1 Increased Time to Market... 121
Figure 5.2 Class Diagram Template 123
Figure 5.3 Package Structure for Microservices.. 125
Figure 5.4 Request to generate Upload URLs 127
Figure 5.5 Response of obtaining Image URLs... 127
Figure 5.6 Azurite and Redis containers in test class 128
Figure 5.7 Persistent Set in Redis Created by Spring Data 128
Figure 5.8 Hexagonal Architecture preventing changes to upper layers....... 129
Figure 5.9 Class Diagram of Generate Image URLs Use Case 129
Figure 5.10 Custom UseCase Annotation... 130
Figure 5.11 Service using Custom UseCase Annotation 131
Figure 5.12 Save Product OpenAPI Schema 132
Figure 5.13 Class Diagram of Create Product Use Case............................. 133
Figure 5.14 Contract to specify the Message Schema 135
Figure 5.15 Method that will be used by Spring Cloud Contract to send the message 136
Figure 5.16 Autogenerated Test by Spring Cloud Contract 136
Figure 5.17 Containers automatically boot up used by the tests 137
Figure 5.18 Class Diagram of Create Product Query 138
Figure 5.19 Specify Contract location for contract testing 139
Figure 5.20 Custom Consumer test using Spring Cloud Contract 139
Figure 5.21 Retrieve Products Summary OpenAPI Schema 140
Figure 5.22 Class Diagram of Retrieve Products Summary......................... 141
Figure 5.23 Validate Items Price OpenAPI Schema.................................... 142
Figure 5.24 Class Diagram of Validate Items Price 143
Figure 5.25 Custom Annotation to specify the Discount Restriction........... 144
Figure 5.26 Specific Validator Using the Custom Annotation 144

Figure 5.27 Checkout Shopping Cart OpenAPI Schema 145
Figure 5.28 Class Diagram of Validate Shopping Cart 146
Figure 5.29 Configuration of OpenAPI Generator Plugin............................. 147
Figure 5.30 Generated Classes from OpenAPI Generator............................ 148
Figure 5.31 Retry with backoff to call the Payment Gateway 148
Figure 5.32 Usage of the Circuit Breaker to call the Payment Gateway 149
Figure 5.33 Customization of the Circuit Breaker 149
Figure 5.34 Pay Shopping Cart OpenAPI Schema...................................... 150
Figure 5.35 Class Diagram of Cart Payment... 151
Figure 5.36 Class Diagram of Create Inventory 152
Figure 5.37 Reserve Inventory OpenAPI Schema 153
Figure 5.38 Class Diagram of Inventory Reservation 154
Figure 5.39 Class Diagram of Save Order... 156
Figure 5.40 Class Diagram of Creating Order for Queries........................... 157
Figure 5.41 Register for Notifications OpenAPI Schema............................. 159
Figure 5.42 Class Diagram of Receive Notifications 159
Figure 5.43 Use of text/event-stream and Project Reactor's Flux 160
Figure 5.44 Flux Structure. Adapted from **Reactor Reference Guide** (Project Reactor, 2025) 160
Figure 5.45 Push New Notifications Using Sink.. 160
Figure 5.46 Transforming the Sink into a Flux... 161
Figure 5.47 Perform Checkout OpenAPI Schema.. 162
Figure 5.48 Class Diagram of Purchase Cart.. 163
Figure 5.49 Use of Jakarta Validation Annotations 165
Figure 6.1 Inventory Service in Docker Compose File 167
Figure 6.2 Shift from local resources to cloud resources 168
Figure 6.3 Application's ConfigMap 169
Figure 6.4 Application's secrets.. 169
Figure 6.5 Payment Adapter Deployment 170
Figure 6.6 Payment Adapter K8s service 171
Figure 6.7 Payment Adapter's HPA.. 172
Figure 6.8 Liveness and Readiness Properties 173
Figure 6.9 Application's Ingress 174
Figure 6.10 Deployments in AKS... 175
Figure 6.11 AKS Nodepools 175
Figure 6.12 Pods maintained by Kubernetes 176
Figure 6.13 HPA in AKS analyzed from Open Lens 176
Figure 6.14 Basic Monitoring from AKS 177
Figure 6.15 Application Insights Agent Resource 177
Figure 6.16 Product Microservice in Application Map................................ 178
Figure 6.17 Checkout Process in the Application Map............................... 178
Figure 6.18 Application Map used to spot problems.................................. 179
Figure 6.19 Order Creation in Application Map 179
Figure 6.20 Error Codes in Application Insights... 180
Figure 7.1 Success Rate for Purchasing Process Test Results..................... 181
Figure 7.2 Requests per Second for 'Success Rate for Purchasing Process' Test 182
Figure 7.3 Response Time for 'Success Rate for Purchasing' Process Test ... 182
Figure 7.4 'Performance for Purchasing Process' Test Results 183
Figure 7.5 Requests per Second for 'Performance for Purchasing Process' Test 183
Figure 7.6 Response Time for 'Performance for Purchasing Process' Test 183
Figure 7.7 99,000 Products in Azure Search 184

Figure 7.8 ‘Success Rate for Product Catalog’ Test Results.......................... 184
Figure 7.9 Requests per Second of ‘Success Rate for Product Catalog’ Test . 185
Figure 7.10 Response Time for ‘Success Rate for Product Catalog’ Test....... 185
Figure 7.11 ‘Performance of Product Catalog’ Test Results 186
Figure 7.12 Requests per Second for ‘Performance of Product Catalog’ Test 186
Figure 7.13 Response Times for ‘Performance of Product Catalog’ Test........ 186

1 Introduction

1.1 Context and Motivation

Colombia is one of the principal coffee producers in the world. According to the USDA Foreign Agricultural Service, it is the third-largest producer with 7% of global production, generating 12.9 million bags per year. Additionally, most of the coffee is exported, because the price paid by other countries is higher. Due to this, local consumption cannot be supplied, which generates the need to import coffee. According to Bloomberg, the amount of imported coffee has increased by 62% in 2021 and by 39% in 2022. The figure below shows the increase in imported coffee between 2019 and 2022.

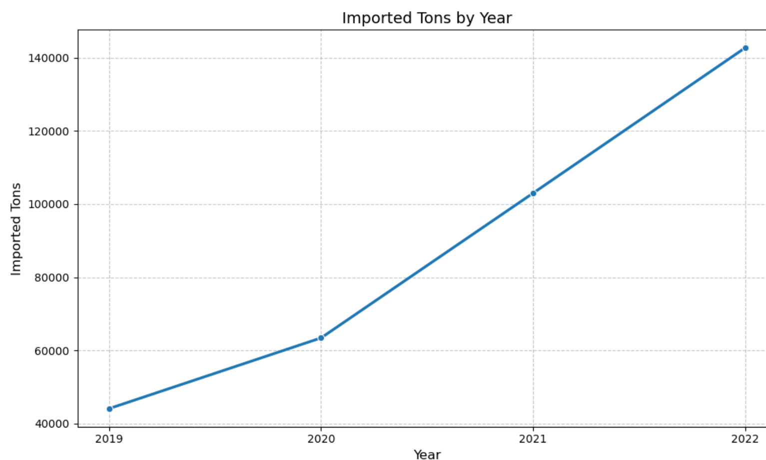

Figure 1.1 Increase in imported coffee in Colombia

The figure below shows the production, exports, and consumption of coffee in Colombia.

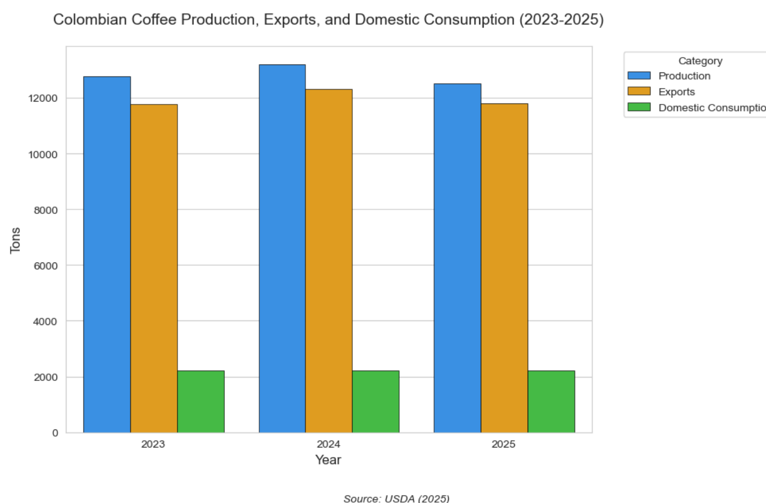

Figure 1.2 Coffee Production, exports, and domestic consumption in Colombia

The Colombian market is evolving, so that more homes are willing to pay the price of the local product. According to the president of the Colombian Federation of Coffee Growers, Germán Bahamón, (2024), “Premium (coffee)

products in the Colombian market are growing at a faster pace, a segment that has grown 53.4% in value and 27.2% in volume over the last year". This market change presents an opportunity to introduce a new marketplace specializing in coffee products.

Selling products through e-commerce benefits coffee producers, as they can sell directly to customers, avoiding long intermediary chains, and increasing their margins. On the other hand, the customer benefits from the reduced cost of quality coffee resulting from the removal of intermediaries. Direct sales can be performed by niche e-commerce, since, according to the Boston Consulting Group (2023), 70% of customers prefer shopping in specialized marketplaces rather than stores that offer the same products.

The primary objective of the platform is to provide a specialized sales channel for local coffee producers, promoting internal consumption at a competitive price, thereby handling the increased demand for premium coffee.

1.2 Objectives and Scope

This thesis outlines the software lifecycle steps for implementing a coffee e-commerce platform, where local producers and small businesses can offer their products, which vary from brewing equipment to roasted coffee beans. This software will be developed with a special focus on architectural design to ensure a reliable, scalable, and maintainable system that can handle increased demand and evolve rapidly in response to new user requirements. Furthermore, the project will employ best practices at each stage to ensure its success, addressing common industry problems such as poor design, a lack of shared architecture, and inadequate observability tools.

The main functionalities of the coffee e-commerce platform are the following:

- **User Management**:
    - Onboarding (authentication, authorization).
    - Profile customization (delivery addresses, notification preferences).
- **Product Operations**:
    - Lifecycle management (creation, updates, deletion).
    - Discovery (browsing, listing, and searching).
    - Feedback and recommendations (List reviews, create reviews).
    - Personalized recommendations.
- **Order Workflow**:
    - Shopping cart (add and remove items, modify quantities).
    - Purchase processing (checkout, payments).
    - Order tracking (for both customers and sellers).
- **Support & Administration**:
    - Customer support interactions.

The main objective of the project is to design the cloud-native e-commerce platform presented above, using multiple design approaches and cloud patterns to ensure the correct operation of the system.

The project focuses only on the back-end side, where the user interface and experience are not part of the project. Moreover, real payment gateway integration, shipping process, and mobile application are not part of the project.

1.3 Design and Development Approach

Firstly, the user requirements were specified through user stories and acceptance criteria covering the entire application.

Next, following the DDD methodology, the domain and context model were clearly defined for the entire system, providing a high-level view of the system's architecture. This helps identify the external services and primary entities that the system will utilize. After that, the definition of limits through bounded contexts was performed, followed by the decomposition of microservices, which can be done easily once the bounded contexts are defined.

Later, a minimum viable product (MVP) was defined to be further refined, which provides the most essential functionality of the system and includes the use case of purchase of products along with all the required functionality to perform this action, such as creating products, managing the shopping cart by adding and deleting items, product listing, detailed views, and checkout. These functionalities, which were also included in the high-level design, were detailed with the microservices interaction diagrams and the data model. Furthermore, a subset of the MVP was selected for detailed design, implementation, and deployment in the cloud.

The microservices interaction involved in the MVP was thoroughly designed using microservice communication diagrams, where all synchronous calls and asynchronous events were clearly defined to provide a solid foundation for building the subsequent code.

The most impactful microservices interaction, for which a wide variety of architectural patterns can be applied, was selected for development, which was performed using good programming practices, including SOLID principles, clean code, and the use of design patterns such as Strategy and Factory patterns. The services use a hexagonal architecture, which was slightly modified after the development of two microservices to complete the required functionality in time. The interaction with the system was performed through HTTP requests using Postman to test and validate the APIs while developing.

In addition, to deploy the microservices into the cloud, Kubernetes, the de facto standard for container orchestration, was chosen. Each microservice has its own manifest that describes its components, encouraging a declarative approach to defining the infrastructure, which provides replicability and traceability. To monitor the deployed microservices, an observability tool was included, providing real-time insights about the system's status.

Finally, to verify the application's performance with respect to the non functional requirements, multiple load tests were conducted, providing valuable insights about the application's behavior.

1.4 Best Practices of Software Engineering

A summary of the best practices applied in this project is shown below:

- The use of **Microservices architecture**, which provides multiple benefits that will be covered later, such as:
  - Independent scalability.
  - Responsibility segregation.
  - Fast introduction of new features.
  - Loose coupling between different responsibility areas.
  - Freedom of technology choice, for both data store and APIs.
- The use of **Domain-Driven Design (DDD)**, aligning business domain and technology to identify bounded contexts that serve to clearly define the responsibilities of each microservice, creating a maintainable and modifiable system.
- The use of **Hexagonal Architecture**, which leverages the domain-driven concepts by isolating the application logic to make it independent from the infrastructure.
- The use of multiple **cloud design patterns** that provide a solid background to develop a high-quality system, such as:
  - Circuit breaker to improve response time and reliability.
  - Saga and Transactional outbox to ensure reliability.
  - CQRS to enable single responsibility between operations and independent scaling.
  - Event-driven to ensure loosely coupled services, improving modifiability.
  - Database per service to reduce data store coupling between microservices and to encourage technology flexibility and independent scaling.
- The use of technologies widely adopted by the industry and their best practices, such as **Spring Boot**, which is used by companies such as Netflix and Walmart.
- An initial approach of implementing testing practices, such as:
  - **Test-Driven Development (TDD)** to demonstrate how it leverages a robust software that covers a wide variety of scenarios that could occur in a functionality.
  - **Contract tests** to ensure thorough integration testing for microservices, verifying that the answer provided by an external service is the same as the one used in the test.
  - **Testcontainers**, allowing making tests with infrastructure as similar as possible to the production environment, thereby reducing the complexity of integration tests in microservices.

1.5 Structure of the document

Section 2 provides the theoretical and technological foundations for the development of the platform, including architectural patterns and protocols.

Section 3 covers the requirements specification, where the user stories and their acceptance criteria are described. It begins by specifying some concepts to provide a smooth understanding of the user stories, and concludes with the non-functional requirements and quality attribute scenarios.

Section 4 presents the system's design. It starts with a high-level overview of the domain and context model in sections 4.1 and 4.3. Then, section 4.4 divides the business domain into bounded contexts, following the DDD approach. Based on the bounded contexts, the microservices decomposition was performed in section 4.5. Once the microservices are defined, section 4.6 shows the interaction between them for high-priority user stories. Subsequently, section 4.7 addresses the infrastructure decisions for the technologies used.

Section 5 covers the development of microservices and their detailed design. Sections 5.1, 5.2, and 5.3 describe the technologies, class responsibilities, and packages used in each microservice, all aligned with hexagonal architecture principles. Later, sections 5.4 to 5.13 present key implementation details, including notable code snippets (such as custom annotations or server-sent events) and class diagrams for each microservice.

Section 6 describes the deployment process for the project, including the creation of cloud services, the containerization of microservices, and the setup of the Kubernetes cluster.

Section 7 presents the performance tests conducted on the platform and its relationship with the non-functional requirements specified in section 3.

Section 8 presents the conclusions and outlines how the project can be extended.

2 Theoretical and Technological Foundations

This chapter covers the theoretical baseline for the concepts, principles, and technologies used to implement the system. It begins with the definition of software quality attributes that guide the architectural design decisions, followed by the problems that a monolithic architecture has and how microservices and a cloud-native approach can address them. The chapter then provides a brief definition of some data management technologies and communication protocols, along with architectural patterns used to provide reliability and scalability to the application. Additionally, it describes the testing technologies that are commonly used in microservices, followed by observability tools to check the system's health. Finally, the chapter describes briefly the concept of cloud computing and then focuses on a description of the Azure services used in the project.

2.1 Software Quality Attributes

This section provides a brief description of the definition of a subset of quality attributes that are most relevant to the developed system, which guided the architectural decisions.

2.1.1 Reliability

A reliable system is one that performs its intended function (Eldar Jahijagic, 2022). Furthermore, it is robust against failures and can preserve its integrity. In distributed applications, maintaining reliability becomes harder due to the increased complexity and the distributed data store.

Reliability involves multiple aspects, such as:

Fault Tolerance: The system can continue working after a failure.

Error Handling: The system must be able to handle unexpected errors.

Consistency: The system must maintain data integrity.

One of the most used metrics to assess reliability is the Service-Level Agreement (SLA), which defines the expected uptime. This metric is specially used for cloud providers to promote and express the uptime of their services (AWS, 2025).

2.1.2 Maintainability

A maintainable system is one that can be modified easily (ISE, 2024). These modifications can be to add new functionality or to correct bugs. To have a maintainable system, multiple aspects should be taken into account, such as:

1. The existence of documentation, which can be formal or informal, that provides an overview of what the system is doing.
2. The presence of unit tests, which are a live documentation of the system, allowing developers to understand what the software is doing and make changes without breaking existing functionality.

3. The existence of a standard structure, where developers can know how the classes are structured. This can be achieved by having a well-defined architecture that clearly states the responsibility of each class and how communication is between layers.
4. Following good programming practices, which allow developers to deal with clean and structured code.

Some of the most important metrics to assess maintainability are:

- **Cyclomatic Complexity:** Measure the number of independent paths in the source code.
- **Code Smells:** They are the number of issues that a program has, which can lead to bigger problems as time passes. There are popular tools like SonarQube [1] (Sonar, 2025) that can detect them, and they can be integrated into the continuous integration pipeline to avoid passing the smells to production environments. These tools contain a list of rules for each programming language. For example, Sonar defines 446 code smell rules[2] (Sonar, 2025) for the Java programming language.

2.1.3 Scalability

A scalable system is one that can increase its capacity according to user traffic (Azure, 2025). The two most mentioned attributes of scalability are:

1. **Horizontal Scalability (Scale-Up):** It is defined as the ability to add more instances to the program. For example, if the application is running on a single machine, horizontal scalability would be running the program on multiple machines and using a load balancer to redirect the traffic to the instances. In a distributed environment, especially those that run on containers and use a container orchestrator such as Kubernetes, horizontal scalability is the ability to increase the containers that run the application. Furthermore, in a microservice environment, the scalability can be applied to specific microservices of the system that require it, leaving the remaining parts unchanged.

2. **Vertical Scalability (Scale-Out):** It is defined as the ability to add more computing resources to the machine that is running the application, for example, more CPU, RAM, or storage.

Apart from the attributes mentioned above, there are others that are less common but also important, for example:

- **Elasticity:** It is the ability to dynamically add resources according to current demand.
- **Data Scalability:** It is the ability to manage large amounts of data without degrading performance.

[1] https://www.sonarsource.com/products/sonarqube/
[2] https://rules.sonarsource.com/java/type/Code%20Smell/

2.1.4 Modifiability

A modifiable system can be easily changed without breaking its existing functionality (Bachmann et al., 2007). This quality attribute is closely related to "Maintainability", as some of its properties are shared between them; for example, a code with tests is easier to maintain and modify. However, some aspects are more important to make a modifiable project, such as:

1. **Loose coupling and high cohesion:** Loose coupling prevents changes in one class from requiring multiple changes in other classes. High cohesion allows all the required changes to be in one place.
2. **Encapsulation**: Hides internal details to prevent high coupling between classes.
3. **Domain-Driven Design (DDD):** Having the code as close as possible to the business domain increases the understanding and responsibility of each class, making changes easier.

In a cloud-native application, the principles mentioned earlier are used to define microservices, where each has a single responsibility and owns its bounded context. Additionally, as they communicate using contracts (either through API calls or asynchronous events), other components do not need to change as long as the contract remains the same.

2.1.5 Performance

A system with high performance is capable of receive and process multiple requests within the expected response time (Ashanin, 2018), this measure depends on each system and how are the expected load that it will have, for example, the required performance for a small e-commerce that will have a traffic of one hundred users per day is not the same as the performance needed for a big e-commerce such as Amazon or Shein. Therefore, performance must be measured in each particular context.

Performance can be measured using multiple values, such as:

- **Response Time:** The time that the system takes to answer a user request.
- **Throughput:** The number of operations that the system can handle in a period of time.
- **Resource utilization:** It measures the resources the system uses to perform an operation.
- **Startup time:** It is the time that the system needs to start receiving requests, which is tightly related to "Scalability" and is critical to process requests during an application's usage peak. For example, the Spring Boot team is working on providing a faster start with projects like Ahead of Time (AOT) optimizations[3], which allow the usage of a GraalVM[4] native

[3] https://docs.spring.io/spring-framework/reference/core/aot.html
[4] https://www.graalvm.org/

image, which significantly improves the startup time of Spring Boot applications (Spring, 2025).

There are multiple ways to improve performance in an application, such as analyzing and rewriting database queries, using optimized data structures, caching frequent data, reducing memory by using the singleton pattern, etc. However, these optimizations have to be justified to avoid optimizing something that is not required. As Knuth (1974) advises, "premature optimization is the root of all evil" (p. 268).

2.2 Software Paradigms and Architectures

This section covers the design principles and software architectures used to implement the system. It also covers the definition of monolithic architecture to bring context about the needs that led to the creation of microservices.

2.2.1 Domain-Driven Design

According to Fowler (2020), "Domain-Driven Design is an approach to software development that centers the development on programming a domain model that has a rich understanding of the processes and rules of a domain.". This strong connection between the domain and the software enables developers to take ownership of the software they build, knowing precisely what they are creating rather than producing disconnected functionalities (Evans, 2003, p.9).

Some of the main components of DDD are:

- **Ubiquitous Language:** It is the main connection between the software and the business domain (Evans, 2003, p.12). There are concepts extracted from the domain that will be reflected in the software, so that "programmers can show business experts technical artifacts, … thereby closing the feedback loop" (Evans, 2003, p.10). It closes the communication gap between the entire team, which improves productivity by avoiding constant translations between teams. Furthermore, it enables business rules to be explicitly stated in the code.

- **Bounded Contexts:** When the system is large, "Total unification of the domain model … will not be feasible or cost-effective" (Evans, 2003, p.206). To tackle this issue, there are clearly defined areas of responsibility called Bounded contexts, each of which represents an area and clearly states the limit between them, allowing the area to focus on its responsibilities without colliding with others, instead of trying to build an enormous, disconnected object. For example, the entity "Order" can have different attributes depending on the area of responsibility dealing with it, such as the accounting team, which may be interested in the

price, tax segment, etc., and the marketing team, which may be interested in customer segments, color, etc.

- **Entity:** It is an object that is defined by an identity that will be the same throughout its entire lifecycle (Evans, 2003, p.50). This concept is especially important in distributed systems since multiple representations of one entity can exist in different bounded contexts; the identifier is the key element to relate these representations.

- **Value Objects:** There are objects used to describe characteristics, which do not have an identity (Evans, 2003, p.54). For example, the "transaction" entity has a "currency" value; this last concept does not need an identity, since the currency of transaction A can be the same as the currency of transaction B. However, it can be treated as an object rather than a primitive value because it has its business rules and invariants, such as it should be an existing currency and have a fixed number of letters.

- **Services:** There are stateless objects that perform operations that do not fit into an entity or a value object (Evans, 2003, p.59). They are defined with a verb instead of a noun, since they perform a specific operation.

2.2.2 Monolithic Architecture

Monolithic architecture is a traditional development model where the entire application is contained within a single software project (Powell & Smalley, 2024).

Figure 2.1 Monolith Architecture

Monolithic architecture was thought to be used with the most commonly used product lifecycle model at that time, which was waterfall, where software was

delivered at the end of the project in one phase. However, nowadays, agile methodologies have been replacing it, and, although monoliths can work with agile teams, they introduce coordination burden, and, according to Conway's Law, "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure." (Conway, 1968, p. 28). Therefore, if an organization is composed of multiple agile teams of 5 to 8 people each, monoliths become a problem since they lack the required structure for teams to work efficiently. Some of the communication problems that arise using monoliths and agile teams are:

- **Merge conflicts:** Tangled monoliths where multiple teams work can lead to problems during code merging. For example, if a team refactors some class that the other team is using, it will result in a code freeze until the merge conflict is resolved.

  Uploading changes to production becomes difficult since multiple teams are working on functionalities at the same time, and, a feature that is in the test environment not ready to be pushed to production can block another feature release, moreover, to deploy it in time, developers have to break the git branching model to push directly to production from the team's branch, which can cause inconsistency issues.

- **Slow CI[5]/CD[6] Pipelines:** In a monolith, every change triggers all the application tests, making the deployment pipeline slow, which increases the deployment time and encourages teams not to deploy small features to avoid the waiting time while the pipeline is processed, which, in the end, breaks the agile methodology.

- **Ownership ambiguity:** Since multiple teams work on the same codebase, making a change could require coordination between them, which is slow. Another critical drawback is the lack of freedom that developers have when they want to modify the existing codebase because of the fear of breaking something that another team uses.

- **Scaling problems:** Adding more developers to an existing monolith can be counterproductive, since, according to Brooks' law, "adding manpower to a late software project makes it later" (Brooks, 1975, p. 25). Therefore, the company has a bottleneck in delivering new functionalities.

- **Resistance to new technologies (Powell & Smalley, 2024):** Having all the codebase in the same project cuts the team's ability to innovate with new tools, for example:

  - If the monolith is written in Java 11, to use features of Java 17 requires the upgrade of the whole monolith, which is not feasible

---

[5] Continuous Integration: Constantly merging the code changes into a common branch (GitLab, 2022)

[6] Continuous Delivery: Constantly deploying changes to production environments (GitLab, 2022)

in some cases due to the lack of time to perform the required adjustments.
    - Adding a functionality that would fit better using another programming language is not possible.

- **Cascading Failures:** In monolithic architecture, if a critical component fails, such as a memory leak or insufficient resources to run the application, all functionalities become unavailable, causing a complete disruption of service.

- **Lack of independent scaling:** If there is a part of the application that needs more resources due to increased demand, there is no way to scale this part independently; the whole system must be scaled, wasting resources and making it more expensive.

Despite the problems mentioned above, monolithic architectures have multiple benefits, such as:

- **Simplified testing:** Having all the functionality in the same codebase make the integration and end-to-end tests easier to perform (Harris, 2024).
- **Less network hops:** For performance-critical applications, having multiple network hops because of the system distribution is not feasible.
- **Fast prototypes:** When building a prototype, instead of focusing on concerns like eventual consistency, configuration of API calls, etc., building it as a monolith is faster and cheaper.
- **Simple applications:** When the system complexity can be handled in one codebase, a monolithic architecture is the right choice.

Apart from that, projects like Spring Modulith help build modular monoliths (Spring, 2022), where each module can be viewed as a bounded context, with clear communication channels between them that avoid high coupling. This approach addresses some of the problems of traditional monolithic architecture, while also bringing some of the benefits of distributed applications.

2.2.3 Microservices

The problems that monolithic architecture faces, along with the rise of containerization and pipeline automation, led to microservices, which aim to solve the problems mentioned above.

Microservice architecture is a development model that consists of splitting the codebase into independent services (Azure, 2025), where each of them has a single responsibility and manages its own data. Each microservice is intended to be loosely coupled, since it does not depend on other services, and highly cohesive, where all the related functionality is located within the same service.

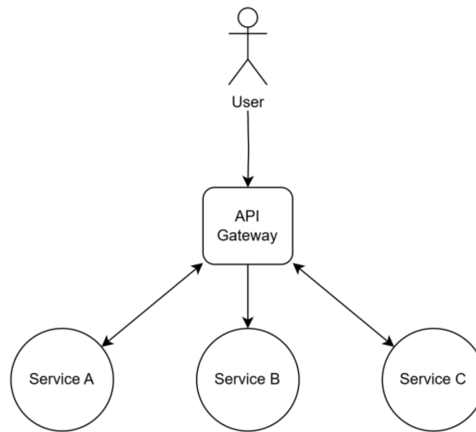

Figure 2.2 Microservices architecture

Microservices aim to resolve common problems that monoliths have, for example:

- **Fewer merge conflicts:** Each team is responsible for their microservice, and no multiple teams are working in the same codebase.

- **Independent deployments:** The CI/CD pipeline does not trigger unnecessary tests from other functionalities.

- **Clear ownership:** with each microservice having a clear responsible team.

- **Scalable deployments:** The company can increase the development throughput by adding more teams to develop microservices, since it does not have any dependencies, and there are no coordination problems.

- **Technology flexibility:** Teams can experiment with new technologies without worrying about breaking existing features or having compatibility problems with other libraries.

- **Failures Isolation:** Microservices are designed to work independently, so if one microservice is failing, the others can still work.

- **Independent scaling:** All the microservices can be scaled horizontally and vertically independently. They can also differ in the initial resources, which saves cost and provides flexibility.

- **Security overhead:** Microservices introduce security complexities due to the communication between services. It requires complex tools, such as mutual TLS (mTLS) or service meshes like Istio, to ensure secure communication between services. Another option is not to expose the services directly to the clients by using an API gateway. However, it introduces a system's single point of failure.

According to Lewis and Fowler (2014), the key characteristics of microservices are:

- **Componentization via services:** The system is composed of independent services that communicate with each other by using well-defined contracts, which are typically HTTP endpoints, that hide the implementation details.

- **Organized around business capabilities:** In agile methodologies, teams are organized by business capability (i.e., a feature) instead of by a specific technology. For example, instead of having Group A of Developers, Group B of DevOps specialists, and Group C of UI specialists. The team will be organized to have all the required members to develop the functionality. For example, Group A is composed of one developer, one DevOps specialist, and one UI specialist, making a cross-functional team that reduces the coordination between teams and speeds up the development and delivery of value.

- **Products, not Projects:** The team that develops a microservice is responsible for it throughout its entire lifecycle. The idea is to empower developers so they can take responsibility for what they build.

- **Smart endpoints and dumb pipes:** One of the main problems of Service-Oriented architecture (SOA) was the business leakage to the communication channel, making the projects difficult to modify and maintain, since they are dependent on the service bus. To overcome this problem, all the business logic is inside the microservice, while the communication channels, like REST, are simple.

- **Decentralized Governance:** Thanks to the microservices modularity, multiple technologies can be used in the project; they can be chosen based on the best fit for the specific feature, rather than sticking to a common one.

- **Decentralized data management:** According to Fowler and Lewis (2014), "the conceptual model of the world will differ between systems", this phrase can be applied to entities in the same system, for example, projects with different areas of responsibility, such as marketing and accounting team can use the same entity for different purposes, for example, the entity "product" can have specific attributes depending on which context is being analyzed.
    - The marketing team can have attributes like customer segment or campaign performance
    - The accounting team can have attributes like unit cost and tax category

  For this reason, instead of having one overloaded entity with all the values from different concepts, the entity can be represented in each context with the values relevant to it.

- **Infrastructure automation:** This is a critical point for the success of a microservices project, since agile methodologies' core goal is to deliver value fast, which implies deploying changes constantly. With microservices, this becomes more important, as having multiple moving parts requires more deploys, and it is required to have automation to focus on developing functionalities rather than worrying about how to deploy the system.

- **Design for failure:** Because of the extensive communication between services, failures are prone to occur. The system has to be designed to handle these errors gracefully and to have an observability service to analyze what happened.

- **Evolutionary Design:** Microservices have to be:

    - Replaceable without making changes in other services.
    - Iteratively refined, such as splitting a service into multiple ones.
    - Deprecated gracefully

Microservices are not a silver bullet; they come with some challenges, which are:

- **Complexity:** The application is divided into multiple services, which makes observability and debugging more difficult.
- **Latency:** Multiple calls between microservices are performed to serve a request, which increases the response time due to the network hops.
- **Data Integrity:** Since data is distributed, maintaining integrity becomes difficult because the data can be replicated in multiple parts of the application, and ACID transactions across multiple services are not enforced by the database.
- **Complicated testing:** Performing end-to-end tests becomes much harder in microservices than monoliths, and special techniques such as consumer-driven contracts have to be applied to try to solve this problem.

2.2.4 Cloud Native

Cloud-native is the combination of practices for building applications specifically to be deployed in a cloud environment.

According to the Cloud Native Computing Foundation (2014), “Cloud native technologies and architectures typically consist of some combination of containers, service meshes, multi-tenancy, microservices, immutable infrastructure, serverless, and declarative APIs.”

It uses technologies like:

a) Containerized applications for standardized run and fast replacement.
b) CI/CD pipelines for fast deployment of new features.
c) Health checks that monitor the system automatically, such as Kubernetes liveness and readiness.
d) Observability tools to analyze the system in real time.

e) Dynamic service discovery, allowing for easy addition and replacement of services, such as Kubernetes DNS.

In conclusion, cloud-native is a combination of practices and tools that enhance the development of a system intended to be deployed in the cloud from the design phase, where all architectural decisions are made to take advantage of cloud capabilities.

2.3 Data Management Technologies

This section provides a brief overview of the database types utilized in the system, specifically SQL, NoSQL, and cache databases (which are part of NoSQL).

2.3.1 Relational Databases

Relational databases are one of the most used data storage technologies; they organize information in tables and attributes. Each table has its primary key, which is the identifier of the record. The relation between tables is made through foreign keys, which reference the primary key of another (or the same) table.

Data is manipulated using Structured Query Language (SQL). One of the most important features of relational databases is the ACID compliance, which is an acronym that states the following properties:

- **Atomicity:** Each transaction is treated as a single unit; it is either completely performed or completely discarded.
- **Consistency:** It ensures that the database remains in a consistent state, where all the invariants, such as constraints, cascades, and triggers, are enforced.
- **Isolation:** Each transaction is isolated from others.
- **Durability:** Once the transaction is committed, the information is preserved.

2.3.2 NoSQL Databases

NoSQL databases are another type of data storage; their names come from the acronym "Not Only SQL".

There are multiple types of databases that fall into this classification, such as document-based databases like MongoDB, key-value databases like Redis, and graph databases like Neo4j.

These databases are commonly known for their flexible data structure and their ability to manage semi-structured data (MongoDB, 2025).

Projects can benefit from using both relational and NoSQL databases, using them according to their specific requirements.

2.3.3 Cache Databases

Cache databases are a subgroup of NoSQL databases; they store information in memory to allow fast access. Some use cases of them are.

- **Session storage:** REST APIs are stateless; one way to have memory without breaking the REST philosophy is by saving relevant session information in the cache database.
- **Frequently accessed data:** Instead of retrieving the information from the primary database, the frequently accessed information can be saved in cache, which allows fast retrieval of information and improves the response time. There are specific patterns, like the cache-aside, to implement this behavior.

2.4 Communication in Distributed Systems

This section describes two mechanisms used to communicate between services: synchronous communication via HTTP and asynchronous communication, focusing primarily on Kafka, which is the technology used in the system.

2.4.1 HTTPS

Hypertext Transfer Protocol (HTTP) is a lightweight communication channel for client-server architectures (Mozilla, 2025). Its main functionality is to send requests using verbs such as GET and POST and receive responses using status codes, such as 200 and 400.

Modern web application uses HTTP to maintain **resources,** which is an essential term in the Representational State Transfer (REST) paradigm (IBM, 2024). A resource is a relevant entity in the application, such as a User or a Product. Moreover, in domain-driven design, a resource is commonly associated with a **domain entity**, which is the representation of a business object.

The combination of URI paths, verbs, and headers provides a simple but highly extensible communication protocol for modern applications.

HTTPS is the secure version of HTTP, where the communication protocol remains the same but with a security layer, which encrypts the information to make it secure while it is going from client to server and vice versa (Cloudflare, 2025). The protocol to encrypt communication is called Transport Security Layer (TLS), which shares an SSL certificate that has the key to start the secure connection.

HTTP protocol is highly extensible, and can be used to maintain an open unidirectional communication using **server-sent events**, where the server can push new events to the client. This mechanism is useful to provide notifications to the application in real time without major implementation. Server-sent events can be triggered by using the *Content-Type* header with the value *text/event-stream.* After sending the response with this header, the client and the server will maintain an open connection.

2.4.2 Asynchronous Messaging

Asynchronous messaging is a paradigm of communication with two main actors, producers and consumers. Producers are responsible for sending messages; they do not know who the consumers are, and they do not expect a response (Azure, 2022). Message Brokers, such as Kafka and RabbitMQ, are responsible for receiving messages and publishing them to the appropriate customers.

The main benefits of this communication protocol are:

- Decoupling between producers and consumers: If a new consumer needs to be added, no changes have to be made on the producer side.
- Improve response time: For long-running tasks, instead of waiting for task termination, the producer can trigger it by publishing a message, allowing the task to run in the background.

Asynchronous messaging requires careful implementation, since handling errors and ensuring message idempotency are critical to maintaining the application in a consistent state.

Apache Kafka was chosen to be the message broker to enable event-driven communication in the e-commerce platform. It is one of the most popular technologies for this purpose since it supports a large amount of data (Red Hat, 2024). The most important elements of Kafka are:

- **Message broker:** It is responsible for storing messages.

- **Topic:** It is the destination of the message; for example, the producer sends messages to topic A, and consumers who want to receive messages must listen to this topic.

- **Consumer Groups:** There are groups made by consumers that perform the same actions. Each consumer group receives the message once, so each group has its own message offset. For example, if a consumer group does not have consumers, when a new consumer is connected, it will receive all the messages of the topic that are stored according to the Kafka retention policy.

- **Partition:** They are logical separations of a topic; they must be specified at the topic creation. For example, if a topic has two consumers in the same consumer group, each of them will be assigned to a different partition; by default, the message will be sent to each partition using a round-robin approach. In case the topic has only one partition, the two consumers will be constantly connected and disconnected from the partition, which could cause unexpected behavior.

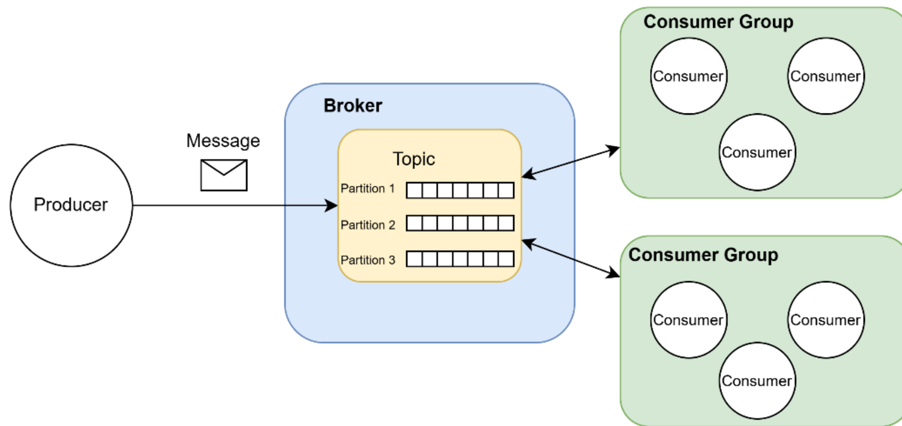

Figure 2.3 Kafka Architecture

2.5 Architectural Patterns

Distributed applications present multiple challenges related to performance, data integrity, and other factors. To address this, multiple architectural patterns can be applied. For instance, Azure documents more than 40 cloud design patterns (Azure, 2025). This section describes the key patterns implemented in the e-commerce platform.

2.5.1 Hexagonal Architecture

Hexagonal architecture is a layered approach to organizing software, where all layers serve the business logic (Radhakrishnan Periyasamy, 2020). It is also known as ports and adapters (AWS, 2025), which are mechanisms that preprocess data to communicate with the business layers.

The primary goal of hexagonal architecture is to isolate business logic from infrastructure concerns, thereby decoupling layers to facilitate changes in upper layers without affecting inner ones.

To achieve this decoupling, each component of the architecture plays a well-defined role, which is described below.

- **Driving Adapters:** They are the public face of the application; they receive data from the external world and transform it into the structure defined by the driving port.
- **Driving Port:** It is the contract defined by the application core that driving adapters must follow. Typically, there are command objects that follow the command pattern, serving as a way to decouple the layers.
- **Application Layer:** It is responsible for defining and implementing the driving port, following the dependency inversion principle. It represents

an application's use case, such as "Create Order" or "Make a Purchase", which serves as an orchestrator that makes the necessary calls to execute the use case. Application classes do not perform business rules nor enforce invariants; this responsibility is delegated to the domain entities and services.

- **Domain Entities:** These classes are the most important part of the architecture, each entity enforces their business logic and invariants, for example, the business rules for the status change of an entity Order is performed inside the entity rather than having it sparse through multiple services, this allows having consistent entities independent from the use case. To have a complete domain entity, there are multiple practices to follow, such as:

    - **Alignment with DDD:** Entities must reflect business concepts, using the ubiquitous language (Ramalingam, 2020).

    - **Revealing method names:** Instead of having unexpressive getter and setter methods that do not reflect the intention and impact of the process, method names must reveal the performed business action, for example, *applyDiscount* instead of *setDiscount*.

    - **Encapsulation of behavior:** The entities must perform their business logic and enforce their invariants.

- **Domain Services:** They are helper classes that implement business logic related to multiple entities.

- **Driven Port:** It is the contract defined by the core layer to communicate with the external world without leaking infrastructure details. The application and domain layers are just aware of this contract; the implementation behind it is responsible for the other classes.

- **Driven Adapters:** They are responsible for transforming domain objects into the format required by the infrastructure, which can include a database, external API, or message broker.

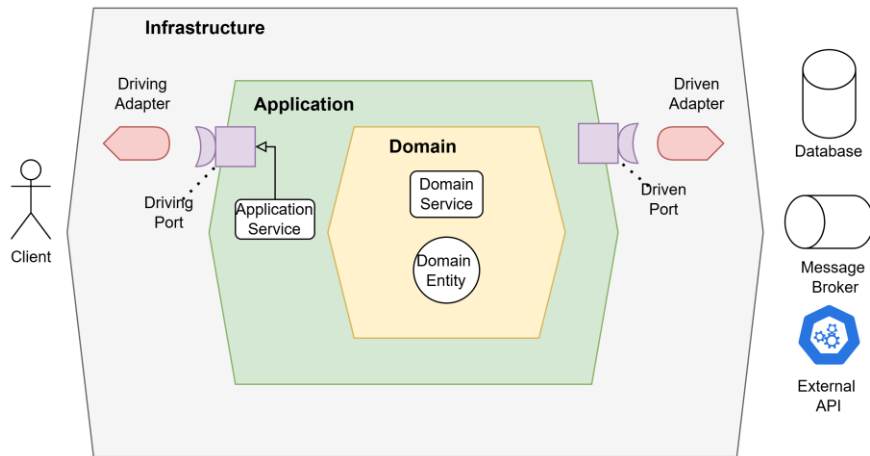

Figure 2.4 Hexagonal architecture

2.5.2 Database per Service

Database per service is a commonly used pattern in microservices architecture. It is based on the single responsibility principle, where each microservice is responsible for the lifecycle of specific domain entities, serving as the source of truth and the communication channel for them (Richardson, 2017). It provides several advantages, such as:

- Knowing where an entity is being modified reduces the time spent finding errors and performing modifications.
- Freedom to choose the database that is the best fit for the feature.
- Reduce the risk of a single point of failure.
- The scaling requirements for each data store can be defined independently.

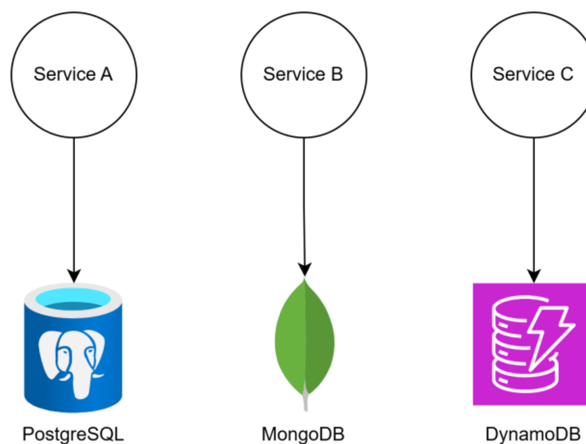

Figure 2.5 Database per service pattern

2.5.3 CQRS

Command Query Responsibility Segregation (CQRS) is a pattern used to decouple command instructions, such as write, update, and delete, from query instructions, which are used to retrieve information (Fowler, 2011). This distinction is made to optimize retrieval operations without compromising the constraints enforced by the command side. The main benefits of this pattern are:

- **Independent Optimizing:** Each model can be optimized independently according to the business requirements. For example, while the command must preserve consistency and enforce invariants, the main objective of the query is to retrieve data fast, without worrying about consistency, since the other side validates it.
- **Ad-hoc schema:** The schema for the query side can be made according to the required read representations, simplifying queries.
- **Independent scaling:** The command and query can be scaled independently for business cases where the load is different. For example, this approach is commonly used in e-commerce, where product retrieval is much more common than product creation.

While CQRS can be implemented in multiple ways, such as having a serverless function that is triggered by a database change, or by custom database features, such as ingesting data from other databases, the most common way is to publish an event through a message broker after saving the information, this message is received by the query microservice, which will replicate the information in their data store.

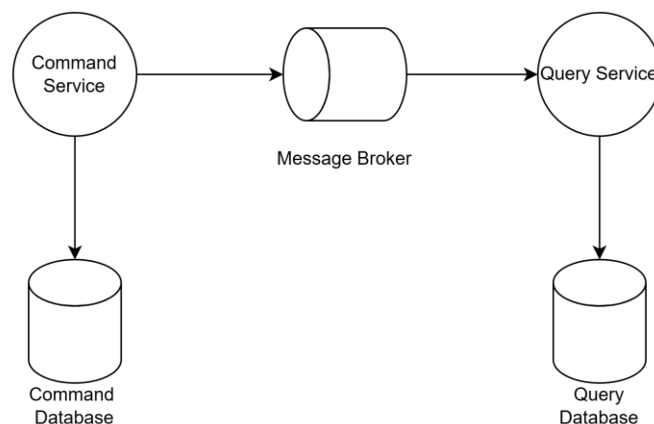

Figure 2.6 CQRS pattern

The process of replicating the information in the query data store is not instantaneous; it takes some time while sending, receiving, and storing the information. During this period of time, which is typically in the range of milliseconds, the information is not consistent. This trade-off is known as eventual consistency and should be considered when making design decisions to select the appropriate patterns.

2.5.4 Saga

In distributed architectures, such as microservices, managing consistency becomes a challenge, since a single transaction can be spread across multiple components.

The saga pattern is a sequence of transactions. When an error occurs while performing a distributed operation, compensating transactions are used to maintain the application in a consistent state (Azure, n.d.).

There are two ways to implement the saga pattern:

- **Choreography:** Each service publishes an event after performing a local transaction, which the next service in the chain will consume.

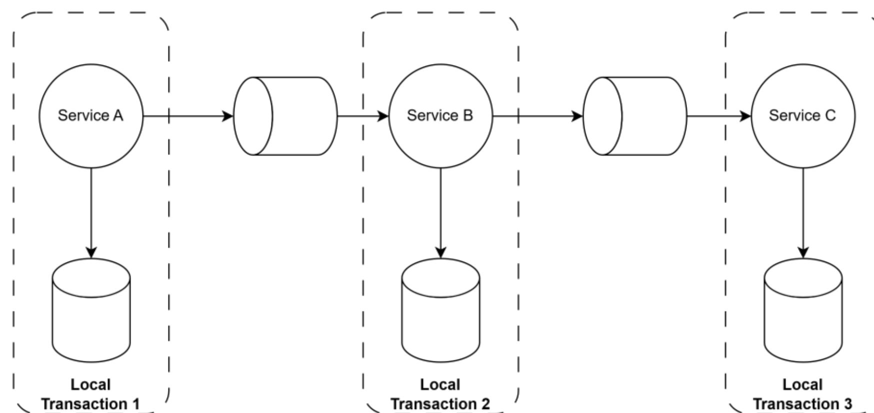

Figure 2.7 Saga pattern with choreography

- **Orchestrator:** A service is in charge of performing the events to trigger the local transactions.

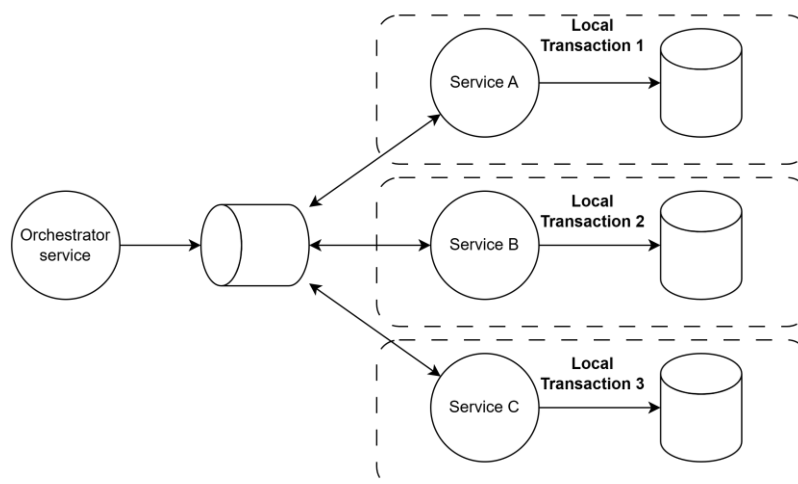

Figure 2.8 Saga pattern with orchestration

2.5.5 Circuit Breaker

One of the microservices principles is design for failure, as having multiple moving parts means that one of them may be down. The Circuit breaker pattern prevents calling a service that has been failing to give it some time to recover, avoids unnecessary calls, and reduces the response time, indicating that the service is not operational (Fowler, 2014).

The circuit breaker pattern has the following states:

- **Closed:** The target service is correctly working; in this case, the circuit breaker is just an observer to count the failures.
- **Open:** The number of failures has passed the threshold, and the circuit breaker blocks any calls to the target service.
- **Half-open:** After a period of time, the circuit breaker starts allowing service calls to verify if the failure has been resolved; if it is resolved, the circuit breaker will transition to the closed state.

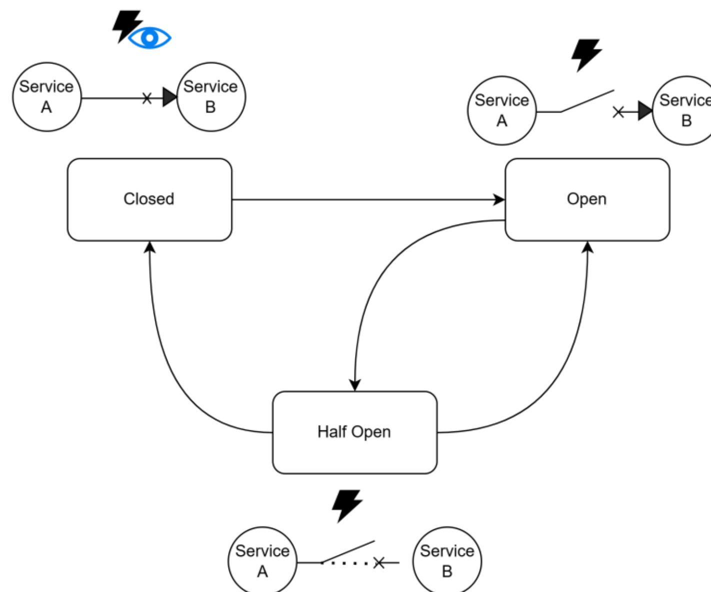

Figure 2.9 Circuit breaker pattern

2.5.6 Event-Driven Architecture

Event-driven architecture (EDA) is a paradigm in which services communicate with each other asynchronously through the exchange of events. It uses the same elements as asynchronous communication, such as producers, consumers, and message brokers, in combination with domain-driven design concepts, such as domain events, which are meaningful business situations like *OrderPlaced* or *InventoryReserved*. The main benefits of EDA are:

- **Loose coupling between services:** The producer does not know which consumers are receiving its messages, thereby adding a new consumer does not require changes in the producer.
- **Reduced response time:** Since the producer does not block waiting for a response from consumers when it sends a message, the server can respond to requests more quickly.
- **Asynchronous processing:** Long-running or resource-intensive tasks can be processed asynchronously, saving resources by not maintaining a connection while waiting for a response.

2.5.7 Transactional Outbox

The Transactional Outbox pattern is used to maintain the system's consistency, treating the publication of a message and writing to the database as a single transaction.

In distributed systems, it is common for a service to perform changes to a database and then publish a message (a dual write operation). However, if the message publication fails and the data has already been saved in the database, the system becomes inconsistent. If the message is first published and the data saving fails, all consumers will perform their logic, which also leaves the system inconsistent.

To avoid this problem, the event can be stored in the database in the same transaction, while other component (a serverless function, a scheduled job, etc.) will be in charge of publishing the message to the broker, once the message is published, the record can be deleted or modified to state that it was already sent.

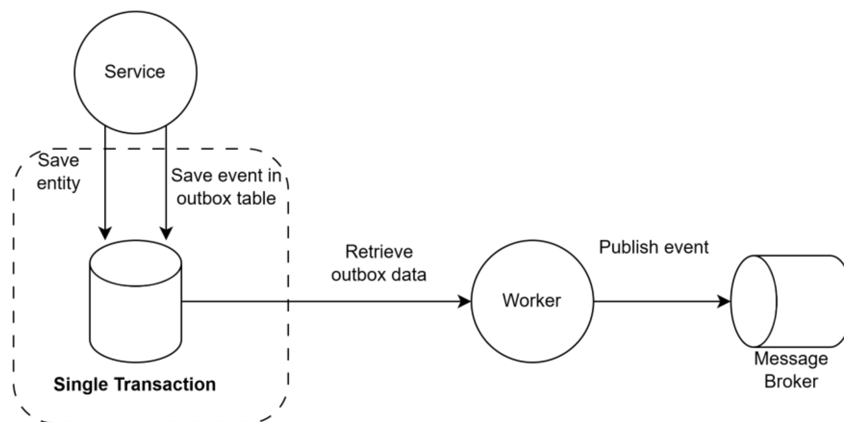

Figure 2.10 Transactional outbox pattern

2.5.8 Anti-corruption layer

The Anti-Corruption Layer (ACL) pattern is used to isolate the business domain from external concepts; it translates these concepts from external services into the domain's ubiquitous language. It prevents the mixing of concerns between systems, keeping the application agnostic and avoiding the system's degradation into a Big Ball of Mud (Foote, B., & Yoder, 1999). The ACL combines the concepts of the Adapter pattern and Domain-Driven Design.

2.5.9 Backends for Frontends

Backends for frontends (BFF) pattern adds a new service that is intended to reduce chatty communication between frontend applications and backend services. It moves the responsibility of performing multiple calls to the server, thereby avoiding the leakage of business logic and service orchestration to the client. The main responsibility of the frontend is to present the information, not to aggregate data from multiple sources.

Another key benefit of having a BFF service is that it is specifically designed for the client, for example, it can be beneficial having one backend that serves request made through the web and another backend that serves request made through the mobile application, making this, each response schema can be created specifically to satisfy the requirements of each clients, instead of having a massive response that returns all the information due to the multiple clients that it serves, which causes leakage of information and waste resources.

2.6 Testing Microservices

This section describes the testing practices used in microservices to ensure the system's correctness.

2.6.1 Testcontainers

One of the goals when testing an application is to have a test environment as close as possible to the production environment, since mocking the infrastructure or using embedded technologies can allow errors that will be detected by final users in production, which is costly.

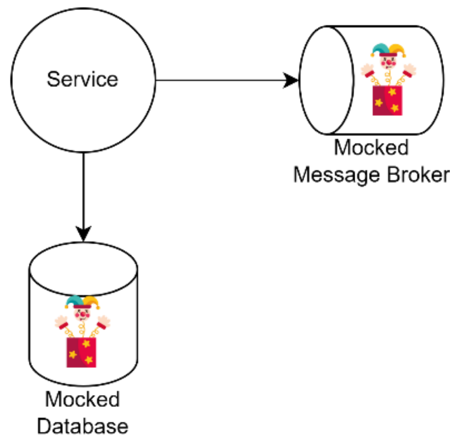

Figure 2.11 Traditional way of mocking infrastructure

Testcontainers is a library that enables developers to start containers in test classes in an easy way, which allows testing with the real infrastructure without worrying about configuration details (Testcontainers, 2023).

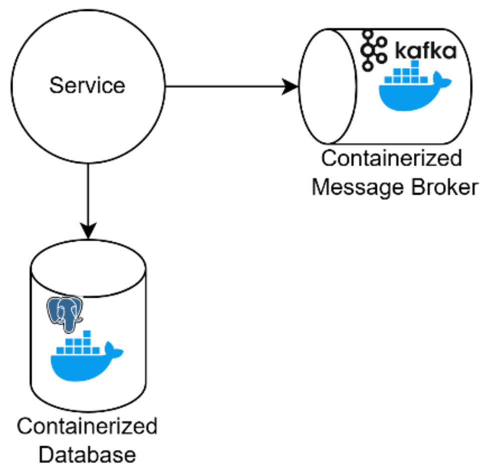

Figure 2.12 Perform tests using Testcontainers

Testcontainers provides multiple benefits to ensure that it improves the system's reliability and reduces the maintenance costs. These benefits are:

- Testing the real application behavior instead of relying on mocks or stubs
- Catch bugs early
- Reduces configurations, integrating with test engines like Junit to control the containers' lifecycle

2.6.2 Test-driven development

Test-driven development (TDD) is a software development approach of writing software where all new features start with a failing test. It improves the correctness of the application, since it forces the developer to understand the requirements, the use cases, and the possibilities of an action before writing any line of code. It aligns with the Extreme Programming (XP) principle, “You Aren’t Gonna Need It” (YAGNI), where only the minimum necessary code is written to make the test pass.

Test-driven development is a cycle composed of three phases:

1. **Red:** It is the initial step, where a test for the functionality that is going to be implemented is written; it will fail since there is no implementation yet.
2. **Green:** The developer writes the minimum necessary code to make the test pass
3. **Refactor:** The developer performs changes to clean up the code eithout changing the behavior, which can include the separation of methods or classes, the implementation of design patterns, and other similar improvements.

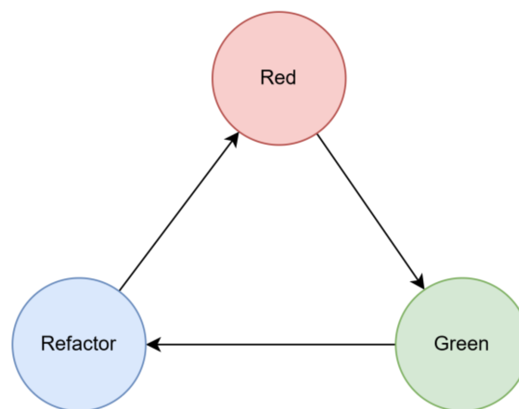

Figure 2.13 TDD cycle

2.6.3 Contract testing

Testing multiple components in distributed applications is a complicated task. Some teams choose to mock the services that communicate with the tested one while performing tests. However, this approach is correct for unit tests; performing integration tests with mocks or stubs can lead to incoherent behavior when all the services are deployed in production. Common errors that can occur are:

- **Incorrect API calls:** Bad formed URIs, missing header or invalid parameters
- **Schema mismatches:** The tested service expects attributes that the external API does not provide.

- **Asynchronous communication errors:** consumers subscribed to the wrong topics, or producers are not sending the data that it is supposed to send

All of these problems are undetectable when mocking third-party services, since it is the test developer who is making assumptions about how the external service works.

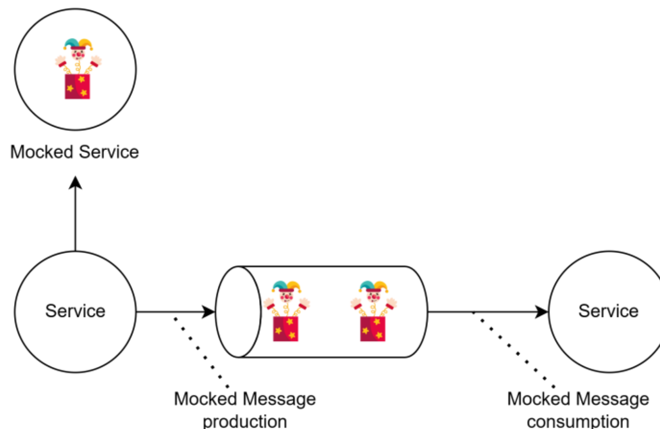

Figure 2.14 Traditional wat of testing with mocking communication

Contract testing defines a way of standardizing the test, based on verified stubs.

Consumer-driven contract is one of the most popular ways of performing contract testing. This is a collaborative way of building the contract, i.e., an agreement on communication.

One of the most used libraries to perform contract testing in Spring Boot applications is **Spring Cloud Contract,** which, according to its documentation, "*moves TDD to the level of software architecture*" (Spring Cloud, n.d.). The steps to perform the tests are:

1. The producer creates the contract, which specifies the request and response formats.
2. Producer tests are automatically generated to verify that the service operates according to the contract.
3. Once the producer tests are passed, verified stubs are created to test the other side.
4. The consumer can test their logic using either the stubs or the producer contract to start a server that will answer the verified stubs.

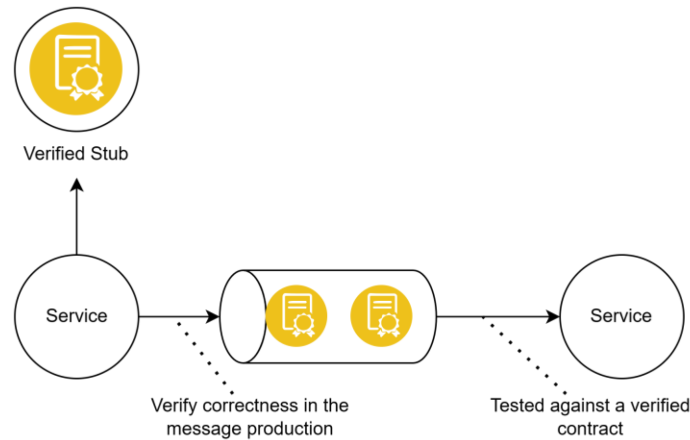

Figure 2.15 Testing using verified contracts

2.7 System Observability and Monitoring

Observability and monitoring are critical in distributed applications, as multiple components require a centralized tool to manage the system's internal state. The main benefits are:

- **Debug in real-time:** Observability tools provide metrics such as component response time, response errors, and resource usage.
- **Performance optimization:** Analyzing the components' metrics is required to make data-informed decisions and identify bottlenecks in the application.
- **Centralized Management:** Some projects can have hundreds, even thousands of microservices, and analyzing each of them separately is almost impossible.

The most important components of observability are logs, metrics, and traces.

2.7.1 Health Endpoint Monitoring

Container orchestrators provide a way to automatically check the application status so they can know if the application is healthy and ready to receive requests. In Kubernetes, liveness and readiness probes are responsible for performing this verification.

Liveness probe is used to check if the application is healthy; if the application does not answer correctly after a custom number of calls, Kubernetes will kill the pod and start a new one. On the other hand, the readiness probe is used to check if the application is ready to receive traffic; otherwise, Kubernetes won't pass traffic to this pod.

In a Spring Boot application, the Actuator module provides the health endpoint, which automatically checks that the application is in a healthy state through auto-configured health indicators. It verifies that:

- The connection to the data source can be obtained

- There is sufficient disk space
- It will check the connection with popular technologies when they are used, such as RabbitMQ, Redis, Mongo, etc.

2.8 Cloud Computing Foundations

This section provides a brief definition of cloud computing and the Azure ecosystem, followed by a description of each cloud service used in the system.

2.8.1 Introduction to Cloud Computing

Cloud computing provides services that people can use, reducing the maintenance costs. The main benefits of using cloud services are:

- **Cost efficiency:** The cloud services principle is to pay for what you use. Instead of having a proprietary cluster with unused resources.
- **Scalability and Flexibility:** Cloud providers have the resources and the technology to scale services fast, whether it is a database or a Kubernetes cluster.
- **Reliability:** Cloud providers' availability zones are a way to ensure uptime; if an error happens in an availability zone, the services can be up and running in another zone to ensure zero downtime.
- **Security:** Cloud providers invest continuously in improving the security of their services, and some of them include security features by default.

2.8.2 Azure Ecosystem

Azure is one of the most popular cloud providers, offering well-known services such as Azure Kubernetes Service (AKS) and Cosmos DB. According to the 2024 Stack Overflow survey, Azure has a 30% market share, making it the second most used cloud provider among professional developers.

2.8.3 Azure Services

This section briefly describes the Azure services used in this project.

Service	Category	Description	Source
Azure Database for PostgreSQL flexible server	Databases	Database-as-a-service based on Postgres.	Azure (n.d.)

Azure CosmosDB for MongoDB	Databases	Database-as-a-service based on MongoDB.	Azure (2024)
Azure AI Search	AI	Retrieval system with AI capabilities, such as RAG and LLM integrations.	Azure (2025)
Storage Accounts	Storage	Storage service to manage Binary Large Objects (BLOBs), such as images.	Azure (2025)
Azure Cache for Redis	Databases	In-memory data store based on Redis.	Azure (2024)
Event Hubs	Messaging	Data ingestion service that allows integration with Kafka.	Azure (2024)
Azure Container Registry (ACR)	Compute	Managed registry based on Docker Registry to store and manage container images.	Azure (2024)
Azure Kubernetes Service (AKS)	Compute	Managed Kubernetes service used to deploy containerized applications.	Azure (2024)
Application Insights	Monitoring	Provides features to monitor application performance, such as the application map, live metrics, and performance view.	Azure (2025)
Azure Load Testing	Testing	Generates high-scale load to the application, abstracting the details of the underlying testing tool	Azure (2024)

3 Requirements Specification

This section describes the platform requirements through user stories. Since the project is intended to be as close as possible to an agile project, user stories are the best option for capturing user needs; also, the acceptance criteria enable more details and the specific elements that must be met to consider a user story finished.

The user story's priority and acceptance criteria in a real environment can vary throughout the project, since just the user stories at the top of the backlog are described thoroughly. As the project progresses and the backlog refinement ceremonies are held, the user stories are polished and completed. For those reasons, the acceptance criteria of the user stories exposed below are with the necessary granularity to create the platform architecture, however, if the project is going to be developed, it would be required to hold the backlog refinement ceremonies to add more acceptance criteria, specifically the ones related to user experience.

As mentioned in chapter 1, there are many e-commerce systems in the Colombian market, but none of them specialize in coffee, despite being one of the most important products in the country.

The platform's objective is to be the reference marketplace if someone wants to buy products related to coffee, it can be coffee beans, ground coffee, seeds, tools to prepare coffee such as Chemex, French Press, etc., and all the products that are derived from coffee, like coffee liqueur and coffee food. For that reason, the system must be able to grow efficiently and smoothly in product variety.

To improve customer experience and make the marketplace more accessible to all users, a chatbot that recommends products will be implemented. This ensures that customers do not need extensive knowledge about coffee to find the right product.

The platform has three main roles that will be described in further sections, which are the administrator, who has the authority to suspend or delete products; the seller, who offers products to customers; and the customer, who is the one who will buy products in the application.

Due to all the possible products that the platform can have, and to promote growth in the long term, another responsibility of the administrator is to create categories and tags, which are used to classify the products.

Given the geographically dispersed nature of coffee producers, the platform cannot automatically track the order's shipment updates, for example, a producer located in the department of Huila uses different shipment companies than a producer located in Antioquia. As a result, it is the seller's responsibility to mark the order as shipped.

In conclusion, the requirements specification focuses on creating a specialized e-commerce platform for Colombian coffee products, where the user stories provide flexibility to refine them throughout the project's development.

3.1 Product Characteristics

This section provides a detailed explanation of core product concepts, including categories, tags, and variants. These descriptions are intended to be a clear baseline for subsequent user stories, which can be interpreted with greater clarity.

3.1.1.1 Product Category

From a user experience perspective, categories are the main way to classify products. The application will include a section where users can browse all available categories. For example, a category like “Accessories” may also have subcategories, such as “Containers”. Figure 3.1 presents a sketch illustrating how the categories might be shown in the application.

Categories will be created by administrators, as described in further user story AT-05.

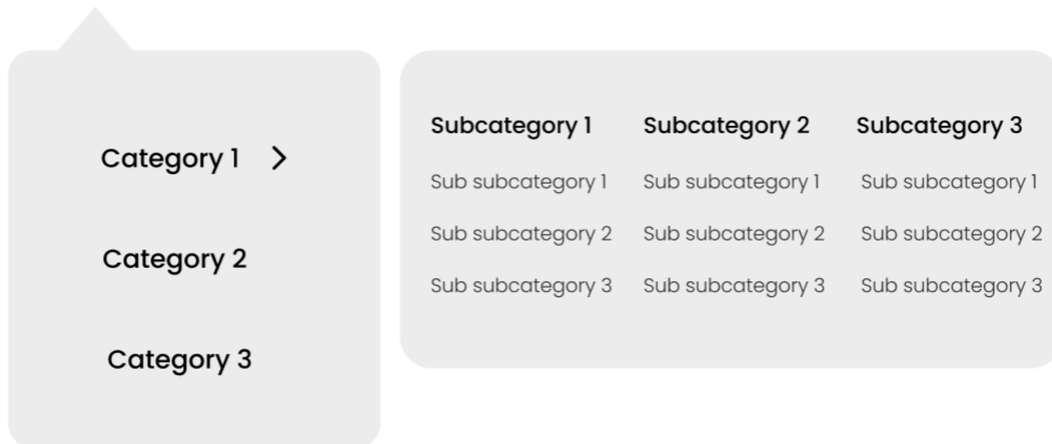

Figure 3.1 Categories Layout

3.1.1.2 Tags

Tags are a key-value feature used to describe the product characteristics. For example, the altitude at which the coffee was produced could be represented as a tag, where the key is “Altitude” and the value is “1400-1600 meters”.

The administrator is responsible for tag creation, defining the keys, and providing a list of possible values. Products can have an unlimited number of tags.

Product Characteristics

Region	Caldas
Altitude	1400 - 1800 m
Processing	Washed
Variety	Arabica - Castillo

Figure 3.2 Tags for a Product

3.1.1.3 Product Variations

Product variations are a key element in the flexibility of the product. Sellers can avoid creating a whole new product that differs from another just in a few attributes. It helps sellers keep their products organized and helps customers find the specific variant faster since the variants will be shown on the same product details page.

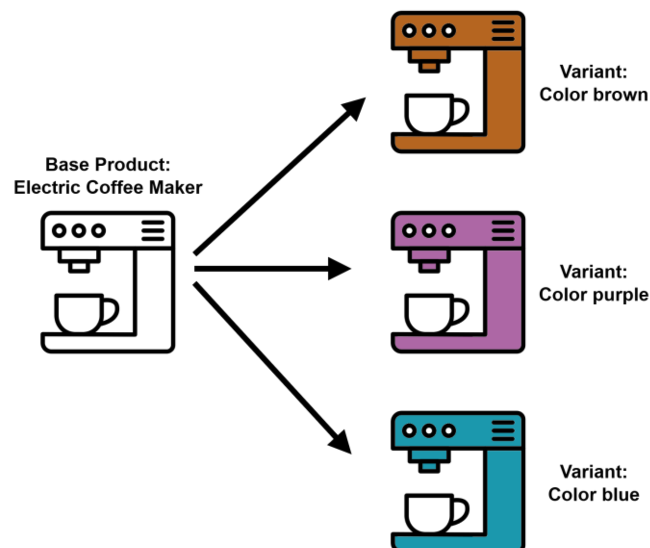

Figure 3.3 Variants Example[7]

[7] Icon by Nawicon from https://www.flaticon.com

Products can have more than one attribute that changes; in this case, sellers can create variants based on the combination of the dynamic attributes. Figure 3.4 illustrates how a coffee product with multiple attributes that can have different values, in this case, weight and grind type. If there are two possible values for weight and another two possible values for grind type, then the seller has to create four different variants to express all the possible combinations.

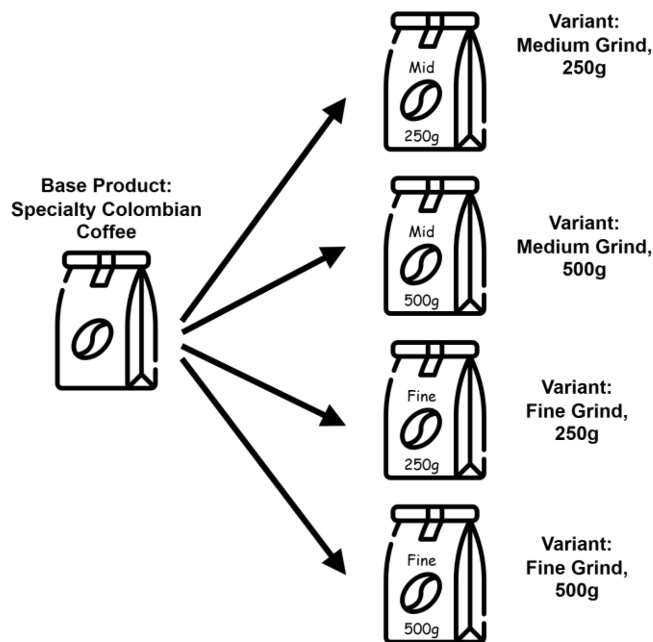

Figure 3.4 Product with multiple dynamic attributes[8]

Variations offer a flexible way to organize the products; however, they are not mandatory. Each seller can decide how they want to manage their products. For example, one seller might choose to create eight variants for one product, while another may prefer to list eight products with no variants.

3.2 Issue Management

Customers can report issues if they encounter any problems with the application's use. For example, if a customer forgets their password and cannot change it because they forgot the recovery email, they can create an issue to request help with the process. Another example is if a product is suspended and the seller disagrees with the decision, they can create an issue to ask for the reversal of the suspension.

---

[8] Icon by Vector Portal from https://www.flaticon.com

A customer support platform is a complex system with its business requirements that potentially exceed the scope of the e-commerce platform itself. Building and maintaining such a system in-house would deviate from the core e-commerce functionalities. To maintain a strong focus on delivering a robust platform, it is more effective to externalize the issue tracking to a well-established third-party service such as Zendesk, Freshdesk, Gorgias, etc. so that, in the e-commerce platform, the unique actions that can be related to issues are the creation, seeing its status, and receiving the response. The focus of the e-commerce application will be providing cohesive integration with the issue platform.

3.3 Marketing Campaigns

The marketing campaigns are defined in a third-party email system, such as Mailchimp, Omnisend, or Klaviyo. The definition of mail templates, content strategy, and campaign configuration is out of scope. It is assumed to be managed by the marketing team using the tools mentioned earlier.

While the system saves relevant information such as user notification preferences and personalized product recommendations, the synchronization with the external system via API calls is not part of the minimum viable product. In conclusion, the information is present in the system; however, the synchronization needs to be refined and implemented in later development stages.

3.4 User Roles

This section outlines the designated roles that will use the application

- **Administrator:** Is the one in charge of verifying that products comply with the company policies; therefore, it can suspend and delete a product publication.

  To keep the e-commerce updated, the administrator can create and modify categories and tags. These changes will be reflected in both sellers and customers.

  Another responsibility of the administrator is to assist users when needed, which is part of the customer support user stories, described in section 3.5.11. To perform this task, the administrator has access to the user information and all orders that the user has made.

- **Seller:** Is the one in charge of creating products and keeping them updated. They can update the product stock, create new products and variations, delete products, create discounts, etc. Also, the platform provides the ability to manage their order efficiently, where they can view sell analytics and download the orders according to desired filter criteria.

- **Customer:** This is the principal role in the application since it will perform the purchases, which is the central part of the platform.

  Customers have three main functionalities, the first one is the one related to products, where they can list, search, filter, and buy them; the second one is related to customer support, where they can create an issue and get help from the administrators; the last one is the interaction and history system, where they can review past orders and rate purchased products.

There are a wide variety of coffee plants like Arabica or Robusta, and below that, there are multiple types of Varietals, which are subtypes of the coffee plant, like Caturra or Castillo, even more, many farms produce their specialty coffee, which is different from others because of the process method, terrain, irrigation, etc. As a result, it is common for a coffee producer to buy coffee from other farms. For example, a local producer who has a cafeteria would want to buy other coffee besides their own to have a varied menu for its customers.

For these reasons, the application assigns both customer and seller roles to the user once they have registered, so they can buy and sell without having to create two accounts.

3.5 User Stories

This section states the functional requirements of the system, described in user stories and acceptance criteria.

3.5.1 User Onboarding

<table>
<tr><td>UR-01</td><td>Manual User Registration</td><td>Priority: Medium</td></tr>
<tr><td colspan="3">As a customer, I want to register on the platform, so that I can save my information and purchases.</td></tr>
<tr><td colspan="3">1. Given I am filling in the registration form, when I leave a required field empty, the system highlights the invalid field and shows a message indicating the problem below it.
2. Given I am filling in the registration form, when I provide invalid data for a field, the system highlights the invalid field and shows a message indicating the problem below it.
3. Given I am filling in the registration form, when I provide an already registered email, the system shows a message indicating that.
4. Given I have completed the registration form successfully, when I submit the form, then the system sends a verification email with a link to my registered address.
5. Given I am on the registration page, when I enter all the required fields, then the system shows a message indicating that a verification email was sent.
6. Given I have registered and verified my email, when I return to the platform, then I see a confirmation message and I am redirected to the login page.</td></tr>
</table>

<table>
<tr><td>UR-02</td><td>User Registration with a Third-Party Application</td><td>Priority: Low</td></tr>
<tr><td colspan="3">As a customer, I want to register on the platform using my existing credentials from another application, so that I can quickly save my information and purchases without creating a new password.</td></tr>
<tr><td colspan="3">Given I am on the registration page, when I click the icon of the third-party application, then, the system redirects me to the application authentication page.
Given I am redirected back to the platform after the authentication in the third-party application, then the system creates a new user account using the data provided by the application, shows a success message indicating that the account was created and redirects me to the login page.
Given I attempt to register using a third-party application, when the application mail is already associated with an existing account, then the system shows a message indicating that.</td></tr>
</table>

<table>
<tr><td>UR-03</td><td>User Manual Login</td><td>Priority: Medium</td></tr>
<tr><td colspan="3">As a customer, I want to log in to the platform using my credentials, so that I can securely access my account and its functionalities.</td></tr>
<tr><td colspan="3">Given I am on the login page, when I enter my registered email and password, then the system authenticates me and redirects me to the platform's main page.
Given I am on the login page, when I enter an invalid email or password, the system shows an error message indicating that either the email or the password is incorrect.
Given I try to log in multiple times with incorrect credentials, when I exceed the number of permitted attempts, then the system locks my account, shows a message indicating the blocking and sends an email notifying it.</td></tr>
</table>

<table>
<tr><td>UR-04</td><td>User Third-Party App Login</td><td>Priority: Low</td></tr>
<tr><td colspan="3">As a customer, I want to log in to the platform using my third-party app credentials, so that I can securely access my account and its functionalities without needing to remember a new password.</td></tr>
<tr><td colspan="3">Given I am on the login page, when I click the third-party app button, then the system redirects me to the app’s authentication page.
Given I am redirected back to the platform after the third-party app authentication, when there’s no account linked to a platform account, the system will show a message indicating that there is no account associated.
Given I am redirected back to the platform after the third-party app authentication, when there’s an account linked to a platform account,</td></tr>
</table>

the system authenticates me and I am redirected to the main page of the platform.

UR-05	**Password Recovery**	Priority: Low
As a customer, I want to be able to recover my password if I forget it so that I can regain access to my account and continue using the platform.		
1. Given I am on the login page, when I click the “Forgot Password” link, then I am redirected to a page where I can enter my email address to start the recovery process. 2. Given I am on the password recovery form, when I enter my registered email, then the system sends a password reset email with a password reset link. 3. Given I am on the password recovery form, when I enter a non-registered email, the system shows a message indicating that there’s no account associated with the provided email. 4. Given I receive the password reset email, when I click on the link after it has expired, the system shows a message indicating that the link is no longer valid and redirects me to the forgot password page. 5. Given I receive the password reset email, when I click on the link, then I am redirected to a form when I can enter a new password. 6. Given I am resetting my password, when I enter a new password that does not meet the requirements, then the system highlights the fields and indicates the problem below it. 7. Given I am resetting my password, when I enter a new valid password, then the system shows a message indicating that the password has been changed and redirects me to the login page.		

UR-06	**Log Out**	Priority: Medium
As a customer, I want to log out of my account so that I can end my session and prevent unauthorized access.		
1. Given I am logged into the application, when I click the “Logout” button, then the system logs me out and redirects me to the main page.		

3.5.2 Profile Management

PRM-01	**Update Profile**	Priority: Medium
As a marketplace user, I want to update my personal information, so that I can have my information updated.		
1. When I navigate to the profile management section, then the system redirects me to the desired section and shows the editable profile information, such as name, email, contact details, profile picture, etc. in editable fields. 2. Given I am in the profile management section, when I modify some information and click save, then the system saves the information and shows a confirmation dialog.		

PRM-02	**Create Delivery Address**	Priority: Medium
As a customer, I want to add a new delivery address so that I can receive my orders at the desired location.		
1. When I navigate to the delivery address section, then the system redirects me to the form where I can fill in the necessary information, such as street line 1, street line 2, postal code, city, department, country, and address observations. 2. Given I am on the “Create Delivery Address” form, when I fill in all the information and click save, then the system saves the information and shows a confirmation message. 3. Given I am on the “Create Delivery Address” form, when I click save without filling in the required fields, then the system doesn’t save the information and shows an error indicating that the required fields are missing.		

PRM-03	**List Delivery Addresses**	Priority: Medium
As a customer, I want to list my delivery addresses so that I can manage them easily.		
1. When I navigate to the “Addresses” section, then the system redirects me and shows the list of saved addresses. 2. Given I don’t have any address saved, when I navigate to the “Addresses” section, then the system shows a message indicating that there are no addresses available.		

PRM-04	**Update Delivery Address**	Priority: Low
As a customer, I want to update my delivery address so that it is accurate.		
1. Given I am in the “Addresses” section, when I click the update icon next to an address, then the system redirects me to the form to update it.		

2. Given I am on the form to update my address, when I click on an address field, then I can update the existing information by deleting or modifying it. 3. Given I am in the confirmation dialog to update an address, when I click "Update", then the system updates the existing address and shows a message indicating that the change was performed.		

PRM-05	**Delete Delivery Address**	Priority: Low
As a customer, I want to delete a delivery address so that I can no longer receive products in an old address.		
4. Given I am in the "Addresses" section, when I click the delete icon next to an address, then the system displays a confirmation dialog asking if I want to proceed with the deletion 5. Given I am in the confirmation dialog to delete an address, when I click "Delete", then the system deletes the address and shows a message indicating that the deletion was performed.		

PRM-06	**Manage Notification Preferences**	Priority: Low
As a customer, I want to manage my notification preferences, so that I can receive the information that is valuable to me.		
1. When I navigate to the notification preferences section, then the system displays a list of notification categories, such as marketing campaigns, order updates, order confirmation, and issue updates, with toggle options for each channel (email, in-app notifications).[1] 2. Given I am in the "Notification Preferences" section when I interact with the toggle, then the system saves the preference and displays a pop-up message indicating that the preference was updated. [1.] The possibility of modifying the notification categories by the administrator or by a new custom role should be considered in future iterations. So far, the categories are stored in the database, but without the possibility of changing them.		

3.5.3 Product Management

PM-01	**Create New Product**	Priority: High
As a seller, I want to create a new product so that I can make it available for purchase.		
1. Given I am in the seller's product dashboard, when I click the button "Create new product", then the system redirects me to the product creation page. 2. Given I am on the product creation page, then the system shows me a form to fill in all required information, which includes product name, description, category, stock, tags, visibility, images, variants, etc.		

3. When I am on the product creation page, then the system shows me an optional field to put the minimum and maximum amount that can be purchased.
4. Given I am on the product creation page, when I try to create a product with fewer than three images, then the system stops the product creation and shows an error indicating that the minimum number of product images is three.
5. Given I am on the product creation page, when I fill in all the required information, which includes product name, category, tags, visibility, description, stock (either for the product if it does not have variants, or for each variant) and at least three images, then the system shows me a list of possible tags that I haven't added and could be related to my product.
6. Given I am on the tags suggestion screen, when I accept or reject the suggestion, then the system creates the product, sends me an email indicating the product creation, shows a successful message, and redirects me to the product's dashboard.
7. Given I am on the product creation page, when I do not fill in all the required information and click the "Create Product" button, then the system shows an error message indicating it.
8. Given I am on the product creation page, when I try to create a product with the same name as an existing one in my account, then the system shows an error message indicating that a product with this name already exists.
9. Given I am uploading images or videos of the product, when I select and upload the files, then the files should be successfully attached to the product and displayed as previews.
10. Given I am on the product creation page and my product has variants, when I click the button "Create Variant", then the system shows the new fields to create it, which are the variant name and the list of options, which is composed by the option name and an icon to attach an image to it.
11. Given I am in the variant creation section of the product creation page, when I click the button to add another option for the variant, then the system shows me the fields to create the information for the new variant option, which are the option name and the icon to attach an image to it.
12. Given I am on the product creation page, when I have created a variant, then the system blocks the button to create a variant, since the product can only have one variant with multiple options, for example, the product cannot have a variant for colors with options red and blue and another variant for weight with options 250g and 500g, in this case, the seller have to create one variant with the combination of all four possibilities. (see section 3.1.1.3)

PM-02	**Update Product Information**	Priority: Medium
As a seller, I want to update the details of the product so that I can reflect the correct and updated information for my customers.		

1. Given I am in the seller's products dashboard, when I click the desired product to update, then the system redirects me to the "product update page", where I can see the product details and update it. 2. Given I am on the "Product Update Page", when I view the update form, then the fields description, visibility, tags, category, stock or variants stock, minimum order, maximum order, variants, and images should be editable, and the product name should be displayed as a read-only. 3. Given I am on the product edit form, when I click the "Save Product" button, then the system should save the changes and reflect them in the product catalog. 4. Given I am editing the product images, when I delete some images and the number of product images is below three, then the system will show a warning indicating that the minimum number of product images is three. 5. Given I am updating the product details, when I leave the required fields blank or enter invalid data, then the system shows an error message and prevents saving the product. 6. Given I am updating the product details for a product with no variants, when I click the button to add a variant, then the system displays a warning message that the actual stock will be deleted and now the stock must be managed at the variant level. 7. Given I am in the warning message to create a variant for a product with no variants, when I click "Accept", then the system removes the current product stock and creates the variant.

PM-03	**Delete Product**	Priority: Medium
As a seller, I want to delete a product from my catalog, so that I can remove irrelevant or outdated products from my store.		
1. Given I am on the seller's products dashboard, when I click the delete icon of a product, then the system shows a confirmation message telling me if I'm sure to perform the deletion. 2. Given I am on the deletion confirmation message, when I click "Delete", then the system shows a message indicating that the deletion was performed, and reflects the changes in the platform.		

PM-03	**Delete Variant**	Priority: Medium
As a seller, I want to delete a variant from a product, so that I can keep my product stock updated.		
1. When I navigate to the product details, the system shows the list of variants with an icon to delete each of them. 2. Given I am in the product details, when I click the icon to delete the variant, then the system shows a warning message asking me if I'm sure to delete the variant. 3. Given I am on the warning message to delete the variant, when I click "Delete", then the system deletes the variant.		

4. Given I am on the warning message to delete the variant, when I click "Cancel", the system does not delete the variant.

PM-05	**Add Discount**	Priority: Low
As a seller, I want to apply discounts to a product, so that I can attract more customers and boost sales.		
1. Given I am on the product details page, when I click the "Add promotion", then the system shows the "Product Promotion" form. 2. Given I am on the "Product Promotion" form, when my product has variants, then the system displays a section with checkboxes for each variant and an additional "All variants" checkbox. The "All Variants" checkbox is selected by default, allowing me to choose the ones that will be included in the promotion. 3. Given I am on the "Product Promotion" form and my product has variants, when I select the "All variants" option, then the system selects all individual variant checkboxes and disables their modification. 4. Given I am on the "Product Promotion" form and my product has variants, when I unselect the "All variants" option, then the system unselects all individual variant checkboxes and enables their modification. 5. Given I am on the "Product Promotion" form, when I fill in all the required fields, which are the start date, end date, discount percentage, or discount amount, the variants where it will apply, a checkbox to indicate if the discount requires code and, if it does need it, the code that customers have to write to apply it, then the system saves the information and reflects it in the catalog when the promotion start. 6. Given I am on the "Product Promotion" form, when I leave a required field blank, then the system highlights the field and shows the error below it		

3.5.4 Product Browsing and Discovery

PD-01	**List products**	Priority: High
As a customer, I want to view a list of products, so that I can find what I want to buy.		
1. Given I am on the application, when I click on the "Product Catalog" page, then the system displays a list of all available products with their most important information, such as name, price, images, and availability.		

PD-02	**See product details**	Priority: High
As a customer, I want to view detailed information about a product, so that I can understand it to make an informed purchase.		
1. Given I am on the product listing page, when I click on a product, then the system opens the product detail page. 2. Given I am on the product detail page, when I view the product details, the system provides me the product information, such as name, full description, number of reviews, average rating, images, price, variants, stock, etc. 3. Given I am on the product detail page, when I see the product's price, then the system shows the discounts if there are any available.		

PD-03	**Filter products**	Priority: Medium
As a customer, I want to filter products so that I can find the products faster.		
1. Given I am on the product list page, when I apply one or more filters, such as minimum price, maximum price, category, product name, tags, number of reviews, and minimum rating, then the system updates the product list to show only the products that meet the filter criteria. 2. Given I have applied filters, when I remove the filters, then the system reloads the product list to show all available products.		

PD-04	**Search products by image**	Priority: Medium
As a customer, I want to search for a product using an image, so that I can find it faster.		
1. Given I am on the product list page, when I upload an image of a product, then the system processes the image and shows similar products from the catalog. 2. Given I am on the product list page, when I upload an image of a product and there are no similar products, then the system shows a message indicating that no products were found.		

PD-05	**Sort products**	Priority: Low
As a customer, I want to sort products so that I can easily find the products that I'm looking for.		
1. Given I am on the product list page, when I apply a sorting option, such as price (high to low and vice versa), average rating (high to low), and name (A to Z and vice versa), then the system updates the product list to reflect the order.		

PD-06	**Receive Marketing Campaigns**	Priority: Low
As a customer, I want to receive marketing campaigns, so that I can discover relevant products or promotions that match my interests.		
1. Given I have the option to receive marketing communications when a new campaign is launched, then I should receive the campaign by email. 2. Given I do not have the option to receive marketing communications, when a new campaign is launched, then I should not receive it.		

3.5.5 Shopping Cart Management

SC-01	**Add Item to the Cart**	Priority: High
As a customer, I want to add products to my cart so that I can purchase them later.		
1. Given I am on the product detail page, when I click the "Add to Cart Button", then the system adds the selected product to my cart and confirms it with a message. 2. Given a product that has a variation, for example, a coffee can be in multiple presentations, such as 250g, 500g, and 1Kg; when I try to add the product to the cart without selecting the necessary options, then the system shows an error message indicating the options that haven't been selected. 3. Given the product is out of stock, when I try to add it to the cart, then the system prevents the action. 4. Given I have items in the cart, when I navigate through the application, then the cart persists the content until I remove them or complete the purchase.		

SC-02	**Update the quantity of products in the cart**	Priority: Medium
As a customer, I want to update the quantity of products in my cart, so that I can adjust my order.		
1. Given I have a product in my cart, when I change the quantity of it, then the cart updates the total price and displays the new quantity. 2. Given I change the quantity of a product in my cart, when the new amount is higher than the stock available, then the system does not perform the update and shows a message indicating that there is not enough stock to perform the operation. 3. Given I have a cart with some products, when I update the quantity of a product to zero, then the system removes the product from the cart.		

SC-03	**Delete a product from the cart**	Priority: High
As a customer, I want to remove products from my cart, so that I can adjust my order.		
1. Given I have products in my cart, when I click the “Remove” icon next to the product, then the system removes the product from the cart and updates the total price. 2. Given I remove a product from my cart, when the cart becomes empty, then the system shows a message indicating that there are no items in the cart. 3. Given I have products in my cart, when I click the “Clear shopping cart” at the end of the details, then the system shows a message to confirm the action. 4. Given I am in the Clear shopping cart confirmation window, when I confirm the option, then the systems remove all the products from the cart.		

SC-04	**Checkout shopping cart**	Priority: High
As a customer, I want to proceed to the checkout process, so that I can continue the process to complete my order.		
1. Given I have items in my shopping cart, when I click the “Checkout” button, then the system redirects me to the purchase window. 2. Given my shopping cart is empty, when I attempt to click the “Checkout” button, then the system displays indicating that to proceed to the checkout the cart cannot be empty.		

3.5.6 Product Purchase

PP-01	**Checkout Product**	Priority: Medium
As a customer, I want to check out an individual product, so that I can continue with the purchase.		
1. Given I am on the product details page, when I click the “Purchase Product” button, then the system redirects me to the purchase window. 2. Given I am on the product details page, and I haven’t selected all the required product options, when I click the “Purchase Product” button, then the system shows an error message indicating that there are required fields missing. 3. Given I am on the product details page, when the product is out of stock, then the system disables the “Purchase Product” button.		

<table>
<tr><td>PP-02</td><td>Fill in Card Information</td><td>Priority: High</td></tr>
<tr><td colspan="3">As a customer, I want to fill in my card information so that I can provide my payment details during checkout.</td></tr>
<tr><td colspan="3">1. Given I am in the payment section of the purchase window, when I have saved cards, then I can select a card to use for the purchase.
2. Given I am in the payment section of the purchase window, when I want to add a new card, then the system allows me to fill in the card details and save the card securely for future use.
3. Given I am in the payment section of the purchase window, when I choose not to save the card, then the system processes the payment without storing my details.</td></tr>
</table>

<table>
<tr><td>PP-03</td><td>Apply Product Discount</td><td>Priority: High</td></tr>
<tr><td colspan="3">As a customer, I want to apply a coupon during the checkout process so that I can receive a discount on eligible products.</td></tr>
<tr><td colspan="3">1. Given I am in the purchase window, when I write a code in the coupon field and click “Apply”, then the system checks if the coupon is valid (the coupon code exists, and it is not expired) and applicable to a product in the cart; and add it to the list of the coupons and informs that the coupon was applied.
2. Given am in the purchase window, when I add an invalid coupon, then the system shows an error message indicating that the coupon cannot be added to the order.</td></tr>
</table>

<table>
<tr><td>PP-04</td><td>Purchase Products</td><td>Priority: High</td></tr>
<tr><td colspan="3">As a customer, I want to complete the purchase process, so that I can confirm my order and pay for the selected products.</td></tr>
<tr><td colspan="3">1. Given I am in the product purchase window, when I proceed to check out, then the system displays a summary of the order, including the total price.
2. Given I have multiple saved addresses, when I select one, then the system updates the order summary with the specified address.
3. Given I have reviewed all the purchase details, when I click the “Confirm Order” button, then the system processes the payment and redirects me to a confirmation page.</td></tr>
</table>

<table>
<tr><td>PP-05</td><td>Purchase Confirmation</td><td>Priority: Medium</td></tr>
<tr><td colspan="3">As a customer, I want to see a confirmation page after completing my purchase, so that I can verify my order details.</td></tr>
</table>

1. Given I have completed the purchase, when I am redirected to the confirmation page, then the system displays the order summary, including the purchased product, prices, and total amount. 2. Given I am on the confirmation page, when I view the details, then the system shows an order reference number. 3. Given I have completed my purchase, when I view the confirmation page, then the system allows me to download the receipt.

PP-06	**Receive Order Confirmation Email**	Priority: Low
As a customer, I want to receive confirmation via email after completing my purchase, so that I can have a recording of my order.		
1. When I have completed my purchase, then the system sends an order confirmation email to my email registered address. 2. Given I receive the confirmation email, when I open it, then the email includes the order number, a summary of the purchased items, and a message indicating that the payment is being processed.		

PP-07	**Re-enter Payment Details for Failed Order**	Priority: Medium
As a customer, I want to re-enter my payment details for an order where the payment failed, so that I re-try the purchase.		
1. Given my payment for an order has failed, when I check the purchase details, the system provides an option to re-enter my payment details. 2. Given I have entered my new payment details, when I submit them, then the system redirects me to the confirmation page.		

3.5.7 Customer Order Management

OM-01	**List Previews Orders**	Priority: Medium
As a customer, I want to see a list of my previous orders, so that I can review them.		
1. When I navigate to the “Order History” page, then I see a list of the previous orders that I made. 2. Given I have no previous orders, when I navigate to the “Order History” page, then the system shows a message indicating that. 3. Given I am on the “Order History” page, when I click on the order, then the system redirects me to the details of that order.		

OM-02	**Search Past Orders**	Priority: Low
As a customer, I want to search for past orders, so that I can easily find them.		
1. Given I am on the "Order History" page, when I write something into the search bar, then the system filters the orders to show the ones that match. 2. Given I am on the "Order History" page, when I write something into the search bar and there is no coincidence in the past orders, then the system shows a message indicating that there are no orders according to the filter.		

OM-03	**Download Past Order Invoice**	Priority: Medium
As a customer, I want to download the invoice for a past order, so that I can have a copy of the purchase.		
1. Given I am on the "Order History" page when I view the list of my past orders, then each includes a "Download Invoice" button. 2. When I click the "Download Invoice" button, then the system starts downloading a PDF that contains the order details.		

OM-04	**Receive Notification for Order Updates**	Priority: Medium
As a customer, I want to receive notifications of my order updates, so that I can stay informed about any changes.		
1. Given I have one or more orders, when one of them changes, then the system sends me a notification indicating this change. 2. Given I'm not logged in and I have one or more orders, when I log in, then the system shows me the pending notifications that arrived when I wasn't in the application.		

OM-05	**Receive Emails for Order Updates**	Priority: Low
As a customer, I want to receive an email of my order updates, so that I can stay informed about any changes even if I'm not using the application.		
1. Given I have one or more orders, when one of them changes, then the system sends me an email indicating the order number and the change.		

OM-06	Automatically Update Order Status as Received	Priority: Low
As a customer, I want the system to automatically update my order status as "Received" after a certain period, so I can have the order information updated.		
1. Given I have received my order, when one month has passed since the order was marked as shipped, then the system should automatically update the order status to "Received".		

OM-07	Mark Order as Received	Priority: Low
As a customer, I want to mark my order as "Received" after receiving it, so that I can keep my order information updated.		
1. When I navigate to the "Order History" page, then the system shows a button to indicate the order as received for all orders that have been shipped. 2. Given I have received an order, when I go to the "Order History" page and click the button to mark it as received, then the system updates the status of the order.		

3.5.8 Seller Order Management

SOM-01	View All Orders	Priority: Medium
As a seller, I want to view the orders that customers have made to my store so that I can manage them.		
1. When I navigate to the "Orders" section, then the system shows the list of orders sorted by purchase date.		

SOM-02	Filter Orders	Priority: Medium
As a seller, I want to filter orders, so that I can focus on some of them.		
1. Given I am in the "Orders" section, when I apply a filter, such as date range, amount, order ID, or name of one product that is included, then the system shows just the orders that match the specified criteria. 2. Given I am on the "Order History" page, when I apply a filter combination and there is no coincidence, then the system shows a message indicating that there are no orders according to the filter.		

SOM-03	**Search Orders**	Priority: Medium
As a seller, I want to search for orders, so that I can easily locate them.		
1. Given I am on the "Order History" page, when I write something into the search bar, then the system filters the orders to show the ones that match. 2. Given I am on the "Order History" page, when I write something into the search bar and there is no coincidence in the past orders, then the system shows a message indicating that there are no orders according to the search.		

SOM-04	**Register Order Shipping**	Priority: Medium
As a seller, I want to mark orders as shipped, so that I can continue the order processing.		
1. Given I am in the "Orders" section, when I have orders that haven't been shipped, then the system shows a button to mark the order as shipped. 2. When I click the ship order icon, then the system shows a form to fill in the shipping information, which contains the shipping company and track number. 3. Given I am on the Shipping Details form, when I fill in the information and click "Continue", then the system updates the order status and includes the shipping information.		

SOM-05	**Download Order List**	Priority: Medium
As a seller, I want to download the list of orders, so that I can have the information available offline.		
1. Given I am in the "Orders" section, when I have at least one order, then the system shows a button to download the orders. 2. When I click the "Download Orders" button, then the system starts downloading a PDF that contains the orders. 3. Given I am downloading an order list, when I try to navigate to another section, then the system allows the action without blocking me.		

SOM-06	**View Sales Revenue**	Priority: Low
As a seller, I want to view the revenue of my sales over time, so I can make informed decisions to improve my business.		
1. Given I am on the seller section, when I navigate to the "Sales Analytics" section, then the system shows a line chart with the x-axis representing time and the y-axis representing the total sales revenue.		

2. Given I am in the “Sales Analytics” section, when I select a specific date range, then the system updates the sales chart to show the revenue related to that range. 3. When I am in the “Sales Analytics” section, then the system shows a button to download the graph. 4. Given I am in the “Sales Analytics” section, when I click the button to download the graph, then the system starts the downloading of a file with the requested data.

SOM-07	**View Top-Selling Products**	Priority: Low
As a seller, I want to see my top-selling products, so that I can identify which products perform better than others.		
1. When I navigate to the “Sale Analytics” section, then the system displays a bar chart with the x-axis representing the product name, and the y-axis representing the number of units sold. 2. Given there are multiple products sold, when the bar is displayed, then the system orders the products from highest to lowest sales. 3. Given I am in the “Sale Analytics” section, when I select a specific date range, then the system updates the top-selling products to show the top-selling products related to that range. 4. Given I am in the “Sales Analytics” section, when I click the button to download the graph, then the system shows the format options to start the download, which are CSV, PDF, and XLSX. 5. Given I am in the “Sales Analytics” Section, when I click the format to download the graph data, then the system starts the downloading of a file with the requested data and the selected format.		

SOM-08	**View Revenue by Category**	Priority: Low
As a seller, I want to see the distribution of my sales by category so that I can adjust my business strategy based on data.		
1. When I navigate to the “Sales Analytics” section, then the system displays a pie chart showing the segments representing the product categories and the size of each one representing the total sales revenue generated by it. 2. When I hover over a category segment, then the system displays the revenue amount and percentage for that category. 3. Given I am in the “Sale Analytics” section, when I select a specific date range, then the system updates the revenue by category chart to show the top-selling products related to that range. 4. Given I am in the “Sales Analytics” section, when I click the button to download the graph, then the system starts the downloading of a file with the requested data.		

<table>
<tr><td>SOM-09</td><td>Print Invoice for an Order</td><td>Priority: Medium</td></tr>
<tr><td colspan="3">As a seller, I want to print an invoice for an order, so that I can keep it for my records.</td></tr>
<tr><td colspan="3">1. When I navigate to the “Orders” section, then the system shows a button to download the invoice for each order.
2. Given I am in the “Orders” section, when I click the button to download the invoice for an order, then the system starts the download the order invoice.</td></tr>
</table>

3.5.9 Product Feedback

<table>
<tr><td>PF-01</td><td>Leave a Product Rating and Review</td><td>Priority: Low</td></tr>
<tr><td colspan="3">As a customer, I want to leave a rating and a review for a product I have purchased, so I can share my experience.</td></tr>
<tr><td colspan="3">1. When I navigate to the product details of a product that I have purchased before, the system shows a section where I can add a rating and write a comment.
2. Given I have added a rating and written a comment, when I click the confirmation button, then the system adds the new review and comment to the product.</td></tr>
</table>

<table>
<tr><td>PF-02</td><td>See Product Reviews</td><td>Priority: Low</td></tr>
<tr><td colspan="3">As a customer, I want to see the product reviews, so that I can make an informed purchase.</td></tr>
<tr><td colspan="3">1. When I navigate to the product details page, if the product has any reviews, then the system shows the average rating, with the number of ratings, and each review.</td></tr>
</table>

3.5.10 Personalized Recommendations

<table>
<tr><td>PR-01</td><td>List Recommended Products</td><td>Priority: Low</td></tr>
<tr><td colspan="3">As a customer, I want to see the recommended products that I might like, so that I can discover new items to buy.</td></tr>
<tr><td colspan="3">1. Given I have previously browsed, purchased, or rated products, when I view the recommended products, then the system shows personalized recommendations for me.
2. Given I perform actions in the platform like purchasing products, when I request new recommendations, then the system refreshes my recommendations.</td></tr>
</table>

PR-02	**View Similar Products**	Priority: Low
As a customer, I want to see similar products when I view the details of an item so that I can complement my purchase.		
1. Given I am in the details of a specific product when I scroll down, then the system displays a section where I can see a list of similar products. 2. Given I am in the similar products section, when I click on one of them, then the system redirects me to the product details. 3. Given I am in the similar products section, when I click the icon to add to the cart, then the system updates my shopping cart accordingly.		

PR-03	**Ask for Recommendations**	Priority: Medium
As a customer, I want to receive personalized coffee recommendations through a conversational chatbot, so that I can find the right product in a simpler way.		
1. When I click the chatbot icon, then the system shows me the chat window with a welcome message to start asking questions. 2. Given I request a coffee or product recommendation, when I provide my preferences (flavor, roast type, brewing method) then the chatbot suggests products that match. 3. Given I provide incomplete information, when the chatbot needs more details, then it asks more questions to understand what I'm looking for.		

PR-04	**Add products to the shopping cart with the chatbot**	Priority: Low
As a customer, I want to add products to the cart with the chatbot so that I can easily add recommended products.		
1. Given the chatbot gives me a recommendation, when I click the button "Add to Cart", then the system adds the product to my existing shopping cart. 2. Given I am chatting with the chatbot, when I ask to view my cart, then the chatbot shows a summary of the items that are in my cart.		

3.5.11 Customer Support

CS-01	**Contact Support**	Priority: Medium
As a user, I want to contact support so I can get help with problems.		
1. Given I am in the application, whether logged in or not, when I click "Contact Support", then the system redirects me to the form where I can fill in the necessary information to create the issue.		

2. Given I am on the “Contact Support” page, when I fill in the information and click “Submit”, then the system shows a confirmation message indicating that the request was received and sends me an email message indicating the issue details, which contains a unique identifier.

CS-02	**Receive Notification When Support Issue Status Changes**	Priority: Low
As a user, I want to receive a notification when the status of my support issue changes, so I can keep informed about the progress of it.		
1. Given I have submitted a support issue, when the status of my issue changes, then the system sends me a notification via email and through the platform.		

CS-03	**List Issues**	Priority: Medium
As a user, I want to view the list of my past issues, so that I can keep track of them.		
1. When I navigate to the “Issues” page, then I the system shows me the list of issues that I’ve made, with a brief description and status.		

CS-04	**See Issue details**	Priority: Medium
As a user, I want to view the issue details, so that I can remember accurately the problem that I have.		
1. Given I am in the “Issues” section, when I click on an issue, then the system redirects me to the issue details. 2. When I am on the issue details page, then the system shows the whole issue information, including the changes in status and the dates that it has had.		

CS-05	**Close Issue**	Priority: Low
As a user, I want to close an open issue, so that I can mark it as resolved if I don’t need assistance anymore.		
1. Given I have submitted an issue and I am in the “Issues” section and I no longer need assistance, when I click the button “Mark as resolved”, then the system resolves and closes the issue.		

3.5.12 Administrator Tools

AT-01	Pause Product Publication	Priority: Medium
As an administrator, I want to pause product publication, so that I temporarily hide it from the customers if it requires review.		
1. When I'm in the product details, then the system shows a button to pause the product publication. 2. Given I am in the product details, when I click the button to pause the product publication, then the system reflects the change and does not show the product to customers.		

AT-02	Delete a Product	Priority: Medium
As an administrator, I want to delete a product from the marketplace, so that I can permanently remove it if it does not follow the application policies.		
1. When I'm in the product details, then the system shows a button to delete the product publication. 2. Given I am in the product details when I click the button to delete the product publication, then the system shows a confirmation dialog to confirm the action. 3. Given I am in the confirmation dialog of the deletion of a product, when I click confirm, then the system redirects me to a deletion form where I can write the reasons for the deletion. 4. Given I am in the deletion form, when I fill in all the necessary data and click "Remove", then the system removes the product from the platform and it does not appear from customers, from users, it appears as "Deleted".		

AT-03	List All Orders	Priority: Medium
As an administrator, I want to list all orders so that I can monitor transactions and provide support if needed.		
1. When I navigate to the "Orders" section, then the system redirects me to the "Order List" section, where I can see all the orders. 2. Given I am in the "Order List" section, when I click on an order, then the system shows me the order details.		

AT-04	View Customer Details	Priority: Medium
As an administrator, I want to see the customer details, so that I can assist customers.		
1. When I navigate to the "Customers" section, then the system shows me a list of the users registered into the platform.		

2. Given I am in the “Customers” section, when I click on a customer, then the system redirects me to the “Customer Details” section and shows me the customer information. 3. Given I am in the “Customer Details” section, when I click the button, “Show Orders”, then the system shows me the list of orders made by the user.

AT-05	Create Product Category	Priority: Medium
As an administrator, I want to create a product category, so that I can organize the marketplace.		
1. When I navigate to the “Product Categories” section, then the system shows the list of current categories. 2. Given I am in the “Product Categories” section, when I click the button “Add Category”, then the system shows me the form to create a new category, which is composed of two fields, category name and description. 3. Given I am in the form to create a product category, when I fill in all the fields and click create, then the system saves the new category.		

AT-05	Remove Product Category	Priority: Low
As an administrator, I want to delete a product category, so that I can remove unnecessary categories.		
1. Given I am in the “Product Categories” section, when I click the button “Delete Category” of a specific category, then the system shows me the form to delete a new category, which is composed of one field, which is a selectable field to choose the product category to which the products associated to the deleted category will migrate. 2. Given I am in the form to delete a product category, when I select the category where all products are going to migrate and click confirm, then the system deletes the category and migrates all the products to the selected category.		

AT-06	Update Product Category	Priority: Medium
As an administrator, I want to update a product category, so that I can keep the product information updated.		
1. Given I am in the “Product Categories” section, when I click the button “Update Category” of a specific category, then the system shows me the form to update the category, which is composed of the fields “name” and “description”. 2. Given I am in the form to update a category, when I perform the changes and click the button “Update”, then the system updates the category and reflects the changes in the e-commerce.		

3. Given the category has been changed, when I browse the products under this category, then the system shows me all the products that were originally assigned to this category before the change.

AT-07	**Create Tag**	Priority: Medium
As an administrator, I want to create a tag, so that I can categorize the products efficiently.		
4. When I navigate to the "Product Tags" section, then the system shows the currently available tags. 5. Given I am in the "Product Tags" section, when I click the button "Add Category", then the system redirects me to the form where I can create a Tag. 6. Given I am on the tag creation page, when I fill in all the required fields, which are name and possible values (e.g. Coffee type: robusta, arabica, etc.), then the system creates the tag and redirects me again to the tag list.		

AT-08	**Remove Tag**	Priority: Low
As an administrator, I want to delete a tag so that I can keep the product information updated.		
3. Given I am in the "Product Tags" section, when I click the button "Delete Tag" over one specific tag, then the system deletes the tag and all possible values that it has.		

AT-09	**Update Tag**	Priority: Low
As an administrator, I want to update a product tag, so that I can keep them valuable and relevant.		
1. Given I am in the "Product Tags" section, when I click a tag name, then the system shows me all the possible options that it has, each option has a delete icon, furthermore, at the end of the option there is a button to add a new one. 2. Given I am in the tag section, when I click the icon to delete a tag option, then the system removes it and shows a message indicating the deletion. 3. Given I am modifying an existing tag, when I click the button to add a new option, then the system shows a new field to write the option name. 4. Given I am modifying an existing tag, when I fill in the new option name and click "Save", then the system saves it and shows a message indicating the addition.		

AT-10	**List Issues**	Priority: Low
As an administrator, I want to list the user issues, so that I can monitor and manage them.		
1. When I navigate to the issues management section, then the system shows all reported issues with the main information, such as status, creation date, and the customer who raised it.		

AT-11	**Change Issue Status**	Priority: Low
As an administrator, I want to update the status of a user's issue, so that I can track and manage their resolution.		
2. Given I am in the issue details section, when I update its status, then the system saves the change and informs the user through a notification that the status was changed. 3. Given that an issue's status has changed, when the customer consults the details in the platform, then the system shows the updated status.		

3.6 Non-Functional Requirements

This section states the principal non-functional requirements that the application must have to ensure a reliable, efficient, and secure user experience. It also outlines some quality attribute scenarios of the application, where key quality attributes, such as performance, scalability, and modifiability, are addressed. While not all quality scenarios are covered, the ones described represent well the overall scenarios that the system will have.

One distinction present in non-functional requirements is between critical and non-critical operations. Critical operations are those that are part of the application's core process, which is selling products, for example, listing products, viewing product details, adding products to the cart, and purchasing them. On the other hand, non-critical operations are those that enhance the application core, but are not required for its operation, for example, recommending products, sending notifications, and recording user interactions.

The platform's normal operational baseline is defined as:

- 1,000 daily purchases
- 50,000 daily product details views, assuming a 2% conversion rate, which is a standard e-commerce value (Shopify, 2024)

1. NFR-1 Success Rate for Purchase Process

The purchase process is one of the most critical steps in the application. Additionally, it has an external payment gateway dependency, which introduces latency. For these reasons, a dedicated non-functional requirement has been created.

The system should achieve a 99% success rate in transactions related to the purchase process.

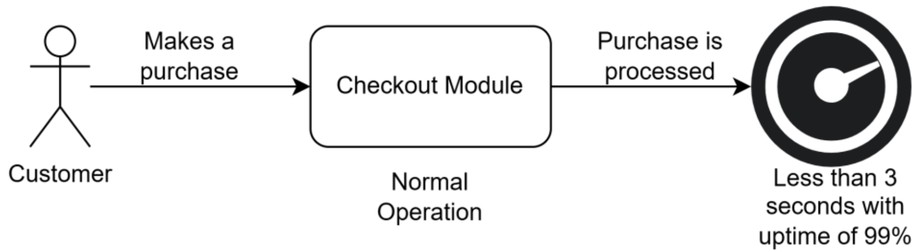

Figure 3.5 Make a purchase quality scenario

2. NFR-2 Success Rate for Critical Application Features

The system should achieve a 99% success rate in critical operations that do not involve third-party services.

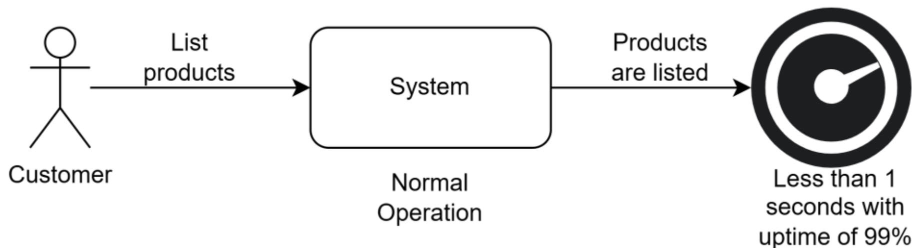

Figure 3.6 List product quality scenario

3. NFR-3 Success Rate for Non-Critical Application Features

The system should achieve a 95% success rate in non-critical operations that do not involve third-party services.

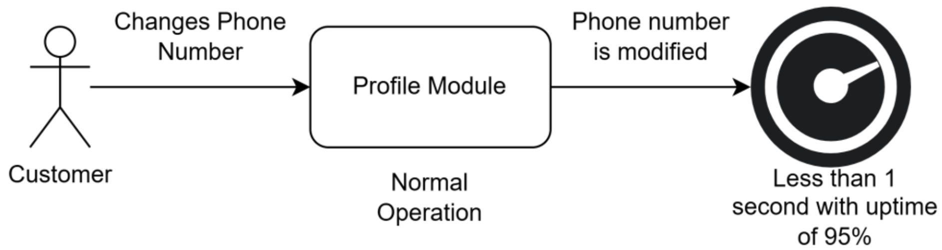

Figure 3.7 Change phone number quality scenario

4. NFR-4 Error Handling

The system should handle errors not caused by the users, ensuring the application does not crash, maintains a valid state, and provides meaningful messages to guide the user.

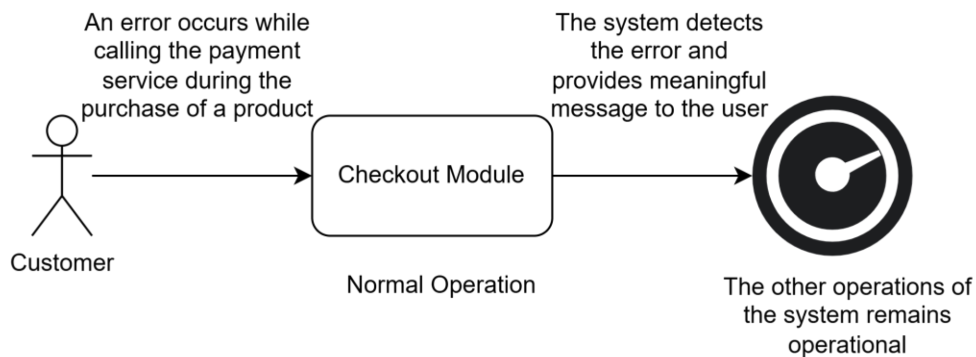

Figure 3.8 Prevent crashing quality scenario

5. NFR-5 Performance for Concurrent Users

The platform should support up to 1.000 concurrent users in a ten-minute window during peak times.

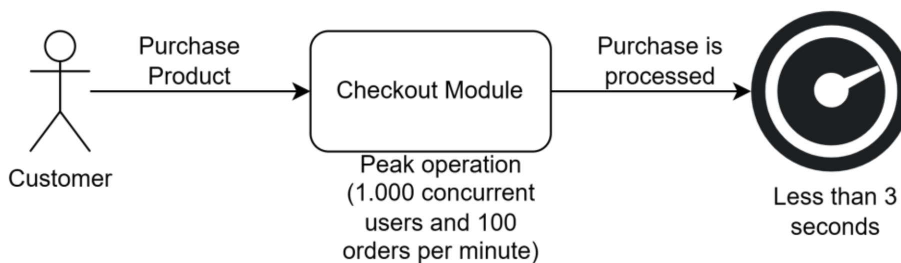

Figure 3.9 Purchase product during peak traffic quality scenario

6. NFR-6 Performance of Product Catalog

The product catalog is one of the key features of the platform, since it manages the most important entity, which is the product. This non-functional requirement ensures that, regardless of a high number of products, the system can serve the requests while maintaining an acceptable response time.

The product catalog should scale to support up to 100.000 products without degradation in search performance.

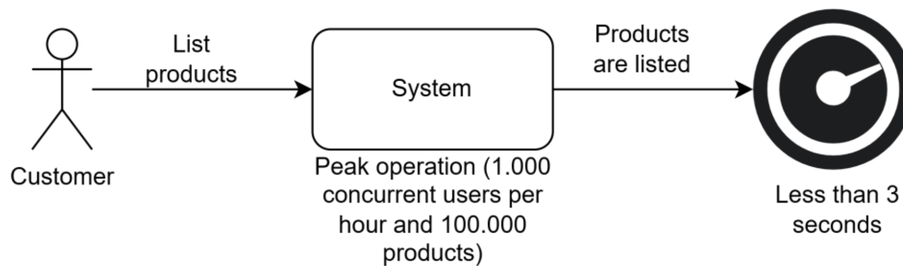

Figure 3.10 List products during peak traffic quality scenario

7. NFR-7 Modularity

The system should be modular, with loose components to allow easy modifications and additions.

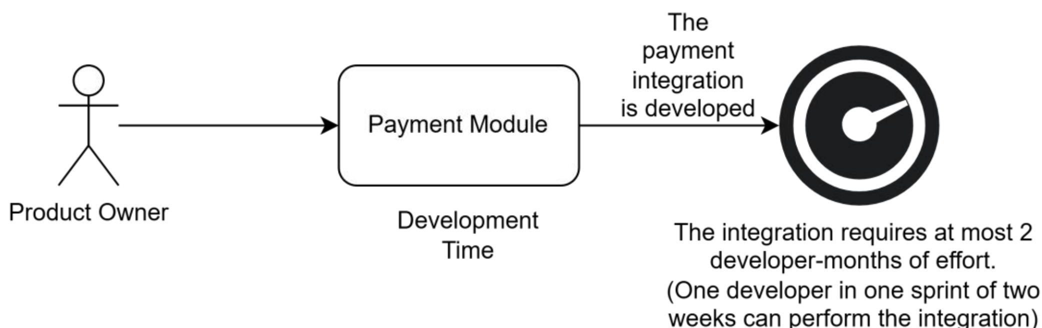

Figure 3.11 New payment integration quality scenario

8. NFR-8 Logging

The system should have a centralized logging mechanism to allow quick identification of issues.

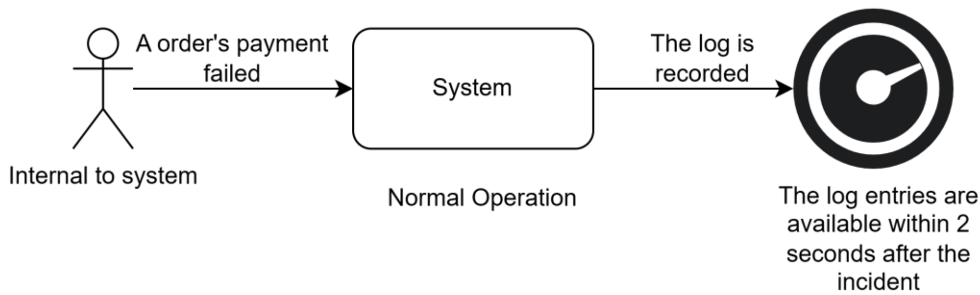

Figure 3.12 Log failed payment quality scenario

9. NFR-9 Security

The system should prevent requests from accessing operations that require authentication/authorization.

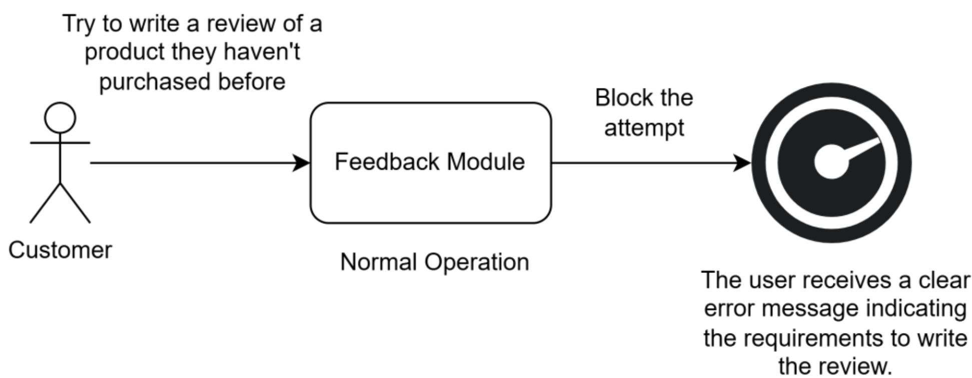

Figure 3.13 Authorization quality scenario

10. NFR-10 Availability

The system should maintain an uptime of 99.9%

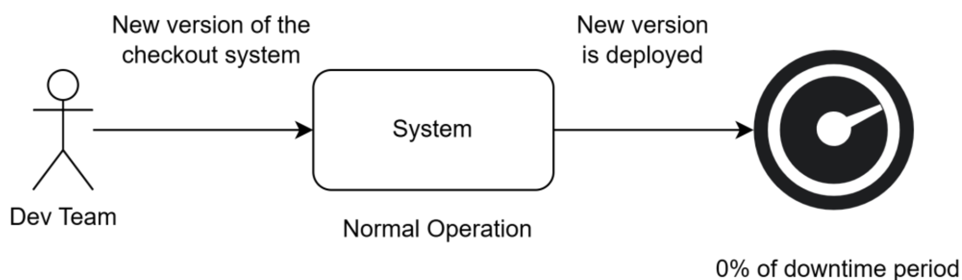

Figure 3.14 Availability quality scenario

The non-functional requirements were tested against the application. Although the tests are performed on specific study cases, the results provide insights about the system's performance and compliance. The results are presented in section 7.

4 Design

This section describes the design process of defining the microservices using the domain-driven design methodology, where each service reflects a bounded context, where, according to Sam Newman (2021), “if our service boundaries align to the bounded contexts in our domain, and our microservices represent those bounded contexts, we are off to an excellent start in ensuring that our microservices are loosely coupled and strongly cohesive.” (p. 33).

4.1 Design Roadmap

Due to the project's time limitations, the complete design of the system is not feasible. For this reason, as the software development lifecycle progresses, the scope is narrowed without affecting the project’s main objectives.

The high-level design was performed for the whole system, which includes the domain and context model, the definition of the bounded contexts, and the subsequent microservices decomposition.

The microservices interaction scope was chosen to develop the Minimum Viable Product (MVP), which is defined as the process of performing a purchase. This process includes the creation of a product, which implies the creation of multiple other services, the shopping cart management, and the process of ordering the purchase.

The microservices infrastructure decisions, stated in Section 4.7, were made for a subset of the MVP, which was chosen so that its complexity permits to show the implementation of the main cloud patterns, which is one of the project objectives.

This scope enables the system to deliver value fast and provide a baseline for future iterations. Furthermore, it provides the necessary complexity to implement the intended architectural patterns, such as CQRS, transactional outbox, event-driven communication, and SAGA orchestration.

4.2 Domain Model

The domain model is represented in *Figure 4.1*. It provides an overview of the key entities, their attributes, and the relationships between them.

The system has three main roles: administrator, customer, and seller. The last two roles are merged into one user so that they can act as both.

The administrator can see all orders, products, and marketplace user information to resolve issues and help other users. They can also suspend or delete products if they don’t follow the platform rules. Finally, they can create, delete, and update product categories and tags, which are a critical part of the user experience, since they will be the ones that the customers can select when filtering the products and the ones that sellers will use to classify them.

The seller is responsible for creating the products, they must add all the necessary information to put the product in the marketplace, such as name, description, images, etc. Also, they can change a product visibility if they want to temporarily suspend the sale, or they can delete it if they don't want to sell it anymore.

Sellers can promote their products by creating product-specific discounts. They can define the duration and specify whether a code is required for activation or if it will be applied by default.

Furthermore, sellers are responsible for marking the product as shipped, since the application does not manage the shipment process due to the variety of geographical locations of the sellers. Related to the sell analytics, the sellers can generate reports of the orders and download them based on filter criteria, and they can view sell analytics, like the number of sales based on a period, number of products sold, etc.

The customer is the one who will make purchases on the application. To achieve that, they can list all products, filter and sort them, see the product details, and add them to a shopping cart. After they have selected all the desired products, they can start the checkout process, where they provide the information (if it's not saved in the application yet) to generate the order. If they have a problem with the order or with any other thing related to the application, they can submit an issue to get help, they can see the status of each issue. Finally, customers can write reviews and rate the products they have purchased. These ratings and reviews help other customers make informed decisions to make future purchases.

Another cross-functionality is the notification system, where the customers and sellers will receive information regarding the orders, for example, if the seller marks the order as shipped, the user will receive a notification with this information; or if a customer buys a product, the seller will also receive the notification of the purchase. This functionality provides valuable information for both roles, so they can stay informed about the purchase process and easily track the information.

The platform will run periodic marketing campaigns to keep users informed about new product arrivals and personalized recommendations based on their interests.

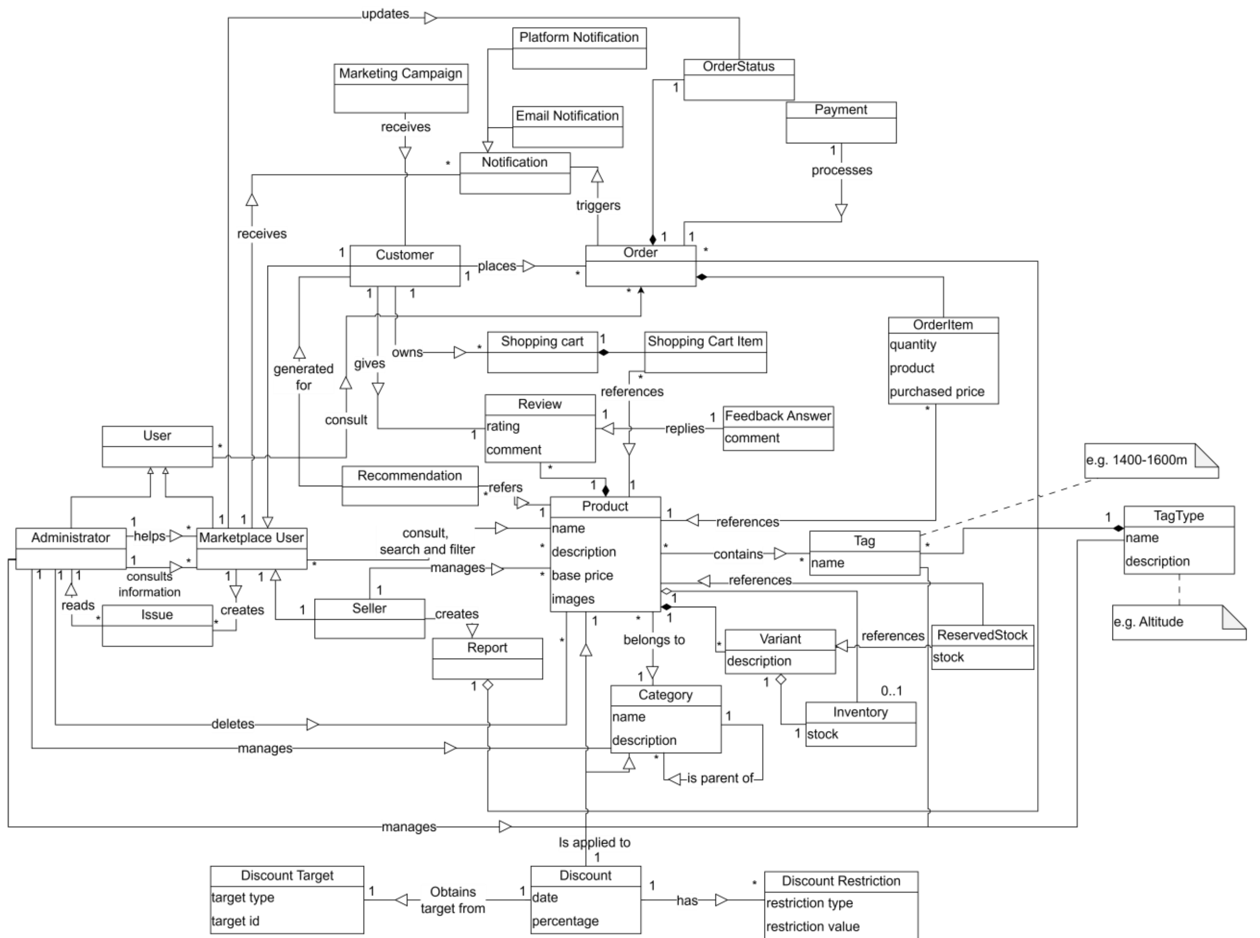

Figure 4.1 Domain Model

4.3 Context Model

Figure 4.2 Illustrate the external actors that interact with the system and their main functionalities. It outlines the responsibility of each actor in the system workflow.

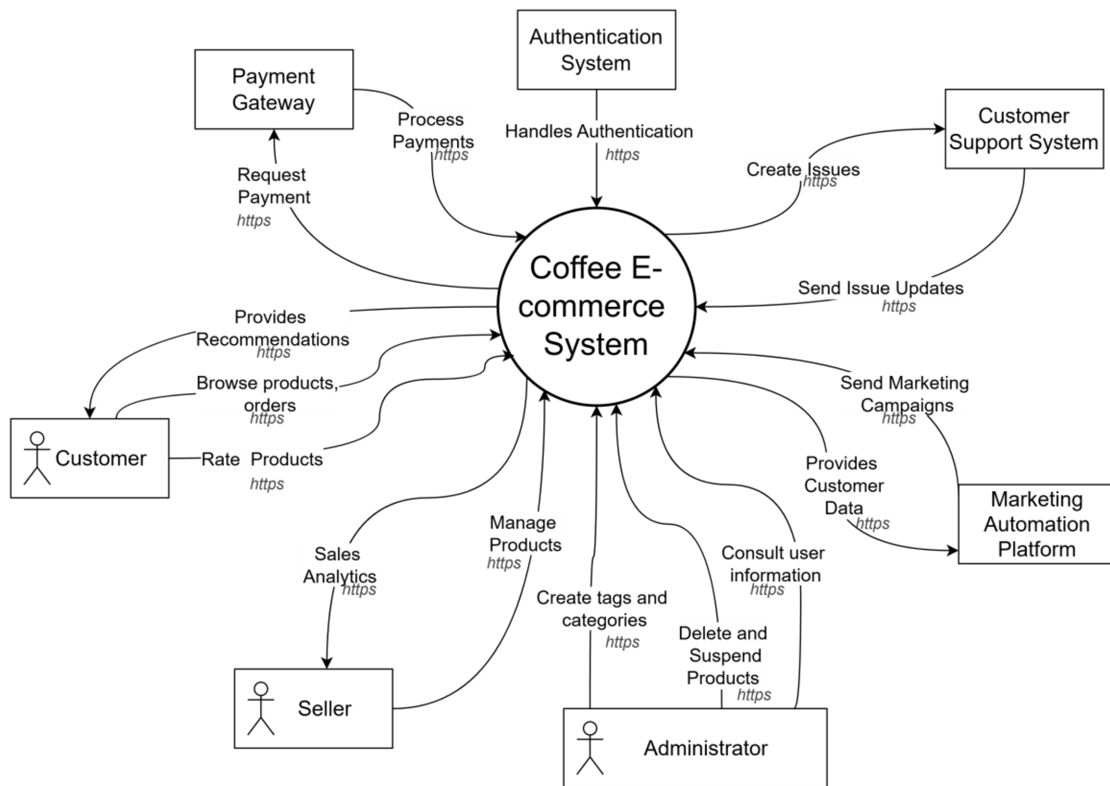

Figure 4.2 Context model

4.4 Bounded Contexts and Domains

This section describes the different bounded contexts of the application, which are obtained using the domain-driven design framework (Evans, 2003). Within each context, key responsibilities, entities, value objects, and ubiquitous language were identified.

4.4.1 Identity

Authentication is necessary to build the application, however, it is not the central part of the system. For this reason, it is part of the support subdomain rather than the core domain. Additionally, to ensure a robust authentication system, the platform will utilize a well-proven third-party service.

The identity subdomain is responsible for authenticating users and maintaining access.

Ubiquitous Language

User: A person who can authenticate and access the system; in this case, it could be an administrator, a customer, or a seller.

Identity: It is the unique representation of the user in the system

Authentication: The Process to verify the user identity

Authorization: The Process to verify if the authenticated user has access to a protected resource

Role: It determines the set of permission that the user has

Token: A credential that provides authentication

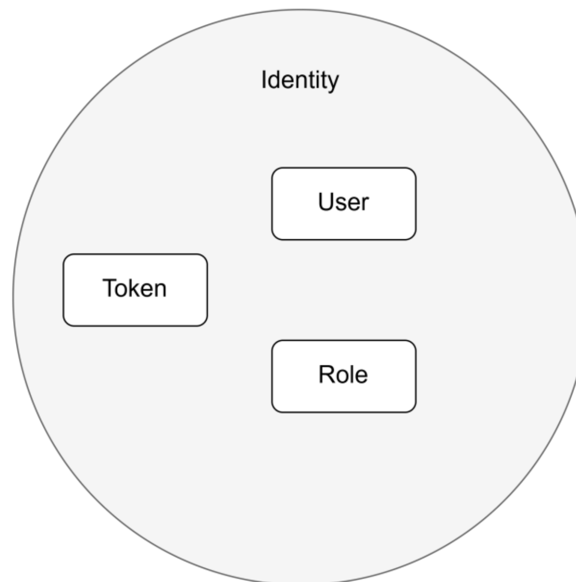

Figure 4.3 Identity Entities and Value Objects

4.4.2 Profile Management

The profile management context is responsible for managing user information, except for the one related to authentication. This information includes personal details, such as contact phone numbers, addresses, and profile images.

Ubiquitous Language

Profile: It contains the user's information.

Customer profile: It contains information related exclusively to customers, such as the delivery address.

Seller profile: It contains information related exclusively to sellers, such as business direction or description.

Admin profile: It contains information related exclusively to the administrator, such as admin status

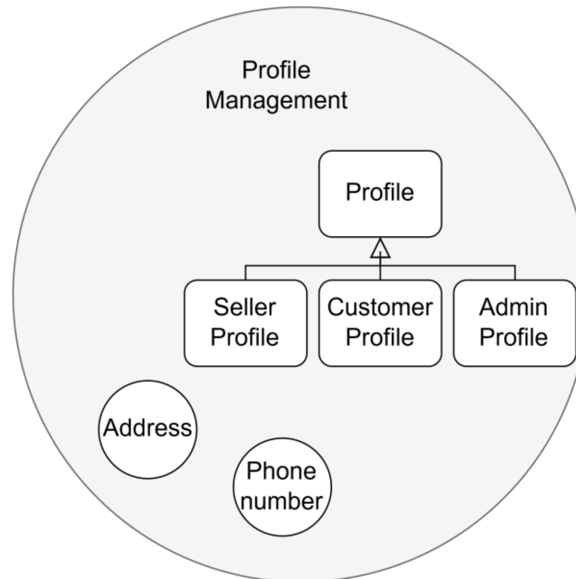

Figure 4.4 Profile Management Entities and Value Objects

4.4.3 Product Management

The product management context is responsible for handling the information related to products and their categorization.

Ubiquitous Language

Product: An item available for sale on the platform

Variant: A specific version of the product

Category: The principal classification for products

Tag Type: A classification to group products (E.g., Altitude)

Tag: The classification specific value where the product belongs (E.g., 1600 meters)

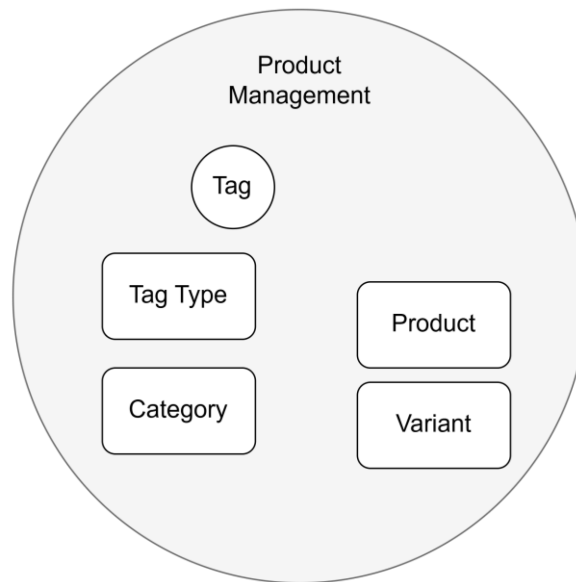

Figure 4.5 Product Management Entities and Value Objects

4.4.4 Inventory Management

Having a separate context to manage the inventory lifecycle is recommended, as the scaling requirements for product management differ from those for inventory management. For example, the times that a stock is updated are much higher than the times that a product is updated by the seller. Also, this context can grow in the future to perform low-cost alerts or other features that can arise while the platform is growing.

Ubiquitous Language

Inventory: Current quantity of products available for sale and shown to other customers

Inventory Reservation: Temporary stock allocation when the customer is in the checkout process

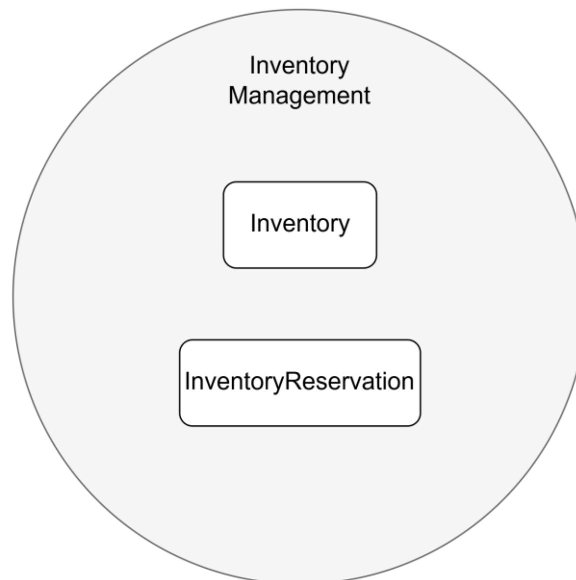

Figure 4.6 Inventory Management Entities and Value Objects

4.4.5 Pricing and Promotions

Since the intention is to provide a flexible and scalable pricing mechanism, it is recommended to have a separate context to manage these concepts. This design will allow further improvements and extensions to be implemented without affecting other elements, respecting the single responsibility principle.

The pricing and promotions context is responsible for managing pricing strategies, such as discounts with and without coupons.

Ubiquitous Language

Base price: The price of a product before the discounts

Discount Restriction: A rule that determines if a discount can be applied

Discount Target: Defines to which entity the discount is applied (E.g., Product or Category)

Discount: A price reduction

Coupon: A code that allows customers to apply for a discount

Price adjustment: A modification to the base price after discounts

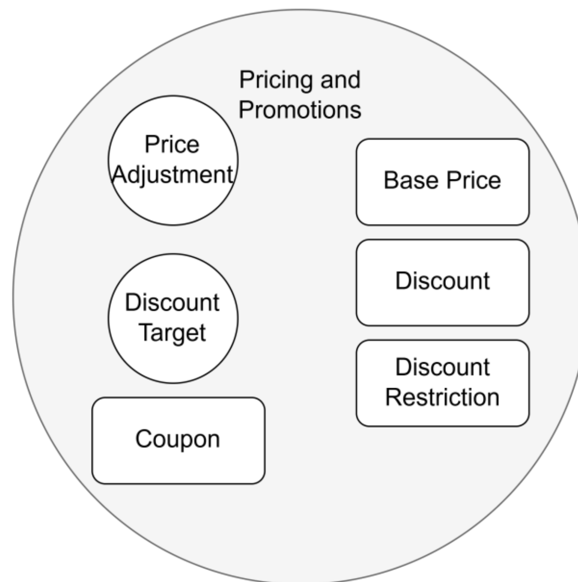

Figure 4.7 Pricing and Promotions Entities and Value Objects

4.4.6 Customer Feedback

The customer feedback context is responsible for managing the information related to reviews and ratings.

Ubiquitous Language

Review: A customer's opinion in text for a product.

Rating: A numeric scale to assign a score to a product or seller

Feedback Answer: A reply to a customer review made by the seller

Figure 4.8 Feedback Entities and Value Objects

4.4.7 Shopping Cart Management

It is responsible for managing the customer shopping experience, where they can add and remove products, modify the quantities, and retrieve an existing cart.

Ubiquitous Language

Cart: A Collection of items (i.e., products) that the user will buy

Cart item: A product in the cart with the quantity

Cart snapshot: A version of the cart at a specific moment; it will be helpful to pass to the checkout context

Cart restoration: When a logged-in user continues the shopping experience with a previously created cart

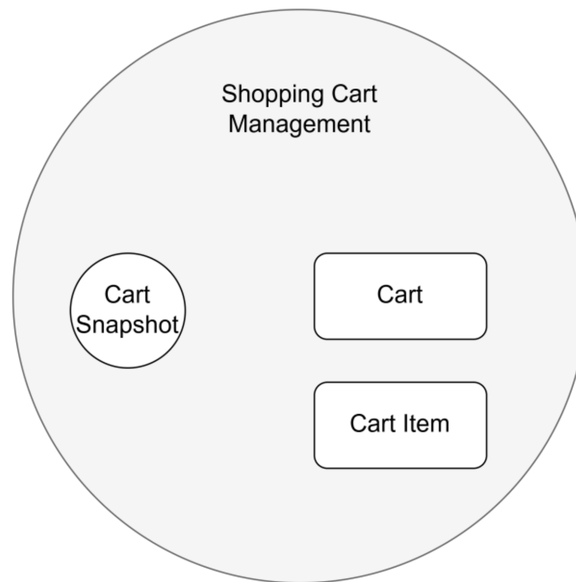

Figure 4.9 Shopping Cart Management Entities and Value Objects

4.4.8 Checkout & Order Management

The checkout & order management context is responsible for:

1. Converting the cart into an order, which includes validating the cart, calculating the final prices, interacting with the payment domain to perform the purchase, and confirming the order creation.
2. Manage the order lifecycle until it reaches a final state, such as delivered.

Ubiquitous Language

Checkout session: Process where a customer finalizes the purchase

Cart snapshot: Contains the products and quantities that will be purchased

Order confirmation: The Process of placing the order after the payment was validated

Order: A confirmed request for products that a user has purchased

Order Line: An individual product entry

Order status: Current state of the order

Bill: A document that summarizes the order details

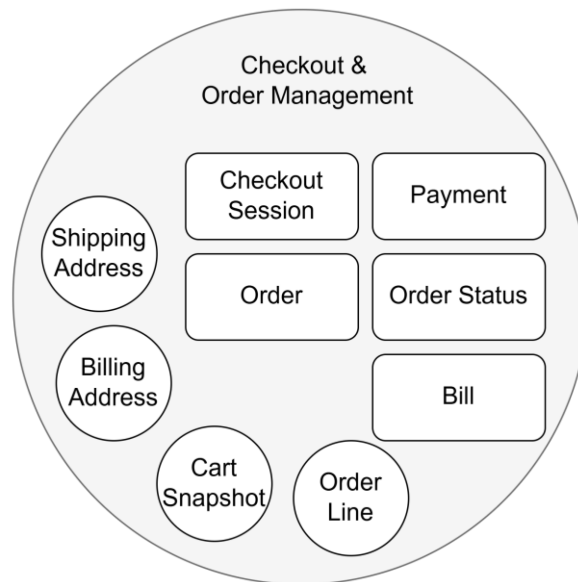

Figure 4.10 Checkout & Order Management Entities and Value Objects

4.4.9 Payment Context

It is part of a generic subdomain, where the functionality is externalized to a third-party service. The primary responsibility is to handle the customer's payment for an order.

Ubiquitous Language

Payment: A transaction performed to complete an order

Payment gateway: The third-party service that processes the payment

Payment status: The state of the payment

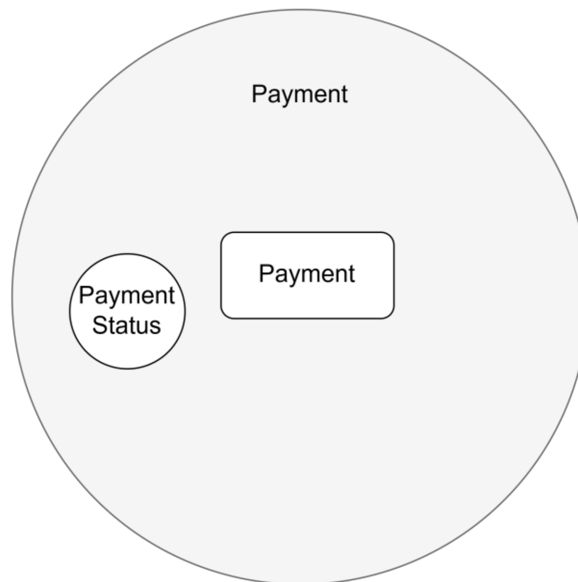

Figure 4.11 Payment Entities and Value Objects

4.4.10 Notifications Context

This context is part of the supporting subdomain, where the primary responsibility is to deliver notifications to users through email or in-app.

Ubiquitous Language

Notification: Message sent to a user

Delivery Channel: Platform through the notification is sent

Notification preference: User settings that define which notifications they want to receive

Notification Type: Defines the purpose of the notification, such as order updates, product releases, login attempts, etc.

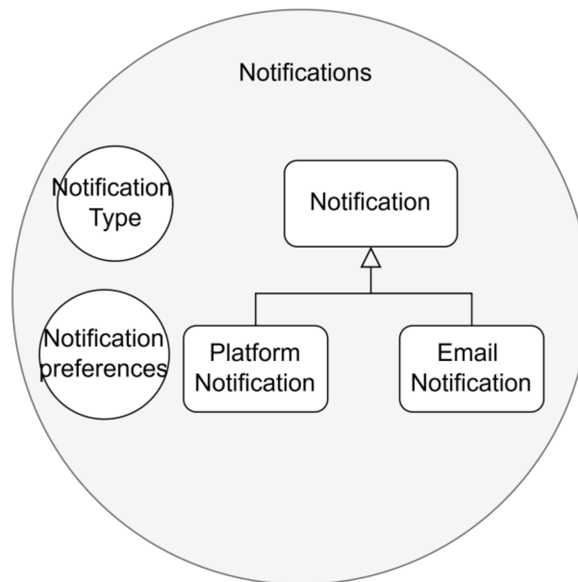

Figure 4.12 Notifications Entities and Value Objects

4.4.11 User Interaction

This context tracks the users' actions as they interact with the platform. It stores all information related to the customer's journey, including product views, cart activity, and purchases. This is important to maintain a track of user actions, which is helpful for internal system usage, such as analytics and usage insights.

Ubiquitous Language

Event: A user action, such as viewing a product or making a purchase

User Session: Determines the session where the event was performed. It is useful to aggregate events and analyze the behavior in a specific period.

Product Interaction: It is product-related information for an event.

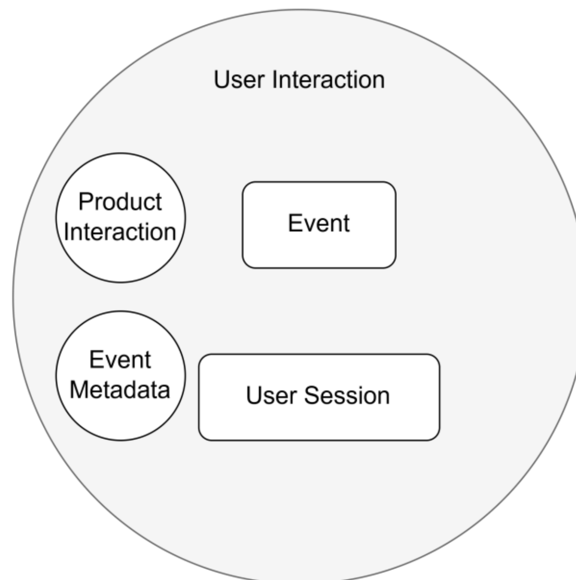

Figure 4.13 User Interaction Entities and Value Objects

4.4.12 Recommendations

This context provides useful product recommendations to users according to their purchase history and interactions with the platform. Also, the recommendations can be based on the current product that the user is watching.

The recommendations will be used to conduct marketing campaigns for customers, where they will receive personalized products based on their interests.

Ubiquitous Language

Campaign: A scheduled message sent to users

Campaign Type: It is used to differentiate each type of campaign (discounts, product recommendations, etc.)

Recommendation Request: It is the trigger to generate a recommendation.

Recommendation: A Product that a user may like based on their platform interactions or the current product that they are watching.

User profile: It contains the user's interactions and purchase history.

Recommendation Score: A ranking that defines the relevance of the recommended product.

Recommendation model: The algorithm used to suggest the products.

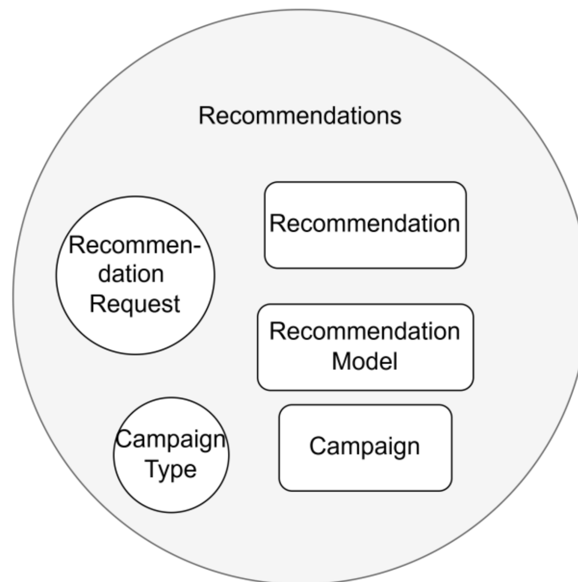

Figure 4.14 Recommendations Entities and Value Objects

4.4.13 Conversational AI

This context is responsible for managing the interactions with the user performed using natural language to provide product recommendations. It can be easily scaled to perform other actions, like answering questions.

Ubiquitous Language

User Query: The text input made by the customer

Intent: Goal the customer wants to achieve

Recommendation: List of products suggested by the system after analyzing the user query

Chat Session: A conversation between the user and the chatbot

Response Template: Standard structure to answer the user query

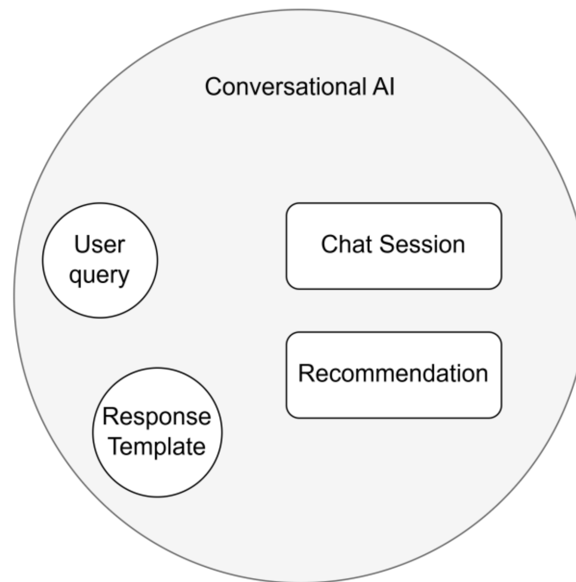

Figure 4.15 Conversational AI Entities and Value Objects

4.4.14 Analytics and Reporting

The responsibility of this context is to work with aggregated data to provide useful information to sellers. This context can be easily scalable to provide insights at the platform level, which can be useful to the platform owner.

Ubiquitous Language

Sales Report: A summary of revenue for a period.

Top-selling products: A list of products with the most sales.

Product performance: The Number of sales that a product obtains for a period.

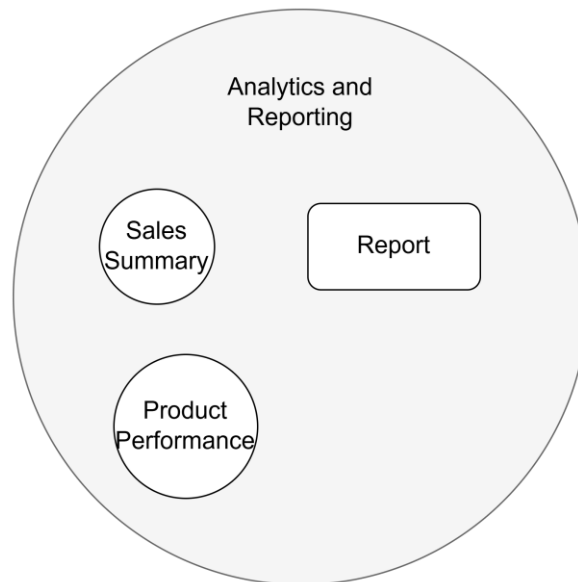

Figure 4.16 Analytics and Reporting Entities and Value Objects

4.4.15 Visual Search

It manages the processes related to image search. The main responsibilities are processing the uploaded images, obtaining a suitable representation, performing similarity searches, etc.

Ubiquitous Language

Image Query: The image uploaded by a user for searching products

Feature Vector: The representation of the image in numerical format

Similarity Score: A numerical score indicating how similar two images are

Search Result: List of products obtained after the search

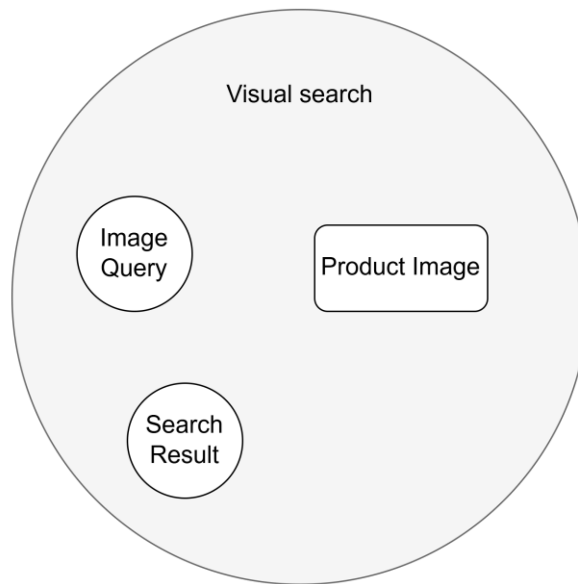

Figure 4.17 Visual Search Entities and Value Objects

4.4.16 Customer Support

It is part of the support subdomain, where the main responsibility is issues management through the whole life cycle. As this functionality will be externalized to a third-party service, the main concern for the platform is the issue and its status, the other parts should be managed as a black box.

Ubiquitous Language

Issue: Is the customer inquiry

Issue status: It represents the current state of the issue

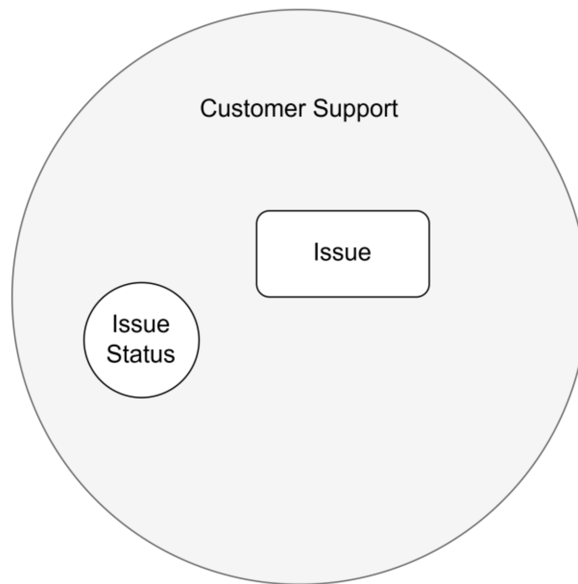

Figure 4.18 Customer Support Entities and Value Objects

The Figure 4.19 illustrates the previously explained contexts grouped by domain.

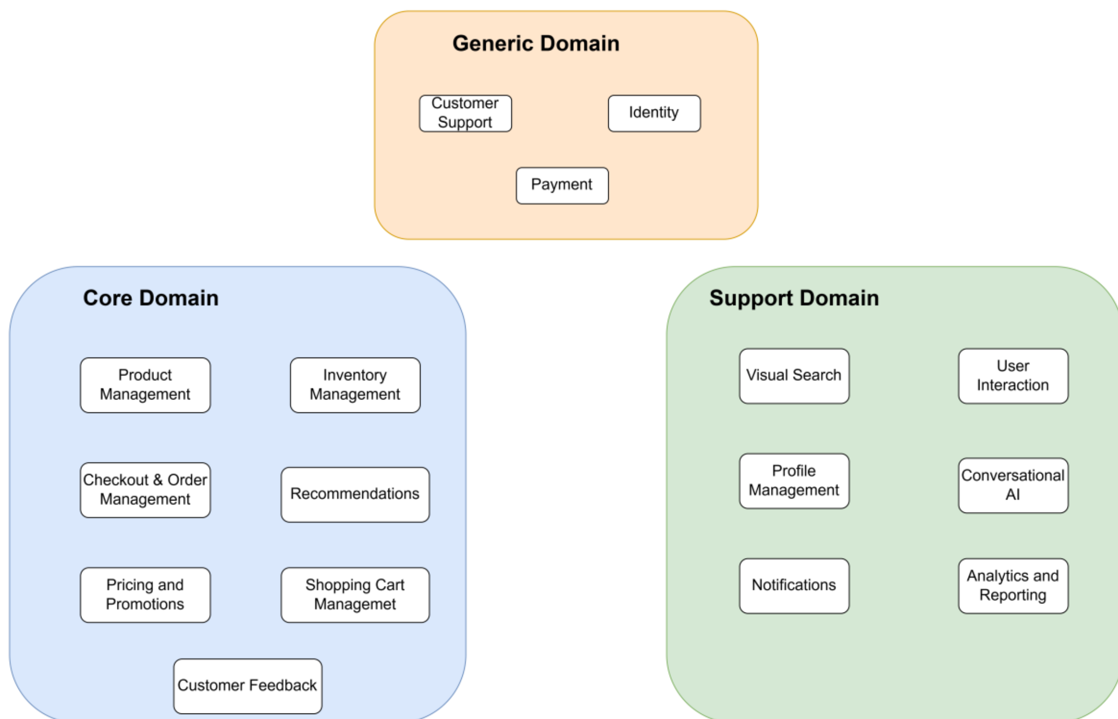

Figure 4.19 Contexts Grouped by Domain Type

4.5 Microservices Decomposition

Based on the bounded contexts specified above, the decomposition into microservices can be done smoothly. Each microservice is highly cohesive, loosely coupled, and follows the single responsibility principle.

For some contexts, additional microservices will be added to accomplish the non-functional requirements of performance and scalability. For example, the **Command Query Responsibility Segregation Pattern (CQRS)**, explained in detail in section 2.5.3, can be used for the product management context, which can be divided into two services: one for the update, delete, and create operations, which is called the **command** service; and the other for retrieving the information, which is called the **query** service. This distinction is made because the number of requests to retrieve products is significantly higher than the number of requests to create a product, resulting in unbalanced scaling requirements. Additionally, the data used for the query can be explicitly organized to improve the efficiency of retrieval, while the other one used for the command can be implemented to ensure business constraints and integrity.

Another reason to create additional microservices is the **anti-corruption layer**, which safeguards the internal system's ability to maintain the domain model secure and unchanged. Another service is in charge of adapting the content and translating every communication with the external system.

The Figure 4.20 illustrates the microservice decomposition and the services used in the application. Each microservice is a Kubernetes deployment, and each service has a Horizontal Pod Autoscaler to manage dynamic loading. Inside the Kubernetes service, Istio is used to manage intra-service communication. This will enable observability and easy incorporation of cross-cutting concerns like mutual TLS and metrics. It uses the sidecar pattern, which respects the single responsibility principle of each microservice.

An application gateway is used as a single entry point for the application to increase security and customization. It manages API authentication by integrating with the authentication support domain. This service should offer capabilities such as header modification and injection. This can be useful for structured logging, as the application gateway can generate the correlation ID. Using this component also provides better governance by providing features like API documentation.

Some microservices present in the diagram are not part of the context models. These are not part of any business domain, but they are technical tools for developing the system. For example, Catalog Aggregator aggregates information from multiple services, following the **backend-for-frontend** pattern.

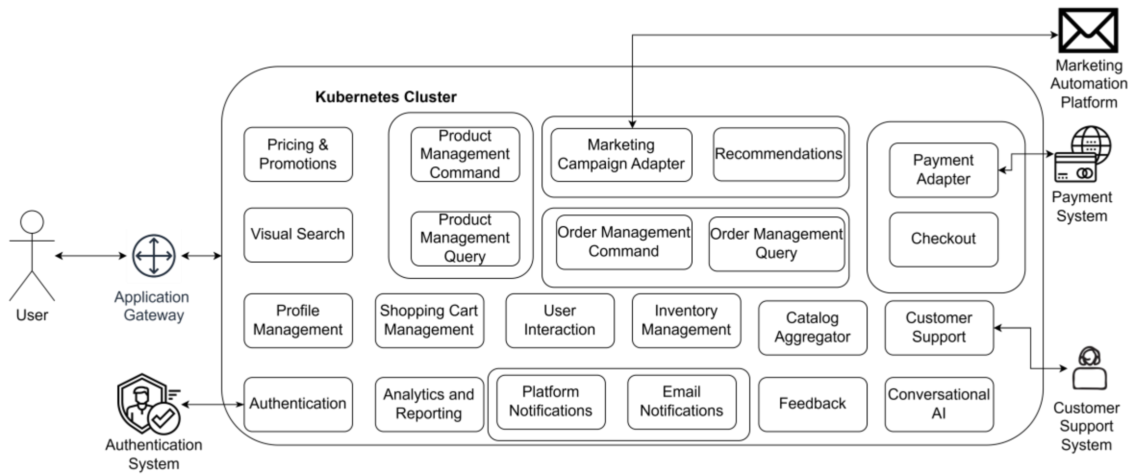

Figure 4.20 Microservice Decomposition

4.6 Microservices Interaction

This section will describe the interaction of microservices to perform some of the user stories (those considered in the MVP). Communication diagrams will be used to describe the interaction.

The main notation elements used are illustrated in the Figure 4.21. Messages are numbered to specify their sequence. Synchronous messages are represented with a filled arrow. To increase clarity, the HTTP verb, URI, and response body are included in the request, while the response specifies the HTTP verb, response body, and headers.

Asynchronous messages are represented with a non-filled arrow, alongside the name of the event published and the object name passed through. If there is no specific object passed, the most important attributes are stated, as in this example, the resource ID is sent with the message.

The number to publish the message is reused to represent the order for services that consume the produced message, since the real order cannot be known; however, to keep the diagram simple, the consumption of the messages and further interactions will be performed in an auxiliary diagram in most cases.

The decimal number can indicate two things:

1. If the process happens in other services that are not the primary use case. For example, if service A calls service B, which internally calls service C, the communication between service B and C will be denoted by decimal numbers since it is not the main process.
2. If the main service performs two calls in parallel, the multiple calls will be numbered with decimal numbers.

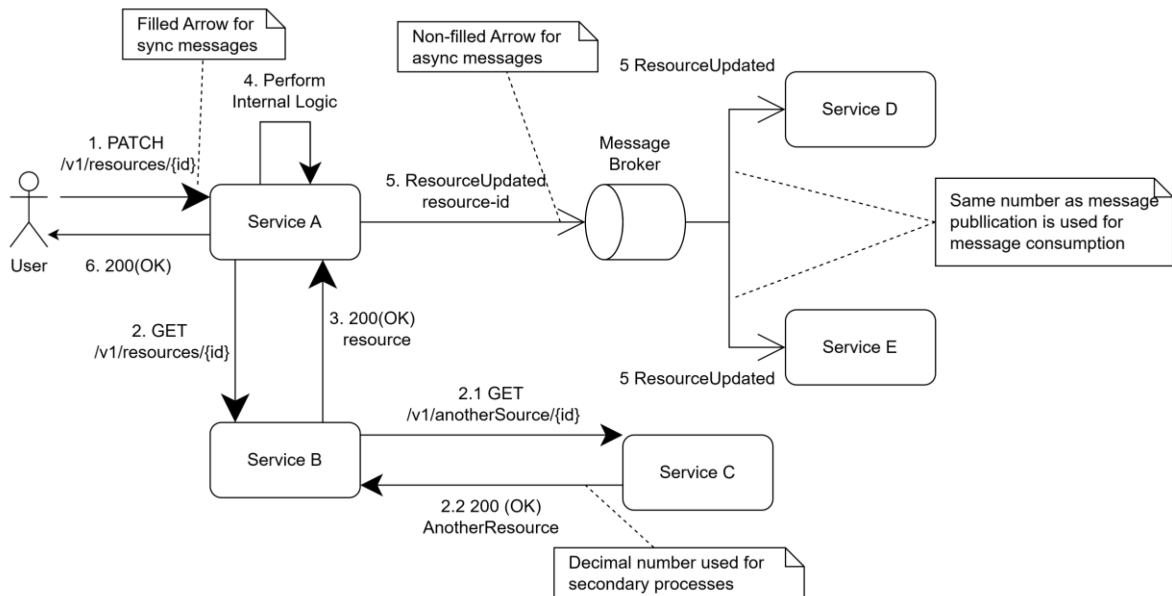

Figure 4.21 Notation to Describe Communication Between Services

4.6.1 Create Product

Creating the product involves multiple steps, which are described below.

4.6.1.1 Upload Product Images

This process is made right after the user uploads an image, which improves performance and user experience since the product creation will be faster.

The seller calls the Product Management Command service with several file names to obtain signed URLs that allow uploading each image. The service generates URLs, which refer to a temporal location in the file storage, and returns them to the front-end. Then, the front end uses these URLs to upload the images directly to the file server.

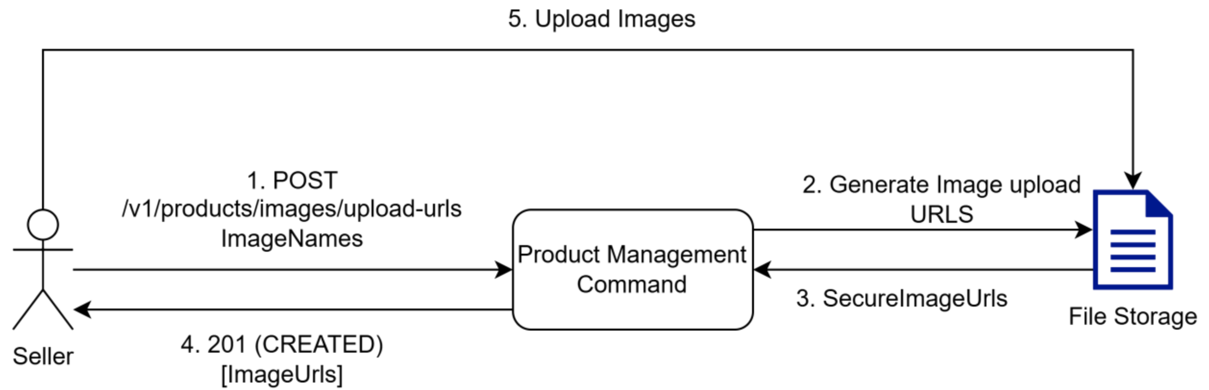

Figure 4.22 Uploading Product Images

4.6.1.2 Suggest Tags

This step helps the user obtain possible tags that the user hasn't put on the product to ensure the completeness of the product information.

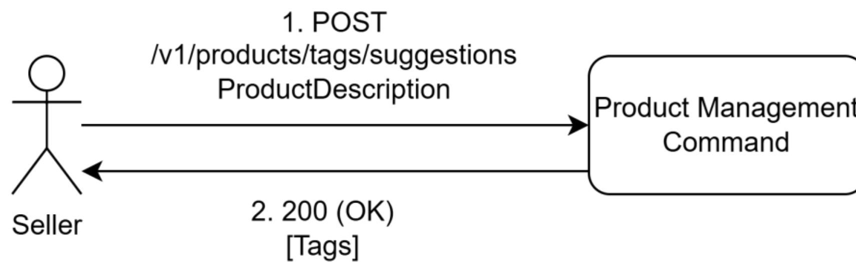

Figure 4.23 Retrieval of Tags Suggestion

4.6.1.3 Product Creation

After the tag suggestion, the seller can accept or reject the tags to perform the second call to create the product. Figure 4.24 illustrates the process, where the temporary files are moved to a permanent location.

One of the most critical metrics in e-commerce applications is the conversion rate, which is the percentage of visits that are transformed into a purchase. According to Shopify[9], their conversion rate is 1.4%, which means that for each 200 visits, around three are converted into a sale. This information provides valuable insight into how different the scaling requirements are from the

---

[9] https://www.littledata.io/ecommerce-conversion-rate

command operations, which generally imply changes in the resource, and the read-only query operations.

For this reason, the “Product Management Command” microservice publishes an event that will be consumed by the “Product Management Query” to replicate the information, following the Command Query Responsibility Segregation (**CQRS**) pattern, which enables independent data modeling and scaling.

Later, the “Email Notification” service sends an email to the seller indicating the creation, and finally, the “Inventory Management” service creates their product representation.

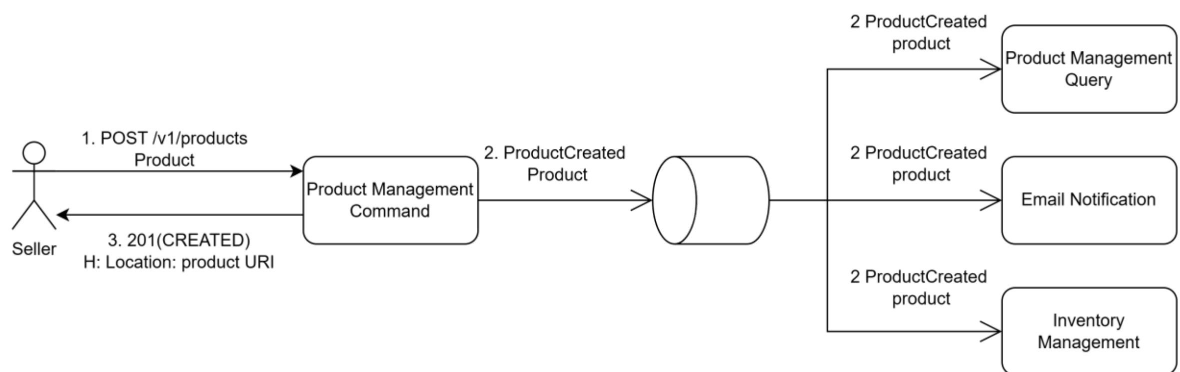

Figure 4.24 Communication Diagram of Product Creation

4.6.2 Filter Product

Product filtering involves the “Product Management Query” service and the “Pricing and Promotions” service. It also includes a cache to reduce the interactions between the services to avoid recalculation of past information.

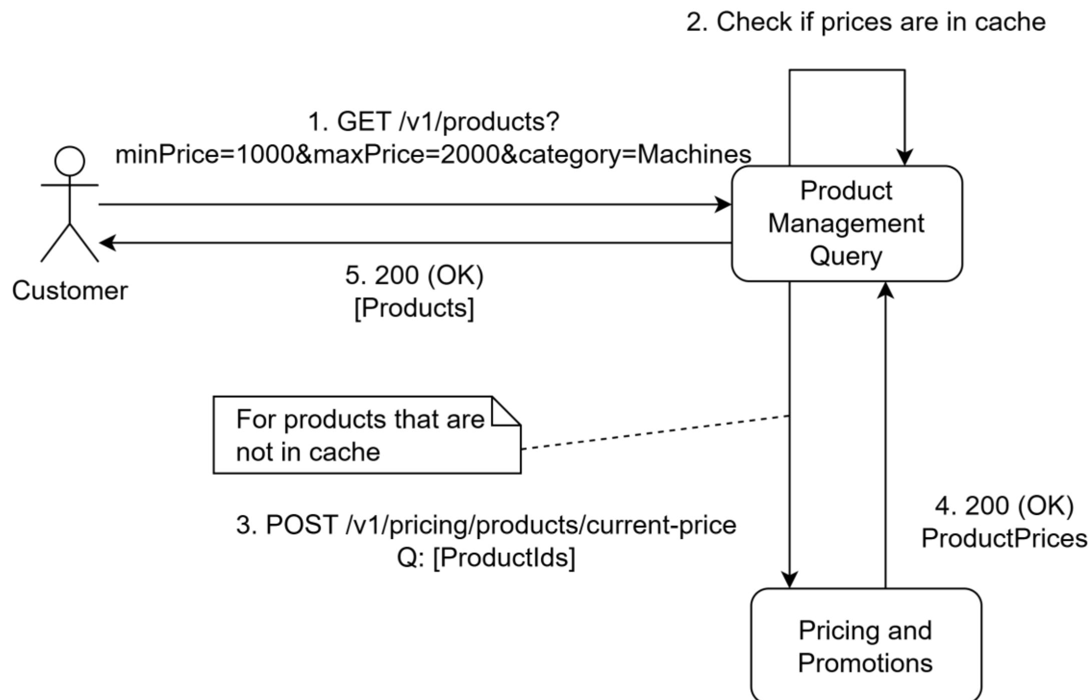

Figure 4.25 Communication Diagram of Product Filtering

4.6.3 View Product Details

The product details view page is affected by two user stories: See product details, stated in the section 3.5.4, and View Similar products, as stated in the section 3.5.10. To retrieve the information, two calls to different services are necessary, one for the recommendation and another for obtaining the product details.

The service calls can be performed in two ways

1. Directly from the front end
2. Perform the calls using a dedicated aggregation service

However, delegating the service calls to the frontend introduces several drawbacks:

1. **Business logic leakage:** One part of the logic is pulled outside the services, and, when the aggregation needs a change, such as integrating a new component or changing the communication between the services, all this responsibility would be taken by the front.
2. **Performance Overhead:** Chatty communication between the front and the back increases the latency, since the intra-service communication inside the Kubernetes cluster is faster.

For those reasons, an aggregation service is chosen. Figure 4.26 illustrates the process of obtaining the information, where the Catalog Aggregator service calls both services in parallel to retrieve the information, following the Backend for Frontends (**BFF**) pattern. It also shows that the Product Management Query service emits an event indicating that the product detail was viewed. This information will help update the recommendation service with a new entry of implicit feedback.

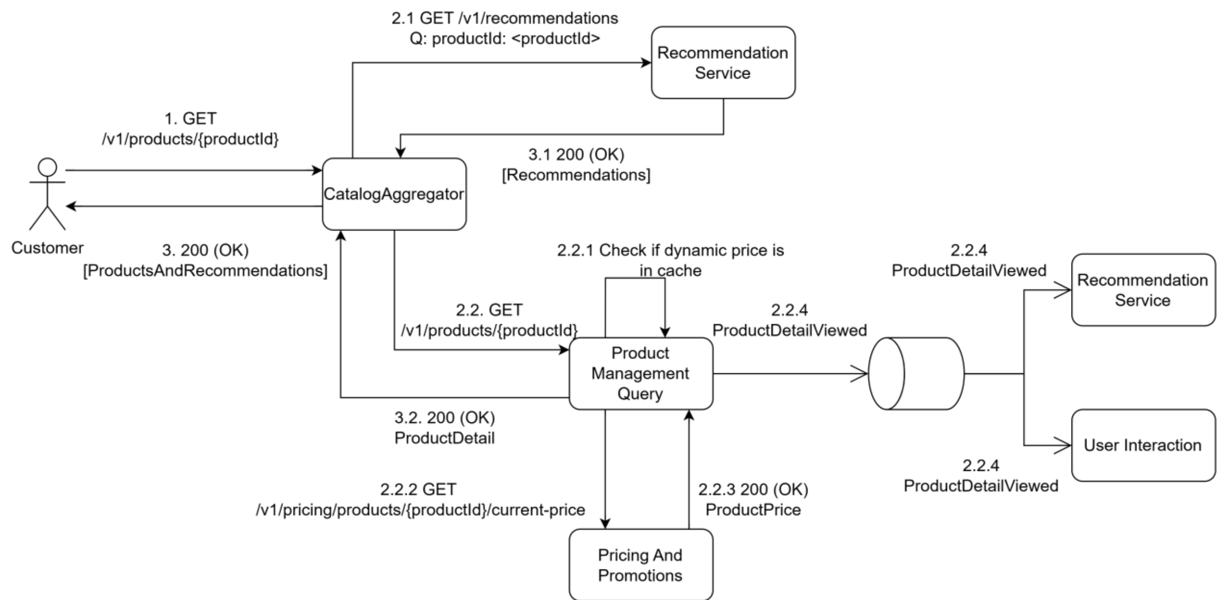

Figure 4.26 Communication Diagram to Obtain Product Details View

4.6.4 Add Product to Shopping Cart

The Shopping Cart Management service will validate the business logic for adding an item to the cart. Figure 4.27 illustrate the process, where it first validate that the stock for the item is available, if there are sufficient stock, it obtains the product information, validates the maximum of items that can be added to the shopping cart, verifies the price with the help of “Pricing and Promotions” service, and finally, emits and event that will be consumed by the recommendation service to update the implicit feedback, and by the User Interactions service to record the operation.

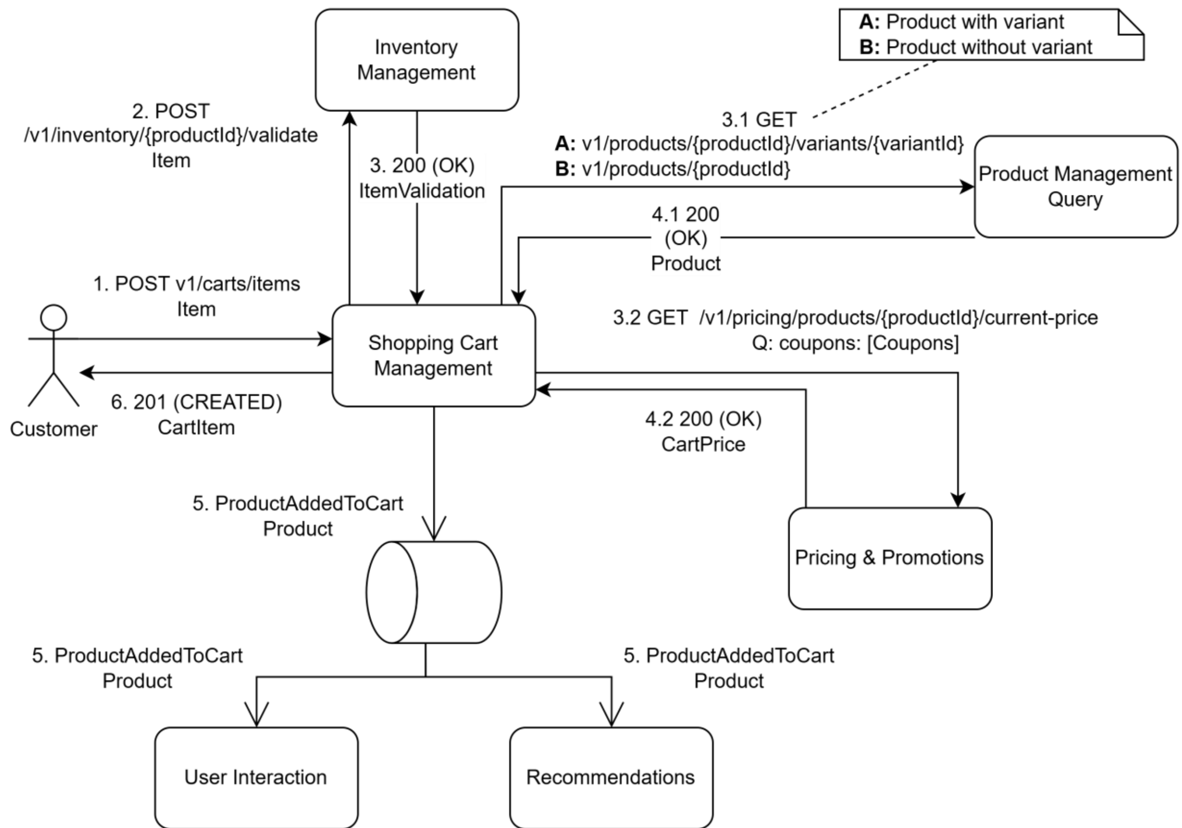

Figure 4.27 Communication Diagram to Add an Item to the Cart

4.6.5 Delete Product from the Shopping Cart

Removing the product is straightforward since it does not have to validate the price, stock, or maximum item amount; it simply updates the internal representation.

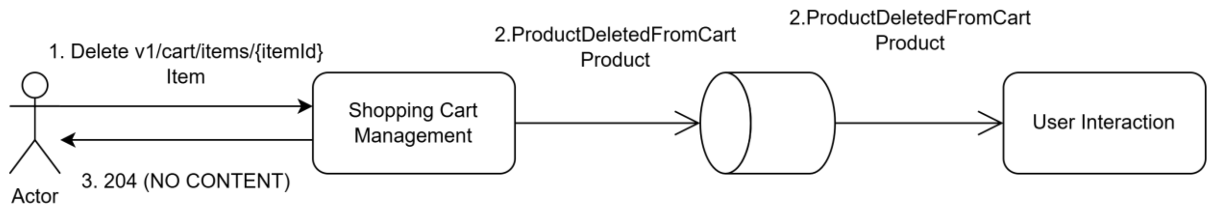

Figure 4.28 Communication Diagram to Delete a Product from the Cart

4.6.6 Purchase Products

Product purchase is one of the application's most important elements. It must ensure the integrity of the information and consistency across the microservices while preserving performance and scalability. For this operation, multiple architectural patterns were implemented to ensure the correctness of the use case; these are described below.

Before delving into the architectural patterns, the primary operations illustrated in Figure 4.29 will be described:

1. The customer sends a POST request to the checkout service; it does not need to send a cart ID since the authorization header contains the user details, and, by design, a user can have only one active shopping cart.
2. The checkout obtains the cart details through the Shopping Cart Management service, which internally validates the prices of all the items and **blocks the shopping cart's modifications** to avoid changes in other devices.
3. The Checkout service reserves the stock of all car items by calling the Inventory Management service, which is called instead of the Query to avoid eventual consistency problems and to access the source of truth. Internally, Product Command reserves the stock and emits a message to notify of this reservation, which is consumed by the Product Query microservice to reflect the changes to other users who are watching products.
4. The Checkout service uses the cart ID as an **idempotency key**, which the payment gateway uses to avoid duplicate payments. After that, it calls the Payment Adapter service, which will call the payment gateway and perform retries in case of timeouts or network errors.
5. After the payment is successful, the Checkout service **generates an order ID**, which will help provide instant feedback to the user.
6. The Checkout service obtains the product summaries to include them in the domain event, providing complete context for the event, and following the event-carried state principle.
7. The Checkout service emits the event *OrderPlaced* using the **transactional outbox pattern** to avoid data loss. In the same transaction, it will save the payment details in the checkout table and the event in the outbox table to ensure that both information belong to the same transaction, ensuring atomicity. This is useful to prevent order losses in case the message broker is unavailable. This event will be read by:
    a. Order Command: To create the order and emit the *OrderCreated* event to populate the information in the Order Query.
    b. Inventory Management: To confirm the purchase of the already reserved items
    c. Shopping Cart Management: To update the shopping cart status to Completed.
8. The checkout service emits the event *PurchasePerformed* with the generated order ID. This event will be read by:
    a. User Interaction: To record the purchase for later usage and audit
    b. Recommendations: To update the user's implicit feedback for the purchased products

c. Email Notification: To send an email to the customer and sellers confirming the purchase
d. App Notification: To notify sellers about the products sold
9. After emitting the event, the Checkout service returns the order ID to the customer.

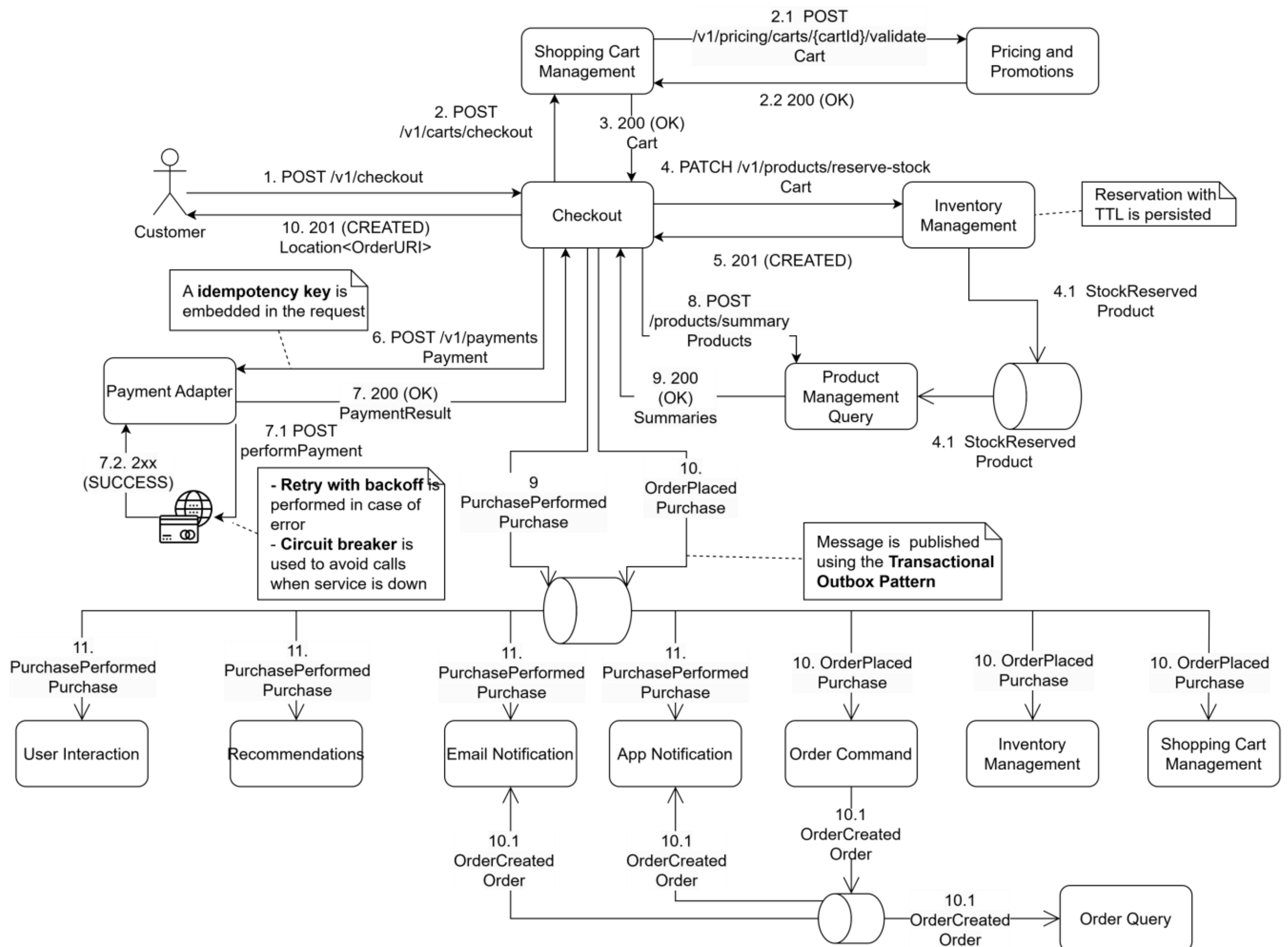

Figure 4.29 Communication Diagram for the Product Purchase

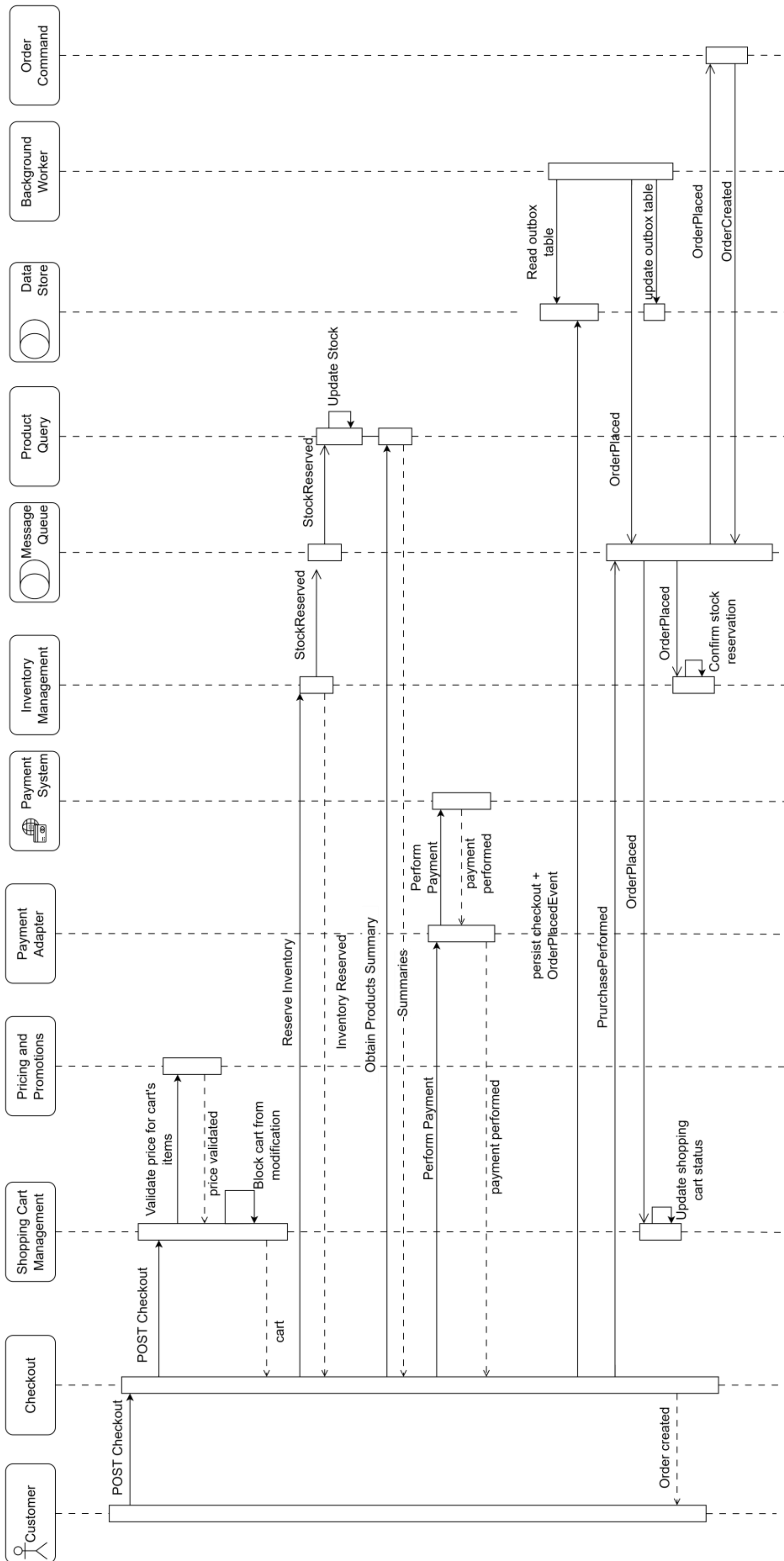

Figure 4.30 Sequence Diagram for Product Purchase

4.6.6.1 Used Patterns

Transactional Outbox, as explained in section 2.5.7, is used to ensure shipment of the event *OrderPlaced*, which is critical to ensure consistency and completion across microservices. If, for any reason, the message is not sent, and we don't have a retry mechanism, the whole application will end in an inconsistent state. To avoid these problems, we can perform a single transaction to write both the business information, which in this case, is the checkout data, and the event information in an outbox table, which a worker will constantly poll to emit the message to the queue, once it successfully writes the message, it will update the information in the outbox table marking the message as shipped.

Saga Pattern, as explained in the section 2.5.4, is used to apply compensation transactions to roll back changes in a distributed transaction across microservices. Specifically, this pattern is used to unreserve the items and unblock the shopping cart if the payment fails.

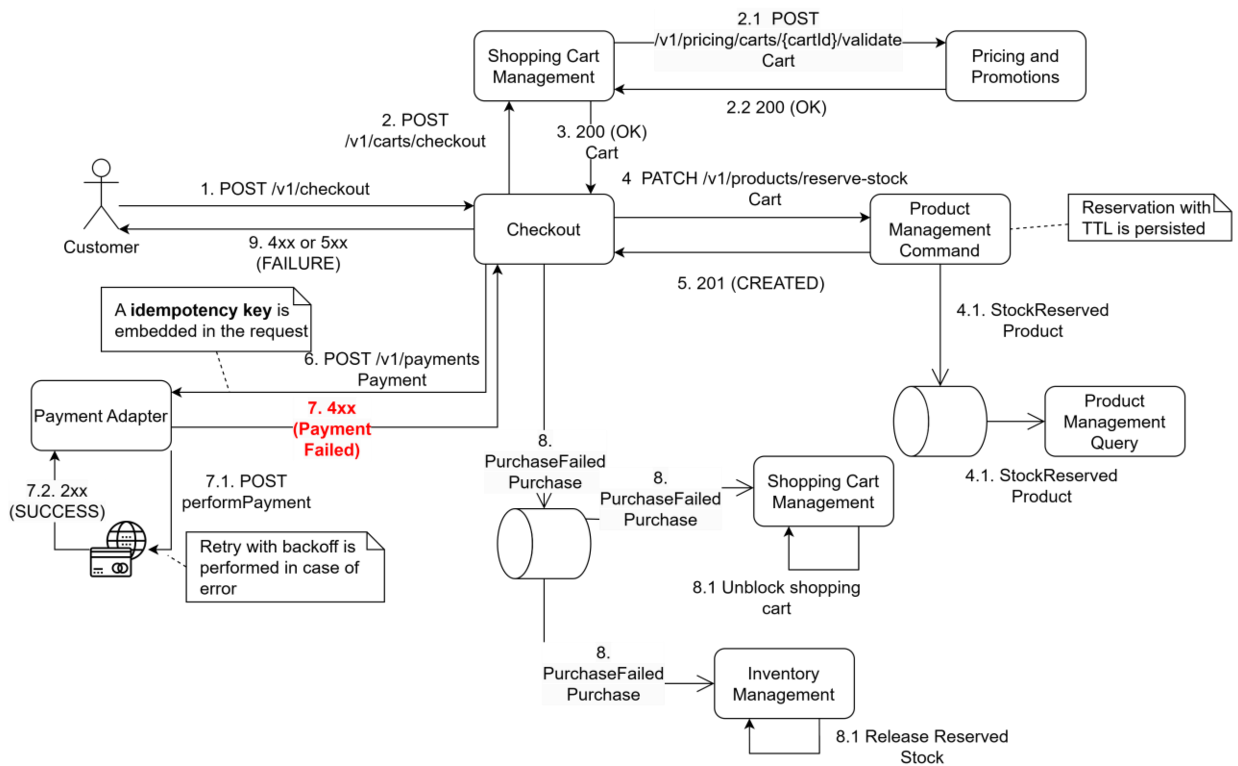

Figure 4.31 Saga Pattern if Purchase Payment Fails

The application can also fail when the stock reservation is attempted; in this case, a compensation event *StockReservationFailed* is emitted

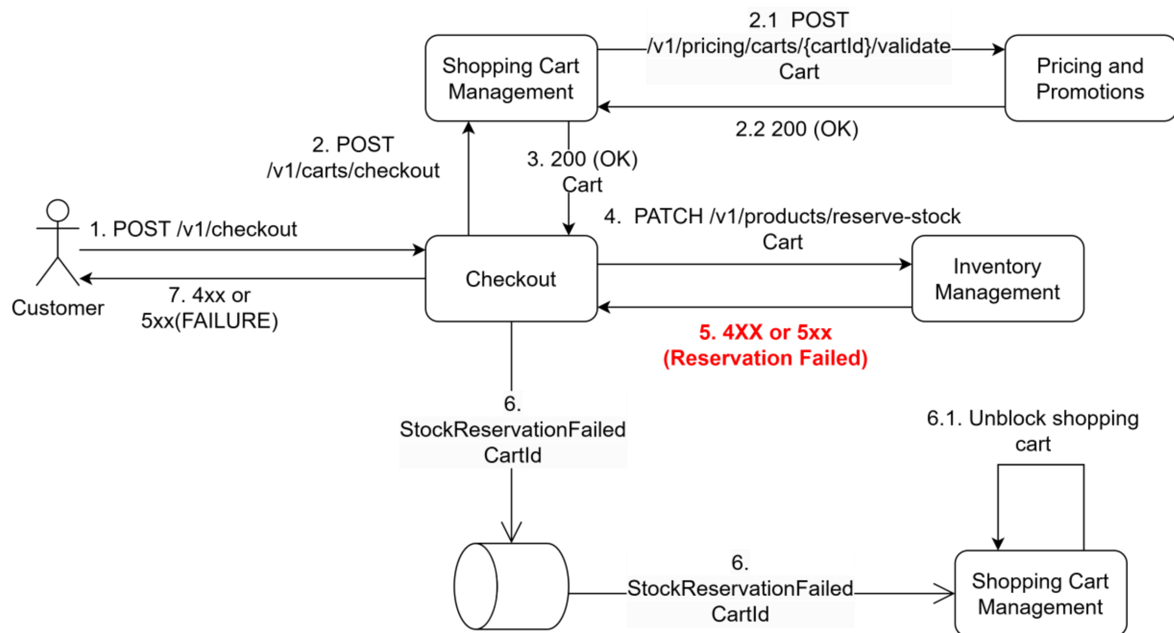

Figure 4.32 Saga Pattern if Stock Reservation Fails

CQRS, as explained in section 2.5.3 is used to enhance performance and independence between command operations (Create, Update, Delete) and query operations (Read). This pattern can be seen in order creation, where the Order Command service emits an event to notify of the order creation. The Query service reads this event and stores the data in a read-optimized database.

Circuit breaker, as explained in the section 2.5.5, is used to prevent unnecessary call attempts to the payment gateway if it is down. The payment adapter will open the circuit breaker to give a default answer and not call the gateway if it has failed multiple times, avoiding the unnecessary timeout waiting for an answer from the down gateway.

Anti-corruption Layer, as explained in the section 2.5.8 is used to avoid merging unrelated concepts between the system's domain and external systems. It is also used to isolate all the changes that the external systems may have. In this specific case, all the translations between the domain language and the external system are performed by the payment gateway, preventing changes in multiple parts of the application if we want to change the gateway in the future or if it implements a breaking change.

4.7 Microservices Infrastructure Decisions

This section states the key technologies employed to implement each microservice within the MVP subset chosen for development.

4.7.1 Programming Language and Framework

Java, which is widely adopted to build enterprise applications, will be the language most microservices use. Its large community helps overcome development problems.

The framework will be Spring Boot, which is the de facto framework for Java, although there are other options, like Quarkus and Micronaut, the percentage of usage is minimal. The features and integration with other applications are limited.

Java, used with Spring Boot, provides a robust foundation for building e-commerce systems. It gives multiple benefits related to SOLID principles (Martin, 2000). For example, related to the **single responsibility principle**, Spring annotations like @Service, @Repository, and @Controller help us to limit the scope of each class. For instance, @Controller exclusively manages HTTP request/response marshalling. Furthermore, this design complements hexagonal architecture, where @Controller acts as the **driving adapter,** while @Service encapsulates core business logic.

Spring aspect-oriented programming (AOP) allows us to add cross-cutting concerns to existing code, following the **open/closed** principle.

4.7.2 Message Broker

Multiple broker technologies, such as RabbitMQ, Kafka, Redis Pub-Sub, etc., support event-driven architecture. One of the main options to consider is Azure Event Hub, which "is a native data-streaming service in the cloud that can stream millions of events per second" (Azure, 2024). Event Hub has the advantage of being an Apache Kafka wrapped in a cloud-native SaaS, which provides the best of both worlds.

Kafka is one of the most popular message broker technologies, it is well known for providing the ability for replay messages, for example, if a Kafka topic has multiple consumer groups, each group know which is the last message read, so they can stop reading the messages (if a consumer crashes or there is a problem deploying a new version) without losing them; once a consumer is connected to the group, all pending messages will be delivered to it. The complete explanation of Kafka can be read in the section 2.4.2.

Cloud-native SaaS provides seamless integration with other cloud services, which speeds up application development and continuous application delivery.

4.7.3 Data Store

4.7.3.1 Product Management

The product management domain is divided into two microservices, following the Command-Query Segregation (CQRS) pattern, which enables independent scaling and improves performance.

The database for the command microservice must ensure the business rules and be ACID compliant; for those reasons, a relational database is the best option. PostgreSQL is a widely used relational database that has gained popularity over the years, according to Stack Overflow[10] (2024), it is the most popular database, which provides it with a robust community and constant upgrades to evolving standards. Azure provides its fully managed PostgreSQL through Azure Database for PostgreSQL, which provides features like automatic backup, zone-redundant high availability, with an SLA of 99.95%

The database used for the query microservice must ensure fast access and flexible search. It should be optimized for rapid retrieval, combined with a strong denormalized approach. Azure AI search is a read-optimized database; it provides the same benefits as other services like Elasticsearch, but with extensive functionality. For example, it provides the option to transform images into searchable chunks, which reduces the complexity overhead of implementing it from scratch. The downside is that it is a vendor-specific platform, which will reduce the simplicity of migrating to another cloud provider. However, the benefits provided make it a good choice.

The Product Management Query service can store information in a cache database to reduce database calls and improve performance. Redis is the de facto standard for cache databases; it can handle up to 100,000 queries per second and provides handy features like TTL expiration. To ensure high availability and reduce maintenance time, Redis is available as a service through Azure Cache for Redis. This database is also used to store temporal images in the product management command service.

Azure Blob Storage will be used to manage product images since it provides highly scalable and robust file storage, which enables us to meet the demands of a growing catalog.

---

[10]https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database

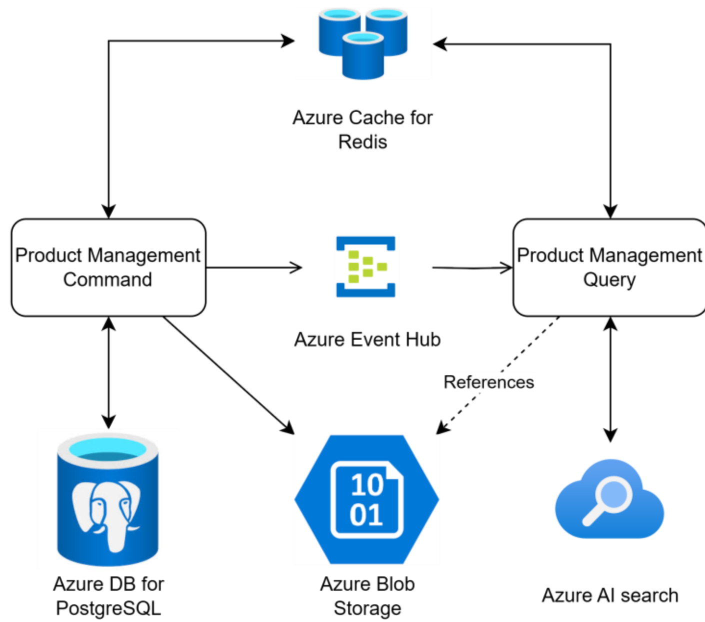

Figure 4.33 Data Stores for Product Management Services

The image below shows the most critical information stored in the relational database; for simplicity, it does not show all the fields. Information like the creation date and the updated date must be stored. The data stored in Azure Search is a denormalized version of the relational database.

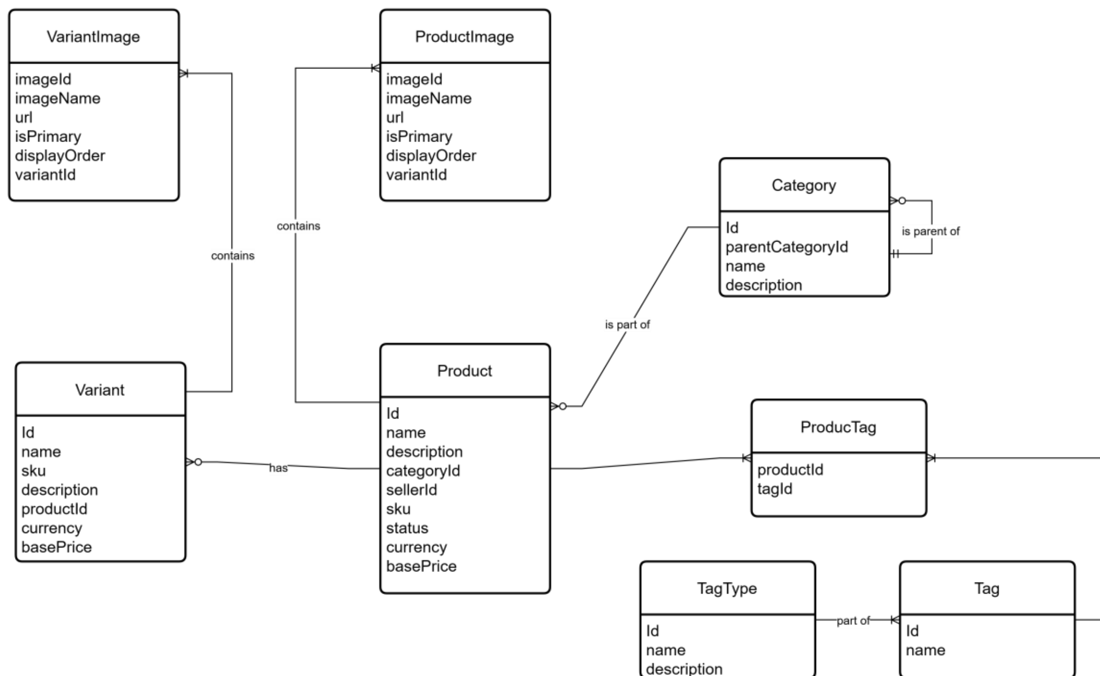

Figure 4.34 Product Management Data Model

4.7.3.2 Shopping Cart

The shopping cart service can be implemented in a relational DB to enforce business rules and ACID compliance.

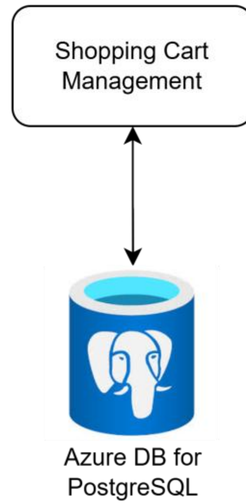

Figure 4.35 Data Store for Shopping Cart

The image below shows the most important fields to store.

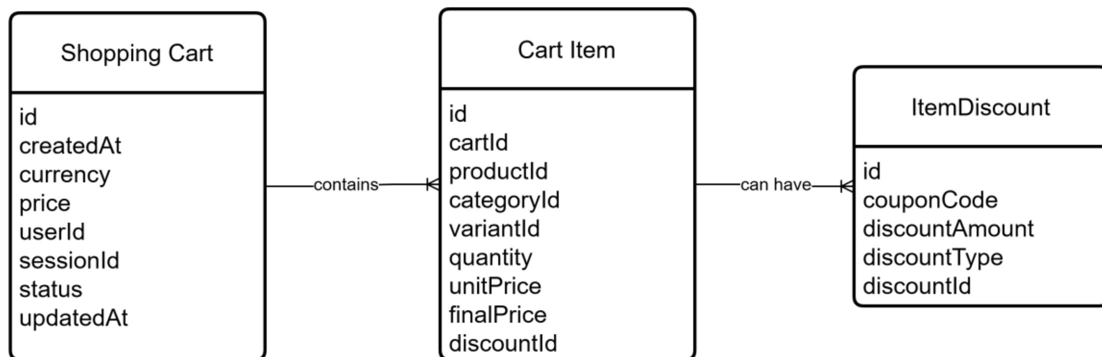

Figure 4.36 Shopping Cart Data Model

4.7.3.3 Checkout

The checkout context is an orchestrator process; therefore, it does not store information about the order, such as the shipping address or discounts. The main focus of the data stored is to save the raw interaction with the payment gateway, mainly to address audit concerns.

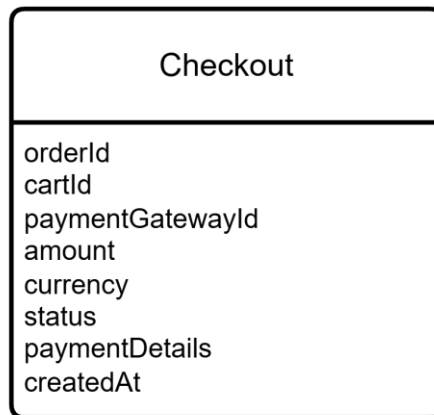

Figure 4.37 Checkout Data Model

The checkout information does not need to be query optimized. Therefore, a relational database is a good fit.

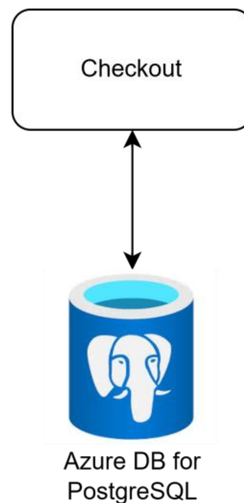

Figure 4.38 Data Store for Checkout

4.7.3.4 Order Management

The order management information should be optimized to allow frequent access since customers and sellers can browse past orders to obtain tickets or search for a product. Although it will be retrieved frequently, the demand for obtaining an order differs from the demand for browsing products. Also, it does not require AI capabilities and custom searching, which Azure Search provides. Therefore, a cheaper service like MongoDB will also work and reduce the platform costs. CosmosDB for MongoDB will provide cloud reliability to the database, reducing maintenance and downtime costs.

Order status can change throughout time. For example, when the order is shipped or received, the command service will hold these actions.

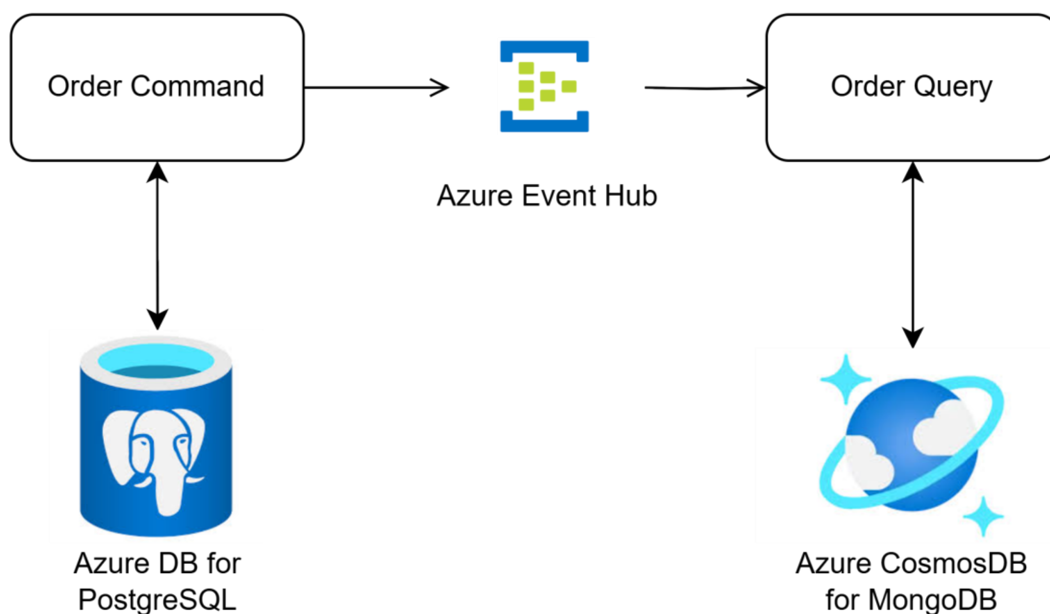

Figure 4.39 Data Store for Order Management

The information stored in the order is mostly a snapshot of the time it was purchased, for example, the price charged for each product.

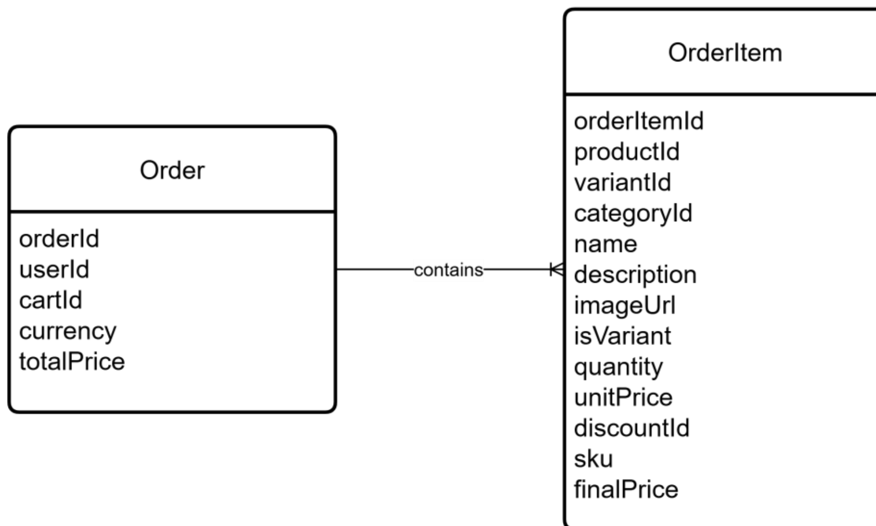

Figure 4.40 Order Management Data Model

4.7.3.5 Price and promotions

The information managed for price and promotions should follow multiple business rules. A relational database is useful for imposing the needed constraints and ensuring the integrity of the information.

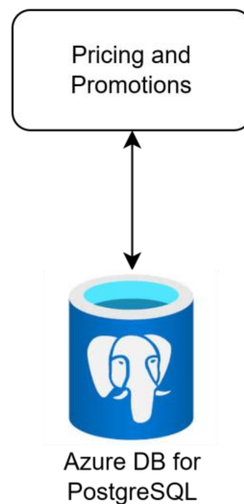

Figure 4.41 Data Store for Pricing and Promotions

The schema used to save promotions should be flexible enough to allow new discount rules in the future. For example, if a new user requirement arises that says an administrator could create discounts for an entire category, the database schema should be flexible enough to develop the functionality without significant changes. Although all the future use cases cannot be covered because of the variety that they can have, the proposed data store is intended to support extensibility.

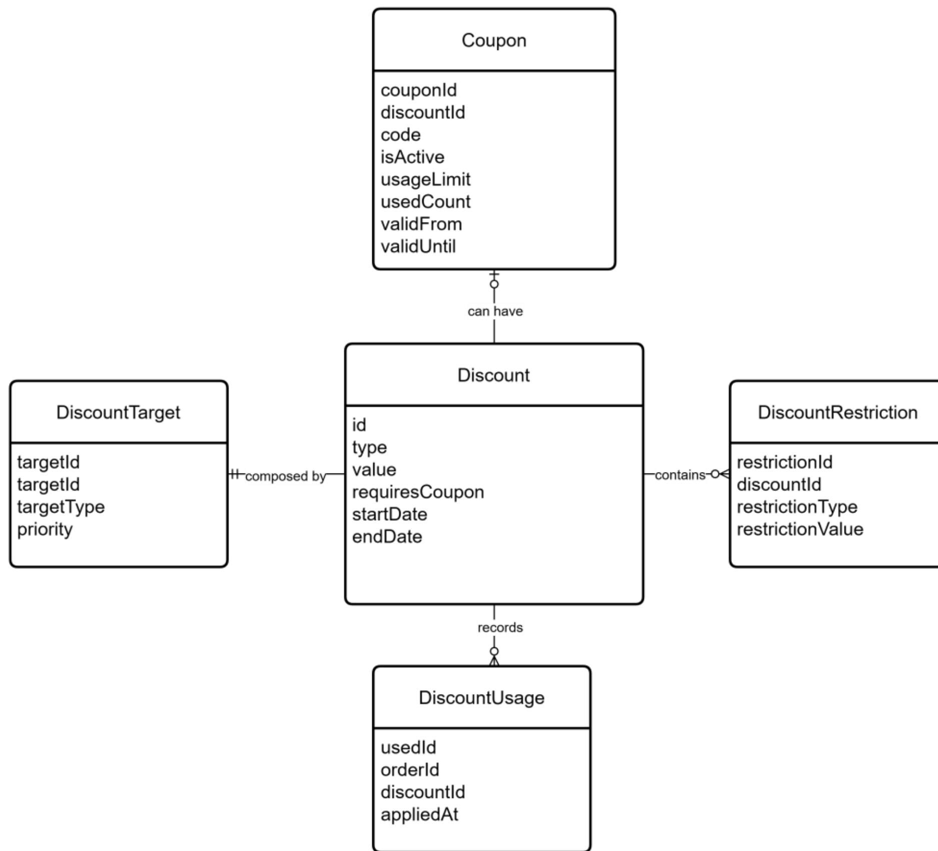

Figure 4.42 Pricing and Promotions Data Model

4.7.3.6 Inventory

To maintain transactions and enforce constraints, a relational database was chosen for this microservice.

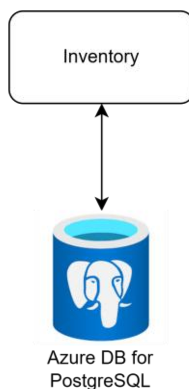

Figure 4.43 Inventory Database

The data saved in the service is straightforward since the product management service owns all the other attributes. It stores the inventory and the reservations.

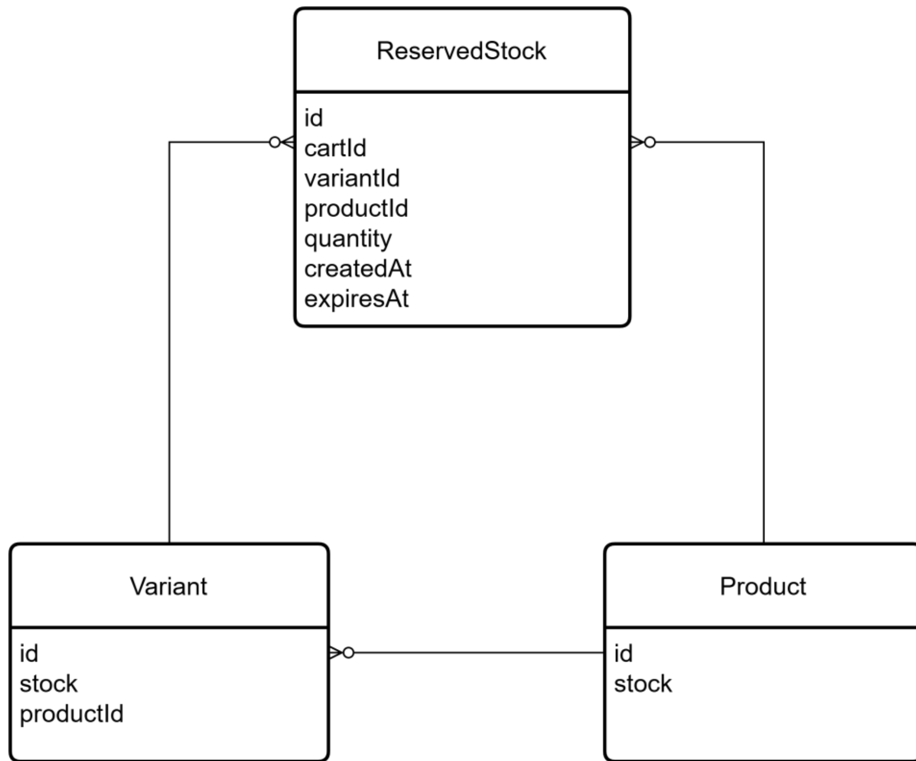

Figure 4.44 Inventory Data Model

5 Development and Detailed Design

This section describes the detailed design with class diagrams and the development process of the required microservices that are part of the minimum viable product (MVP) subset, which involves:

a) Product creation, which involves the upload of product images, the product registration in the command and query services, and the inventory registration, described in section 4.6.1.
b) The process to purchase a shopping cart, described in section 4.6.6

The microservices' GitHub repositories are available here[11].

The detailed design through class diagrams and the development are presented together to maintain clarity and continuity in explaining each microservice's behavior. Separating these sections would break the logical flow, making readers move between sections to understand how the design translates into actual code.

The summaries of the implemented functionalities, which are described thoroughly in later sections, are:

1) **Create a Product:** This process is composed of two endpoints to improve the application's response time, which are

   a) Generate Image URL: It will return a signed URL so that the client (which, in the real product, would be the frontend framework) can upload the file directly to the file storage.
   b) Create product: Receives all the information, including the image ID returned before, to create a new product. A domain event is published when the product is created, activating multiple microservices, like inventory, product query, etc. All of these interactions are implemented.

2) **Validate Item Prices:** The pricing microservice receives product or variant identification, base price, and reduced price. It validates whether a promotion lowers the base price to the final price. The system is designed to support individual product or variant promotions and promotions applicable to entire categories. It can also validate specific promotion restrictions, such as the maximum number of uses per customer.

3) **Checkout Cart:** The cart microservice receives a cart ID, calls the pricing microservice to validate the item prices, and changes the cart status to PROCESSING if it is in a valid state to perform this transition. Otherwise, it will return an error.

4) **Reserve Inventory:** In addition to the inventory creation that occurs when the product is created, the inventory microservice exposes an endpoint that

[11] https://github.com/stars/davidcediel12/lists/coffee-e-commerce-project

receives a list of product/variant IDs and the amount that needs to be reserved. It reserves the inventory and publishes an event indicating this reservation, which the product query service consumes to update its inventory.

5) **Perform Payment:** The payment adapter microservice communicates with the payment gateway to perform the payment.

6) **Order Creation:** The order and order query microservices implement the functionality to create an order, received by a domain event published by the checkout service.

7) **App Notifications:** The app notification service maintains a unidirectional communication channel with the client through server-sent events to send notifications.

8) **Purchase Products:** The checkout microservice, which is the central orchestrator, performs the required calls to pay for a shopping cart and create an order.

The OpenAPI schema will be shown for all the HTTP endpoints since it provides valuable information at first glance, such as the URI, request body, response code, response body, headers, etc.

The objective is to apply the best development practices to build the system, increasing the initial development time but providing a robust solution that can be extended and modified easily. These practices are the following:

- **Test-driven development (TDD):** It is helpful to create a thorough test suite that addresses various scenarios. The TDD cycle slows development, but live documentation empowers developers to make modifications without breaking the code. This technique was only used to develop the product microservice, as described in section 5.3. However, TDD was not applied in other microservices due to time constraints and the prioritization of the MVP's development.

- **Observability:** Configuring an observability tool like Application Insights requires time, but it provides multiple benefits, such as the ability to find bugs faster, analyze the system health, see potential problems in advance, etc.

- **API-first:** Writing a contract for every endpoint before starting the development is time-consuming. A significant percentage of projects obtain the API contracts as documentation with a plugin after the development. However, writing the API specification helps the quality of the contract since it is a shared responsibility between the consumers and the producers. It also specifies the rules before making the API, reducing refactors due to missing parameters.

- **Use of test containers:** Test containers allow testing services to use the same technology that is in production. For example, suppose the microservice sends a message through Kafka. In that case, instead of

mocking the message's sending, it can be sent to an actual Kafka instance used for testing, running in a booted container just for the test.

- **Contract test:** Since microservices architecture has many moving parts, testing them appropriately is essential to the system's success. Contract testing allows for testing the interaction between services with a specification verified among the participants, which is more robust than a mock that cannot represent the real answer of the service. It requires time to create and maintain the contracts, but provides tests that simulate the actual service behavior.

- **Hexagonal Architecture:** Hexagonal architecture requires a clear distinction between the adapters, ports, application services, and domain services. Properly performing these distinctions requires more time and sometimes more mapping between different layers' classes. For example, the distinction between domain and persistence entities isolates the business domain but requires more classes and mappings between them. This distinction was no longer made after the development of the Product Management microservice, since it adds unnecessary complexity to the system. However, the principles of hexagonal architecture regarding rich domain entities have been kept.

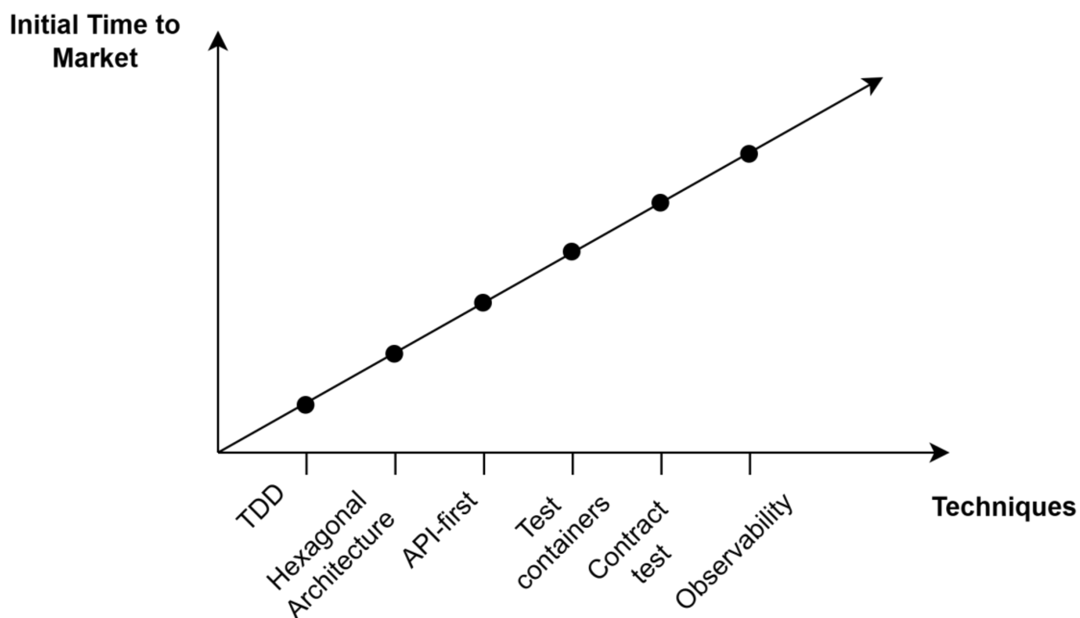

Figure 5.1 Increased Time to Market

5.1 Shared Architecture

Most services follow a hexagonal architecture, separating business logic from external concerns such as databases and messaging systems.

Each service typically includes:

1. **Driving Adapters:** These classes are the entrance to the application, either through a REST endpoint or message consumption. Their primary responsibilities are to receive the information, deserialize it properly, and transform it into command objects required by the driving ports. These classes can also perform basic attribute validation, such as format, nullability, etc.

2. **Driving Ports:** These classes are the contract that driving adapters must follow to communicate with the application core.

   One of the fundamental concepts in hexagonal architecture is the application's central role. The infrastructure layer is designed to adapt to the core needs, not vice versa. To support this separation of concerns, the architecture uses the **inversion of control** and **dependency inversion** principles, which gives the app core complete control by defining contracts (ports) that describe how it expects to interact with the upper layers. This design ensures that the core remains independent and agnostic to implementation details.

3. **Application Service:** These classes are orchestrators implementing a specific use case. They perform the required calls to achieve the process. However, they do not implement business logic since it is implemented in the rich-domain entities and, in some cases, in the domain services when a process involves multiple entities.

4. **Domain Service:** These classes implement business rules and invariants depending on multiple entities.

5. **Domain Entities:** These classes represent the entities related to the bounded context. They are rich domain models since, rather than simple data stores, they perform business rules, invariants, state changes, etc. This allows the business logic to be in the right place rather than spread across multiple services.

6. **Driven Ports:** These classes are the contract that driven adapters must follow to receive the calls the application core performs.

7. **Driven Adapters:** These classes are the communication point between the application and external elements. For example, if the application saves the information to a database, like PostgreSQL, the driven adapter is aware of this detail and will communicate with it.

Although the domain entities are responsible for implementing the business rules, they will be intentionally omitted from the class diagrams to provide a higher-level architecture view. This simplification improves readability and focuses the diagrams on the interactions between services, ports, and adapters, which are the most relevant aspects of the hexagonal architecture.

To implement the hexagonal architecture, many mappings exist between the layers' objects. These classes are omitted from the class diagram to provide clarity and focus on the described use case.

The figure below presents the template that will be used to describe the majority of the microservices.

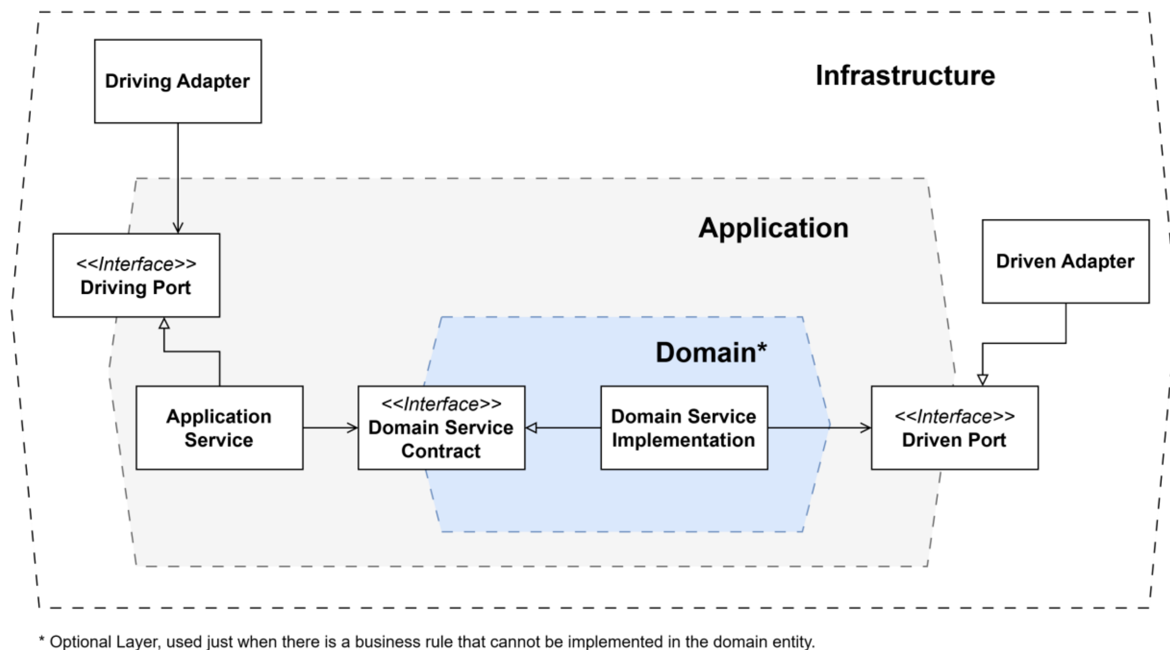

Figure 5.2 Class Diagram Template

5.2 Technologies

All the microservices were developed using the following technologies:

- **Gradle** as a build and dependency management tool.
- **Java 21** (LTS) as the development language.
- **Spring Boot** 3.4.4 as the web framework.
- **Junit5** as the testing framework.
- **Docker** as a containerization engine.
- **Hibernate** as the Object Relational Mapper.
- **Kubernetes** as a container orchestrator.

5.3 Package organization

This section presents the foundational class structure followed by each microservice. While each service is independent and may vary from this design, this is the baseline to follow.

The package structure used by each microservice is intended to follow the **Hexagonal Architecture**, as explained in the section 2.5.1. It isolates the business logic by using driving ports to receive external calls, and driven ports to send information to external services.

Hexagonal Architecture can still be preserved using annotations. For example, @Controller and @KafkaListener classes will be understood as driving adapters, while @Repository classes will be driven adapters. Another improvement that can be made is storing all the domain classes in a plain package called *domain*, which improves project navigation. The package structure is illustrated in the image below.

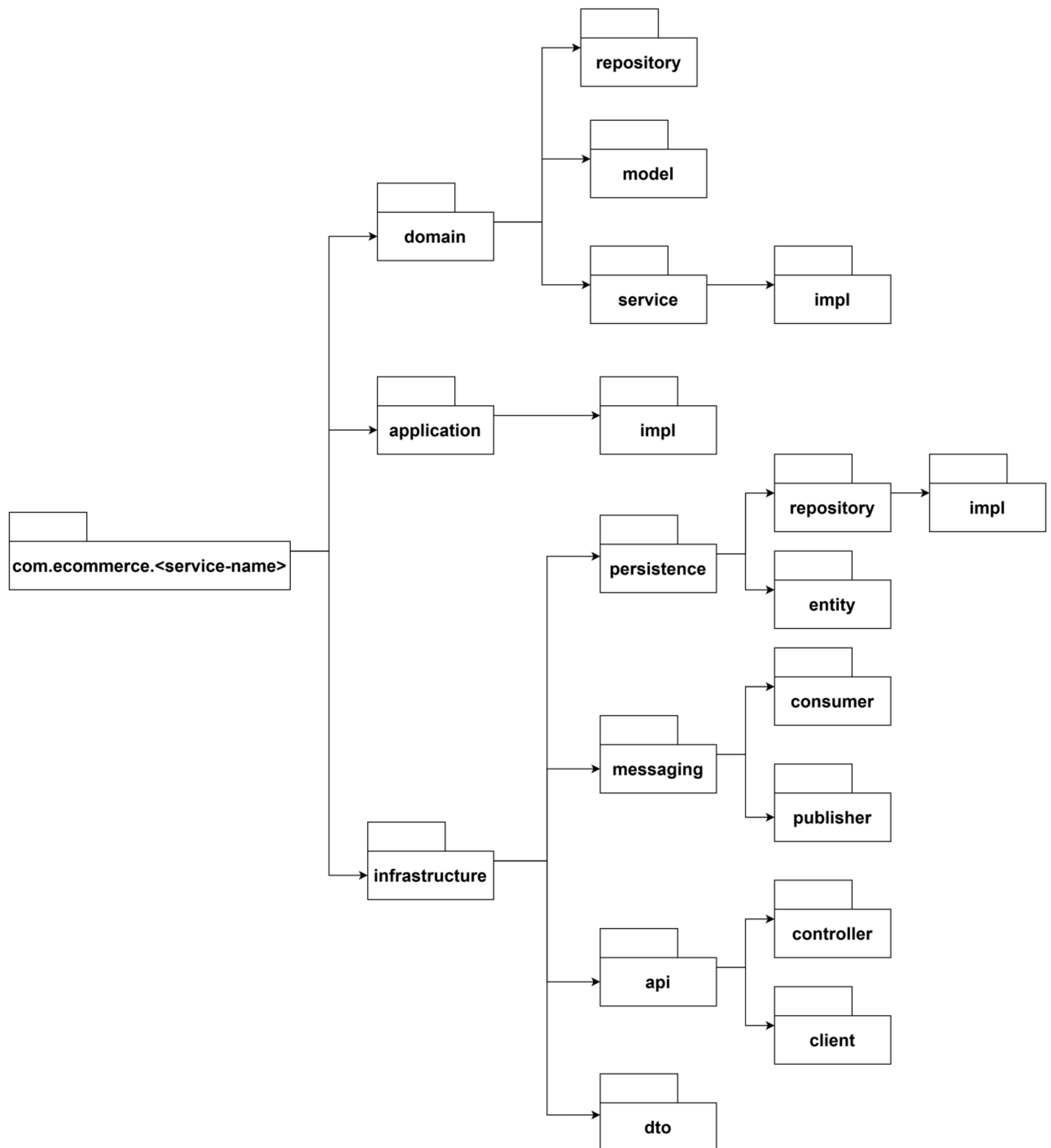

Figure 5.3 Package Structure for Microservices

The responsibility for each package is:

a) **Domain:** It is the internal part of the hexagon, which contains the business logic and the DDD entities. These classes should not depend on the technical details of the framework.

   The *service* package contains the interfaces that serve as an entry point to the domain services, while the *impl* package contains the implementation, which includes the domain logic that applies to multiple entities. This package is optional, since in some cases the business logic

can be performed on the entity, and the remaining operations are part of the orchestration.

b) **Application:** It contains the services that implement the application's use cases. Application services are orchestrators that call domain services to perform a specific scenario.
c) **Infrastructure:** It is the outer part of the hexagon, which contains all the classes needed to connect with the external world. It is divided into multiple subpackages.
    a. **Persistence:** Contains the classes related to the data store, such as repositories and entities
    b. **Messaging:** Contains the classes related to event-driven elements, such as publishers and consumers
    c. **API:** Contains the classes related to REST, such as controllers and clients that call external APIs
    d. **DTO:** Contains all the classes used to transfer data with external services, either by events or by sync calls.

Each package can have its sub-packages, such as mappers, exceptions, or utility classes; these depend on each microservice and are avoided to improve the diagram clarity.

After the development of the Product Management Command and Query microservices, the domain entities and infrastructure entities were merged to increase the development speed and avoid redundancies. Although the entities will have infrastructure annotations after this change, they will retain their business logic and invariant enforcement. If a modification needs to be made, the code does not have to be modified; only its annotations require changes.

5.4 Product Management

This section describes the process of developing the product management's bounded context.

5.4.1 Generate Upload URLs

The first step in creating a product is uploading the images immediately after the seller submits them. The first step is making the API contract. This activity is one of the most important since this is how clients interact with external systems, and once clients are integrated with it, making changes is expensive.

Figure 5.4 shows the URI and the request body, which is a list of image names.

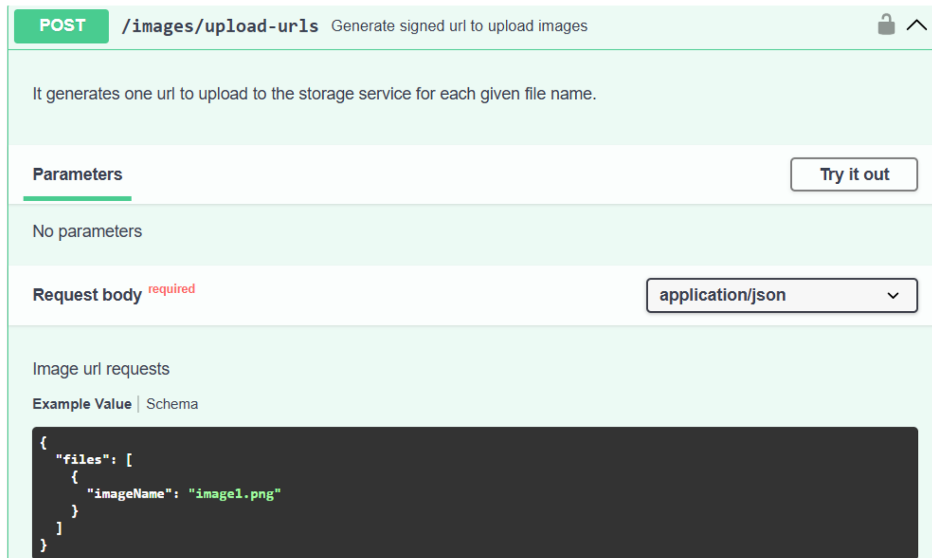

Figure 5.4 Request to generate Upload URLs

If the operation is successful, the service will return a list of URLs along with an ID that can be used later when the product is created.

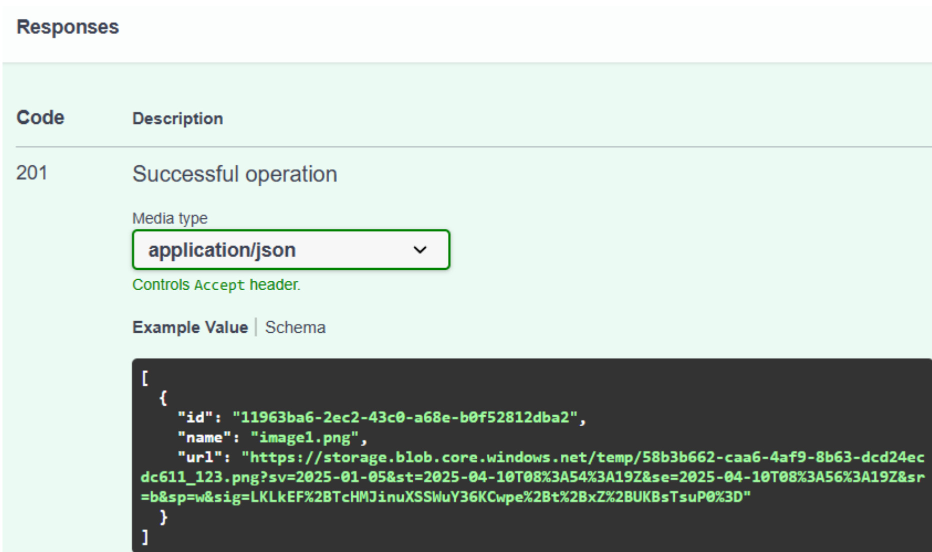

Figure 5.5 Response of obtaining Image URLs

After defining the contract, development is performed using TDD, ensuring that most scenarios are tested. This practice results in 24 tests, 19 of which are unit tests, 3 are integration tests, and 2 are end-to-end tests.

The integration and end-to-end tests were performed using the tool “Testcontainers” described in the section 2.6.1. It allows us to start containers for Redis and Azure Storage to perform the test with an environment as close as possible to the production environment.

```
@Container  1 usage
private static final GenericContainer<?> AZURITE_CONTAINER = new GenericContainer<>(
        DockerImageName.parse( fullImageName: "mcr.microsoft.com/azure-storage/azurite:latest"))
        .withExposedPorts(10000, 10001, 10002);

@Container  2 usages
static RedisContainer redis = new RedisContainer(DockerImageName.parse( fullImageName: "redis:7-alpine"));
```

Figure 5.6 Azurite and Redis containers in test class

The benefits of using hexagonal architecture were perceived during the development of this endpoint. Initially, Spring Data repositories were used to manage the interaction with Redis. However, after analyzing the Redis database, it can be seen that Spring Data creates a set of temporal image IDs without expiration time, causing the set to grow indefinitely, wasting resources and increasing the infrastructure cost.

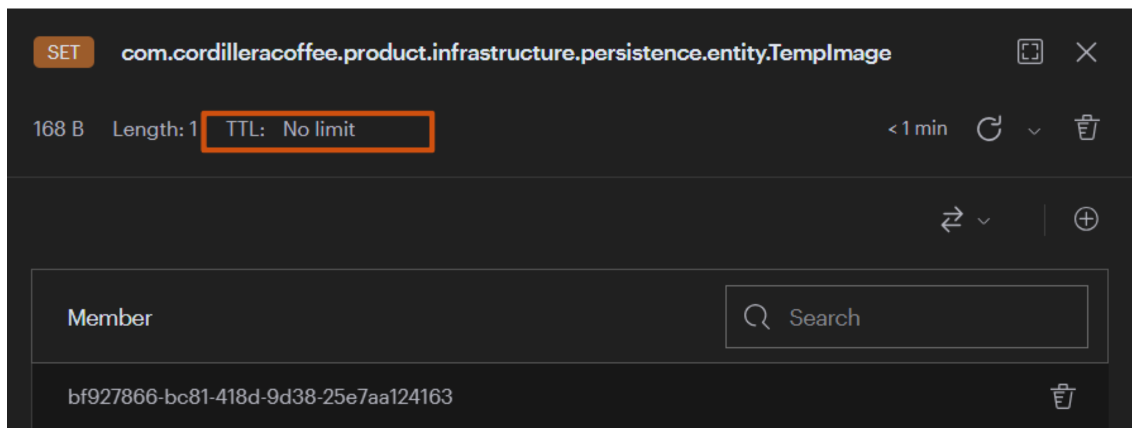

Figure 5.7 Persistent Set in Redis Created by Spring Data

To resolve this, Spring Data repositories were changed to RedisTemplate, a low-level way to interact with the database. This change requires modification only in the infrastructure package, without modifying lower layers. The application layer, which holds the use case, remains intact.

product.main 4 files src\main
java\com\cordilleracoffee\product 4 files
config 1 file
RedisConfig.java
infrastructure\persistence 3 files
entity 1 file
TempImage.java
impl 1 file
RedisImageRepository.java
TempImageRepository.java
product.test 1 file src\test
java\com\cordilleracoffee\product\infrastructure\persistence\impl
RedisImageStorageRepositoryTest.java

```
Use Redis template instead of Spring data repositories to
manage redis access
```

Figure 5.8 Hexagonal Architecture preventing changes to upper layers

Figure 5.9 illustrates the class diagram of the use case and the hexagonal architecture's different layers, with their respective ports and adapters. The scenario has no domain layer since the domain entities have not yet been created.

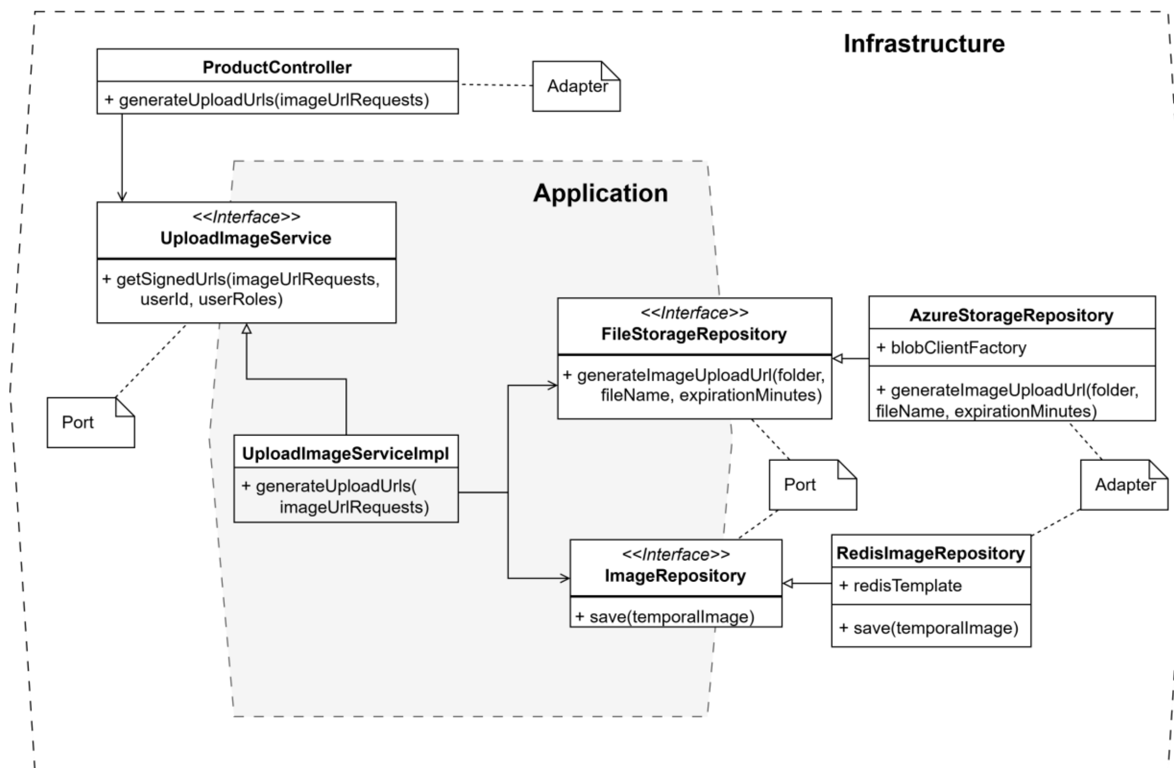

Figure 5.9 Class Diagram of Generate Image URLs Use Case

The primary responsibilities of each class are:

Class	Type	Responsibility
ProductController	Driving Adapter	Receives and transforms the HTTP request into a command object
UploadImageService	Driving Port	Communication door to the core layer
UploadImageServiceImpl	Application Service	Use case orchestrator, performs the required calls to upload the image
FileStorageRepository	Driven Port	Abstraction to remain the core agnostic from infrastructure details
ImageRepository	Driven Port	Abstraction to remain the core agnostic from infrastructure details
AzureStorageRepository	Driven Adapter	Concrete implementation that communicates with Azure Blob Storage
RedisImageRepository	Driven Adapter	Concrete implementation that saves the temporal image details

In the application, "Use cases orchestrate the flow of data to and from the entities, and direct those entities to use their Critical Business Rules to achieve the goals of the use case" (Martin, 2017, p. 26). A custom annotation has been created to state that a class is a use case.

UseCase.java

```
@Service  4 usages  David Cediel
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Validated
public @interface UseCase {
}
```

Figure 5.10 Custom UseCase Annotation

This annotation can be used for services in the application layer, providing context to people analyzing the project.

Figure 5.11 Service using Custom UseCase Annotation

5.4.2 Save Product

The product service moves the product and variant images to a final storage path and publishes an event indicating a new product has been created. The figure below presents the implemented endpoint contract.

POST / Creates a product

Parameters

Try it out

No parameters

Request body

application/json

Object to create a new product

Example Value | Schema

```
{
  "name": "Grinded coffee beans",
  "description": "Grinded coffe beans in a black package with the image of the farm",
  "category": {
    "id": 20
  },
  "sku": "CAF-MR",
  "stock": 55,
  "status": "AVAILABLE",
  "basePrice": 30.5,
  "images": [
    {
      "id": "b08ee4b0-22bb-446c-bd33-0343a77e9b11",
      "isPrimary": true,
      "displayOrder": 2
    }
  ],
  "variants": [
    {
      "name": "package of 250gr",
      "description": "Small package",
      "stock": 55,
      "basePrice": 30.5,
      "isPrimary": true,
      "images": [
        {
          "id": "b08ee4b0-22bb-446c-bd33-0343a77e9b11",
```

Responses

Code	Description	Links
201	Successful operation	*No links*

Headers:

Name	Description	Type
Location		string *Example: http://localhost:8080/v1/products/product-id*

Figure 5.12 Save Product OpenAPI Schema

The Figure 5.13 illustrates the classes used to implement the create product use case, with the layer where they belong according to the hexagonal architecture.

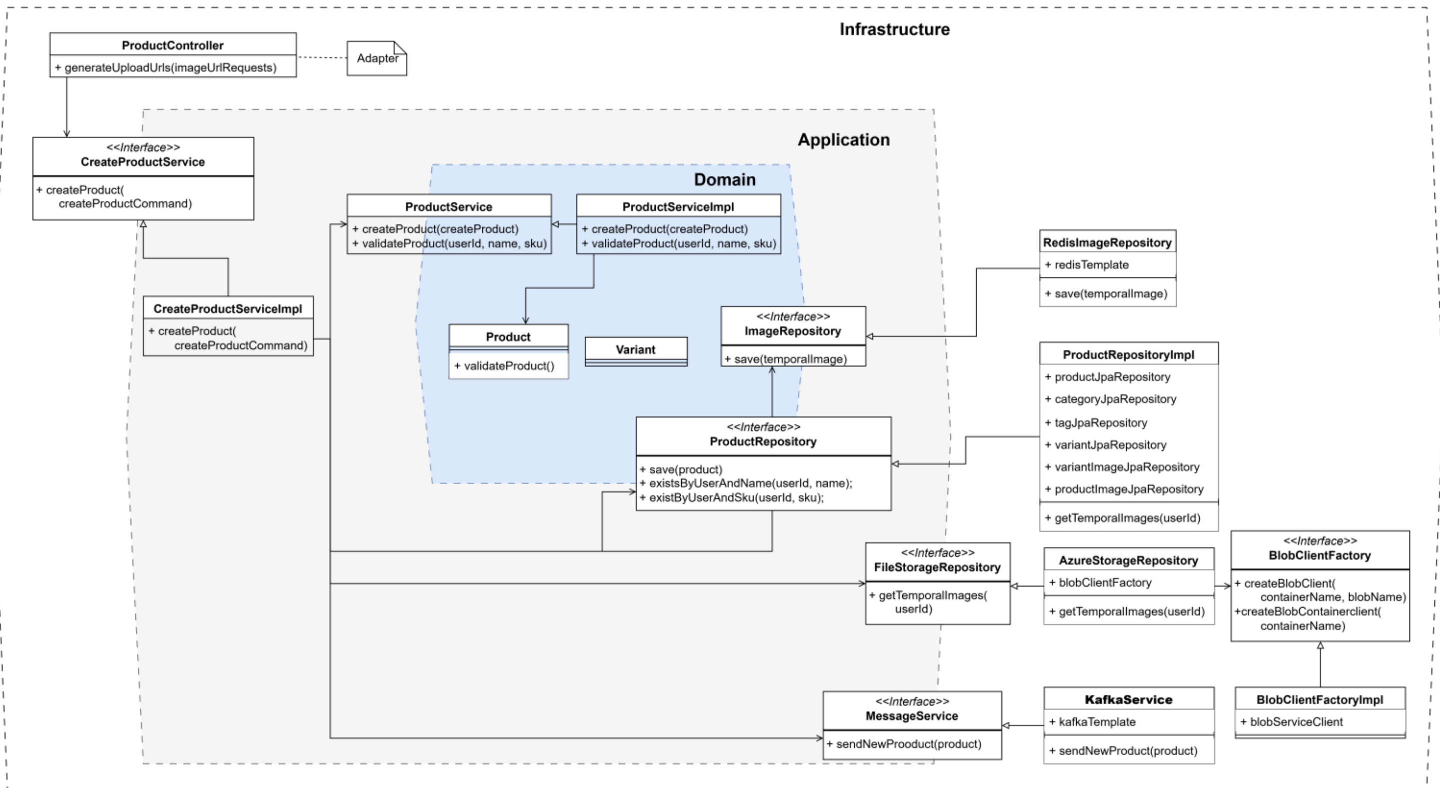

Figure 5.13 Class Diagram of Create Product Use Case

The primary responsibilities of each class are:

Class	Type	Responsibility
ProductController	Driving Adapter	Receives the user's request and transforms the incoming object to a command object.
CreateProductService	Driving Port	Communication channel between the infrastructure and the application layer
CreateProductServiceIml	Application Service	Orchestrator of the use case; it performs the respective calls to create the product.
ProductService	Domain Service Contract	Contract of the domain service, used to follow the Dependency Inversion principle.
ProductServiceImpl	Domain Service Implementation	Performs business rules beyond a single entity. For example, validating that the

		product name does not exist involves multiple entities.
Product	Domain Entity	Enforce business rules and invariants related to the product.
Variant	Domain Entity	Enforce business rules and invariants related to the variant.
ImageRepository	Driven Port	Abstraction to remain the core agnostic from infrastructure details
ProductRepository	Driven Port	Abstraction to remain the core agnostic from infrastructure details
FileStorageRepository	Driven Port	Abstraction to remain the core agnostic from infrastructure details
MessageService	Driven Port	Abstraction to remain the core agnostic from infrastructure details
RedisImageRepository	Driven Adapter	Concrete implementation that retrieves temporal image information from Redis.
ProductRepositoryImpl	Driven Adapter	Concrete implementation that saves the product into PostgreSQL
AzureStorageRepository	Driven Adapter	Concrete implementation that moves the image to the final location using Azure Blob Storage
KafkaService	Driven Adapter	Concrete implementation that sends the product created event to Kafka.
BlobClientFactory	Factory used by Driven Adapter	Contract of the Factory class, used to preserve the Dependency Inversion principle
BlobClientFactoryImpl	Factory used by Driven Adapter	Provides an interface to create clients to communicate with Azure Storage, which delegates the logic and allows a cleaner version of AzureStorageRepository

The library Spring Cloud Contract was used to verify the correctness of the message sent to the broker once the product is created. To achieve this, a schema specifying the contract structure must be created; the image below presents one contract section.

```
shouldSendProductWithVariants.groovy
Contract.make {
    name( name: "shouldSendProductCreatedEventWithVariant")
    label( label: "product_created_event_variant")
    description( description: "Sends a ProductCreated event to Kafka with consumer values")

    input {
        triggeredBy( triggeredBy: "triggerProductWithVariantsCreated()")
    }

    outputMessage {
        sentTo( sentTo: "product")
        headers {
            header( headerKey: "contentType", applicationJson())
        }
        body(
                id: anyPositiveInt(),
                sellerId: "seller-001",
                name: "Coffee Maker",
                description: "High-quality coffee maker",
                status: "AVAILABLE",
```

Figure 5.14 Contract to specify the Message Schema

This contract specifies a method present in a custom base test class, which triggers the message sending. A Spring Cloud Contract class will use this method to automatically create the integration test. To enhance test reliability and reduce the gap between the development and production environments, Spring Cloud Contract can be integrated with Testcontainers, allowing the generated tests to run against real infrastructure components.

```java
BaseTestClass.java
/**
 * Used by shouldSendProductWithVariants.groovy
 */
public void triggerProductWithVariantsCreated() {  1 usage  David Cediel
    var variant = new Variant( id: 10L,  name: "Black Edition",  description: "Premium coffee maker in black color",
            new Stock( amount: 100L), new Money(BigDecimal.valueOf(199.99),  currency: "USD"),  isPrimary: true,
            new Sku("CM-BLK-001"), Set.of(new VariantImage( id: 2L,  name: "coffee-maker-black",  displayOrder: 1,  isPrimary: true,
             url: "https://example.com/images/coffee-maker-black.jpg")));


    Product product = productBuilder
            .id(10L)
            .status(ProductStatus.AVAILABLE)
            .variants(Set.of(variant))
            .tagIds(Set.of(101L, 205L, 307L))
            .build();

    kafkaService.sendNewProduct(product);
}
```

Figure 5.15 Method that will be used by Spring Cloud Contract to send the message

Finally, the framework generates a new class, verifying that the message is correctly published in Kafka.

```java
ContractVerifierTest.java
@Test
public void validate_shouldSendProductCreatedEventWithVariant() throws Exception {
    // when:
        triggerProductWithVariantsCreated();

    // then:
        ContractVerifierMessage response = contractVerifierMessaging.receive( destination: "product",
                contract( testClass: this,  relativePath: "shouldSendProductCreatedEventWithVariant.yml"));
        assertThat(response).isNotNull();

    // and:
        assertThat(response.getHeader( name: "contentType")).isNotNull();
        assertThat(response.getHeader( name: "contentType").toString()).isEqualTo( expected: "application/json");

    // and:
        DocumentContext parsedJson = JsonPath.parse(contractVerifierObjectMapper.writeValueAsString(response.getPayload()));
        assertThatJson(parsedJson).field( o: "['id']").matches( s: "([1-9]\\d*)");
        assertThatJson(parsedJson).field( o: "['sellerId']").isEqualTo( s: "seller-001");
        assertThatJson(parsedJson).field( o: "['name']").isEqualTo( s: "Coffee Maker");
```

Figure 5.16 Autogenerated Test by Spring Cloud Contract

In summary, the process to validate that the message sent is the desired one is the following:

1. Create a contract schema with the desired attributes and their formats.

2. Define a base test class with test containers that boot up the Kafka container when the test runs and trigger the message sending.
3. Run the autogenerated test, verifying that the message structure follows the contract.

If the test is successful, it will generate stubs that the consumer can use to ensure that the message that it expects to receive is the one sent by the producer.

The product microservice concludes with 46 tests, where all integration and end-to-end tests are run using Testcontainers to ensure similarity with the real environment. The technologies used to implement the tests are:

- **Junit5:** It is the foundation for writing and running tests; it provides annotations like @Test to annotate each method. It executes the tests and controls their lifecycle
- **Mockito:** It is used to isolate unit tests from dependencies by mocking their behaviors, allowing us to focus on class functionality without depending on external classes.
- **AssertJ:** It provides a fluent API to perform assertions. It improves the readability and maintainability of the test code base.
- **Spring Test Utilities:** It provides an easy way to upload the required beans to run a test, for example, @WebMvcTest will boot up the necessary context to receive an HTTP request.

Name	Container ID	Image	Port(s)
vigorous_wright	e50967ba5407	apache/kafka:4.0.0	33700:9092
testcontainers-ryuk-68689	e787efa314f9	testcontainers/ryuk:0.11.0	
interesting_lewin	59723b58b460	postgres:16-alpine	33702:5432
flamboyant_buck	2b18d8beb3ea	azure-storage/azurite:lates	
brave_kirch	c8a486f93ab3	redis:7-alpine	

Figure 5.17 Containers automatically boot up used by the tests

5.5 Product Query

This section describes the operations created for the Product Query microservice.

5.5.1 Create Product

The Product Query service receives the product creation event and replicates the product entity into a query-optimized format, following the CQRS pattern.

The Figure 5.13 illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

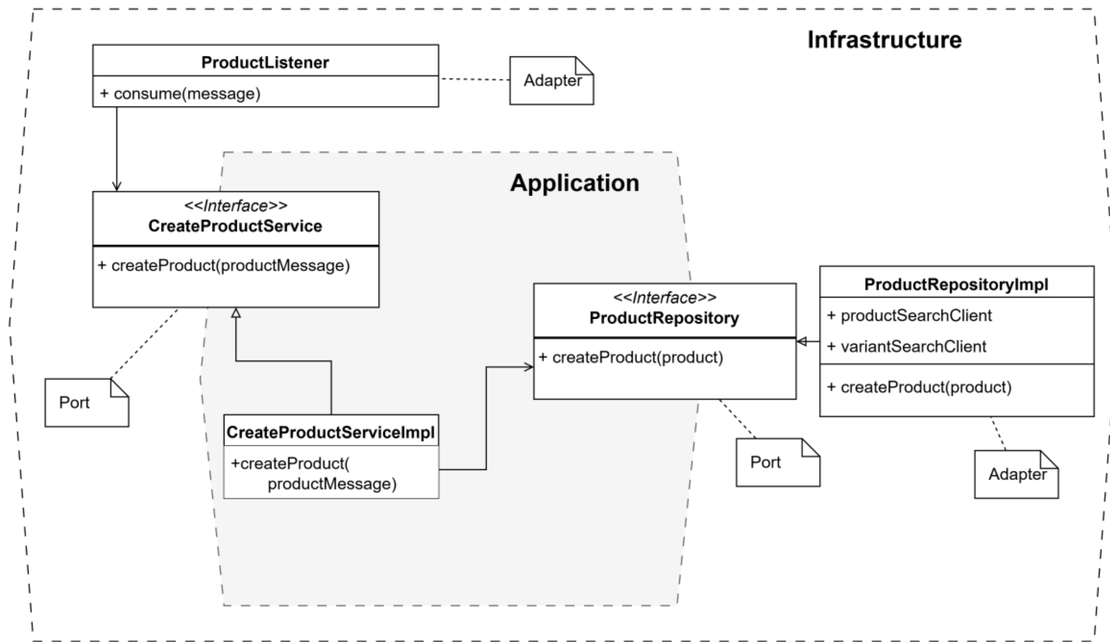

Figure 5.18 Class Diagram of Create Product Query

The primary responsibilities of each class are:

Class	Type	Responsibility
ProductListener	Driving Adapter	Receives Kafka messages and calls the corresponding service handler.
CreateProductService	Driving Port	Defines the contract of the application core.
CreateProductServiceImpl	Application Service	Orchestrator that performs the required calls to create the product.
ProductRepository	Driven Port	Isolates the application core from infrastructure details
ProductRepositoryImpl	Driven Adapter	It saves the product in Azure Search

This process was tested using Spring Cloud Contract, which validates that the product listener expects the message sent by the producer. To achieve this, a new test class must be created specifying the location of the producer contract.

```
ProductListenerTest.java
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.NONE,   David Cediel
        classes = {ContainersConfig.class, ProductListenerTest.TestConfig.class, ProductqueryApplication.class})
@AutoConfigureStubRunner(ids = "com.cordilleracoffee:product:+:8081",
        repositoryRoot = "git://https://github.com/davidcediel12/coffee-contracts.git",
        stubsMode = StubRunnerProperties.StubsMode.REMOTE,
        properties = {"git.branch=main"})
class ProductListenerTest {
```

Figure 5.19 Specify Contract location for contract testing

When the contract location is specified, which in this case is located in a remote Git repository and is defined using the *repositoryRoot* and *ids* parameters, a specific contract can be triggered by referencing its associated label. Once the message is produced, the consumer can validate the expected behavior.

```
ProductListenerTest.java
@Test   David Cediel *
void shouldReceiveProductCorrectly() {

    Awaitility.await()
            .atMost( timeout: 30, TimeUnit.SECONDS)
            .pollInterval(Duration.ofSeconds(2))
            .until(() -> isKafkaReady( topicName: "product"));

    trigger.trigger( labelName: "product_created_event_variant");

    doNothing().when(createProductService).createProduct(any());

    Awaitility.await().atMost(Duration.ofSeconds(30))
            .pollInterval(Duration.ofSeconds(5))
            .untilAsserted(() ->
                    verify(createProductService)
                            .createProduct(argThat(this::verifyProductAttributesMatchesWithContract))
            );

}
```

Figure 5.20 Custom Consumer test using Spring Cloud Contract

5.5.2 Get Product Summary

After the cart payment is completed, to follow the Event-Carried State principle of the Event-Driven architecture, the checkout obtains a snapshot of the purchased products to send in the *OrderPlaced* domain event, providing it with the system's current state, avoiding consistency problems. The figure below presents the implemented endpoint contract.

POST /products/summary Get the summary of product and variants

Parameters

Try it out

No parameters

Request body application/json

Products and variants to retrieve

Example Value | Schema

```
{
  "productDetails": [
    {
      "productId": 5,
      "variantId": 10
    }
  ]
}
```

Responses

Code	Description	Links
200	Items retrieved	No links

Media type

application/json

Controls Accept header.

Example Value | Schema

```
[
  {
    "productId": 5,
    "variantId": 10,
    "name": "coffe machine",
    "description": "ultimate coffe machine",
    "sku": "MA-01-02",
    "primaryImageUrl": "https://cordillera.blob/image25",
    "isVariant": true
  }
]
```

Figure 5.21 Retrieve Products Summary OpenAPI Schema

The Figure 5.13 illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

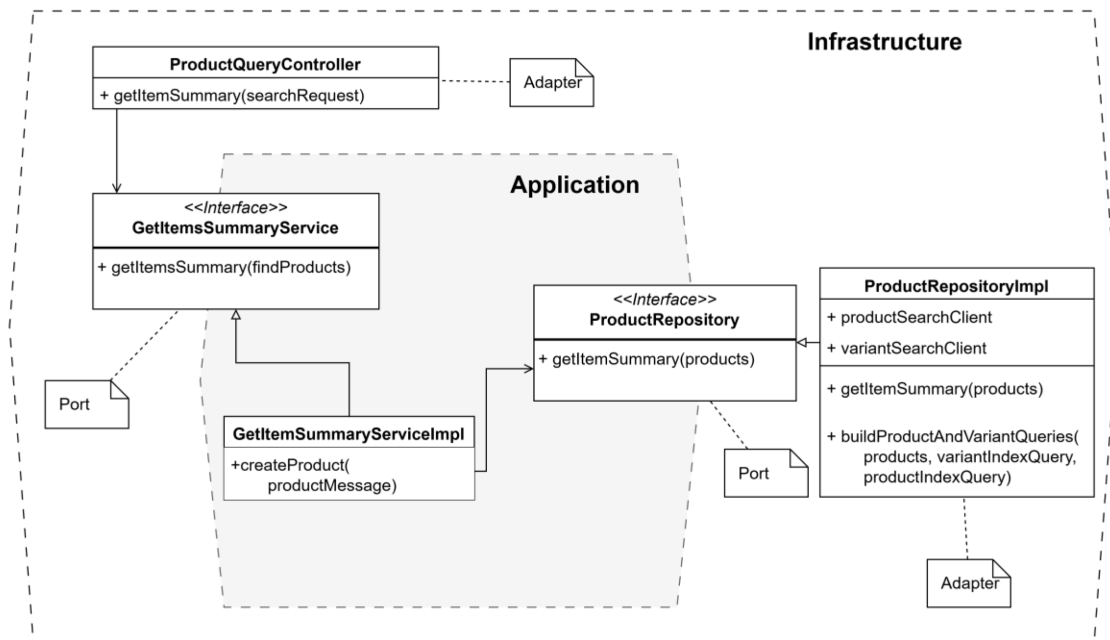

Figure 5.22 Class Diagram of Retrieve Products Summary

The primary responsibilities of each class are:

Class	Type	Responsibility
ProductQueryController	Driving Adapter	Receives the HTTP request and transforms it into a command object.
GetItemSummaryService	Driving Port	Defines the contract of the application core.
GetItemSummaryServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to retrieve the product
ProductRepository	Driven Port	Isolates the application core from infrastructure details
ProductRepositoryImpl	Driven Adapter	It retrieves the products in Azure Search

5.6 Pricing and Promotions

This section describes the process of developing the Pricing and Promotions bounded context.

5.6.1 Validate Prices

The pricing microservice receives the item identifications, the final base price, and an optional discount ID parameter. With this information, the service checks if an active promotion makes the final price the same as the one received. It returns a successful HTTP status code if all the item prices are valid. Otherwise, it returns an error with a response indicating the incorrect item and the expected price according to the current promotions. The figure below presents the implemented endpoint contract.

POST **/validate** Validate that items has a correct final price

It validates that the discount for each item exists and its valild.

1. If a discount ID is provided, it will only look up for this discount.
2. If the discount ID is not provided, it will iterate over all valid discounts to check if the final price is valid. If it does not found a discount, it will return the most relevant discount for the item that is active.

Parameters Try it out

No parameters

Request body application/json

Object to create a new product

Example Value | Schema


```json
{
  "userId": "user-123",
  "currency": "EUR",
  "products": [
    {
      "productId": 20,
      "variantId": 21,
      "categoryId": 5,
      "basePrice": 19.99,
      "finalPrice": 15.99,
      "discountId": "7ff595f5-1176-4a4a-a6bc-195808f78034",
      "couponCode": "CODE20"
    }
  ]
}
```


Responses

Code	Description	Links
200	Successful operation Media type	*No links*

Figure 5.23 Validate Items Price OpenAPI Schema

The Figure 5.24 illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

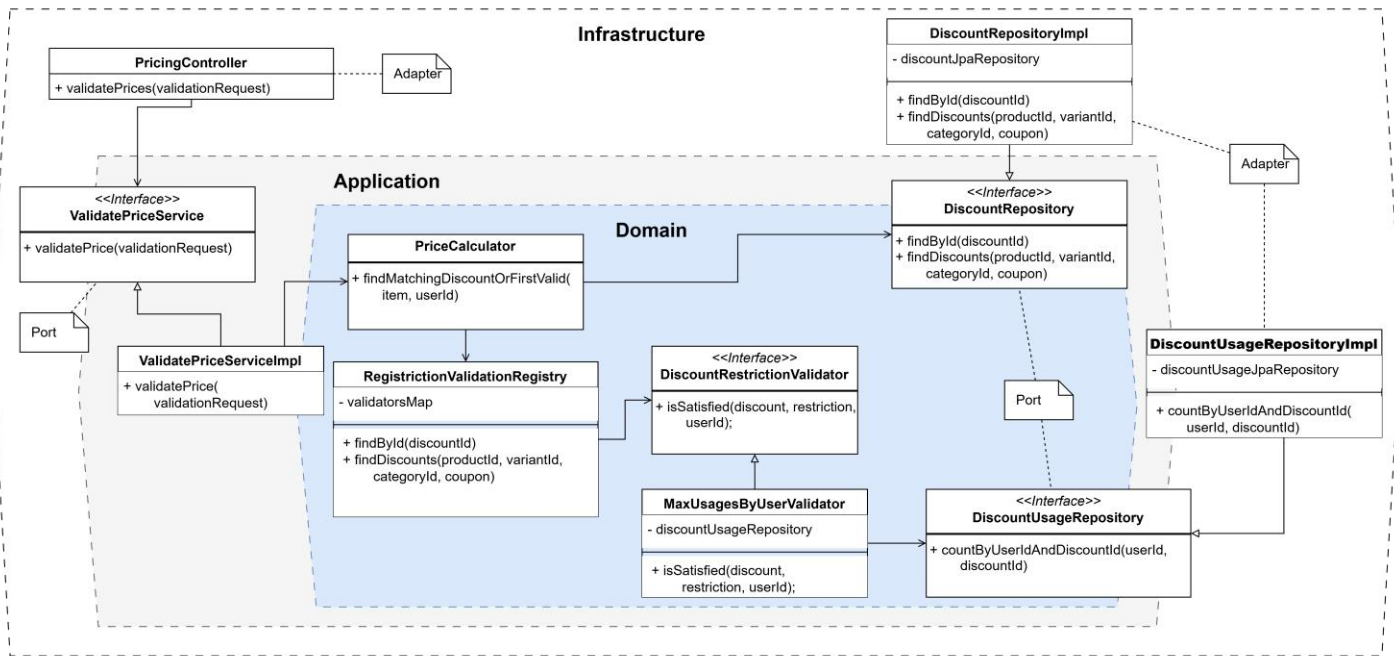

Figure 5.24 Class Diagram of Validate Items Price

The primary responsibilities of each class are:

Class	Type	Responsibility
PricingController	Driving Adapter	Receives the HTTP request and transforms it into a command object.
ValidatePriceService	Driving Port	Defines the contract of the application core.
ValidatePriceServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to validate the items' price
PriceCalculator	Domain Service	Finds the correct discount according to the given price, it needs to use multiple Discount domain entities.
DiscountUsageRepository	Driven Port	Isolates the application core from infrastructure details

DiscountRepository	Driven Port	Isolates the application core from infrastructure details
DiscountUsageRepositoryImpl	Driven Adapter	It retrieves the discount usages from PostgreSQL
DiscountRepositoryImpl	Driven Adapter	Retrieves the discounts from PostgreSQL

The strategy pattern to validate if the discount restrictions are satisfied was implemented with the classes RestrictionValidationRegistry**,** DiscountRestrictionValidator, and MaxUsagesByUserValidator**.** The pattern follows SOLID's Open/Closed principle, which states that entities should be open for extension but closed for modification. Therefore, if another restriction type is added, a new validator class that implements the DiscountRestrictionValidator interface can be added without modifying the existing code.

A custom annotation can be created to implement this pattern. Using the dependency injection paradigm of the Spring Framework, the new validators can be registered automatically in the RestrictionValidationRegistry.

```
SupportedRestriction.java
@Target(ElementType.TYPE)   5 usages   Dav
@Retention(RetentionPolicy.RUNTIME)
public @interface SupportedRestriction {
    RestrictionType value();   David Cedi
}
```

Figure 5.25 Custom Annotation to specify the Discount Restriction

After the annotation is created, custom validators can use it to specify which restriction type they can handle.

```
MaxUsagesByUserValidator.java
@Component   David Cediel *
@SupportedRestriction(RestrictionType.MAX_USAGES_BY_USER)
@RequiredArgsConstructor
public class MaxUsagesByUserValidator implements DiscountRestrictionValidator {
```

Figure 5.26 Specific Validator Using the Custom Annotation

5.7 Shopping Cart

This section describes the process of developing the Shopping Cart bounded context.

5.7.1 Validate Cart

Since the service's bounded context owns the shopping cart, this process will validate it and change its status to processed. If the cart is valid, it will return a snapshot so the client can have a materialized view of the current cart state. The figure below presents the implemented endpoint contract.

POST /{cartId}/checkout

Parameters

Try it out

Name	Description
cartId * required string($uuid) *(path)*	Shopping cart id *Example* : 3fa85f64-5717-4562-b3fc-2c963f66afa6 3fa85f64-5717-4562-b3fc-2c963f66afa6
App-User-ID * required string *(header)*	user that owns the shopping cart *Example* : user-123 user-123

Responses

Code	Description	Links
200	Successful checkout validation	*No links*

Media type

application/json

Controls Accept header.

Example Value | Schema


```json
{
  "cartId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "userId": "user-123",
  "status": "ACTIVE",
  "currency": "USD",
  "price": 55.58,
  "createdAt": "2025-04-25T15:30:00Z",
  "items": [
    {
      "itemId": "3fa85f64-5717-4562-b3fc-2c963f66afa7",
      "productId": 20,
      "variantId": 21,
      "categoryId": 5,
      "quantity": 2,
      "unitPrice": 10.99,
      "finalBasePrice": 15.99,
      "discount": {
        "discountId": "7ff595f5-1176-4a4a-a6bc-195808f78034",
        "couponCode": "SPRING20",
        "discountType": "FIXED",
        "discountAmount": 5
      }
    }
  ]
}
```


Figure 5.27 Checkout Shopping Cart OpenAPI Schema

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

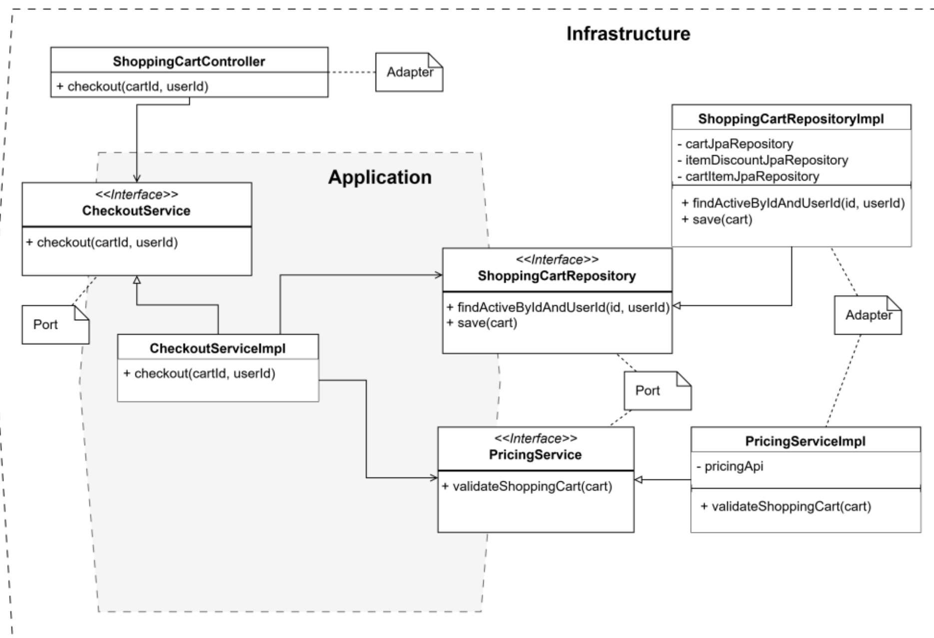

Figure 5.28 Class Diagram of Validate Shopping Cart

The primary responsibilities of each class are:

Class	Type	Responsibility
ShoppingCartController	Driving Adapter	Receives the HTTP request and transforms it into a command object.
CheckoutService	Driving Port	Defines the contract of the application core.
CheckoutServiceImpl	Application Service	Orchestrator that will perform the required callscheckout the shopping cart.
ShoppingCartRepository	Driven Port	Isolates the application core from data store details

PricingService	Driven Port	Isolates the application core from communication with third party services details
ShoppingCartRepositoryImpl	Driven Adapter	It retrieves the shopping cart from PostgreSQL
PricingServiceImpl	Driven Adapter	Communicates with pricing microservices to validate the item prices

The PricingServiceImpl injects the pricing API, which is the class that manages communication with the pricing microservice and performs the request to validate the items. The injected class is generated through the OpenAPI Generator plugin, following the **API-first paradigm** described earlier.

build.gradle

```
compileJava.dependsOn tasks.openApiGenerate

openApiGenerate {
    generatorName.set('java')
    configOptions.set([
            library: 'restclient',
            openApiNullable: 'false'
    ])
    inputSpec = "$rootDir/src/main/resources/contracts/client/openapi-pricing.yaml".toString()
    ignoreFileOverride = ".openapi-generator-java-sources.ignore"
    invokerPackage = 'com.cordilleracoffee.infrastructure.api.client.pricing'
    modelPackage = 'com.cordilleracoffee.infrastructure.api.client.pricing.model'
    apiPackage = 'com.cordilleracoffee.infrastructure.api.client.pricing'

}
```

Figure 5.29 Configuration of OpenAPI Generator Plugin

After configuring the plugin, it generates the necessary classes to perform the pricing microservice requests stated in the OpenAPI specification file.

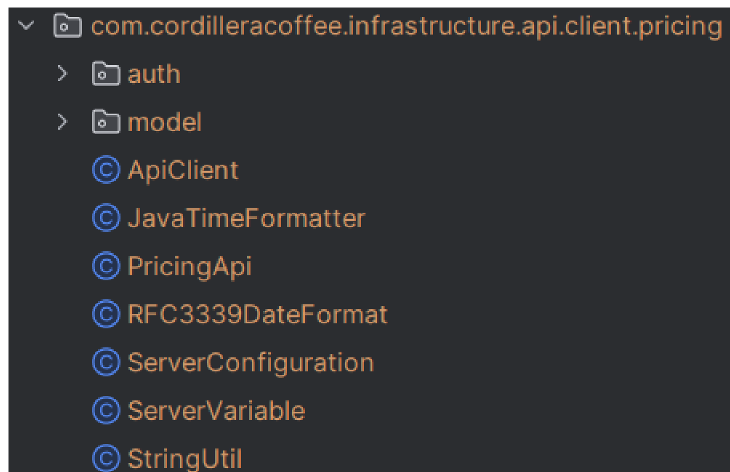

Figure 5.30 Generated Classes from OpenAPI Generator

5.8 Payment Adapter

This section describes the development of the payment adapter microservice, which works as an **anti-corruption layer** between the core application and the external payment system. While it is not a bounded context, it isolates the platform from the Payment generic subdomain, which is managed by a third-party service.

5.8.1 Pay Shopping Cart

The cart service receives the payment information and builds the request object that the payment gateway needs.

To maximize the success rate, a retry with an exponential back-off pattern is made if the payment gateway returns an error, which, according to Amazon Web Services (AWS, n.d.), “Exponential backoff is a technique where operations are retried by increasing wait times for a specified number of retry attempts.”. The call to the payment gateway is made at most three times.

```
PerformPaymentServiceImpl.java
Mono<PaymentGatewayResponse> paymentCall = callPaymentGateway(request, idempotencyKey)
        .retryWhen(Retry.backoff( maxAttempts: 3, Duration.ofMillis(500))
                .filter(PaymentGatewayFailureException.class::isInstance));
```

Figure 5.31 Retry with backoff to call the Payment Gateway

Furthermore, if the payment gateway is constantly failing and it is not a temporary error, the circuit-breaker pattern is implemented to avoid calling the unavailable service and to fail fast.

```java
// PerformPaymentServiceImpl.java
return reactiveCircuitBreaker.run(paymentCall, this::openCircuit);
```

Figure 5.32 Usage of the Circuit Breaker to call the Payment Gateway

Finally, the circuit breaker parameters, such as failure rate threshold, duration in open state, etc., can be customized in the properties file, and Spring Boot will create the corresponding implementation.

```yaml
# application.yaml
resilience4j:
  circuitbreaker:
    instances:
      paymentGateway:
        slidingWindowSize: 10
        minimumNumberOfCalls: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 10s
```

Figure 5.33 Customization of the Circuit Breaker

The figure below presents the implemented endpoint contract.

POST / Performs the payment of a shopping cart

Parameters — Try it out

Name	Description
Idempotency-Key * required string($uuid) (header)	key that identifies the payment, used to avoid duplicated payment attempts 3fa85f64-5717-4562-b3fc-2c963f66afa6

Request body — application/json

Payment Request

Example Value | Schema

```
{
  "paymentMethod": "CREDIT_CARD",
  "cardNumber": "4111111111111111",
  "expirationDate": "10/28",
  "cardHolder": "Jhon Doe",
  "cvv": "123",
  "amount": 59.99,
  "currency": "USD",
  "cartId": "3fa85f64-5717-4562-b3fc-2c963f66afa6"
}
```

Responses

Code	Description	Links
200	Reservation was created	No links

Media type: application/json

Controls Accept header.

Example Value | Schema

```
{
  "status": "success",
  "transactionId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "cartId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "amount": 59.99,
  "currency": "USD",
  "message": "Payment processed successfully"
}
```

Figure 5.34 Pay Shopping Cart OpenAPI Schema

Since this microservice does not represent any bounded context, it lacks a domain; therefore, a layered architecture, rather than a hexagonal architecture, was chosen for its development. The figure below illustrates the classes used to implement the use case and the layer to which they belong according to the layered architecture.

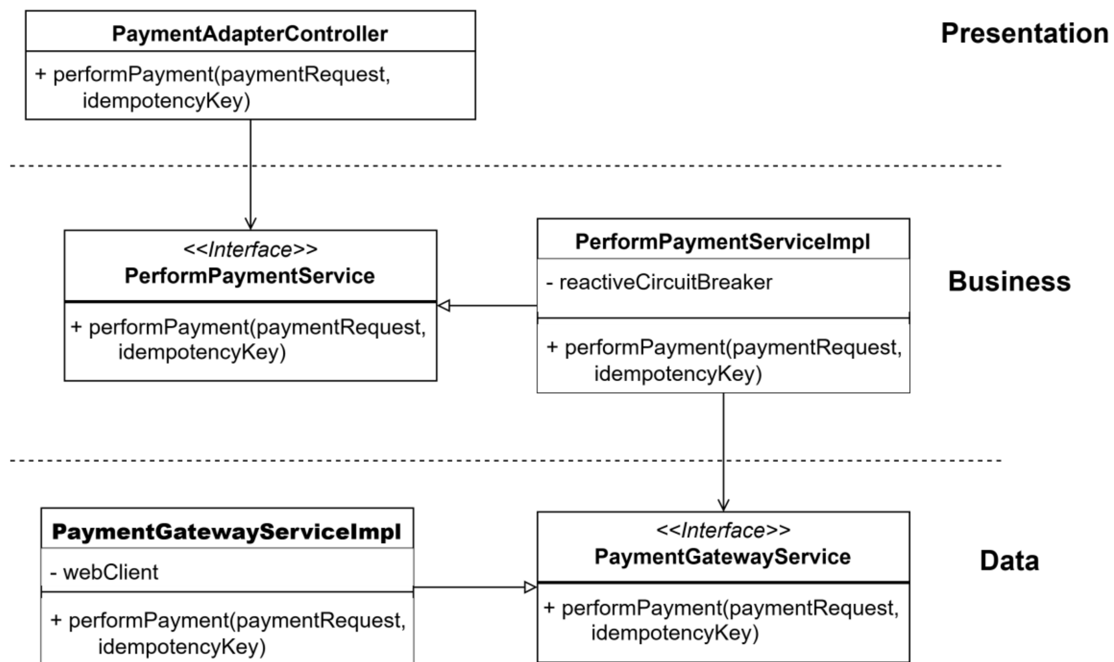

Figure 5.35 Class Diagram of Cart Payment

The primary responsibilities of each class are:

- **PaymentAdapterController:** It receives the request, validates that the fields contain the correct value, and prepares the objects needed by the business layer.
- **PerformPaymentService:** Contract that defines how the business layer works.
- **PerformPaymentServiceImpl:** This is the orchestrator of the use case. It contains the business logic, which, in this case, is the back-off retries and the circuit breaker implementation.
- **PaymentGatewayService:** Contract defined by the business layer that isolates the application core using dependency inversion.
- **PaymentGatewayServiceImpl:** Concrete implementation of the PaymentGatewayService contract, which uses opinionated technologies to call the payment gateway, in this case, WebClient.

5.9 Inventory

This section describes the process of developing the Inventory Management bounded context.

5.9.1 Create Inventory

The inventory service receives the event generated by the Product Management and creates a new record for each product and variant with its respective stock.

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

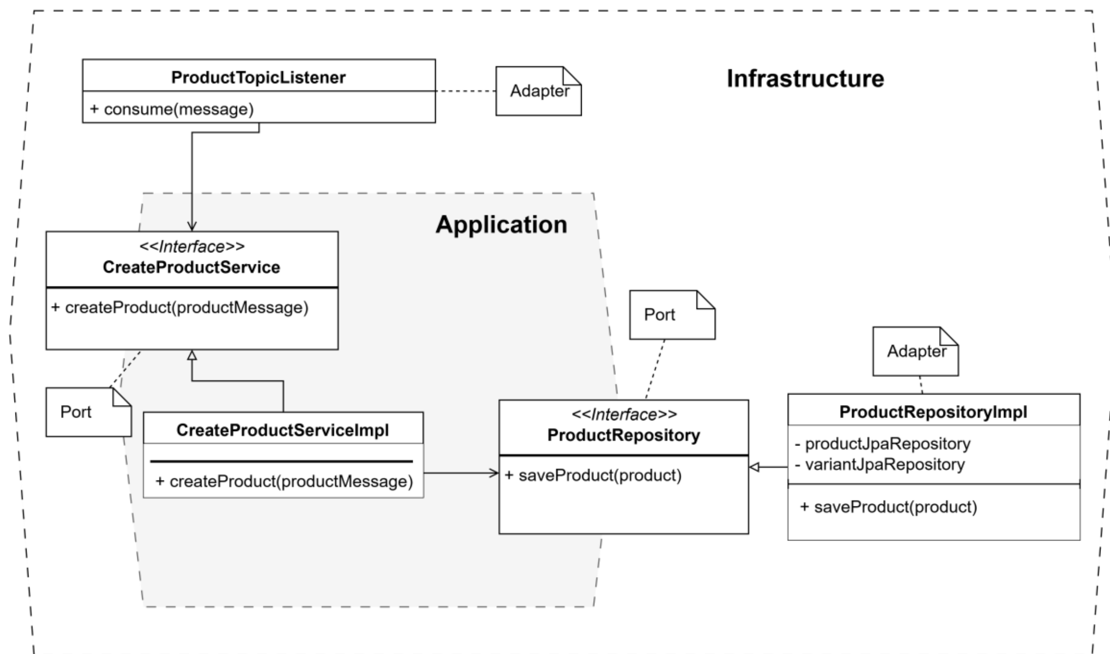

Figure 5.36 Class Diagram of Create Inventory

The primary responsibilities of each class are:

Class	Type	Responsibility
ProductTopicListener	Driving Adapter	Consumes the Kafka message, deserializes it, and adjusts the data to send it to the core.
CreateProductService	Driving Port	Defines the contract of the application core.
CreateProductServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to create the inventory.
ProductRepository	Driven Port	Isolates the application core from infrastructure details
ProductRepositoryImpl	Driven Adapter	Saves the inventory in PostgreSQL

5.9.2 Reserve Inventory

Once the shopping cart is validated, the items' inventory must be reserved. Since this microservice is responsible for the inventory domain, it handles reservations if there is enough stock. Otherwise, an error will be returned indicating that the items cannot be reserved due to insufficient stock. The figure below presents the implemented endpoint contract.

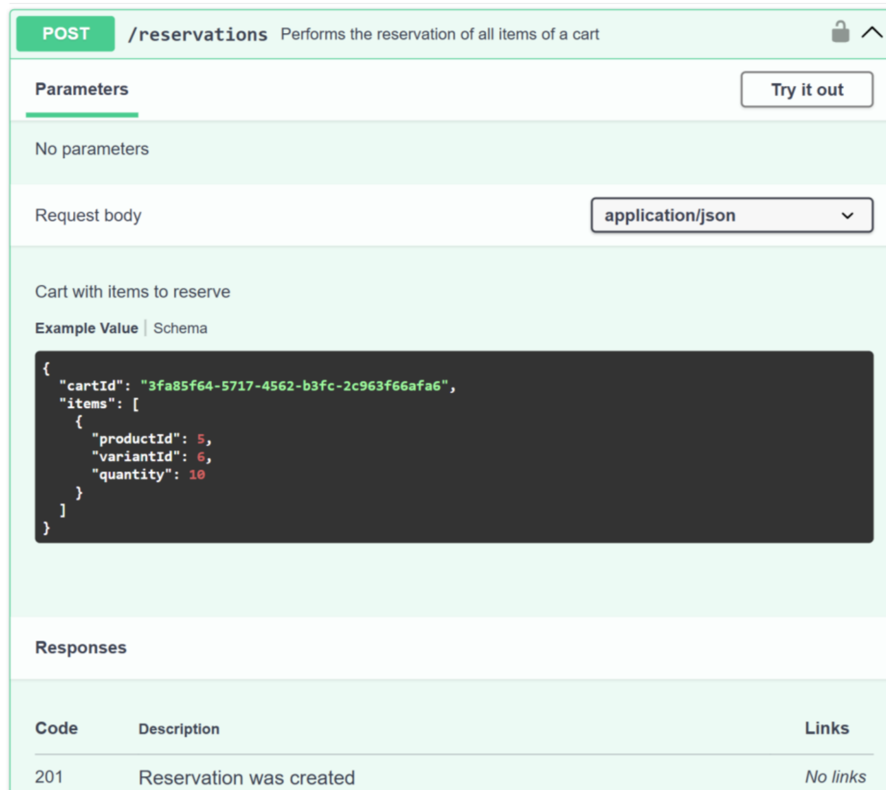

Figure 5.37 Reserve Inventory OpenAPI Schema

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

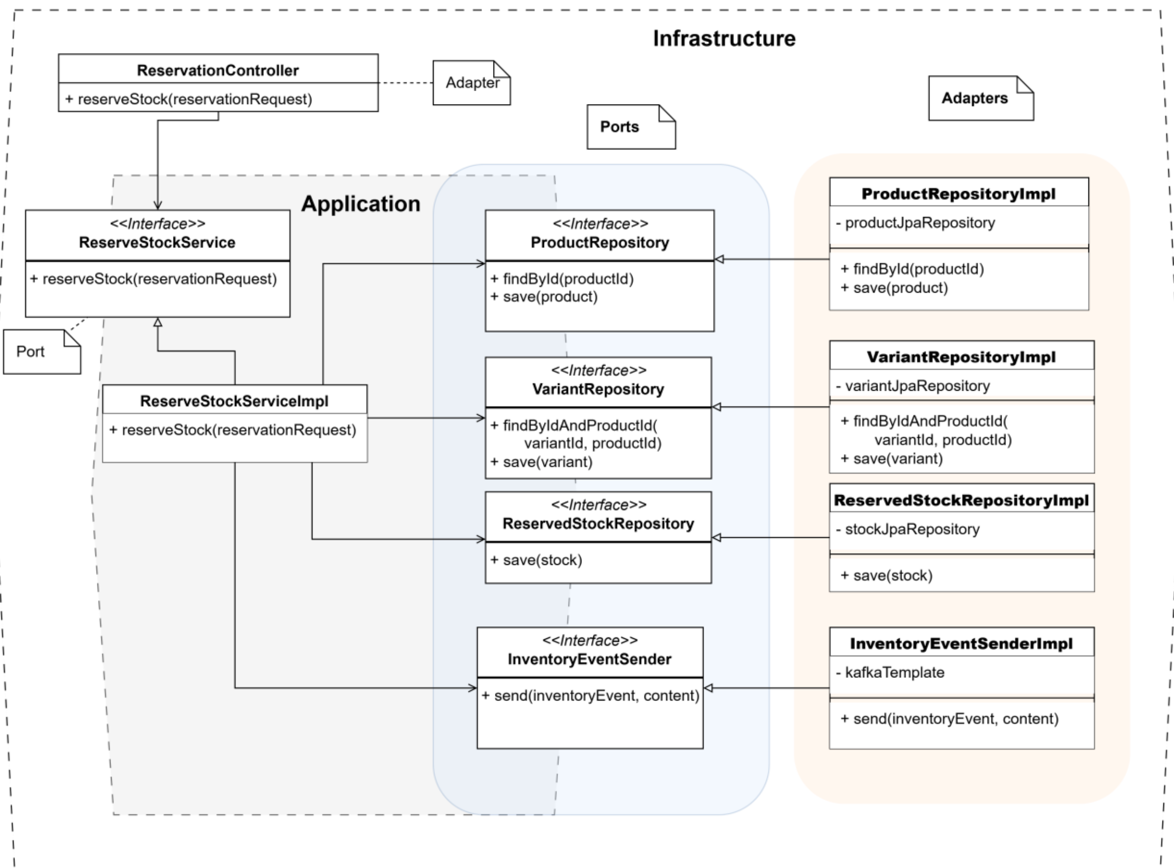

Figure 5.38 Class Diagram of Inventory Reservation

The primary responsibilities of each class are:

Class	Type	Responsibility
ReservationController	Driving Adapter	Receives the HTTP request and prepares the data for the application core.
ReserveStockService	Driving Port	Defines the contract of the application core.
ReserveStockServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to reserve the stock.
ProductRepository	Driven Port	Isolates the application core from infrastructure details
VariantRepository	Driven Port	Isolates the application core from infrastructure details

ReservedStockRepository	Driven Port	Isolates the application core from infrastructure details
InventoryEventSender	Driven Port	Isolates the application core from infrastructure details
ProductRepositoryImpl	Driven Adapter	Retrieves the product inventory from PostgreSQL
VariantRepositoryImpl	Driven Adapter	Retrieves the variant inventory from PostgreSQL
ReservedStockRepositoryImpl	Driven Adapter	Creates a new stock reservation in PostgreSQL
InventoryEventSenderImpl	Driven Adapter	Sends the event to notify the new reservation

5.10 Order Management

This section describes the process of developing the Order Management bounded context.

5.10.1 Create Order

Once the payment is made, the checkout service will publish an event that the order service receives and creates a new order.

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

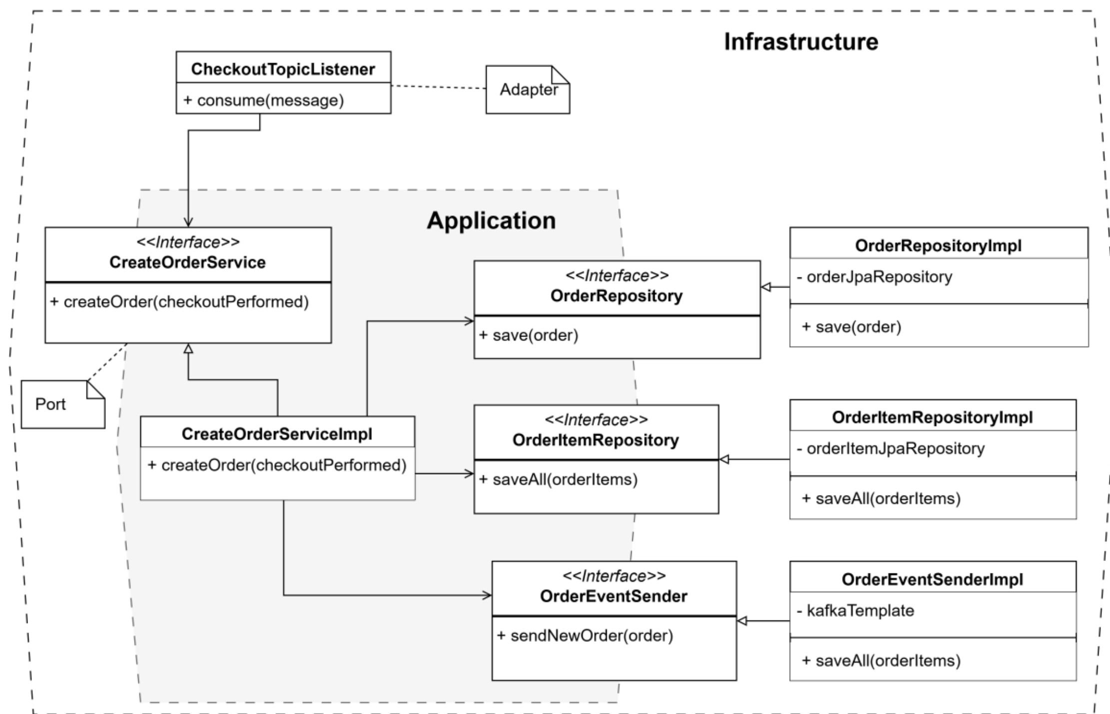

Figure 5.39 Class Diagram of Save Order

The primary responsibilities of each class are:

Class	Type	Responsibility
CheckoutTopicListener	Driving Adapter	Consumes the Kafka message, deserializes it, and adjusts the data to send it to the core.
CreateOrderService	Driving Port	Defines the contract of the application core.
CreateOrderServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to create the order.
OrderRepository	Driven Port	Isolates the application core from data store details
OrderItemRepository	Driven Port	Isolates the application core from data store details
OrderEventSender	Driven Port	Isolates the application core from message broker details

OrderRepositoryImpl	Driven Adapter	Saves the order in PostgreSQL using Spring Data JPA
OrderItemRepositoryImpl	Driven Adapter	Saves the items in PostgreSQL using Spring Data JPA
OrderEventSender	Driven Adapter	Sends the order created event through Kafka.

5.11 Order Query

This section describes the operations created for the Order Query microservice. Although it does not represent a bounded context, it reflects the Order domain entity in a read-optimized form, following the CQRS pattern.

5.11.1 Create Order

The Order Query service receives the event emitted by the Order Management service and replicates the information in a denormalized form.

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

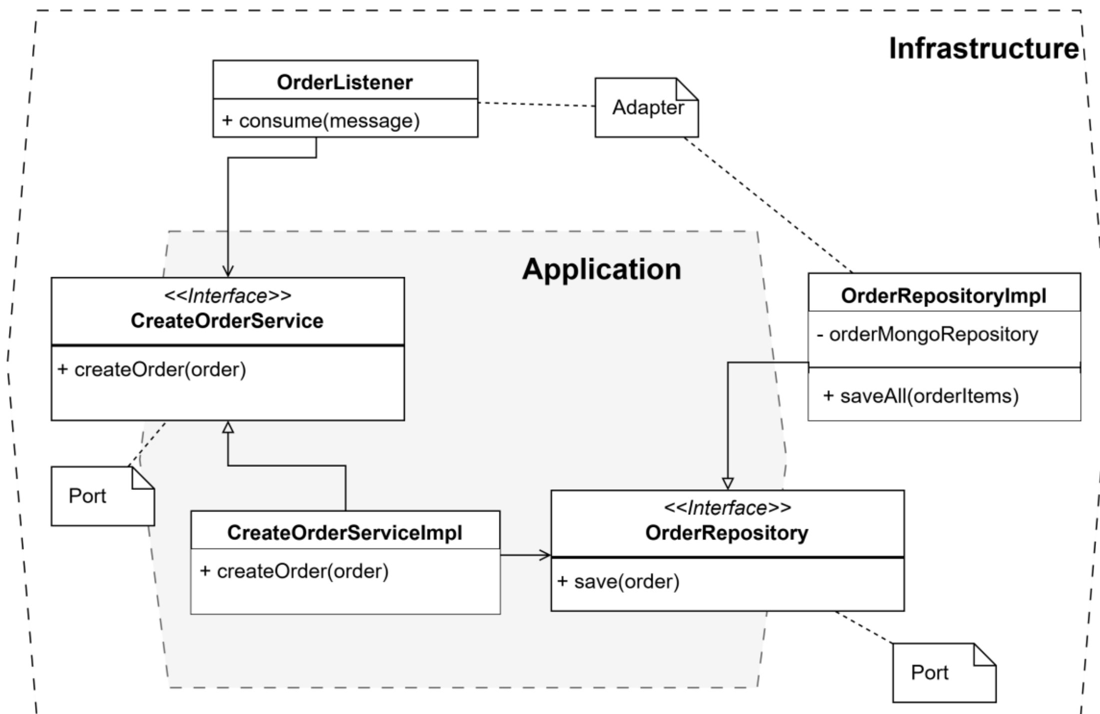

Figure 5.40 Class Diagram of Creating Order for Queries

The primary responsibilities of each class are:

Class	Type	Responsibility
OrderListener	Driving Adapter	Consumes the Kafka message, deserializes it, and adjusts the data to send it to the core.
CreateOrderService	Driving Port	Defines the contract of the application core.
CreateOrderServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to create the order.
OrderRepository	Driven Port	Isolates the application core from data store details
OrderItemRepositoryImpl	Driven Adapter	Saves the order in MongoDB using Spring Data Mongo

5.12 Platform Notifications

This section describes the development of the Platform Notifications microservice, which is part of the Notifications bounded context.

5.12.1 Notify when an Order is Created

Once the payment is completed, the checkout service emits a message notifying it; this message is received by the Platform Notifications microservice, which sends the information to the user's active connection through server-side events, which was created by the frontend when the user logs in and is related to the user performing the purchase.

The notifications are not filtered for development purposes and will be received by anyone connected to the application through the defined endpoint.

The figure below presents the implemented endpoint contract.

POST /register
Register to the platform notifications, it will receive any app notification produced

Parameters

Try it out

Name	Description
App-User-ID * required string (header)	User that sends the request **It will be used later to filter notifications** App-User-ID

Responses

Code	Description	Links
200	Connected to receive messages	*No links*

Media type
text/event-stream
Controls Accept header.

Example Value | Schema

```
{
  "event": "New Notification!",
  "data": {
    "foo": "bar",
    "jhon": "doe"
  }
}
```

Figure 5.41 Register for Notifications OpenAPI Schema

Since this microservice is not part of the product purchase core, the design decision was to create the minimum code needed to make the service run. The figure below illustrates the classes used to implement the use case.

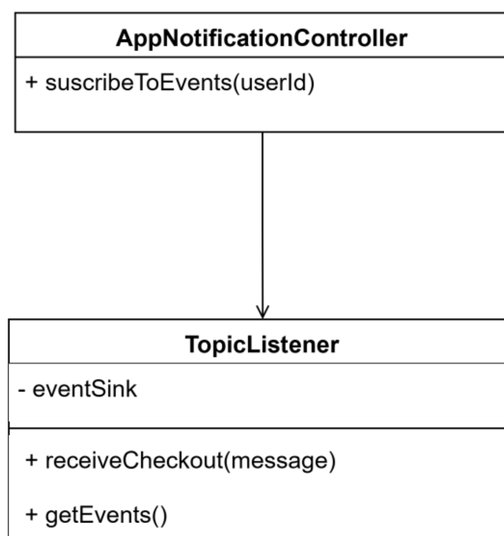

Figure 5.42 Class Diagram of Receive Notifications

The primary responsibilities of each class are:

- **AppNotificationController:** It receives the request to register for notifications. Implementing server-side events is straightforward using Project Reactor and returns the media type text/event-stream, which maintains an open unidirectional connection between the client and the server.

```
AppNotificationController.java
@PostMapping(value = "/register", produces = MediaType.TEXT_EVENT_STREAM_VALUE)  David Cediel
public Flux<ServerSentEvent<String>> suscribeToEvents(@RequestHeader("App-User-ID") String userId){
```

Figure 5.43 Use of text/event-stream and Project Reactor's Flux

According to the Project Reactor Reference Guide (2025), a Flux "represents an asynchronous sequence of 0 to N emitted items." This makes Flux the perfect tool for pushing new notifications when they arrive. The Flux population with new values is delegated to the TopicListener class.

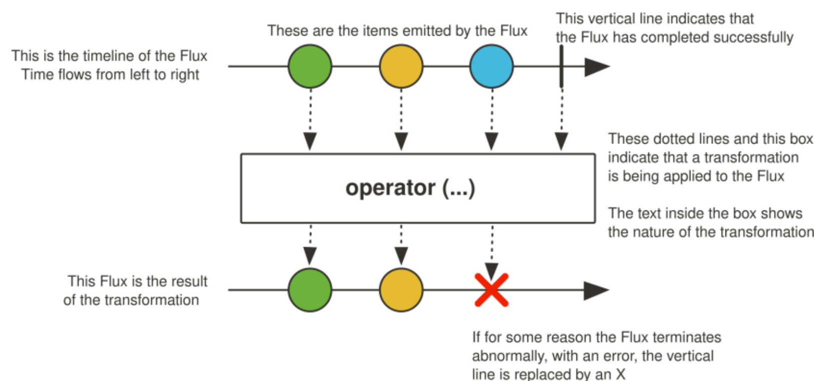

Figure 5.44 Flux Structure. Adapted from ***Reactor Reference Guide*** *(Project Reactor, 2025)*

- **Topic Listener:** It is in charge of receiving new events and populating the notifications. A Project Reactor Sink is used to achieve this, which is the way to push new values into the flux.

```
TopicListener.java
Sinks.EmitResult emitResult = eventSink.tryEmitNext(checkoutMessage.content().toString());
```

Figure 5.45 Push New Notifications Using Sink

When the controller asks for the notification, the Sink is transformed into a flux.

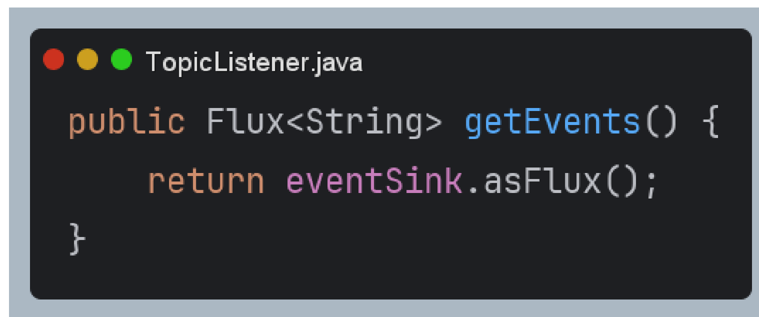

Figure 5.46 Transforming the Sink into a Flux

5.13 Checkout

This section describes the process of developing the Checkout service, which is part of the Checkout & Order Management bounded context.

5.13.1 Perform Checkout

The checkout service is an orchestrator that calls the required microservices to perform the payment correctly and create the order. It validates the shopping cart, reserves the inventory, performs the payment, and publishes the event to make an order.

To maintain the system's consistency in a distributed environment, the checkout microservice implements the SAGA pattern to coordinate transactions across multiple microservices. In a failure, it applies compensating transactions to undo previous steps and maintain data integrity. Two key scenarios are:

1. **Inventory reservation fails or price mismatch:** If the inventory service cannot reserve stock or if there is a discrepancy between the request price and the cart price, it emits an event to compensate the transaction so that the shopping cart can change its status to ACTIVE again, allowing the user to modify and resubmit the order.
2. **Payment fails:** If the payment service is down or declines the transaction, it emits an event to release the cart, which changes its status to ACTIVE, and the inventory reservation is released.

The image below presents the implemented endpoint.

POST / Performs the checkout of a shopping cart

Parameters

Try it out

Name	Description
App-User-ID * required string (header)	user that is purchasing user-123

Request body — application/json

Payment Request

Example Value | Schema

```
{
  "cartId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "paymentDetails": {
    "paymentMethod": "CREDIT_CARD",
    "cardNumber": "4111111111111111",
    "expirationDate": "10/28",
    "cardHolder": "Jhon Doe",
    "cvv": "123",
    "amount": 59.99,
    "currency": "USD"
  }
}
```

Responses

Code	Description	Links
200	Checkout performed	No links

Media type: application/json

Controls Accept header.

Example Value | Schema

```
{
  "orderId": "3fa85f64-5717-4562-b3fc-2c963f66afa6",
  "message": "Checkout was successful"
}
```

Figure 5.47 Perform Checkout OpenAPI Schema

The figure below illustrates the classes used to implement the use case, with the layer where they belong according to the hexagonal architecture.

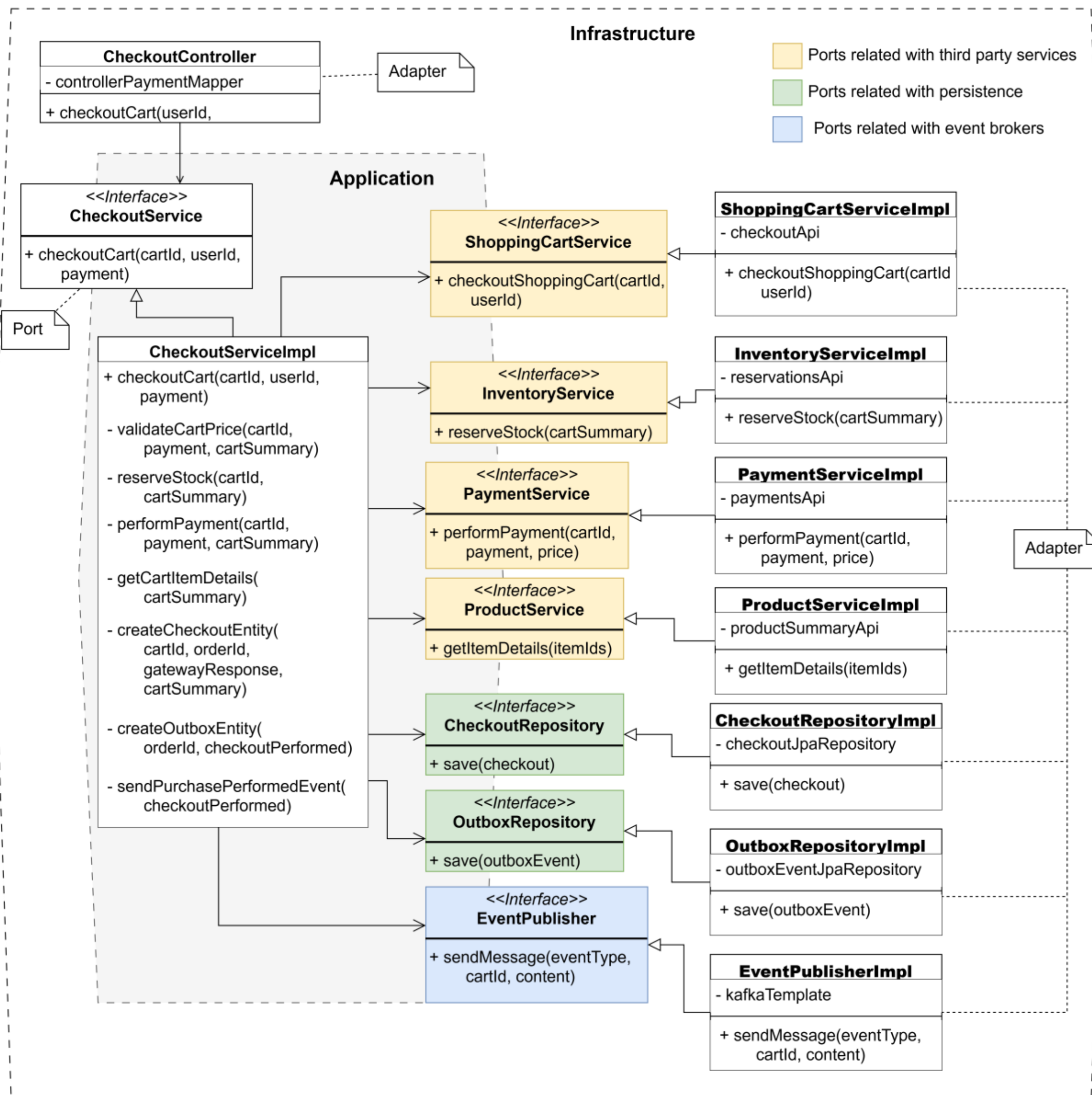

Figure 5.48 Class Diagram of Purchase Cart

Class	Type	Responsibility
CheckoutController	Driving Adapter	Consumes the Kafka message, deserializes it, and adjusts the data to send it to the core.
CheckoutService	Driving Port	Defines the contract of the application core.
CheckoutServiceImpl	Application Service	Orchestrator of the use case. Performs the calls to create the order.

ShoppingCartService	Driven Port	Isolates the application core from service communication details
InventoryService	Driven Port	Isolates the application core from service communication details
PaymentService	Driven Port	Isolates the application core from service communication details
ProductService	Driven Port	Isolates the application core from service communication details
CheckoutRepository	Driven Port	Isolates the application core from data store details
OutboxRepository	Driven Port	Isolates the application core from data store details
EventPublisher	Driven Port	Isolates the application core from message broker details
ShoppingCartServiceImpl	Driven Adapter	Calls the cart checkout HTTP endpoint from the cart microservice to lock the cart and retrieve it
InventoryServiceImpl	Driven Adapter	Calls the inventory reservation HTTP endpoint from the inventory microservice to reserve the cart items.
PaymentServiceImpl	Driven Adapter	Calls the perform payment HTTP endpoint from the payment adapter microservice to pay the cart
ProductServiceImpl	Driven Adapter	Calls the perform payment HTTP endpoint from the payment adapter microservice to pay the cart
CheckoutRepository Impl	Driven Adapter	Saves the checkout information in PostgreSQL
OutboxRepositoryImpl	Driven Adapter	Saves the event in PostgreSQL, implementing the **transactional outbox pattern**
EventPublisherImpl	Driven Adapter	Publishes the event indicating that the purchase is performed through Kafka.

The controller validates input parameters using Jakarta validation annotations, which provide a declarative way of validating an attribute.

```
PaymentDetailsRequest.java
@NotBlank(message = "Card number is required")  1 usage
@Size(min = 15, max = 16, message = "Card number must be 15 or 16 digits")
@Pattern(regexp = "\\d+", message = "Card number must contain only digits")
String cardNumber,
```

Figure 5.49 Use of Jakarta Validation Annotations

6 Deployment

This section describes the deployment process, including creating cloud resources, setting up Docker Compose, and transitioning to Kubernetes.

6.1 Initial Development Setup

At the beginning of development, when only a few microservices were present, they were run using IntelliJ IDEA as the IDE, and the JDK was downloaded locally. The required infrastructure, such as databases and message brokers, is booted up using the Spring Docker Compose library, which ties the containers' lifecycle to the application, i.e. starts when the application is started and stops when the application is stopped. These dependencies were specified in a local Docker Compose file.

While the number of microservices has been growing, the infrastructure that is shared between them was moved from the application's Docker Compose file to a shared Compose file that is manually started, thereby sharing the infrastructure and avoiding coupling between services.

6.2 Docker Integration

Once a considerable number of microservices were finished, the initialization through the IDE was not optimal. To run them locally and fast without IDE overhead, services can be started as Docker containers, which implies creating the Docker image. Instead of creating the Dockerfile manually, the Cloud Native Buildpacks[12] functionality was used, which allows image creation using the Gradle task *bootBuildImage*.

A Docker Compose file was created to customize the container creation with exposed ports and environment variables. In the early stages of development, the infrastructure needed for the service was stated in the Compose file, which reduces the development cost since there were no instances in the cloud yet. To properly start the container, the dependencies from other services can be stated in the Compose file to wait for their start before running the container.

[12] https://docs.spring.io/spring-boot/reference/packaging/container-images/cloud-native-buildpacks.html

```yaml
docker-compose.yml
inventory:
  image: inventory:0.0.1-SNAPSHOT
  restart: unless-stopped
  container_name: inventory
  depends_on:
    kafka:
      condition: service_healthy
    postgres:
      condition: service_started
  environment:
    - POSTGRES_HOST=postgres
    - POSTGRES_PORT=5432
    - POSTGRES_USER=postgres
    - POSTGRES_PASSWORD=postgres
    - POSTGRES_DB=cordillera-coffee
  ports:
    - "8084:8080"
  networks:
    - cordillera-network
```

Figure 6.1 Inventory Service in Docker Compose File

The Docker network allows services in the compose file to communicate with each other. Although a network is created by default, it is recommended to create it manually to specify its name and driver. The communication can be performed using their service name rather than an IP address.

6.3 Cloud Resources Creation

Once all the microservices were finished, the cloud resources were created, except for

- Azure Container Registry, Application Insights, and Azure Kubernetes cluster, which were created later.

- Azure Storage and Azure Search, which were already created and used during the development phase, since they are cloud-specific technologies, and there were no available containers to run them locally.

The resource creation was performed through the Azure Portal. The created resources were:

1. Azure Redis Cache
2. Azure Database for PostgreSQL
3. Azure Cosmos for MongoDB
4. Azure Event Hub

After that, a new Docker Compose file was created without the infrastructure containers, as there is no need to run them locally, since the microservices will point to the cloud instances. The environment variables were also changed to reference the cloud resources.

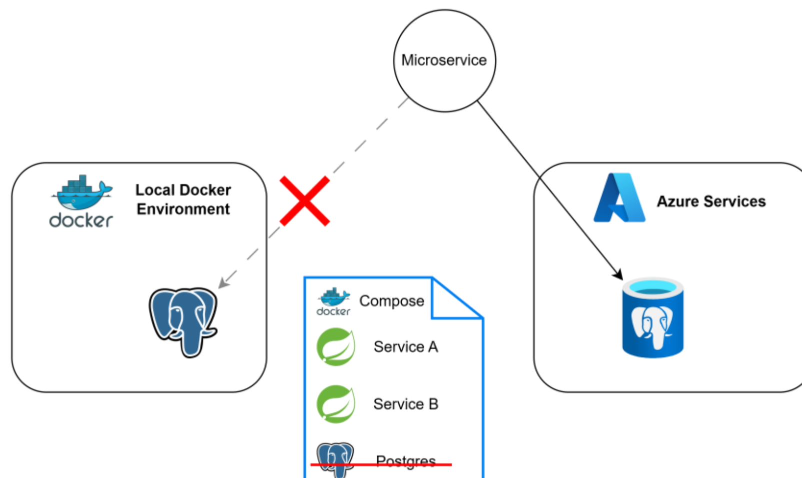

Figure 6.2 Shift from local resources to cloud resources

6.4 Kubernetes Integration

The Azure Container Registry (ACR) was created to facilitate image pulls from the cluster. This process used the Azure Platform to create the resource and then Azure CLI and Docker commands to upload all the images.

The first step in running the microservices in a Kubernetes environment was to create the infrastructure manifests to deploy the ConfigMap, which contains the environment variables that pods will use.

```yaml
configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cordillera-variables
data:
  POSTGRES_HOST: "cordillera-coffee-db.postgres.database.azure.com"
  POSTGRES_PORT: "5432"
  POSTGRES_DB: "cordillera_coffee"
  POSTGRES_USER: "cordillera_admin_postgres"
  AZURE_STORAGE_ACCOUNT_NAME: "cordillera"
```

Figure 6.3 Application's ConfigMap

Another Kubernetes object needed before the Pods were created was the secrets, which are environment variables that hold sensitive information, such as database passwords.

```yaml
cordillera-secrets.yaml
apiVersion: v1
kind: Secret
metadata:
  name: redis-password
type: Opaque
stringData:
  REDIS_PASSWORD: '<redis password>'
---
apiVersion: v1
kind: Secret
metadata:
  name: eventhub-connection-string
type: Opaque
stringData:
  EVENTHUB_CONNECTION_STRING: '<connection string>'
```

Figure 6.4 Application's secrets

The Kubernetes deployment for each microservice can reference the ConfigMap and required secrets to run each microservice as a Pod.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-adapter
  labels:
    app: payment-adapter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: payment-adapter
  template:
    metadata: ObjectMeta
    spec:
      containers:
        - name: payment-adapter
          image: cordilleracoffee.azurecr.io/samples/paymentadapter:0.0.1-SNAPSHOT
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
              protocol: TCP
          resources:
            limits:
              cpu: 300m
              memory: 1Gi
            requests:
              cpu: 150m
              memory: 512Mi
          envFrom:
            - configMapRef:
                name: cordillera-variables
      restartPolicy: Always
```

Figure 6.5 Payment Adapter Deployment

The main elements in the manifest are:

1. **Replicas:** This value is used by Kubernetes to determine the number of Pods that will be running. Since the project will use Horizontal Pod Autoscaler (HPA), it will be used at the beginning of the deployment; then, the HPA will control this value
2. **Image:** Reference the Docker image that, in this case, is uploaded to Azure Container Registry
3. **Port:** container port that will be exposed
4. **Resources:** The "requests" section specifies the initial memory and CPU assigned to each Pod; it can grow until it reaches the specified limits.

The next step was to create the k8s Service, which provides a way to access pods by using a unified name.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: payment-adapter
spec:
  selector:
    app: payment-adapter
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

Figure 6.6 Payment Adapter K8s service

The “ClusterIP” service type does not expose the pods outside the cluster, so they can be accessed only by other pods in the same cluster. It redirects the request received on port 80 to port 8080, which is the port exposed by the pod.

For each deployment, a horizontal pod autoscaler was created. It verifies a specified metric, in this case, CPU usage, and if it passes a threshold, the HPA will increase the number of pods.

```yaml
payment-adapter.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: payment-adapter-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: payment-adapter
  minReplicas: 1
  maxReplicas: 4
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

Figure 6.7 Payment Adapter's HPA

The HPA configuration is structured as follows:

1. Defines the target on which the HPA will act.
2. The HPA will act on a Kubernetes deployment, in this case, the payment-adapter deployment
3. The minimum number of replicas that the deployment will have. It is the default value if no traffic is received.
4. The maximum number of replicas that the HPA can start; in this case, no more than four pods of the payment-adapter service will be started.
5. This section defines the metrics that the HPA must observe to scale up or down the number of replicas.
6. The HPA will check the CPU metrics.
7. Specifically, the HPA will start a new pod if the CPU usage of the available pods is greater than 50%

Currently, Kubernetes does not have a way to validate if the Spring Boot application is healthy. So far, it is responsible for creating the container and

keeping it running, but there is no way to know what happens inside it. To solve this problem, Spring Boot Actuator[13] can be used to enable application health checks. Once it is enabled, the deployment's liveness and readiness can be specified.

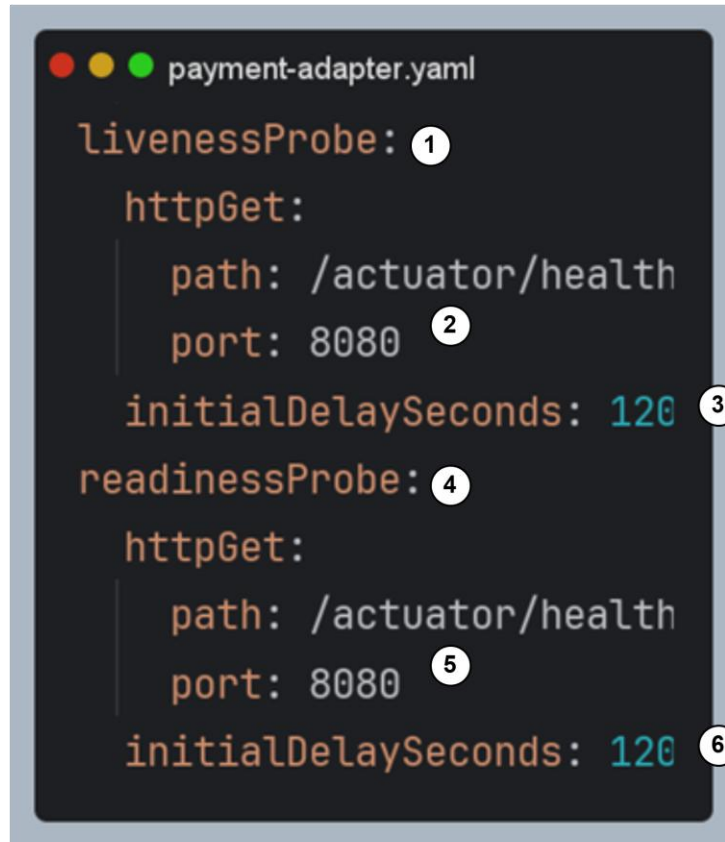

Figure 6.8 Liveness and Readiness Properties

The liveness and readiness probes configuration is structured as follows:

1. The liveness probe is used to check if the application is alive. If the application does not respond three times, Kubernetes will mark the pod as unhealthy, terminate it, and start a new one. In this case, the liveness probe will be related to an HTTP GET request.
2. Kubernetes will periodically request the endpoint /actuator/health, exposed by the microservice, to check the liveness.
3. This attribute is the number of seconds before starting to check the liveness.
4. The readiness probe is used to check if the application is ready to receive requests. If the application does not respond three times, Kubernetes will redirect the request to other pods of the same deployment.

[13] https://docs.spring.io/spring-boot/reference/actuator/enabling.html

5. Kubernetes will periodically request the endpoint /actuator/health, exposed by the microservice, to check the readiness.
6. Represents the number of seconds before the readiness check begins.

Spring Boot applications consume a high number of resources during startup because bean scanning is performed at runtime, which causes a slow start process. Therefore, if this delay is not specified, Kubernetes will assume that the service is not working and will repeatedly terminate the pod.

Furthermore, each microservice has minimal CPU and memory resources, which were intentionally defined, given the experimental nature of this work, and to reduce cloud costs. This allocation means that internal starting processes must be queued, thereby increasing the time it takes to start. For these reasons, the initial delay seconds for liveness and readiness probes were fixed at two minutes.

Finally, an Ingress was created to access the application's HTTP endpoints. It serves as a reverse proxy that allows calls to be made using a single IP and redirects them to their corresponding service.

cordilera-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: cordillera-ingress
spec:
  ingressClassName: webapprouting.kubernetes.azure.com
  rules:
    - http:
        paths:
          - path: /v1/products
            pathType: Prefix
            backend:
              service:
                name: product
                port:
                  number: 80
          - path: /v1/pricing
            pathType: Prefix
            backend:
              service:
                name: pricing
                port:
                  number: 80
```

Figure 6.9 Application's Ingress

Once all the Kubernetes manifests have been created, a cluster was created using Azure's platform. It was created using the Free Tier and the D2as_v6 machines for the node pool, which provides two vCPUs and 8 GiB of RAM for each node.

Finally, the cluster connection with the local environment was performed, and using *kubectl apply -f <file or dir>*, all the resources were deployed.

Using the Azure portal, the deployments can be listed with the following information:

- Deployment name.
- Namespace.
- Number of pods running.
- Deployment live time.
- Percentage of CPU used by the pods that are related to this deployment.
- Percentage of memory used by the pods that are related to this deployment.

product	default	1/1	1 hour	0%	44%
payment-adapter	default	1/1	1 hour	0%	23%
mock-payment	default	1/1	57 minutes	0%	17%
pricing	default	1/1	54 minutes	0%	45%
inventory	default	1/1	36 minutes	3%	37%
cart	default	1/1	30 minutes	0%	37%
checkout	default	1/1	26 minutes	0%	35%
product-query	default	1/1	22 minutes	0%	33%
order	default	1/1	20 minutes	0%	35%
order-query	default	1/1	17 minutes	3%	26%
app-notifications	default	1/1	15 minutes	3%	24%

Figure 6.10 Deployments in AKS

The number of nodes can be configured to be automatically autoscaled; once a node pool does not have enough resources to start a pod, AKS will scale up the number of nodes.

Node	Status ⓘ	CPU ⓘ	Memory ⓘ	Disk ⓘ	Pods
aks-nodepoolamd-38412631-vmss000000	Ready	7%	74%	-	20
aks-nodepoolamd-38412631-vmss000001	Ready	10%	74%	-	13
aks-nodepoolamd-38412631-vmss000002	Ready	7%	72%	-	12
aks-nodepoolamd-38412631-vmss000004	Ready	9%	83%	-	13

Figure 6.11 AKS Nodepools

The number of pods is higher than the number of microservices deployed because Kubernetes automatically runs multiple deployments to manage the cluster. For example, the coredns deployment “is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS.” (Kubernetes, 2024).

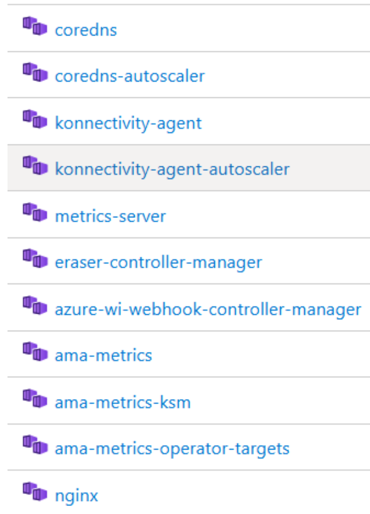

Figure 6.12 Pods maintained by Kubernetes

The resources were monitored using both the Azure platform and the Open Lens IDE.

Name	Namespace	Metrics	Min Pods	Max Pods	Replicas
app-notifications-hpa	default	unknown / 50%	1	4	1
cart-hpa	default	2% / 50%	1	4	1
checkout-hpa	default	1% / 50%	1	4	1
inventory-hpa	default	2% / 50%	1	4	1
order-hpa	default	3% / 50%	1	4	2
order-query-hpa	default	6% / 50%	1	4	2
payment-adapter-hpa	default	0% / 50%	1	4	1
pricing-hpa	default	0% / 50%	1	4	1
product-hpa	default	1% / 50%	1	4	1
product-query-hpa	default	6% / 50%	1	4	3

Figure 6.13 HPA in AKS analyzed from Open Lens

6.5 Observability

Azure Platform provides basic observability features by default to analyze what is happening in the Kubernetes cluster.

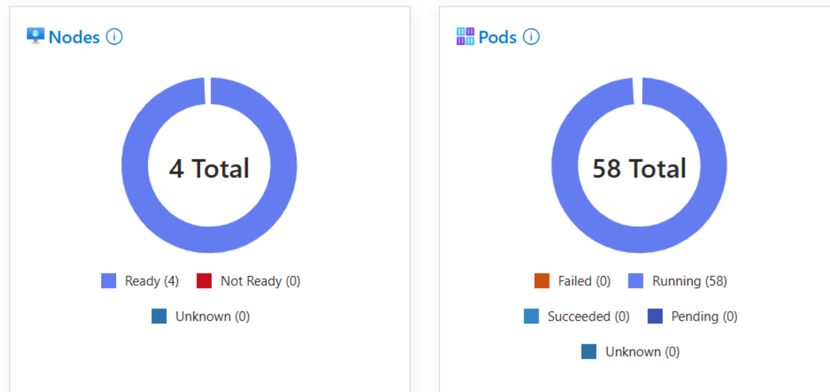

Figure 6.14 Basic Monitoring from AKS

To improve the observability and analyze what is happening in the Spring Boot applications rather than just in their containers, an **Application Insights** resource was created using the Azure Portal.

Azure Application Insights will gather information about microservices and applications hosted in the cloud. It will provide a central point to analyze the application performance and track system failures. It centralizes the logs and metrics that can be visualized through the Azure portal.

The Application Insights agent deployed in the Kubernetes cluster gathers information from the Spring Boot application and sends it to the cloud.

```yaml
# app-insights.yaml
apiVersion: monitor.azure.com/v1
kind: Instrumentation
metadata:
  name: default
  namespace: default
spec:
  settings:
    autoInstrumentationPlatforms: ["Java"]
  destination:
    applicationInsightsConnectionString: "<application-insights-connection-string>"
```

Figure 6.15 Application Insights Agent Resource

The Spring Boot applications will send data periodically to Application Insights, which provides complete observability features.

Application Insights provides an application map that shows the Kubernetes deployments and their connections. Initially, just the Product microservice was deployed to verify the implementation.

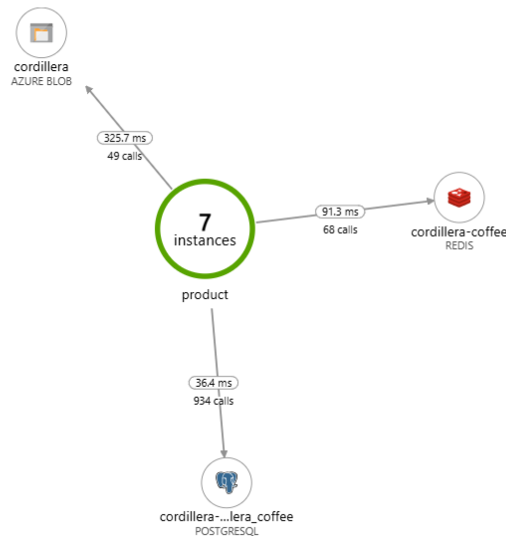

Figure 6.16 Product Microservice in Application Map

Once this product was shown correctly, the remaining microservices were deployed. The image below illustrates the checkout process.

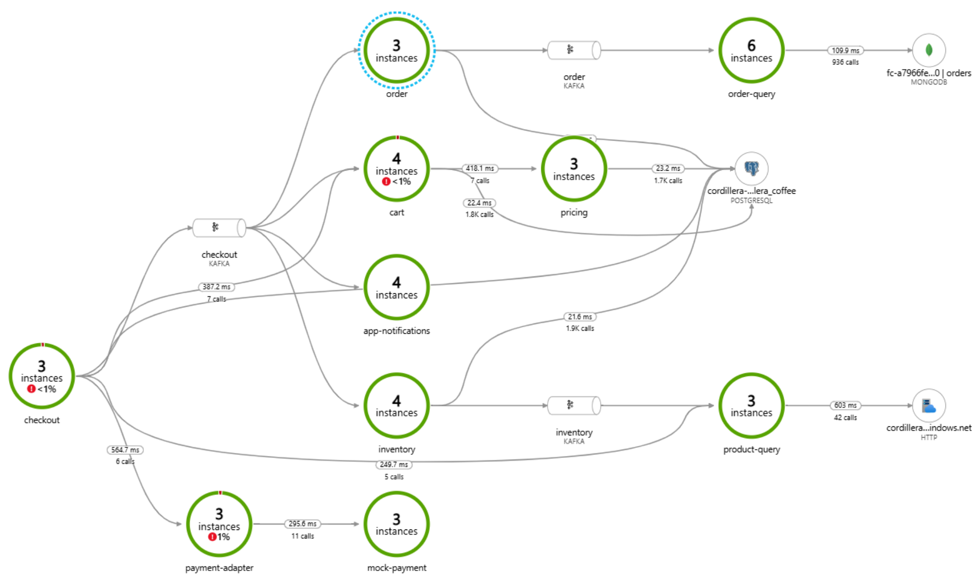

Figure 6.17 Checkout Process in the Application Map

The application map provides multiple information at a first glance. It shows the average response time in a connection, the percentage of failed calls, and the number of instances used to show the information (for example, if the pod has been replicated via HPA or relaunched due to a liveness problem). When clicking a node, all the details can be seen, such as the instances that were taken into account to produce these metrics, which are identified by the pod name.

One problem that occurred and could be recognized using the application map was the wrong port while connecting with other services.

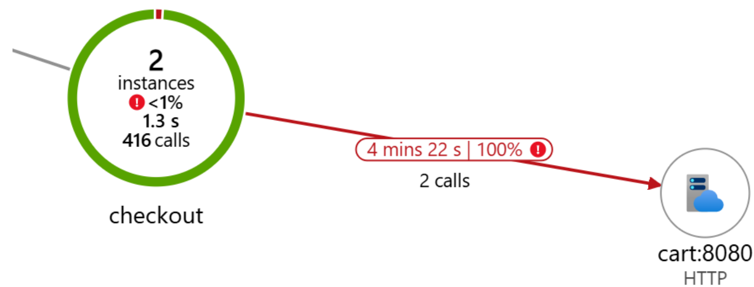

Figure 6.18 Application Map used to spot problems

With this visual information, the problem was easily recognized and fixed. In this case, the port specified in the config map used by the deployment as an environment variable to connect to the cart was set to 8080, while the service exposes port 80.

The application map can be filtered to show specific nodes of interest. For example, the image below shows the order creation in the application map and the usage of the CQRS pattern.

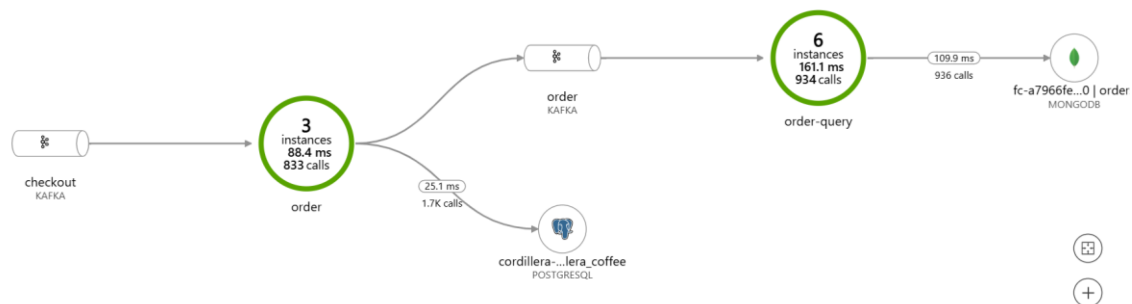

Figure 6.19 Order Creation in Application Map

Application Insights also provides multiple metrics to analyze performance and failures, for example, it shows the most frequent error response codes in the application.

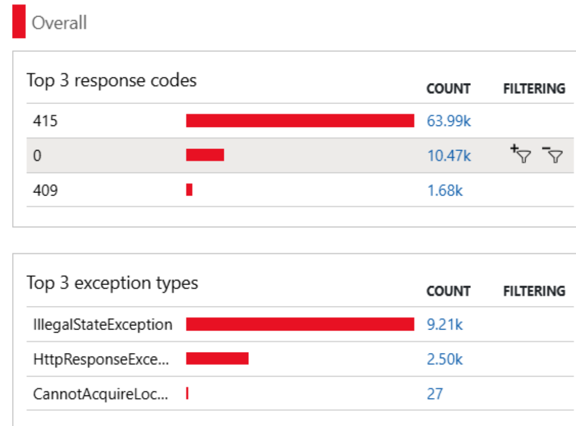

Figure 6.20 Error Codes in Application Insights

These errors can be filtered to analyze in detail what happened.

7 Performance Tests and Non-functional Requirements Check

This section describes the load test made to the application through Azure Load tests with the obtained results and the relation with non-functional requirements, as described in section 3.6.

The tests were performed with the same lightweight and cost-efficient resource configuration for all the microservices, which is:

- 150 millicores of CPU as base and a limit of 300 millicores of CPU
- 512 Mebibytes of memory as base and a limit of 1 Gibibyte
- 3 replicas per microservice, which were fixed during the test

7.1 Tests for Purchasing Products

7.1.1 Success Rate for Purchasing Process

According to the non-functional requirement NFR-1, a 99% success rate is required in a normal operation. According to the baseline defined in section 3.6, the application will receive one thousand purchases per day during normal operation.

The results obtained in the load test were:

- 1,107 requests were performed in 5 minutes and 7 seconds, which means approximately 300,000 requests per day.
- A success rate of 99.46% was achieved.
- The average response time was of 430 milliseconds.

As mentioned earlier, a normal operation is defined as 1,000 purchases per day. Therefore, the results obtained demonstrate that the system can handle many more requests than the proposed requirement while maintaining the success rate.

Statistics

Load	Duration	Response time	Error percentage	Throughput
1107	5 mins, 7 secs	430.00 ms	0.54 %	3.61 /s
Total requests		90th percentile response time	Aggregate requests which failed	Request rate

Figure 7.1 Success Rate for Purchasing Process Test Results

The image below shows the requests per second during the test. Which, on average, is 3.57 requests per second.

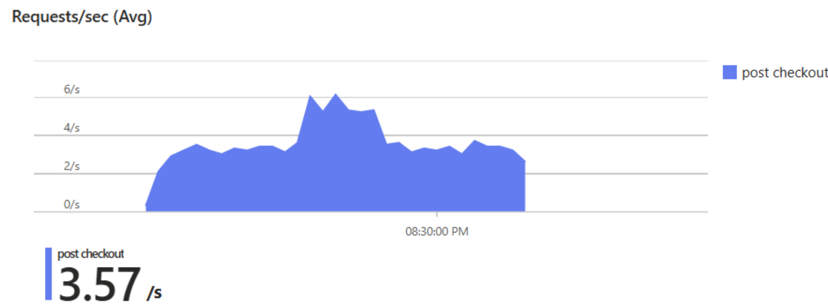

Figure 7.2 Requests per Second for 'Success Rate for Purchasing Process' Test

Finally, we can observe that the response time remains consistent during the test, and is smaller than the three seconds proposed in the quality scenario.

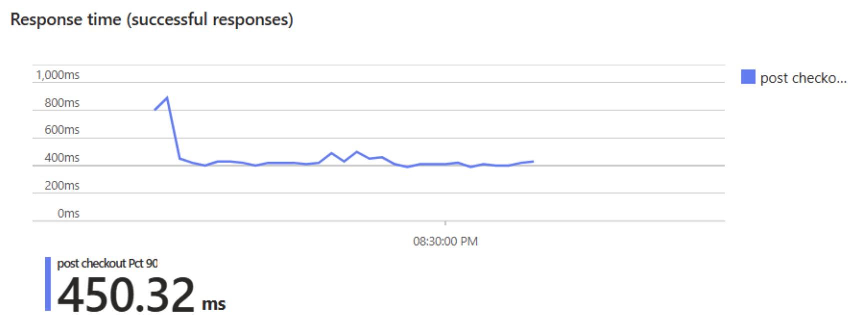

Figure 7.3 Response Time for 'Success Rate for Purchasing' Process Test

7.1.2 Performance for Purchasing Process

The non-functional requirement NFR-5 states that the system must support up to 1,000 users within a ten-minute window and process 100 orders per minute with a response time of up to 3 seconds.

To validate this requirement, the endpoint for performing the checkout was chosen, as it reflects a critical system operation and is the one used in the quality scenario.

The concurrent users cannot be simulated to perform the test. However, despite the requests being made by a single virtual user, the number of requests is so large that it effectively simulates the activity of 1,000 users. We have satisfied this requirement through previous tests; however, a test with a higher load was conducted.

The test obtained the following results:

- The checkout process handled 1.618 requests in 5 minutes and 3 seconds, which would be 320 orders per minute and approximately 460.000 requests in one day.
- A success rate of 98.39%
- An average response time of 580ms

We can see a degradation of the success rate, where 98.39% of the requests were successful. This degradation of performance could be due to the minimal resources that microservices have to operate, a thorough analysis can be performed in future work to determine the right amount of memory, CPU, and number of replicas each microservice should have to serve the required demand.

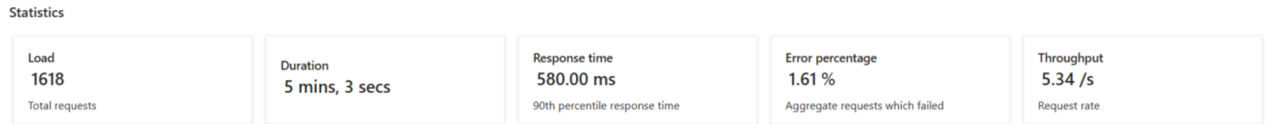

Figure 7.4 'Performance for Purchasing Process' Test Results

The average number of requests per second was 5.22.

Figure 7.5 Requests per Second for 'Performance for Purchasing Process' Test

We can also observe an increment in the average response time (904 ms). However, it is still below the threshold of three seconds stated in the quality scenario of the NFR-5.

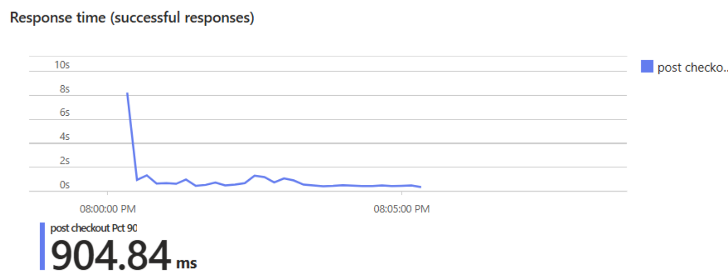

Figure 7.6 Response Time for 'Performance for Purchasing Process' Test

7.2 Test for Retrieving Products

7.2.1 Success Rate for Product Catalog

The non-functional requirement NFR-2 states that a 99% success rate is required in operations that are critical and do not involve third-party services. The product summary of the product query microservice was chosen for this experiment, since it can be considered a critical operation.

According to the baseline defined in section 3.6, a normal operation is defined as 50,000 product views per day, since an e-commerce site's conversion rate is around 2% (Shopify, 2024). This test was performed with 99.000 products in the database.

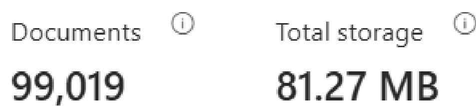

Figure 7.7 99,000 Products in Azure Search

These documents were uploaded in batches of 1,000 products using a custom endpoint in the Product Query microservices, which was created specifically for this purpose.

The test obtained the following results:

- The product query served 12,510 requests (to obtain product details) in 4 minutes and 32 seconds, which would be 3,900,000 requests per day.
- A success rate of 100%
- An average response time of 44 milliseconds

The results obtained demonstrate that the system can handle many more requests, which in this case are to obtain product information, than the proposed by the non-functional requirement.

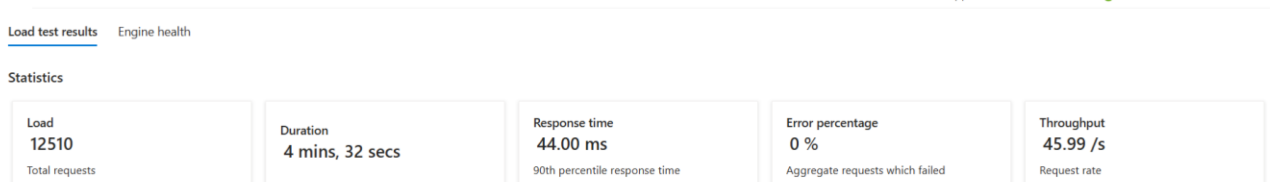

Figure 7.8 'Success Rate for Product Catalog' Test Results

The image below shows the requests per second during the test.

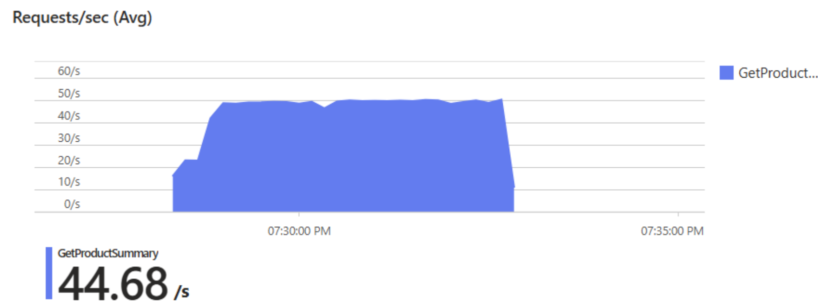

Figure 7.9 Requests per Second of 'Success Rate for Product Catalog' Test

We can observe that the response time remains consistent during the test. And it is much less than the one second proposed by the quality scenario.

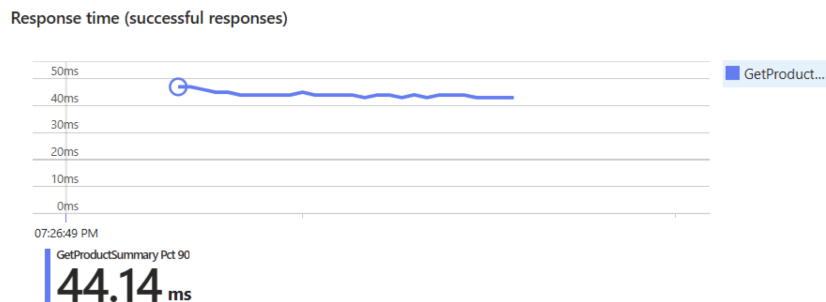

Figure 7.10 Response Time for 'Success Rate for Product Catalog' Test

7.2.2 Performance of Product Catalog

The non-functional requirement NFR-6 states that the number of products must scale to 100,000 without degrading the performance during peak operations. According to the quality scenario, the system must handle 1,000 users per hour. The concurrent users cannot be simulated to perform the test. However, as mentioned earlier, the number of requests is so large that it effectively simulates the activity of 1,000 users. We have already satisfied this requirement with the previous test. However, a test with more load was performed.

The results obtained by the test are:

- 45,027 requests to obtain product information were performed in 5 minutes and 8 seconds, which would be approximately 12,630.000 requests per day.
- A 99.9% success rate was achieved.
- An average response time of 163 milliseconds

According to the quality scenario of the NFR-2, list products must be performed in less than one second with a 99% success rate. This test verifies that even during peak traffic, response times remain significantly lower than the specified threshold, thereby complying with the non-degradation of performance proposed by the quality scenario of NFR-6, which proposes a response time threshold of 3 seconds.

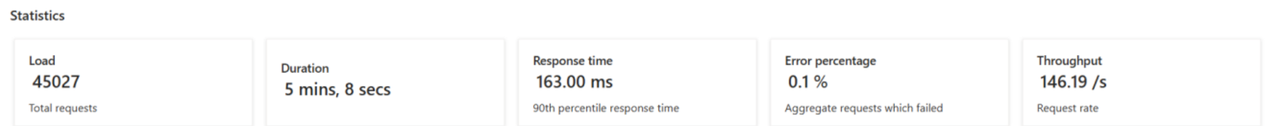

Figure 7.11 'Performance of Product Catalog' Test Results

The service receives 150 requests per second on average.

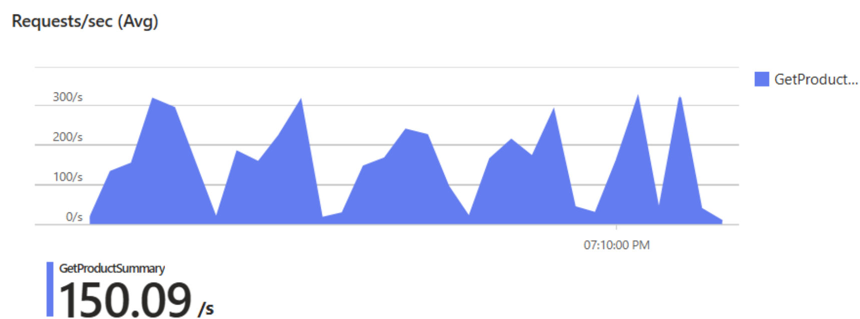

Figure 7.12 Requests per Second for 'Performance of Product Catalog' Test

The response time was more intermittent than in other tests. However, the average of 163 ms is acceptable according to the NFR and quality scenarios.

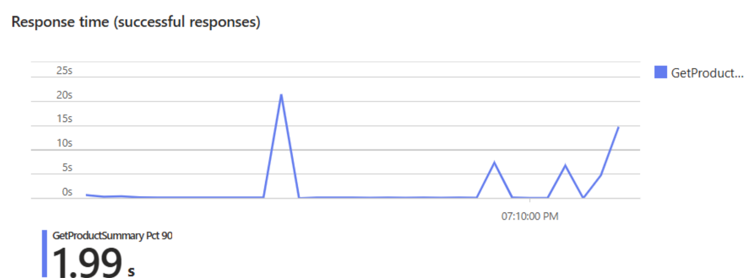

Figure 7.13 Response Times for 'Performance of Product Catalog' Test

7.3 Error Handling Non-functional requirement

The non-functional requirement NFR-4 states that the application does not crash during an error and remains in a consistent state. It is accomplished by using the following techniques:

1. Microservices' modularity provides the benefit that if one service crashes, the rest of the application can continue operating.
2. Using Kubernetes with readiness and liveness probes ensures that if one pod is not healthy, Kubernetes will start another pod to replace the unhealthy one.
3. Using the SAGA pattern to perform compensating transactions ensures that the application's state remains consistent. For example, if the payment fails, thanks to SAGA, the inventory microservice will release the reserved stock, and the cart microservice will unlock the cart, allowing the user to retry the payment without any issues.

7.4 Modularity Non-functional Requirement

The non-functional requirement NFR-7 states that new features have to be easily integrated into the system. This is accomplished by using a microservices structure, where a new module does not depend on other modules or developers. The continuous integration pipeline allows the team to push changes to production without any obstacles, without worrying about breaking other module tests that may be misconfigured, or performing Git tricks to upload to production only the team's changes. There are no communication issues with other teams that are developing in the same repo, as could happen if the project were a monolith.

7.5 Logging Non-functional Requirement

The non-functional requirement NFR-8 states that the system should have a logging mechanism to allow quick issue identification. The quality scenario states that logs should be available within 2 seconds.

The technology used accomplishes the requirement and the quality scenario. Kubernetes enables easy log retrieval using the command kubectl logs <pod-name>, which stores logs in the pod worker and makes them available within milliseconds of generation. However, the cluster can be enhanced with Grafana and Prometheus instances to retrieve and thoroughly analyze the logs.

7.6 Security Non-functional Requirement

The non-functional requirement NFR-9 states that the system should prevent requests from accessing operations that require authentication or authorization. Although security was not implemented in the MVP, the Kubernetes cluster is ready to be wrapped by a service like API management, which validates the JSON Web Token and calls the Ingress. Additionally, the entire Kubernetes cluster is in a virtual network and cannot be accessed from outside. Implementing these improvements does not affect the microservices or the application features.

7.7 Availability Non-functional Requirement

The non-functional requirement NFR-10 states that the system should maintain an uptime of 99.9%. This can be accomplished by:

- According to the Microsoft AKS documentation, “The Uptime SLA feature guarantees 99.95% availability of the Kubernetes API server endpoint for clusters using Availability Zones, and 99.9% of availability for clusters that aren't using Availability Zones”

- Apart from the cluster SLA, tools like readiness and liveness help to reduce the downtime if a microservice fails.

8 Conclusions and Future Work

8.1 Conclusions

The thesis presented a cloud native solution for a coffee e-commerce application, providing a scalable, reliable, maintainable, and modifiable system that enables local coffee producers to sell directly to consumers, increasing their earnings. On the other hand, consumers can find a wide variety of high-quality coffee products in one marketplace specifically designed for them.

The requirements specification and high-level design were developed for the entire application, while the interaction between microservices were performed for the user stories that comprise the Minimum Viable Product (MVP), which includes the processes of creating a product, adding and removing products to the cart, listing products, viewing the product details, and performing the checkout. The developed and deployed microservices, along with the detailed design, were a subset of the microservices interactions and were chosen to apply cloud patterns and measure system performance. The included user stories are the process of creating the product and the checkout.

The user stories were thoroughly defined by adding acceptance criteria following the Given-When-Then structure, which provides a better understanding of the user's needs and helps to determine when a story is finished.

The design was an essential part of the project, where the high-level design was performed following Domain Driven Design by creating the domain and context models, which provided a common understanding of the system's characteristics. Next, the bounded contexts were defined to derive the microservices from them, ensuring high cohesion within each microservice, and describing the responsibilities of each one.

Moreover, the detailed design was performed to describe the communication between services and the publication of events through the message broker, taking into account multiple cloud architectural patterns, specifically:

- Command Query Responsibility Segregation (CQRS)
- Saga
- Circuit breaker
- Backend for frontends
- Transactional Outbox
- Event-driven architecture (EDA)
- Anti-corruption layer

The microservices were developed using modern technologies and following design principles. Some of them included design patterns such as the strategy and factory patterns, apart from the ones that come with the framework, such as the singleton pattern. Most of the microservices adhere to a modified version of the hexagonal architecture, having rich domain entities that enforce their business rules and invariants. The changes made to the architecture include merging domain and persistence entities into a single class.

Docker and Kubernetes were chosen as technologies that leverage mechanisms to constantly deploy microservices, providing multiple benefits such as DNS for each service, autoscaling capabilities, and health monitoring. Furthermore, the Application Insights service was selected since it provides complete observability over the system, offering metrics such as response time, number of requests, used resources, and common failures.

Finally, the application's performance was tested through load tests, yielding outstanding results despite the low-cost infrastructure configuration and showing that non-functional requirements can be met successfully.

An important lesson learned during development was the trade-off between architectural perfectionism and project pragmatism, specifically using hexagonal architecture. While it provides a clear separation between layers, the number of classes required to achieve this can be high, which slows down project development due to the constant mapping between objects that belong to different layers.

Another key learned lesson was the importance of assigning each service boundaries from the beginning of the application, since changes after this definition are costly in both time and resources. For example, the Inventory Management context was not conceived from the beginning, which causes multiple changes to microservices interactions and data stores to incorporate this new context.

Related to microservices architecture, as the system grows and more functionalities are needed, it can expand considerably, resulting in a large number of microservices. For example, to create the subset of the MVP, which is a small portion of the whole system, 11 microservices were deployed. This complexity needs solid observability features and automatic alerts, as well as strong architectural decisions to keep the system manageable. Furthermore, this architecture brings challenges that have to be addressed from the beginning, such as distributed transactions, eventual consistency, and asynchronous communication.

8.2 Future Work

The system's functionalities demonstrate robustness, scalability, and modifiability, providing a solid baseline for further development. Despite this, multiple improvements can be performed, which are described below:

1. **Full System Implementation:** The current project has developed a small subset of the system's use cases, which allows the thesis objectives to be met. Therefore, the design and implementation of all use cases related to the user stories described in Section 3 could be completed.
2. **Integration of an API gateway and user authentication:** The developed microservices are intended to be as agnostic as possible of security infrastructure concerns. Hence, integrating with a customer identity

access management system and communicating with it through an API gateway must be considered for deploying the system in a production environment.

3. **Front-End Development:** To provide a user interface, which is the primary mechanism for users to communicate with the application, building a responsive front-end using technologies like Angular or React is necessary.
4. **Analysis of required resources for each microservice:** Currently, all microservices are configured to have the same memory, CPU, and number of replicas. However, a thorough analysis of the requirements for each functionality is necessary to determine the expected load, ensuring that the given resources can support it. This analysis also helps to optimize cloud resources, which are directly related to infrastructure costs.
5. **Infrastructure as Code (IaC) and GitOps:** To ensure the replicability of the system's infrastructure, rather than manually creating cloud services through the Azure portal, a declarative infrastructure approach using Terraform or Bicep would improve consistency and traceability across infrastructure changes.
6. **Helm charts:** To reduce the complexity overhead of Kubernetes manifests, a Helm Chart could be used, providing default values for properties that do not change across microservices.
7. **Performance Optimization:** Although Spring Boot is a widely used framework for enterprise applications, it has some performance issues at the start of the applications. To reduce this problem, Spring Ahead of Time Optimizations (AOT) and a GraalVM native image can be used in the future.
8. **Observability Improvements:** To identify problems faster, receiving notifications about issues in real-time, the configuration of alerts through email or communication platforms can be performed.

9 Bibliography

Ashanin, N. (2018, March 28). *Quality attributes in software architecture - Nikolay Ashanin - Medium.* Medium. https://medium.com/@nvashanin/quality-attributes-in-software-architecture-3844ea482732

AWS. (n.d.). *What is SLA? - Service Level Agreement Explained - AWS.* Amazon Web Services, Inc. Retrieved June 4, 2025, from https://aws.amazon.com/what-is/service-level-agreement/

AWS. (2025). *Hexagonal architecture pattern - AWS Prescriptive Guidance.* Aws.amazon.com. https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/hexagonal-architecture.html

AWS. (2025). *Retry with backoff pattern - AWS Prescriptive Guidance.* Amazon.com. https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/retry-backoff.html

Azure. (n.d.). *Azure Database for PostgreSQL documentation.* Microsoft. Retrieved May 31, 2025, from https://learn.microsoft.com/en-us/azure/postgresql/

Azure. (n.d.). *Saga Design Pattern - Azure Architecture Center.* Microsoft.com. Retrieved June 5, 2025, from https://learn.microsoft.com/en-us/azure/architecture/patterns/saga

Azure. (2022, October 29). *Asynchronous message-based communication.* Microsoft.com. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communication

Azure. (2024, May 12). *What is Azure Kubernetes Service (AKS)? - Azure Kubernetes Service.* Microsoft.com. https://learn.microsoft.com/en-us/azure/aks/what-is-aks

Azure. (2024, May 21). *What is Azure Load Testing?* Microsoft.com. https://learn.microsoft.com/en-us/azure/load-testing/overview-what-is-azure-load-testing

Azure. (2024, August 14). *Introduction/Overview - Azure Cosmos DB for MongoDB.* Microsoft.com. https://learn.microsoft.com/en-us/azure/cosmos-db/mongodb/introduction

Azure. (2024, September 19). *Introduction to Azure Container Registry - Azure Container Registry.* Microsoft. https://learn.microsoft.com/en-us/azure/container-registry/container-registry-intro

Azure. (2024, November 15). *What is Azure Cache for Redis? - Azure Cache for Redis.* Microsoft. https://learn.microsoft.com/en-us/azure/azure-cache-for-redis/cache-overview

Azure. (2024, December 17). *Azure Event Hubs: Data streaming platform with Kafka support - Azure Event Hubs.*

Microsoft.com. https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about

Azure. (2024, December 19). *Azure Event Hubs for Apache Kafka - Azure Event Hubs*. Microsoft.com. https://learn.microsoft.com/en-us/azure/event-hubs/azure-event-hubs-apache-kafka-overview

Azure. (2025). *Cloud Design Patterns - Azure Architecture Center*. Microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/patterns/

Azure. (2025). *Microservice architecture style - Azure Architecture Center*. Microsoft.com. https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Azure. (2025, March 27). *Introduction to Azure Storage - Cloud storage on Azure*. Microsoft.com. https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction

Azure. (2025, May 15). *Introduction to Azure AI Search - Azure AI Search*. Microsoft.com. https://learn.microsoft.com/en-us/azure/search/search-what-is-azure-search

Azure. (2025, May 24). *Application Insights OpenTelemetry observability overview - Azure Monitor*. Microsoft.com. https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

Bachmann, F., Bass, L., & Nord, R. (2007). *Modifiability tactics*. https://insights.sei.cmu.edu/documents/778/2007_005_001_14858.pdf

Bahamón, G. (2024, July 27). *El consumo de café en hogares: más marcas y mayor valor*. Diario La República. https://www.larepublica.co/analisis/german-bahamon-3587238/el-consumo-de-cafe-en-hogares-mas-marcas-y-mayor-valor-3916939

Cloud Native Computing Foundation. (2024, February 26). *Cloud native definition*. GitHub. https://github.com/cncf/toc/blob/main/DEFINITION.md

Cloudflare. (2025). *What is HTTPS?* Cloudflare.com. https://www.cloudflare.com/learning/ssl/what-is-https/

Conway, M. E. (1968). How do committees invent? *Datamation, 14*(5), 28–31.

Dudczak, A. (2015). *Introducing Spring Cloud Contract*. Spring Documentation. https://docs.spring.io/spring-cloud-contract/reference/getting-started/introducing-spring-cloud-contract.html

Eldar Jahijagic. (2022, June 4). *Reliability in software engineering - CodeX - Medium*. Medium; CodeX. https://medium.com/codex/reliability-in-software-engineering-b1c8286eefb7

Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, M. (2011, July 14). *CQRS*. Martinfowler.com. https://martinfowler.com/bliki/CQRS.html

Fowler, M. (2014, March 6). *Circuit Breaker*. Martinfowler.com. https://martinfowler.com/bliki/CircuitBreaker.html

Fowler, M. (2020). *Domain Driven Design*. Martinfowler.com. https://martinfowler.com/bliki/DomainDrivenDesign.html

Fowler, M. (2025). *Is Design Dead?* Martinfowler.com. https://www.martinfowler.com/articles/designDead.html

GitLab. (2022, January 26). *What is CI/CD?* Gitlab.com; GitLab. https://about.gitlab.com/topics/ci-cd/

Harris, C. (2024). *Microservices vs. monolithic architecture | Atlassian*. Atlassian. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

IBM. (2024, June 18). *Defining the resources in RESTful applications*. Ibm.com. https://www.ibm.com/docs/en/was-nd/9.0.5?topic=applications-defining-resources-in-restful

ISE, M. (2024). *Maintainability - Engineering Fundamentals Playbook*. Github.io. https://microsoft.github.io/code-with-engineering-playbook/non-functional-requirements/maintainability/

Jakub Lambrych. (2024, June 25). *Domain-Driven Design (DDD): Strategic Design Explained*. Medium. https://medium.com/@lambrych/domain-driven-design-ddd-strategic-design-explained-55e10b7ecc0f

Knuth, D. E. (1974). *Structured programming with go to statements. ACM Computing Surveys, 6(4), 268.* https://doi.org/10.1145/356635.356640

Kubernetes. (2024, January 14). *Using CoreDNS for Service Discovery*. Kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/coredns/

Lewis, J., & Fowler, M. (2014, March 25). *Microservices*. Martinfowler.com. https://martinfowler.com/articles/microservices.html

Maldini, S., & Baslé, S. (2025). *Flux, an asynchronous sequence of 0-N items. Reactor Core Reference Guide* Projectreactor.io. https://projectreactor.io/docs/core/release/reference/coreFeatures/flux.html

Martin, R. C. (2000). *Design principles and design patterns*. Object Mentor Inc. Retrieved from https://staff.cs.utu.fi/~jounsmed/doos_06/material/DesignPrinciplesAndPatterns.pdf

Martin, R. C. (2017). *Clean architecture: A craftsman's guide to software structure and design*. Prentice Hall.

Miro. (2022, May 18). *What is a context diagram and how do you use it? | MiroBlog*. MiroBlog. https://miro.com/blog/context-diagram/

MongoDB. (2025). *What Is NoSQL? NoSQL Databases Explained*. MongoDB. https://www.mongodb.com/resources/basics/databases/nosql-explained

Mozilla. (2025, May 9). *HTTP*. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/HTTP

Newman, S. (2021). *Building microservices: Designing fine-grained systems* (2nd ed.). O'Reilly Media.

Postman. (2022). *What is API-first? The API-first Approach Explained | Postman*. Postman.com. https://www.postman.com/api-first/

Powell, P., & Smalley, I. (2024, September 26). *Monolithic architecture*. Ibm.com. https://www.ibm.com/think/topics/monolithic-architecture

Radhakrishnan Periyasamy. (2020, August 12). *Implementing cart microservice using Domain Driven Design, and Port and Adapter pattern - Part 1*. Medium; Walmart Global Tech Blog. https://medium.com/walmartglobaltech/implementing-cart-service-with-ddd-hexagonal-port-adapter-architecture-part-1-4dab93b3fa9f

Radhakrishnan Periyasamy. (2020, August 12). *Implementing cart microservice using Domain Driven Design, and Port and Adapter pattern - Part 2*. Medium; Walmart Global Tech Blog. https://medium.com/walmartglobaltech/implementing-cart-service-with-ddd-hexagonal-port-adapter-architecture-part-2-d9c00e290ab

Ramalingam, C. (2020, July). *Building Domain Driven Microservices - Walmart Global Tech Blog - Medium*. Medium; Walmart Global Tech Blog. https://medium.com/walmartglobaltech/building-domain-driven-microservices-af688aa1b1b8

Red Hat. (2024). *What is Apache Kafka?* Redhat.com. https://www.redhat.com/en/topics/integration/what-is-apache-kafka

Redis. (2023, August 15). *Domain-Driven Design (DDD) - Fundamentals - Redis*. Redis. https://redis.io/glossary/domain-driven-design-ddd/

Richardson, C. (2017). *Pattern: Database per service*. Microservices.io; Chris Richardson. https://microservices.io/patterns/data/database-per-service.html

Shopify. (2024, November 8). *What's a good average ecommerce conversion rate in 2025?*. Shopify. https://www.shopify.com/blog/ecommerce-conversion-rate

Sonar. (2025). *Java static code analysis | Code Smell*. Sonarsource.com. https://rules.sonarsource.com/java/type/Code%20Smell/

Sonar. (2025). *SonarQube*. Sonarsource.com. https://www.sonarsource.com/products/sonarqube/

Spring. (2022). *Spring Modulith*. Spring Reference Documentation. https://docs.spring.io/spring-modulith/reference/

Spring. (2025). *Enabling Production-ready Features :: Spring Boot*. Spring.io. https://docs.spring.io/spring-boot/reference/actuator/enabling.html

Spring. (2025). *GraalVM Native Images*. Spring Boot Referece Documentation. https://docs.spring.io/spring-boot/reference/packaging/native-image/index.html

Stackoverflow. (2024). *Stack Overflow Developer Survey*. Stackoverflow.co. https://survey.stackoverflow.co/2024

Swagger. (2015). *Best Practices in API Design*. Swagger.io. https://swagger.io/resources/articles/best-practices-in-api-design/

Talha Şahin. (2024, February 10). *Preventing duplicate payments with idempotency keys by Stripe, PayPal and Adyen | Medium*. Medium. https://medium.com/@sahintalha1/the-way-psps-such-as-paypal-stripe-and-adyen-prevent-duplicate-payment-idempotency-keys-615845c185bf

Tawfik, K., Jain, T., Harcourt, A., & Ammaarah Subjally. (2024, June 13). *The rise of the B2C specialty marketplace*. BCG Global. https://www.bcg.com/publications/2024/the-rise-of-the-b2c-specialty-marketplace

Testcontainers. (2023). *Testcontainers overview*. Testcontainers. https://testcontainers.com/

USDA Foreign Agricultural Service. (2025). *Production - Coffee*. Usda.gov. https://www.fas.usda.gov/data/production/commodity/0711100

USDA Foreign Agricultural Service. (2025, May 16). *Colombia: Coffee Annual*. USDA Foreign Agricultural Service. https://www.fas.usda.gov/data/colombia-coffee-annual-9