INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Multi-Concession Cloud-Based Toll Collection and Validation System

HÉLDER FILIPE FERREIRA DOS SANTOS DINIZ

Outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de
Engenharia do Porto

Multi-Concession Cloud-Based Toll Collection and Validation System

Hélder Filipe Ferreira dos Santos Diniz

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Software Engineering

Supervisor: Dr. Paulo Gandra de Sousa
Co-Supervisor: Eng. Ricardo Vilela

Porto, October 15, 2020

Abstract

Society is under an exponential growth of adopting technology for its everyday activities. Countless pieces of engineering are developed, in order to provide services and platforms, aimed at facilitating everyday tasks, to which people are becoming accustomed and dependent on. This is caused by the growth of number of devices *per capita*, as well as based on the proven efficiency that these solutions provide to the needs of its users.

Organisations specialized in providing toll collection services, and managing motorway infrastructure, are no different, and their wish is not to be left behind, joining in the technology revolution, by overhauling how toll collection systems operate, with the addition of new technology into the state of the art, and other services to facilitate payments and the usage of the infrastructure.

In this dissertation, the research and implementation grounded on the proof of concept of a multi-concession, cloud-based, scalable toll collection and validation system, is documented. It is based on the development of a solution for a real case, designed for the Globalvia organisation, that plans at using this multi-concession system in its toll collection infrastructure.

For this, a contextualization of the business key values is made, as well as describing the existing toll mechanisms and toll collection techniques, in order to provide context for understanding the best way to implement good practices and add value to the business and the customer. This contextualization leads to the need of researching the state of the art for toll collection, as well as software architecture, in order to evaluate the existing solutions for the problem in hand.

This platform will be composed of several software components, specialized in handling information generated by specialized components of character recognition, and infrared readers, placed in the road side equipment. These are aimed at offering a solution of managing transactions, providing a segmented response to the needs of two distinct domains: Operational BackOffice and Commercial BackOffice.

It is expected that in the closure of this project, the solution has been analysed and evaluated with success, and the proof of concept has been validated.

Keywords: Toll collection, multi-concession, cloud-based, micro-services, operational and commercial management

Acknowledgement

My deepest gratitude goes towards everyone that crossed paths with me, on my academic, professional and personal life.

I begin by thanking all the academic institutions that gave me an opportunity. Escola Profissional Ruiz Costa, ATEC and Instituto Superior de Engenharia do Porto lead me to become a passionate software engineer. All the passed knowledge, from best practices, technology and critical thinking, will never be forgotten.

To all my family, that has never left my side, supported me on good and bad moments and provided me more than I could ever ask for.

To all my friends, for the continuous support, sticking with me on this journey and keeping me sane, all these years. A special mention for my *homies* at "*Sala de Chuto*".

To all my colleagues, "*Inocentes e Estudiosos*", that have shown resilience, companionship and true friendship since CDIO. I feel extremely fortunate for sharing my academic path with such a bright group of engineering masterminds.

To all my folks at Mindera, you're the best teammates I could ever ask for.

To Globalvia Transmontana, for allowing me to show this solution, in an academic context. Thank you for letting me participate in such a revolutionary software overhaul.

Ultimately, the biggest acknowledge goes to the only person that makes it all worth it, Susana Caetano.

"*A life worth living, is a life worth sharing*", obrigado.
Hélder Diniz

Contents

List of Figures xi

List of Tables xiii

List of Source Code xv

List of Acronyms xvii

1 Introduction 1
1.1 Context 1
1.1.1 Stakeholder preface 2
1.2 Problem 2
1.3 Goals 3
1.4 Approach 4
1.5 Dissertation structure 5

2 Contextualization 7
2.1 Operational Context 7
2.1.1 Concession 7
2.1.2 Toll collection 8
2.2 Globalvia Toll Collection 9
2.2.1 Current System 10
2.2.1.1 Optical Character Recognition component 10
2.2.1.2 Globalvia Lane Controller 10
2.2.1.3 Globalvia Plaza Controller 11
2.2.1.4 Globalvia Remote Controller 12
2.2.1.5 Back Office component 12
2.2.1.6 Concession Back Office Frontend 13
2.3 Toll Collection Solutions 13
2.3.1 Ascendi 13
2.3.2 A-to-Be by Brisa 15
2.3.3 Comparative Analysis 16

3 Value Analysis 19
3.1 Innovation Process 19
3.1.1 New Concept Development Model 20
3.2 Solution Value 22
3.2.1 Value for the Customer and Perceived Value 22
3.2.2 Value Proposition 23
3.3 Quality Function Development 23
3.3.1 Contextualization 23
3.3.2 Conception 26

4 State Of The Art 29
4.1 Software Architecture Patterns . . . 29
4.1.1 Monolithic Architecture . . . 29
4.1.2 Micro-services Architecture . . . 30
4.1.3 Event Driven Architecture . . . 32
4.2 Infrastructure . . . 33
4.2.1 Cloud Computing . . . 33
4.2.2 Software delivery . . . 34
4.2.3 Scalability . . . 35
4.2.4 Infrastructre as Code . . . 36
4.3 Multi-tenancy . . . 37
4.3.1 Single Instance . . . 38
4.3.2 Single Configurable Instance . . . 38
4.3.3 Multiple Instances . . . 38

5 Analysis 41
5.1 Requirements . . . 41
5.1.1 Non-functional requirements . . . 41
5.1.2 Functional requirements . . . 42
5.1.2.1 BackOffice Operator use cases . . . 43
5.1.2.2 Concession Manager use cases . . . 44
5.1.2.3 Manual Reviewer use cases . . . 45
5.1.2.4 Commercial Reviewer use cases . . . 46
5.1.2.5 Motorway User use cases . . . 47
5.1.2.6 Issuer use cases . . . 47
5.1.2.7 System use cases . . . 48
5.2 Domain . . . 48
5.2.1 Domain model . . . 48
5.2.2 Concepts . . . 50
5.2.2.1 Concession . . . 50
5.2.2.2 Transaction . . . 51
5.2.2.3 Transaction Status . . . 51
5.2.2.4 Photo . . . 52
5.2.2.5 Payment . . . 53
5.2.2.6 Issuer . . . 53
5.2.2.7 Validation . . . 53

6 Architecture 55
6.1 Preface . . . 55
6.2 Logical view . . . 57
6.3 Process view . . . 61
6.4 Development view . . . 62
6.5 Physical view . . . 64
6.6 Use case view . . . 65
6.6.1 MU01 - Use an automatic toll with OBU . . . 65
6.6.2 I01 - Configure a periodicity to receive the Transactions that used the motorway . . . 69
6.6.3 S01 - Trigger issuer periodic batch dispatch . . . 70
6.6.4 MR01 - List all Transactions for Manual Review . . . 71

6.6.5 MR02 - Edit a Transaction in Manual Review to Collectable 73
6.7 Software Delivery 74

7 Implementation 75
7.1 Infrastructure 75
7.1.1 Infrastructure as Code 75
7.1.2 Cluster 77
7.1.2.1 Virtual Private Cloud (VPC) 78
7.1.2.2 Node Groups 79
7.1.2.3 Identity and Access Management (IAM) 79
7.1.2.4 Elastic Kubernetes Service (EKS) 80
7.1.3 Databases 80
7.1.4 Message Queues 81
7.1.5 Storage Components 82
7.2 Micro-services 82
7.2.1 Configuration 83
7.2.2 Event Messages 84
7.2.3 Message Broker 85
7.2.4 Provisioning 86
7.2.4.1 Deployment 86
7.2.4.2 Service 87
7.2.4.3 Ingress 87
7.2.5 Pipeline 88
7.2.5.1 Build and Testing 88
7.2.5.2 Deployment 89
7.2.6 Multi-tenancy 89
7.2.6.1 Configuration 90
7.2.6.2 Tenant Context 90
7.2.6.3 Communication 91
7.2.6.4 Database Selection 92
7.3 API Gateway 93
7.4 API Composition 94
7.4.1 Schema 95
7.4.2 Query 95
7.4.3 Mutation 96

8 Evaluation 99
8.1 Experimentation and Testing 99
8.1.1 Measurements 99
8.1.2 Hypotheses 100
8.1.3 Methodology 100
8.2 Results 101
8.2.1 Technical Quality 101
8.2.2 Isolated Load Tests 105
8.2.3 Production Monitoring 107

9 Conclusion 111
9.1 Critical Analysis 111
9.2 Future Work 112

Bibliography **113**

A Quality Function Deployment **117**

B Complete domain model **119**

C Class Mismatch validation decision mechanism **121**

D Kubernetes Terminology **123**

E SonarQube Code Analysis **125**

List of Figures

2.1 Abstraction of a Globalvia concession toll system 10
2.2 Ascendi toll collection system architecture - Source [17] 14

3.1 The three phases of the innovation process - Source [23] 19
3.2 New Concept Development (NCD) Model - Source [24] 20
3.3 Strength symbol pattern for the relationship matrix - Source [28] 25
3.4 Strength symbol pattern for the conflict matrix - Source [28] 25
3.5 Relationship and conflicts matrix 27

4.1 Standard build and delivery pipeline - source [41] 35
4.2 Scale cube model - adapted from [44] 36

5.1 Use case diagram for BackOffice Operator 43
5.2 Use case diagram for Concession Manager 44
5.3 Use case diagram for Manual Reviewer 45
5.4 Use case diagram for Commercial Reviewer 46
5.5 Use case diagram for Motorway User 47
5.6 Use case diagram for Issuer 47
5.7 Use case diagram for System 48
5.8 Operational domain model 49
5.9 Commercial domain model 49
5.10 Operational status state diagram 52
5.11 Commercial status state diagram 52

6.1 4+1 architectural view model - Source [53] 57
6.2 Component diagram 58
6.3 Process view of a micro-service 62
6.4 Development view of a micro-service 62
6.5 Deployment diagram of the multi-concession platform 64
6.6 Architecturally relevant use case diagram 65
6.7 Sequence diagram for MU01 use case 66
6.8 Sequence diagram for MU01 use case - validation processing 67
6.9 Sequence diagram for MU01 use case - Transaction Flow 68
6.10 Sequence diagram for I01 use case 69
6.11 Sequence diagram for S01 use case 71
6.12 Sequence diagram for MR01 use case 72
6.13 Sequence diagram for MR02 use case 73
6.14 Continuous Integration (CI) & Continuous Delivery (CD) pipeline design . . 74

7.1 Amazon Web Services (AWS) deployment diagram 78
7.2 Activity diagram of micro-services and config-server interactions 84
7.3 Ambassador registered endpoints table 94

7.4 GraphQL query client invocation and result 96
7.5 GraphQL mutation client invocation and result 97

8.1 SonarQube code analysis: Validation micro-service 104
8.2 Load test for Transaction creation: response time percentiles over time . . 105
8.3 Response time average values for Transaction creation 107
8.4 Average age value for Transaction creation message event 108
8.5 Response time average values for Transaction validation 109

A.1 Quality Function Deployment system 117

B.1 Complete domain model 119

E.1 SonarQube code analysis: Transaction micro-service 125
E.2 SonarQube code analysis: Collection micro-service 126
E.3 SonarQube code analysis: Issuer micro-service 126
E.4 SonarQube code analysis: Photo micro-service 127
E.5 SonarQube code analysis: Proxy micro-service 127
E.6 SonarQube code analysis: Scheduler micro-service 128
E.7 SonarQube code analysis: Tenant micro-service 128
E.8 SonarQube code analysis: Exempt micro-service 129
E.9 SonarQube code analysis: GraphQL component 129

List of Tables

2.1 Comparative analysis of toll collection systems 17

3.1 Benefits and sacrifices of the multi-concession system 23

6.1 Pros and Cons of each considered architecture 56

8.1 SonarQube code analysis summary - average values 104
8.2 Load test hardware specifications 105
8.3 Load test for Transaction creation: executions and response time 105
8.4 Load test for Transaction creation: response times for two cases 106
8.5 Load test for Transaction creation: response times variance 106
8.6 Response time average for Transaction creation: summary 107
8.7 Average age value for Transaction creation message event: summary . . . 108
8.8 Response time average for Transaction validation: summary 109

C.1 Decision mechanism of the class mismatch validation 121

D.1 Kubernetes fundamental terminology - adapted from [73] 123

List of Source Code

7.1 Example of an execution plan produced by Terraform 76
7.2 Terraform provider configuration for Amazon Web Services 77
7.3 Terraform backend configuration using Amazon S3 77
7.4 Amazon Virtual Private Cloud configuration 78
7.5 Node groups configuration 79
7.6 Amazon Identity and Access Management configuration 79
7.7 Elastic Kubernetes Service configuration 80
7.8 Relational Database Service instance, for concession Transmontana, OBO domain 80
7.9 Network peering configuration for concession Transmontana 81
7.10 Message queue configuration 81
7.11 Dead-letter message queue configuration 81
7.12 S3 bucket configuration 82
7.13 Spring Cloud Config micro-service configuration 84
7.14 Protocol Buffer message for Scheduler registration 84
7.15 Scheduler registration consumer class 85
7.16 Dockerfile for Validation micro-service 86
7.17 Kubernetes deployment snippet for Validation micro-service 86
7.18 Kubernetes service snippet for Validation micro-service 87
7.19 Kubernetes ingress snippet for Validation micro-service 88
7.20 Jenkinsfile - build & testing stage 88
7.21 Jenkinsfile - deployment stage - preparation 89
7.22 Jenkinsfile - deployment stage - update 89
7.23 Multi-tenancy configurations 90
7.24 Tenant context holder class 90
7.25 HTTP multi-tenant interceptor 91
7.26 HTTP multi-tenant interceptor 92
7.27 MultitenancyDatabaseConfiguration implementation 93
7.28 CurrentTenantIdentifierResolverImpl implementation 93
7.29 DataSourceMultiTenantConnectionProvider implementation 93
7.30 Endpoint registration for Validation micro-service 94
7.31 Transaction type schema representations 95
7.32 Query type representation for Transaction requests 96
7.33 Mutation type representation for Transaction 96
8.1 Validation functionality unit test 101
8.2 Scheduler event registration integration test 103

List of Acronyms

API Application Programming Interface.
ATPM Automatic Toll Payment Machine.
AWS Amazon Web Services.

CBO Commercial Back Office.
CD Continuous Delivery.
CI Continuous Integration.
CQRS Command Query Responsibility Segregation.
CRUD Create, Read, Update, Delete.

DDD Domain Driven Design.
DSRC Dedicated Short-Range Communications.
DSRM Design Science Research Methodology.
DTO Data Transfer Object.

EC2 Elastic Compute Cloud.
ECR Elastic Container Registry.
EKS Elastic Kubernetes Service.
ESB Enterprise Service Bus.

FaaS Function as a Service.
FEE Fuzzy Front End.

GLC Globalvia Lane Controller.
GPC Globalvia Plaza Controller.

HTML HyperText Markup Language.
HTTP HyperText Transfer Protocol.

IaaS Infrastructure as a Service.
IaC Infrastructure as Code.
IAM Identity and Access Management.

JDBC Java Database Connectivity.
JMS Java Message Service.
JSON JavaScript Object Notation.

MLFF Multi-Lane-Free-Flow.

NCD New Concept Development.
NPD New Product Development.

OBO Operational Back Office.
OBU On-Board Unit.
OCR Optical Character Recognition.

PaaS Platform as a Service.
PPP Public-Private Partnership.

QFD Quality Function Deployment.

RDS Relational Database Service.
REST Representational State Transfer.
RFID Radio Frequency Identification Systems.
RSE Road Side Equipment.

S3 Simple Storage Service.
SaaS Software as a Service.
SDL Schema Definition Language.
SQS Simple Queue System.
SSH Secure Shell.
SSL Secure Sockets Layer.

TLS Transport Layer Security.

UML Unified Modeling Language.
UUID Universally Unique Identifier.

VPC Virtual Private Cloud.
VPN Virtual Private Network.

Chapter 1

Introduction

In this first chapter, a contextualization and a description of the problem at hand is provided, together with the adjacent goals to achieve and the research approach, concluding the chapter with a brief description of the document structure.

1.1 Context

The main focus of this document is to describe the research process, design and implementation of a cloud-based, scalable, multi-concession toll collection and validation system. Designed with standards of a production-ready system, it is meant to be used in Globalvia's motorway toll collection infrastructure, integrating with part of the current system in place. Currently, Globalvia's concessions with toll motorways are supporting two distinct passage modes: passage with manual payment and electronic passage with the support of a mounted On-Board Unit (OBU). The first one requires the user to stop his commute temporarily, in order to pay the toll manually to proceed, while the second one requires the user to have a valid, and recognized by the concession, OBU hardware installed on the used vehicle and a Road Side Equipment (RSE) to support Radio Frequency Identification Systems (RFID) communication.

The current toll collection system is a set of distributed software and hardware components, which are variant between concessions, with three distinct layers for each type of data and process. The data reading layer, the first layer of the system, composed by the RSE and an Optical Character Recognition (OCR) system is responsible for reading and building the necessary information of the passage, in order to create the system representation concept named Transaction. This information is consolidated in the Operational Back Office (OBO) domain and is then dispatched to the second layer of the system, the data processing layer, which is represented by a software component that will validate, process and store the transaction information. The third layer of the system, which is present on the same system as the data processing layer, is named data handling layer and it is responsible for the toll collection workflows of the transactions that have been successfully validated and deemed collectable by the previous layer. This layer is consolidated in the Commercial Back Office (CBO) domain.

There are specific scenarios where the transaction information is not validated by the data processing layer and its information takes an alternative flow to a *manual review* status. When a transaction is found on that status, a specialized operator will evaluate every piece of information for that transaction and decide if the information is correct or that it should be re-evaluated by the system upon some corrections or even cancelled. The information

evaluated by the operator is the detected vehicle class, OBU identifier, toll rate applied, time of day, licence plate and others.

The work and research developed is submitted as fulfillment for the Master Thesis of the Informatics Engineering Course program at Polytechnic of Porto - School of Engineering and has been enabled in virtue of a partnership established between Globalvia Transmontana and Mindera Software Craft.

1.1.1 Stakeholder preface

Globalvia[1] is an international engineering consulting company, established in 2007 in Madrid, Spain, specialized in the development of transportation engineering and infrastructure, with special focus in motorway maintenance and toll collecting. It is currently present on several countries, in the form of concessions, such as Andorra, Chile, Costa Rica, Ireland, Mexico, Portugal, Spain and United States of America. Globalvia Transmontana[2] is a concession located in Portugal and its main technological focus is to be the pioneer in the motorway technology revolution by contributing to the evolution of the state of art in software development and autonomous driving. Considering the large amount of concessions, Globalvia is focusing in improving the current software by creating a highly available and scalable cloud system for toll collection and transaction validation to be used by all the concessions in the group.

Mindera Software Craft[3] is an international self-organisation company specialized in software engineering and product designing. Established in Porto, Portugal, it has grown into several locations and is currently present in England, India, Portugal and United States of America. Mindera builds internal products as well as partner products, while following its technological focus grounded in engineering high performance, resilient and scalable software system to fuel businesses across locations, with special focus in web and mobile systems.

1.2 Problem

The current toll collection system suffers from the early naive decisions of being built to a very specific domain and requirements, with little to no plan for the natural requirement evolution, future expansions or usage growth. The application flows and logic are heavily loaded with concession business rules, which causes the necessity of exhaustive system overhaul and refactoring to be applied when trying to deploy it for different concessions. This is causing numerous derivations of the original system, which can be considered as subsystems to a degree, considering that they're completely transformed into another system for each concession, endangering the system maintainability altogether, while at the same time, jeopardizing any possibility of a multi-concession system.

The number of derivations of the original software, per each concession, is specially concerning for the vision of Globalvia lead group. The existence of more systems means the need of constantly hiring engineers to maintain the systems, leaving little no room for improvements and new requirement implementations. The development tend to stall in most

[1] https://globalvia.com
[2] https://aetransmontana.pt
[3] https://mindera.com

cases, as specialized engineering labour tend to be scarce in some of the operating countries. Furthermore, an increased difficulty is imposed due to the business's nature being very domain-driven, which poses as a down-side for newly signed engineers.

Another problem of the system currently in place, is that the application domain concepts are mixed in the same context and interfaces. As aforementioned, the system is composed by three different layered with two different, but similar, domain models, OBO and CBO. This is a special concern because the problem is inside a complex domain space and underestimating the design process could lead to a poorly design domain with infringement of good software engineering practices, such as the Domain Driven Design (DDD) strategic design techniques, which claims that "a successful model, large or small, has to be logically consistent throughout, without any contradictory or overlapping definitions" [1].

A separate concern happening with current system is that it generates a considerate amount of transactions that need to be manually reviewed by specialized operators, due to lack of information or even due to the confidence of the OCR reading representing a value lower than usual or being unable to identify the issuer of the OBU. This opens up a possibility to develop systems that validate the information before the operator has to be called for action, reallocating some of the workload. Although many problems could be considered about this topic, such as user fraud, the correctness evaluation of vehicle positioning system techniques [2], problems with infrared readers [3] or even issues with RFID techniques [4], this dissertation will focus mainly on the software side of the problem for information validation.

1.3 Goals

The main objective of the project and research of this dissertation, is to plan and design a proof of concept, multi-concession toll collection Software as a Service (SaaS) system, following a cloud-based scalable architecture, with regard to Globalvia's vision to use the same system for all their concessions' infrastructure. In order to tackle the aforementioned problems and the stakeholder vision, the following goals are established for this project:

- Plan and design a multi-concession toll collection SaaS system;
- Plan and design a cloud-based scalable architecture for the concession toll collection system with Continuous Integration (CI) and Continuous Delivery (CD), also known as CI & CD;
- Implement a software level Validation techniques;
- Implement the cloud infrastructure of the aforementioned system;
- Implement the CI & CD process.

The achievement of the prior presented goals will lead to the establishment of a full team of developers to expand and implement the fully featured system, following the defined architecture and CI & CD process.

1.4 Approach

Information System solutions are usually based on well established research methodologies to define the path to investigate and achieve knowledge about the inherent problem. Typically, the study's methodology settles on one of the two processes: behavior science or design science. The first one "(...) seeks to develop and justify theories that explain or predict organizational and human phenomena (...)", [5] while the latter, according to the same source, "(...) a problem solving paradigm. It seeks to create innovations (...) and use of information systems can be effectively and efficiently accomplished".

The carried research on this dissertation is applied in solving a recurring real world problem, in the business context of Globalvia. This enables the conclusion that the work for this study conforms with the methodology of design science, categorically the Design Science Research Methodology (DSRM). DSRM is a framework specially designed to provide a set of practices, principles and procedures to aid the implementation of design science research. This methodology fits on the organizational context of the problem and will be used as the foundation for the development of this dissertation. It is composed by 6 activities [6]:

1. Problem identification and motivation: action of defining the specific research topic with addition of a justification of the value of the solution, in order to understand the reasoning behind the researcher's understanding of the problem and to accept the results. The description of the problem concerned is present on the section 1.2 and its value analysis are present on chapter 3.

2. Define the objectives for a solution: definition of the feasible objectives of a solution from the concerned problem. The objectives can be classified as quantitative or qualitative and should be development with knowledge of the state of the art related to the problem and other solutions. The goals related to this problem are present on section 1.3.

3. Design and development: activity of designing and creating the artifact, which includes determining the artifact's functionality and architecture, leading to the development of the designed artifact. This activity is detailed on chapters 5 and 6, for the analysis and design, while for the implementation it is detailed on chapter 7.

4. Demonstration: process of demonstrating if the use of the artifact can solve the problem, which could be achieved through experimentation, simulation or other similar activities. In this project the activity is performed in Globalvia's environment.

5. Evaluation: observation and measuring on how the artifact solves the problem in its entirety or parts of it. This activity can be done continuously for each time the previous activity was executed, until the researcher is satisfied with the results for the stated objectives. The evaluation of this project is present on chapter 8.

6. Communication: communicate the problem, solution and the artifact to other researchers or other relevant entities, such as professionals. This goal is achieved by the presentation of the work to Globalvia stakeholders and an academic evaluation committee.

1.5 Dissertation structure

This section is focused on providing an overview of the present document, in order to introduce a general idea of all approached aspects of this dissertation, which is divided into nine chapters.

Introduction is the first chapter, and it is focused on providing a brief contextualization of the dissertation, as well as describing the problem, goals and the approach followed during development and research.

The next chapter *Contextualization*, provides a detailed description of the key concepts grounded to motorway operation and management, in order to provide a broader knowledge of the business to correlate with the problem in hand. Additionally, a comparative analysis between toll collection solutions is demonstrated.

It is followed by the *Value Analysis* chapter, which provides a theoretical approach on how an opportunity is identified, the value for the customer, the perceived value and even a value proposition, by using an organized and analytical process of evaluation.

The next chapter *State Of The Art*, analyzes the literature regarding the concepts of the problem in hand, such as architectural styles, infrastructure good practices and concepts, and multi-tenancy approaches.

The fifth chapter, *Analysis*, is the gateway to the technical concepts of this dissertation. It demonstrates the thought processes around developing functional and non-functional requirements, which causes an identification of the use cases of this system, as well as its domain concepts.

It is followed by the *Architecture* chapter, which establishes the bridge between requirement engineering and a proposal of the architecture to be implemented. It demonstrates the extensive architectural design process, as well as providing a detailed description of architecturally relevant use cases.

The next chapter *Implementation*, is concerned about the most technical component of this dissertation. It demonstrates the rationale behind technical decisions, as well as showcasing a detailed view of the lead implementation of the system.

Chapter *Evaluation* is the follow-up to the implementation process, where the evaluation methodology is presented, as well as its measures, hypotheses and experimentation. Each experimentation and test are detailed, backed up with measures and hypothesis validation.

The last chapter, *Conclusion*, presents a critical analysis to the developed solution, as well as the future work for the multi-concession system.

Chapter 2

Contextualization

In order to understand the project in its entirety, it is important to understand the key concepts grounded on the business domain, as the project described in this dissertation is targeted to a very information-driven business. In this section, key concepts of motorway operation are described in detail, as well as a detailed view of the current system in place.

2.1 Operational Context

2.1.1 Concession

Transportation links and road networks are considered one of the pillars of economic growth and country development, as governments are constantly looking for innovative ways to develop the road networks in order to meet political and social needs. Building new motorways is expensive and governments are often found unable, or reluctant, to include these expenses as fiscal expenditures. Furthermore, physical resources for building infrastructure is scarce and heavy regulated, which has driven the creation of new concepts for the financial and management part of motorway. This has lead to the creation of a new organisation type, grounded on the recourse of private funding, under a sophisticated and highly regulated long-term finance business model, the motorway concession.

A concession is formed by one or more private organisations, specially assembled by a cooperative arrangement between the State, a public entity, and these organisations. The State is the owner of the road network and "grants specific long term rights (...) to construct, overhaul, maintain and operate an infrastructure" [7] and to provide a service to improve the country's capacity to operate in a more efficient manner. The concession is then responsible to provide for the service, in a way that the State decides to, while the concession is responsible to carry out all the road management and operations. In contrast, the asset ownership remains with the concession and revert back to the private organisations at the end of the concession period. This type of partnership is also known as a Public-Private Partnership (PPP) [8], which is usually of a long-term nature.

To understand how a concession is important for the development of road networks, Alain Fayard conducted an analysis of motorway concessions in Europe in 2005 [9], where it was determined that out of 57542 km of European motorway area, 38% is under a concession's management. This represents 52 state-owned concessions and 68 PPP concessions, which lead to the rapid development of road networks all over continent.

2.1.2 Toll collection

Considering the heavy investment by the organisations to build the necessary infrastructure declared on the concession agreement, and the State's unwillingness to invest into the newly formed concession, the revenue stream must be provided directly from the usage of the motorway. The introduction of road tolling policies on road networks, that seek investment return, has become a reliable option for concessions to seek for a stable revenue stream, in order to fund its operations. Applying road toll systems, in the road transport section, seek to reach one, or more, of the following objectives:

- **Tolls as a new and dedicated source of income** [7]:
  The revenues generated in toll collection represent a new source of revenue, which is not tied to the government's annual budget and can be used to directly run the concession, by providing support for construction and maintenance of infrastructure.
- **Tolls to address pay principle and internalizing value** [7]:
  Toll systems are crucial when providing for a sustainable transport policy, as it address the pay principle and enables the increase of value internalization.
- **Tolls as tool for developing road infrastructure** [7]:
  This objective is based on the introduction of tolls systems with the intent of supporting the development of new infrastructure for road networks on the country, or part of the country. It can also enable the distribution of wealth from one region to another.
- **Tolls as tool for private sector development** [7]:
  Tolls can also be seen as a tool for developing the private sector, this is mostly used in the concession scheme aforementioned, where the state seeks for investment in the private section to develop the wanted road network and infrastructure. Introducing road tolls on these roads can allow to ease the investment by providing a stable revenue stream.

With the intent of achieving any of the prior objectives, several toll collection mechanisms were developed. The majority of these mechanisms are grounded on the premise that the state delegates the construction, funding and management of the road to one or more managing organisation. The agreement established between the state and these organisations, leads to the inception of a concession.

Direct road tolling

The direct road tolling mechanism describes the tolling method where the concession directly charges the users of the motorway, in order to payback the investment on the road networks, infrastructure and its maintenance [7].

The value of the charged toll is usually influenced by the distance traveled, the vehicle class and the type of road. This concept is also known as distance-based charge [10]. Additionally to the typical charge over distance and vehicle class, other variants could be applied to its price, such as time of day, day of week, variation of road construction cost, traffic congestion related variables, or even loyalty programs could influence the price of the toll.

The implementation of the direct road toll mechanism is made possible by two different methods: manual tolling and electronic tolling. Manual tolling requires the users of the motorway to stop their vehicle in order to pay the toll. This brings several constraints related to traffic flow, as well as the need of more tool booths in order not to stall the

traffic completely. As for the second method, electronic tolling, it requires the users of the motorway to lower the speed when passing through the tool booths and to carry an On-Board Unit (OBU) in their vehicles, which traffic-wise is an advantage. However, this creates the need of adding support and maintenance of these OBU devices, which is often outsourced to other organisations to issue the devices. These devices typically work over Radio Frequency Identification Systems (RFID).

The most common implementation of direct road tolling, in Europe, is a mixed implementation of manual and electronic tolling [11].

Indirect road tolling

The indirect road tolling mechanism is a method where the toll is indirectly charged to the users of the motorway, in the form of a subscription. Its most famous form is the *vignette*, by paying a fee to the State. The payment for the *vignette* happens in a yearly manner, in most cases [12], and the State will then distribute the proceedings of these fees for all the concessions, in order to promote infrastructure improvements.

The toll charge rate is influenced by the territory, time of year, availability of road networks and in the type of pay in installments, remunerated for to the concession. This charging concept is also known as time-based charge [13].

Shadow tolling

The shadow tolling mechanism describes the tolling method in which the paid tolling amounts are based on utilisation of the road networks, such as the number of vehicles, and on the performance of the concession, while providing for the infrastructure service [14]. This means that the concession does not collect any toll amount from the user directly, but does receive its revenue stream from the State instead, which comes "all taxpayers, users and non-users of these roads" [15]. This mechanism is similar to indirect road tolling, however it is funded from the tax payers fund, instead of requesting an explicit payment to the users, like a *vignette*.

The main goal of this toll collection system, is to attract private organisations to invest into a business that requires a considerate amount of initial investment with low return, while at the same time, being able to invest into the social development branch of the country, by promoting the use of road networks and providing toll-free motorways.

2.2 Globalvia Toll Collection

As previously mentioned in section 1.1.1, Globalvia is composed by several concessions in different continents, with several types of direct tolling, indirect tolling and even mixed type tolling. The scope of this dissertation is focused in concessions with direct and mixed toll collection capabilities, due to their nature of depending on software infrastructure, therefore, this section describes the current system in place for mixed tolling concessions, as well as some of the key concepts.

2.2.1 Current System

The current system in place is different for each concession, which is causing deviations of the original idea and creating numerous sub-systems with different requirements and structures. Positively, each sub-system's architecture are very alike, due to the fact that there's an existing hard dependency of Road Side Equipment (RSE) that need to exist on each toll gantry and lane. However, apart from the RSE and the software present on the toll gantry - named G-Toll - the following user and operator backoffices are different for each concession. Picture 2.1 shows an abstract component diagram, following the Unified Modeling Language (UML) notation, of the Globalvia Toll Collection system used in mixed tolling concessions.

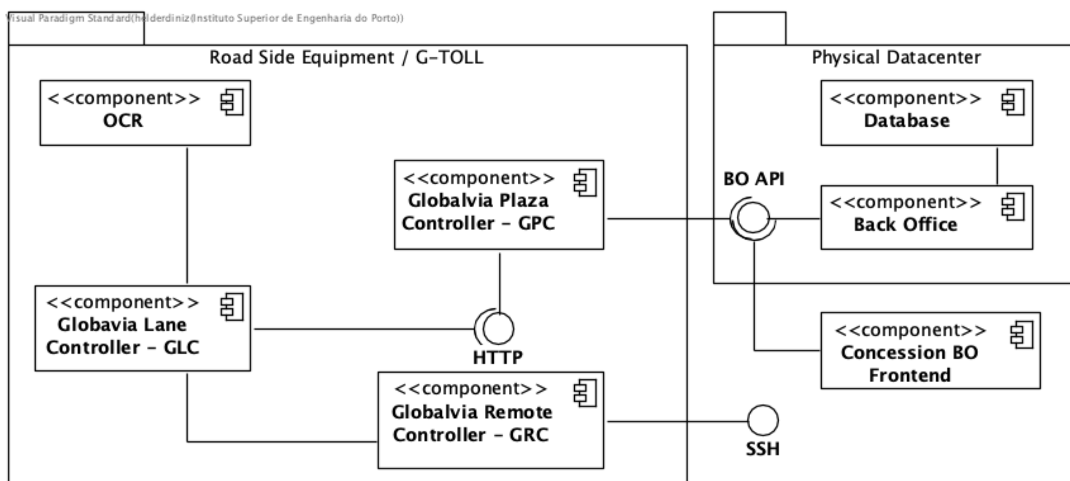

Figure 2.1: Abstraction of a Globalvia concession toll system

The system represented above, is physically separated into two layers, one layer being the RSE and G-Toll system, which is physically located on the motorway's toll gantry infrastructure. While the other layer, is the back office and database component, which is located in a physical data center, in a location defined by the concession.

2.2.1.1 Optical Character Recognition component

The Optical Character Recognition (OCR) component is responsible for capturing pictures of every vehicle that passes on the installed lane of the toll gantry, and using these photos apply character recognition functions in order to identify the vehicle's licence plate. The recognized information is then stored and used by the remaining components of the system. There is an OCR component for each existing lane in the toll gantry, responsible for its own lane.

2.2.1.2 Globalvia Lane Controller

The lane controller is considered to be the main component of the G-Toll system, as it provides connectivity for all other components to provide for their full functionality. Similarly to the component described above, an instance of the Globalvia Lane Controller (GLC) is

installed in each existing lane of the toll gantry. Moreover, it is constantly communicating with OCR component, in order to retrieve the captured images and recognized information of the passing vehicle, in order to attach the captured images and interpreted information, into the transaction details.

This component is responsible for controlling the lane and supports the two methods of direct tolling, manual and electronic. On the manual tolling method, the lane controller and the OCR component will gather the information for the toll booth operator and will only allow the vehicle to pass on demand of the operator. For the electronic tolling method, the lane controller and OCR will behave differently if the toll has a gate or not. Considering the toll gate option firstly, the lane controller is responsible for controlling the toll gate and it only opens when the needed information has been gathered and validated. Conversely, when the lane does not have a toll gate, the lane controller will notify other components of the system that a specific transaction is invalid and a specialized operator will take action as needed.

In addition to the lane controlling functionality, this component will also, periodically, dispatch every passing vehicle's information to the plaza (gantry) controller component, by using HyperText Transfer Protocol (HTTP) communication, as well as generate alerts when unforeseen actions happen.

2.2.1.3 Globalvia Plaza Controller

The Globalvia Plaza Controller (GPC) component is responsible for managing the toll gantry infrastructure, as well as being the exit point of all transactions on the G-Toll system for the back office and the entry point for the toll road operators. This component works as an orchestrator for the toll gantry, which means that it will be managing every GLC that the toll supports by providing lane configurations, as well as activating the component itself. GPC offers the following functionality:

- GLC component configuration;
- Transaction information dispatching for back office;
- Provide connection between the lane's call for support from customers and specialized operators;
- Lane monitoring services, such as alert management and remote control;
- Image storing;
- Video streaming.

The GLC components configuration happen by executing deployment actions for the physical infrastructure on the toll gantry, where several configurations can be set, such as lane state, toll type (manual or electronic), rate to display and others.

The dispatching functionality, of the transaction information originated in the GLC and OCR, is the exit point of the system, which happens when the gantry controller receives communication from the GLC. The communication is executed by communicating with the back office Application Programming Interface (API), sending all the information generated on the passage of the vehicle on the toll gantry. This information contains the vehicle data, payment data (when applied), passage data and passage timestamp.

The lane monitoring functionality is designed to notify the concession about problems in the lane and toll gantry. The generic supported alerts are included in the range of external and internal communication problems, auditing alerts, problems in classifying vehicles, fraud attempt or even devices alert, such as OCR being unable to capture images. This monitoring service is also bundled with image storing and video streaming, in order to allow the specialized operators to investigate when problems arise.

2.2.1.4 Globalvia Remote Controller

The remote controller component allows for the GLC component to be controlled remotely, over a Secure Shell (SSH) connection. This component was created as a failover for the existing remote access in GPC and allows direct access to a specific GLC, in which the operator will be able to view and control the lane, by accessing contextual information of the passing vehicles, as well as using the installed camera on the lane for video streaming, similarly to the lane monitoring services existing on the GPC.

2.2.1.5 Back Office component

Outside of the RSE infrastructure, the back office component is a full fledged, layered, monolith application, built to process all the information originated in the RSE layer. This component acts as a backend for the system and provides a Representational State Transfer (REST) API that produces data to be read by the concession frontend component, as described in the following section 2.2.1.6, as well as consuming data from the GPC component, for persisting the transactions information, originated in the RSE layer. It contains two layers of data processing and domain, the data processing layer, which consolidates the Operational Back Office (OBO) domain, and the data processing layer, which handles the Commercial Back Office (CBO) domain. This component handles the transaction information in order to orchestrate the its state and lifespan throughout the system, over the course of the two domains in the application.

The data processing layer is the first layer to operate, when information is consumed from the REST API triggered by the GPC component. Grounded on the OBO domain, this layer is responsible to validate the information received and evaluate if the transaction could be deemed as collectable, or if it will stay on the OBO domain. A transaction is classified as an operational transaction when it has no payment associated, or the user was detected as an exempt, such as police officers on duty or fire department vehicles. If the information is considered invalid or suspicious, it is transitioned to a state of *manual review*, where specialized operators will evaluate all the information and make a decision about its next state.

The data handling layer is the second layer to operate on the transaction information, it is consolidated in the CBO domain. It is invoked by the data processing layer when the transaction state is set to *collectable*, classified as a commercial transaction. This state is achieved in the system when the information has been properly validated and if the transaction has payment information to charge, such as credit cards authorizations or OBU. The system will then gather all *collectable* transactions and charge them using to the appropriate user by using the existing integrations of banks, or OBU issuers, payment gateway.

Additionally, the back office component uses the database component to store all the information it needs, which typically is represented by a relational database, such as MySQL[1].

2.2.1.6 Concession Back Office Frontend

Used by concession managers, operators and finance specialists, the concession back office frontend component is responsible for providing filterable views of operational and commercial transactions, concession details and payment data. This component is also used by the specialized operators to evaluate the transaction whose state is set on *manual review* due to failed validations or inconsistent data, deemed earlier by the system.

This component is distributed and used as an on-premise software, which is installed locally on several computers in the network of the concession, making it only usable under a Virtual Private Network (VPN), in order to safely connect to the back office component located in the data center.

2.3 Toll Collection Solutions

The evidence suggests that the toll collection business is a highly competitive and lucrative target market, mostly due to its nature of long-term agreements when establishing PPP agreements for new concessions. The correlation between the competitive market and its competitors, is the constant strive for infrastructure improvement, where in the last couple of years, the main focus has been software improvements.

With regard to aid the decision making of this dissertation, it is important to study and analyse existing similar solutions for toll collection, in order to determine their advantages, disadvantages and main differences. In order to induce a comparative analysis, a realistic amount of sample data must be gathered. With this in mind, the studied solutions alternatives are Globalvia's main competitors in the Portuguese market.

2.3.1 Ascendi

Ascendi[2] is a Portuguese organisation founded in 1999, specialized in constructing, operating, maintaining and managing the concession of motorway infrastructure and toll collection systems. The organisation is currently operating 7 concessions in Portugal and is responsible for motorway maintenance and construction, as well as the operational and commercial software for toll collection in all the concessions [16].

Ascendi's toll collection system is a full fledged system layered in three different tiers: RSE & OBO, CBO and Enterprise Corporate Back Office. The system supports toll collection for manual and electronic tolling, as well as an emerging system in Portugal, Multi-Lane-Free-Flow (MLFF) [16] tolling.

[1] https://mysql.com/
[2] https://ascendi.pt

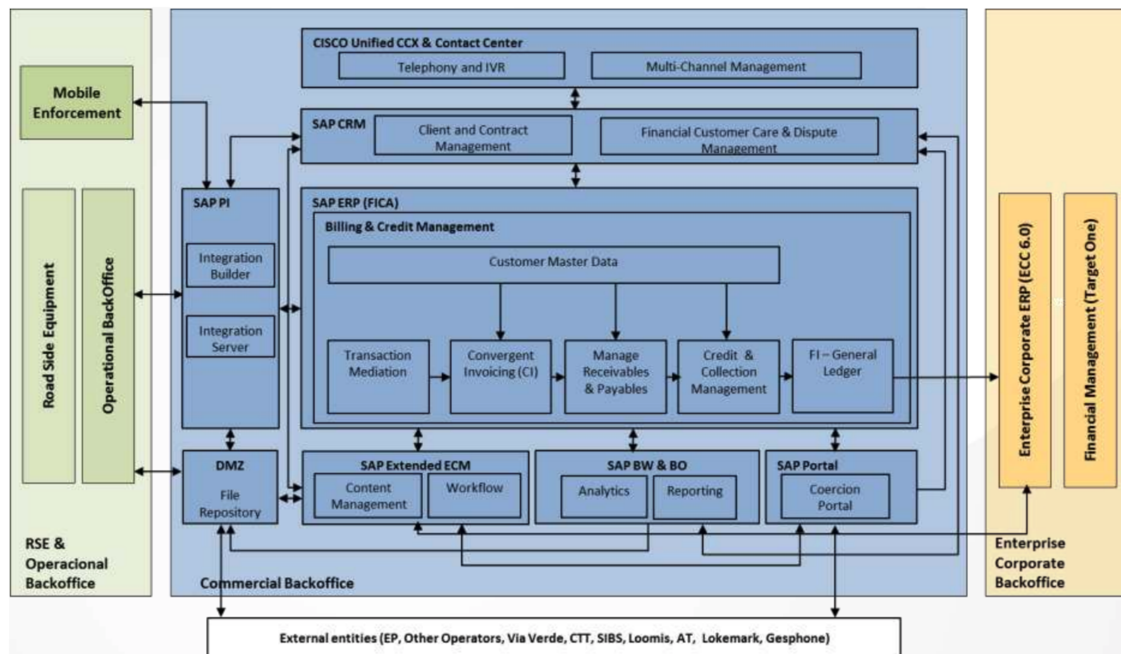

Figure 2.2: Ascendi toll collection system architecture - Source [17]

According to Ascendi's system architecture representation [17], the RSE component offers the following functionality:

- Vehicle passage and classification detection;
- OBU interpretation;
- Capture of context images;
- Automatic license plate recognition by OCR techniques.

While for the OBO component, the offered functionality is [17]:

- Support of Dedicated Short-Range Communications (DSRC) and RFID technology;
- Transaction information validation;
- Second level OCR;
- Image review;
- Trip aggregation engine;
- Automatic charge calculation.

Lastly, the CBO component offers the following functionality [17]:

- Account management;
- Transaction collection;
- Billing and notice issuing;
- Automatic payment processing;
- External interface.

Although the RSE and OBO components are mixed in the same layer, the OBO and CBO components are segregated, which offers versatility and adaptation to each domain's language needs. Furthermore, the figure 2.2 shows that the CBO component is tightly coupled with SAP[3] enterprise systems which could cause a ripple effect of changes in the whole system, if the integration ever needs to be changed.

Considering the multi-concession functionality, Ascendi makes no mention of offering multi-concession support and doesn't include cloud infrastructure in their architecture, which delegates the need of maintenance effort of physical data centers for each concession system.

2.3.2 A-to-Be by Brisa

A-to-Be[4] is a Portuguese organisation created in 2009, by Brisa, originally intended to lead Brisa's innovation process and technology development. Re-branded from *Brisa Inovação e Tecnologia* in 2017 [18], A-to-Be is specialized in motorway technology investigation, software development and maintenance for toll collection systems, while Brisa is responsible for the construction and operation of the motorway infrastructure. A-to-Be motorway solutions are split between component bundles, which are targeted for different concession needs. The existing product components are `MoveBeyond`, `LinkBeyond` and `Atlas`.

The `MoveBeyond` product is aimed at providing a mobility platform for road operations, which is then bundled in three different modular components, for tolling back offices, mobility services platforms and dematerialized satellite-based tolling. The `MoveBeyond Tolling Backoffice` is a component specially designed to manage operational and commercial transactions in tolling systems, layered with two different tiers: OBO and CBO.

The OBO layer offers the following functionality [19]:

- Transaction information validation;
- Trip building engine;
- Image review;
- Automatic charge calculation.

While for the CBO layer, the offered functionality is [19]:

- Transaction collection;
- User billing issuing;
- Automatic payment processing;
- Dynamic pricing;
- Account management;
- Document management;
- Dunning management;
- External interface.

[3] `https://sap.com/`

[4] `https://a-to-be.com`

The `LinkBeyond` product is A-to-Be's equivalent of the RSE layer evaluated earlier in Globalvia and Ascendi systems. This component handles all the RSE, such as toll gantry, lanes, cameras and toll gates [20]. Directed for connecting the road with other software components, this product supports manual tolling, electronic tolling, MLFF tolling and automatic tolling, while offering the following technology:

- Vehicle passage and classification detection;
- OBU interpretation;
- Capture of vehicle images;
- Automatic license plate recognition by using OCR techniques.

Lastly, the `Atlas` product is an operational management platform, road monitoring and a complete view of the operational scenarios, aimed at the operational support of a concession. This component can be used by road operators or operational managers in the concession, "whether by taking action on its own or by allowing operators to take preventive and proactive actions" [21].

Considering the prior description of the A-to-Be's products, it's possible to verify that the RSE, OBO and CBO are properly segregated by components, instead of overlapping in domain language or space. There is no mention of multi-concession functionality, however, it is mentioned that the data storage server resources may be shared between concessions.

2.3.3 Comparative Analysis

Succeeding a study and analysis of existing similar solutions on the market, a comparative analysis must be executed in order to understand the contrast and union of the evaluated solutions.

The accomplished study led to conclude that the two main competitors of Globalvia are not planning on building a system to support more than one concession, in order to provide it as a service, or at least stated it publicly. Simultaneously, the analysis has pointed that that Globalvia systems are lacking in a few details, most specifically, the most relevant being the lack of support for MLFF tolling, in the RSE layer.

Table 2.1 describes the accomplished comparison between the current Globalvia, Ascendi and A-to-Be tolling systems. The (x) denotation represents if the system in place supports, or not, the analysed functionality.

Table 2.1: Comparative analysis of toll collection systems

Functionality	**Globalvia**	**Ascendi**	**A-to-Be**
Multi-concession system			
Manual and electronic tolling	x	x	x
MLFF tolling		x	x
Automatic vehicle classification	x	x	x
OBU support	x	x	x
OCR & image capture	x	x	x
OBO support	x	x	x
CBO support	x	x	x
OBO and CBO segregation			x
Cash, Card and OBU collection	x	x	x
Transaction information validation	x	x	x
Payment gateways integration	x	x	x
Data decentralization			

After examining the represented information on the table above, it is possible to conclude that the none of current systems in place support a multi-concession approach or follow data decentralization principles, which are the key objectives of the research present in this dissertation.

Moreover, the support of MLFF tolling mechanism is not present on Globalvia's system, this is mainly caused by the fact that G-Toll was initially developed when the mechanism was still non-existent. However, it is in Globalvia's plan to add support for that mechanism on the RSE layer.

Ultimately, the A-to-Be system is the one that comes on top, due to the broad support of the majority of the wanted features for toll collecting, operational and commercial transaction management, although, it misses the key features grounded on this dissertation goals. Inside its scope, it is possible to conclude that the current Globalvia system is a strong candidate in comparison with its direct competitors.

Chapter 3

Value Analysis

Value Analysis is an organized and analytical process of evaluation and analysis of a concept, product or service. This process seeks to increase the value of a product or service, while at the same time, minimize the cost of production while maintaining the desired quality.

In this chapter, the complete innovation process is described in detail, as well as the value for the customer, perceived value, benefits, sacrifices and a value proposition is provided.

3.1 Innovation Process

The definition of innovation is very broad and it's directly dependent of its contextualization and environment. In a business contextualization, the execution of an innovation could lead to the creation of a new product, a change in the development process, or even a change in an existing product [22]. That execution is made possible by the development of the innovation process.

The innovation process can be applied to a product or a business, and is considered to be branched into three sequential parts: Fuzzy Front End (FEE), the New Product Development (NPD) process and commercialization [23]. This process traces the complete life cycle of an idea since its conception, implementation and its commercialization. Each part is executed sequentially and it's only expected that the commercialization only happens if the value of the idea has been proven, as represented in figure 3.1.

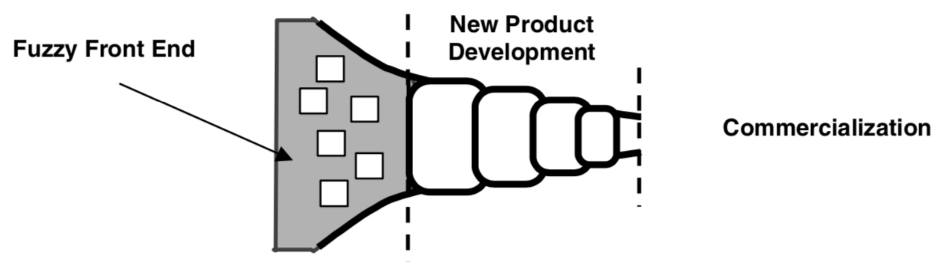

Figure 3.1: The three phases of the innovation process - Source [23]

The FEE part is the initial step to start the process of innovation, its goal is to create a concept based on an business, or idea opportunity and "is generally regarded as one of the greatest opportunities for improvement of the overall innovation process" [23]. In order to reach the initial goal, it is necessary to understand the idea and the existing opportunities,

generate ideas to add value, validate them, select the validated ideas and define the concept to be implemented on the next phase, the NPD.

NPD phase reside in the implementation of the concepts created in the FEE phase. The goal of this phase is to transform the conceptual idea into an actual product and to study the market in order to understand how to reach clients.

Lastly, the commercialization part is focused in selling the constructed product and to reach the targeted audience identified in the previous phase. In this phase, the concept has already been transformed into a product, has been tested, approved and ready to yield value to the user.

3.1.1 New Concept Development Model

The New Concept Development (NCD) model, created by Peter A. Koen, enables the transformation of an opportunity into a concept by evaluating in detail the key elements of the model, in order to understand if the opportunity is able to provide value to the organization [24].

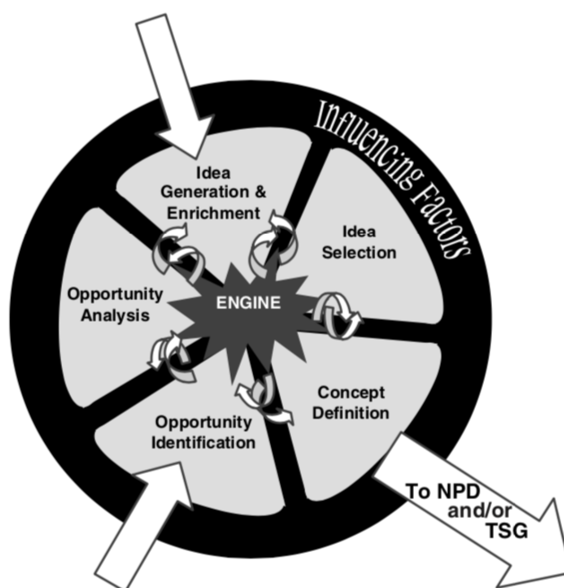

Figure 3.2: NCD Model - Source [24]

As represented by the figure above, the model is composed by three components [24]:

- In the inner area, the five key elements of the front end of innovation;
- In the outer area, the influencing factors of the organization, such as competitors or customers and organization capabilities;
- The engine that is influenced by the culture and leadership of the company, drives the five front end element.

Opportunity Identification

Opportunities can emerge from several ways, such as the need of a new functionality in a given product, or even the creating of a brand new concept or product by a competitor. The

opportunity identification element is where the organization identifies the opportunities that will possibly go after, with the product and "this element is typically driven by the goals of the business" [24].

The solution documented in this dissertation emerged from the problems identified in Globalvia's current system, as aforementioned in section 1.2. With the implementation of a multi-concession system, the problems developed by early architectural decisions, could be tackled to a point that the sub-systems could become unnecessary.

Opportunity Analysis

In the opportunity analysis element, additional information should be gathered with the intent of translating the identified opportunities, on the previous stage, into specific opportunities for the business and to assess if the opportunity is worth ensuing its execution.

The values and vision of Globalvia showcase a constant strive for continuous learning, continuous evolution of the technology in all systems, or processes, and constant improvement of business processes and client support. The company's footprint in the motorway transportation engineering is quite substantial, counting with several concessions in 8 different countries. This is considered both a market strength and a technology one, because the company's infrastructure and technological work is present in every concession. This fits under the analysis method of **strategic framing**, which is labeled as the "determination of how this opportunity fits within the company's market and technology strengths, gaps and threads" [23].

Idea Generation & Enrichment

The idea generation and enrichment element is followed by the executed opportunity study, and it consists in transforming the concepts developed, and the defined opportunity, into a new concrete idea. This process should be iterative and it isn't expected that the idea is perfect at its genesis, but it should be worked on constantly with the purpose of enrichment.

In Globalvia, the idea of having a multi-concession software system for toll collection, transaction validation and billing, has always been something highly anticipated, but recurrently deemed unfeasible, due to very high maintenance and cost of software infrastructure, such as databases and data centers. With the constant rise in popularity of cloud hosted systems, distributed billing services and Platform as a Service (PaaS), it was deemed as an opportunity and a research started with the objective of finding possible solutions for the opportunities identified earlier. Upon the work executed on the research, the following approaches emerged:

- Refactor the current system and subsystems to support a multi-concession approach;
- Implement a brand new multi-concession system, with updated technology and new requirements for Globalvia concessions.

The two approaches are able to tackle the emerged opportunities, however they're on two extremes and the characteristics of each are very different.

Idea Selection

In the idea selection element, the goal is to select the idea to develop, from the set of ideas identified in the previous element, in which it is seen as the idea with the most value to the client, or in this case, Globalvia itself. For this choice, several selection criteria are considered, such as inherent costs, technological risk or even financial return.

The current system in place was developed originally by Globalvia, but with the rapid integration of new concessions into the Globalvia group, it has been manifested that the system is not corresponding to the requirement evolution and the development of subsystems for each concession were heavily favored for outsourcing teams. The current state of the system clearly does not meet the company's vision for the project, therefore the transformation into a multi-concession system is a must. Upon negotiations with the current development teams, it was shown that the idea of refactoring every system wasn't the most motivating objective, while in the other hand, implementing a new system with updated technology and decentralization of hardware was considered highly motivational for stakeholders and developers. Considering the feedback from all interested parts, the idea of implementing a brand new multi-concession system with updated technology and requirements was selected.

Concept Definition

As the final element of the NCD model, the concept definition element highlights that it's important to to select the concept to pursue in order to exit to the implementation state and prove that the opportunity brings value.

There aren't many risks associated with this project, as the current system is working in production, the main objective is to define a set of goals and work over these n iterations without setting deadlines, allowing for an exponential rise of requirements later on. The concept of the new system is expressed as a scalable, distributed and multi-concession system with a special motivation to decrease the need of physical hardware in the new system's architecture.

3.2 Solution Value

3.2.1 Value for the Customer and Perceived Value

Value is not a matter of minimizing cost, but to provide usefulness to an affair. It is a subjective concept interpreted in a distinct way by different components of a given context. Contextualizing the value of a product, the value of itself is focused on the relevance of the functionality, to the client, and how much is the client willing to pay for the product or service.

Reviewing the value for customers of a business it can be represented as a need, interest or even preference. The same product or service, may have different values for different customers, as the value is perceived by the profile and preferences of a specific customer. Moving forward with the idea of the implementation of a new multi-concession system, it is expected that the customer feels an increase in the product's value, as in the experience should be renewed while keeping the same, or more, functionality.

The concept of perceived value is based on the execution of an analysis of the benefits and sacrifices for the client [25]. In order to apply the concept directly on the context of the current project, an analysis of the benefits and sacrifices must be weighted, which is represented on the next section.

The perceived value of a product, aimed at a client, can be described as a contrast of its benefits and sacrifices. The benefits and sacrifices can be classified in a specific scope, such as product, service and relationship. The scope is related to the value based drivers, for

instance the flexibility of a given product is in scope of the service it serves, while the effort and energy allocated is in the relationship scope [26]. The table 3.1 serves the contrast of benefits and sacrifices for the perceived value of the product of this dissertation.

Table 3.1: Benefits and sacrifices of the multi-concession system

Domain \ Scope	Product	Service	Relationship
Benefit	- Operational benefits - Product quality - Customization	- Responsiveness - Scalability - Reliability	- Trust - Supplier's image
Sacrifice	- Monetary costs	- Monetary costs - Training and maintenance costs	- Time / Effort - Human Energy

3.2.2 Value Proposition

A value proposition aims to clarify the motives for which a client shall acquire, or use, a product or service, and demonstrate the added retrieved benefits of its usage. It should be explicit and describe the intent of the product or service ad in the obtained value. It is essential to showcase the potential value of an organization's services to potential new clients, as Osterwalder describes "a value proposition is an overall view of a company's bundle of products (...)" [27].

Within the scope of this dissertation, the value proposition subsides in the creation of an unified multi-concession scalable system, that maintains the same key features of the previous system, but with some revisions. This solution brings simplicity to the usage of the system, consistency between concessions and prompts the possibility of becoming the global platform for Globalvia concession operations.

3.3 Quality Function Development

3.3.1 Contextualization

The development of a new concept, or product, brings several challenges when conceptualizing its idea and analysing its potential value. The idea generation phase can be overwhelming, due to all the interested part's interest being complex, to the point that some projects may fall short because of that. This brings the need to develop protocols and models for new concepts generation such as Quality Function Deployment (QFD), that could bring improved communication between departments, potential failures with the product, possibility for new technology and cost reduction.

QFD is a system, or protocol, which main objective is to allow the product development team to add the potential client's needs into the product as a need or improvement, or as described by Paul Roberts "QFD is a system for designing a product or service based on customer demands that involves all members of the producer or supplier organisation" [28]. The foundation of this system relies on an analysis of the requirements, or needs, proposed by the potential client, to segment them in order to correlate how to achieve them.

Considering that this system can allow to get lower costs and higher quality products to market, based on the customer needs, it enables a tracking system for future product designs [28]. This system can yield several results, such as understanding customer needs, improvement of organisation's product development and the increase of the organisation's reputation around the customer base due to endeavoring in quality development. These results are achieved by properly breaking down customer needs into identifiable segments and dictate the means to achieve the previous segments.

The QFD process is split into five different levels:

- Customer requirements
- Design requirements
- Component characteristics
- Operations requirements
- Working procedures

Customer Requirements

The customer requirements level is the foundation of the QFD. A company's key action is to establish continuous customer satisfaction during the usage of their product, or service. To achieve that, gathering, using and understanding of customer requirements in the product design is a must. To conceptualize the customer requirements area, three factors must be identified: the customer chain, the meaning of *customer* in QFD and how to retrieve the customer requirements.

The first step in the customer requirements element is to define the customer chain, which will be the targeted customers for the created product. The main objective is to define a group big enough to reach the biggest number of clients, therefore the following group of clients are considered: those who bought products of the organisation, who bought competitive products, who switched to a competitor, who are satisfied and lastly, those who are not satisfied [28].

The following step consists in defining the gathered customer requirements, which should be defined in the customer's own language, in order to start shaping a customer-driven design to the design requirements step. To do this, data must be produced using field research techniques [28], such as questionnaires or interviews and discussion techniques or focus groups.

Design Requirements

With the completion of the customer requirements definition, it is necessary to investigate and set the necessary engineering techniques that must be employed and optimized, to ensure customer satisfaction. These techniques can be identified as technical and regulatory requirements, which fall under the design requirements level. "Regulatory requirements are such things as government legislation, safety requirements, quality standard requirements, classification requirements (...) Technical requirements occur because the company may have some specific plans for the new product" [28].

Upon the definition of customer requirements and engineering characteristics, the phase of establishing links between both is started. This phase consists in developing the relationship

and conflict matrices, which main intent is to highlight the relationships and conflicts between requirements [28].

between engineering characteristics, the customer requirements, the technical and regulatory requirements [28]. The matrices are used to highlight the relationships, but also their relative strengths with the support of a range of symbols.

Relationship Matrix

The relationship matrix is used to highlight the relationship between the customer requirements level and the engineering characteristics. These relationships should be weighted in strength, so a set of symbols are set as standard to be used to identify them:

⊙ Strong relationship

○ Medium relationship

△ Weak relationship

Figure 3.3: Strength symbol pattern for the relationship matrix - Source [28]

As figure 3.3 describes, there are three existing symbols, each symbolizing the strength of the relationship. The strong relationship represents a connection between the two components, while the other two symbols, represent medium and weak relationships, respectively.

Conflict Matrix

Similarly to the relationship matrix, the conflict matrix is used to highlight the relationships between the engineering characteristics and can be measured as strength, but in addition, an indication is provided if it is a positive supportive relationship or a negative conflicting relationship.

⊙	Strong positive relationship	Supportive
○	Positive relationship	
×	Negative relationship	Conflicting
✖	Strong negative relationship	

Figure 3.4: Strength symbol pattern for the conflict matrix - Source [28]

Figure 3.4 shows the symbols used for the conflict matrix, which follows a similar approach of the relationship matrix, where the main difference is that there is an indication whether the relationship is positive or negative, rather than only the strength classification.

Target Values

Once the correlation and conflicts are set on the relationship and conflict matrices, the following step is evaluate the engineering characteristics in order to determine target values for each. The target values represent what is necessary to reach customer satisfaction and the necessary metrics to determine are an estimation rating of the difficulty of achieving the evaluated target and an importance rating for the characteristic [28]. In order to choose

target values, the metrics that should be weighted are comprehended between any relationships, whether being positive and negative, that were highlighted in the conflicts matrix and the importance of the customer requirements tackled.

Technical Evaluation

Considering that time is often a constraint in software engineering, the analysis stage is influenced by the expected technical difficulty associated with the engineering characteristics stage, because it could lead to time deviations in the project, which directly affects the value to the customer. Each QFD should represent a unique study and analysis and each team should develop its own approach, target values and decide the level of technical difficulty [28].

3.3.2 Conception

Considering the project documented in this dissertation, the application of the QFD system has the intent of evaluating the customer requirements and the engineering characteristics grounded on the analysis. In the context of this dissertation, the target customer is Globalvia - not the motorway users - as the intent of the system is to be deployed to deal with the motorway infrastructure.

The customer requirements identified are based on Globalvia's feedback, as well as the goals identified in the section 1.3. The following customer requirements were determined and selected for analysis:

- Toll collection;
- Toll information validation;
- Scheduling of system actions;
- Multi-concession support;
- Automated deployment process.

After the gathering of customer requirements has been completed, the next step is to quantify the importance of each requirement for the customer. This step requires customer input and was inducted by Globalvia's feedback, where it has been graded by the customer, in a scale of 1 to 9, where 9 being the most important and 1 less important. The customer importance is included, for reference, in the relationship and conflict matrix represented on figure 3.5.

The following step is to delineate the engineering characteristics, which should take into account the customer requirements previously identified, because the characteristics "must be optimised to assure customer satisfaction" [28]. The following characteristics were identified:

- Multi-concession system design;
- Cloud system infrastructure;
- Continuous Integration (CI) & Continuous Delivery (CD) process;
- Automated task scheduler;
- Information validator;

- System performance;
- System scalability.

With the finalization of the steps in regard to customer requirements, customer importance and engineering characteristics, the adequate information for the QFD system has been gathered and the succeeding step is the establishing links phase, where the relationship and conflict matrices are filled, as represented in the figure 3.5.

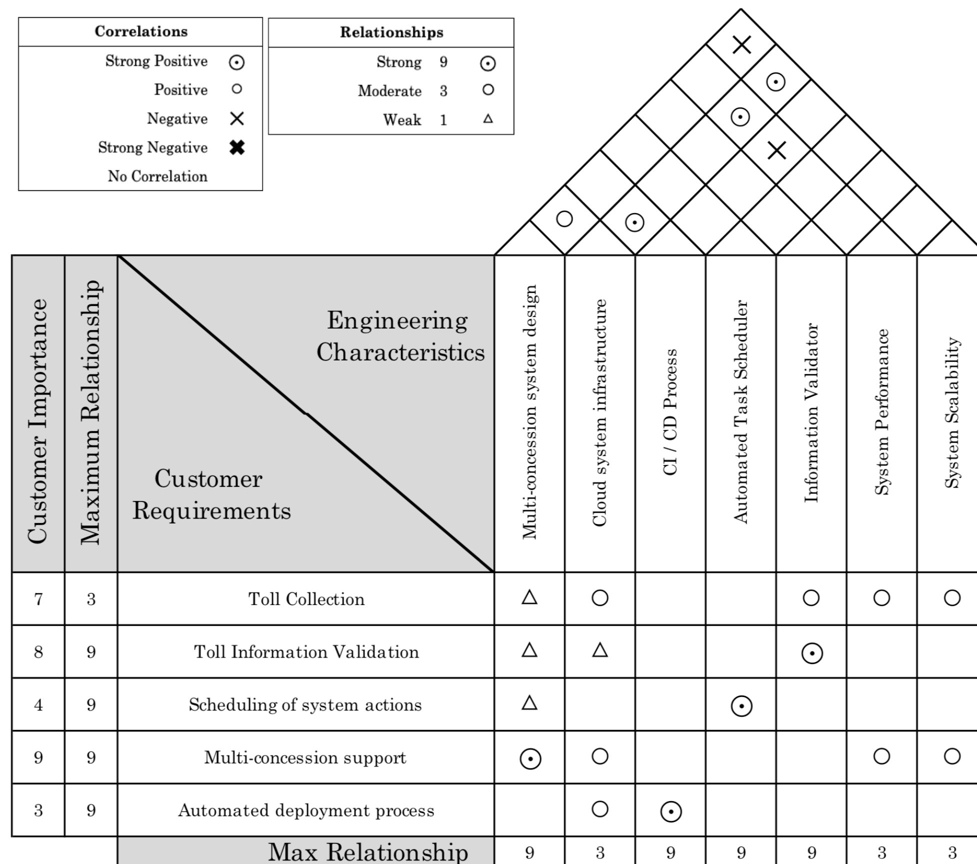

Figure 3.5: Relationship and conflicts matrix

Evaluating the relationship matrix, the customer requirement *Toll Collection* is denoted as the most linked requirement, although at the same time it has the weakest links. Although the toll collection functionality is the crucial feature for the system, it should be able to do its job without heavy influence from other engineering characteristics, such as the information validation or the automatic task scheduler, and only be weakly affected by the creation of these components. Additionally, the the *Multi-concession support* requirement is marked as the most important for the customer, as well as one of the highest relationships. This is due to the key functionality of this project being the support for multi-concession toll collection capabilities.

For the conflict matrix, it is apparent that the engineering characteristic *Cloud system infrastructure* is the most influencing one, due to having strong positive correlations with *CI / CD Process*, *System Performance* and *System Scalability*, while the *Multi-concession system design* is negatively correlated with *System Scalability* due to the fact that a limiting design could endanger the scalability, although not canceling it.

Analysing the customer importance step, it is possible to identify that the crucial customer requirements are *Multi-concession support* and *Toll Information Validation*, which should be achieved in order to provide value for the customer.

Having the aforementioned steps been successfully analyzed, it is possible to calculate the technical importance ratings and weights, which can be verified the in the appendix A.

Chapter 4

State Of The Art

In consideration of Globalvia's vision and objectives, as well as the comparison analysis of competitors represented in the previous sections, the decided path of action onward is the creation of a multi-concession toll collection and validation system, based in cloud infrastructure and on-demand resource scalability, as a proof of concept. A successful proof of concept will then kick-start a roadmap for the development of a fully fledged Software as a Service (SaaS) for toll collection.

This chapter describes the state of the art around the concepts that enables the implementation of the solution in hand. It is started by describing software design concepts, such as the monolith and micro-services architectural patterns. Leading into infrastructure, where theoretical concepts of scalability, cloud computing and software delivery are presented and its models are detailed. Finally, an analysis of multi-tenancy design design patterns is described.

4.1 Software Architecture Patterns

Software architecture is "the set of principal design decisions made during its development and any subsequent evolution" [29]. This makes architecture as the primary focus of software engineering, especially on the system's undeveloped state being that, typically, the early decisions of a system aims to become the standard forward.

As in most of the software engineering, the architecture decision process is usually repeatable for different goals with similar problems. This is where the patterns come into cooperation with the process. "Each pattern is relatively independent, but patterns aren't isolated from each other" [30], thus generating the need of comparing and evaluating how software architecture patterns behave and intersect. The research process for the project is narrowed down to three distinct patterns: monolith, micro-services and event driven.

4.1.1 Monolithic Architecture

A monolithic system can be defined as a unique single built software component, containing all the necessary code base, services, resources and artifacts in order to allow a fully functional application to solve a specific problem. A typical enterprise application is built with three parts: user interface, a database and a server-side application. The server-side application can offer several different communication interfaces, such as Representational State Transfer (REST) services, as well as execute domain logic, read from the database and even rendering HyperText Markup Language (HTML) pages [31].

The monolithic system is also considered to be fully deployed component, as a unit. Some of its advantages are:

- Easy to start development;
- Easy to deploy, only a single deployment process needed;
- Easy to scale, simply duplicate the whole system;
- Consistent developer environment;
- Cross-team features are easier to implement as a single code base is used.

Which also brings some disadvantages:

- Inefficient to scale, as its only option to scale is to duplicate the whole system;
- Troublesome maintenance when the application has become too rich with features;
- Limited to the same technology and/or framework, not taking advantage of other technology that could be better for a certain feature or requirement.

Ultimately, monolithic applications are easy to build and can be successful. However, with the rise of cloud computing and the need of scalability, some architectural alternatives are emerging to tackle the limitations around scaling.

4.1.2 Micro-services Architecture

The micro-services architectural pattern describes the approach of developing a software system distributed under a collection of small (micro) services [32]. Each micro-service has its own environment, process, follows its own standard and may be implemented in a different technology. These services should be easy to manage, low on information centralization, fully autonomous, have their own deployment process and be designed around their specific domain. This architectural pattern brings several advantages, such as:

- Low coupling;
- Independently scale as necessary, allowing to scale only a specific service;
- Definition of boundary context between services;
- If a service fails, the system as a whole is not affected.
- No need of an Enterprise Service Bus (ESB) layer as the micro service has its own domain - "smart endpoints and dumb pipes" [32].

As for the disadvantages:

- Increase in complexity due to its distributed system nature;
- Services need to be fault tolerant and highly available;
- An inter-service communication mechanism must be maintained;
- Testing dependency is troublesome;
- Adopting an automated deployment process, such as Continuous Integration (CI), is a must in order to make development efficient.

The micro-services architectural pattern's is by nature, an advocate for distributed systems, which makes this pattern very suitable to pair it with cloud computing core techniques, as aforementioned in section 4.2.1.

Several concerns are raised when implementing a micro-services architecture, such as how to ensure data consistency, how to query information that uses more than one service, or even how to maintain dependencies between services. These concerns can be addressed using design patterns, as shown in the next sections, aimed at micro-services architectures.

API Gateway

An Application Programming Interface (API) Gateway component supplies an entry point to communicate with a specific part of one or several existent micro-services [33]. This enables the possibility to address the issue of exposing too much of the micro-services APIs, avoiding the need of the user to know where to fetch for a specific kind of information. It can also handle clients of different natures, apply authorization and authentication, logging and tracking of bad requests. Nonetheless, the pattern needs to be used simultaneously with Service Discovery, in order to the avoid the problem needing to know the specific routing to every service.

Service Discovery and Registry

Service Discovery pattern aims to solve the problem of locating services that are not statically located and known at design time or runtime [33]. This can happen in several environments, but it is mainly in cloud-based systems, which can relocate, scale or remove services continuously at runtime. The discovery process is usually tied directly with the service registry pattern.

A service registry is used by other services, and the service discovery pattern, in order to retrieve imperative information about other services locations [33]. Micro-services usually communicate with the service registry at boot time to publish their locations, but also to query about other services locations.

API Composition

In a micro-services architecture, querying for information that is split over more than one service is not as straightforward as in a monolithic architecture. This is caused by the segregation of concepts and boundary domain contexts between services. The API Composition pattern aims to provide a simple way to query information over several services, by producing a single result joining several responses [34].

Messaging

Inter-process non-blocking communication is often needed in distributed systems and this comes as a must have in micro-services due to the architecture's nature of constant communication. The messaging pattern consists in two or more services exchanging messages over messaging-specific channels [34]. They can follow several styles of communication, such as notifications or publish and subscribe domain, and gravitate to asynchronous behavior. Asynchronous operations can be specially useful for micro-services to address availability and the fault tolerant concerns.

Shared Database between Services

The adoption of the shared database between services pattern, simplifies some concerns aforementioned, such as data querying. Furthermore, it also enables micro-services to fetch

data from other domain entities without needing to resort to inter-process communication [34], considering that the service is able to directly fetch the wanted data from the database when wanted.

However, this pattern has its drawbacks. Sharing a database between services may cause tight coupling, introduce extra development time constraints, as the schema must be coordinated between one or more services and could even lead to blocking runtime processes, caused by other services holding a lock on the wanted table.

Database per Service

The database per service pattern denotes that each micro-service should have its own database and should only be queried by the service to which it belongs [34]. This enables low coupling between micro-services and their data access layer, considering that a change to the database schema only affects a single service, while also enables each service to use the database technology that suits the best for the functionality provided.

The major drawback of this pattern is the added complexity of implementing business transactions that cover several services, which could lead to several data inconsistencies. Implementing queries that span over several services is also a problem, which could be tackled with the API Composition pattern, as covered previously.

4.1.3 Event Driven Architecture

The event driven architectural pattern describes the approach of developing systems of several decoupled components or services, in which events are transmitted between them, exchanging information [35]. In this architectural style, the components will be subscribed to themed topics, in which they're interested to. When something notable happens in the system, it will be disseminated to all components subscribed to that topic, which can lead to a component optionally taking action based on the event.

Event driven systems are dependent on the communication process between components. The process is lead by the usage of messaging, whereas the message represents a business event. The process is usually designed following the publish-subscribe mechanism, where "subscribers have the ability to express their interest in an event, or a pattern of events, and are subsequently notified of any event, generated by a publisher, which matches their registered interest" [36].

Similarly to micro-services architecture, several concerns arise when designing systems using the even driven architecture pattern. Due to its distributed nature, the concerns are alike to micro-services, adding extra worries for managing the communication process. To address these concerns, it is advisable to evaluate specific design patterns.

Event Notification

An event notification "happens when a system sends event messages to notify other systems of a change in its domain" [37], without the need of any response from the receivers of the sent notification. The components that work under this pattern expect no response to their events, and if a response happens from a subscriber, it is usually sent by another notification to a distinct topic.

This pattern is present at the core of the event driven architecture, whereas events are considered facts of something that just happened in the domain of the system, as well as a

notification. It promotes low coupling between components, but it may bring problems to understand the typical flow of the application if the data throughput values are high and the logical flow iterates over multiple event notifications.

Message Broker

A message broker is an intermediary software component designed for queuing, managing and publishing messages for its subscribers [35]. Being one of the most important components in an event driven architecture, a message broker makes the use of the publish-subscribe mechanism to function. The publish action is executed by a specific component of the system that works directly with the broker, which will send a notification with a message to a specific topic, intended for the broker to store and forward to all the subscribers of that topic, in a reliable and timely manner, making it perfect for "streaming or fire-and-forget messaging" [35].

Event Sourcing

Event sourcing pattern is described as the observation of events at a core level of the system, or alternatively described in the words of Martin Fowler "whenever we make a change to the state of a system, we record that state change as an event, and we can confidently rebuild the system state by reprocessing the events at any time in the future" [37].

This pattern can be seen as a way to apply a version control system for system domain data, so if the data is immutable and stored in order that it was created in, then a listing of the event log provides an extensive informative report of the actions that the system executed. The key for this pattern to success is that there's a need to guarantee that all changes of domain objects are started by listening to domain events. This enables the usage of certain resources, such as event replay, temporal queries and complete rebuild of domain data by rebuilding all events.

Command Query Responsibility Segregation (CQRS)

CQRS is a logical segregation of models used in reading and writing operations. This means that two distinct models must be developed, which can help when maintaining a single model to handle read and write operations becomes too complex. However, this pattern adds an extra layer of complexity and maintenance to the component, but does become specially appealing when the component has a differentiated path of access points, for instance several read operation but a lack of write operations.

The CQRS pattern isn't directly bounded to event driven architecture, however it is usually implemented together with the aforementioned event patterns [37].

4.2 Infrastructure

4.2.1 Cloud Computing

Cloud computing is a model designed to allow "convenient, on-demand network access to a shared pool of configurable computing resources" [38], that can be used by enterprises to design and deploy cloud-based applications, which provides new opportunities to deploy scalable applications in a more efficient way. In the extreme case, cloud computing enables developers to dedicate their time into software development, process automation and testing,

by delegating the worry of resource management for the cloud provider. This model can be described by the following characteristics:

- On-demand access;
- Broad network resources;
- Resource pooling;
- Resource elasticity;
- Service usage measurement.

Cloud services are usually offered publicly under a *pay-per-use* principle, where the costs of the infrastructure is scaled as much as the used cloud computational power. These services are distributed in different models, such as SaaS, Platform as a Service (PaaS), Infrastructure as a Service (IaaS) and Function as a Service (FaaS).

Cloud computing has been considered one of the popular trends in software engineering, an argument that is backed up with the growth of migration processes from *bare bones* to cloud computing, as well as more and more companies are rolling out cloud-computing services, as for the most popular services being Amazon Web Services (AWS), Google Cloud Platform and Microsoft Azure.

4.2.2 Software delivery

A well established software delivery process is one of the key aspects for software success. Several process models exist to achieve that, but CI & Continuous Delivery (CD) are considered the *de facto* standard. Being one of the main software engineering best practices, CI & CD is focused to facilitate testing, integration and publishing of the continuous development executed in the lifespan of a given project.

In the words of Martin Fowler, CI is "a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible" [39]. Following this practice brings other practices that should be followed as well, such as:

- Build automation;
- Regular commits to the mainline;
- Every commit should build the solution;
- Strive for fast builds.

CD is a software development principle where the software is developed in such way that "the software can be released to production at any time" [40]. This principle is achieved when there's an existing CI process in the software development team, with automated tests and an automated software deployment process is also built.

The theory behind software delivery practices is in a rapid growth and at the same time reaching its maturity. A standard set of tasks in a delivery pipeline is represented in the following figure.

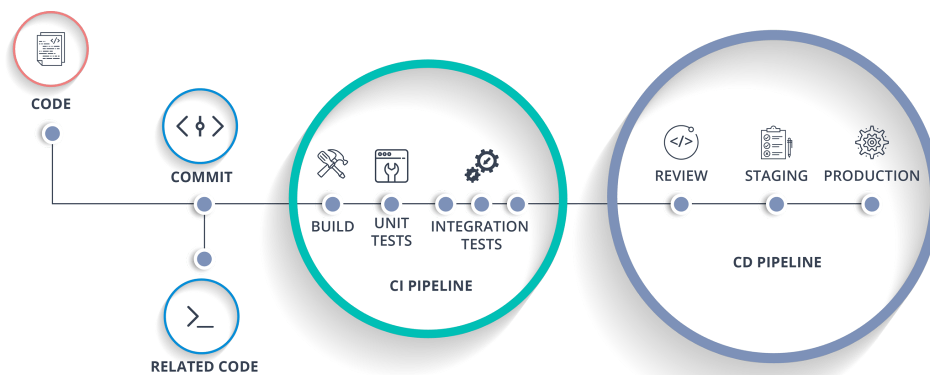

Figure 4.1: Standard build and delivery pipeline - source [41]

4.2.3 Scalability

Scalability, in a software development context, can be described in two different dimensions. Firstly, there is load scalability which is the ability of a certain application to grow in order to handle more load. And lastly, structure scalability is the ability of a component to expand in a specific aspect without the need of an architectural modification [42]. The focus of this section is on load scalability.

When a software component workload grows to handle more tasks simultaneously, the resource limits of the instance can quickly be breached. To tackle this problem, one can investigate ways to optimize the component in order to take advantage of the existing hardware and use it more efficiently. Alternatively, the hardware resources of the instance can be increased in order to ease the load, which is the best option in most cases, although it is limited to the instance's physical resources. When the component becomes too big for a single instance, it is necessary to introduce another instance of the application. This will cause the load to be shared between the two instances, with the help of a load balancer. Additionally, the introduction of a second instance can also benefit software redundancy, as the case of failure of instance, the other will remain working and will take over the load of the crashed instance.

The role of scalability takes a greater focus when software and cloud computing are combined. As previously mentioned on section 4.2.1, cloud computing offers the possibility of a continuously growing stream of computing resources. This contributes directly to the load scalability characteristic, as the number of instances grow, hardware limits can be easily be increased by increasing the number of cloud nodes or increase instance resource tiers. Additionally, some cloud providers might offer auto-scaling functionality when combined with specific technology.

Scale Cube

Martin Abott and Michael Fisher developed a scalability model, named the Scale Cube [43]. This model helps the thought process on how to split processes, services, data and even databases to find a path on how software can be scaled to reach its maximum scalability potential. This model claims that the weakest point of scalability is a system using monolithic architecture, while point of maximum scalability being a distributed system with partitioned data and several cloned services. Figure 4.2 shows the model's selected three scalability dimensions in detail.

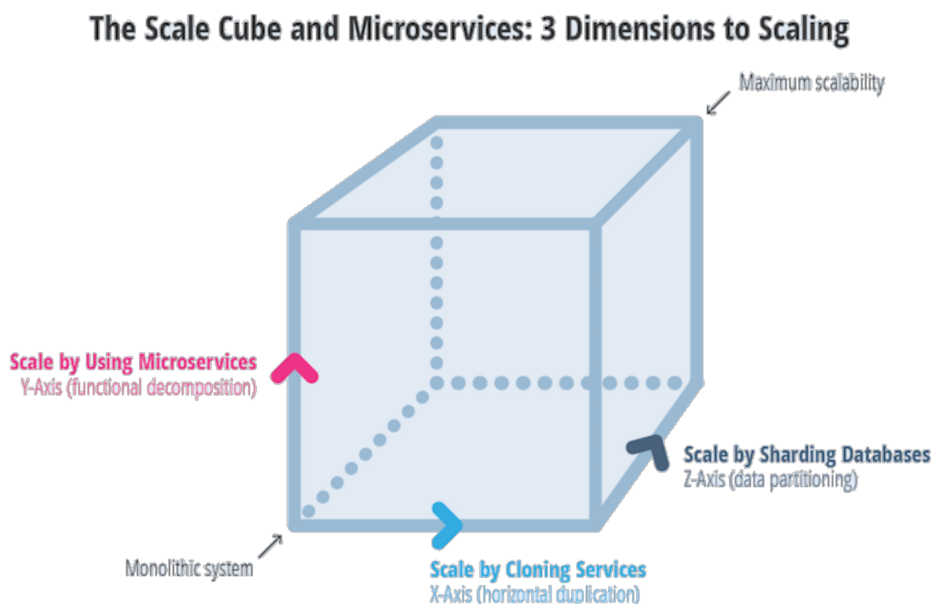

Figure 4.2: Scale cube model - adapted from [44]

Starting with the x-axis dimension - horizontal duplication - is achieved on adding additional instances by cloning the component with no criteria. This kind of scaling is easy to conceptualize and to apply, but if the component is too big, the creation of new instances might exceed the environment's resources. Consecutively the y-axis dimension, focused on functional decomposition, tackles the x-axis main problem by segregating the monolithic component into a set of several processes. This enables the possibility of scaling just the components with heavier load, instead of scaling the whole component. Lastly, the z-axis dimension engages on the data layer. Supported by partitioning techniques, the scaling is based on creating additional components to run on an identical copy of the code, similar to the x-axis, but each component is responsible for a subset of the available data. This dimension is usually applied to database scaling, where the data is sharded across a set of several databases based on a specific attribute of each record. The biggest drawback of this technique is the exponential increase of application complexity, as well as the need of having a planned partitioning scheme, which can be troubled to implement on existing data. However, it does reduce memory usage, as well as saving on disk usage due to cache utilization and I/O influx.

4.2.4 Infrastructre as Code

Managing IT infrastructure has always been a specialized and manual process throughout the years [45]. Specialized engineers would put several hours in planning, configuring and implementing infrastructure designs, in physical data-centers and then deploying systems. This kind of process is manual and long, and it brings several problems such as cost, scalability, availability and inconsistency. Firstly, the costs would account for hiring technicians to do the initial implementation, as well as server maintenance and on top of that, the cost of buying hardware for all necessary servers, as well as the maintenance around them. The next problems are scalability and availability, with an unexpected rise of workload, software

components could struggle and manual intervention would be required to scale the system, which would directly impact the system availability. Lastly comes the inconsistency problem. Taking into account the fact that the software deployment process was manual and slow, several engineers would need the know-how on the process, which could bring inconsistencies, as discrepancies tend to happen.

Following the revolution started by cloud computing, the processes around designing, developing and the maintenance of IT infrastructure are also changing and adapting to its new reality. This concept frees up the need of having physical data-centers and the costs associated with it, allowing to set up infrastructure quickly and helping the problems related to scalability and availability. However, cloud computing in itself doesn't aim to solve the inconsistency problem, as setting up cloud infrastructure may still be a manual process.

Infrastructure as Code (IaC) is an approach to manage IT infrastructure through definition configuration files, rather than physical hardware configuration [46]. This concept tackles the inconsistency problem of manual processes, by emphasizing "consistent, repeatable routines for provisioning and changing systems and their configuration" [46]. This allows to have a versioned code base for system infrastructure, repeatable and described in a declarative manner. This is made possible by taking advantage of dynamic cloud infrastructure providers API.

The premise of IaC is that the infrastructure process is included in the software architecture, just like another software component included in the system and have the modern tools treating it just like software. It enables to explore software development practices, such as automation, testing and even the integration of infrastructure creation in CI & CD pipelines.

All things considered, the value that is extracted from this concept is laid down on the code reusability, versioning, as well as the possibility of having an automated process on top of it.

4.3 Multi-tenancy

Multi-tenancy is a software architecture principle that classifies a software component being tenant agnostic, which means that in order to offer its full functionality, it does not require to be fully aware of the tenant it works for. This principle allows for a fairer use of the infrastructure resources [47], because tenants share the same application instance, while at the same time, having the component full functionality working.

In a SaaS application context, a tenant "subscribes or pays to use the SaaS application, however a tenant comprises many end-users" [48], which correlating with the project in hand, a tenant is considered to be a concession, while the comprised end-users are the concession managers and specialized back office operators.

According to a study induced at the Institute of Architecture of Application Systems, University of Stuttgart [49], service tenancy patterns can be classified with direct relationship of software life-cycle: development and deployment. At the time of software development, multi-tenancy can be accomplished by creating structures of configuration options per tenant. While for the deployment cycle, service instances can be scaled to as many existing tenants, which may or may not share workload on the same instance. The combination of the two classification concepts lead to the introduction of multi-tenancy design patterns, as described in the following sections.

4.3.1 Single Instance

The single instance multi-tenancy pattern is described as having a software component that its entry-points and configurations are the same for all tenants that interact with it, as well having a consistent behavior between all tenants. It is best described by its characteristics [49]:

- Deployable as a unit for all tenants;
- Behaves the same for all tenants;
- No custom configurations for multi-tenancy;
- Tenants identification is not required;
- All data used in the component is shared between tenants that interact with it.

This pattern is passively present most systems that do not have any multi-tenancy consideration, as applications that do not implement any multi-tenancy patterns are naturally not ready for tenant-aware environments.

4.3.2 Single Configurable Instance

Similarly to the single instance multi-tenancy pattern aforementioned, this pattern offers the same entry points for all tenants, however it takes advantage of tenant-specific configuration to behave differently [49]. The tenant configurations can be deployed in a database, or being present actual configuration files in the component's environment.

This pattern requires special attention on its entry-points, as it is typically required for a tenant to identify itself in order to select the proper configuration to be used in the task. This can be achieved by requiring some kind of tenant identification during the task request, for instance a tenant-specific token provided by a secure header in a HyperText Transfer Protocol (HTTP) REST request that will then match a configuration in the database, also known as a tenant context [49]. All things considered, this pattern's characteristics are defined as:

- Deployable as a unit for all tenants;
- Behaves differently for each tenant;
- Custom configurations are required for each tenant;
- Tenant identification is required;
- Component implementation must ensure that a specific tenant data is isolated from the remaining tenants.

4.3.3 Multiple Instances

The multiple instances multi-tenancy pattern is described as having a software component dedicated to serve one or more tenants per instance. This means that each deployable instance will have tenant configuration for only the specific tenants it is aware of. This pattern is characterized as [49]:

- A deploy must occur for each tenant;
- Behaves differently for each tenant;
- Custom configurations are required for each tenant;
- Tenant identification is optional;
- Component implementation does not need to ensure isolation, as it does not share the same runtime environment between tenants.

The biggest drawback of this pattern is maintainability, as a change in the component will require all instances to be redeployed with that change, which could represent an issue on medium-sized architecture with several tenants.

Chapter 5

Analysis

In this chapter, it is presented the analysis and design of the solution discussed in this dissertation. Unified Modeling Language (UML) notation is used to represent the different components of the system to be developed. Firstly, a software requirement engineering process is induced, in order to identify the functional and non-functional requirements of the system. Followed by a detailed explanation of the key domain concepts for the business, also illustrated with UML diagrams.

5.1 Requirements

During the early stages of software engineering, it is important to produce an extensive requirement engineering, in order to achieve early validation of the plan from the stakeholders. These requirements can take form in functional or non-functional requirements. Functional requirements represent the main product functionality, or a functionality that the system must address. Non-functional requirements illustrate characteristics that the system must follow in specific categories, such as performance or security, or how the system should behave.

In order to induce the software requirement engineering process, the FURPS+ [50] requirement classification system can be used. The model's name is derived from an acronym formed out of its categories functionality, usability, reliability, performance, supportability and the + for additional concerns, such as design, implementation, interface and physical requirements.

5.1.1 Non-functional requirements

Consequent to the execution of the FURPS+ requirement classification system, the non-functional requirements fall on the following:

Usability

- Interface must be easy to use;
- Informational pages should be domain-driven.

Reliability

- The system must scale upon increased workload from the Road Side Equipment (RSE);
- The system must scale with no downtime;

- The system should have a low rate of errors;
- Error tolerance must be ensured;
- The system must give reproducible results.

Performance

- Fast Application Programming Interface (API) response time must be ensured;
- The usage of hardware resources must be minimized.

Supportability

- The system should work for any concession without the need for major configuration overhaul;
- System metrics must be monitored;
- The system must be extensible;
- The system must be ready to support external services.

Design constraints

- System must be designed for cloud computing;
- Cloud hosting provider must be Amazon Web Services (AWS);
- Adoption of software design patterns;
- One concession sensitive data must not be stored in shared databases, such as payment and personal user data;
- The system should have different more than one environment, such as production and staging.

Implementation

- Adoption of coding standards;
- Early adoption of Continuous Integration (CI) & Continuous Delivery (CD) processes.

Interface

- The system must support HyperText Transfer Protocol (HTTP) Representational State Transfer (REST) communication from the RSE.

Physical

- The system network should be installed in Globalvia's Virtual Private Network (VPN).

5.1.2 Functional requirements

As aforementioned, functional requirements represent the main product functionality, or a functionality that the system must address. As a result of the requirements engineering phase, the identified actors for the main functionality of the application are: BackOffice Operator, Concession Manager, Manual Reviewer, Commercial Reviewer, Motorway User, Issuer and System. In the following sections, the use case diagrams for each actor are displayed, as well as a description of each functionality.

5.1.2.1 BackOffice Operator use cases

Based on the identified functional requirements, the determined use cases for the backoffice operator role are the following:

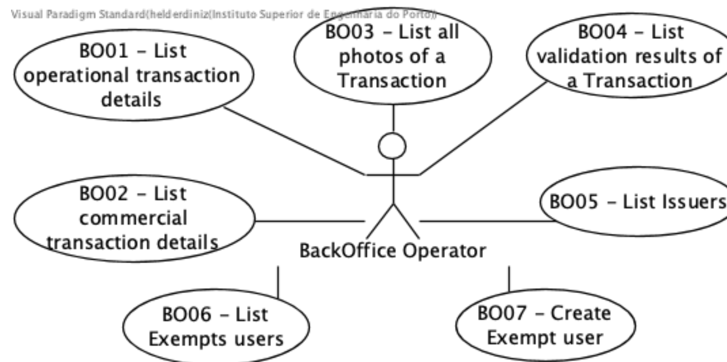

Figure 5.1: Use case diagram for BackOffice Operator

BO01 - List operational transaction details

The BackOffice Operator, at any given time, is interested in listing all the operational transactions for a concession. Additionally, list filtering functionality must be provided, in order to filter for specific operational transaction attributes, such as transaction date, gantry, lane and even classifications.

BO02 - List commercial transaction details

The BackOffice Operator, at any given time, is interested in listing all the commercial transactions for a concession. Similarly to operational transactions listing, filtering functionality must be provided, in order to filter for specific commercial transaction attributes, such as payment method, gantry, lane and others.

BO03 - List all photos of a Transaction

As a BackOffice Operator, while navigating the list of operational or commercial transactions (BO01 or BO02), it must be able to fetch a detailed view of transaction details, that includes all photos for that specific transaction. Subsequently, each photo should list its attributes, such as license plate and photo view.

BO04 - List validation results of a Transaction

As a BackOffice Operator, while navigating the list of operational or commercial transactions (BO01 or BO02), it must be able to fetch a detailed view of transaction details, which includes the validation results for the specified transaction. This detailed view should also list validation details, such as type, if it passed the validation or not and when it was executed.

BO05 - List Issuers

The BackOffice Operator, should at any given time, be able to verify a full listing of all issuers that the provided concession supports. Additionally, the detailed list view should include the issuer information, such as name, time to charge and the indication if it is a financial or non-financial issuer.

BO06 - List Exempt users

As a BackOffice Operator, it should be possible that at any given time, a full listing of all existing exempts is provided, for the concession. In addition, the detailed list view should include the exempt information, such as license plate, card number or On-Board Unit (OBU) number.

BO07 - Create an Exempt user

The BackOffice Operator, should be allowed to create a new exempt user, for a specific concession. To do so, it should be preceded by the feature BO06, in which will contain an option to create a new exempt. Subsequently, a form to insert the exempt information should be displayed, as well as a confirmation if the information is valid and has been created.

5.1.2.2 Concession Manager use cases

Based on the identified functional requirements, the determined use cases for the concession manager role are the following:

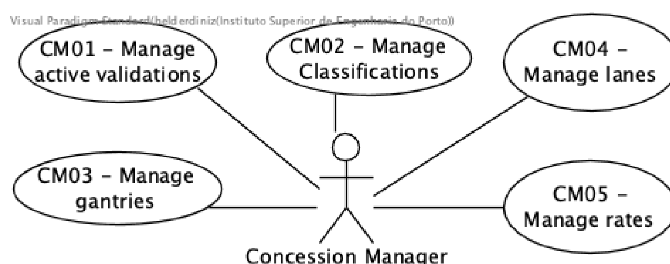

Figure 5.2: Use case diagram for Concession Manager

CM01 - Manage active validations

As a Concession Manager, it should be possible to manage the active validations for the concession the manager works for. This functionality should offer a possibility to list all validations, as well as activate or deactivate each validation.

CM02 - Manage classifications

As a Concession Manager, it should be possible to manage the vehicle classifications that are used throughout all gantries and lanes, for the concession the manager works for. This functionality should offer a possibility to list classifications, as well as edit any existing one, or create a new classification.

CM03 - Manage gantries

The Concession Manager, should be able to manage the existing gantries that are installed throughout all motorway infrastructure, for the concession the manager works for. This functionality should offer a possibility to list gantries, as well as edit any existing one, or create a new gantry.

CM04 - Manage lanes

The Concession Manager, should be able to manage the existing lanes, that are associated with a gantry, throughout all motorway infrastructure, for the concession the manager works

for. This functionality should offer a possibility to list lanes, as well as edit any existing one, or create a new lane. Furthermore, each lane should be linked to an existing gantry and the functionality must offer a link to the use case CM03 if the intended gantry does not exist.

CM05 - Manage rates

As a Concession Manager, it should be possible to manage the charging rates, that are applied to each vehicle when passing the toll gantries, throughout all motorway infrastructure, for the concession the manager works for. This functionality should offer a possibility to list all existent rates to date, as well as edit the current active rate, or create a new rate. Commonly, motorway charging rates are applied over a specific period of time, so only one rate can be active at a time. As a result, the concession manager will not be able to add a new rate with overlapping periods.

5.1.2.3 Manual Reviewer use cases

Based on the identified functional requirements, the determined use cases for the manual reviewer role are the following:

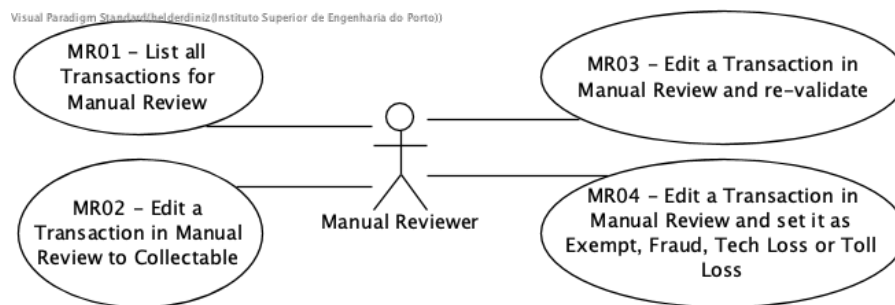

Figure 5.3: Use case diagram for Manual Reviewer

MR01 - List all Transactions for Manual Review

The Manual Reviewer, at any given time, should be able to access a listing of all the operational transactions, that are on status manual review, for a concession. Additionally, list filtering functionality must be provided, in order to filter for specific operational transaction attributes, such as transaction date, lane, gantry, and others.

MR02 - Edit a Transaction in Manual Review to Collectable

As a Manual Reviewer, it should be possible to edit a transaction that is in manual review status, in order to set it in collectable status and have the transaction moving on the typical data flow. For this functionality, the manual reviewer user must be able to see a detailed view of the transaction information in hand, in an editable manner, such as a form. Additionally, the manual reviewer can provide a comment that will be associated with the transaction.

MR03 - Edit a Transaction in Manual Review and re-validate

As a Manual Reviewer, it should be possible to edit a transaction that is in manual review status, in order to have it re-validated by the system. For this functionality, the manual reviewer user must be able to see a detailed view of the transaction information in hand,

in an editable manner, such as a form. Additionally, the manual reviewer can provide a comment that will be associated with the transaction.

MR04 - Edit a Transaction in Manual Review and set it as Exempt, Fraud, Tech Loss or Toll Loss

The Manual Reviewer user, should be able to edit a transaction that is in manual review status, in order to set it in one of the four terminating statuses: exempt, fraud, tech loss or toll loss. For this functionality, the manual reviewer must be able to see a detailed view of the transaction information in hand, in an editable manner, such as a form. Additionally, the manual reviewer can provide a comment that will be associated with the transaction.

5.1.2.4 Commercial Reviewer use cases

Based on the identified functional requirements, the determined use cases for the commercial reviewer role are the following:

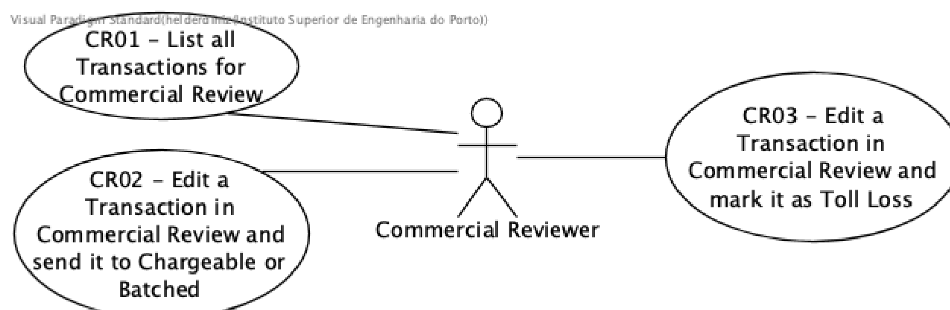

Figure 5.4: Use case diagram for Commercial Reviewer

CR01 - List all Transactions for Commercial Review

The Commercial Reviewer, at any given time, should be able to access a listing of all the commercial transactions, that are on status commercial review, for a concession. Additionally, list filtering functionality must be provided, in order to filter for specific commercial transaction attributes, such as payment method, lane, gantry, and others.

CR02 - Edit a Transaction in Commercial Review and send to Chargeable or Batched

As a Commercial Reviewer user, it should be possible to edit a transaction that is in commercial review status, in order to have it flow to chargeable or batched status, with the intent to have it sent to the client. For this functionality, the commercial reviewer must be able to see a detailed view of the transaction information in hand, in an editable manner, such as a form. Additionally, the commercial reviewer can provide a comment that will be associated with the transaction. With the form submission, the transaction should flow to chargeable or batched status, in order for it to be included in the next issuer's periodic sending phase.

CR03 - Edit a Transaction in Commercial Review and mark it as Toll Loss

The Commercial Reviewer user, should be able to edit a transaction that is in commercial review status, in order to set it in the terminating status: toll loss. For this functionality, the commercial reviewer must be able to see a detailed view of the transaction information in hand, in an editable manner, such as a form. Additionally, the commercial reviewer can

provide a comment that will be associated with the transaction. With the form submission, the transaction should flow to toll loss status, and its transaction status flow should end.

5.1.2.5 Motorway User use cases

Based on the identified functional requirements, the determined use cases for the motorway user (the client) role are the following:

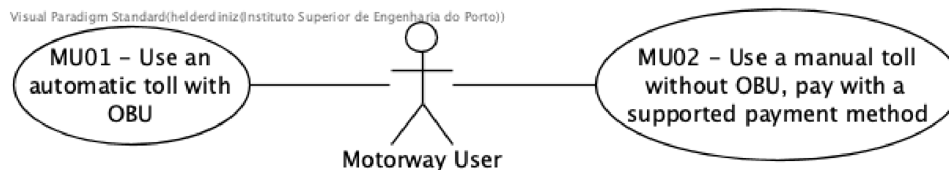

Figure 5.5: Use case diagram for Motorway User

MU01 - Use an automatic toll with OBU

As a Motorway User, it should be possible to use a vehicle with a mounted OBU, in the motorway infrastructure, and pass on a gantry with a lane that contains an automatic toll. This should then trigger the action of sending the vehicle's information to the system, and seeing the transaction charged by the issuer of the OBU, between the passage date and the Issuer's maximum time to charge.

MU02 - Use a manual toll without OBU, pay with a supported payment method

As a Motorway User, it should be possible to use a vehicle without a mounted OBU, in the motorway infrastructure, and pass on a gantry with a lane that contains a manual toll, while paying with a supported payment method by the concession, such as cash or card. The payment method could range from credit card, issuer card (financial issuer) or cash. This should then trigger the action of sending the vehicle's information to the system. On the case of issuer card payment, the motorway user expects seeing the transaction charged by the issuer of the card, between the passage date and the Issuer's maximum time to charge.

5.1.2.6 Issuer use cases

Based on the identified functional requirements, the determined use cases for the issuer role are the following:

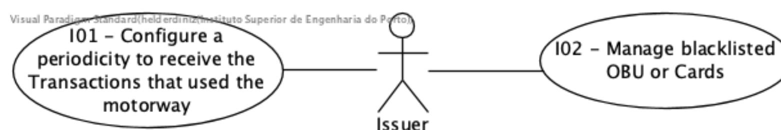

Figure 5.6: Use case diagram for Issuer

I01 - Configure a periodicity to receive the Transactions that used the motorway

As an Issuer, it should be possible to configure a periodic value to be used as a time value decider to receive the Transactions, from motorway users, that used payment method issued

by the issuer's entity, whether being a financial issuer (cards) or non-financial (OBU). For this functionality, the main actor of this use case must be able to access a view with the issuer's information, and be able to edit it.

I02 - Manage blacklisted OBU or cards

The Issuer user, should be able to manage a list of banned OBU or cards, whether the issuer is non-financial or financial, respectively. As a result of this motivation, the issuer should be able to list the blacklist, as well as adding or removing any entries from that list. Each entry on the list should have three attributes: start date, end date and the card number or OBU number.

5.1.2.7 System use cases

Based on the identified functional requirements, the determined use cases for the system - such as scheduled actions - are the following:

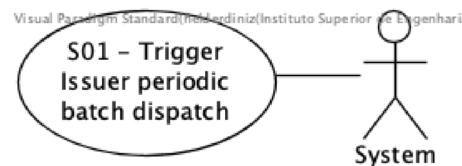

Figure 5.7: Use case diagram for System

S01 - Trigger Issuer periodic batch dispatch

The use case S01 is not based on a user triggered action, but instead a system task. This system task should be able to trigger an action to send a batch of transactions to the Issuer, based on the definition of their periodicity value.

5.2 Domain

5.2.1 Domain model

Visual representation of the most relevant domain concepts is imperative in a successful requirement analysis. A domain model is a visual representation of the business key concepts, where each object represents meaningful data and is connected to other concepts of the business. Figures 5.8 and 5.9 contain the key concepts of Globalvia's toll collection system, as well as the relationship between concepts.

As aforementioned on section 1.2, one of the motivations of this system is to create an effective division by OBO and CBO domain, and so two domain models were developed to take emphasis on this requirement. However, a complete domain model is also available on annex B. In order to provide consistency and information-rich features, some entities must co-exist in both domains. This is the case for Issuer and Concession entities, that contain critical information for both domains, in order to execute validations or orchestrate client billing systems.

Operational Back Office (OBO) domain

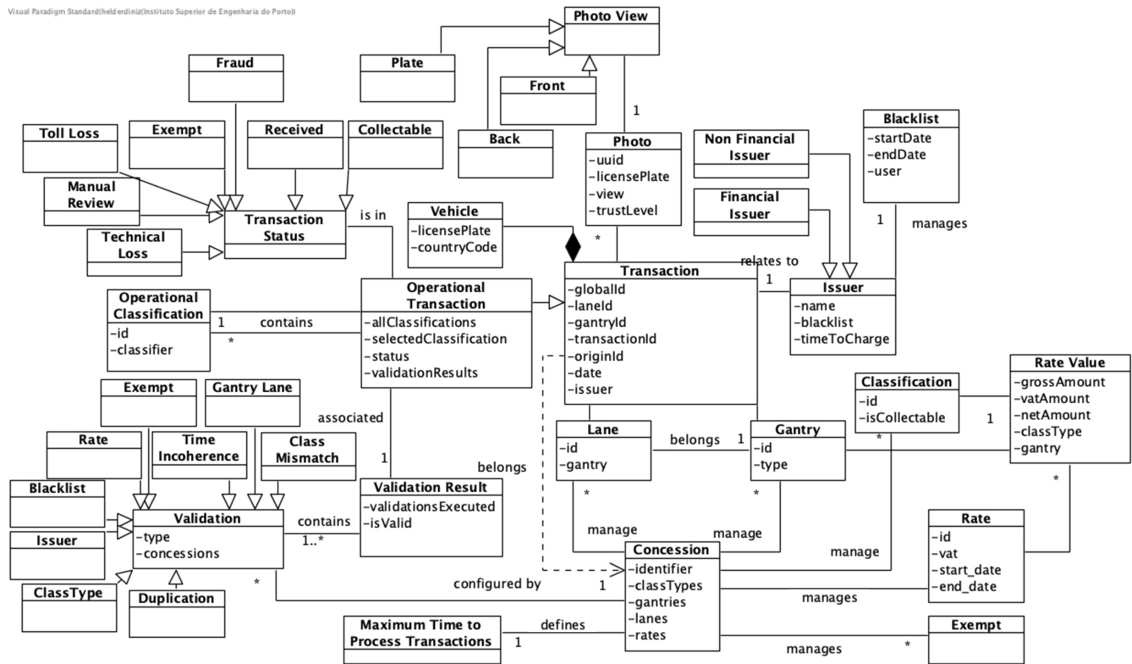

Figure 5.8: Operational domain model

Commercial Back Office (CBO) domain

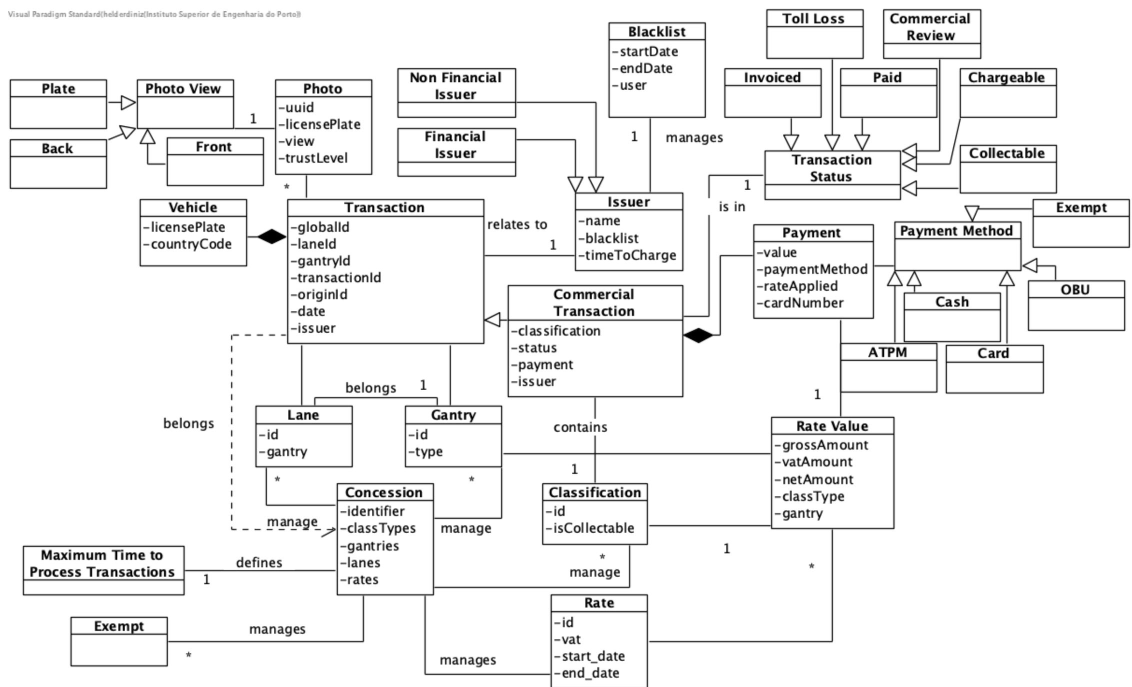

Figure 5.9: Commercial domain model

5.2.2 Concepts

This section has the goal of describing the key concepts represented in the domain model above, dedicating a subsection for each concept. Subsequently, the rationale on how the concepts relate to each other is described.

5.2.2.1 Concession

The domain representation of a concession (ref. 2.1.1) in the system. This concept is a critical part of the information, as its attributes are used throughout the whole application. A concession is classified by an identifier value that follows the 36 character Universally Unique Identifier (UUID) standard, as well as responsible for managing its gantries, lanes, classifications and rates charged. Additionally, a concession also configures the validations applied to the transactions that happen in their motorway infrastructure.

Gantry

Gantry is a motorway assembly structure that is mounted over several lanes, intended to display overhead traffic information. Specifically in the case of Globalvia motorway infrastructure, these gantries also introduce tolls and contain traffic monitoring systems and cameras. This concept is represented by an identifier value set by the concession, a type, representing if the gantry is of manual or automatic tolling, and it contains one or more lanes.

Lane

A lane is part of a motorway gantry, that is designated to be used by a vehicle in order to use the motorway toll services. Depending on the type of gantry, the lane can have cameras, toll booths or Automatic Toll Payment Machine (ATPM) interfaces. This concept is characterized by an identifier value set by the concession and the gantry it belongs to.

Classification

A classification is a category of a vehicle that is allowed to use the motorway infrastructures. These classifications can be based on vehicle body style, number of doors and even passenger capacity. A classification scheme is often created by the government departments in order to establish a consistent charging model to equivalent vehicles.

This concept is identified by an identifier value, set by the concession but its value is predefined in the government classification scheme. Additionally, it has a `boolean` attribute identifying if the classification is collectable, which means that a classification matches the government scheme, and not custom concession classifications.

Rate

The concept rate is the representation of values to be paid in the highway in a time interval. It contains multiple rate values, for all the gantries and class types combinations.A rate value is a single record that contains the monetary value to pay for a class type for a given gantry.

Exempt

The concept exempt is the representation of an entity that is entitled to use the motorway infrastructures with no toll charge. It is managed by the concession, and an exempt could be identified by its license plate, or a special OBU device placed in the vehicle. An exempt is typically police cars, ambulances, firefighter trucks or even concession's work vehicles.

5.2.2.2 Transaction

Transaction is the key concept of the system, it represents a passage of a vehicle in a specific toll gantry and lane. It is uniquely identified by three attributes `laneId`, `gantryId` and `transactionId`. The `laneId` and `gantryId` fields are a reference for the concession's lane and gantry, while the `transactionId` is a sequential identifier valued generated by the RSE component. These three attributes will generate an additional attribute, `globalId`, which is a UUID to easily identify a unique transaction for a specific concession throughout the system. This entity also includes a date attribute that represents the vehicle's passage date, as well as vehicle's information and photos captured by the RSE Optical Character Recognition (OCR) component.

The attributes aforementioned are the base of all transactions in the system, whether being included in the OBO or CBO domain. This data will then take a concrete structure in the domain it belongs to.

Operational Transaction

The entity Operational Transaction is an extension of the base Transaction, which contains the necessary attributes to work in the OBO domain. It is characterized by a list of classifications interpreted by the RSE OCR component, the RSE OBU reader and an additional class provided by the toll both operator (if applicable). Furthermore, any operational transaction will have a status field, as well as the list of validation results.

Every passage that is registered by the RSE component is considered an operational transaction, as the OBO domain is the first layer of data processing. If the transaction is also considered in a commercial nature, then it will propagated to the CBO domain. An example of an operational transaction is a concession worker passing on the toll booth, while being registered as an exempt in the system.

Commercial Transaction

Similarly to operational transaction, the entity Commercial Transaction is an extension of the base Transaction, but aimed for the CBO domain. It is characterized by a single classification, which is selected during the validation process of an operational transaction. Additionally, it contains a status field, payment and issuer information.

A passage is considered a commercial transaction if it does contain payment information, or uses an OBU supported by the origin concession. An example of a commercial transaction is a regular user of the motorway, passing on automatic toll booth with an OBU on its vehicle.

5.2.2.3 Transaction Status

The entity Transaction Status is found in the two predominant domains, OBO and CBO, embedded in the Transaction entity. It represents the status of the transaction in the specific domain, containing distinct possible values for each domain. However, some statuses are shared between domains, such as the Collectable and Toll Loss status.

In the following figures 5.10 and 5.11, state diagrams are presented that show the status transitions for each domain.

OBO Status

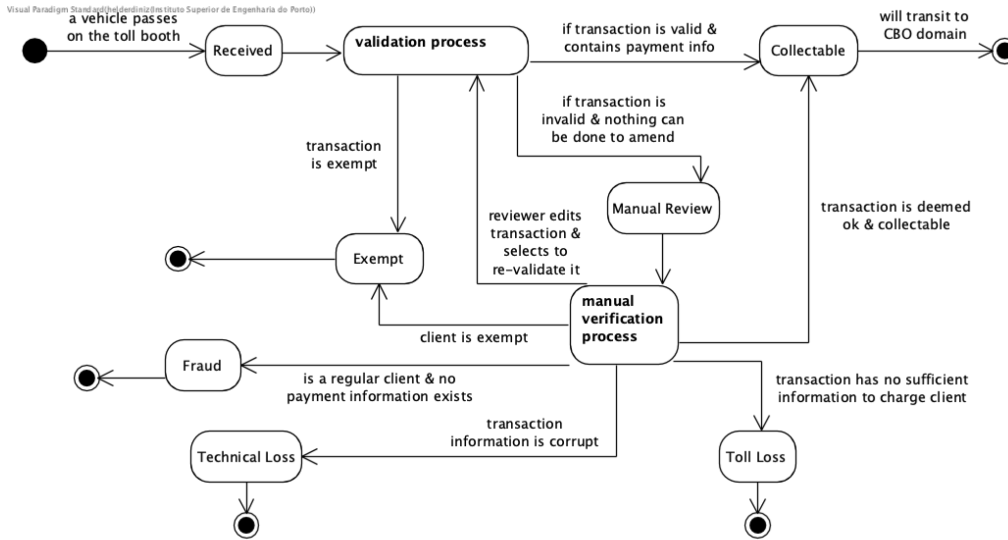

Figure 5.10: Operational status state diagram

CBO Status

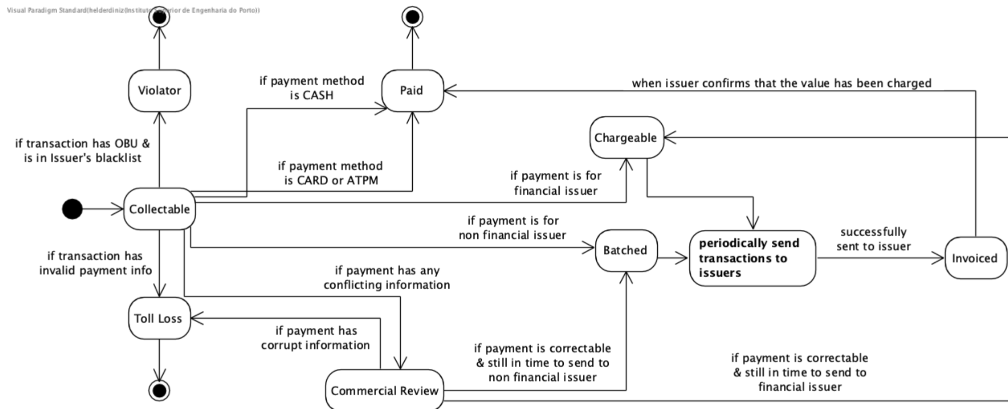

Figure 5.11: Commercial status state diagram

Some statuses are shared between domains, such as Collectable and Toll Loss. In the case of the collectable status, it is achieved when the OBO domain components have deemed that the Transaction contains the necessary information to move the information to the CBO domain, whereas on the CBO domain, the collectable status is the start of the operation.

5.2.2.4 Photo

The domain representation of a photo taken to a user of the motorway, in the exact moment the vehicle crosses the RSE OCR component placed on the gantry. This entity is characterized by an identified value formatted as an UUID, the vehicle's license plate and a value of the OCR's trust level of the recognized license plate. Additionally, this entity also includes an attribute identifying the photo view, with the following possible values: front, back and plate.

5.2.2.5 Payment

Included in the Commercial Transaction entity with a composition relationship, the payment entity contains all the information related to a possible client or issuer bill. It is defined by the card number (OBU or card), the payment value, the rate applied in the RSE and validated in the OBO domain. Subsequently, it also includes the payment method, that directly correlated with the card number. The possible payment methods are: cash, OBU, card, exempt and ATPM.

5.2.2.6 Issuer

An issuer is an institution which provides a service for highway clients, to facilitate the use the motorway infrastructures. This service is provided by either the purchase of an OBU, placed in the client's vehicle, or by facilitating an institutional card to the client, in order to use it to pay the charges of toll booths.

Issuers can take form of financial or non-financial, in which the different is dealing with cards or OBUs, respectively. The entity is also classified with a name and an attribute indicating the maximum time to bill the client, after the usage of the toll services. Additionally, an Issuer also manages a Blacklist, which indicates that a specific client is banned from using its services on a specified time interval.

5.2.2.7 Validation

The validation entity represents a business validation rule to apply to a given transaction, that is configurable per concession. These validations can take form in the following implementations:

- **Class Mismatch**
  This validation verifies if all the classifications, in the operational transaction, contain the same value. Additionally, this validation has a defined business rule that it should select the classification to use throughout the system, based a specific optional classifier that a concession can configure. The decision mechanism is represented in appendix C.
- **Gantry Lane**
  The gantry lane validation certifies if the transaction's gantry and lane fields are authentic, as well as if the lane belongs to the provided gantry.
- **Time Incoherence**
  This validation verifies if the transaction's date field is within the concession's specified limit date to process, as represented by the Maximum Time to Process Transactions entity associated with the Concession entity.
- **Exempt**
  The exempt validation ensures if the transaction was correctly marked as exempt or not exempt.

- **Rate**
  The rate validation certifies if the RSE has selected the appropriate rate value for the given Transaction's classification and transaction date.

- **Blacklist**
  This validation is responsible for verifying if the OBU, or card, registered in the Transaction is blacklisted, at a given date, in any Issuer, being financial or non-financial.

- **Issuer**
  The issuer validation is responsible for verifying if the motorway client has used any payment method that is supported by the registered Issuers in the system.

- **ClassType**
  This validation is responsible for validating the classification fields, in order to maintain consistency and the authenticity of valid classifications in the system, in order to charge the client when it arrives in a `collectable` state.

- **Duplication**
  The duplication validation will verify if there is any submitted transaction with the same gantry, OBU or card number, with a very close transaction date time range. It avoids charging the same transaction submitted in two distinct lanes by the RSE, due to vehicle misalignment on the motorway lanes.

Chapter 6

Architecture

Software architecture is considered the bridge between requirement engineering and a feasible implementation of system, while fulfilling the client's objectives. It is a plan of action to assemble components, with a certain number of architectural elements that follow a number of well-chosen patterns to satisfy the major functionality of the system. The analysis is directed at the realisation of a thorough analysis and design process of a software architecture, in order to address the problems introduced in section 1.2.

This chapter is started by a preface section, where the architectural design rationale is presented, which includes the design decisions inherent the architectural style and component technology. It is followed by an application of an architectural description model, that uses the rationale and sets de software components in a logical and physical manner. Lastly, the work methodology, caused by the implementation of the software delivery pipeline, is demonstrated.

6.1 Preface

The problem in hand is aimed at evaluating if the creation of a multi-concession, scalable toll collection system, would be a feasible solution to the problem of numerous sub-systems. The proposal is grounded on building a cloud-ready distributed system with multi-tenancy capabilities, that could easily scale on the verge of increasing work-load, aimed to replace the numerous implemented back-office systems, while keeping the Road Side Equipment (RSE) components for each concession.

The decision behind the architectural style to use, was based on the consideration of three distinct options: a single monolith, a monolith per domain and micro-services. The single monolith option, covers the implementation of a multi-concession monolith with Operational Back Office (OBO) and Commercial Back Office (CBO) domains. Monolith per domain involves the implementation of two distributed components, being one component responsible for managing operational domain operations, and another for commercial domain. Finally, the micro-services style, is based on developing a multi-concession system with small business units for each domain. Table 6.1 demonstrates pros and cons of each style.

Table 6.1: Pros and Cons of each considered architecture

Architecture	Pros	Cons
Single Monolith	- Component is developed as a unit - No asynchronous messaging needed - Business logic is centralized, or split per module	- OBO and CBO domains are mixed - Over-scaling, as it is impossible to scale per feature or component - Any change requires a new deploy of the whole platform
Monolith per domain	- Logical domain segregation - Easy to implement - Feature-rich tests are developed in a component	- Suffers from the same problems as a typical monolith - Over-scaling still happens, as the whole domain would need to be scaled
Micro-services	- Individual service scaling - Individual deployments - Responsive and fault tolerant maintenance costs	- Increased complexity in communication mechanisms - Continuous segregation must be employed, otherwise micro-services turn into mini-monoliths

A monolithic architectural style would not be a good fit for a multi-concession system, as scaling would be extremely cost and resource inefficient, and the mixed OBO and CBO domain would still be a problem. The following alternative, monolith per domain, still caused issues of over-scaling, as OBO domain does not necessarily need scaling when CBO does, and vice-versa. Conclusively, the choice of architectural style is grounded on the principles of micro-services with event driven attributes. As introduced in section 4.1.2 this architectural style offers scalability per component, allowing the services that are under heavy load to be scaled independently from the remaining domain. Additionally, its distributed nature paired together with low coupling principles, enables for logical domain segregation between services, which is a motivator factor for the OBO and CBO domain separation goal.

The design process of micro-services is started by an analysis of the domain concepts, with the intent to initiate a decomposition process to achieve fine-grained services, with strongly typed responsibilities. To induce this process, the decomposition by business capability [51] pattern is used. A business capability corresponds to a business entity, something that a business needs to generate value. This pattern is focused on executing the decomposition process by splitting the capabilities per service, which means that a service should know how to handle everything related to the business concept it fits in. Considering the photo entity as an example to apply this pattern, the micro-service that handles this entity must have the necessary responsibility to offer photo functionality, such as querying, or data storing. This process also allows to identify the data model concepts, which lead to the implementation of the database per service pattern, that ensures that each micro-service has its own data model, assuring that data mutation on the database is only executed by a single service.

Regarding the anticipated goal of a multi-concession system, in section 4.3 some patterns are presented for implementing this requirement. The pattern to choose should not be conflicting with the database per service pattern, as a means to avoid micro-service tight coupling, therefore the choice of design pattern is grounded on implementing a single configurable instance. This pattern allows for a micro-service to be the only component to interact with

its own database, however the component must have a database configuration per tenant, allowing the service to have a single database per tenant.

Taking into consideration the motivations for a cloud-based environment, pairing micro-services with cloud computing provides an unparalleled elasticity and high availability characteristics. Amazon Web Services (AWS) is the selected cloud provider, for the project described in this dissertation, due to its maturity and amount of services it offers, as well as offering fast infrastructure setup by offering fully managed infrastructure services.

Technology for software infrastructure, should also be addressed during the architecture process, as it may influence the overall architectural style of the platform to design. In regards to infrastructure, the design decision is grounded on each micro-service having its own environment, therefore each component will be executed in different containers. Software containerisation opens the possibility of using Amazon's fully managed services for service infrastructure, which lead to the adoption of Amazon Elastic Kubernetes Service (EKS)[1] to deploy and manage the containerised micro-services, using Kubernetes[2], that "works by managing a cluster of compute instances and scheduling containers to run on the cluster based on the available compute resources and the resource requirements of each container" [52]. Each micro-service will be isolated, deployed in its own pod[3] with containerisation, using Docker[4].

To describe the architecture of the software analysed on this dissertation, the architectural model "4 + 1" is discussed in detail in the following sections.

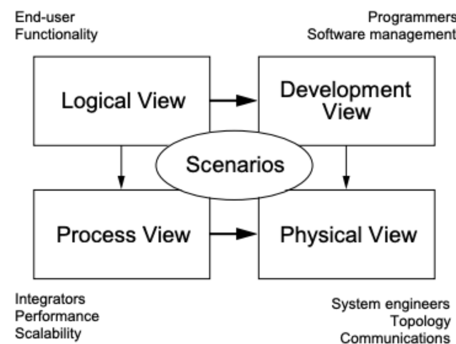

Figure 6.1: 4+1 architectural view model - Source [53]

6.2 Logical view

The logical view is intended to identify the necessary components, in order to provide the necessary functionality to the end-users of the application [53]. It is represented using Unified Modeling Language (UML) notation and provides a logical segregation of responsibilities.

[1] https://aws.amazon.com/eks/
[2] https://kubernetes.io/
[3] refer to annex D for terminology definition
[4] https://docker.com/

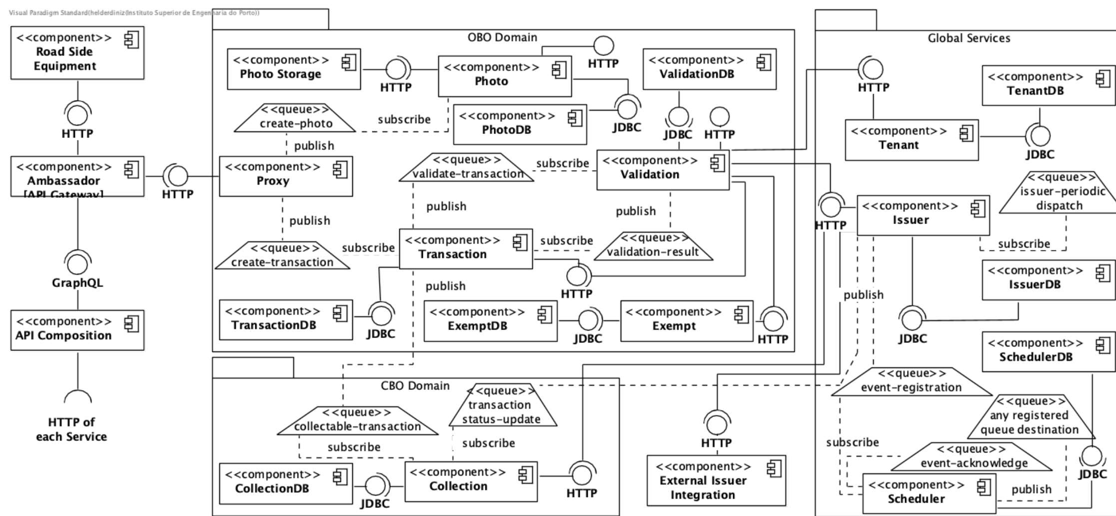

Figure 6.2: Component diagram

The component diagram represented in figure 6.2, demonstrates a system that can be logically separated in two layers: an external and an internal layer.

The external layer contains the integration of the RSE component of a concession, which should communication with the API Gateway component over HyperText Transfer Protocol (HTTP), in order to dispatch that request to the internal layer. Besides these components, the external layer also contains an API Composition component that provides information, from back-office services, for potential concession external components, dashboards or data handlers.

The internal layer is then decoupled into three distinct ones: OBO domain which include micro-services that operate exclusively over OBO operations and offer functionality, related to operational domain. CBO domain, similarly to OBO layer, this includes micro-services that offer CBO functionality. Lastly, the global services layer, offers functionality for both domains of the back-office. Each back-office micro-service provide, or consume, interfaces over three different approaches: message queuing, HTTP and Java Database Connectivity (JDBC) for data reading.

Regarding message queuing interfaces, each micro-service should contain a message broker that works in runtime, implemented inside the service. The broker should be connected to queues provided by Amazon Simple Queue System (SQS)[5] and support messages formatted using Google Protocol Buffer[6] contracts. Advantages of this approach lay down on benefits, offered by SQS, of providing a fully-scalable, highly available and fully managed queue system, that eliminates administrative overhead. Additionally, Protocol Buffers are a simple and effective approach of serializing structured data with tight schema rules, as well as providing code generation tools for several languages.

HTTP interface communication is provided by an Application Programming Interface (API) implementation in each micro-service, that consume and produce JavaScript Object Notation (JSON) content related to its entities. This interface will provide business-related information for the component that implements the API Composition pattern.

[5] https://aws.amazon.com/sqs/
[6] https://developers.google.com/protocol-buffers

Lastly, the JDBC interface is consumed uniquely by the micro-service it belongs, considering that each service contain their own isolated relational database. The database uses a MySQL[7] engine provided by the Amazon Relational Database Service (RDS)[8].

Road Side Equipment

Introduced in section 2.2.1, the road side equipment component, also named as g-toll, is the component that triggers all the functionality of the back-office operations. It is responsible for interpreting all kinds of passages in the gantries and manual toll booths, as well as taking photos of these passages, and then forwarding that information for the back-office component, through an API gateway.

API Gateway

Being one of the key components when pairing it with a micro-service architecture, as aforementioned in section 4.1.2, the API gateway component is designed to handle all requests from the external layer, into the micro-services present in the internal layer, in the back-office. The main responsibility of this component is to reroute incoming traffic to the intended services, which causes easier integrations of RSE components from future concessions.

API Composition

Similarly to the API gateway pattern, the design decision around the adoption of the API composition pattern is to handle all requests from the external layers, into the micro-services layer, through providing a simple interface to query for information by a single joined response. This component provides an interface, with a strongly-typed schema developed with GraphQL[9] technology, that provides a simple and declarative query language, that allows for fetching multiple resources in a single request, while maintaining a simplistic query idiom. As described by the creators, the operation mode of this API composition is started by describing the available data on the server side, requiring to the user to query for what's needed and provide predictable results.

Proxy

The Proxy micro-service operates in the OBO domain layer, designed to transform the RSE information into a neutral-representation and eloquent representation of transactions and photos, for the back-office entities. Proxy is the main entry point for the back-office, and it implements the adapter pattern [30], with specific information mapping for each integrated concession. The decision of creating this component is directly correlated with the decision of maintaining the RSE components for each concession, in order not to force a complete overhaul of systems for all concessions to be integrated in the multi-concession platform. Consequent to receiving the data on its HTTP API endpoints, Proxy will translate the payload to message events, and dispatch these events to message queues.

Photo

Present in the OBO domain layer, the Photo micro-service is responsible for handling all operations related to the photo entity. Its operation is prompted by consuming an event that reaches the message queue intended for photo creation, which expects for the micro-service to store the information of that photo, as well as storing the photo file. The storage

[7] https://mysql.com/
[8] https://aws.amazon.com/rds/
[9] https://graphql.org/

of photo files is achieved by the usage of Photo Storage component. Additionally, this service provides a HTTP interface to allow fetching data related to the photo entity, as well as a temporary URL to access the intended photo file.

Photo Storage

The Photo Storage component is a highly available component that facilitates the storing and fetching functionality for photo files. It is used by the Photo micro-service in order to store photo files, or to generate temporary URLs to grant access to these files. Similarly to data storage, this is also cloud-based by using an AWS service, named Amazon Simple Storage Service (S3)[10]. This storage service provides scalable, available and hybrid cloud storage with a wide-range of cost tiers, that could be configured for long or short-term.

Transaction

Being the key of the OBO domain, the Transaction micro-service is designed for handling all operations related to the transaction entity, in the operational space. It is responsible for the creation of a transaction in the system, dispatching that transaction for a validation process and then orchestrate the destination of the transaction date in the system, upon the response of the validation process. Additionally, this micro-service offers an HTTP Representational State Transfer (REST) interface that allows data fetch, as well as editing a transaction, in status manual review, if the a manual reviewer intends so. This micro-service is naturally asynchronous, as the main communication method is grounded on the usage message queuing.

Validation

The Validation micro-service is another key component, operating in the operational space. Designed as a versatile and configurable, this micro-service is responsible for applying a set of data validation rules on transaction information, in order to verify the authenticity of the information based on business rules. As figure 6.2 demonstrates, the Validation micro-service is heavily dependent on consuming HTTP interfaces from other micro-services, in order to perform the business rules, which means that the services are required to be highly available for the success of the operations. In order to mitigate these issues, the Validation service should have special concerns regarding fault tolerance, therefore, cache and retry mechanisms should be implemented.

Exempt

Present in the OBO domain layer, the Exempt micro-service is a simplistic component that provides functionality to manage exempt motorway users. This micro-service does not contribute, directly, for the typical incoming transaction flow. However, it does provide for a HTTP interface for Validation service to consume, in order to identify if a certain transaction should be considered as an exempt, as well as for API composition component to execute Create, Read, Update, Delete (CRUD) operations.

Collection

Collection micro-service is designed to be the core of CBO domain, as it is responsible for handling all operations related to the transaction entity, in the commercial context. This service is similar to Transaction service, but it is uniquely driven for finance-related work. The workload is triggered by an event published on a message queue, that is published by

[10] `https://aws.amazon.com/s3/`

Transaction micro-service after the transaction information has been deemed collectable. Collection service will then store and validate the transaction information, and wait for potential statuses updates by consuming an event in a different message queue. Additionally, this service provides an API interface intended to provide commercial transactions information.

Tenant

The Tenant micro-service is present on the global service layer, and is responsible for providing key concession information, such as gantries, rates and configurations, for all micro-services in the system. This service has the requirement for being highly available, as most components of the system rely on acceptable response times to validate concession information. Similarly to other micro-services, Tenant provides an API interface for CRUD operations.

Issuer

Issuer service is responsible for keeping all issuer's related attributes, such as configurations and time to charge. It should also assure that all issuer's periodic billing tasks will be scheduled according to the issuer's preference. Additionally, this service contains the list of the cards or On-Board Unit (OBU) that are banned from using the motorway infrastructure, also known as blacklist. Issuer micro-service is placed in the global services layer due to its dependency from OBO domain, for validations, as well as the CBO domain for client billing.

This service is considered a heavy worker, considering that it is responsible for managing issuer information, register and receive client billing events, verifying which transactions are included in the issuer billing dispatching, as well as updating the transaction's statuses upon a response from the external issuer integration.

Scheduler

The Scheduler micro-service provides functionality to register jobs that trigger certain events, according to a provided cron expression. The job registration action is executed by a specific micro-service, in the back-office, by publishing in a message queue that Scheduler will consume, leading to a registration of the job in its database. Periodically, the Scheduler service will verify its registered jobs, and generate events for the provided message queue, supplemented with the provided payload to include in the message to send. Additionally, Scheduler provides a structure, in the job registration message, for the services to indicate if the events should be retried on a lack of an acknowledge message.

6.3 Process view

The process view describes how the dynamic processes of the system communicate with each other, with a relation of the component runtime behavior [53]. In this context, a process represents an instance of the system, with shared memory between threads.

In the following figure 6.3, the process view describes the runtime for a sample micro-service with a single database, HTTP interface and message queues capabilities.

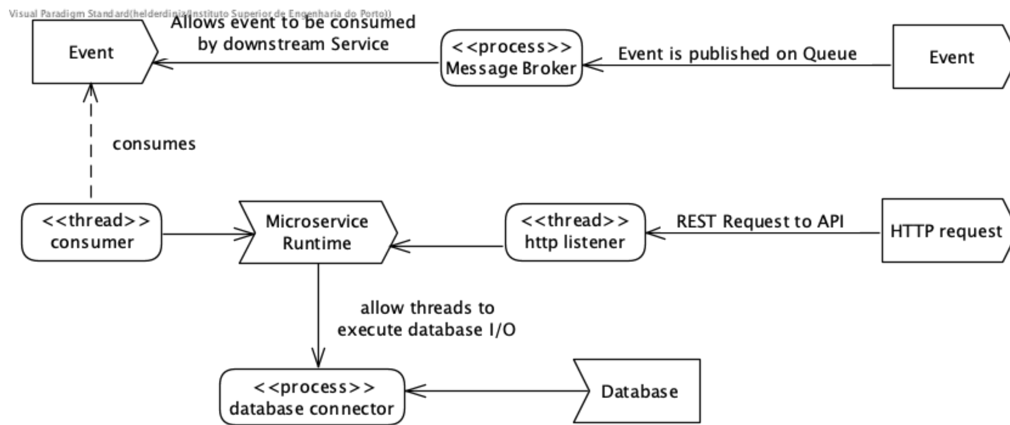

Figure 6.3: Process view of a micro-service

As figure 6.3 demonstrates the micro-service runtime contains the threads of message queue consumers, as well as the HTTP listener threads. On each inbound HTTP request, or event message, the runtime will execute that request in a specific concurrent thread, generated by a thread pool executor. Additionally, there is a process dedicated for the message broker, that will dequeue message events, and dispatching these events for consumer threads. The message queue is completely independent from the micro-service runtime, this is due to the usage of the Amazon SQS technology.

6.4 Development view

Development view illustrates the implementation structure of a given component, by presenting the relationships between the existing packages [53]. This view is mostly concerned with software management as a whole, and is developed in the programmer's perspective, in order to define a standard of organisation for current or future components.

In the following figure 6.4, the development view describes the packaging for a sample micro-service with database, HTTP interface and message queues capabilities.

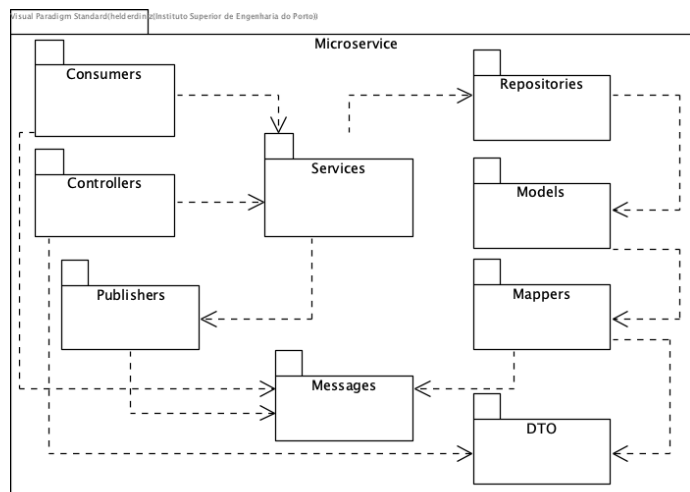

Figure 6.4: Development view of a micro-service

Consumers

The consumers package is designed to contain all necessary classes to consume a message from the queue system, that the broker has selected. The classes are responsible for parsing the message, validating it and then dispatching the relevant data for the Services package.

Controllers

This package is aimed at holding classes that handle requests consumed by the HTTP interface. These classes should parse the message, validating and sanitizing the input, and then dispatching the request for the Services package.

Publishers

The publishers package is designed to contain classes that create event messages, in order to publish them into a given destination message queue. Similarly to the Consumers package, these classes are connected with the message broker process in order to publish the messages.

Services

Each class in the services package, is responsible for forwarding data to other services, or handling business logic that result in dispatching information to the repository layer. Additionally, this package may also contain classes to dispatch information to external services.

Repositories

The repositories package contains all classes that handle data from the persistence layer, whether being read or create operations. Each operation is based on a model class, that is usually considered domain.

Models

The models package is present on the business data layer, and it contains class representation of domain concepts. Additionally, these classes are used as data bags in the repositories package.

Data Transfer Object (DTO)

Being present on the data layer, the DTO package contains all classes that are responsible for providing relevant data, to consuming down-stream services, in a efficient and simple way. These classes can be seem as a projection of a model class's attribute, including only what it should be seen by clients, to avoid exposing business logic.

Mappers

The mappers package contains all classes that apply transformation functions on data, in order to generate a distinct object. In this context, a mapper will transform a model class into a DTO or a message class.

Messages

Each existing class in the messages package, represents an event used in message queuing communication. Therefore, a message class is also a Protocol Buffer schema representation. These classes are used in publisher and consumer packages.

6.5 Physical view

Reflected from the system engineer's point of view, the physical view describes the software components, and its connections, in a physical context [53]. In the following figure 6.5, a deployment diagram displaying the physical view is observed.

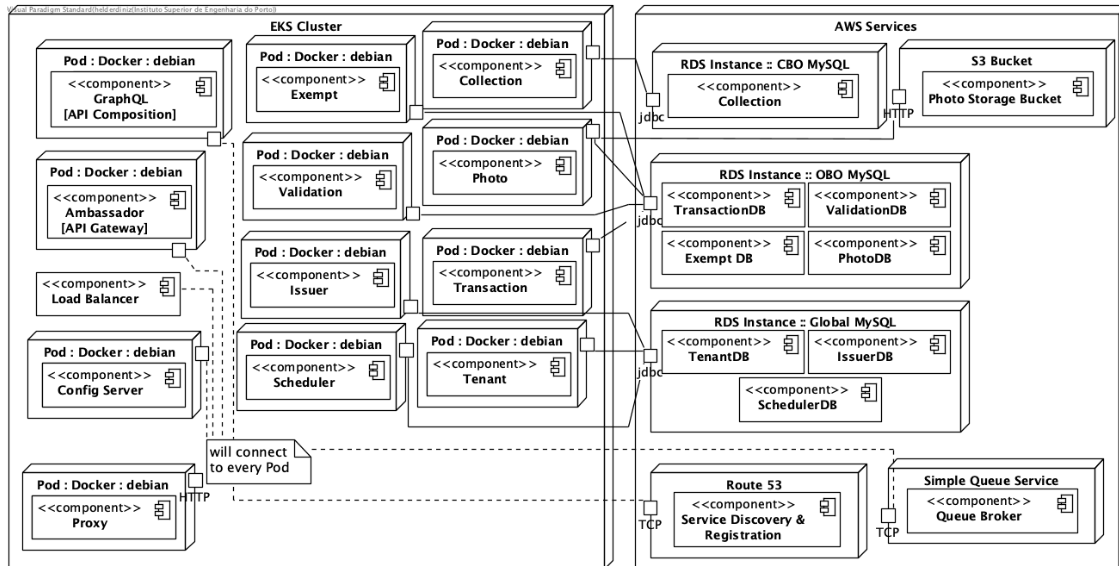

Figure 6.5: Deployment diagram of the multi-concession platform

The micro-services are deployed on a Kubernetes cluster, provided by the fully managed Amazon EKS, as figure 6.5 demonstrates. EKS does automatic calculations to determine the number of nodes needed to support the system, in order to leverage the computing power efficiently. The calculation is based on the amount of deployed pods, the configuration value set on `ReplicaSet` (amount of pod instances), and the resource limits provided by the Kubernetes provision files. Each micro-service contains a single deployment process, which means that each service is deployed as a unit, whereas Kubernetes will perform a rolling update, which means that a new pod will be created, containing the newly deployed code, with the older pod being terminated after the indication of success for the deploy. Another essential point for the micro-services, is that for each incoming request, Kubernetes will automatically lead it through the load balancer component, in order to distribute the workload for the existing pod instances.

As previously described in section 6.2, the RSE component is deployed on the motorway infrastructure, for each gantry, therefore an entry-point for certain services in cluster must be provided to allow external communication. Amazon Route 53 Service[11] facilitates the registration of domain names present in Kubernetes ingress resources, by running service discovery tasks and automatically validating name-spaces on each pod deployment. This service will register micro-services that are logically exposed to external communication, most specifically Ambassador and GraphQL.

Regarding the data storage facilities demonstrated, on figure 6.5, present the AWS services node, three MySQL database instances are deployed in the system, provided by Amazon RDS. Each instance provides connectivity to a specific domain, having one for the OBO

[11] `https://aws.amazon.com/route53/`

domain, another for CBO domain, and lastly one instance for the services that work on both domains.

In addition to data storage facilitated over MySQL, the system also takes advantage of an object storage service, provided by Amazon S3. This service offers a highly available, and scalable storage interface and it accessed by the Photo micro-service over HTTP.

6.6 Use case view

The use case view, also known as scenarios, provides a description of the system's architecture by demonstrating a small set of functionality use cases [53]. Scenarios describe an abstraction of sequences and interactions between services, components and processes.

In order to understand the rationale behind the architectural decisions aforementioned, providing a context during the scenario development is important. It has been defined that the most architecturally relevant use cases are the ones that deal with the core parts of the system, as well as provide the most value for the solution. In figure 6.6, a representation of these use cases are illustrated.

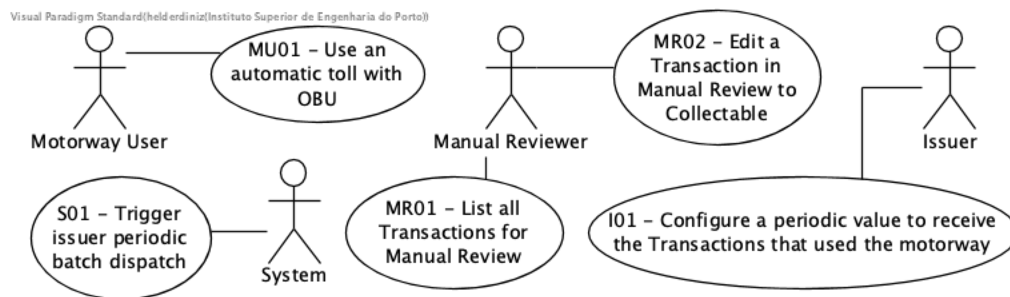

Figure 6.6: Architecturally relevant use case diagram

In the following sections, each use case is described in detail, containing a clarification on how the use case is architecturally relevant, as well as a sequence diagram explaining the functionality and how the components interact with each other. The selected use cases contain interactions from several components of the system, so sequences that happen inside the micro-service's packages are omitted to promote readability.

6.6.1 MU01 - Use an automatic toll with OBU

This use case is considered to be the most important functionality-wise, and the most architecturally relevant because it interacts with most of the components of the system. As previously mentioned at section 5.1.2.5, this use case allows the motorway user to use a vehicle with a mounted OBU to use the motorway infrastructure. The use case presents the following prerequisites:

- An allowed vehicle is used, and classifiable by law;
- The vehicle contains a mounted OBU;
- The mounted OBU is certified by an Issuer supported by the Concession.

In order to understand how the use case will handle the actor's action, and handle the functionality it provides, a sequence diagram is present, in figure 6.7, to demonstrate the communication between components of the system, upon an immediate passage on the toll booth.

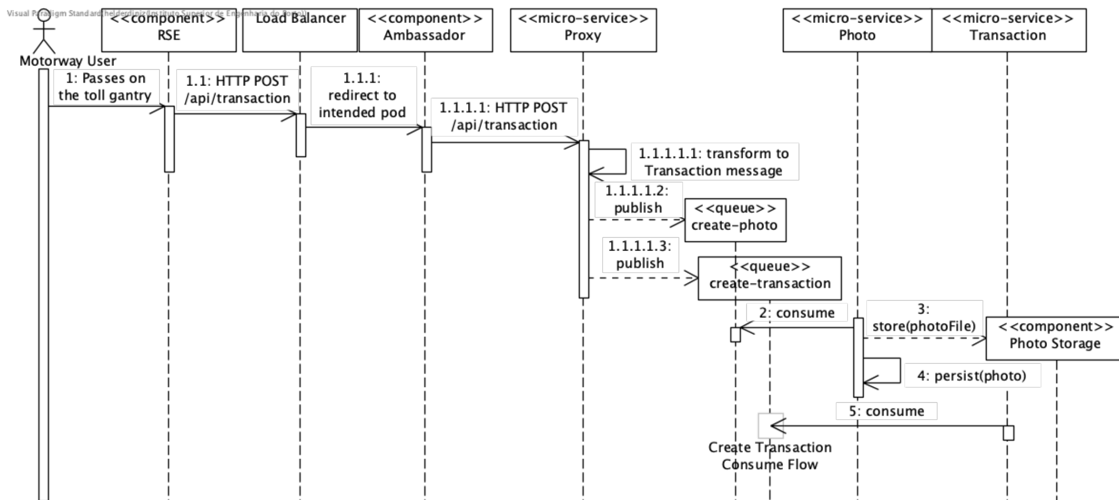

Figure 6.7: Sequence diagram for MU01 use case

The sequence of actions triggered by invoking the functionality, will then proceed on executing both synchronous and asynchronous operations, in two different stages, until it is stored on Transaction and Collection micro-services. The first stage of the sequence diagram - synchronous operations - can be interpreted as following:

1. A motorway user passes on the toll gantry using a mounted OBU on its vehicle;
2. RSE validates and dispatches the information for the API gateway component (Ambassador);
3. API gateway will forward the request for Proxy micro-service;
4. Proxy will then identify in which concession the transaction originated, fetch the schema for that concession's RSE and transform into a message that is globally understood by back-office micro-services and publish it on a message queue aimed at transaction creation. Proxy should, in parallel, do the same actions for photo creation.

Following the message publication on `create-transaction` and `create-photo` queues, the second stage of the sequence is started, which contains asynchronous operations.

Photo micro-service should consume messages present on `create-photo` queue, transform the message payload into a photo entity, store it in the database, and then uploading the file to Photo Storage component, provided by Amazon S3.

Transaction micro-service consumes messages present on `create-transaction` queue, transform the message payload into a transaction entity, store it in the database with status `received`, followed by publishing a message event in `validate-transaction` queue.

Validation micro-service is followed on the pipeline of asynchronous operations. It should consume messages from `validate-transaction` queue and apply business validation rules

for the transaction information present on the message. A sequence diagram illustrating the operations applied to a message is demonstrated on figure 6.8.

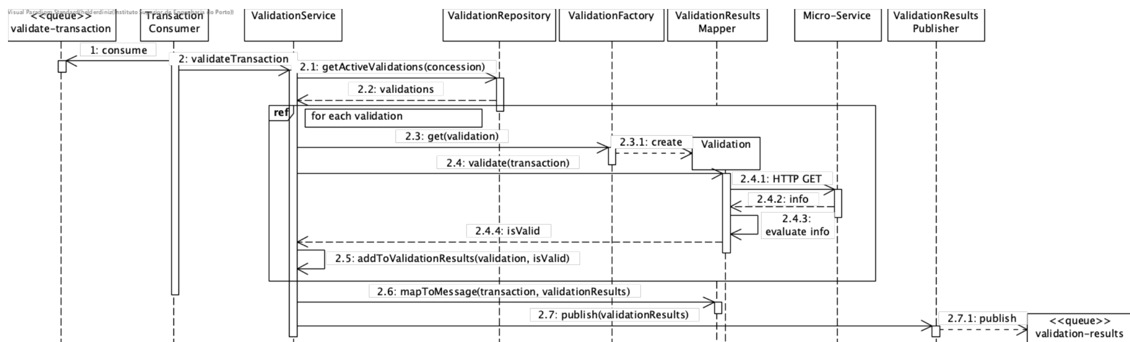

Figure 6.8: Sequence diagram for MU01 use case - validation processing

Based on the sequence diagram demonstrated on figure 6.8, the micro-service's processing operations can be summarized as following:

1. Consume a message from the queue `validate-transaction`;
2. Retrieve active validations for the concession of the transaction;
3. For each active validation, fetch information from a specific micro-service (e.g. for an Issuer validation, it should verify on Issuer micro-service if the provided OBU number belongs to a valid Issuer);
4. Transform the generated validation results into a message, in order to publish it on the `validation-results` queue.

Subsequently, as Transaction service is subscribed to the `validation-results` queue, it will consume messages published by Validation service, as demonstrated on figure 6.9. These messages contain the validation results of the transaction, which means that a transaction status can be defined, in which a decision is based on the following:

- if the only failed validation is `exempt`, then transaction status becomes `exempt` and its lifeline terminates;
- if any other validation has failed, then the transaction status becomes `manual review` and awaits manual intervention from a concession worker;
- if no validations failed and the transaction contains payment information, it should proceed to CBO domain, therefore the status becomes `collectable`.

With the condition that a transaction status is set as `collectable` by the decision mechanism, that transaction should be automatically dispatched for CBO domain, which will lead Transaction service to publish on `collectable-transaction` queue.

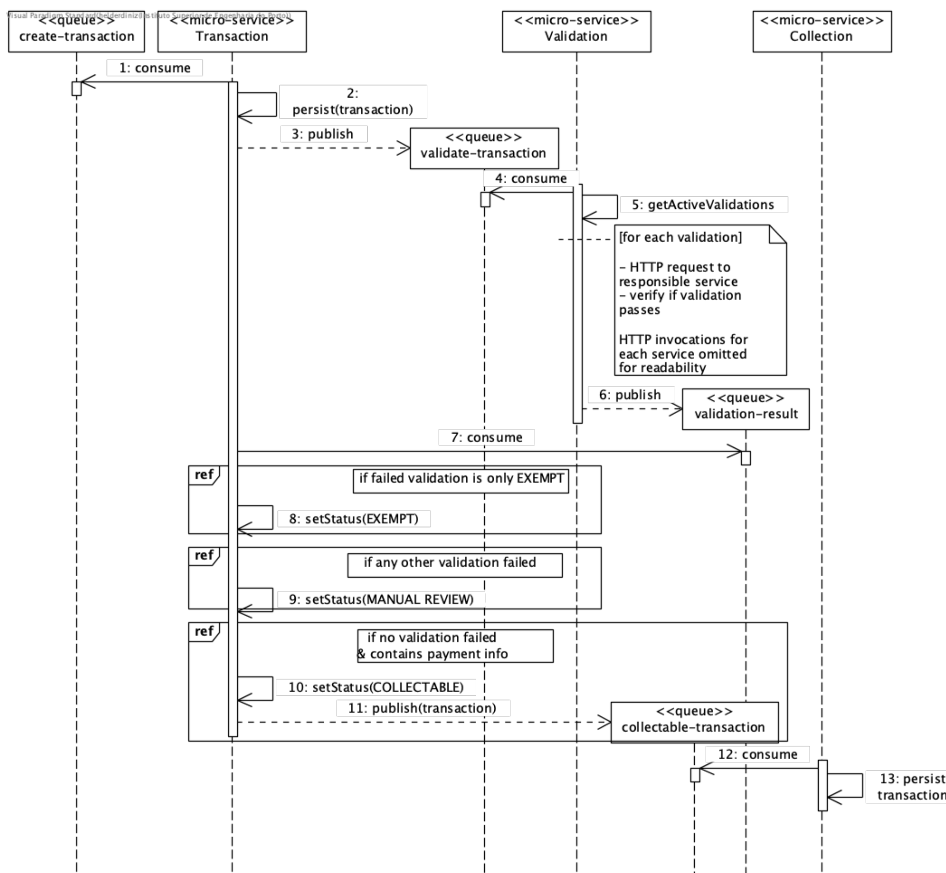

Figure 6.9: Sequence diagram for MU01 use case - Transaction Flow

Collection micro-service consumes messages present on `collectable-transaction` queue, transform the message payload into a commercial transaction entity, and store it in the database with status `collectable`. However, right after the attribution of status collectable, the Collection service will run a set of payment validations that will lead to another status change, in which a decision is based on the following:

- if transaction has invalid payment info, the `toll loss` status is automatically assigned;
- if payment information is correct and its Issuer is of type financial, the assigned status is `chargeable`;
- if payment information is correct and its Issuer is of type non-financial, the assigned status is `batched`;
- if payment method is between card, cash or Automatic Toll Payment Machine (ATPM), the assigned status is `paid` and its lifeline ends;
- if payment information has any conflicting information, the assigned status is `commercial review` and awaits manual intervention from a concession worker.

With the status successfully set on commercial transactions consumed from `collectable-transaction` queue, operations on Collection service ends for the time being. Eventually, Issuer micro-service will query `chargeable` or `batched` transactions in order to proceed with client billing, as demonstrated on section 6.6.3.

6.6.2 I01 - Configure a periodicity to receive the Transactions that used the motorway

The I01 use case has been classified as architecturally relevant, due to its nature of offering functionality similar to CRUD. It is focused on allow a Issuer to set a periodic value to receive the Transactions from its clients. The use case presents the following prerequisites:

- The issuer has been previously approved by the concession;
- The issuer information is present on the system.

In order to understand how the use case will handle the actor's action, and the provided functionality, a sequence diagram is pictured, in figure 6.10, to demonstrate the communication between components of the system, upon action from the issuer.

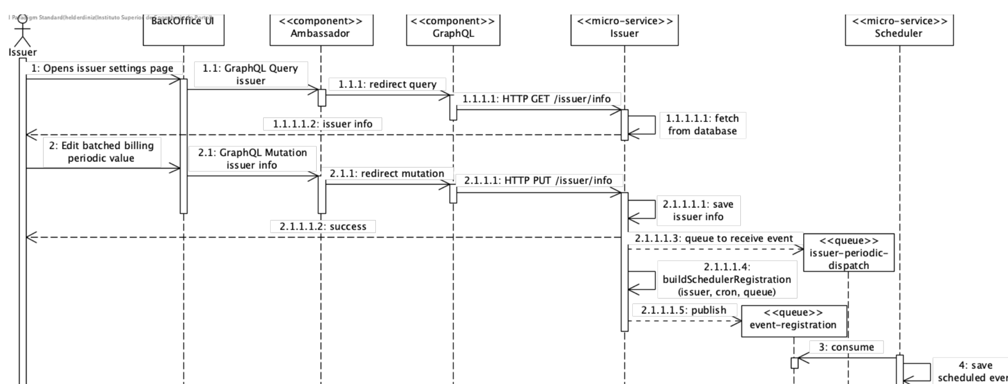

Figure 6.10: Sequence diagram for I01 use case

The use case represented on thee sequence diagram in figure 6.10, is achieved by two user actions and will trigger two operations. The first operation deals with user action and happens in synchronous matter, and can be interpreted as following:

1. An issuer opens its settings page, on the concession back-office component;
2. The back-office component will execute a GraphQL query and dispatch it to the exposed endpoint of the API Gateway, Ambassador;
3. Ambassador will forward the request to the API Composition service, GraphQL;
4. GraphQL will execute a HTTP request to Issuer micro-service, in order to fetch the issuer's information;
5. The information is displayed on the back-office, for the issuer;
6. The issuer decides to edit the batched billing periodic value;
7. The back-office component will execute a GraphQL mutation and dispatch it to the exposed endpoint of the API Gateway, Ambassador;
8. Ambassador will forward the request to the GraphQL service;
9. GraphQL will execute a HTTP request to Issuer micro-service, in order to update the issuer's information;
10. A success message is displayed, for the issuer.

The remaining of the operation happens in an asynchronous manner:

- Issuer micro-service will publish a scheduled registration event on queue `event-registration`, in order to schedule the issuer's next batch dispatch. The event is repeatable, contains retry configuration and a cron value;
- Scheduler micro-service will consume messages from queue `event-registration` and register provided scheduled events.

6.6.3 S01 - Trigger issuer periodic batch dispatch

The S01 use case has been classified as architecturally relevant, given the fact that is a scheduled action based on a value provided by the issuer. This functionality requirement was specially considered, during the design process, in order to create a scheduler component that was able to deal all kinds of system scheduled tasks. This use case is the follow up action that happens after the periodic value for issuer batch dispatching is reached, provided by the use case demonstrated in section 6.6.2. The use case presents the following prerequisites:

- The Issuer micro-service has registered a scheduled job;
- At least one commercial transaction is in `chargeable` or `batched` status for the specific time-frame.

In order to understand how the use case is triggered by the scheduled action, and the functionality perceived value is delivered, a sequence diagram is pictured, in figure 6.11, to demonstrate the communication between components of the system.

The use case represented on the sequence diagram in figure 6.10, is achieved by the interaction of three micro-services and by the issuer's external integration component. The first operation happens in Scheduler micro-service and it can be described as following:

1. A periodic polling task is executed in Scheduler service, in order to fetch for events to execute;
2. Scheduler will query its database to find executable events;
3. If any event is found, Scheduler should publish the provided message in the provided queue (for this specific use case, the `issuer-periodic-dispatch` queue is used);
4. Scheduler event will register a retry event, to be executed after a specific amount of seconds have passed if a message, from that event, is not received on `event-acknowledge` queue.

Following the published message on `issuer-periodic-dispatch` queue, in which the Issuer micro-service is subscribed to, the billing dispatch functionality is triggered, which should succeed the following operations:

1. Issuer micro-service consumes a message from `issuer-periodic-dispatch` queue;
2. Issuer executes a HTTP request to Collection micro-service fetch chargeable, or batched, transactions in the given time-frame, for that given issuer;
3. For each transaction provided by Collection with valid payment information, a message is published on `transaction-status-update` queue, indicating that the transaction is included in a issuer's invoice, in order to transition the transaction status to `invoiced`;

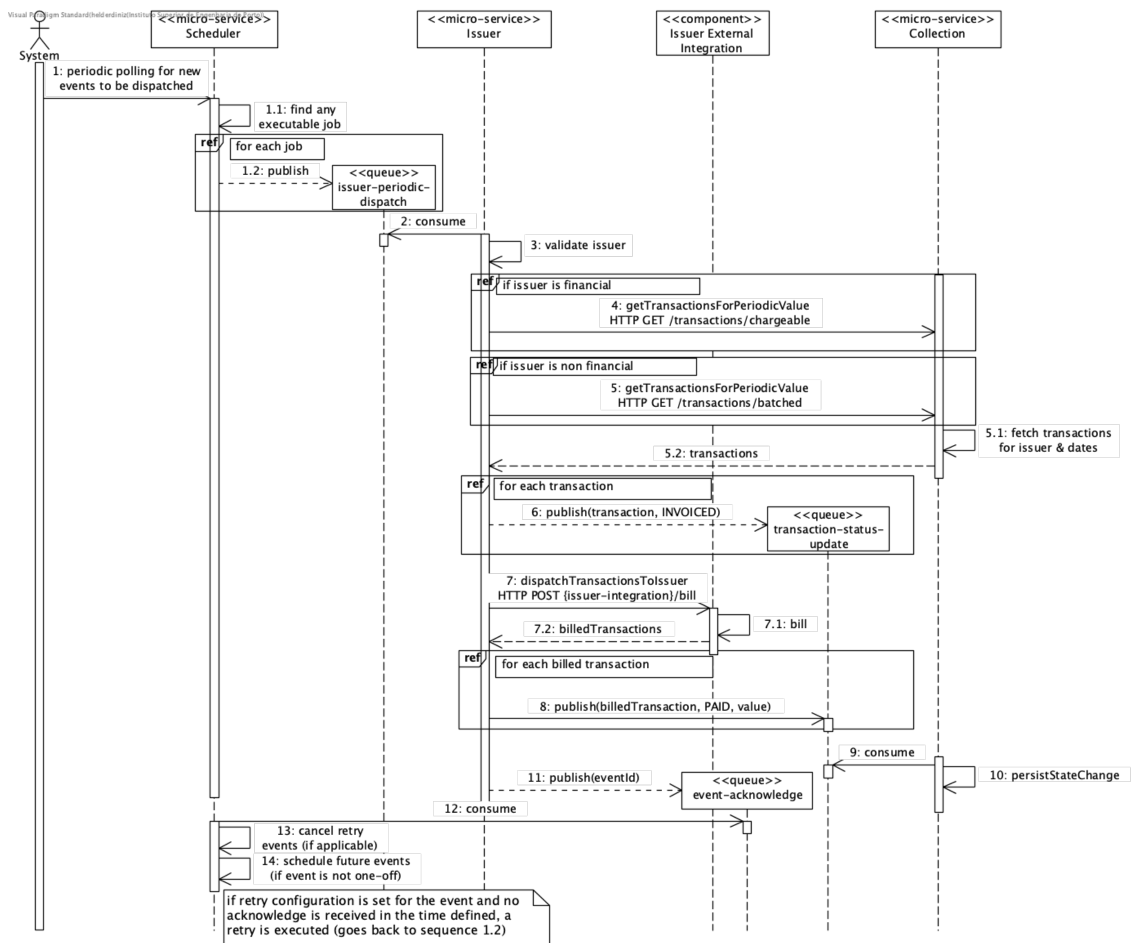

Figure 6.11: Sequence diagram for S01 use case

4. Issuer should execute a HTTP request to the external issuer integration, containing all the transaction information to bill;
5. External issuer integration will respond with the charged value per transaction, as it may change due to card limits or issuer promotions;
6. For each billed transaction, Issuer service will publish a message on `transaction-status-update` queue, indicating the transaction has been charged with a specific value, in order to transition the transaction status to `paid`.

6.6.4 MR01 - List all Transactions for Manual Review

The MR01 use case has been classified as architecturally relevant, as a result of offering a functionally that queries information from more than one micro-service, therefore, making use of the API composition component. It is focused on allowing a manual reviewer to list all transactions that are awaiting manual action. The use case presents the following prerequisites:

- At least an operational transaction is in `manual review` status.

For the purpose of understanding how the actor interacts with the system, and how the components handle the actor's actions, a sequence diagram is demonstrated in figure 6.12.

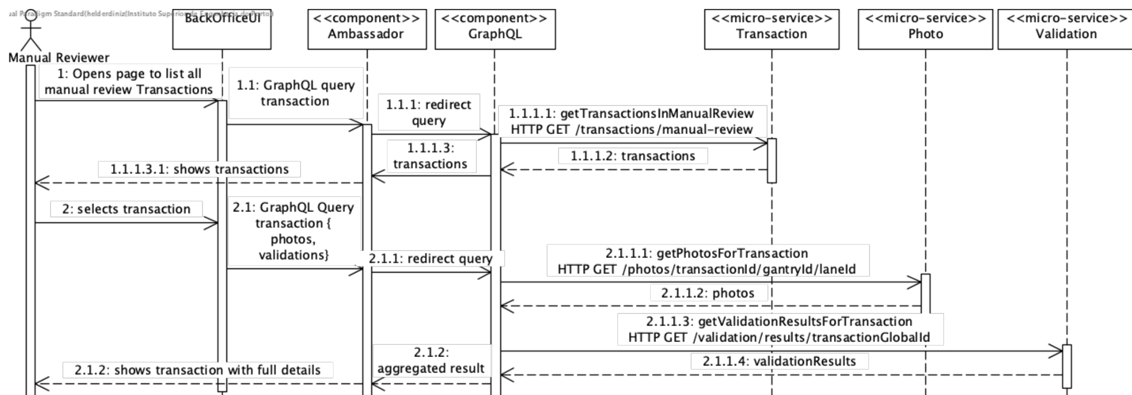

Figure 6.12: Sequence diagram for MR01 use case

This use case full functionality is achieved by two preceding actions, one being the listing of all `manual review` transactions. The first action can be interpreted as following:

1. The manual reviewer opens the `manual review` transactions page;
2. Back-office component performs a GraphQL query, and dispatches it to the exposed endpoint of the API Gateway, Ambassador;
3. Ambassador will forward the request to the API Composition service, GraphQL;
4. GraphQL will execute a HTTP request to Transaction micro-service, in order to fetch all operational transactions in status `manual review`;
5. The information is displayed on the back-office, for the manual reviewer.

Succeeding a successful listing of the operational transactions in `manual review`, the actor's second action is to get a listing containing the full details of a transaction, which should include the photos and validation results. This action will trigger the following sequences:

1. Back-office component performs a GraphQL query, and dispatches it to the exposed endpoint of the API Gateway, Ambassador;
2. Ambassador will forward the request to the API Composition service, GraphQL;
3. GraphQL will execute two HTTP micro-services requests, one to Photo and another to Validation, in order to fetch all photos and validation results for the given transaction;
4. GraphQL will aggregate the results and display it in a single response.

With a successful listing of operational transaction details, the following intention of the manual reviewer is to edit the transaction, in order to proceed in the processing pipeline, by manually setting a terminating status to it or add information, in order to be re-validated. The use case functionality should be present in this use case, however. The follow-up use case is covered in the following section 6.6.5.

6.6.5 MR02 - Edit a Transaction in Manual Review to Collectable

This use case has been classified as architecturally relevant, as a result of being a continuation of the line of action that happens in use case MR01, described in section 6.6.4. It is focused on allowing a manual reviewer to edit a transaction, and enabling the transaction to be transitioned to a `collectable` status.

This use case presents the following prerequisites:

- At least an operational transaction is in `manual review` status;
- The manual reviewer has selected which transaction to edit.

For the purpose of understanding how the actor interacts with the system, and how the components handle the actor's actions, a sequence diagram is demonstrated in figure 6.13.

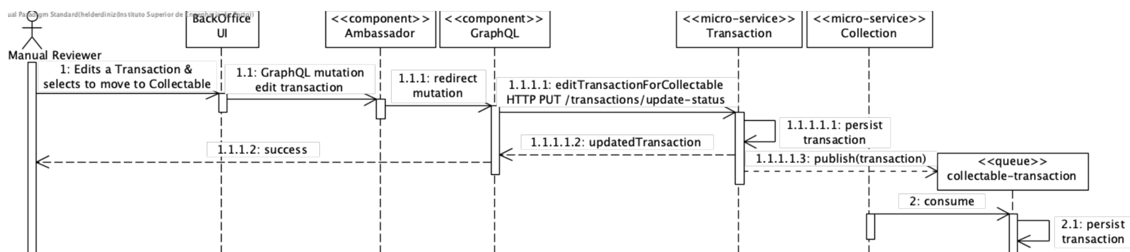

Figure 6.13: Sequence diagram for MR02 use case

The functionality of this use case is offered by the interaction of three micro-services, with both synchronous and asynchronous operations. The first operation deals with user action and happens in synchronous matter, and can be interpreted as following:

1. The manual reviewer edits the transaction and selects the open to set the status as `collectable`;
2. Back-office component performs a GraphQL mutation, and dispatches it to the exposed endpoint of the API Gateway, Ambassador;
3. Ambassador will forward the request to the API Composition service, GraphQL;
4. GraphQL will execute a HTTP request to Transaction micro-service, in order to update its status and the operational transaction information;
5. Transaction executes a synchronous data persistence operation and returns the updated transaction;
6. A success message is displayed on the back-office, for the manual reviewer.

The persistence operation that happens on Transaction, will trigger an event message that is published on `collectable-transaction` queue, with the intent of transitioning the transaction to the CBO domain. This message h is then consumed by Collection micro-service and persisted in its database.

6.7 Software Delivery

Considering the objectives presented in previous sections, one of the main drives for this dissertation was developing a work methodology that could deliver software in fast and continuous manner. In section 4.2.2, concepts for software delivery were introduced, in order to understand the existing solutions for a possible new methodology. The solution is grounded on the adoption of Continuous Integration (CI) & Continuous Delivery (CD) patterns, following principles to automate the build and deployment processes.

An overview of the new CI & CD process, to implement, is presented in a pipeline diagram in figure 6.14.

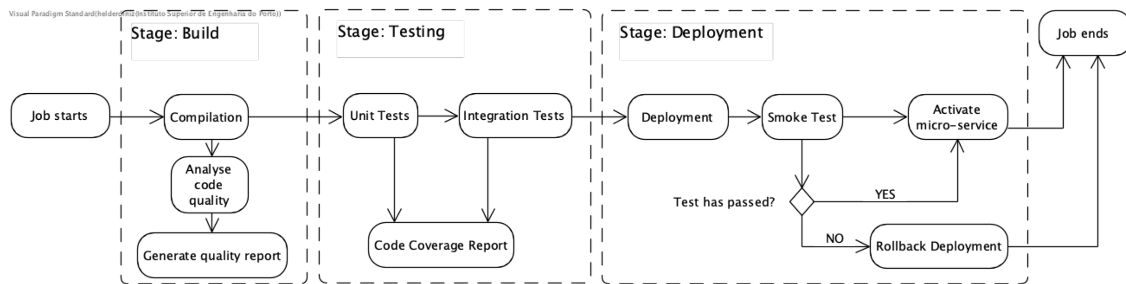

Figure 6.14: CI & CD pipeline design

The designed pipeline is composed by three distinct stages, build, testing and deployment.

Coming first in the pipeline, the build stage is responsible for compiling the solution with the newly integrated code, and for analysing code security and quality, by applying technology that helps highlights issues found on new code, such as SonarQube[12]. Additionally, this stage should generate a code quality report as its artifact.

Following the build, comes the testing stage. This stage should run all existing tests on the source code, such as unit tests and integration tests. Code coverage reports should also be generated, in order to provide a general overview of the testing coverage for the micro-service.

Lastly, the deployment stage is executed. This stage is responsible for delivering the software. Alternatively stated, this step of the pipeline is aimed to deploy the micro-service into the Kubernetes cluster. It should be prepared to execute the deployment and run a smoke test. If the smoke test fails, the deployment stage should trigger a rollback operation, in order to cancel the deployment. If no error occurs the job should end successfully and its result be reported for the interested people, regardless of the result.

[12] https://sonarqube.org/

Chapter 7

Implementation

The implementation chapter concerns the most technical component of this dissertation. This chapter is focused on explaining the design decisions, present in chapters 5 and 6, in a technical manner. Starting from the infrastructure decisions and implementation details, proceeding to good practices, patterns and standards that were considered and implemented for micro-services, such as configuration, pipelines and even multi-tenancy implementation details. Finally, the concepts of Application Programming Interface (API) composition and API gateway are presented, as well as the decisions that lead into their implementation.

7.1 Infrastructure

This section is focused on demonstrating the implementation process for the project's infrastructure. It is started by showcasing how the principle Infrastructure as Code was applied and implemented throughout the creation of the infrastructure process. It is followed by a detailed explanation on the implementation of the Amazon Elastic Kubernetes Service (EKS) cluster, and its underlying dependencies. Subsequently, the implementation of the relational database is demonstrated, as well as the decisions regarding the service used. Moreover, an overview of message queuing and its implementation is provided, followed by demonstrating the decisions and implementation of the file storing component.

7.1.1 Infrastructure as Code

Previously presented at section 4.2.4, Infrastructure as Code (IaC) is an approach to manage infrastructure through definition files, following a specific technology, in order to allow a blueprint of the project's infrastructure, as well as providing versioning, treating it as code.

The implementation process of IaC pattern is grounded on the usage of Terraform[1], which is a tool that allows teams to take advantage of the IaC pattern by providing routines to build, change and version infrastructure in a safe and efficient way [54], by managing cloud provider services directly.

Terraform configuration files can describe the necessary components to run an entire datacenter, grouped applications, or even describe single instances of infrastructure to provide them as services. It starts by analysing the configuration file, in order to outline a plan of action that needs to be executed in order to reach the desired state, and then executes the

[1] `https://terraform.io/`

plan to build, create or edit resources. Cloud infrastructure tends to increment over time, Terraform is able to keep up with the configuration changes and determine what changed, and create incremental execution plans that can be applied at any time. Additionally, Teraform also offers fault tolerance behavior, by providing rollback functionality in the case of any error arises during the plan execution.

In order to understand Terraform's key features, this tool introduces some key concepts that need to be introduced to understand the flow of action.

Resource Graph

A Terraform resource graph is a dependency graph built from configuration files, that is meant for Terraform to walk over the graph, in order to generate plans or refresh the infrastructure states [55]. This graph contains all infrastructure resources, which makes Terraform take advantage of parallel computing for the creation and modification of non-dependent resources.

Execution Plans

Execution plans is a routine, that Terraform handles with before executing any kind of action on the infrastructure. It is a planning step that evaluates the resource graph, and generate a plan output that will showcase what infrastructure changes are going to be created, deleted or modified.

The execution plan can be accessed by the command `terraform plan`, that should provide a list of entries, like a change log, in order to verify the infrastructure change matches the expectation. The following code listing shows an example of an execution plan that should delete a queue, and modify a specific attribute of a queue.

```
An execution plan has been generated and is shown below. Resource actions are indicated with the
    following symbols:
  ~ update in-place
  - destroy

  # aws_sqs_queue.get-pan-list-dev will be destroyed
  - resource "aws_sqs_queue" "get-pan-list-dev" {
      - arn              = "arn:aws:sqs:eu-west-1:240391193615:get-pan-list-dev" -> null
      - id               = "https://sqs.eu-west-1.amazonaws.com/240391193615/get-pan-list-dev" -> null
      - max_message_size = 262144 -> null
      - tags             = dev -> null
    }

  # aws_sqs_queue_policy.dev-policy-transaction will be updated in-place
  ~ resource "aws_sqs_queue_policy" "dev-policy-transaction" {
        id               = "https://sqs.eu-west-1.amazonaws.com/240391193615/transaction-dev"
      ~ max_message_size = 262144 -> 1024
    }

Plan: 0 to add, 1 to change, 1 to destroy.
```

Listing 7.1: Example of an execution plan produced by Terraform

The execution plan can then be applied by executing `terraform apply`.

Provider

In Terraform, a provider is a module responsible interpreting API interactions and exposing resources, of that given provider. Providers also configure a specific infrastructure platform, such as cloud or bare-bones, and supply resource definitions for their services.

Providers are managed by Terraform, and available through a registry platform[2], that is publicly available to use in configuration files. Considering the project's technological stack, the provider to use is Amazon Web Services (AWS), and it can be configured by its registry

[2] `https://registry.terraform.io/browse/providers`

key, mapped with the region, account access key and secret key, as described in code block 7.2.

```
variable "aws_access_key" {}
variable "aws_secret_key" {}

provider "aws" {
  version = ">= 2.28.1"
  region = "eu-west-1"
  access_key = "${var.aws_access_key}"
  secret_key = "${var.aws_secret_key}"
}
```

Listing 7.2: Terraform provider configuration for Amazon Web Services

The provider can be setup by executing `terraform init`.

Backend

A backend resource, in Terraform, declares how the infrastructure state is loaded, where it is stored and how an execution plan is executed. This resource enables developers to store a Terraform stage in a controlled environment, in order to avoid infrastructure corruption and even keeping sensitive information safely.

Terraform support several types of backend, having provider implementations for databases, HTTP requests or even Amazon Simple Storage Service (S3). Grounded to the fact that this project's infrastructure is heavily based on AWS, the choosen backend provider is Amazon S3, as described in the following code block 7.3.

```
terraform {
  backend "s3" {
    bucket = "globalvia-terraform-state"
    key = "terraform/dev/resource_state.tfstate"
    region = "eu-west-1"
  }
}
```

Listing 7.3: Terraform backend configuration using Amazon S3

7.1.2 Cluster

During the architecture analysis and design process, presented at 6, the technology around infrastructure was considered with great detail, as using a poorly fitting clustering technique could make the advantages of micro-services architecture useless. Kubernetes is selected as the engine for the system's infrastructure process.

Kubernetes is an open source system designed to deploy, scale and manage containerized applications in a wide-spread of providers [52], by providing declarative and automated configuration, also known as provision. Kubernetes automates operational tasks of resource management, such as increasing or decreasing software instances, rolling out a deployment process and even provides tools to monitor applications, by evaluating pods[3] and services statuses.

When pairing Kubernetes technology with cloud infrastructure provided by AWS, conventional configured Kubernetes should manage clusters of Amazon Elastic Compute Cloud (EC2)[4] instances, by creating pods and running containers on a Kubernetes cluster, installed in an Amazon Virtual Private Cloud (VPC)[5], backed up by the computing power of

[3] refer to annex D for terminology definition
[4] https://aws.amazon.com/ec2/
[5] https://aws.amazon.com/vpc/

cloud instances. However, in the sense of taking advantage of Amazon cloud services, the decided is grounded on using Amazon's managed Kubernetes solution.

Amazon EKS is a fully managed solution for Kubernetes management, that offers a higher level of connectivity to Amazon services, such as Amazon VPC, EC2 and auto scaling groups. Additionally, its "management infrastructure across multiple AWS Availability Zones, automatically detects and replaces unhealthy control plane nodes, and provides on-demand, zero downtime upgrades and patching" [56].

Regarding the implementation of an Amazon EKS cluster, principles of IaC will be applied. A versioned file should be created, in order to contain all the necessary resources for the EKS cluster, as well as a backend configuration to store the file state. There are resources that need to be created together with the EKS cluster, such as a VPC, Identity and Access Management (IAM)[6] roles, security groups, subnets, auto-scaling node group, route table. In figure 7.1, a deployment diagram with a representation of the cluster is demonstrated.

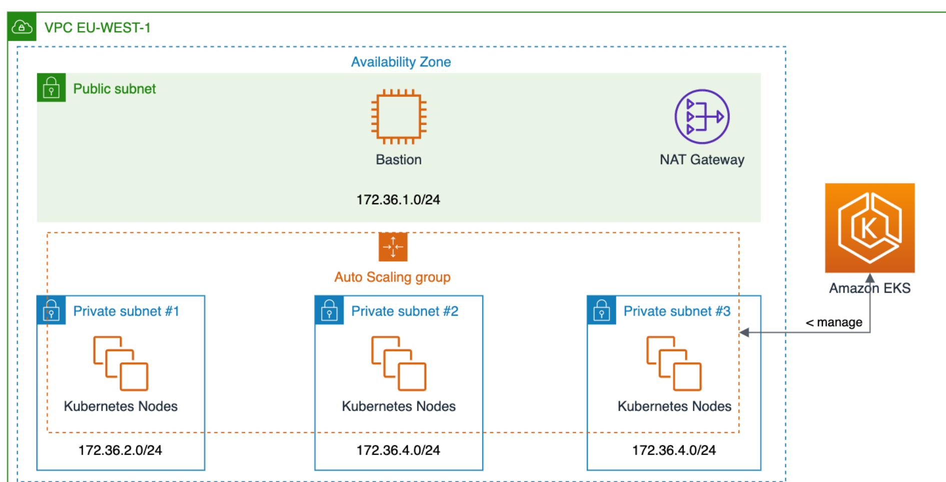

Figure 7.1: AWS deployment diagram

7.1.2.1 Virtual Private Cloud (VPC)

Amazon VPC enables the leverage of an isolated virtual network, in order to take advantage of the usage of AWS resources in a private environment [57]. This network is also represented as a subnet of Amazon EC2 instances, that act as the private network in the cloud, which means that using a VPC keeps the benefits of AWS scalability.

The VPC is the mainframe for the EKS cluster, in which it will be operated in. The configuration grounded on VPC includes subnets, security groups, routing tables and NAT gateway. The following code listing showcases the creation of the VPC that will be used by the EKS cluster.

```
module "vpc" {
  source  = "terraform-aws-modules/vpc/aws"
  version = "2.6.0"

  name                 = "gbos-stage-vpc"
  cidr                 = "172.36.0.0/16"
  azs                  = data.aws_availability_zones.available.names
  private_subnets      = ["172.36.2.0/24", "172.36.3.0/24", "172.36.4.0/24"]
  public_subnets       = ["172.36.1.0/24"]
  enable_nat_gateway   = true
```

[6] https://aws.amazon.com/iam/

```
11   single_nat_gateway   = true
12   enable_dns_hostnames = true
13
14   tags = {
15     "kubernetes.io/cluster/${local.cluster_name}" = "shared"
16   }
17
18   public_subnet_tags = {
19     "kubernetes.io/cluster/${local.cluster_name}" = "shared"
20     "kubernetes.io/role/elb"                      = "1"
21   }
22
23   private_subnet_tags = {
24     "kubernetes.io/cluster/${local.cluster_name}" = "shared"
25     "kubernetes.io/role/internal-elb"             = "1"
26   }
27 }
```

Listing 7.4: Amazon Virtual Private Cloud configuration

7.1.2.2 Node Groups

Node groups is a key concept for EKS, as it has the responsibility of automating the provisioning and lifecycle instance nodes, specifically EC2 instances [58]. This resource is configured together with a scaling configuration, indicating the minimum and maximum number of nodes to scale. This will allow EKS to scale the nodes automatically, based on that configuration, in order to handle the incoming workload in a efficient manner. For each private subnet configured in the VPC, a node-group must be created. The code listing 7.5 demonstrates a configuration of a single node group, for a specific subnet in VPC.

```
1  resource "aws_eks_node_group" "gbos-stage-node-group-0" {
2    cluster_name    = local.cluster_name
3    node_group_name = "gbos-stage-node-group-0"
4    node_role_arn   = aws_iam_role.gbos-stage-node-group.arn
5    subnet_ids      = [element(module.vpc.private_subnets, 0)]
6    instance_types  = ["t3a.medium"]
7
8    scaling_config {
9      desired_size  = 3
10     max_size      = 9
11     min_size      = 3
12   }
13   depends_on = [
14     aws_iam_role_policy_attachment.gbos-stage-node-group-AmazonEKSWorkerNodePolicy,
15     aws_iam_role_policy_attachment.gbos-stage-node-group-AmazonEKS_CNI_Policy,
16     aws_iam_role_policy_attachment.gbos-stage-node-group-AmazonEC2ContainerRegistryReadOnly,
17     aws_iam_role_policy_attachment.gbos-stage-node-group-AmazonRoute53FullAccess
18   ]
19 }
```

Listing 7.5: Node groups configuration

7.1.2.3 Identity and Access Management (IAM)

"AWS Identity and Access Management (IAM) enables you to manage access to AWS services and resources securely. Using IAM, you can create and manage AWS users and groups, and use permissions to allow and deny their access to AWS resources" [59]. IAM, in the context of EKS, is responsible for configuring a path for both EKS, and Node Groups, to access AWS services, such as Route 53, Simple Queue System (SQS) or S3. The following code listing 7.6 demonstrates a the roles configuration, as well as the access resource policies.

```
1 resource "aws_iam_role" "gbos-stage-node-group" {
2   name = "eks-node-group-gbos-stage-node-group"
3
4   assume_role_policy = jsonencode({
5     Statement = [{
6       Action    = "sts:AssumeRole"
7       Effect    = "Allow"
```

```
        Principal = {
          Service = "ec2.amazonaws.com"
        }
      }]
      Version = "2012-10-17"
    })
}

"iam_policies" = ["AmazonEKSWorkerNodePolicy", "AmazonEKS_CNI_Policy", "AmazonEC2ContainerRegistryReadOnly",
  "AmazonSQSFullAccess", "AmazonS3FullAccess", "AmazonRoute53FullAccess"]

for policy in var.iam_policies {
  resource "aws_iam_role_policy_attachment" "gbos-stage-node-group-${policy}" {
    policy_arn = "arn:aws:iam::aws:policy/${policy}"
    role       = aws_iam_role.gbos-stage-node-group.name
    depends_on = [aws_iam_role.gbos-stage-node-group]
  }
}
```

Listing 7.6: Amazon Identity and Access Management configuration

7.1.2.4 Elastic Kubernetes Service (EKS)

Given that the VPC, node groups and IAM are setup, the last component for the implementation of the diagram, in figure 7.1, to be fulfilled is the EKS component itself. This configuration module should contain the cluster name and all other attributes to associate it with the aforementioned VPC.

```
module "eks" {
  cluster_name                    = local.cluster_name
  subnets                         = module.vpc.private_subnets
  cluster_endpoint_private_access = true
  cluster_endpoint_public_access  = false
  vpc_id                          = module.vpc.vpc_id
  map_accounts                    = var.map_accounts
  tags = { Environment = "gbos-stage" }
}
```

Listing 7.7: Elastic Kubernetes Service configuration

7.1.3 Databases

Amazon Relational Database Service (RDS) provides scalable and resizable database cloud instances, "while automating time-consuming administration tasks such as hardware provisioning, database setup, patching and backup" [60].

As described previously in section 6, MySQL is used as the database engine, by taking advantage of instances provided by Amazon RDS. Each concession should have three distinct RDS instances, being one for Operational Back Office (OBO) domain, another for Commercial Back Office (CBO) domain and lastly for the global services, with each micro-service having its own database, created on the instance its domain belongs to. The following code listing shows the implemented configuration for a RDS instance, in the OBO domain, for the concession Globalvia Transmontana.

```
resource "aws_db_instance" "a4-devstage-obo" {
  allocated_storage      = 75
  identifier             = "a4-devstage-obo"
  storage_type           = "gp2"
  engine                 = "mysql"
  engine_version         = "5.7.26"
  instance_class         = "db.t2.small"
  parameter_group_name   = "default.mysql5.7"
  db_subnet_group_name   = "rds vpc subnet groups"
  vpc_security_group_ids = ["sg-0f59b34b199ec770f"]
}
```

Listing 7.8: Relational Database Service instance, for concession Transmontana, OBO domain

Additionally, a network peering configuration must be added in order allow incoming requests from the nodes in EKS cluster.

```
resource "aws_vpc_peering_connection" "gbos-stage-obo-rds" {
  peer_vpc_id    = "vpc-0e1c63d93c2ca9a2d"
  vpc_id         = module.vpc.vpc_id
  auto_accept    = true

  accepter { allow_remote_vpc_dns_resolution = true }
  requester { allow_remote_vpc_dns_resolution = true }
  tags = { Name = "VPC peering between RDS and gbos-stage-vpc for OBO" }
}
```

Listing 7.9: Network peering configuration for concession Transmontana

7.1.4 Message Queues

During the architecture analysis and design process, the adoption of asynchronous behaviors was one of the key drivers for the implementation of a micro-services architectural pattern. This allowed the micro-services to be reactive to events, rather than blocking and producing responses instantly for downstream micro-services.

Regarding the implementation of a messages queue system, the adoption of Amazon SQS was an easy decision, grounded to the fact that this project's stack was heavily AWS-driven. SQS offers a fully managed message queuing system, that leverages the management of middle-ware queue systems out of the team's scalability worries. This service offers message exchanging through micro-services in a "maximum throughput, best-effort ordering" [61] manner.

The implementation of SQS resources, followed the IaC principle like previously created AWS resources. The following code listing shows the implemented configuration for a SQS queue for the interaction between Transaction and Validation micro-services.

```
resource "aws_sqs_queue" "transaction-validation-dev" {
  name = "transaction-validation-dev"
  delay_seconds = "20"
  receive_wait_time_seconds = "20"
  visibility_timeout_seconds = "120"
  message_retention_seconds = "1209600"
  redrive_policy = "{\"deadLetterTargetArn\":\"${aws_sqs_queue.transaction-validation-dev-deadletter.arn
    }\",\"maxReceiveCount\":5}"

  tags = { Environment = "dev"}
}
```

Listing 7.10: Message queue configuration

Several types of errors can happen during the exchange of a message between micro-services, such as non-existing queues or even errors in parsing message payloads. However, considering that message queuing communication is asynchronous, the service that dispatched the message is not aware that an error occurs. To mitigate this issue, dead-letter queues were created. A dead-letter queue is a holding queue for messages that are causing errors in the consumers [62], being for parsing or communication errors, these queues will be used to alert that the asynchronous operation was not successful.

Code listing 7.11 demonstrates the creation of a dead-letter queue, as well as its redrive policy.

```
resource "aws_sqs_queue" "transaction-validation-dev-deadletter" {
  name = "transaction-validation-dev-deadletter"
  delay_seconds = "20"
  receive_wait_time_seconds = "20"
  visibility_timeout_seconds = "120"
  message_retention_seconds = "1209600"
}
```

```
resource "aws_sqs_queue_policy" "dev-policy-transaction-validation" {
  queue_url = "${aws_sqs_queue.transaction-validation-dev.id}"
  policy = <<POLICY
{
  "Version": "2012-10-17",
  "Id": "sqspolicy",
  "Statement": [
    {
      "Sid": "First",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "sqs:SendMessage",
      "Resource": "${aws_sqs_queue.transaction-validation-dev-deadletter.arn}",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "${aws_sqs_queue.transaction-validation-dev-deadletter.arn}"
        }
      }
    }
  ]
}
POLICY
}
```

Listing 7.11: Dead-letter message queue configuration

7.1.5 Storage Components

During the design realization process of the use case view on section 6.6.1, it was identified that a highly available and reliable data storage facility was required, in order to store all photos generated by the Optical Character Recognition (OCR) component deployed in the Road Side Equipment (RSE) component. In order to achieve this requirement, Amazon S3 was adopted.

Amazon S3 is an object storage service that provides high availability, scalability and performance, for storing objects, files or data [63]. Each storing unit is referred as a `bucket`, as the data stored in the service is object-based, meaning that each file will have an unique identifier. S3 also offers security policies out-of-the-box, by allowing the definition of access control lists during its creation.

Similarly to the implementation of SQS resources, S3 follows the IaC principle by declaring the resources to create. Code listing 7.12 demonstrates the implemented configuration for a S3 storage component for the photo file storing functionality.

```
resource "aws_s3_bucket" "photo-create-dev" {
  bucket = "photo-create-dev"
  acl    = "private"
  tags = {
    Name        = "photo-create-dev"
    Environment = "dev"
  }
}
```

Listing 7.12: S3 bucket configuration

7.2 Micro-services

Regarding the development of micro-services identified in section 6.2, a set of good practices and patterns were defined as the base for each micro-service, which lay on the following:

- Follow the packaging structure presented in section 6.4;
- Implement database per service pattern;

- Implement event messaging and provide HyperText Transfer Protocol (HTTP) Representational State Transfer (REST) interfaces for information querying;
- Keep configuration at a minimum, leaving the responsibility of managing environment-specific configurations for a specialized components;
- Design for failure;
- Include a message broker in its runtime.

When developing micro-services from scratch in a fast-paced environment, establishing early structural consistency is key. Although micro-services remove the need for sticking with a single language for the development of the system, it is advised against it in early stages of development, as it can bring unforeseen problems in a unnecessary manner. Additionally, using a single language for several services, in the beginning, can save time and money by re-using the same infrastructure processes for deployment.

Regarding programming languages and frameworks, Java Spring[7] was chosen due to its mature and future-rich framework. This framework provides tools for authentication, logging, data access and even specific modules for message brokers. Spring often suffers from extensive configuration processes, which can be a great drawback for micro-services development, however, a specialized module, called Spring Boot[8], is also offered by the framework. Spring Boot was created to make it easy to create stand-alone applications, with minimal configuration. Its main purpose is to make it easy to build and run micro-services, while taking advantage of the extensive Spring framework features.

In the following sections, implementation details of micro-service's inherent components is analysed and demonstration.

7.2.1 Configuration

A software configuration tends to include the selection of software dependencies and setting parameters of applications and underlying infrastructure components. Software configuration parameters provide context for the software package, as well as offering the possibility of adapting components based on functionality.

During the development of micro-services, it was deemed that configuration management was ruled as crucial for the success of the micro-services deployment. Given the fact that each concession needed its own database, and the need for the application to work under several environments, such as production and staging. To tackle this issue, a component specialized in configuration management was added to the design of the solution.

Spring Cloud Config[9] provides externalized configuration for distributed software systems. This supplies a central place to store configuration parameters across all environments, with the setup of a HTTP resource-based API, services query for the configurations at boot-time, based on the name of the application and the desired environment. In a sense, this component can be seen as a micro-service that serves other micro-services. Similarly to other micro-services, the configuration server also needs to be deployed in the EKS cluster, as demonstrated further in section 7.2.4.

[7] https://spring.io/
[8] https://spring.io/projects/spring-boot
[9] https://cloud.spring.io/spring-cloud-config/

The implementation of Spring Cloud Config in the server-side, follows the premise of Spring Boot, whereas it is created with minimal configuration. A micro-service is created, with configurations for a Git backend. The following code listing 7.13 demonstrates the configurations needed.

```
spring.cloud.config.server:
    health.enabled: false
    git:
      uri: gitlab.gbosplatform.com/globalvia-mindera/config-repo.git
      skipSslValidation: true
      timeout: 4
      username: ${GIT_USERNAME}
      password: ${GIT_PASSWORD}
      force-pull: true
```

Listing 7.13: Spring Cloud Config micro-service configuration

The configuration is dependent of environment variables (GIT_USERNAME and GIT_PASSWORD), that set during service provisioning. Based on this configuration, the micro-service will provide an API to fetch configurations, that will be retrieved from the defined Git. The repository should contain yml files with the configurations for each service, defined by the application name and the environment. The activity diagram, in figure 7.2, showcases a micro-service, during boot phase, requesting a configuration for a environment to the server.

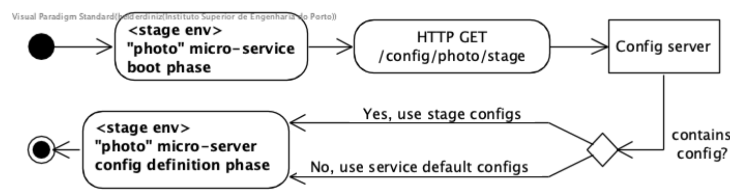

Figure 7.2: Activity diagram of micro-services and config-server interactions

7.2.2 Event Messages

A successful implementation of a scalable message queuing communication process, is usually based on tight message schema that are fulfilled by message publishers, and parsed by message consumers. When pairing message queuing processes with Java systems, the path of following strongly-typed principles for message schema is a natural decision, this caused the need to define a way to achieve those principles in a easy way, and thus the usage of Google Protocol Buffers was adopted.

Google Protocol Buffers are a simple and effective approach of serializing structured data with tight schema rules, as well as providing code generation tools for several languages [64]. Each message is defined in a proto-neutral fashion, that provides the typical primitive types, and even custom fields for complex messages. Additionally, Protocol Buffers offers tools to generate source code based on the defined messages.

In code listing 7.14, a representation of a Protocol Buffer message that is used on the scheduled job registration on Scheduler micro-service, as demonstrated by the use case in section 6.6.3.

```
syntax = "proto3";

import "google/protobuf/any.proto";

option java_package = "com.globalvia.protospec";

message SchedulingRegistration {
    string service_name = 1;
```

```
9       string event_name = 2;
10      string queue_name = 3;
11      string payload_string = 4;
12      string cron_expression = 5;
13      bool is_one_off = 6;
14      bool should_retry = 7;
15      RetryConfiguration retry_configuration = 8;
16      google.protobuf.Any payload_proto = 9;
17      string time_zone = 10;
18      string tenant_id = 11;
19
20      message RetryConfiguration {
21          int64 initial_backoff_seconds = 1;
22          float backoff_multiplier = 2;
23          int64 max_backoff_seconds = 3;
24          int32 max_retries = 4;
25      }
26  }
```

Listing 7.14: Protocol Buffer message for Scheduler registration

Having the message defined successfully, it must be included in the micro-services that need it, regardless of being a publisher or a consumer.

7.2.3 Message Broker

Another key component for a successful implementation of a scalable message queuing communication, is choosing a messaging API that follows the same principles as the architectural style, scalable and highly available. Considering the project's technological stack, being in Java Spring, Java Message Service (JMS) is the choice as the messaging API.

JMS is a Java API aimed at offering a middleware layer, for message-oriented communication. This API should be used by the message broker, in order to parse and transform code into readable event messages for the system.

Regarding the implementation of the message broker, it is implemented by taking advantage of two distinct interfaces: Spring JMS and Amazon SQS Java Messaging Library. Spring JMS provide easy configuration for JMS messaging, as well as providing all the necessary dependencies for Spring to be able to use it with minimal configuration. Amazon SQS Java Messaging Library is an implementation of JMS specifically for SQS, which facilitates the implementation of message brokers in the application. In order for the micro-services to start consuming messages provided by JMS and SQS, two configuration classes must be implemented: `ContainerFactory`, and consumer class for the given message.

`ContainerFactory` is a configuration class that serves all the necessary settings for JMS to function. It works as a bridge between SQS and the micro-service's consumers and must be configured based on the AWS region and set the AWS account credentials, in order to obtain access to SQS resources.

The message consumer class is where the message is going to be placed, after JMS has deemed it as a valid message. This class is responsible to declare the type of message it consumes, setup the queue it subscribes to, as well as any concurrency it might apply when consuming. Code listing 7.15 shows an implementation of a consumer for the scheduled job registration on Scheduler micro-service, as demonstrated by the use case in section 6.6.3.

```
1   public class RegistrationConsumer extends EventHandler<SchedulingRegistration> {
2       @Override
3       @JmsListener(destination = "${globalvia.queues.registration.name}", concurrency = "${globalvia.queues
        .registration.concurrency:1}")
4       public void listener(final Message message) {
5           super.listener(message);
6       }
7
8
```

```
9      @Override
10     public void process(final SchedulingRegistration schedulingRegistration) {
11         registrationService.register(schedulingRegistration);
12     }
13 }
```

Listing 7.15: Scheduler registration consumer class

7.2.4 Provisioning

Provisioning is the process of setting up a software component, or a server, in a specific network, based on a set of required resources and artifacts. In the context of this dissertation, the provisioning process is carried out by the cloud provider, where it "instantiates a container with each micro-service and all its software dependencies, and orchestrates how the set of containers are assigned in the underlying infrastructure" [65].

Each micro-service is isolated in its own its own environment, therefore each component will be executed in different containers. In order to achieve this, Docker images were created for each micro-service, based on a popular debian OpenJDK distribution. Code listing 7.16 demonstrates an implementation of a `Dockerfile` for the Validation micro-service.

```
1  FROM openjdk:11.0.2-jdk-slim
2
3  ENV APPLICATION_NAME validation
4
5  ADD provision/run.sh run.sh
6  RUN mkdir -p /opt/$APPLICATION_NAME
7
8  COPY ./target/*.jar /opt/$APPLICATION_NAME/$APPLICATION_NAME.jar
9
10 EXPOSE 8080
11 ENTRYPOINT sh ./run.sh
```

Listing 7.16: Dockerfile for Validation micro-service

Having the environment set for each micro-service, the following step is to define the Kubernetes definition of the resources required by the micro-service, which is achieved by the usage of Helm[10]. This tool is aimed at facilitating the creation, and management, of Kubernetes resources, by providing `charts` that will create these resources, based on parameters and variables defined in the chart manifest. The base chart contains the necessary values to the create the micro-services resources, specifically the deployment, service and ingress.

7.2.4.1 Deployment

A Deployment resource is an object which provides declarative updates to a specific application. This is the controller of the micro-service's rolling update actions, as well as rollback actions. This resource allows the definition of the application's lifecycle, such as images to use for the pod, number of pods, how it should execute the update operation and to define custom functions to determine the pod's health. Code listing 7.17 demonstrates a snippet containing the deployment resource description for Validation micro-service. Each micro-service should contain a name, environment data, container image tag and even liveness and readiness probation parameters.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata.name: {{ template "validation.fullname" . }}
4 spec:
```

[10] `https://helm.sh/`

```
5     replicas: {{ .Values.replicaCount }}
6     template:
7       spec:
8         containers:
9         - env:
10          - name: SPRING_PROFILES_ACTIVE
11            value: {{ .Values.springProfilesActive }}
12          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
13          imagePullPolicy: {{ .Values.image.pullPolicy }}
14          ports:
15          - name: http
16            containerPort: {{ .Values.containerPort }}
17          livenessProbe:
18            httpGet.path: {{ .Values.health.path }}
19            initialDelaySeconds: {{ .Values.health.livenessProbe.initialDelaySeconds }}
20            failureThreshold: {{ .Values.health.livenessProbe.failureThreshold }}
21            periodSeconds: {{ .Values.health.livenessProbe.periodSeconds }}
22          readinessProbe:
23            httpGet.path: {{ .Values.health.path }}
24            initialDelaySeconds: {{ .Values.health.readinessProbe.initialDelaySeconds }}
25            failureThreshold: {{ .Values.health.readinessProbe.failureThreshold }}
26            periodSeconds: {{ .Values.health.readinessProbe.periodSeconds }}
```

Listing 7.17: Kubernetes deployment snippet for Validation micro-service

The `livenessProbe` and `readinessProbe` values are used by Kubernetes to determine the pod's health. The `liveness` value is used to know when a cotainer should be restarted, while the `readiness` value is used to indicate when a container is ready to start receiving incoming traffic.

7.2.4.2 Service

A Kubernetes service is an abstraction resource, that defines a policy on a logical set of pods, in order to provide a communication interface to access them. The set of pods is then accessible by the service resource, which is determined by a `selector` tag. Services is the resource that helps the Kubernetes service discovery process to find interfaces for services, which is specially useful because "With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them" [66].

The following code listing 7.18, demonstrates a snippet containing the service resource description for Validation micro-service. Each micro-service should contain a selector and the interface port it should expose.

```
1  apiVersion: v1
2  kind: Service
3  metadata.annotations.external-dns.alpha.kubernetes.io/hostname: {{ .Values.dns }}
4  spec:
5    type: {{ .Values.service.type }}
6    ports:
7    - port: {{ .Values.service.port }}
8      targetPort: http
9      protocol: TCP
10   selector:
11     app: {{ template "validation.name" . }}
12     release: {{ .Release.Name }}
```

Listing 7.18: Kubernetes service snippet for Validation micro-service

7.2.4.3 Ingress

Ingress is a Kubernetes definition for an object that manages external access to services inside the cluster, using interfaces such as HTTP or WebSocket. It may be configured to declare service's external URLs, mediate Secure Sockets Layer (SSL) / Transport Layer Security

(TLS) or offer named resources. Additionally, a resource that contains an Ingress controller is responsible for fulfilling the Ingress definition with a load balancer. Code listing 7.19, demonstrates a snippet containing the ingress resource description for Validation micro-service. If a micro-service offers a HTTP interface, it should contain an backend configured for its exposed port, as well as registering its service name to that port.

```
{{- if .Values.ingress.enabled -}}
{{- $fullName := include "validation.fullname" . -}}
{{- $ingressPath := .Values.ingress.path -}}
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: {{ $fullName }}
  labels.app: {{ template "validation.name" . }}
{{- with .Values.ingress.annotations }}
  annotations:
{{ toYaml . | indent 4 }}
{{- end }}
spec:
  rules:
    - host: {{ .Values.dns }}
      http:
        paths:
          - backend:
              serviceName: {{ $fullName }}
              servicePort: http
{{- end }}
```

Listing 7.19: Kubernetes ingress snippet for Validation micro-service

7.2.5 Pipeline

Regarding the implementation of the Continuous Integration (CI) & Continuous Delivery (CD) process, demonstrated at section 6.7, the build tool used to implement the process is Jenkins. This build tool is an open source builder server, that can be used to automate building, testing and even software deployment. Jenkins offers easy configuration, extensible plugins and a domain-specific language based on Groovy.

A multi-stage `Jenkinsfile` was developed, based on the stages demonstrated at figure 6.14, in order to deploy rolling upgrades to new builds into the Amazon EKS cluster. The implemented stages were: build, testing and deployment.

7.2.5.1 Build and Testing

In order to run building and testing tasks, the dependency manager Maven[11] was used. The build stage is focused on compiling the application, and its work is merged into the testing stage because when Maven runs `package` goals, it also executes all tests present in the micro-service. Additionally, the generation of code quality reports and code coverage analysis is executed by SonarQube, both in the same stage. The following code listing 7.20 demonstrates a snippet of the tasks executed by Jenkins during the build and testing phase.

```
git branch: '${gitlabTargetBranch}', credentialsId: '55d5a7ef-35a6-4e38-8aff-354699c89566', url:
    repository
commitHash = firstNChars(7, "${gitlabMergeRequestLastCommit}")
slackSend channel: "${params.slackChannel}", color: "grey", message: "Build Started - ${env.JOB_NAME} ${
    commitHash} (<${env.BUILD_URL}|Open>)"

sh 'mvn -s $MAVEN_SETTINGS clean package'
sh 'mvn -s $MAVEN_SETTINGS sonar:sonar'
```

Listing 7.20: Jenkinsfile - build & testing stage

[11] `http://maven.apache.org/`

7.2.5.2 Deployment

The deployment stage is responsible for building the container to deploy, storing it and then lead the deployment into the cluster. This is a complex task due to interaction with several components, so it should evaluated in two sub-steps: preparation and update.

During the preparation step, Jenkins will use Docker to create an image of the built micro-service, based on the `Dockerfile` included in the build work path. Upon building the image, it is registered in a Docker Registry provided by Amazon Elastic Container Registry (ECR) with the latest commit hash.

```
1 def dockerLoginData = sh(script: "AWS_DEFAULT_REGION=eu-west-1 aws ecr get-login | sed 's/-e//g' | sed 's
      /none//g'", returnStdout: true).trim()
2 def dockerRegistry = sh(script: "AWS_DEFAULT_REGION=eu-west-1 aws ecr get-login | sed 's/-e//g' | sed 's/
      none//g' | awk '{print \$NF}' | sed 's+https://++g'", returnStdout: true).trim()
3
4 sh "docker build -t " + name + " ${params.chartPath}"
5 sh dockerLoginData
6 sh "docker tag ${name}:latest ${dockerRegistry}/${name}:${commitHash}"
7 sh "docker tag ${name}:latest ${dockerRegistry}/${name}:latest"
8 def dockerImage = "${dockerRegistry}/${name}"
9 sh "docker logout ${dockerRegistry}"
```

Listing 7.21: Jenkinsfile - deployment stage - preparation

Having the Docker image set on the registry, Helm should be executed in order to dispatch an update to Kubernetes API to perform a rolling update. Before dispatching the event, Helm is configured with deployment parameters. Firstly, a timeout value is set, which is based on the amount of running pods, for that given moment, for the micro-service to deploy. It is followed by a linting operation, in order to validate the configurations and the Helm file present in the micro-service. Finally, a rolling update event is executed based on the deployment parameters. If the micro-service is not healthy after the timeout value, or it reports errors on its boot phase, Helm should perform a rollback operation.

```
1  stage("${environment}") {
2      def replicasCount = sh(script: "kubectl get deployment ${name}-${environment} -n${environment} --
       output json | grep 'replicas' -m1 | cut -d':' -f2 | cut -d ',' -f1", returnStdout: true).trim().
       toInteger()
3      def timeout = replicasCount * 300 * timeoutFactor // 5 min for each replica
4
5      sh "helm lint ${chartPath}/provision/${name}"
6      try {
7          sh "helm upgrade ${name}-${environment} -i" +
8                  " -f ${chartPath}/provision/${name}/values-${environment}.yaml" +
9                  " --force" +
10                 " --set image.repository=${image}" +
11                 " --set-string image.tag=${tag}" +
12                 " --namespace ${environment}" +
13                 " --timeout=${timeout}" +
14                 " --wait" +
15                 " --debug ${chartPath}/provision/${name}"
16     } catch (err) {
17         echo "Error deploying to ${environment}, deployment is not healthy after ${timeout} seconds.
       Rolling back to previous version."
18         def revisionToRollback = sh(script: "helm history ${name}-${environment} | grep -v \"Rollback\" |
       grep DEPLOYED | awk '{print \$1}'", returnStdout: true)
19         sh "helm rollback ${name}-${environment} ${revisionToRollback}"
20     }
21 }
```

Listing 7.22: Jenkinsfile - deployment stage - update

7.2.6 Multi-tenancy

Introduced in section 4.3, the multi-tenant requirement is the main driver of this dissertation. Providing all micro-services for any tenant, using the same code-base and same infrastructure was the goal, and the implementation should not be conflicting with the database per service pattern, having the micro-services follow the single configurable instance pattern.

This means that each micro-service will be configurable to work with any tenant, having a database per tenant and per micro-service.

A Spring Boot library was created with the goal of providing multi-tenancy for a micro-service with little to none refactoring, by just adding application configuration properties for each concession wanted. Following the library approach allowed for a greater code organisation, by centralizing all multi-tenancy code in a single module, the micro-services were only required to add tenant-specific configurations, and an annotation to enable multi-tenancy functionality.

7.2.6.1 Configuration

The configuration process is the first step for the micro-service to become multi-tenant, the process is started by explicitly adding the concessions that the micro-service is interested in working with, by implementing the structured configuration in the service's application properties. The required fields are `identifier`, `url`, `username` and `password`. Field `identifier` is the representational Universally Unique Identifier (UUID) of the concession and it is used in incoming and outgoing communications, as authentication. Code listing 7.23 demonstrates a sample configuration for two concessions.

```
multitenancy.tenants:
  - name: "acega_ap50"
    identifier: "56e95601-b181-40a7-9ad3-94aa782c59fb"
    url: "jdbc:mysql://acega:3306/micro_service"
    username: "username"
    password: "password"
  - name: "transmontana_a4"
    identifier: "7236ea7e-1c14-4154-be52-d80c85c10507"
    url: "jdbc:mysql://transmontana_a4:3306/micro_service"
    username: "username"
    password: "password"
```

Listing 7.23: Multi-tenancy configurations

Having the configurations successfully defined, the last step is to enable the library, by using the `@EnableMultiTenancy` annotation on an application configuration class, and the configurations can be accessed by accessing `MultitenancyConfigurationProperties` class.

7.2.6.2 Tenant Context

Regardless of the fact that micro-services are designed with tenant-agnostic business logic, there is a need to be aware of the tenant that owns the information it is handling. This is specially important when considering data storing concerns, as storing data in the wrong tenant can lead to complex problems. With this in mind, the concept of tenant context was created.

The tenant context message is a representation of the concession's identifier, and optionally, the user that is carrying out the operation. The context is stored in `ThreadLocal` short-term memory, and it is set by interceptors on incoming HTTP or message queuing communication. Each thread has an indication of which tenant owns the information, and it can be accessed at any time, by a component present on that thread. Code listing 7.24 contains a snippet of the class that provides tenant context throughout the thread's operation.

```
public class TenantContextHolder implements ApplicationContextAware {
    private static final ThreadLocal<ContextMessage> currentContext = new ThreadLocal<>();
    private static MultitenancyConfigurationProperties configProperties;

    private TenantContextHolder() { }

    public static ContextMessage getCurrentContext() {
```

```
8            return currentContext.get();
9        }
10
11       public static ContextMessage getAndFlushCurrentTenant() {
12           final ContextMessage contextMessage = getCurrentContext();
13           clearContext();
14           return contextMessage;
15       }
16
17       public static void setCurrentTenant(final String tenant) {
18           setCurrentContext(ContextMessage.Factory.getContext(tenant));
19       }
20
21       public static void setCurrentContext(final ContextMessage context) {
22           if (currentContext.get() != null)
23               throw new IllegalArgumentException("ContextMessage should be null before assigned. It was: "
         + currentContext.get());
24
25           final boolean isInvalidValidTenant = configProperties.getTenants().stream()
26                   .filter(t -> t.getIdentifier().equals(context.getTenantId()))
27                   .findFirst()
28                   .isEmpty();
29
30           if (isInvalidValidTenant)
31               throw new InvalidTenantException("Invalid X-Tenant-Id: " + context.getTenantId());
32
33           currentContext.set(context);
34       }
35
36       public static void clearContext() {
37           currentContext.remove();
38       }
39   }
```

Listing 7.24: Tenant context holder class

`TenantContextHolder` also provides an entry point to set the tenant context, on method `setCurrentContext(context)`, while validating if it exists on the micro-service's configuration properties.

Regarding multi-threading and asynchronous operations, each thread should set the tenant context on the moment it starts executing code, and clear it when it finishes. The context is propagated between threads in asynchronous operations, whereas the thread will explicitly indicate that its local context should be set on the future thread, when it starts its work.

7.2.6.3 Communication

Multi-tenancy requirements should also be considered for incoming communication, as there is a need to supply a method of indicating the concession that should be considered for the request. Two communication methods are supported by the implemented micro-services, HTTP and message queuing. Each communication method has its own implementation, therefore specifically interceptors made for each type.

HTTP

In order to provide multi-tenancy functionality for HTTP communication, each micro-service is required to add a HTTP header named `X-Tenant-Id`, that includes the UUID field of the concession. It is then validated by an interceptor, registered in Spring's HTTP security layer. The following code listing 7.25 demonstrates the HTTP interceptor, responsible for capturing the `X-Tenant-Id` and requesting the context holder class to set it.

```
1  public class MultitenancyInterceptor extends HandlerInterceptorAdapter {
2      public static final String TENANT_HEADER_KEY = "X-Tenant-Id";
3
4      @Override
5      public boolean preHandle(HttpServletRequest req, HttpServletResponse res, Object handler) {
6          final String tenantIdentifier = Optional.ofNullable(req.getHeader(TENANT_HEADER_KEY))
7                  .orElseThrow(() -> new InvalidTenantException(String.format("%s is required",
       TENANT_HEADER_KEY)));
8          req.setAttribute(CustomRequestAttributes.CURRENT_TENANT_IDENTIFIER, tenantIdentifier);
9          TenantContextHolder.setCurrentContext(ContextMessage.Factory.getContext(req.getHeader(
       TENANT_HEADER_KEY), req.getHeader(USER_HEADER_KEY)));
```

```
10            return true;
11        }
12
13        @Override
14        public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
           ModelAndView modelAndView) {
15            TenantContextHolder.clearContext();
16            super.postHandle(request, response, handler, modelAndView);
17        }
18
19        @Override
20        public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
           Exception ex) {
21            TenantContextHolder.clearContext();
22            super.afterCompletion(request, response, handler, ex);
23        }
24
25        @Override
26        public void afterConcurrentHandlingStarted(HttpServletRequest request, HttpServletResponse response,
           Object handler) {
27            TenantContextHolder.clearContext();
28        }
29    }
```

Listing 7.25: HTTP multi-tenant interceptor

Message Queuing

Similarly to the HTTP communication method, the component must provide the concession identifier to which the message belongs to. However, considering that message queuing does not include headers, the identifier must be present on a field with the name `tenant_id`. On a micro-service with multi-tenancy enabled, an interceptor is registered to validate each incoming message. The following code listing 7.26 demonstrates the message queuing interceptor, responsible for capturing the `tenant_id` field and requesting the context holder class to set it.

```
1   public abstract class MultitenancyEventHandler<T extends GeneratedMessageV3> extends EventHandler<T> {
2       private static final String TENANT_ID_PROTO_FIELD_NAME = "tenant_id";
3
4       @Override
5       protected void preHandle(final GeneratedMessageV3 message) {
6           final Descriptors.FieldDescriptor fieldDescriptor = message.getDescriptorForType()
7                   .findFieldByName(TENANT_ID_PROTO_FIELD_NAME);
8
9           if (isNull(fieldDescriptor))
10              throw new InvalidTenantException("Protobuf message must contain field tenant_id");
11
12          final String tenantId = (String) message.getField(fieldDescriptor);
13
14          if (isBlank(tenantId))
15              throw new InvalidTenantException("Event message requires tenant_id");
16
17          TenantContextHolder.setCurrentTenant(tenantId);
18      }
19
20      @Override
21      protected void onComplete() {
22          TenantContextHolder.clearContext();
23      }
24  }
```

Listing 7.26: HTTP multi-tenant interceptor

7.2.6.4 Database Selection

The last piece of functionality offered by the multi-tenancy library, is the possibility of changing data sources in runtime. Taking advantage of Spring Boot JPA, the library registers a specific connection provider object, that is responsible for indicating which data source should be used for the tenant present in the current context. This will cause JPA to verify which data source it should select, before executing any database-related work, ensuring that the proper data source is selected for the tenant in hand. JPA multi-tenancy is achieved by the implementation of three components.

MultitenancyDatabaseConfiguration is the first component to implement, this class is responsible for setting up the configuration for runtime data source selection, by evaluating the application properties added in the configuration step, as described on step on section 7.2.6.1. Code listing 7.27 demonstrates the implementation for this component.

```
@Configuration
public class MultitenancyDatabaseConfiguration {
    private final MultitenancyConfigurationProperties multitenancyProperties;

    @Bean(name = "multitenantProvider")
    public DataSourceBasedMultiTenantConnectionProviderImpl dataSourceBasedMultiTenantConnectionProvider
    () {
        final Map<String, DataSource> dataSources = multitenancyProperties.getTenants()
                .stream()
                .collect(Collectors.toMap(MultitenancyConfigurationProperties.Tenant::getIdentifier, this
    ::buildDataSource));

        return new DataSourceBasedMultiTenantConnectionProviderImpl(dataSources);
    }
}
```

Listing 7.27: MultitenancyDatabaseConfiguration implementation

CurrentTenantIdentifierResolverImpl is the second component on the list, this class is responsible for indicating to JPA, where to retrieve the tenant identifier value, as its implementation shows on code listing 7.28.

```
public class CurrentTenantIdentifierResolverImpl implements CurrentTenantIdentifierResolver {
    @Override
    public String resolveCurrentTenantIdentifier() {
        return TenantContextHolder.getCurrentContext().getTenantId();
    }
}
```

Listing 7.28: CurrentTenantIdentifierResolverImpl implementation

Finally, DataSourceBasedMultiTenantConnectionProviderImpl is the last component to implement. This class is responsible for providing the data source for JPA to use, when executing queries. This class is initially created in the MultitenancyDatabaseConfiguration, but it is then used by JPA on the moment it decides to select the data source, which should be invoked after the second implemented component. Code listing 7.29 demonstrates the implementation of this component.

```
public class DataSourceBasedMultiTenantConnectionProviderImpl extends
    AbstractDataSourceBasedMultiTenantConnectionProviderImpl {
    private final transient Map<String, DataSource> dataSourceMap;

    public DataSourceBasedMultiTenantConnectionProviderImpl(final Map<String, DataSource> dataSourceMap)
    {
        this.dataSourceMap = dataSourceMap;
    }

    @Override
    protected DataSource selectDataSource(final String tenantIdentifier) {
        return dataSourceMap.get(tenantIdentifier);
    }
}
```

Listing 7.29: DataSourceMultiTenantConnectionProvider implementation

7.3 API Gateway

API gateway provides an entry point for external services to communicate with a specific micro-service in the architecture. The adoption of this pattern is aimed at facilitating external access into the micro-services of the architecture, as this allows to only expose the necessary endpoints, as well as removing the need of the user to know where to fetch for a

specific kind of information, most specifically in this context, the RSE and API composition components.

The implementation process of API gateway pattern is grounded on the usage of Ambassador[12] API Gateway, which "is an open-source Kubernetes-native API Gateway designed to easily expose, secure, and manage traffic to your Kubernetes microservices" [67]. Additionally, an API gateway is usually dependent on a service discovery pattern, however, since Kubernetes implement it in an explicit way, this facilitates the adoption of Ambassador.

The registration of endpoints or services in Ambassador is achieved via Kubernetes annotations. The annotations are appended to the service resource (described in section 7.2.4.2), and are updated on each deployment. Code listing 7.30 demonstrates the registration of the base HTTP interface for Validation micro-service.

```
annotations.getambassador.io/config: |
    ---
    apiVersion: getambassador.io/v2
    kind: Mapping
    name: validation_http_endpoint
    prefix: /validation/v1/
    rewrite: /v1/
    service: {{ template "validation.fullname" . }}
    ambassador_id: {{ .Values.ambassador_private_id }}
```

Listing 7.30: Endpoint registration for Validation micro-service

With the annotation registration in place, Kubernetes will automatically register this endpoint on Ambassador, when a new deployment is successfully delivered into the cluster. The registered endpoint is then available to be used, as well as displayed on Ambassador dashboard, as demonstrated on figure 7.3.

Ambassador Route Table

URL	Service	Weight
[internal route] http://prelive-api.gbosplatform.com/.ambassador/ precedence 1000000	127.0.0.1:8500	100.0%
[internal route] http://prelive-api.gbosplatform.com/edge_stack/ precedence 1000000	127.0.0.1:8500	100.0%
http://prelive-api.gbosplatform.com/validation/v1	validation-prelive	100.0%
http://prelive-api.gbosplatform.com/transaction/v1/transactionInfos	transaction-prelive	100.0%
http://prelive-api.gbosplatform.com/transaction/v1/transactionInfos	transaction-prelive	100.0%
http://prelive-api.gbosplatform.com/ambassador/v0/check_ready	127.0.0.1:8877	100.0%
http://prelive-api.gbosplatform.com/ambassador/v0/check_alive	127.0.0.1:8877	100.0%

Figure 7.3: Ambassador registered endpoints table

7.4 API Composition

Adopting a micro-services architectural style tends to bring numerous advantages for complex systems, however, as these systems segregate and grow, querying for information is not as straightforward as wanted. In the context of this dissertation, providing a solution for easy querying information was considered a priority, considering that each domain is heavily segregated between micro-services, however, when operators intend to query information for transactions, concessions or even clients, the data is aggregated in a single result. The API composition pattern provides a simple solution for this problem.

[12] https://www.getambassador.io/

GraphQL[13] is an API query language, that allows the definition of query and mutations, in order to provide data in a domain-driven way, in a "ask for what you need, get exactly that" [68] manner. This allows clients to simplify their requests, as well as compose several requests, if the GraphQL schema allows it.

7.4.1 Schema

GraphQL schema is a typed dataset that is made available via a GraphQL interface, in order to be used in queries or mutations. The schema types are usually represented in edge-labeled graphs [69], whereas the nodes represent a type, and each node contains properties, representing the type's attributes, fulfilling the ultimate objective of the schema, that is introducing a notion of types for objects.

Schema also introduce the concept of scalars and objects. A scalar is the final representation of a given attribute, as its value is represented by the scalar definition. A scalar can take form of strings, integers or even dates. In the same way as scalars, objects are also used by the schema graph, however, they're considered gates to scalars. Objects are structured data object, such as types, to represent typed interfaces that lead to final representations.

The GraphQL schema definition follows a Schema Definition Language (SDL), called GraphQL schema language, in order to describe type, queries or mutation. Having an updated schema definition is essential, because the clients are based on that schema in order to know how to communicate with the server.

Code listing 7.31 demonstrates the usage of schema definition in order to create the type `Transaction`.

```
1  type Transaction {
2      globalId: String
3      transactionId: String
4      transactionDate: String
5      originId: Int
6      originName: String
7      status: String
8      gantryId: Int
9      gantryName: String
10     laneId: Int
11     laneType: String
12     passageMode: String
13     vehicle: Vehicle
14     photos: [Photo]
15     validationHistoryList: [ValidationHistory]
16     payments: [Payment]
17 }
```

Listing 7.31: Transaction type schema representations

The `Transaction` type is composed by several attributes, most of them value scalars, but does also contain objects and lists. For instance, the `photos` attribute is a list of type `Photo`, which is another defined type in GraphQL schema language.

7.4.2 Query

A GraphQL query is a root type, and it represents a reading operation that allows clients to fetch data, with the possibility of selecting the fields it is interested in evaluating. A query definition is set by an operation name, query parameters and the response object it yields. The query name follows no particular convention, however it is the identification of

[13] `https://graphql.org/`

the operation for the clients. Query parameters are named and ordering is not important, as GraphQL query validator will automatically match the name with the query definition.

Code listing 7.32 demonstrates the definition of query operations, for the `Transaction` type.

```
# TransactionQuery #
type Query {
    transaction(id: String!): Transaction
    transactions(filter: TransactionFilter!): [Transaction]
}
```

Listing 7.32: Query type representation for Transaction requests

With the creation of the query type demonstrated in code listing 7.32, GraphQL will be able to evaluate operations that return the aforementioned type. For instance, the first operation named `transaction`, is expecting a query parameter of type String, which should yield a result of type `Transaction`, and it can be invoked by clients. The following picture 7.4 demonstrates an example of a GraphQL client query invocation and its result, using Playground[14].

```
query transaction($id: String!) {
  transaction(id: $id) {
    globalId
    transactionId
    originName
    transactionDate
  }
}
```

QUERY VARIABLES HTTP HEADERS (1)

```
{
  "id": "8dfd8451-cb7e-4b86-96db-0851534440ef"
}
```

```
{
  "data": {
    "transaction": {
      "globalId": "8dfd8451-cb7e-4b86-96db-0851534440ef",
      "transactionId": "540727",
      "originName": "Santiago",
      "transactionDate": "2020-10-11T16:13:41Z"
    }
  }
}
```

Figure 7.4: GraphQL query client invocation and result

7.4.3 Mutation

Similarly to the query definition, a GraphQL mutation is a root type, that represents a write, or update, operation that allows clients to change (mutate) data in the micro-services. This type also offers the possibility of selecting certain type attributes of the response, as long as the mutation yields any result. A mutation definition is set by an operation name, query parameters and an optional response object it yields. This is very similar to the query operation, however, the focus part is on the query parameters, whereas the input type should contain the logic for mutating data, such as a Data Transfer Object (DTO) for a specific micro-service.

Code listing 7.33 demonstrates the definition of a mutation operation, that allows editing a status of the `Transaction` type.

```
# TransactionMutation #
type Mutation {
    editTransaction(globalId: String!, status: TransactionStatus!): Transaction
}
```

Listing 7.33: Mutation type representation for Transaction

[14] https://github.com/graphql/graphql-playground

This mutation operation is expecting two query parameters, one that indicate the transaction's id, and another that indicates its status, which is an enumeration object of type `TransactionStatus`. The following picture 7.5 demonstrates an example of a GraphQL client mutation invocation and its result.

Figure 7.5: GraphQL mutation client invocation and result

Chapter 8

Evaluation

Succeeding the implementation of any product, solution or project, it is necessary to review if the final product meets the initial expectations, reaches the goals it planned, and if it does solve the problem in hand. It is equally important to be constantly evaluating the product, throughout its development, when possible. This allows for early validation, however, there are evaluations that are only made possible when a specific part of the development has terminated.

In this chapter, the system is evaluated in several aspects of measurements and dimensions, by inducing experiments and tests, while taking into account the developed requirements. Additionally, the results of these experiments are presented and analysed.

8.1 Experimentation and Testing

Experiments, or tests, are responsible for evaluating one or more measures, using a specific methodology in order to validate or deny a given hypothesis. The first step to define an experiment is to define their measurements, followed by the hypotheses definition and finally, the evaluation methodology.

8.1.1 Measurements

In order to induce an evaluation process, some measures have to be defined to support the evaluation of the solution described in this dissertation. These measurements allow the definition of the concepts to evaluate, in an analytical manner. Based on problems and requirements of the solution aforementioned, to validate the proposed solution, the following measures were defined:

- **Technical quality**: implementation quality must be assured, in order to discard implementation errors from the result analysis;
- **Performance**: aimed at evaluating the component performance, by measuring the API response time, and time to process messages present in queue;
- **Scalability**: aimed at evaluating if the system is able to scale and handle an increase of work-load.

These measures will support the evaluation process, by being assessed during the executed experiments, in order to identify if the goals were reached. To do so, defining the hypotheses is required.

8.1.2 Hypotheses

An hypothesis is a statement that can be made, by the exploration of a specific set of data, used by statistical tests in order to verify its accuracy. The following hypothesis were identified:

- **H1**: Complete success in unit and integration tests: if verified, this hypothesis ensures that the quality of the solution is viable;
- **H2**: Services must be able to handle several concession data: ensures that services implement multi-tenancy, as well as ensuring that data is not stored in the wrong tenant's database;
- **H3**: Transaction and Photo creation endpoints should respond in less than two seconds: ensures that the Road Side Equipment (RSE) component receives early validation from the system, however this hypothesis does not require a complete success;
- **H4**: Messages should be consumed in less than 2 minutes: it is expected that message queues fill up constantly, so an acceptable time of processing should be ensured.

8.1.3 Methodology

Within the purpose of acquiring the necessary data to evaluate the measures and hypotheses aforementioned, a testing plan was defined. The methodology used is based on the execution of experiments, in an environment as close as possible to the production environment. It is a consecutive three step process, started by technical quality, proceeded by isolated load tests, ended by a monitoring process of the workload for two concessions.

The evaluation process is started by assessing the software components in a technical way. The technical quality of each micro-service was assured with the implementation of unit and integration tests, with the support of Junit 5[1] and Mockito[2], as well as the usage of MockServer[3] for the cases with external communication, and Testcontainers[4] as an in-memory data storage facility. Additionally, the code quality and test coverage was also considered, for the micro-service's quality, by the usage of SonarQube code analysis.

Having the technical quality ensured, it is followed by a testing phase of isolated load tests. This step is focused on testing the limits of the micro-services, by evaluating the resource management, such as memory and CPU usage, as well as API response times. These tests were developed with the support of Gatling[5].

With the isolated load tests phase terminated, and the micro-services resources fine-tuned, it is time for an extensive monitoring process, to a simulation of the production workload. This step is focused on monitoring all software components, by capturing its metrics and measures, with the usage of Statful[6] as a telemetry tool.

After collecting the results from each measure, it is important to access how these results relate to the hypotheses and if they verify, or not.

[1] https://junit.org/junit5/
[2] https://site.mockito.org/
[3] https://mock-server.com/
[4] https://testcontainers.org/
[5] https://gatling.io/
[6] https://statful.com/

8.2 Results

This section demonstrates the analysis of the results acquired from following the testing methodology aforementioned. The result analysis enables the evaluation of the solution, with the intent to verify if the goals were reached, to determine the remaining work, and other concerns that may arise. The following sections cover the testing methodology applied to each step mentioned on section 8.1.3. For the sake of simplicity and readability, the results present on this dissertation are focused on the use case presented at section 6.6.1, which includes the main entry-point of the system.

8.2.1 Technical Quality

The first step in the experimentation phase is to test the technical quality of the solution. This experimentation is essential to ensure that the results are not influenced by poorly implemented requirements. It is based on developing unit and integration tests for system features, with a special consideration for its test coverage. Additionally, the code-base of the micro-service is analysed, in order evaluate code quality standards, with the support of SonarQube.

Unit Testing

Unit testing is a technique of software testing, where individual units of software are tested. In the context of object-oriented design, an unit is considered as a single class, this means that testing is focused on covering the class functionality and behavior. This causes, in most cases, mocking of external classes. The purpose of these tests is to validate the behavior of the unit, in order to verify if the offered functionality matches with the requirements.

The frameworks Junit 5 and Mockito were used to develop unit tests. Junit 5 is an extensive test engine, developed specially for Java and has been the *de facto* standard. On the other hand, Mockito is a mocking framework that was used to describe the behavior of the mocked component, following a behavior driven dialect [70].

The implemented unit tests are targeted at testing classes present in `services` package, which contain testable business logic. The rationale for selecting which units to test, is based on evaluating if the unit has relevant code to be tested. For instance, testing DTO classes are not important, as they're just data bags. This impacts code coverage measuring to an extent, however the test quality is superior, as the time spent to write data bag tests, can be used to implement more functionality-driven tests.

```
1  @ExtendWith(MockitoExtension.class)
2  class ValidationServiceTest {
3      final String expectedTenant = "TENANT_ID";
4      ValidationService victim;
5      @Mock ClassMismatchValidation classMismatchValidation;
6      @Mock ValidationAssigner validationAssigner;
7
8      @Test
9      void testFailedClassMismatchValidation() {
10         given(validationAssigner.getFirstStageValidations(any()))
11                 .willReturn(Collections.singletonList(classMismatchValidation));
12         given(classMismatchValidation.validateTransaction(any(), eq(null)))
13                 .willReturn(ValidationResult.<ClassMismatchRule>newBuilder()
14                     .withValidationType(ValidationType.CLASS_MISMATCH)
15                     .withParameter(ClassMismatchRule.HIGHER_CLASS)
16                     .withSuccessful(false).build());
17
18         final var result = supplyForTenant(expectedTenant, () -> victim.validate(transactionValidation));
        // Act
19         final var expected = List.of(ValidationType.CLASS_MISMATCH);
20         assertEquals(expected, result.getFailedValidationsList());
21         assertEquals(expectedTenant, result.getTenantId());
```

```
    }
    @Test
    void testPassingExemptValidationByCategory() {
        final var transactionValidation = TransactionValidation.newBuilder()
                .setExemptCategory(1).build();

        given(validationAssigner.getExemptValidation(any()))
                .willReturn(Optional.of(exemptValidation));
        given(exemptValidation.validateTransactionWithParameter(any(), any(), eq(null)))
                .willReturn(ValidationResult.<Void>newBuilder()
                    .withValidationType(ValidationType.EXEMPT)
                    .withSuccessful(true).build());

        final var result = supplyForTenant(expectedTenant, () ->
                victim.validate(transactionValidation)); // Act
        final var expected = Collections.emptyList();

        assertEquals(expected, result.getFailedValidationsList());
        assertEquals(expectedTenant, result.getTenantId());
    }
}
```

Listing 8.1: Validation functionality unit test

Code listing 8.1 demonstrates an implementation of a unit test, with two test cases, focused on the functionality offered by the class `ValidationService`, present in the Validation micro-service. This class is responsible for applying validations to a given Transaction, and generating a result output that contains the failed validations (if any), for the provided Transaction.

Each micro-service contains unit tests, with its test cases having a success rate of 100%. This step of the experiment falls under the hypothesis H1, which validates it with success.

Integration Testing

Integration testing is a technique of software testing, that "determine if independently developed units of software work correctly when they are connected to each other" [71]. In the context of this project, a module can be seen as a micro-service, which leads to design integration tests as functionality of the micro-service. For instance, for Validation micro-service, an integration test covered the functionality offered by the `validate-transaction` queue, having another test covering the data fetching of active validations for the concession. The main purpose of these tests is to ensure if data inputs are not voided, as well as evaluating the behavior of the micro-service as a whole for the tested functionality. MockServer, TestContainers and Spring Test libraries were used to develop integration tests.

MockServer library allows to mock HTTP requests easily, by providing configuration endpoints to return specific payloads for specific requests, also known as HTTP expectations. This is used to test features that need to communicate over HTTP, with other micro-services in order to provide the described functionality, and its setup is executed during the test case setup phase.

Testcontainers is a Java library developed to provide throwaway instances of commonly used database systems, such as MySQL. During test setup phase, this library provides a database instance of choice, in order to use as a persistence layer. In contrast to common in-memory databases, such as H2[7], the database instance provided by the library is an actual MySQL engine. It is containerized and disposable, and it allows the selection of a wide variety of engines, to reach a persistence layer as close as possible to a production environment.

Spring Test is a Spring module that provides several support components, that simplify the writing of tests that uses the Spring Framework. This library provides a middleware layer, with extensions to Junit framework, that makes it easy to handle a Spring's application

[7] `https://h2database.com`

context and removes the need to manage database transaction during tests. Spring Test is used as a base for an integration test.

```
@ExtendWith(SpringExtension.class)
class RegistrationConsumerTest extends BaseIntegrationTest {
    @Autowired RegistrationConsumer registrationConsumer;

    @Test
    void registerValidMessage() {
        final SchedulingRegistration schedulingRegistration = buildRegistration()
                .setTenantId(BASE_TENANT_ID)
                .setCronExpression(CRON_EACH_HOUR).build();

        registrationConsumer.listener(createSqsMessage(schedulingRegistration)); // Act

        final var jobs = jobRepository.findAll();
        assertEquals(1, jobs.size());
        assertEquals(schedulingRegistration.getServiceName(), jobs.get(0).getJobId().getServiceName());
        assertEquals(schedulingRegistration.getEventName(), jobs.get(0).getJobId().getEventName());

        final var events = eventRepository.findAll();
        assertEquals(1, events.size());
        assertEquals(schedulingRegistration.getServiceName(), events.get(0).getJob().getJobId().
 getServiceName());
        assertEquals(schedulingRegistration.getEventName(), events.get(0).getJob().getJobId().
 getEventName());
        assertEquals(1, OffsetDateTime.now().plusHours(1).compareTo(events.get(0).getExecuteAt()));
    }

    @Test
    void registerEventInThePast() {
        final SchedulingRegistration schedulingRegistration = buildRegistration()
                .setTenantId(BASE_TENANT_ID)
                .setCronExpression("0 0 0 ? * * 1990").build();
        assertThrows(InvalidRegistrationException.class, () ->
                registrationConsumer.listener(createSqsMessage(schedulingRegistration)));
    }

    @Test
    void registerInvalidCronExpression() {
        final SchedulingRegistration schedulingRegistration = buildRegistration()
                .setTenantId(BASE_TENANT_ID)
                .setCronExpression("invalid cron").build();
        assertThrows(InvalidRegistrationException.class, () ->
                registrationConsumer.listener(createSqsMessage(schedulingRegistration)));
    }
}
```

Listing 8.2: Scheduler event registration integration test

Code listing 8.2 demonstrates an implementation of an integration test, with three test cases, focused on the functionality offered by Scheduler micro-service, the registration of a scheduled event. This use case is triggered by a message event, received in a specific queue, with a message containing event payloads and crons, and will store them in the database for a later execution.

Each micro-service contains several integration tests, with its test cases having a success rate of 100%. This step of the experiment falls under the hypothesis H1 and H2, which validates them with success.

Code Quality

In order to evaluate the code quality in a quantitative manner, code analysis routines were executed for each micro-service, with the support of SonarQube. This platform is designed to provide continuous inspection of code quality, in order to induce stat analysis to detect code smells, bugs, duplicated code and even security vulnerabilities. This analysis will lead to the generation of a user-accessible report, that details each issue present on the code-base, along with a suggestion on how to address the issue.

In the context of this dissertation, SonarQube is included in the build process, demonstrated at section 6.7, which means that it will run the static analysis for each pipeline run, and generate a report for the micro-service in question, for every micro-service in the platform.

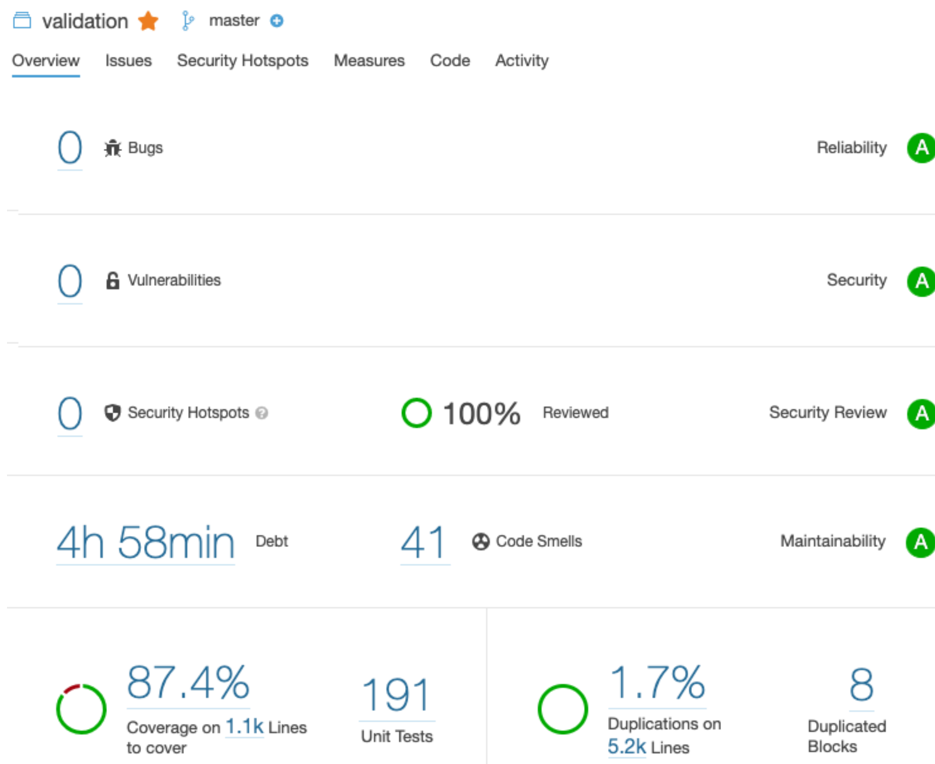

Figure 8.1: SonarQube code analysis: Validation micro-service

Figure 8.1 displays a SonarQube code analysis dashboard, for Validation micro-service. As demonstrated by the figure, this micro-service is free from bugs, vulnerabilities and security hotspots, which are the most important measures concerning micro-service security. Additionally, it has a high test coverage, of 87.4% with 191 unit tests. However, it does still need some work to do, as it contains 41 code smells, which adds up to the micro-service's technical debt.

Annex E contains all SonarQube code analysis measures, for each micro-service. A general state of the project's quality is demonstrated the following table, where averages of the measures are presented.

Table 8.1: SonarQube code analysis summary - average values

Bugs	Vulnerabilities	Security Hotspots	Technical Debt	Code Smells	Test Coverage	Unit Tests	Code Duplication
0	0	0	20h 46m	58	78,75%	94	2.24%

After examining table 8.1, it can be seen that the system is widely covered by unit and integration tests, whereas it counts with an average value of 78,75% test coverage. Most micro-services are suffering from technical debt, due to code smells. However, SonarQube classifies the systems with its highest grade for maintainability, which indicates that the present code smells causes no harm. Nonetheless, the technical debt should be addressed in the near future.

It has been demonstrated that each step of the experiment has been fulfilled with success, correlated with the aforementioned hypotheses. Therefore it is possible to conclude that the multi-concession system has technical quality, fully confirming hypothesis H1.

8.2.2 Isolated Load Tests

The isolated load test experiment is focused on testing the limits of a micro-service, by filling the micro-service's interfaces with workload, using HTTP or message queuing, and evaluating its resource management, such as memory and CPU usage, and response times percentiles. This experiment is supported with the usage of the tool Gatling.

Gatling is an open-source, highly capable tool aimed at load testing, built for highly performant services. Its premise is providing an easy to use load testing tool as code, and supports HTTP, WebSocket and Java Message Service (JMS). Gatling also provides an extensive set of measures, such as response time percentiles, success requests, and even number of active users at a given time.

With the intent of maintaining consistency between micro-services, all tests were executed with the same hardware, in a containerized context, as demonstrated in table 8.2.

Table 8.2: Load test hardware specifications

CPU	RAM	Operating System
Intel Core i7-4770HQ @ 2.20GHz	1GB	debian 9.3 (bpo9+1)

Each micro-service has been load tested, however, the test case is focus for this section is based on the use case demonstrated at section 6.6.1, and it is focused on the communication that happens between RSE component and Proxy micro-service, most specifically, targeting the HTTP endpoint that will lead to the creation of a transaction in the system.

In order to establish a base line, for the micro-service, of a low usage model, an initial test run will be executed for 2100 requests, with up to 6 simultaneous users, for a single instance of the Proxy micro-service.

Table 8.3: Load test for Transaction creation: executions and response time

Executions				Response Time (ms)							
Total	OK	KO	Req/s[1]	Min	50th pct[2]	75th pct[2]	95th pct[2]	99th pct[2]	Max	Mean	Std Dev[3]
2100	2100	0	10.55	98	178	198	391	522	4191	198	289

1 Requests per second 2 Percentile 3 Standard Deviation

The data in table 8.3 demonstrates that no HTTP request has failed to deliver, with a rate of 10.55 requests per second, and a mean response time of 198ms. However, a maximum response time of 4191ms was registered, which will be considered as an outlier, considering that it was only reached once during the load test.

Figure 8.2: Load test for Transaction creation: response time percentiles over time

Figure 8.2 demonstrates the API response time percentiles, over time, with the indication of the active users during the requests. This graph is generated automatically by Gatling, at the end of each test run.

Having the baseline established, a few test cases must be executed, in order to evaluate the solution's scalability, in order to assess hypothesis H3. The following test cases are going to be covered and compared:

- [A] 2100 requests, with a rate of 10.55 requests per second - for 1 and 3 instances;
- [B] 10000 requests, with a rate of 33.88 requests per second - for 1 and 3 instances;

Table 8.4: Load test for Transaction creation: response times for two cases

Scenario	Instances	Response Time (ms)		
		Mean	Min	Max
[A] 2100 10.55 Req/s	1	198	98	4191
	3	143	86	411
[B] 10000 33.88 Req/s	1	1230	90	4930
	3	480	91	1300

Table 8.4 demonstrates the obtained the results for the two test cases aforementioned. As expected, mean response times for three instances are lower than for the case with a single instance, however some spikes of maximum response times are still present when performing with just one instance. As detected earlier, an outlier value was detected for scenario [A], however on scenario [B] the outlier is not present as in general, the response time of the API is mostly over 1500ms, being the mean value 1230 due to earlier requests, that does have a lower amount of active users.

Table 8.5: Load test for Transaction creation: response times variance

Scenario	Mean Response Time Variance
[A] 2100 Req/s 1 vs 3	28%
[B] 10000 Req/s 1 vs 3	61%

Table 8.5 exhibits a comparative analysis between the two load tests, where the variance of mean response times between number of instances is calculated. It is apparent that the three instances approach as out-performed vastly, reaching an amount of 61% for scenario [B].

It is safe to conclude that using a scaled system, with multiple instances, will allow for transactions to be processed by the system in a performant manner. The more instances, the better the response time and the better is the micro-service's throughput and responsiveness. Conclusively, hypothesis H3 is considered valid, given the fact that average response times were registered under 2000ms in both test cases.

8.2.3 Production Monitoring

The production monitoring experiment is based on analysing the fully deployed system, for a specific amount of time, while the system is working with real data, just like it would in a production manner. For this experiment, the system was deployed in a cluster that is a copy of the production environment, named `prelive`, and its measures were monitored with the support of Statful as a telemetry tool.

Statful is a telemetry system, that allows for the instrumentation of software components, by capturing and orchestrating metrics in real-time. Statful offers "a Metrics as a Service platform that enables users to monitor and visualize real-time data in a centralized point-of-view for business, application and system metrics" [72].

In the context of this dissertation, the Statful tool was used to monitor micro-service's HTTP and message queuing interfaces, in order to measure average response times, amount of requests and even percentiles. In order to correlate the measure analysis with the hypotheses introduced earlier, the monitoring process is split by two test cases: transaction creation, both HTTP and message queuing, and transaction validation.

Transaction Creation - HTTP

Similarly to the load tests aforementioned, this test case is focused on monitoring the HTTP communication between the RSE component and Proxy micro-service, in order to measure and analyse the API response time under the normal workload of the system. The sample data generated by this test case, is provided by measuring the system for 24 hours, dated at 15th of August 2020, for two concessions: Transmontana (A4, Portugal) and ACEGA (AP-50, Spain).

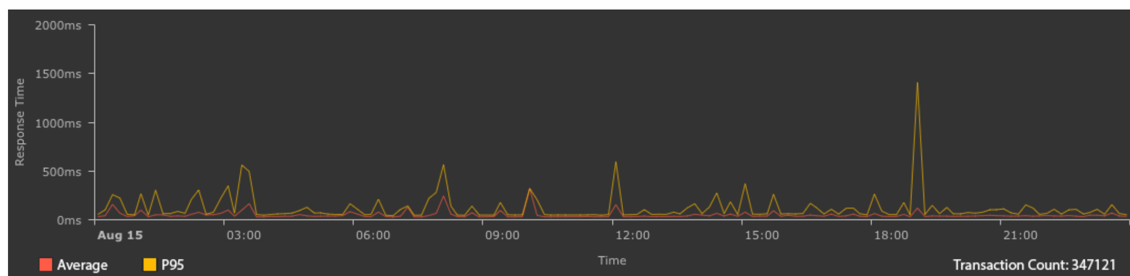

Figure 8.3: Response time average values for Transaction creation

Figure 8.3 demonstrates a line chart containing averages, over time, of all entry points of the measured response times. A total of 347121 transactions were processed during these 24 hours, registering a spike of response time around 19:00, at the 95 percentile. Table 8.6 shows a summary of this chart.

Table 8.6: Response time average for Transaction creation: summary

Response Time (ms)		
Min	**Average**	**Max**
23.6	49.6	1592

Transactions: 347121

Evaluating the information present in figure 8.3, together with the summary demonstrated at table 8.6, it is possible to rule hypothesis H3 as valid, given the fact that the average response time falls way below the 2000ms threshold, when using the system for two concessions.

Transaction Creation - Message Queuing

This test case is the variant of message queuing communication, for the use case of creation a transaction. This test case is focused on monitoring the message queue communication between Proxy micro-service and Transaction micro-service, in order to measure and analyse amount of time a transaction creation event is waiting on the queue in order to be processed. The sample data generated by this test case, is provided by measuring the system for 24 hours, dated at 15th of August 2020, for two concessions: Transmontana (A4, Portugal) and ACEGA (AP-50, Spain).

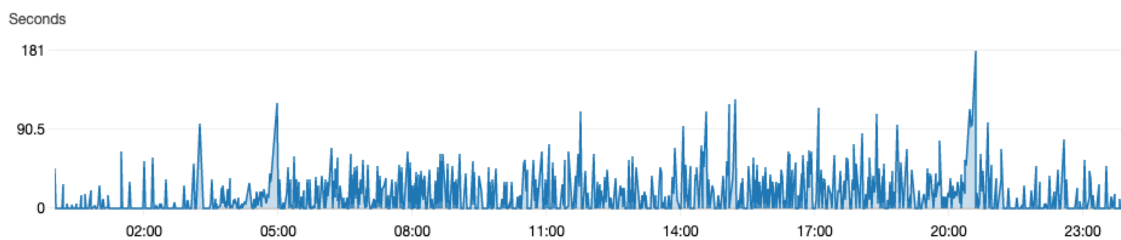

Figure 8.4: Average age value for Transaction creation message event

Figure 8.4 demonstrates a line chart containing averages of seconds, over time, of event messages that were awaiting, in the message queue, to be processed by the micro-service. A total of 347121 transactions were processed during these 24 hours, registering a spike of response time around 20:30. Table 8.7 shows a summary of this chart.

Table 8.7: Average age value for Transaction creation message event: summary

Response Time (s)		
Min	**Average**	**Max**
1	15.6	181
Transactions: 347121		

Evaluating the information present in figure 8.4, together with the summary demonstrated at table 8.7, it is possible to rule hypothesis H4 as valid, given the fact that the average response time falls way below the 120 seconds threshold, when using the system for two concessions. However, a spike of 181 seconds was detected and it must be addressed, in order to reduce the mean time to process, as this message queue orchestrates the most important part of the system, which is the transition of transactions status, that will lead to client billing.

Transaction Validation

In contrast to the transaction creation test case, the transaction validation test case is focused on monitoring the message queuing communication. It is focused on the communication between the Transaction micro-service and Validation micro-service, in order to

measure the processing time of each message queue response time under the normal workload of the system. The sample data generated by this test case, is provided by measuring the system for 24 hours, dated at 15th of August 2020, for two concessions: Transmontana (A4, Portugal) and ACEGA (AP-50, Spain).

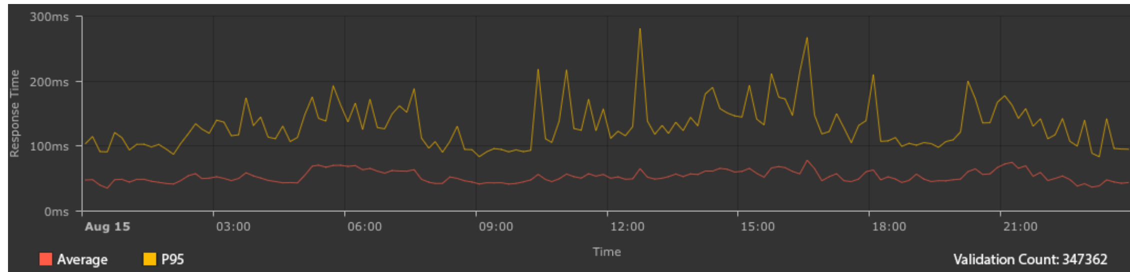

Figure 8.5: Response time average values for Transaction validation

Figure 8.5 demonstrates a line chart containing averages, over time, of all registered messages of the measured response times. A total of 347362 transaction validations were executed, during these 24 hours, registering no spikes, showing a steady message processing mechanism. However, a different between the executed validations and the created transactions was detected, as only 347121 transactions were created but Validation micro-service registered 347362 validations, which represents a variance of 241 transactions (0,0007%). Table 8.8 shows a summary of this chart.

Table 8.8: Response time average for Transaction validation: summary

Response Time (ms)			Validation Count	
Min	**Average**	**Max**	**Expected**	**Actual**
16	120.75	1148.33	347121	347362
Validations: 347362			Variance: 241 (0,0007%)	

Evaluating the information present in figure 8.5, together with the summary demonstrated at table 8.8, it is possible to rule hypothesis H3 as valid, for the Validation part of the process. Given the fact that the average response time falls around 120ms, which is considerably lower than the threshold. However, a variance between created and validated transactions was noticed. This variance will not be considered relevant for the following reasons:

- The variance percentage of transactions is minimal;
- Considering the fact that the communication between Transaction and Validation is asynchronous, some of the validated transactions could have been in queue before the aforementioned time-frame, but only processed during the testing time-frame;
- A micro-service possibly could deny a HTTP request, which would trigger the retry mechanism of Validation, which would cause a reprocess of the validation event.

Chapter 9

Conclusion

The conclusion is the final chapter of this dissertation, it is focused on demonstrating a critical analysis of the project as a whole, and is divided into two sections. The first section is focused on approaching the solution and the dissertation problem, in a critical manner, by showcasing the objectives, a general outline of the implemented solution, its challenges, the evaluation process and finally, limitations that the solution might have. The second section, demonstrates points to be considered for the future work of this solution, considering the growth of the platform.

9.1 Critical Analysis

The main objective of the development of this dissertation, was to prove that the implementation of a multi-concession toll collection system, would be a feasible solution for the problem presented by Globalvia, of numerous sub-systems per concession. Several challenges arose when evaluating the functional and non-functional requirements for this solution, considering that the problem is included in a very business-driven domain, with numerous integrations and possible flows. Additionally, the amount of functional requirements to support the base functionality for a concession were considerable complex. The research process demonstrated that no other company is publicly open about supporting multi-concession for their toll collection systems, which motivated to proceed with the implementation.

In order to solve the problem present in this dissertation, a multi-concession, cloud-based, scalable, toll collection and validation system was developed, that responds to the concerns presented at section 1.2. This system provides the base functionality for a concession to be integrated, with features such as transaction creation, photo processing, configurable transaction validation pipelines, and even scheduled events that could trigger all sorts of components, such as issuer client billing. Additionally, it offers a logical segregation between Operational Back Office (OBO) and Commercial Back Office (CBO) domains. The solution takes advantages of a micro-services architectural style, as well as following a design aimed for the cloud hosting and ready to scale, since its inception. This allows the system to grow at any given whether being adding a new micro-service, or just duplicating instances to process data faster. Each micro-service is fully bundled with production-ready technology, such as API documentation, unit tests, integration tests and even build and deployment automation for Kubernetes.

The biggest challenges of this dissertation were grounded on the design process of the multi-concession system, as numerous constraints were put in place. Initially, a rationale process was executed in order to understand how the OBO and CBO domains could be split into

two logical components, without causing code duplication over the services. With that, the constraints of multi-concession and scalability got in the way, as the services had to be small, in order to promote scalability, but also highly configurable, in order to provide multi-concession support.

The evaluation process presented a challenge in itself, as such system does not directly correlate to the satisfaction of the user of the motorway, but for the concession instead. Additionally, the multi-concession system does not provide a visual component for the back office operators just yet, so the evaluation process is based on evaluating the micro-services scalability, response times and technical quality. The experimentation process demonstrated positive results, validating all defined hypotheses and showcasing highly performant micro-services.

Lastly, the system is deemed production-ready, for concessions that operate toll infrastructure mechanisms based on manual and automatic tolling, such as Globalvia ACEGA (AP-50, Spain). However, an existing limitation exists for concessions that operate with Multi-Lane-Free-Flow (MLFF). This limitation is intentional and was considered during the inception of the dissertation, however, it is fully driven by the implementation of external integrations and it was not considered for the context of the dissertation, whereas the main objective of the research is behind the multi-concession requirement.

9.2 Future Work

Software development never ends. And it is not because of development failure, but every solution needs continuous evolution and support, whereas this multi-concession system is no different. There is always something to improve, or some new requirement to implement. And even if there is not, technology evolves and what is today, the right way to do it, tomorrow it might not be. Software development process is dynamic and it is a constant strive for excellence.

As identified in section 8.2, the implemented solution fulfills the technical quality experiment, however, there is room for improvement. SonarQube provided a list of issues that some micro-services suffer from, such as code smells, that contribute to technical debt, and not an optimal value of test coverage.

Moreover, as demonstrated on the previous section 9.1, this system does not support MLFF tolling mechanism, which is an important feature for Globalvia, considering that it is the main mechanism implemented in Portugal motorways.

The multi-concession system could also use an improvement in its security, by adding a layer of concession-based authentication. This authentication process could replace the tenant identifier in the system, by using a single value that would identify the user, as well as the concession in place. Although the system is deployed under Globalvia private network, this would increase the security for each concession.

Lastly, this platform could also implement a multi-concession back-office component, with the intent of having a single website for Globalvia concession management. This back-office could make use of the GraphQL component presented in section 6.2, and could be designed in order to provide a seamless experience for back-office managers that are responsible for more than one concession.

Bibliography

[1] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[2] Wei-Hsun Lee et al. "Electronic toll collection based on vehicle-positioning system techniques". In: *IEEE International Conference on Networking, Sensing and Control, 2004*. Vol. 1. IEEE. 2004, pp. 643–648.

[3] Wern-Yarng Shieh, Chen-Chien James Hsu, and Ti-Ho Wang. "A problem of infrared electronic-toll-collection systems: the irregularity of LED radiation pattern and emitter design". In: *IEEE Transactions on Intelligent Transportation Systems* 12.1 (2010), pp. 152–163.

[4] Simson L Garfinkel, Ari Juels, and Ravikanth Pappu. "RFID privacy: An overview of problems and proposed solutions". In: *IEEE Security & Privacy* 3.3 (2005), pp. 34–43.

[5] Alan R Hevner et al. "Design science in information systems research". In: *MIS quarterly* (2004), pp. 75–105.

[6] Ken Peffers et al. "A design science research methodology for information systems research". In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[7] PricewaterhouseCoopers Advisory S.p.A. "Evaluation and future of road toll concessions". In: (2014).

[8] Emanuel S Savas and Emanuel S Savas. "Privatization and public-private partnerships". In: (2000).

[9] Alain Fayard et al. "Analysis of highway concession in Europe". In: *Research in Transportation Economics* 15 (2005), pp. 15–28.

[10] Jakub Rajnoch. "New Approaches to Distance Based Charging". In: *14 th World Congress on Intelligent Transport Systems. Beijing*. 2007.

[11] Lucia Manzi. *Study on" State of the Art of Electronic Road Tolling"*. Tech. rep. 2015.

[12] Bertil Hylén, Jari Kauppila, and Edouard Chong. "Road Haulage Charges and Taxes: Summary analysis and data tables 1998-2012". In: International Transport Forum Discussion Paper. 2013.

[13] Margaret O'Mahony, Dermot Geraghty, and Ivor Humphreys. "Distance and time based road pricing trial in Dublin". In: *Transportation* 27.3 (2000), pp. 269–283.

[14] Raymond Tillman. "Shadow tolls and public-private partnerships for transportation projects". In: *Journal of Structured Finance* 3.2 (1997), p. 30.

[15] Mark G Santos and Bruno F Santos. "Shadow-tolls in Portugal: How we got here and what were the impacts of introducing real tolls". In: *Association for European Transport and Contributors* (2012).

[16] Pedro Miguel Lopes Pinto. *Sistemas de Cobrança de Portagens - Operação, Interoperabilidade e Cobrança Coerciva*. Tech. rep. 2011. url: `http://www.crp.pt/docs/A48S167-8_CRP_T7_073.pdf`.

[17] Ascendi Group. *Role Based Access Controls Applied to Electronic Toll Collection*. 2016. url: `https://www.cncs.gov.pt/content/files/c-days_-_ascendi_v28_11_2016.pdf`.

[18] Miguel Pedras. "BIT é agora A-To-Be e aposta no mercado externo". In: (2017). url: http://www.transportesemrevista.com/Default.aspx?tabid=210&language=pt-PT&id=56642.

[19] A-to-Be. *A-to-Be MoveBeyond™ for Tolling*. url: https://www.a-to-be.com/mobility-payments/a-to-be-movebeyond-line/movebeyond-for-roadoperators/.

[20] A-to-Be. *A-to-Be LinkBeyond™*. url: https://www.a-to-be.com/mobility-payments/a-to-be-linkbeyond-line/.

[21] A-to-Be. *A-to-Be Atlas™*. url: https://www.a-to-be.com/mobility-payments/a-to-be-atlas-line/.

[22] Peter Drucker. *Innovation and entrepreneurship*. Routledge, 2014.

[23] Peter A Koen et al. "Fuzzy front end: effective methods, tools, and techniques". In: *The PDMA toolbook 1 for new product development* (2002).

[24] Peter Koen et al. "Providing Clarity and A Common Language to the "Fuzzy Front End"". In: *Research-Technology Management* 44.2 (2001), pp. 46–55. doi: 10.1080/08956308.2001.11671418.

[25] Wolfgang Ulaga and Andreas Eggert. "Relationship value and relationship quality: Broadening the nomological network of business-to-business relationships". In: *European Journal of marketing* 40.3-4 (2006), pp. 311–327.

[26] Jozee Lapierre. "Customer-perceived value in industrial contexts". In: *Journal of business & industrial marketing* (2000).

[27] Alexander Osterwalder. "The business model ontology a proposition in a design science approach". PhD thesis. Universit 'e de Lausanne, Faculty é des hautes 'e tudes commerc, 2004.

[28] Paul Roberts. *Product Excellence using Six Sigma - Quality Function Deployment*. Warwick Manufacturing Group, 2009.

[29] Richard N Taylor, Nenad Medvidovic, and Eric Dashofy. *Software architecture: foundations, theory, and practice*. John Wiley & Sons, 2009.

[30] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[31] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590.

[32] Martin Fawler and James Lewis. *Microservices*. Mar. 2014. url: https://www.martinfowler.com/articles/microservices.html.

[33] Fabrizio Montesi and Janine Weber. "Circuit breakers, discovery, and API gateways in microservices". In: *arXiv preprint arXiv:1609.05830* (2016).

[34] Chris Richardson. *Microservices Patterns: With Examples in Java*. Manning Publications, 2019.

[35] Ben Stopford and Sam Newman. *Designing Event-Driven Systems*. O'Reilly Media, 2018.

[36] Patrick Th Eugster et al. "The many faces of publish/subscribe". In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131.

[37] Martin Fawler. *What do you mean by "Event-Driven"?* Jan. 2017. url: https://martinfowler.com/articles/201701-event-driven.html.

[38] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[39] Martin Fawler. *Continuous Integration*. May 2006. url: https://martinfowler.com/articles/continuousIntegration.html.

[40] Martin Fawler. *Continuous Delivery*. May 2013. url: https://martinfowler.com/bliki/ContinuousDelivery.html.

[41] Samarpit Tuli. *Learn How to Set Up a CI/CD Pipeline From Scratch*. 2018. url: https://dzone.com/articles/learn-how-to-setup-a-cicd-pipeline-from-scratch.

[42] André B Bondi. "Characteristics of scalability and their impact on performance". In: *Proceedings of the 2nd international workshop on Software and performance*. 2000, pp. 195–203.

[43] Martin L Abbott and Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Addison-Wesley Professional, 2015.

[44] Vivek Juneja. *From Monoliths to Microservices: An Architectural Strategy*. 2016. url: https://thenewstack.io/from-monolith-to-microservices.

[45] Matej Artac et al. "DevOps: introducing infrastructure-as-code". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2017, pp. 497–498.

[46] Kief Morris. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.

[47] Cor-Paul Bezemer et al. "Enabling multi-tenancy: An industrial experience report". In: *2010 IEEE International Conference on Software Maintenance*. IEEE. 2010, pp. 1–8.

[48] Adeniyi O Abdul et al. "Multi-tenancy Design Patterns in SaaS Applications: A Performance Evaluation Case Study". In: (2018).

[49] Ralph Mietzner et al. "Combining Different Multi-Tenancy Patterns in Service-Oriented Applications". English. In: *Proceedings of the 13th IEEE Enterprise Distributed Object Conference (EDOC 2009)*. Ed. by IEEE Computer Society. IEEE, Oct. 2009, pp. 131–140. isbn: 978-0-7695-3785-6. doi: 10.1109/EDOC.2009.13. url: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2009-50&engl=0.

[50] Andrew P Sage and William B Rouse. *Handbook of systems engineering and management*. John Wiley & Sons, 2014.

[51] Chris Richardson. *Microservices Pattern: Decompose by business capability*. 2018. url: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html.

[52] Amazon. *Kubernetes on AWS*. 2019. url: https://aws.amazon.com/kubernetes/.

[53] Philippe B Kruchten. "The 4+ 1 view model of architecture". In: *IEEE software* 12.6 (1995), pp. 42–50.

[54] Terraform. *Introduction to Terraform*. 2020. url: https://www.terraform.io/intro/index.html.

[55] Terraform. *Resource Graph*. 2020. url: https://www.terraform.io/docs/internals/graph.html.

[56] Amazon. *Amazon Elastic Kubernetes Service*. 2020. url: https://aws.amazon.com/eks.

[57] Amazon. *Amazon Virtual Private Cloud*. 2020. url: https://aws.amazon.com/vpc.

[58] Amazon. *AWS EKS Managed node groups*. 2020. url: https://docs.aws.amazon.com/eks/latest/userguide/managed-node-groups.html.

[59] Amazon. *AWS Identity and Access Management (IAM)*. 2020. url: https://aws.amazon.com/iam.

[60] Amazon. *Amazon Relational Database Service (RDS)*. 2020. url: https://aws.amazon.com/rds.

[61] Amazon. *Amazon Simple Queue Service*. 2020. url: https://aws.amazon.com/sqs.

[62] IBM. *Dead-letter queues*. 2020. url: https://www.ibm.com/support/knowledgecenter/SSFKSJ_9.1.0/com.ibm.mq.pro.doc/q002680_.htm.

[63] Amazon. *Amazon S3*. 2020. url: https://aws.amazon.com/s3.

[64] Google. *Protocol Buffers - Language Guide proto3*. 2020. url: https://developers.google.com/protocol-buffers/docs/proto3.

[65] Roland Barcia. *How to architect an application with microservices (part 2)*. 2018. url: https://ibm.com/cloud/blog/architecting-applications-microservices-general-principles.

[66] The Linux Foundation. *Kubernetes Documentation - Service*. 2020. url: https://kubernetes.io/docs/concepts/services-networking/service/.

[67] Datawire. *Ambassador API Gateway - Github*. 2020. url: https://github.com/datawire/ambassador.

[68] Facebook. *GraphQL | A query language for your API*. 2020. url: https://graphql.org/.

[69] Olaf Hartig and Jorge Pérez. "Semantics and complexity of GraphQL". In: *Proceedings of the 2018 World Wide Web Conference*. 2018, pp. 1155–1164.

[70] Mazedur Rahman and Jerry Gao. "A reusable automated acceptance testing architecture for microservices in behavior-driven development". In: *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE. 2015, pp. 321–325.

[71] Martin Fawler. *IntegrationTest*. Jan. 2018. url: https://martinfowler.com/bliki/IntegrationTest.html.

[72] Statful. *Metrics for Devs: Introducing Statful's Free Plans*. 2018. url: https://medium.com/@StatfulHQ/metrics-for-devs-introducing-statfuls-free-plans-ddaddcdbf72b.

[73] The Linux Foundation. *Kubernetes Standardized Glossary*. 2019. url: https://kubernetes.io/docs/reference/glossary/?fundamental=true.

Appendix A

Quality Function Deployment

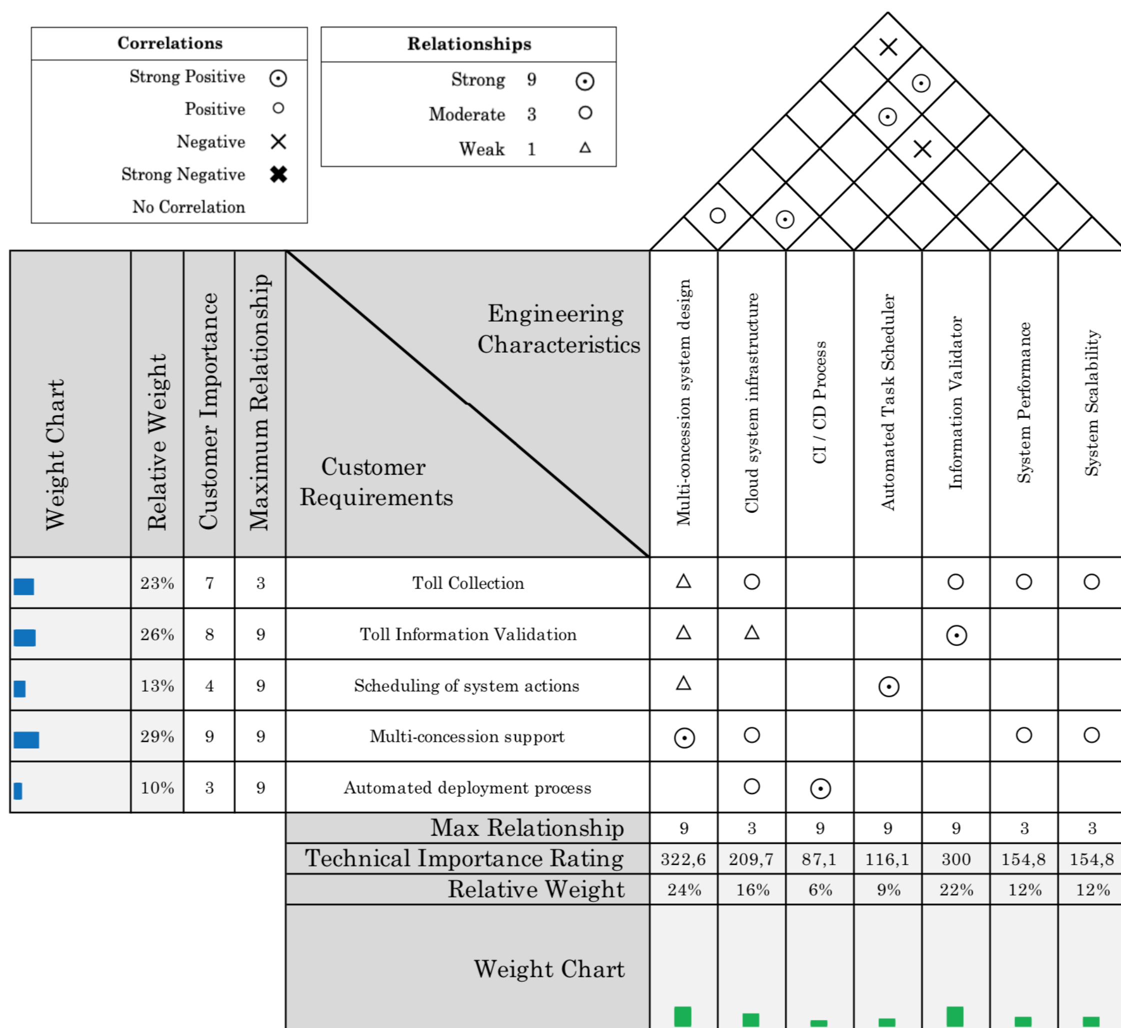

Figure A.1: Quality Function Deployment system

Appendix B

Complete domain model

Having a clear division between OBO and CBO domain is a strong motivation of this multi-concession system for the stakeholders. However, for the sake of simplicity when interpreting the system, a complementary domain model is represented, in the following figure B.1, to show the point of connection between domains.

The following table shows the existing classifier rules, as well as the decision taken.

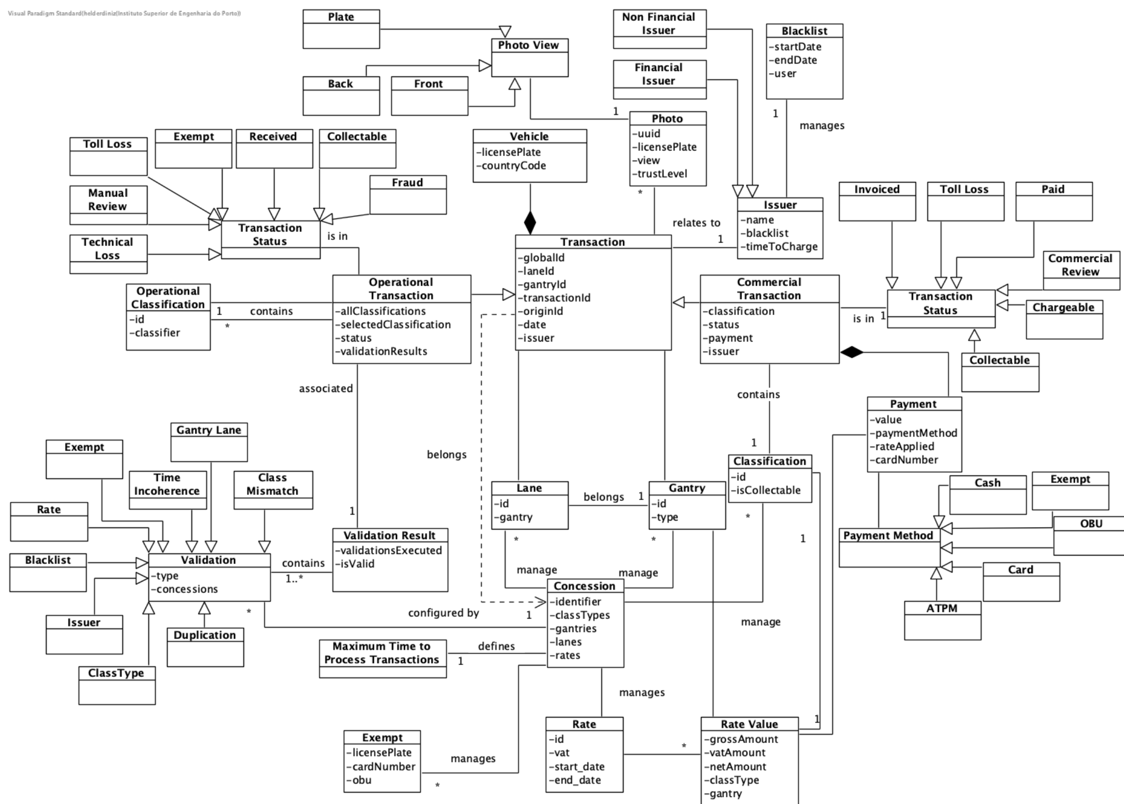

Figure B.1: Complete domain model

Appendix C

Class Mismatch validation decision mechanism

The class mismatch validation verifies if all the classifications, in the operational transaction, contain the same value. When this is not the case, this validation has a defined business rule to select the classification to use throughout the system, based a specific optional classifier rule that a concession can configure. It then matches the *classifier* value in the Operational Classification entity to select the classification to use.

The following table shows the existing classifier rules, as well as the decision taken.

Table C.1: Decision mechanism of the class mismatch validation

Classifier	Rule decision
higher-class	The system should consider the highest classification identifier when applying the rate
system-class	The system should consider the class assigned by the OCR when applying the rate
toll-collector	The system should consider the class assigned by the toll operator when applying the rate
manual-review	The system should send the transaction to manual review where it awaits input from an operator

Appendix D

Kubernetes Terminology

Kubernetes introduces several terms, for resource representation that are very specific for this technology. The following table D.1 includes technical terms and descriptions that provide useful context.

Table D.1: Kubernetes fundamental terminology - adapted from [73]

Term	Description
Annotation	A key-value pair that is used to attach arbitrary non-identifying metadata to objects
Applications	The layer where various containerized applications run
Cluster	A set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node
Container	A lightweight and portable executable image that contains software and all of its dependencies
Controller	control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state
DaemonSet	Ensures a copy of a Pod is running across a set of nodes in a cluster
Deployment	An API object that manages a replicated application, typically by running Pods with no local state
Image	Stored instance of a Container that holds a set of software needed to run an application
Ingress	Exposes HTTP/HTTPS routes from outside the cluster to within. Traffic routing is controlled by rules defined on the Ingress resource
Kubectl	A command line tool for communicating with a Kubernetes API server
Label	Tags objects with identifying attributes that are meaningful and relevant to users
Namespace	An abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster
Node	A node is a worker machine in Kubernetes
Pod	The smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster. A Pod is typically set up to run a single primary container. It can also run optional sidecar containers. Pods are commonly managed by a Deployment.
ReplicaSet	A ReplicaSet maintain a set of replica Pods running at any given time
Service	An abstract way to expose an application running on a set of Pods as a network service

Appendix E

SonarQube Code Analysis

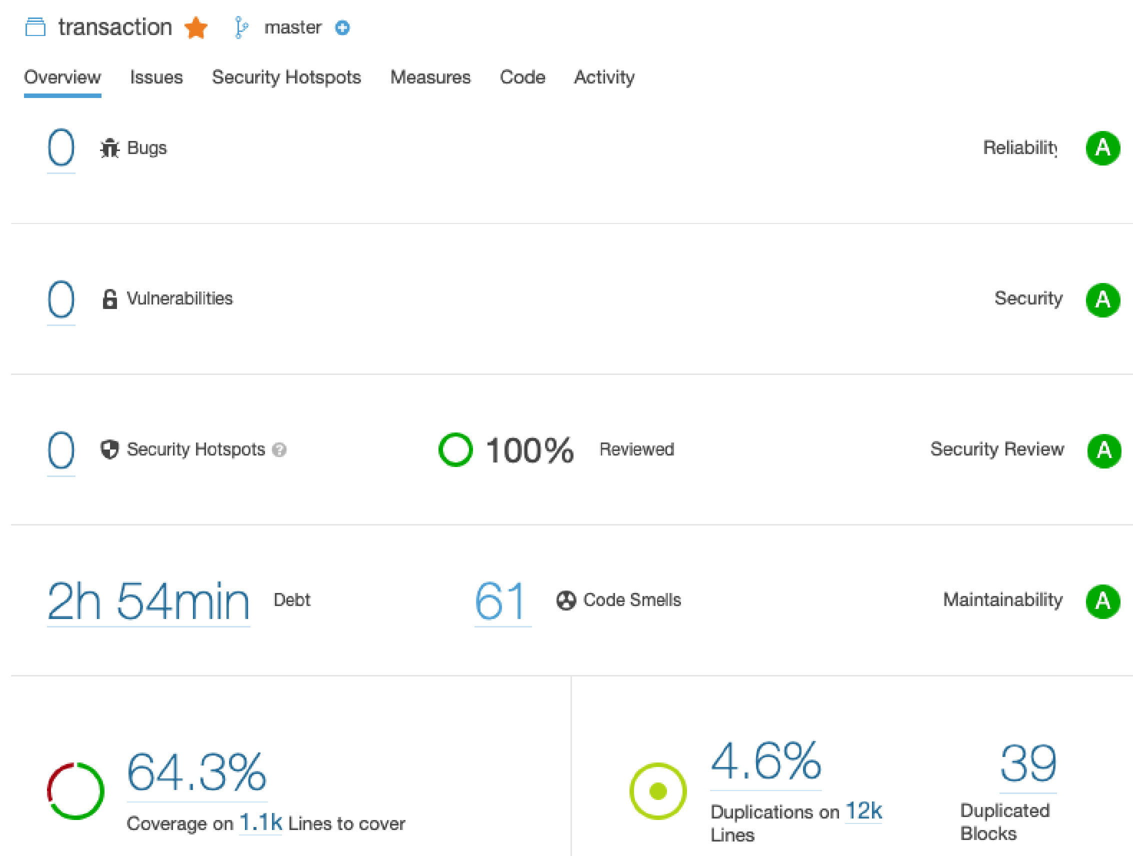

Figure E.1: SonarQube code analysis: Transaction micro-service

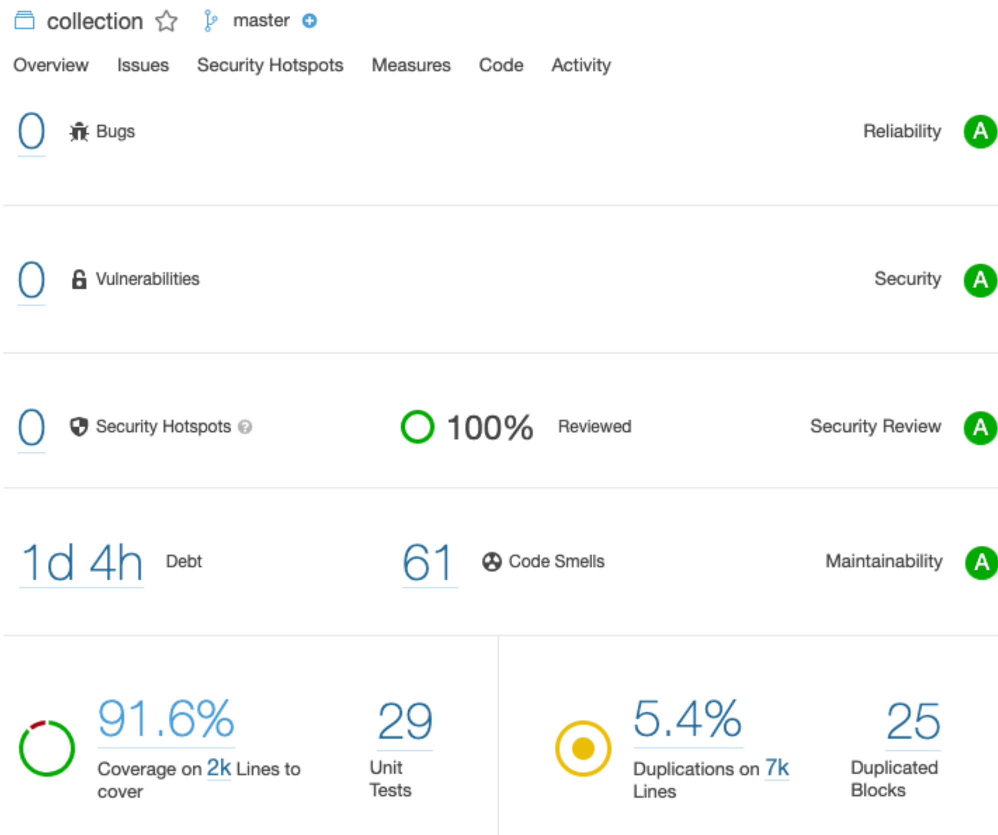

Figure E.2: SonarQube code analysis: Collection micro-service

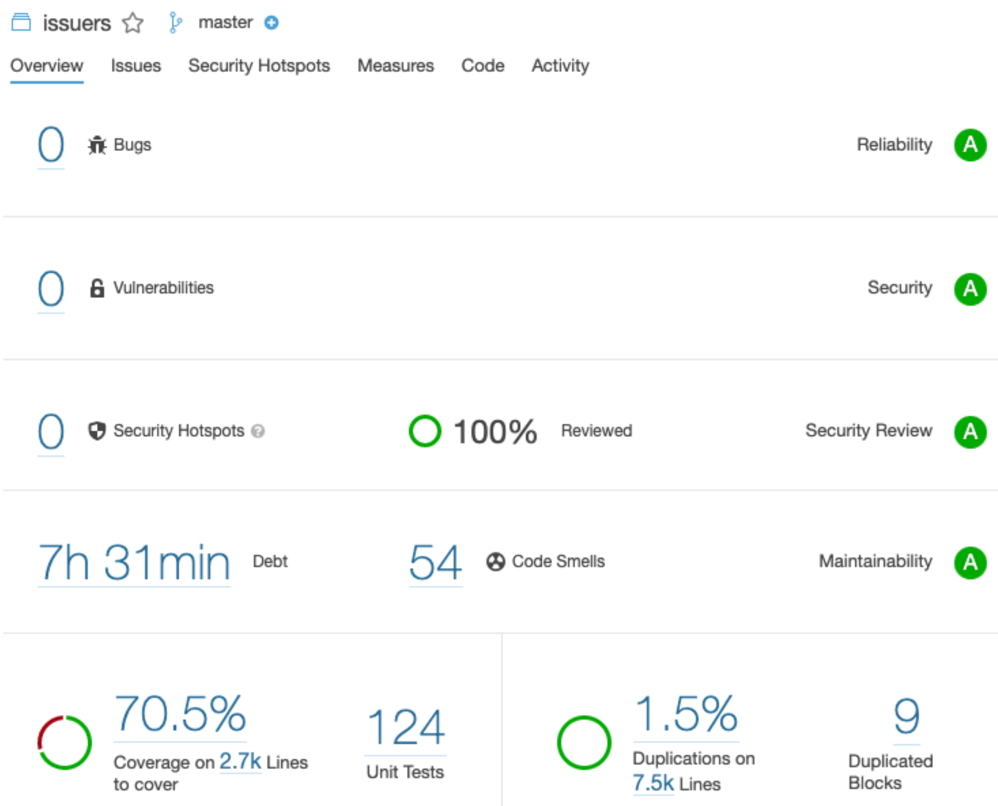

Figure E.3: SonarQube code analysis: Issuer micro-service

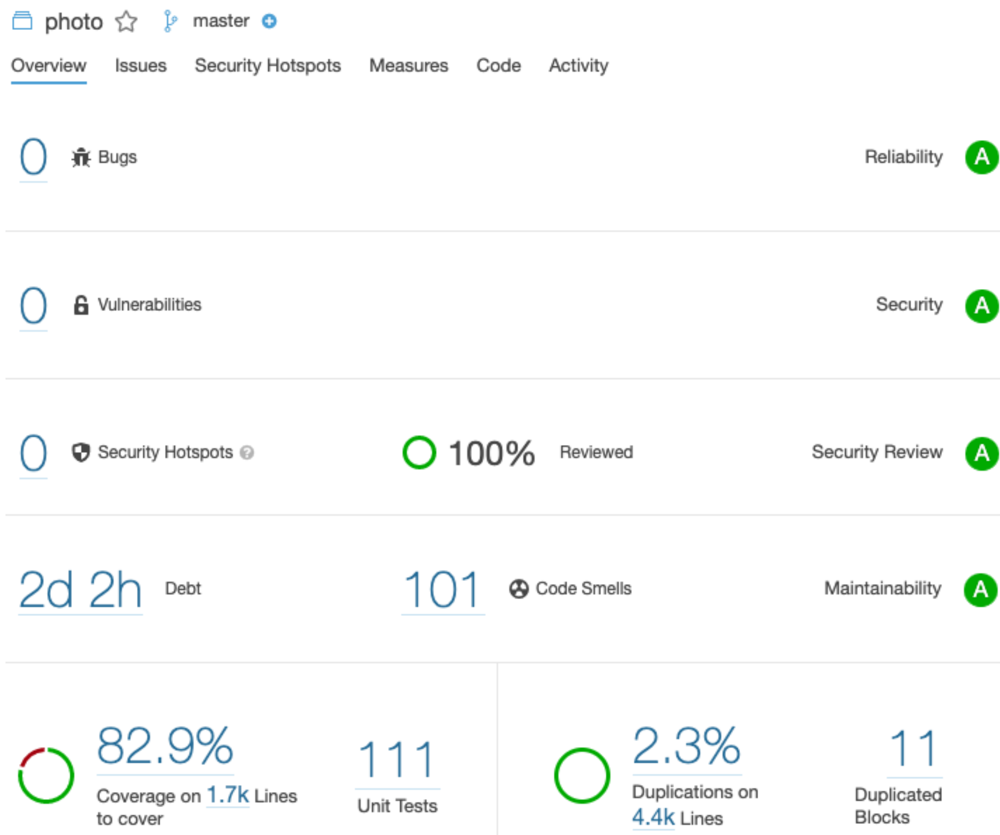

Figure E.4: SonarQube code analysis: Photo micro-service

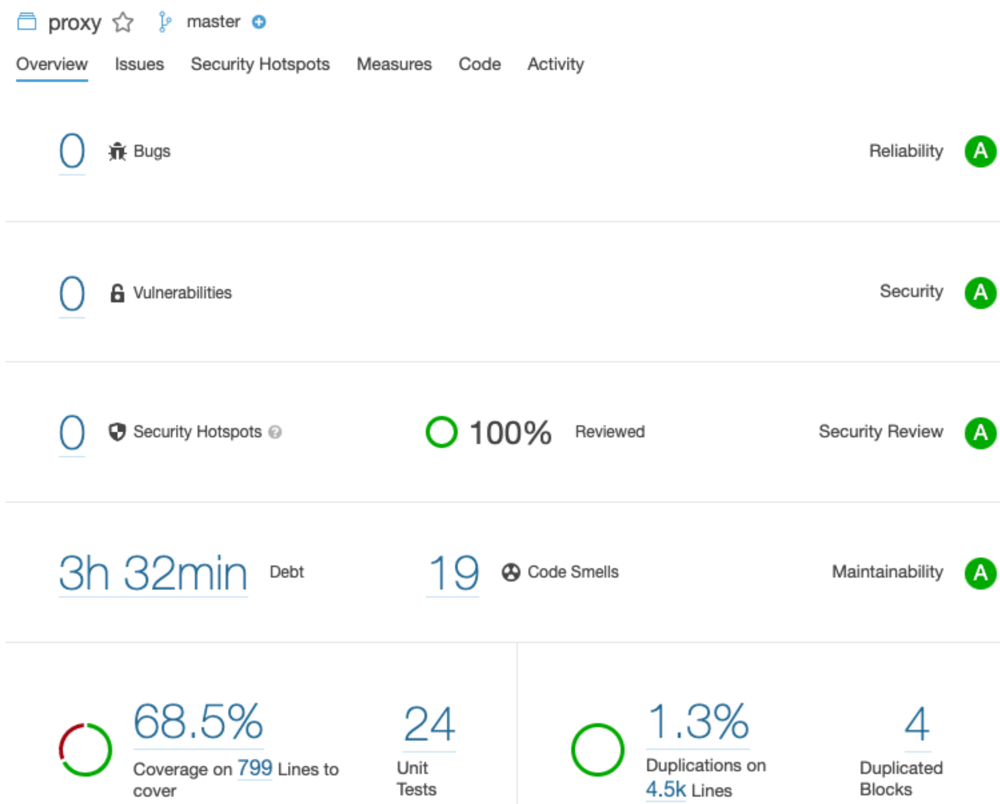

Figure E.5: SonarQube code analysis: Proxy micro-service

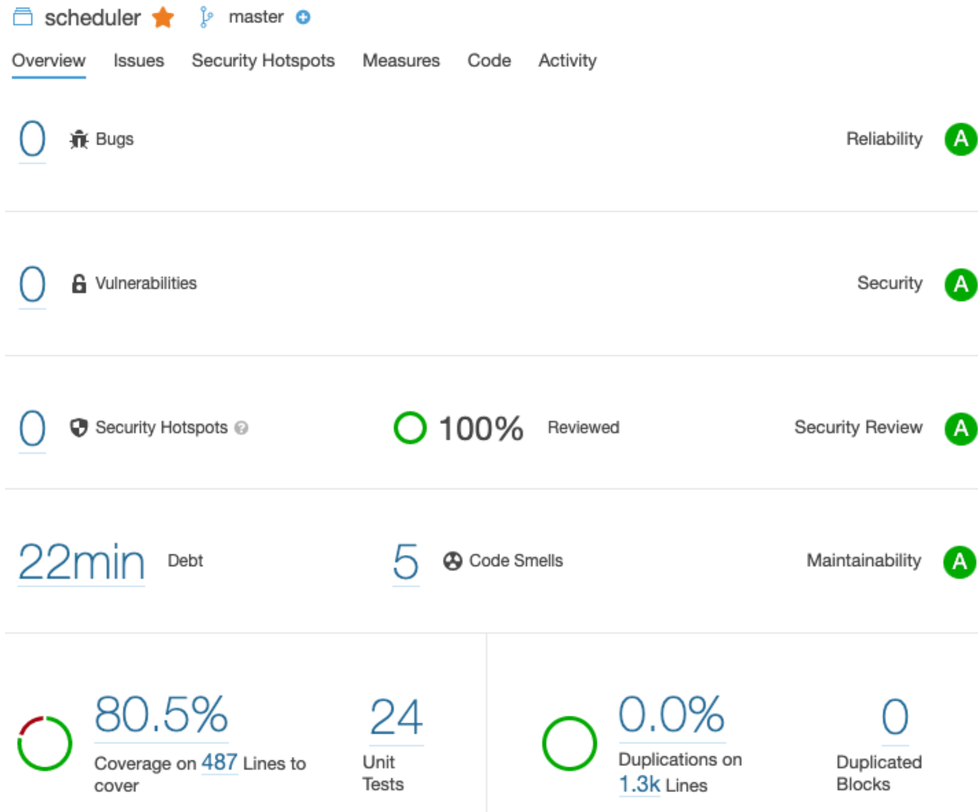

Figure E.6: SonarQube code analysis: Scheduler micro-service

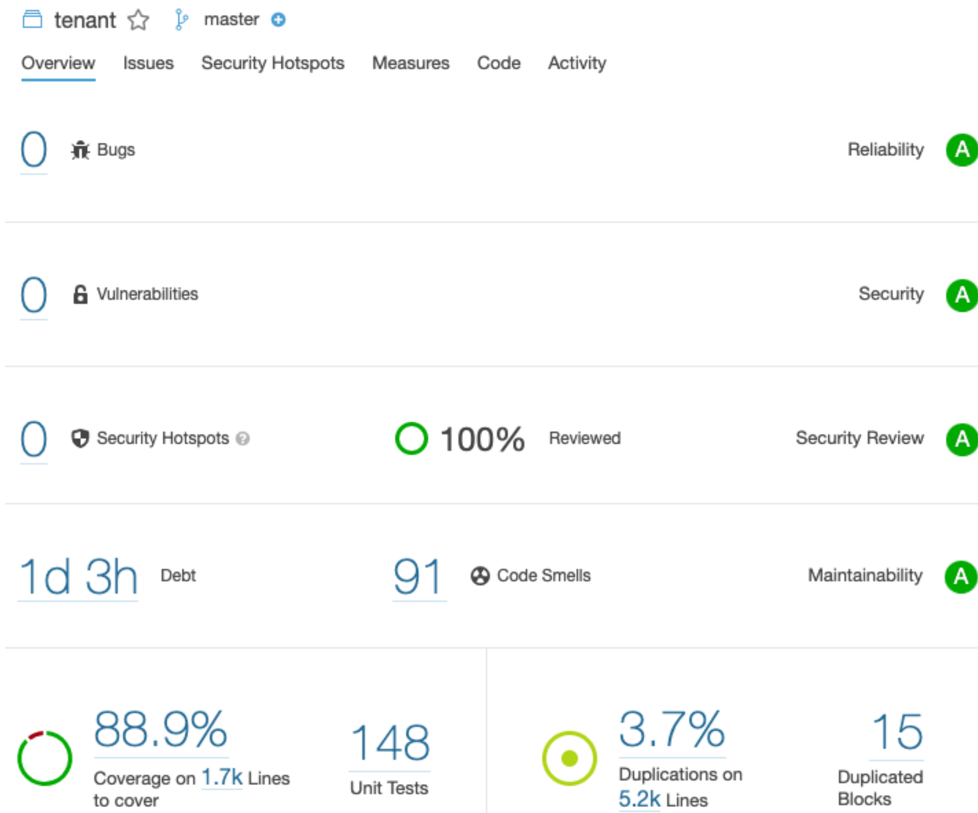

Figure E.7: SonarQube code analysis: Tenant micro-service

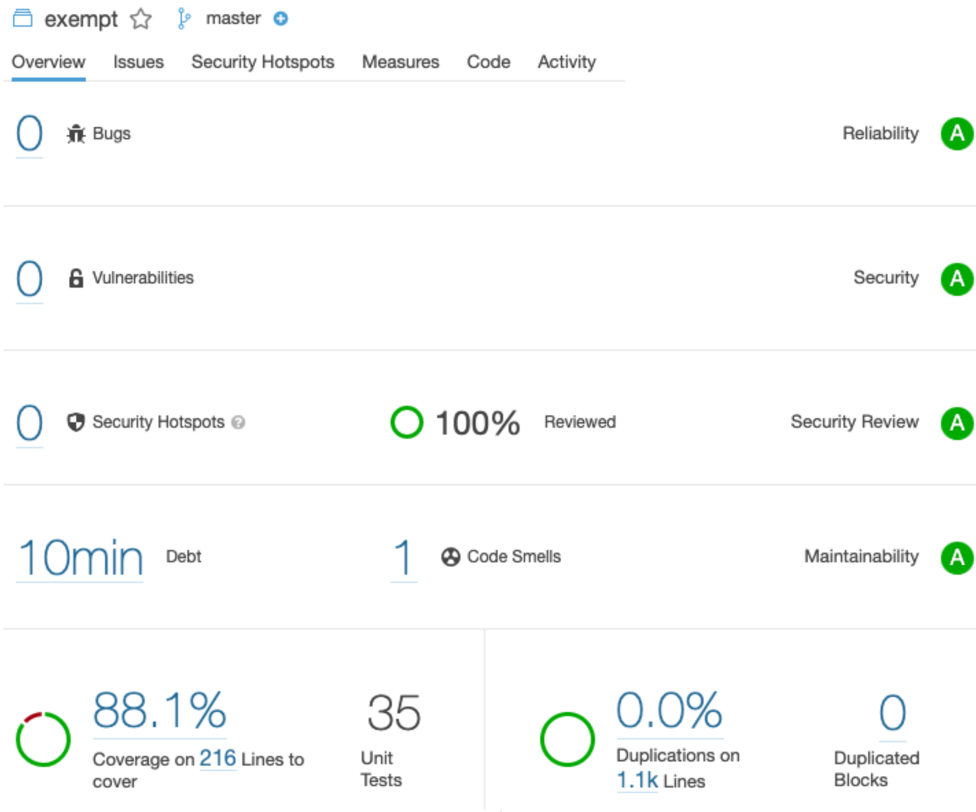

Figure E.8: SonarQube code analysis: Exempt micro-service

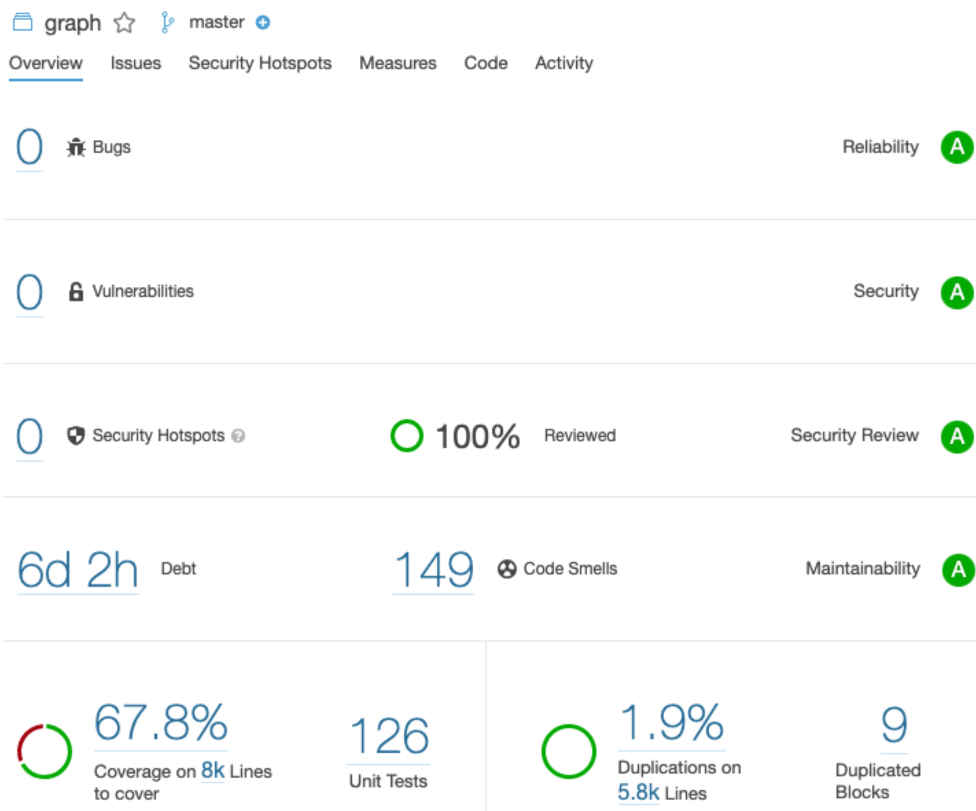

Figure E.9: SonarQube code analysis: GraphQL component