Topic:

Microservices for a Carrying Hailing Service System

Management of Cargo Transports in Real-Time

Submitted by:
César Nicolás Forero Velasco
cn.forero@alumnos.upm.es

Advisor: Jaime Ramirez Rodríguez
Madrid, July 10, 2020
European master’s in software engineering
Title: Microservices for a Carrying Hailing Service System: Management of Cargo Transports in Real-Time
Thesis: EMSE-2020-6
Date: July 10, 2020
Author: César Nicolás Forero Velasco
Supervisor: Jaime Ramírez Rodríguez
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid
Abstract

The work of this master thesis is encompassed in the project of a platform that serves to put in contact through mobile apps customers who need a cargo transport service with drivers available to provide that service. In this sense, this system should be capable of finding hauliers or drivers who fit to the customers’ needs for the freight that they want to transport and close to their location. And for the drivers, this system should be able to show them the customer freight details so that they can decide whether they want to accept to perform the ride or not. Meanwhile, the company should be able to track the rides in all different stages until the freights are delivered, showing up-to-date information in real time and elaborating reports that add value to the business.

The server application will use a Microservice Architecture to break the system functionalities into independent units providing services that comply with the system requirements. This allows us to decouple the business logic into components that can be maintained independently, giving huge flexibility and portability. In the case of availability, microservice resilience is a key issue, because if the failover, replication and monitoring are managed correctly, the microservice will have a high availability. So, the deployment environment is essential to support these features along with scalability. To get a high scalability, resource allocation is very important, being the load balancer the component in charge of this task, splitting microservice workload among system resources, and being responsible for managing infrastructure resources. Performance is very dependent on scalability, supposing that the microservices are well implemented, since a good allocation of resources is essential for not degrading the service when traffic pikes are received.

The server application consists of two back-ends, the Realtime and Administration back-ends. This master thesis focuses on the development of the Realtime back-end.

In the development and deployment of the Realtime back-end, we have used different technologies such as Azure Kubernetes, Docker, NodeJS, MongoDB, etc. The final result is promising, because we built an elastic solution that can be easily configured based on the system requirements without extra coding effort. The proposed architecture and deployment can easily be used for a low or high workload after user demand, making scalability one of the greatest benefits for a company that is starting with the platform and in the future may grow without too much effort for adapting the current software solution.
Contents

1 Introduction 1

2 Background 3

  2.1 Microservices ........................................ 3
    2.1.1 Overview ........................................ 3
    2.1.2 Definition ........................................ 4
    2.1.3 Benefits .......................................... 4
      2.1.3.1 Scalability ................................... 4
      2.1.3.2 Resilience ................................... 5
      2.1.3.3 Maintainability ............................... 5
      2.1.3.4 Technology Heterogeneity ..................... 5
      2.1.3.5 Optimizing ..................................... 6
    2.1.4 Challenges ........................................ 6
      2.1.4.1 Monitoring .................................... 6
      2.1.4.2 Testing ....................................... 6
    2.1.5 Integration ....................................... 7
      2.1.5.1 Microservice Collaboration ..................... 7
      2.1.5.2 Service Composition ............................ 7
      2.1.5.3 Database per Service Pattern ................ 8
    2.1.6 Deployment ........................................ 9
      2.1.6.1 Continuous Integration (CI) .................... 9
      2.1.6.2 Continuous Delivery (CD) ..................... 9
  2.2 Web Server with NodeJS(Express) .......................... 10
    2.2.1 Express .......................................... 10
    2.2.2 SocketIO ........................................ 11
  2.3 MongoDB .................................................. 11
    2.3.1 High Performance ................................... 12
## 3 Software Requirements Specification

3.1 Introduction ....................................... 25
3.2 Purpose ........................................ 25
3.3 Scope ........................................ 25
3.4 Definitions ...................................... 26
3.5 General Description ................................ 26
  3.5.1 Product Characteristics ......................... 26
  3.5.2 Product Functions ................................ 27
3.6 Functional Requirements ............................ 30
  3.6.1 Req (1) New customer ............................ 30
  3.6.2 Req (2) New Customer Account Confirmation ...... 31
  3.6.3 Req (3) SMS Confirmation of Phone number ......... 31
  3.6.4 Req (4) Save and select Payment Method ........... 31
  3.6.5 Req (5) Driver Request .......................... 31
  3.6.6 Req (6) Bank Account Information .................. 32
  3.6.7 Req (7) Driver Images ........................... 32
  3.6.8 Req (8) Application Email Notifications ............ 32
  3.6.9 Req (9) Account Validation ....................... 32
  3.6.10 Req (10) Account Approval ...................... 32
  3.6.11 Req (11) Creation of Driver User ................ 33
  3.6.12 Req (12) Authentication of the Driver Mobile Application .... 33
  3.6.13 Req (13) Vehicle Administration per Driver ....... 33
  3.6.14 Req (14) Request a Transportation Service ........ 33
  3.6.15 Req (15) Route Calculation and Value of the Service .... 34
  3.6.16 Req (16) Location of Transport Units ............. 34
  3.6.17 Req (17) Display of Transport Requests and Approval or Rejection Process .......... 34
  3.6.18 Req (18) Display Location and Driver Information ...... 34
  3.6.19 Req (19) Enable and disable Driver Availability .... 34
3.6.20 Req (20) Start of the Ride .......................... 35
3.6.21 Req (21) Ending the Ride .......................... 35
3.6.22 Req (22) Payment Execution .......................... 35
3.6.23 Req (23) Payment Recalculation .......................... 35
3.6.24 Req (24) Payment Monitoring .......................... 36
3.6.25 Req (25) Cancellation of Cargo Transport Service .......................... 36
3.6.26 Req (26) Qualification of the Provided Service .......................... 36
3.6.27 Req (27) Ride History .......................... 36
3.6.28 Req (28) Change Payment Method .......................... 37
3.6.29 Req (29) User Administration .......................... 37
3.6.30 Req (30) Administration Parameters .......................... 37
3.6.31 Req (31) Report Transaction .......................... 38
3.6.32 Req (32) Report of Transactions with Error .......................... 38
3.6.33 Req (33) Ranking of Drivers .......................... 38
3.6.34 Req (34) Report of Transactions pending reversal .......................... 38
3.6.35 Req (35) Report of Drivers .......................... 38
3.6.36 Req (36) Report of Rides completed by Driver .......................... 39
3.6.37 Req (37) Report of Rejected Drivers .......................... 39
3.6.38 Req (38) Report of Clients .......................... 39
3.6.40 Req (40) Dashboard .......................... 39

3.7 Non-functional Requirements .......................... 40
3.7.1 Req (41) Testability .......................... 40
3.7.2 Req (42) Authentication and Authorization .......................... 40
3.7.3 Req (43) Availability of the server .......................... 40
3.7.4 Req (44) Availability of the mobile apps .......................... 40
3.7.5 Req (45) Scalability and Performance .......................... 41
3.7.6 Req (46) Availability of services .......................... 41
3.7.7 Req (47) Payment Security .......................... 41
4 Development

4.1 Context of the Car Pooling Information System

4.1.1 Mobile Application for the Driver

4.1.1.1 HTTPS

4.1.1.2 Web Sockets

4.1.2 Web Application

4.1.3 Mobile Application for the Customer

4.1.3.1 HTTPS

4.1.3.2 Web Sockets

4.1.4 Mobile Operator

4.1.5 Notification Service

4.1.6 Payment Gateway

4.2 High Level Description of Main Operations

4.2.1 Ride Request Process

4.2.2 Ride Stages Process

4.3 High Level Modular Decomposition

4.3.1 Api Gateway

4.3.2 Real-time

4.3.3 Administration

4.3.4 Inter-service Communication

4.4 Deployment of the Real-time Microservices

4.4.1 Connection

4.4.1.1 Kubernetes Configuration

4.4.1.2 Database Configuration

4.4.2 Driver Location Microservice

4.4.2.1 Kubernetes Configuration

4.4.2.2 Database Configuration

4.4.3 Request Ride Microservice

4.4.3.1 Kubernetes Configuration
4.4.3.2 Database Configuration ........................................... 59
4.4.4 Ride Microservice .................................................. 59
  4.4.4.1 Kubernetes Configuration ...................................... 59
  4.4.4.2 Database Configuration ........................................ 60
4.5 Design of the Realtime Microservices .................................. 61
  4.5.1 Connection Microservice ........................................ 61
    4.5.1.1 Data Model ..................................................... 61
    4.5.1.2 Socket Events ................................................... 62
    4.5.1.3 Create Connection .............................................. 63
    4.5.1.4 Emit Event ....................................................... 64
    4.5.1.5 Send Notification ................................................ 65
  4.5.2 Driver Location Microservice .................................... 66
    4.5.2.1 Data Model ..................................................... 66
    4.5.2.2 Fetch Nearby Drivers ......................................... 67
    4.5.2.3 Get Driver Status .............................................. 68
    4.5.2.4 Is Driver Active ................................................ 69
    4.5.2.5 Update Driver Status ......................................... 70
    4.5.2.6 Disable Driver ................................................. 71
    4.5.2.7 Update Driver Location ....................................... 72
  4.5.3 Request Ride Microservice ....................................... 73
    4.5.3.1 Data Model ..................................................... 73
    4.5.3.2 Calculate Estimation .......................................... 75
    4.5.3.3 Create Request ................................................ 75
    4.5.3.4 Cancel Request ............................................... 76
    4.5.3.5 Request Exists ................................................ 77
    4.5.3.6 Get Pending Requests ......................................... 78
  4.5.4 Ride Microservice ................................................ 79
    4.5.4.1 Data Model ..................................................... 79
    4.5.4.2 Create Ride .................................................... 82
5 Testing

5.1 Mobile Applications ........................................ 94

5.2 Web Application ............................................ 97

5.2.1 Architecture and Configuration ............................... 98

5.2.2 Kubernetes Deployment .................................... 99

5.2.3 Database ................................................ 102

5.2.4 Test Plan ................................................ 102

5.3 Results .................................................. 108

5.3.1 Number of Pods per Socket Connections ................. 108

5.3.2 Maximum CPU Core Usage per Socket Connections .... 109

5.3.3 Memory Usage per Socket Connections .................. 109

5.3.4 CPU Cores per Connections ............................... 110

6 Conclusion and Future Work ................................. 111

References ...................................................... 114
List of Figures

1  Target Scaling [Newman, 2015] ............................................ 5
2  Microservice Orchestration [Chandel, 2018] .......................... 8
3  Microservice Choreography [Chandel, 2018] .......................... 8
4  Continuous Integration Example [Newman, 2015] ...................... 9
5  Continuous Delivery Example [Newman, 2015] ........................ 9
6  Mongodb Cluster Tier ......................................................... 15
7  Cloud services [Nugara, 2017] ............................................. 16
8  Container Cluster Architecture [Pahl et al., 2017] ...................... 17
9  Containers vs VM [Docker, 2020b] ...................................... 18
10 Kubernetes cluster [Kubernetes, 2020a] .................................. 20
11 High level modular decomposition ........................................ 42
12 Request Ride Process ......................................................... 48
13 Ride Stages Process ......................................................... 49
14 Interservice Communication ................................................ 50
15 High level modular decomposition ........................................ 54
16 Connection Data Model ....................................................... 61
17 Create Connection Endpoint ............................................... 64
18 Emit Event ................................................................. 65
19 Emit Event ................................................................. 65
20 Connection Data Model ....................................................... 67
21 Fetch Nearby Drivers Sequence Diagram ................................. 68
22 Get driver status Sequence Diagram ...................................... 69
23 Is Driver active Sequence Diagram ....................................... 70
24 Update driver status Sequence Diagram .................................. 71
25 Disable driver Sequence Diagram ......................................... 72
26 Update driver location Sequence Diagram .............................. 73
27 Request Ride Model ........................................................ 74
<table>
<thead>
<tr>
<th>Page</th>
<th>Title</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Calculate Estimation Sequence Diagram</td>
<td>75</td>
</tr>
<tr>
<td>29</td>
<td>Create Request Sequence Diagram</td>
<td>76</td>
</tr>
<tr>
<td>30</td>
<td>Cancel Request Sequence Diagram</td>
<td>77</td>
</tr>
<tr>
<td>31</td>
<td>Request exists Sequence Diagram</td>
<td>78</td>
</tr>
<tr>
<td>32</td>
<td>Get pending Requests Sequence Diagram</td>
<td>79</td>
</tr>
<tr>
<td>33</td>
<td>Ride Model</td>
<td>81</td>
</tr>
<tr>
<td>34</td>
<td>Create ride Sequence Diagram</td>
<td>83</td>
</tr>
<tr>
<td>35</td>
<td>Alert near customer Sequence Diagram</td>
<td>84</td>
</tr>
<tr>
<td>36</td>
<td>Rate ride Sequence Diagram</td>
<td>85</td>
</tr>
<tr>
<td>37</td>
<td>Verify payment Sequence Diagram</td>
<td>86</td>
</tr>
<tr>
<td>38</td>
<td>Card payment Result Sequence Diagram</td>
<td>87</td>
</tr>
<tr>
<td>39</td>
<td>Finish ride Sequence Diagram</td>
<td>88</td>
</tr>
<tr>
<td>40</td>
<td>Cancel ride by Driver Sequence Diagram</td>
<td>89</td>
</tr>
<tr>
<td>41</td>
<td>Cancel ride by Customer Sequence Diagram</td>
<td>90</td>
</tr>
<tr>
<td>42</td>
<td>Start ride Sequence Diagram</td>
<td>91</td>
</tr>
<tr>
<td>43</td>
<td>Resume ride driver Sequence Diagram</td>
<td>92</td>
</tr>
<tr>
<td>44</td>
<td>Resume ride customer Sequence Diagram</td>
<td>92</td>
</tr>
<tr>
<td>45</td>
<td>Ride Report Sequence Diagram</td>
<td>93</td>
</tr>
<tr>
<td>46</td>
<td>Customer Mobile Application: Ride Configuration</td>
<td>94</td>
</tr>
<tr>
<td>47</td>
<td>Customer Mobile Application: Ride Map</td>
<td>95</td>
</tr>
<tr>
<td>48</td>
<td>Driver Mobile Application: Available for rides</td>
<td>96</td>
</tr>
<tr>
<td>49</td>
<td>Driver Mobile Application: Receive ride requests</td>
<td>97</td>
</tr>
<tr>
<td>50</td>
<td>React Web application</td>
<td>98</td>
</tr>
<tr>
<td>51</td>
<td>React Web application</td>
<td>98</td>
</tr>
<tr>
<td>52</td>
<td>Azure Kubernetes Services</td>
<td>100</td>
</tr>
<tr>
<td>53</td>
<td>Kubernetes Dashboard Nodes</td>
<td>101</td>
</tr>
<tr>
<td>54</td>
<td>Kubernetes Dashboard Overview</td>
<td>101</td>
</tr>
<tr>
<td>55</td>
<td>Kubernetes Dashboard Overview</td>
<td>102</td>
</tr>
<tr>
<td>56</td>
<td>20 Active connections Front-end</td>
<td>103</td>
</tr>
<tr>
<td>Page</td>
<td>Section Description</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>57</td>
<td>20 Active connections Connection Microservice</td>
<td>103</td>
</tr>
<tr>
<td>58</td>
<td>20 Active connections Driver Location Microservice</td>
<td>104</td>
</tr>
<tr>
<td>59</td>
<td>500 Active connections Front-end</td>
<td>104</td>
</tr>
<tr>
<td>60</td>
<td>500 Active connections Connection Microservice</td>
<td>105</td>
</tr>
<tr>
<td>61</td>
<td>500 Active connections Driver Location Microservice</td>
<td>106</td>
</tr>
<tr>
<td>62</td>
<td>2000 Active connections Front-end computer 1</td>
<td>106</td>
</tr>
<tr>
<td>63</td>
<td>2000 Active connections Front-end computer 2</td>
<td>107</td>
</tr>
<tr>
<td>64</td>
<td>2000 Active connections Connection Microservice</td>
<td>107</td>
</tr>
<tr>
<td>65</td>
<td>2000 Active connections Driver Microservice</td>
<td>108</td>
</tr>
<tr>
<td>66</td>
<td>Number of Pods per Socket connections</td>
<td>109</td>
</tr>
<tr>
<td>67</td>
<td>Maximum CPU Cores usage per connections</td>
<td>109</td>
</tr>
<tr>
<td>68</td>
<td>Memory usage per Socket Connections</td>
<td>110</td>
</tr>
<tr>
<td>69</td>
<td>CPU Cores per Connections</td>
<td>110</td>
</tr>
</tbody>
</table>
1 Introduction

Freight transport is defined as the act of transporting goods and cargo by land or air. The majority of the companies use land vehicles, like trucks, driven by hauliers to make the rides and carry the customer goods. It is very common for companies to use in-house systems for reaching their customer and manage their operations. However, there are many hauliers which work independently or for small companies whose main channel to get customers is by a word-of-mouth marketing. Therefore, this cause that hauliers keep offering their services informally using street advertisement and social media mostly as a way to get customers. As a result of this, customers will have to look for hauliers with people who have used the service or by searching in the advertisements.

A freight transport company told us about its visibility problem and the need for a system to not only find customers but also to monitor its operation so that its business value is increased and therefore its profit. In this sense, this system should be capable of finding hauliers or drivers who fit to the customers’ needs for the freight that they want to transport and close to their location. And for the drivers, this system should be able to show them the customer freight details so that they can decide whether they want to accept to perform the ride or not. Meanwhile, the company should be able to track the rides in all different stages until the freights are delivered, showing up-to-date information in real time and elaborating reports that add value to the business.

So, this system will become an additional contact channel between the hauliers or drivers who work for the company, customers who are interested in the transport service and administrators who can monitor the operation.

Thus, the whole solution will be formed by two mobile applications (one for the customers and another for the drivers), a web based application, and a server application constituted by two back-ends, one for administration purposes and another for real time operations. Therefore, the mobile and web applications will use both back-ends that will be exposed through an API Gateway.

The Administration back-end will provide services for managing the users, rides, payments and system configuration. For the rides it will generate historical reports in their different stages. On the other side, the Real-time back-end will be in charge of ride assignment and monitoring, user’s location tracking and push notifications.

This document will be focused only on the design of the Real-time back-end, assuming that the Administration back-end is implemented, even though an overall view of the whole system will be depicted.

The server application will use a Microservice Architecture to break the system functionalities into independent units providing services that comply with the system requirements. This allows us to decouple the business logic into components that can be maintained independently, giving huge flexibility and portability. Also, the system needs to provide up-to-date data about the operation, so the system is committed to some real time constraints, therefore performance will be a primary quality attribute. Additionally, the system will need to be able to handle multiple active users in the platform without downtime, so availability and scalability will be a must. To achieve these quality attribute requirements, operations will be handled independently for just the components that are
receiving the requests, being more cost-effective than a monolithic system.

The microservices will be deployed using an Orchestrator system called Kubernetes. It uses containerization to create a cluster with all their microservices and their different instances. Each microservice is configured independently, supporting to run and schedule containers for automating operational tasks. Each microservice container will be delivered as a Docker image to Kubernetes with a specified configuration. The Kubernetes runtime will be deployed in the Azure cloud computing System, with a Kubernetes services that they provide, AKS (Azure Kubernetes Service). Each microservice has its own database, which it is private, so only the current microservice will have access. The database chosen for the project is MongoDB in the case of the Real-time back-end and SQL for the administration back-end.

One of the biggest Real-time back-end concerns is to allow multiple users to connect to the system for exchanging data and make it available to other users without degrading the service, especially at high traffic situations. This is an scalability issue that needs to be solved by applying mechanisms to keep multiple instances of microservices, which will be analyzed throughout the document. A second challenging task is that the system needs to be able to communicate with users after some events are dispatched, which cannot be normally achieved by using Http for REST Services. Therefore, the system will have to implement Websocket communication in the microservice architecture allowing users to receive/send data on demand following a publish and subscriber design pattern.

In the following sections, first we will introduce the background for the different technologies and concepts used in this project. Followed by a description of the different software requirements that the system will implement. Then the development section will include the following contents: the system context, the description of main operations, the high level modular description of the system, as well as the deployment and the design of the Real-time microservices. The next section will address the testing of the Real-time microservices, showing how the test plan was conceived and the corresponding results. Finally, the last section will explain the conclusions and the future work for the system.
2 Background

The current proposal is a transportation platform to provide a connection between the drivers who offer the cargo services and customers who are willing to pay for the transportation of their goods. These user interfaces are composed by two mobile applications, each for the driver and the customer, respectively, and a web application as a business tool to manage operations.

These interfaces are connected with a Back-end based on a microservices oriented architecture, implemented in NodeJS and deployed in Azure cloud by using Kubernetes. Additionally, this Back-end exposes a REST API and socket services to process the user’s real-time workload and stores this data into a Non-SQL database: MongoDB. In the following sections the techniques, principles and technologies used in this system will be explained in detail:

2.1 Microservices

2.1.1 Overview

Due to the real-time nature of the platform and the location based system of the proposal, the server needs to handle multiple connections with the users to allow them to share their location updates periodically with the Back-end, which needs to keep processing them to provide users accurate data during the operations.

In case of a monolithic architecture, different controllers can be implemented to handle all the workload depending of the type of service, but since all of them are part of the same system and the same database, this is not a good approach for scalability. The main concern is that real-time systems cannot be built with regular API REST services only, since these are stateless and the client would need to start the handshake process in every interaction. So these systems need a communication mechanism to remain connected to the Back-end so that the data flow can go in both ways, having a permanent connection established and therefore a strong impact in resource usage.

Moreover a microservices architecture can provide a cost and time effective solution for a platform, where each microservice is an independent module with its own database. We can take advantage of this approach when the platform needs to scale, since not all microservices will receive the same traffic, therefore only the more demanded modules will have to scale dynamically.

There are two different main patterns/approaches to build a microservice architecture in order to define the interaction between microservices, the choreography and orchestration. In a choreography microservice architecture, there are a set of rules to establish the interaction between endpoints directly by exchanging messages among them. On the other hand, in the case of the orchestration there is a third party involved: the orchestrator, which coordinates the interaction among different services, centralizing communication.

In addition to the microservice patterns, the communication between the microservices and orchestrator (in the case of the orchestration) can be done synchronously or asyn-
chronously, depending on the technology stacks available to build it. For synchronous communication, there is a need of a connection-oriented protocol, the most common and simplest is http. Due to the service endpoints are already exposed to the orchestrator, the communication can be set between microservices using the same endpoints internally to exchange data. On the opposite, the asynchronous is more complex to design but gives better performance, it uses a middleware for communication, the most common is a message broker, allowing to exchange messages between microservices.

2.1.2 Definition

Microservices are an architectural design for software development bringing a high level isolation between autonomous components but still can communicate one another. Each microservice encapsulates different business functionality, allowing large scale applications to enhance flexibility, maintenance and quality of service (QoS) [Hassan et al., 2017].

2.1.3 Benefits

Following a microservice architecture comes with several advantages from different perspectives in the development life cycle. The followings advantages are the main ones:

2.1.3.1 Scalability

It allows to scale on demand by replicating services with specific functionality, instead of the whole system as in a monolithic solution, which can be cost saving in hardware [Newman, 2015] as shown in figure 1. For this reason it is very important to establish the boundaries of each microservice, to get the most out of scalability. This will allow the system to avoid downgrading performance on high peaks, if it is well implemented and countermeasures are designed to identify when the microservice should scale. The way microservices should scale depends strictly on the architecture designed for internal communication and the technology used for deployment. This can be challenging, since architecture should be aware of managing tasks as networking, security and transactions. So a widely spread technology used for these purposes is containerization, which will be discussed further in next sections. But as an introduction, we can say that it is a virtualization method to deploy distributed systems allowing to build clusters and manage load balancing among all containers [Hasselbring and Steinacker, 2017].
2.1.3.2 Resilience

For a monolithic application system health is binary, in the sense that there is a single point of failure for the core functionality of the system, if one of its components fails it will stop the whole system and its functionalities, affecting the overall availability. The classic tactic to mitigate it is to deploy the application on multiple machines having redundancy, so that the system will reduce the chance of failure. But still if the failure propagates in case of a software defect, it will crash the whole system being completely unavailable for all their services it provides. With a microservices system in case one of them fails, the rest of the services will continue working, since the failure won’t cascade [Newman, 2015], but a more complex design for recovering has to be done in order to provide an architecture with safety measures to degrade functionality in a control way.

2.1.3.3 Maintainability

As a monolithic application grows, so it does the code base making difficult for developers to keep track of the different functionalities that the application provides, which can impact the productivity in the team, despite good practices are followed. On the other hand, with microservices where each of them is stored in its own repository and it is treated as an independent project, therefore being granular and more focused on specific functionalities, these changes are easier to maintain for large application among the team, being able to respond quicker to customer needs, especially in agile teams [Viggiato et al., 2018]

2.1.3.4 Technology Heterogeneity

Depending on the functional and non-functional requirements that the microservice will perform, the most appropriate stack of technologies can be selected without worrying for the rest of the application. Isolation also comes in handy for technology, allowing to select the best to fit requirements of each microservice. In addition due to the distribution nature of the system, new technologies can be tested easier reducing the risk to break the whole system for side effects that can have when using a bunch of technologies as can happen in monolithic approach. [Newman, 2015].
2.1.3.5 Optimizing

There is always a risk when it comes to upgrade into a new technology, especially on big old systems. Re implementing a solution is always needed as time pass by, since new technologies can offer better performance or enhancements, but it is not easy to manage for monolithic having tons of LOC (Lines of Code) and integration is complex. For microservices these tasks are less complex to perform, due to the isolation level and size of each microservice [Newman, 2015].

2.1.4 Challenges

Microservices add high end benefits to DevOps and Continuous Deployment for production, once everything is correctly setup. But since microservices systems are distributed the deployments will be independent pipelines, adding extra challenges to monitor and test the whole system [Heinrich et al., 2017], as it is explained next:

2.1.4.1 Monitoring

As it was mentioned earlier microservices are commonly deployed using containers, one for each microservice. So, monitoring these containers implies to capture resource usage, establish tolerance intervals to detect anomalies from the microservice inside container and those which are interrelated. However these microservices can be implemented using different stack of technologies, which adds some complexity to the tools that need to capture monitored data. This data usually is needed for the microservice to auto-scale, as can be necessary for platforms which automate deployment as Kubernetes, which will be introduced later. On top of that, there is a need to establish whether a microservice is behaving as normal or it is having difficulties in its operation at some point, for anomaly detection and executing contingency plans [Heinrich et al., 2017].

2.1.4.2 Testing

To keep a good quality of the software, testing is a mandatory task that needs to be done during development life cycle. But there are different testing levels to apply (Unit, Integration, System, Acceptance testing), being more challenging the last levels since microservices are deployed on a regular basis for CD (Continual Deployment), and quality needs to be ensured on all releases. But industry has very often complemented or even replaced full-scale integration testing with monitoring techniques, due to their volatile nature [Heinrich et al., 2017]. So in case of a failure, new releases from the microservice will be re-deployed in production, as soon as they are identified from monitoring outputs. This technique is applied to improve regression testing, where monitoring performance data is used as an input to speed up CD operations. In this case historical metrics are compared with the ones from new builds to track any introduced defects. But still there is a need to semi-automate and create a strategy to select the right testing technique to perform after each commit [Heinrich et al., 2017].
2.1.5 Integration

It is very important to rely on technology that keep microservices with strong cohesion and loose coupling, so that performing changes won’t have a strong impact in the whole system as it was defined in the maintainability benefit subsection. Next the most important technologies and methodologies to integrate microservices will be explained:

2.1.5.1 Microservice Collaboration

Microservices can collaborate using either synchronous or asynchronous communication, depending on the type of data that is being transmitted and the requirements.

1. For synchronous communication the operation is blocked until the party in charge completes it and sends the response, this is normally the case of API REST using http as message pattern [Wasson, 2019].

2. For asynchronous communication the caller does not have to wait for the operation to complete. This is very useful for transmitting large data sets or when network is not stable so that there is high latency. There can be two types: Asynchronous I/O or protocol, the first means that the operation is not blocked while the I/O completes and for the second the caller does not wait for the response. The most common design pattern based on this type of communication is event-driven, being highly decoupled but can be complex to implement [Wasson, 2019].

2.1.5.2 Service Composition

The previous section stated that microservices can collaborated using any of the two communication methods explained above, but as project grows there is the need of implementing a Service composition to use this collaboration and integrate microservices [Neha Singhal, 2019]. Basically there are two methods for service composition: Orchestration and Choreography, which will be explain next:

1. **Orchestration:** For this method there is a single centralized coordinator and single point of collaboration: the orchestrator as seen in figure 2. It is in charge of controlling and monitoring the microservices and its interactions. It is very common that this style uses synchronous communication [Chandel, 2018] in order to execute sequential calls to microservices.
2. **Choreography**: On the opposite, this style uses all microservices to exchange messages between them, using a decentralized approach for service composition as seen in figure 3. Therefore asynchronous communication fits more to this approach, so each microservice can publish and subscribe to events when exchanging messages, behaving like a State Machine and reacting to those events [Chandel, 2018].

![Figure 3: Microservice Choreography [Chandel, 2018]](image)

### 2.1.5.3 Database per Service Pattern

As was explained before, each microservice should be independent from the others, and this includes its database and schema. In this way each microservice should access to its private database, and no others, having its own API for this purpose [Kasun Indrasiri, 2019]. With a decentralized data management, microservices will be decoupled and they can have their own database implementation that better fits to their needs (SQL or NoSQL).

To implement transaction across multiple databases, there exists the *Saga Pattern* which consists in a sequence of local transactions built to handle distributed transactions. This works regardless the technology used to each microservice database, since the saga is a sequence of local transaction that results in triggering next transaction until it completes. In case of one fails saga will execute a series of compensating transaction for undoing changes [Chellammal Surianarayanan, 2019]. Saga pattern can be implemented to any of the two service compositions (Choreography and Orchestration).
2.1.6 Deployment

Due to the independence nature of microservices, deployment stage adds extra complexity compared with monolithic systems. Therefore, it is very important to choose technologies that support high automation like virtualization, to reduce host management overhead and configuration. This will be explained further in section 2.5.2 containerization. Within the deployment, strategies such as Continuous Integration (CI) and Continuous Delivery (CD) are crucial, they are part of DevOps strategies (CI/CD) and they can be applied on any software development. Next they will be introduced:

2.1.6.1 Continuous Integration (CI)

It is a set of software engineering practices to keep everything in sync and decrease integration times, especially, when working with microservices where releases are delivered with a high frequency. This keeps integration verified by using automated builds detecting conflicts and delivering new functionalities faster, also keeping traceability across integrations [O’Connor et al., 2017].

![Continuous Integration Example](Newman, 2015)

2.1.6.2 Continuous Delivery (CD)

It provides the ability to perform reliable, efficient and rapid releases on demand using pipelines. A pipeline is a production line that performs a series of validation stages prior to the release highly automated [O’Connor et al., 2017]. Pipelines carry out verification tasks (also called artifacts), which provide information about the code quality, like correct compiling and test execution [Newman, 2015]. Therefore CD allows reducing uncertainty in the deployment process, because it is completely automated and since the code can be delivered with a high frequency, it gives diagnosis every time is delivered on any environment, minimizing impact when promoting to production.

![Continuous Delivery Example](Newman, 2015)
2.2 Web Server with NodeJS(Express)

NodeJS is an asynchronous event-driven JavaScript runtime, which is designed to build scalable network applications [NodeJS, 2020]. It is based on Google V8 engine written in C++, implementing ECMAScript, which is a standard specification for JavaScript, and WebAssembly, a binary instruction format for a stack-based virtual machine enabling to run high performance applications on web pages [V8, 2020]. Moreover, NodeJS runs JavaScript using single-threaded, non-blocking, concurrent and asynchronous environment provided by V8 and libuv library also written in c++. Each of these features will be explained next:

1. **Single threaded**: Code execution is done sequentially, one operation at a time, without the thread being interrupted while executing. This is because JavaScript runtime (V8) only has a single stack and heap for static and dynamic memory allocation while the program is executed.

2. **Non-blocking**: Since it is single threaded and one operation is executed at a time, it needs extra components to handle asynchronous operations: libuv library provides an event queue and event loop to dispatch those events using a single thread.

3. **Concurrent**: All asynchronous operations are assigned to worker threads for handling asynchronous I/O events, they are in charge of handling concurrency, so that the main execution thread won’t be blocked. The event loop (single threaded) dispatches these events back and forth from the event queue to the worker and when the callback triggers, because it finishes its execution, goes back to the call stack [NodeJS, 2020][V8, 2020].

Previous behavior is very similar to how web browsers execute JavaScript code, with some differences because it uses WebApi instead of libuv library, but still V8 engine is the same in browsers.

NodeJS uses npm package manager through the CLI (Command Line Interface) to incorporate NodeJS community packages of code to any application that can be used right away within NodeJS environment [NPM, 2020]. One widely known package is Express framework, which it is used to build Web Servers relying on Operating System APIs, such as file system and HTTP, provided by NodeJS.

2.2.1 Express

Express Web Framework for NodeJS allows building a Web Server using JavaScript as back-end language. Express comes by default with API modules to build end-points for a REST API, where all microservices can be defined following a MVC pattern. Its main features are:

1. Create handlers for requests with different HTTP verbs at different URL paths (routes).

2. Use template for the view to generate responses with different rendering engines
3. Set common web application settings like connectivity

4. Manage file system resource to handle requests.

5. Build middlewares to define pipelines for any request [Express, 2020]

### 2.2.2 SocketIO

To handle persistent connection between clients and Web server, a complementary library is needed besides Express Framework. One possible option is Socket.IO. This library enables Real-time, bidirectional and event-based communication between the browser and the server, providing implementation in a wide variety of languages (JavaScript, Java, C++, Swift, Dart, Python, .NET) for integrating it. In the case of NodeJS, code can be deployed on both sides in a straightforward manner [SocketIO, 2020] since client and server use the same language. Moreover SocketIO has the next features relying on Engine.IO:

1. **Reliability:** It provides the most reliable connection available between the client and the server over time. So it starts with long-polling and if connection remains stable then it tries to upgrade it to improve performance and network traffic by using Web Sockets. It also can bypass proxies, firewall and load balancers.

2. **Disconnection Detection:** It implements a heartbeat mechanism to test connectivity all the time and detect when client or server are unavailable using timers. SocketIO will try to reconnect by default indefinitely, but the number of attempts can be set up as well.

3. **Multiplexing:** It uses namespaces to create channels that are called *Rooms*, which allows multiple sockets to connect to the same room and exchange messages.

It is important to emphasize that SocketIO runs their own protocol, designed to connect client and servers that implements. Since connection does not always goes over the same technology and some metadata is added to packets, nodes cannot be connected if any of those only supports one of the communication technologies that SocketIO provides.

### 2.3 MongoDB

MongoDB is one of the most common NoSQL database in the industry. It is document oriented which means that it stores semi-structured data in a binary representation called BSON. This encoding is a JSON extension, including additional data types that are very convenient when working with web development technologies. A record in MongoDB is equivalent to a document, which contains fields with values using JSON format. These documents are stored using collections, which in Relational Databases Management Systems (RDMAS) applies the same concept as a table [MongoDB, 2020c]. Next there are some of the main benefits of using MongoDB and how to use them:
2.3.1 High Performance

As well as in RDBMS, MongoDB supports different types of indexes as compound, geospatial, TTL, text search, sparse, partial, unique, among others. They are used as a query strategy for optimizing system performance, if well implemented (they might give overhead to disk space and memory usage when writing). With MongoDB performance can also be enhanced using the capability of embedded data models instead of a relational database approach, reducing I/O activity when performing queries. This is what is called denormalizing data, which can be more efficient to query over one which is normalized. Additionally Mongodb deployments from 3.2 uses WiredTiger as the default storage engine to provide granular concurrency control and native compression to deliver performance and storage efficiency to applications.

2.3.2 High Availability

MongoDB provides availability with a replica set, a group of daemon processes running on the same or different host servers, which keep a copy of the same data set. This allows delivering redundancy to increase data availability, and also fault tolerance against loss of single database server [MongoDB, 2020e]. Moreover, one member of the replica set will act as the primary and the rest as secondary members. In case of failure of the primary replica, then one of the secondary will be promoted automatically as primary. Any writing attempt done while the switch is performed will be retried by the daemon process, ensuring write availability and data consistency. Also the replica sets are used for rolling upgrade when performing hardware or software maintenance, so all replicas will take turns. The number of replicas is configurable by the DBA (Database Administrator) and the more nodes, the higher is the availability [MongoDB, 2018].

And as well as RDBMS use replicas, MongoDB supports multi-document ACID transaction maintaining data integrity across multiple operations, even through MongoDB principle is to use embedded documents, keeping normalization at its lowest so that performance is not affected.

2.3.3 Query Language

MongoDB as RDBMS supports a query language to read and write operation using two modalities: Text Search and Aggregation Pipeline. The first approach uses text indices, to query on specific string content from any field. On the other hand Aggregation Pipeline is more powerful, since it is a framework for data aggregation using data processing pipelines. These queries are formed for a multi-stage pipeline, where each stage implements a data transformation using framework operations (lookups, unwind, group,...) [MongoDB, 2020a].

2.3.4 Horizontal Scalability

One of the greatest benefits of using MongoDB is that it provides as a core functionality horizontal scalability, which in RDBMS can also be used but it requires high maintainability...
ity effort, so it is more common to use a vertical scalability strategy. MongoDB introduces two concepts to keep up with high demand: Replica Sets and Sharding. For read-heavy applications, a strategy can be to increase the size of the replica set and distribute read operations to secondary members. Whereas for write-heavy applications, sharding strategy can distribute the load among different mongod daemon instances. Sharding enable us to distribute data across multiple nodes to support large data sets and high throughput operations which may exhaust resources from a single server\cite{MongoDB, 2020g}. This is accomplished with a Sharded cluster, made up of multiple replica sets, a query router and config servers. So, as adding new nodes to the cluster, MongoDB automatically will re-balance the data accordingly, without manual intervention.

Sharding does not require to change application level queries, since this is done by the query router (called *mongos*) which dispatches to appropriate shards using a shard key or broadcasting among shards. The shard key is very important, because it is used to distribute the collection’s documents among the cluster’s shards. Furthermore, a shard key is assigned for a single or compound indexes from a collection, and it must meet the next properties \cite{MongoDB, 2020f}:

1. High cardinality: Meaning that the shard key must have as many distinct values as possibles.
2. Low frequency: The shard key values are evenly distributed among shards.
3. Query Isolation: Queries should be targeting to specific shards to maximize scalability
4. Insert Scaling: Non-monotonically changing in value as auto incremental keys.

For coming up with these shard keys, MongoDB supports two different strategies:

1. Range-based sharding: applying this strategy will split the collection’s data into chunks, by grouping documents with related values in the same shard. This is a very common strategy when the previous properties are met as choosing the indexes for the shard key.
2. Hash-based sharding: on the opposite, this is used when the indexes do not comply all the properties. So the selected shard keys will create a hash that guarantees values close to uniform distribution.

### 2.3.5 GeoJSON

MongoDB allows processing GeoJSON open standard, representing geospatial data in JSON format to process location queries using a GeoJSON object (Point, Line String, Polygon, Multi Point, Multi Line, Multi Polygon, Geometry Collection) and their coordinates. As MongoDB supports this type of data by default, it is handy for our application, whose main functionality involves to process geospatial data.
### 2.3.6 Mongoose ODM

To map objects with data models, and perform database operations an ODM (Object Data Mapper) is necessary to provide the interaction with MongoDB. One of the most common ODM for MongoDB is Mongoose, which provides schema validation, translation between objects in code and the representation of those objects in MongoDB. The main concepts to work with Mongoose and MongoDB are the following [Mongoose, 2019]:

1. **Collection**: they are analogous to table in RDBMS, grouping MongoDB documents. They exist within a single database.
2. **Document**: represent a record in MongoDB collection, they are stored with a format analogous to JSON objects called BSON, their binary representation.
3. **Schema**: they support to define the structure of a document, their properties are strongly typed. So everytime a write operation is required, it will be validated to comply to with its type.
4. **Model**: Mongoose constructor that uses a defined Schema creating an instance of a document.

Mongoose also provides a variety of functions to perform CRUD operations. For queries they have built in functions \(\text{find, findById, findOne}\) to be applied on Collections and they also support MongoDB’s aggregation pipelines for data aggregations, as was explained before.

### 2.3.7 MongoDB Deployment

For deployment MongoDB brings different tools and services to manage database depending on the automation level and the needs. For Enterprise Advance edition companies can choose between Ops Manager and Cloud Manager. They deliver tools to run and manage complex operational tasks using agents installed on all the cluster nodes. These agents will perform common tasks automatically as deploying a new cluster, upgrading, creating point in time backups, rolling out new indexes, among others. It also includes a dashboard GUI to monitor activity and deliver metrics to allow administrators to be aware of the database environment [MongoDB, 2018].

Although Ops and Cloud Manager offer a complete solution, they are part of the Enterprise edition which is an on-premise deployment. So users will have to manage infrastructure, install and configure all the tools that the package comes with. But there is another option and it is the one used in the current project: MongoDB Atlas, a Database as a Service For MongoDB. This is an official product of MongoDB, which aims to reduce operational overhead and deliver on-demand service whether it is a large or small project.

MongoDB Atlas [MongoDB, 2020d] offers a fully managed and elastic cloud service, enabling users to choose deployment on different cloud platforms as AWS, Azure and GCP. Its main features are:
1. Automated database and infrastructure provisioning, so that resources will grow elastically with the project needs.

2. Security features like end-to-end encryption, access-control, network isolation and auditing.

3. Replication within and across regions.

4. Global clusters to deploy a distributed database.

5. Backups with point in time recovery.


7. Live migration.

8. Coverage on the major cloud platforms in over 50 cloud regions.

9. Versatility to deploy from proof of concept (free) to dev/test/QA environments.

In figure 6, we can see the current different available Cluster Tier configurations for Atlas MongoDB:

![MongoDB Cluster Tier](image-url)
2.4 Azure

It is platform to provide cloud computing services to build, manage and deploy applications using flexible tools and frameworks. Cloud Computing allows to access to a variety of services like servers, storage, networking and software over the cloud [Azure, 2020]. The most common services are [Nugara, 2017]:

1. **Infrastructure as a Service (IaaS):** It is used for renting IT infrastructure (servers, virtual machines, storage, networks and operating systems) accessing those resources over the internet. User is responsible for managing operating system and the application completely, by accessing through an RDC (Remote Desktop Connection) to the Virtual Machines.

2. **Platform as a Service (PaaS):** It gives access to users to the different components to build and host web applications, saving the effort of managing and setting the infrastructure of servers, storage, networks and databases.

3. **Software as a Service (SaaS):** It is used to host, manage and integrate software applications that can be used with only minimal configuration, giving the benefit of scaling up as the business grows.

In Figure 7, the previous services and the user effort are shown:

![Cloud Models](image)

**Figure 7: Cloud services** [Nugara, 2017]

2.5 Deployment with Kubernetes-Azure

2.5.1 Overview

Kubernetes is a platform for managing workloads and services in containers allowing to build a high reliable cluster management tool by automating deployment, scaling, load balancing, logging and monitoring. Containers offer a variety of benefits such as Cloud and OS distribution portability, loosely coupled, distributed, elastic and liberated microservices, resource isolation, continuous development, integration, and deployment; etc.
But running distributed systems with containers adds extra work to ensure availability and configuration management. For that reason Kubernetes comes in handy, by automating different availability attributes, and taking care of scaling and failover for the application. [Kubernetes, 2019]

2.5.2 Containerisation

It is a virtualization technology for Cloud Computing to make applications lightweight, making their deployment and management simpler and resource cost effective. This is done by the construction of containers, application packages running on isolated execution environments with its own operating system [Syed and Fernández, 2018]. They are ready to deploy with all the middlewares and business logic to run applications, which are mounted over individual images [Pahl et al., 2017].

Containerisation facilitates to create clusters of containers, which can be run individually but also they can be interconnected assembling groups. Container cluster architecture consists of several nodes, that belong to an application service group, which share the same image and can be scaled. To integrate with a data layer there are volumes that can be mounted by containers to connect to databases. Finally links are used for inter-container communication, allowing multiple nodes to interact. All of these components will be discussed in detail on the next section. To give a better understanding, in Figure 8 the whole architecture is modeled.

![Figure 8: Container Cluster Architecture](image)

It is very common to use Containerisation for microservices due to their independent deployment, scalability and portability of containers that fit to the microservices needs [Syed and Fernández, 2018]. In the industry the most popular container solution is Docker, which will be explained further in the next section.

2.5.3 Docker

It is a platform based on containers. As in Virtual Machines (VMs), images are handled by Docker apps, made up from file systems layered, allowing to stack them up forming a container [Pahl et al., 2017]. The big difference between Containers and VMs is that
containers virtualized the operating system, instead of hardware as can be seen on Figure 7 [Docker, 2020b].

![Figure 9: Containers vs VM](Docker, 2020b)

The next are the main concepts around Docker:

### 2.5.3.1 Images

Docker Images contains all assets to run application as containers: Code, runtime, dependencies, configuration files and environment variables.

### 2.5.3.2 Containers

Theses images become containers at run-time with Docker Engine, making them standard, so they guarantee portability, being more efficient by reducing server and licensing cost, and secure due to their isolation. Moreover from one image, there can be multiple containers, which is a great benefit for scaling.

### 2.5.3.3 Docker Builder

Docker is able to build images reading some instructions from a Docker File, which contains commands to assemble the image. To execute these instructions there is a CLI with commands to deploy the application defined in Docker containers. To communicate with external dependencies, which are also containers, this can also be specified within a file called Docker Compose. This can start all services from configuration file by using just one command, which will execute all instructions from the file [Docker, 2020a].

### 2.5.4 Kubernetes

It is an open-source platform for managing workloads and services, to run distributed systems resiliently, automating scaling and failover. Kubernetes provides the next features [Kubernetes, 2020e]:

1. **Service discovery and load balancing:** It provides load balancing and distributes network traffic so that application is stable, using DNS and IP addresses.

2. **Automated rollouts and rollbacks:** Containers provisioning can be automated to create or remove containers based on a desired state.

3. **Automatic bin packing:** Kubernetes can provision resources to fit containers needs, therefore it is cost effective.

4. **Self-healing:** Kubernetes uses health checks that can be configured so that containers can be restarted, replaced and killed in case of failures.

5. **Secret and configuration management:** It can handle sensitive information like passwords and keys with an application configuration without exposing it in case of updates and re-deployments.

It is important to understand the next concepts for further reading:

### 2.5.4.1 Nodes

Kubernetes nodes are hosts (VM or physical), which contains services to run applications and handle workloads known as pods. Each node includes container runtime, kubelet and kube-proxy. They are explained next:

1. **kubelet:** It is in charge on making containers to run on specific pods. For this, each Pod has its own specification, to make sure that is running and healthy.

2. **kube-proxy:** Network Proxy present on each node from cluster, maintaining network rules which allow internal and external communication.

3. **Container-runtime:** Software for running containers, Kubernetes supports: Docker, containerd, CRI-O.

### 2.5.4.2 Pods

It is the basic execution unit from a Kubernetes application that can be deployed. Each pod runs as a process from the cluster. It encapsulates application container(s), storage resources, network IP and settings to configure the execution. It is common to deploy one container per pod, but in case of high coupling there can be multiple containers to share resources.

Kubernetes provides a framework to deploy a cluster. A Kubernetes cluster has two main components: Control Panel and Nodes, which are explained next:

### 2.5.4.3 Kubernetes Control Plane

It is in charge of making global decisions about the cluster and respond to cluster events. In Kubernetes every deployment contains a cluster, a set of worker machines, called nodes,
that run containerized applications. Every cluster has at least one worker node. Control Panel can be run on any cluster host as well as its components, starting everything at the same time for simplicity. The next are the components of a Control Plane [Kubernetes, 2020a]:

1. **kube-apiserver**: It exposes Kubernetes API turning into a front end of the whole control Plane that have faculties to scale horizontally.

2. **etcd**: It is a key value store for Kubernetes cluster data.

3. **kube-scheduler**: Component that watches new created pods and node assignment where they will run. Scheduling decisions are considered based on resource requirements, affinity and anti-affinity specifications, data locality, inter-workload interference and deadlines.

4. **kube-controller-manager**: They run controller processes that handle specific tasks. The main ones are: Node Controller, Replication Controller, Endpoint Controller, Service account and Token Controllers.

5. **cloud-controller-manager**: They run controllers from cloud providers evolving independently from Kubernetes'. Some of the next controllers have provided dependencies: Node Controller, Route Controller, Service Controller and Volume Controller.

![Kubernetes Control Plane](image)

Figure 10: Kubernetes cluster [Kubernetes, 2020a]

### 2.5.4.4 Services

A Kubernetes service is an abstraction made of a set of pods running on a network, adding extra functionality to provide tracking among pods outside a cluster. Pods can eventually shut down and reborn, and still be accessed from other components, even though pods ip assignment is dynamic. This is done with a service discovery feature supported by Kubernetes using environment variables and DNS [Kubernetes, 2020c]. There are different types of Kubernetes services:

1. **Cluster-IP**: Default service type to provide only access within the cluster

2. **Node Port**: Exposes service on a static Node Ip and port combination.

3. **Load Balancer**: Exposes service externally using a cloud’s provider load balancer.
4. **External Name**: Exposes services using a map with the content of the external name.

### 2.5.4.5 Controllers

1. **Replica sets**: It is a self healing feature to guarantee availability among a set of pods running at any given time. They are normally used with deployments to deliver orchestration updates, where their replica set is configured within Deployment setup. Therefore the replica will have a set of pods running the same image instance inside the cluster to provide service.

2. **Deployments**: This controller provides declarative updates for Pods and replica sets, guarantying zero downtime when rolling updates. As it is declarative, it is an extension of the replica set definition described in a desired state deployment. This behaviour is established using a new abstraction of Pods called Workloads, where scheduling, scaling and upgrading pods are configured though a set of rules.

### 2.5.4.6 Volumes

Containers in kubernetes by default are stateless, so everytime it is restarted their state is clean. This is the purpose of volumes to preserve the data across restarts on containers that belong to the same pod. Kubernetes Volumes are directories to store data which it is available to containers through the file system. Their use is very common in database run as microservices.

There is a volume API to remove dependency over a Pod, which is the PersistentVolume (PV) an extra resource that can be attached to the cluster. This subfile system provides the PersistentVolumeClaim (PVC) object, allowing to access resources (CPU and Memory) to the PV unit attaching to it. It is similar to a Pod, but instead of consuming node resources, they use PV’s [Kubernetes, 2020b].

### 2.5.4.7 API Gateway in Kubernetes

There is a native service in Kubernetes for microservice called Ambassador, built on EnvoyProxy (manages a decentralize proxy across cloud platform to ensure overall availability). It is considered a self service, since it can be configured through annotation in the Kubernetes configuration. Its main purpose is to deliver access to microservice, providing diagnosis, authentication, Support for gRPC and HTTP/2, TCP, and WebSockets [Ambassador, 2020]. This is accomplished by using an ingress component that routes all the traffic received into Pods in the Kubernetes cluster. To perform service discovery Ambassador uses Kubernetes DNS and service-level discovery, a resource naming convention mapping for each service. It uses the Kubernetes and cloud provider load balancer configured for the different services, so that the routing will be done to active pods.
2.5.4.8 Label Selectors

They are grouping primitives in Kubernetes, allowing to identify a set of objects by a unique label, so that they can communicate among them if they are attached [Kubernetes, 2020e].

2.5.5 Kubernetes Configuration

The configuration for kubernetes environment can be done using CLI with the different commands and it can also be specified in a YAML file, which stands for Yet Another Markup Language. It is a superset of JSON to define configuration providing convenience, flexibility and maintenance [Kubernetes, 2020e].

In the next configuration files, it will be explained how to give a detail setup of the Kubernetes configuration in deployment. Kubernetes is set up using a declarative configuration file with key values to create the different Kubernetes components. Coming next there are the main configuration for ambassador API Gateway and for each microservice, the service and deployment kubernetes components configurations that remains the same across microservices, except the names. Therefore all names inside brackets ({})) are dynamic depending on the microservice that is being configured.

2.5.5.1 Ambassador

The next set up is for creating a kubernetes service component for the API Gateway. The type of service is Load Balancer since ambassador needs to expose mapped endpoints outside the cluster. The externalTrafficPolicy is set to Local so that the client IP addresses will be propagated to the end Pods, and so the pods will receive the request as if they were invoked directly by the client.

```
apiVersion: v1
group: kubernet
kind: Service
metadata:
  name: {ambassador-name}
spec:
type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
    - port: {ambassador-port}
      targetPort: {service-port}
service: ambassador
```

2.5.5.2 Deployment

Now to create a pod and define desired state in a deployment, a container will be created for each pod. Although multiple containers can be built inside a pod, it is recommended only to support co-located or co-managed helper processes like Sidecar, proxies or bridges that need to share resources. Otherwise it is a better practice to keep the one container per pod rule, for maintainability purposes.
The selector matchlabel fields allow to attach a given name for a pod in the template label with the deployment. So a pod is created from a pull image built in a private Docker registry. Thus it is very important to create a imagePullSecrets resource with the credentials to authenticate with a container registry and pull a the image.

The replicas spec is always configured for just one pod, as can be seen in the rollingUpdate configuration. This is because the scaling will be managed based on observed CPU utilization, and the Horizontal Pod Autoscaler will be deployed for the scaling task using the resource limits specified in the configuration. In the case of the next sample, each container has a request of 0.25 cpu, guaranteeing the specified CPU resource (1/4 cpu core) to the scheduled Pod, and a limit of 0.5 cpu core, meaning they cannot exceed a half core cpu in usage, otherwise the Horizontal Pod Autoscaler will execute its configuration for deploying a new Pod instance.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {deployment-name}
spec:
  replicas: 1
  selector:
    matchLabels:
      app: {pod-name}
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
      minReadySeconds: 5
  template:
    metadata:
      labels:
        app: {pod-name}
spec:
  nodeSelector:
    "beta.kubernetes.io/os": linux
  containers:
  - name: {pod-name}
    image: {image-name}
    ports:
      - containerPort: {container-port}
    resources:
      requests:
        cpu: 250m
      limits:
        cpu: 500m
    imagePullSecrets:
      - name: regcred
```

### 2.5.5.3 Service

To expose an endpoint inside the cluster and give ambassador a route to be publicly accessible, annotations are used for this. Thus a Mapping resource is used to configure how traffic is routed to the service, allowing HTTP, gRPC, and WebSockets protocols. It is important that service-name-mapping is unique across the cluster.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: {service-name}
annotations:
  getambassador.io/config: |
    --
```
2.5.5.4 Autoscaling

Although Deployment can manage scaling with its configuration, the HorizontalPodAutoscaler allows to set a specific resource limit usage for a better resource management. With the next configuration the horizontal scaling is setup to depend on the resource usage. So a minimum and maximum replicas for a deployments is set. And the provisioning of those replicas will depend on a cpu utilization rate threshold. Therefore the Kubernetes cluster ensures the minimum number of Pods specified in the `minReplicas` spec are running for each Kubernetes Deployment resource and will scale, adding new Pods in case the `targetCPUUtilizationPercentage` is reached.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: {autoscaling-name}
spec:
  maxReplicas: {max-replicas}
  minReplicas: {min-replicas}
scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: {deployment-name}
  targetCPUUtilizationPercentage: {use-threshold}
```

2.5.6 Kubernetes Dashboard

Kubernetes includes a web-based user interface to manage the Kubernetes cluster resources as a Dashboard representing all deployment components to be tracked, which also support the creation or the modification of individual Kubernetes resources. It also provides additional resources that can be added to the cluster to provide additional features, like the Metric-server, which will be used in this project, for monitoring resources like kubelet CPU/Memory usage, which can be integrated with the horizontal auto-scaling setup [Kubernetes, 2020d]. The default Dashboard is part of any Kubernetes cluster and can be created through a proxy from the Kubernetes environment by the local administrator.
3 Software Requirements Specification

3.1 Introduction

This document will detail the requirements that the Freight Transport Services Platform “DGCLlevar” must meet.

3.2 Purpose

This specification is addressed to the stakeholders belonging to the positions of transporters’ directors as well as the coordinators, managers and developers responsible for the implementation of the 'DGCLlevar' platform.

3.3 Scope

The implementation “DGCLlevar” platform is intended to: Make available to customers an App that works on the iOS and Android Platforms. Through the App, the customer will be allowed to:

1. Register on the platform
2. Request a cargo transport service
3. Select the type of cargo transport appropriate to each need.
4. Track the location of the selected transport during the transport service.
5. Pay the transport service electronically.

The driver user will be able to:

1. Register on the platform
2. Accept a cargo transport service from the customer
3. Contact the Customer
4. Add new vehicles

In addition, it is necessary to create a Web Site (Back-Office) that supports Administration and Reporting of the information generated by the App. Through the Web Site (Back-Office), technical administrative personnel will be allowed to:

1. Manage drivers with their respective cars
2. Modify the configuration parameters of the App
3. Generate the reports of the sales generated through the transport services

This project does not include:

1. Handling of video cameras inside the means of transport.
2. No accounting management.
3. Show trace of cargo movement. Trucks have a tracking chip.
4. Management of corporate clients

3.4 Definitions

1. APP: Mobile Application
2. Web Site: Set of web pages accessible from the internet.
3. Payment Gateway: Service for the collection of transactions by credit or debit card.
4. Stakeholders: All those who are directly and indirectly involved with the project.

3.5 General Description

The following is a general description of “DGCLlevar” Platform.

3.5.1 Product Characteristics

The “DGCLlevar” platform is a solution that will have the following components that will support its operation.

1. Mobile app for Customers. It is an application that will work on iOS and Android systems. This mobile application will allow end users to access the platform in order to request transportation services.

2. Mobile app for Drivers. It is also an application for IOS and Android systems. This application supports to receive and accept transportation services from the customer. The driver will be able then to add new vehicles to provide his services. The application will also allow drivers to share their location and therefore be chosen to perform the transport service.

3. Back Office Web. Website that will allow administrative staff to carry out any necessary parameterization and to register drivers who will then provide the service to end customers. In this portal, reports on the transactions and the rides will be able to be generated. The indicated reports will be used for the accounting area.
4. Database. Repository that stores user information, rides and transactions made on the platform.

On the other hand, the "DGCLlevar" platform interacts with:

5. Payment Gateway. All transport services provided by the platform can be paid by electronic means. For this purpose, "DGCLlevar" platform will connect to the Payment Gateway, which will be in charge of processing the payment and returning a result to the "DGCLlevar" Platform.

6. SMS Mobile operator: To identify the users, the phone number will be provided in the registration process. To validate the authenticity of the phone number, SMS will be sent with a generated code that will be used in the registration.

7. Notifications: The mobile applications should receive notifications for the system main operations so that the users are aware of the different tasks.

3.5.2 Product Functions

The main functions that should be provided by the “DGCLlevar” platform are:

1. Request cargo transport service
2. Pay for cargo transport service
3. Join the “DGCLlevar” platform as a customer
4. Join the “DGCLlevar” platform as a driver

**Request cargo transport service**  It is the main functionality of the platform. Through this function the customer will be able to request the transport service from his mobile phone. For this to happen the following activities must be accomplished:

1. Download and open the customer mobile application.
2. Register on the platform.
3. Sign in.
4. Choose the position from where you want to contract the service.
5. Choose the position to where you need to deliver the cargo.
6. Take a photo of the cargo to be sent and specify its weight and dimensions.
7. Choose if you want to hire assistants for loading and unloading the property. Indicate how many helpers are required.
8. Review the route and the approximate price.
9. The platform receives the information, searches for the most suitable units that are available and closest to the pickup position, sending the data of the customer’s request.
10. The first driver to accept is the one who will perform the service.
11. The customer is notified of that a driver has accepted the request.

**Pay for freight transport service** By accepting the service the customer agrees to pay the amount calculated by the requested transport service. The activities that must be performed next are:

1. The driver finishes the transport service.
2. The payment is executed from the selected payment method (card or cash).

**Join the "DGCLlevar" platform as a customer** In order to use the transport services offered through the Platform, a client must register on the platform. The activities for the client to register are as follows:

1. Download and open the mobile application (from customer)
2. Choose to register
3. Enter personal information
4. Enter the payment method
5. Enter password
6. The platform sends account confirmation email

**Join the platform " DGCLlevar " as a driver** In order to provide cargo transportation service through the “DGCLlevar ” platform, the driver must register on the platform. There are two ways for a driver to join the platform: Mobile Application. The activities that must be carried out are:

1. Download and open the mobile application (from driver)
2. Choose to register
3. Enter personal information
4. Enter account data
5. Attach the images of:
   (a) Driver’s photo
   (b) License (front and back)
   (c) Registration (front and back)
   (d) Vehicle photo
   (e) Basic service invoice (Water, electricity or telephone).
(f) Vehicle plate.

6. The platform sends confirmation email to receive the request.

7. Approve or deny manually the request ("DGC Llevar" staff).

8. Send result email.

9. Enter password.

And the other is with the Back-office. Activities to be performed in this case:

10. Edit personal information driver

11. Edit account data driver

12. View images:
    (a) Photo of the Driver
    (b) License (front and rear)
    (c) Registration (front and back)
    (d) Photo of the vehicle
    (e) plate vehicular

13. The platform sends email for confirmation

14. Enter a password

**Characteristics User** The users that are going to use the Platform 'DGCLlevar' are grouped into the following groups:

1. Client. Anyone who has downloaded the customer mobile application and has registered on the platform. These users will be people who have or have ever had the need for a cargo transport service.

2. Driver: Any person who has downloaded the driver application and who has registered on the platform and has also been accepted to provide the transport service. This user will be a professional driver who has a vehicle to provide cargo transportation service.

3. Platform Administrator with an administrative profile: who is dedicated to managing the Back-office solution of the 'DGCLlevar' Platform. These users will generally be employees of the company that is operating the 'Truck App' platform.

4. Accountants: Accounting / financial staff: dedicated to generating accounting reports in the Back-office solution of the 'DGCLlevar' Platform. These users will generally be employees of the company operating the platform.

5. Platform Administrator with a technical profile: who is dedicated to managing the Back-office solution of the Platform “DGCLlevar”. These users will be in charge of consulting technical information such as the review of logs generated by the application in text files that can be viewed from the web.
 Dependencies and Assumptions

1. Use of a Payment Gateway.
2. Transportation Service Calculation. The owner of the platform will provide the formula for calculating the corresponding value.
3. Both mobile applications need the following permissions:
   (a) Notifications: To send ride alerts.
   (b) GPS: To track the location in the request of a ride and in progress.
   (c) Camera / Gallery Access: To take / select necessary photos in the registration workflow and the travel request registration.

3.6 Functional Requirements

Join the “DGCLlevar” platform as a customer

3.6.1 Req (1) New customer

The customer mobile application must allow for creating a new user on the Platform with all the required information.

The information required is as follows:

1. First Name
2. Last Name
3. Gender
4. Date of Birth
5. City
6. Landline
7. Phone number
8. Full Address
9. Email
10. Password
11. Acceptance of Terms and Conditions
3.6.2  Req (2) New Customer Account Confirmation

The Platform must send an account confirmation email to the customer in order to confirm his/her email. This confirmation will allow the customer to continue using the application after 7 days of creating the account.

3.6.3  Req (3) SMS Confirmation of Phone number

The Platform must send an SMS with a confirmation code for the client to use it during the registration process. This confirmation must be part of the initial creation of the user.

3.6.4  Req (4) Save and select Payment Method

The Customer mobile application must allow for selecting a payment method and save all the information associated with it. The allowed payment methods are credit card or cash. When the user registers, the platform will assign his/her the cash payment method by default. The customer can add a method of payment by card (credit or debit), once he is logged in, being the gateway in charge of carrying out the card validation.

Join the “DGCLlevar” platform as driver

3.6.5  Req (5) Driver Request

The mobile driver application must support the creation of a platform user request. The data necessary are:

1. First name
2. Last name
3. Date of Birth
4. Email
5. Province
6. City
7. Address
8. Type of identification
9. document Number of identification document
10. Type of driver’s license (C, D or E).
11. Expiry date of driver’s license.
12. Expiration date of vehicle registration.
13. Cooperative to which the driver belongs
14. Password
15. Phone number
16. Acceptance of Terms and Conditions

3.6.6 Req (6) Bank Account Information

The driver’s mobile application must store the driver’s bank account information. It is on this account in which the payments will be collected from all the services that drivers have made.

3.6.7 Req (7) Driver Images

The Mobile application must save the images indicated in Join as a Driver section. All images provided will pass into a validation phase performed by one member of the Administration department. In the event that the images are rejected by the platform administrator, the workflow will allow the images to be re-uploaded for approval. The approval or rejection process will be done by sending an informative email.

3.6.8 Req (8) Application Email Notifications

Email Notifications must be sent to the driver once the registration has been sent to revision to inform driver that the platform will start a one-week validation process. After this validation is made, the driver will receive a notification with the result, i.e., his approval or rejection.

3.6.9 Req (9) Account Validation

The “DGCLlevar” platform must validate the user’s account by sending a confirmation link to the user’s email address. The Validation is made after user clicks on a link in the driver approval and user creation email sent by the platform.

3.6.10 Req (10) Account Approval

The back office of the “DGCLlevar” platform must allow an Administrator to view, approve or deny all requests to join the platform from the drivers of cargo vehicles.
3.6.11 Req (11) Creation of Driver User

The back office of the “DGCLlevar” platform must allow an Administrator to create a driver user by entering all the necessary information described in Req (5).

3.6.12 Req (12) Authentication of the Driver Mobile Application

The mobile application will support authentication of the driver user with an email account and password. The same that will return the information of the driver and all the vehicle(s) associated with it. When authenticating, you can update the list of vehicles registered in the device’s memory. It is associated with the requirement Req (11) Creation of driver user.

3.6.13 Req (13) Vehicle Administration per Driver

The web application will support the administration of vehicles associated with the driver.

To create a new vehicle, the following information must be specified:

1. Make
2. Model
3. Year
4. Plate
5. Color
6. Type of Transport: Depending on the capacity of the truck, select from:
   7. Single Cab
   8. Light Truck (1-3.5 tons)
   9. Truck (up to 5 tons)
10. Truck double cabin

Cargo transportation service management

3.6.14 Req (14) Request a Transportation Service

The customer mobile application must have the ability to request a cargo transportation service. For this you must specify the following information:

1. Origin
2. Destination
3. Type of Transport
4. Referral Guide (Photo)
5. Description of the Cargo
6. Dimensions of the Cargo (Optional): Height, length and width in meters

3.6.15 Req (15) Route Calculation and Value of the Service

The customer mobile application must display the route from the pickup location to the destination. It also needs to show an estimated fare based on the ride configuration parameters and the fare formula.

3.6.16 Req (16) Location of Transport Units

The “DGCLlevar” platform must search for the closest units that are available that meet the customer’s requirements, among which it is filtered depending on the type of vehicle required and proximity. A maximum of 10 drivers will be considered for the request, being the closest ones to the customer’s pickup location.

3.6.17 Req (17) Display of Transport Requests and Approval or Rejection Process

The driver’s mobile application must show the customer’s ride request parameters to the selected drivers. So, they can review them to accept or deny the request. Only one driver will be able to accept the request, so the rest of the drivers must be informed that the ride is no longer available.

3.6.18 Req (18) Display Location and Driver Information

The customer mobile application must display the information of the driver and the location of the transport unit that has agreed to provide the cargo transport service during the whole service.

3.6.19 Req (19) Enable and disable Driver Availability

The driver mobile application must have the ability to enable or disable driver availability to receive transportation service requirements. In the case of enabling, the driver must choose the previously created vehicle with which he wishes to provide the service.
3.6.20 Req (20) Start of the Ride

The driver’s mobile application will have control over the life cycle of the ride. More precisely, to start it once the passenger has been picked up by the driver at the pickup location. This is why the option to start the ride should only be enabled within a radius of 200m from that location.

3.6.21 Req (21) Ending the Ride

Again, the driver’s mobile application will have the option to end the ride once he/she is in the destination location within a radius of 200m. At this point the driver can finish by executing the payment method chosen by the passenger and informing that it is being processed. The execution of the payment method will be detailed in requirements below.

Pay freight transport service

3.6.22 Req (22) Payment Execution

The 'DGCLlevar' platform will process the payment internally taking into account the method selected by the passenger, as soon as the service has been completed. For this, if the selected payment method is card, it will be processed through the payment gateway and a confirmation email is sent to let the customer know that the payment has been charged. If the selected payment method is cash, the driver will be in charge of collecting it. The fare is calculated with the next parameters:

1. Cost of the service. Taking into account the type of vehicle, the distance, the traffic, the number of helpers requested. The specific formula for calculating the service will be provided by the owner of the platform in the future.

2. 25% Commission of the cost of the service. Of this percentage
   6% is retained by the platform.
   17% is retained by the transport company.
   2% is retained by the payment gateway

3.6.23 Req (23) Payment Recalculation

The mobile driver application will be able to perform a recalculation in case it considers that any of the variables has changed from the original requirement. The variables can be the following:

1. Modification of the destination.

2. An additional cost is incurred, such as tolls or customs taxes.

3. An extra assistant is required to receive and / or deliver the cargo.
3.6.24 Req (24) Payment Monitoring

The customer will be alerted regarding the status of payments so that the information is updated by email after each transaction with the platform. The platform will offer a section with the following details of the ride:

1. Pick up location address
2. Destination location address
3. Distance traveled
4. Description of the load
5. Driver’s name
6. Cost of the service
7. Cancel cargo transport service

3.6.25 Req (25) Cancellation of Cargo Transport Service

The customer and driver mobile application must be able to cancel the cargo transport service before the ride has started. In the event of an incident during the ride, the driver will have the option to cancel it with the prior authorization of the customer by means of a confirmation message.

Qualify service provided

3.6.26 Req (26) Qualification of the Provided Service

The customer’s mobile application must be able to rate the transport service received from a private driver as soon as the ride ends. This will not be considered finished until the payment has been approved. This rating will be represented in 5 stars, of which 1 is the lowest rating and 5 the highest. For ratings less than or equal to 3 stars, a required comment field must be added to get customer feedback.

Review Rides History

3.6.27 Req (27) Ride History

The mobile client and driver application must allow users to see the history of the rides users have made on the platform as well as the amounts that have been paid for each ride and the details of each one.
Change payment method

3.6.28  Req (28) Change Payment Method

The customer’s mobile application must support to change the payment method. In the case of card, customer must enter the following information in the payment gateway.

1. Card number
2. Expiration Date
3. Name of the Card Owner
4. CV Code
5. User Management.

Back-office features

3.6.29  Req (29) User Administration

The back-office of the platform must allow managing the users who will be able to access the back office. There are two types of users that must be created:

1. Administrators
2. Access all the reports
3. Create new drivers Req (11)
4. Accountants
5. Access the financial reports.

3.6.30  Req (30) Administration Parameters.

The back-office of the platform must allow managing the parameters required by the client and driver mobile applications to function properly. The parameters initially required will be the following:

1. Application server name
2. Port used for services (Node and .Net)
3. Maximum distance (km) between clients and drivers
4. VAT percentage
5. Email and account credentials for sending emails.
6. Reports

3.6.31 Req (31) Report Transaction

The back-office must offer a Financial report that shows all the transactions (financial movements generated between a passenger and the driver at the time of subscription or when paying for the rides) that have been performed within the platform. They must be classified by type of payment: cash and credit card. The user who will view this report will be that user with an accounting role on the web platform.

3.6.32 Req (32) Report of Transactions with Error

The back office must offer a Financial report that shows all the transactions that have had problems in the execution of the collection.

3.6.33 Req (33) Ranking of Drivers

The back office must offer an Operating report that shows the ranking of all drivers. This ranking is going to be calculated with the average of all the ratings given by customers after the completion of a transportation service. The administrative user will be the one who can access this report.

3.6.34 Req (34) Report of Transactions pending reversal

The back office must provide a report with the transactions pending reversal. This case may arise when making a change in the type of payment from card to cash, after an error occurred in the first attempt to collect by card, pending payment. The accounting administrator will manage this information.

3.6.35 Req (35) Report of Drivers

Report of the drivers who have registered to the platform with personal information and details of rides made on the platform. The reports will list the driver information and will include sub reports that will show more detailed information for each of the drivers.

1. Report columns
2. Name and surname
3. Identification document type Identification
4. document number
5. Date of Birth
6. N° Mobile phone
7. Sub-report columns
   (a) Province
   (b) City
   (c) Address
8. Type of driving license
9. Expiration date of driving license.
10. Cooperative.

3.6.36 Req (36) Report of Rides completed by Driver

In addition, the total of the amounts made in cash and credit card by each driver must be shown. It will be used for accounting and for transferring the money to the drivers. Transfers will be ordered through an external accounting system that will receive the information through a web service.

3.6.37 Req (37) Report of Rejected Drivers

Report of the drivers who have been rejected in the approval process of the documents uploaded through the platform.

3.6.38 Req (38) Report of Clients

Report of the passengers who have registered to the platform with personal information, details of rides made on the platform and attached documents.

3.6.39 Req (39) Billing Market Report

Report that indicates by range of dates the number of rides and the total value generated in the indicated range.

3.6.40 Req (40) Dashboard

The dashboard should show the following information as a summary of general data:

1. Values collected by card (debit and credit)
3.7 Non-functional Requirements

3.7.1 Req (41) Testability

The back-office must keep a log of files that will be stored in the server with the requests made to the API including the endpoints and the transferred data, as well as all the operations carried out by users who make changes to the database. The back-office with the technical user profile administrator will have the option of consulting logs with read permissions that will be ordered by dates.

3.7.2 Req (42) Authentication and Authorization

The client, driver and back-office application must ensure that only previously authenticated and authorized people, depending on the type of user, access the application and can carry out the operations permitted according to their user role.

3.7.3 Req (43) Availability of the server

Recoverability of the servers, in the event of a possible crash, it will be operational again in less than a minute.

3.7.4 Req (44) Availability of the mobile apps

The customer and driver mobile applications must have the ability to resume a transport service that is currently running. This will be useful in case the application has been closed or has lost the Internet connection for any reason.
3.7.5 Req (45) Scalability and Performance

The system must be able to tolerate a 5 seconds frequency rate when drivers are sharing its location once they become available and during a ride. The workload may be increased based on the number of active users, so it is important that the system can handle traffic pikes.

3.7.6 Req (46) Availability of services

Monitoring the availability of services in real-time.

3.7.7 Req (47) Payment Security

Guarantee the confidentiality of payments in such way that they are not accessible by outsiders and implement security tactics in the processes where sensible data of the payment gateway is manipulated.
4 Development

In this section the different architecture abstraction levels will be discuss to provide a full understanding from the overall implementation of the different microservices that make up the Real-time Back-end.

4.1 Context of the Car Pooling Information System

The next graphic represents a high level view of the system, showing the boundaries established with the external components that the system communicates with across different protocols. Also it defines the scope of the current research only focusing in the car pooling information system and consuming/providing different external services.

![Figure 11: High level modular decomposition](image)

The platform has three different user roles: Customers, Drivers and Administrators. For each user, there is a different Front-end application providing the specific features for the different use cases. In the case of Customers and Drivers they have two Mobile applications and for the Administrator a Web application.

The are two types of communication protocols to interact between the CPIS (Car Pooling Information System) and the external components: HTTPS and WebSockets:

1. **HTTPS**: This protocol establish communication between the different components, taking the arrow direction as a reference of the component that starts the communication to consume the API services provided by the invoked resource. Therefore the CPIS will consume the Mobile Operator, Notification and Payment Gateway
services, whereas the Mobiles and Web applications will do it using the CPIS services. And for the Mobile Operator, Notification Service and Payment Gateway it was integrated using their own implementations to connect from the CPIS to the external components.

2. **Websockets**: This communication channel is the key to achieve real time capabilities with the server, since it is more time and message length efficient than HTTPS, and it supports bi-directional communication by allowing the server to start requests. This is only used to provide real time support to the mobile apps. The library used in the Mobile applications and the CPIS is Socket.io [SocketIO, 2020] a Websocket implementation (among other protocols) that has their own implementation for the client and server side.

### 4.1.1 Mobile Application for the Driver

It is a multi-platform mobile application for the driver user that communicates with the CPIS by using HTTPS and Web Sockets. For each communication protocol the next services are consumed by the driver application:

#### 4.1.1.1 HTTPS

1. Login in and out from the application
2. Change password
3. Getting all driver information
4. Create a bank account
5. Confirm images upload in the registration
6. Set the driver mobile phone
7. Register driver in the application.
8. Register a notification Token for receiving notifications
9. Verify SMS code
10. Get the current App version to lock usage on deprecated application versions
11. Accept requested rides
12. Resume a current ride
13. Get pending requested ride
14. Disable driver availability
15. Get driver status
16. Cancel request ride
17. Alert customer proximity
18. Start the current ride
19. Cancel the current ride
20. Finish the current ride
21. Verify an online payment
22. Check rides history
23. Check if driver is active on another device
24. Get the driver registered cars

4.1.1.2 Web Sockets

1. Open and close Socket Connection
2. Subscribe to requested rides
3. Subscribe to cancel requested rides
4. Subscribe to online card payment result
5. Subscribe to customer cancel rides
6. Send location updates when available and during a ride.

4.1.2 Web Application

It is an administration web application that manages the business operation, as specified in the SRS. It communicates with the CPIS using HTTPS to deliver the different features that the operation requires.

4.1.3 Mobile Application for the Customer

It is also a multi-platform mobile application for the driver user that communicates with the CPIS also through HTTPS and Websockets. The requested services consumed by the customer application are the following ones:

4.1.3.1 HTTPS

1. Login in and out from the application
2. Getting all customer information
3. Register customer in the application.
4. Update contact user information.
5. Change password
6. Get the different payment methods
7. Set up a default payment method for rides
8. Verify SMS code
9. Get the current App version to lock usage on deprecated application versions
10. Register a notification Token for receiving notifications
11. Submit issues
12. Estimate the distance and fare of carrying the package from a pickup to a destination location and a type of car.
13. Request rides to carry the package
14. Cancel a current ride
15. Check rides history
16. Resume taken rides
17. Pay with online card and subscription
18. Rate drivers after a service ride

4.1.3.2 Web Sockets
1. Open and close Socket Connection
2. Subscribe to driver ride approval, refused and finish
3. Subscribe to driver cancel, near pickup location, starting and finish ride.
4. Subscribe to driver location updates
5. Subscribe to online card payment request and result

4.1.4 Mobile Operator

It is in charge of sending SMS messages to the users that provided their numbers in the registration process for confirmation purposes. Since it is very important to make sure that the provided phone number is correct, in order for the driver or customer to contact the other party during a ride. The provided service exposes an API that can be used to send the messages from the CPIS when needed using HTTPS attaching the corresponding text message.
4.1.5 Notification Service

It supports to send custom notification messages to registered mobile phones in a Notification Service. This Notification Service is provided by the Operating System. So in case of multiple support, all those services need to be provisioned. For the Notification Provider (the CPIS module in charge of notifications) a Server API key needs to be configured in the case of Android and an Push Notification Certificate in the case of IOS needs to be set up, as explained in the Background section. Once the requested resource is configured in the Notification Provider, the Mobile Phones need to register in the Notification Service and grant specific permissions so that they can start receiving notifications. More details of the implementation for sending notifications will be given in the Design for each Microservice section.

The next service providers are configured in the CPIS, using HTTPS for the mobile registration and sending notifications:

1. Firebase Cloud Messaging: This is an Android Notification Service, an API key needs to be generated in the Firebase console and configured in the Notification Provider. A configuration file will be generated and installed in the Notification Provider to support Android notifications.

2. Apple Push Notification: For IOS support a certificate needs to be created from an Apple Developer licensed account and configured in the Notification Provider.

4.1.6 Payment Gateway

This is a third-party service to authorize and process payments by card using a Point of Service (POS) terminal. There are multiple companies that offer Payment Gateway platforms and the workflow can change across different providers. But for testing purposes the general and more common approach will be followed to show the integration. The implementation of this service is done using its API REST that is consumed with HTTPS from the CPIS and the mobile applications.
4.2 High Level Description of Main Operations

In the current section the two main operations will be described using a process model with BPMN (Business Process Model and Notation), focusing on the CPIS operation for the ride request and the subsequent ride lifecycle. The diagrams will be found in the next pages for a better visualization but they will be explained just below:

4.2.1 Ride Request Process

In the case of the ride request, there are three different actors involved in the process: The mobile application for the Customer, for the Driver and the CPIS. The goal of this flow is to show how the different actors participate in a ride request without specifying the communication protocols used between them. Figure 12 shows the corresponding workflow.

4.2.2 Ride Stages Process

Once a ride has been accepted by the driver, then it will go through different stages based on the interaction between the customer and driver. The last stage is the payment processing, where depending on the chosen payment type (cash or card) different tasks will be executed. Figure 13 shows the required steps.
### 4 Development

#### 4.2 High Level Description of Main Operations

<table>
<thead>
<tr>
<th>Customer Mobile Application</th>
<th>Car Pooling Information System</th>
<th>Driver Mobile Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request a ride</td>
<td>Clear Inactive Drivers</td>
<td></td>
</tr>
<tr>
<td>Fetch available Nearby Drivers in a radius (10km) from the requested location, with the requested car type and a limit of (3) drivers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Group result drivers in two categories: Those who are available to accept rides and those who are reviewing other rides.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inform Customer</td>
<td>Any Available Driver?</td>
<td>Receive Notification Event</td>
</tr>
<tr>
<td>For each Available Driver Send a Notification and Socket Event with request data. And starts 60 seconds timer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inform Customer that drivers are available and reviewing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update status with driver reviewing the current ride</td>
<td>Open Request Event</td>
<td></td>
</tr>
<tr>
<td>Check that ride is still available to be assigned</td>
<td>Accepts ride?</td>
<td></td>
</tr>
<tr>
<td>Inform Customer that there are no drivers available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Is the ride still Available?</td>
<td>Inform Driver</td>
<td>Receive Notification Event</td>
</tr>
<tr>
<td>Create Ride with Driver assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inform rest of drivers, that the ride has been taken</td>
<td>Receive Notification Event</td>
<td></td>
</tr>
<tr>
<td>Inform Customer that the driver has accepted the ride</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 12: Request Ride Process*
4.2 High Level Description of Main Operations

Figure 13: Ride Stages Process
4.3 High Level Modular Decomposition

The previous mentioned CPIS is made up of three components: The API Gateway, Real-time and Administration subsystems. They live inside Kubernetes cluster environment, communicating for transmitting resources on demand as shown in Figure 14. Inside Kubernetes there are not separated boundaries between Real-time and administration Back-ends as shown in Figure 14, since they are part of the same Kubernetes runtime implementation and at the end they are just a set of microservices running inside the cluster. As a strategy to organize the microservices in the environment, the Kubernetes namespace concept will be use in the deployment files discussed in further section, to provide a logical separation between environments under the names Real-time and Administration.

![Figure 14: Interservice Communication](image)

4.3.1 Api Gateway

Both Back-ends Real-time and Administration are part of the Kubernetes implementation. All of their endpoints are exposed publicly using the Ambassador API Gateway, which uses Kubernetes ingress to route web traffic into pods. It also supports in the Cloud Service load balancing components and service discovery. Therefore, all routes are mapped into microservice endpoints, delegating responsibility and proxying request and response resources on demand.

The cluster, as was explained in the background section, is composed by the Control
Plane components, that instead of using directly the kube-apiserver it is replaced by the API Gateway.

4.3.2 Real-time

It consists of all the microservices in charge of providing the ride request and ride lifecycle management, driver location tracking and connection handling. Each microservice will have redundancy and will scale based on the traffic received, this will be explained further in the Allocation View section. This traffic will be received through the API Gateway and proxied to the microservices, which are not visible outside the cluster, but still they can be invoked to each other internally. Next, we can find an overview for all microservices, which will be explained deeper in further sections:

1. **Connection:** It manages the socket connection with the customers and drivers apps, mapping socket identifiers to database users. Therefore, all socket interactions will be handled by this microservice to emit and receive events using SocketIO. It also manages the interaction with the Notification System for all users, regardless the operating system (Android and IOS) using push notifications services.

2. **Driver Location:** It handles location tracking for the driver, keeping track of drivers availability and querying geospatial coordinates to select the most suitable drivers to the corresponding rides requested by the customers.

3. **Request Ride:** It handles all rides requests coming from the customers, giving information for estimated fares, sending request to drivers relying upon Driver Location and Connection microservices.

4. **Ride:** It manages all the different stages of the active rides, once a driver has accepted a request. It handles communication with the Administration Back-end to update ride status and process payments.

4.3.3 Administration

Set of microservices to manage authentication, Ride history, process payment and report generation. As happens with the Real-time, the API Gateway ingress dispatches requests to the different microservices and they can communicate internally. The Administration Back-end provides the following microservices:

1. **Customer:** This microservice allows customers to manage their personal information and stores the tokens for push notification to customers. It also provides information on the customers to other microservices.

2. **Driver:** This microservice allows drivers to manage their personal information and stores the tokens for push notification to drivers. It also provides information on the drivers to other microservices.

3. **Ride:** Ride microservice support to handle rides requested by the customer and accepted by a driver. It stores the different states of the ride from beginning to end.
4. **Payment Gateway:** Payment Gateway microservice allow to handle the payment using a third party Payment Gateway.

5. **Configuration:** Configuration microservice supports to manage general configurations across the application. These configurations are entered through the Administration web application.

6. **Notification:** Notification microservice allow to manage notifications using different channels: Email, SMS, etc.

7. **Reporting:** Report microservice provide different kind of reports specified in the SRS, from the relational database.

### 4.3.4 Inter-service Communication

There is a need to communicate microservices to access external resources to perform multiple operations for some requirements. In the case of databases is very important that each microservice has their own, so their direct access is restricted to external microservices. Therefore they can only be accessed through HTTP calls inside the Kubernetes cluster, using configured networking rules. Consequently those services require extra synchronization and transaction strategies to keep data integrity and consistency. These can be complex specially if microservices are implemented using different technologies, so there is a need of using software patterns.

For inter-service dependencies all the microservices use a synchronous communication with HTTP protocol to make the API calls. This means that the invoked microservice will have to wait until the caller microservice responds to perform the next operation. This is also the case of API Gateway proxy request with HTTP, except for the Connection microservice, which uses WebSockets from the mobile applications. So, the API Gateway will act as a bridge forwarding all received events.

All microservices have their own database hosted with different cloud Providers depending on whether they belong to Real-time (NoSQL) or Administration Back-end (SQL). For all the Real-time microservices the database system used is Mongodb hosted by Mongodb Atlas service, which is fully automated. At the beginning it was considered to deploy a container in the same Kubernetes cluster to provision the database, but database cloud services offer more benefits that are essentials for microservices like transactions in the case of MongoDB and fully automation. And as multiple operations are performed in several databases for same request across their microservices, transactions are essentials.

Next, all the communication links are explained for the different microservices. It is important to recall that the protocol used to established communication is HTTPS.

1. **Link from Connection microservice to Push Notification Service.** This communication allows to send notification to prior registered devices with the same Notification Service directly, using the generated token. This token is stored in the Administration microservices, so it will be requested on the link 8 and 9.

2. **Link from Rides to Connection microservice,** intended to publish events and notifications for ride status updates and payments requests.
3. Link from Rides to Driver Location to switch driver availability whenever a ride is started and finished.

4. Link from Connection to Driver Location microservice, intended to update driver’s location when available to accept rides and during one.

5. Link from Request Ride to Driver Location microservice, performing driver search on requested rides.

6. Link between Request Ride and Rides microservice to transition from a customer requested ride that will store all data in the Request Ride database and then consolidating in the Ride database after the driver approval.

7. Link from Request Ride to Connection microservice, intended to send new ride request to drivers, cancel those requests upon customer or driver demand, and send timeouts to customer in case of ride request have not been accepted.

8. Link between Connection microservice and Administration Driver microservice to store generated notification tokens from driver device. So, the Connection microservice can be requested afterwards.

9. Link between Connection microservice and Administration Customer microservice to store generated notification token from customer device. So, the Connection microservice can be requested afterwards.

10. Link between Rides and Administration Ride and Payment microservice. The different ride status (created, started, canceled by driver and customer), ride rating and all the different finish business logic with the payment will be forwarded to the Administration Ride microservice so that it can be stored in database. The Real-time Rides microservice will expose a report service to the Administration Ride microservice.
4.4 Deployment of the Real-time Microservices

The current section includes the microservices deployment configuration of the Real-time microservices for a production environment. Therefore the kubernetes configuration will be introduced for each microservice in a single file containing the service, deployment and autoscaler Kubernetes objects splitted into sections. Also the selected database configuration will be included for the cloud MongoDB service: Atlas.

![High level modular decomposition](image)

**Figure 15: High level modular decomposition**

4.4.1 Connection

Connection microservice is in charge of managing customers and drivers socket connection and mapping to database users with their corresponding keys.

4.4.1.1 Kubernetes Configuration

All the traffic for the connection microservice will be redirected to the `/connection` route handled by Ambassador. This microservice supports HTTP, which the Api Gateway manages automatically and WebSockets, therefore the flag `use_websocket` is set to `true`. It is expected that this microservice will receive huge traffic because of its dependency among other microservices, and all the socket connections to the driver and customer Mobile Application that will be listening to and dispatching events during a ride. Consequently the scaling of this microservice is extremely important to handle all the demand and spread the load evenly among instances. So this is why the maximum number of replicas is set to 7 with a threshold of 70% CPU usage to add new instances as shown in the autoscaling configuration.

```yaml
apiVersion: v1
class: Service
metadata:
```

[1] apiVersion: vl
[2] kind: Service
[3] metadata:
### 4.4 Deployment of the Real-time Microservices

#### 4.4.1.2 Database Configuration

It is a microservice that will receive large traffic from outside the cluster and within
inter-service communication, database scaling is very important to distribute load among
mongod\cite{MongoDB2020c} instances running on multiple nodes. For this it is necessary to use an Atlas Mongodb cluster that supports sharding. This is provided starting from M30+ Cluster tier, so the most basic configuration for this is with 8GB of RAM with 40GB of storage, deploying two server shards each of them with a 3-replica set.

This Database will have only one collection under the same microservice name: connections. As the database will have a sharded cluster, the sharded key needs to be generated out of the collection indexes. None of the indexes for this collection are possible candidates for a sharded key, since they do not comply with the required properties (High cardinality, low frequency, query isolation and insert scaling\cite{MongoDB2020g}). For this reason the strategy to generate the key will be to use the Hash-based sharding with the socket id and the user id. This will be explained further in the section on the detailed design of this microservice.

### 4.4.2 Driver Location Microservice

This microservice manages driver location tracking, availability and geospatial queries for driver users.

#### 4.4.2.1 Kubernetes Configuration

All the Driver Location HTTP requests will be redirected by Ambassador to the /driverLocation route for the exposed endpoints. The maximum number of instances is set to 5, starting with two instances to handle the demand that will increase with more active available drivers using the application during waiting time and in a ride.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: driver-location-service
annotations:
  getambassador.io/config: |
    ---
    apiVersion: ambassador/v1
    kind: Mapping
    name: driver-location-service-mapping
    prefix: /driverLocation
    service: driver-location-service:3000
    timeout_ms: 20000
    use_websocket: true
spec:
  ports:
  - name: driver-location-pod
    port: 3000
    targetPort: 3000
  selector:
    app: driver-location-pod
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: driver-location-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: driver-location-pod
  strategy:
    rollingUpdate:
```
maxSurge: 1
maxUnavailable: 1
minReadySeconds: 5
template:
  metadata:
    labels:
      app: driver-location-pod
  spec:
    nodeSelector:
      "beta.kubernetes.io/os": linux
    containers:
      - name: driver-location-pod
        image: driver-location-image
        ports:
          - containerPort: 3000
        resources:
          requests:
            cpu: 250m
          limits:
            cpu: 500m
        imagePullSecrets:
          - name: regcred
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: driver-location-autoscaling
spec:
  maxReplicas: 5
  minReplicas: 2
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
  targetCPUUtilizationPercentage: 70

### 4.4.2.2 Database Configuration

As it is expected that this microservice will receive also much traffic, especially for writing operations in the case of location updates, it also requires a sharding cluster to distribute loads across different nodes. So again the selected cluster tier for this database is the M30+ cluster tier, with 8GB RAM and 40GB of storage deploying two server shards each of them with a 3-replica set.

The current database has one collection under the name: `driverlocations`. This collection will store the current location driver every time is received, so that then they can be queried for fetching the closest drivers from the customer’s location. Consequently, the location coordinates must be included as an index to create the sharded key, but not directly because geospatial coordinates are store as a 2d (Array) index [MongoDB, 2020b]. Therefore, we will create two more index fields containing each coordinate and using Range-based sharding to create the sharded index. In this way, the shard groups will be defined close to each other, and therefore the `geonear` query can benefit from query isolation property without adding a performance overhead.

### 4.4.3 Request Ride Microservice

The current microservice is in charge of sending ride request to drivers made by the customers and managing the acceptance workflow of those requests.
4.4.3.1 Kubernetes Configuration

For this microservice the HTTP traffic is redirected to /requestRide route for handling the different exposed endpoints. The majority of these endpoints will be called once per ride, therefore the traffic won’t be as large as the two previous microservices. Therefore only a maximum of three replicas will be deployed if any of the other two instances exceeds 70% CPU usage.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: request-ride-service
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v1
      kind: Mapping
      name: request-ride-service-mapping
      prefix: /requestRide
      service: request-ride-service:3000
      timeout_ms: 20000
      use_websocket: true

spec:
  ports:
    - name: request-ride-pod
      port: 3000
      targetPort: 3000
  selector:
    app: request-ride-pod

apiVersion: apps/v1
kind: Deployment
metadata:
  name: request-ride-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: request-ride-pod
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
      minReadySeconds: 5
  template:
    metadata:
      labels:
        app: request-ride-pod
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: request-ride-pod
          image: request-ride-image
          ports:
            - containerPort: 3000
          resources:
            requests:
              cpu: 250m
            limits:
              cpu: 500m
          imagePullSecrets:
            - name: regcred

apiVersion: autoscaling/v1
kind: HorizontalPod Autoscaler
metadata:
  name: request-ride-autoscaling
spec:
  maxReplicas: 3
  minReplicas: 2
```
4.4.3.2 Database Configuration

The traffic received by this microservice for reading or writing operations does not justify the use of sharding for scaling. Regular database cluster comes with a 3-replica set made up of a primary and two secondary nodes for availability, so that this configuration is enough to handle the load. Thus, a M10+ cluster tier, which it is still a dedicated server, with 2GB RAM and 10GB Storage will be able to handle all the database operations of this microservice.

4.4.4 Ride Microservice

The last microservice manages the active rides and all their status updates, communicating with the Administration microservices to broadcast these updates.

4.4.4.1 Kubernetes Configuration

The Ride Microservice will received the traffic related to only active ride updates. Although this could happen multiple times during a ride, it won’t represent a huge amount of traffic, being more than the Request Ride Microservice but not as much as the remaining microservices. This is why a maximum of 5 replicas will be deployed, starting with a minimum of two, autoscaling if any instance exceeds 70% CPU usage.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ride-service
annotations:
  getambassador.io/config: |
  ---
  apiVersion: ambassador/v1
  kind: Mapping
  name: ride-service-mapping
  prefix: /ride
  service: ride-service:3000
  timeout_ms: 20000
  use_websocket: true
spec:
  ports:
  - name: ride-pod
    port: 3000
    targetPort: 3000
  selector:
    app: ride-pod
-----------------------------
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ride-deployment
spec:
  replicas: 1
  selector:
```
4.4.4.2 Database Configuration

For the last microservice it is expected that the workload received by the database will be higher than Request Ride microservice, but still sharding is not justified since requests won’t have high throughput and fast data growth. Therefore, an M20+ cluster tier with 8GB RAM and 40GB Storage will be able to handle the traffic.
4.5 Design of the Realtime Microservices

In the current section the different microservices for the real-time are described by showing the data models and the different endpoints that will be exposed through the API Gateway, so the different front-end applications can use these resources. To access the MongoDB database the corresponding ODM Mongoose [Mongoose, 2019] is used for all microservices, using the provided API to establish the connection and execute the different database operations.

4.5.1 Connection Microservice

It is in charge of managing the socket connections and notifications for the customer and driver users.

4.5.1.1 Data Model

The Connection Data Model (see Figure 16) has only one Collection with the next fields:

1. \_id The MongoDB auto-generated primary key
2. id A unique user identification from the Administration microservices
3. socketId A Socket.io connection identification so it can be queried to send events
4. role The user role that belongs to the id, being customer or driver
5. createdAt and updatedAt are Mongodb timestamps

<table>
<thead>
<tr>
<th>Connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ _id: PK</td>
</tr>
<tr>
<td>+ id: Number</td>
</tr>
<tr>
<td>+ socketId: String</td>
</tr>
<tr>
<td>+ role: String</td>
</tr>
<tr>
<td>+ createdAt: Date</td>
</tr>
<tr>
<td>+ updatedAt: Date</td>
</tr>
</tbody>
</table>

Figure 16: Connection Data Model

The collection only has one index: the id field, which is unique, and it is meant for querying by the user identification, a common task required by the other microservices to retrieve the socket connection and for sending events.
4.5.1.2 Socket Events

The Socket.io [SocketIO, 2020] library is used to establish communication between the mobile applications and the Connection microservice. Therefore both sides will have to integrate the library for the client and the server in order to support connectivity. Although Socket.io does not always use WebSockets, the connection protocol will depend on how stable the client connection is, starting with XHR or JSONP and being upgraded when possible to WebSockets, in order to guarantee connectivity and avoid degrading user experience.

Socket.io uses publish and subscribe pattern to communicate between Mobile applications and the Connection microservice, supporting to send events from one Mobile application to the other. This is possible when the receiver subscribes to an event that the sender emits always using the microservice as an intermediate level to identify socket connections mapped to a database user identification, and forward the event to the receiver. It is important to clarify that Websocket connection is not reliable, therefore in case some data is transmitted when the connection to the receiver is loss, when it connects back it won’t receive the data unless server sends it back again.

The microservice is able to subscribe to socket event emitted from the Mobile applications, in order to capture the event with the received data and process it. The next are the Socket.io events to which the Connection Microservice will subscribe:

1. Create Connection: This event will be received from the Mobile applications whenever the client wants to establish a connection to start emitting or subscribing to a set of events.

2. Send driver Location: This event will transport the driver location updates. It is one of the most frequent event overall since drivers will share their location once they become available to receive request rides and during an active ride. When drivers are only available to receive request rides, the location data will be stored in the database, whereas on active rides the location besides being updated in the database it will be shared with the customer to be able to track the driver current location. Therefore there is a communication between the Connection and the Driver Location microservices for sharing the location updates.

The following are the different Socket.io events to which the Driver app will subscribe:

1. New Available Ride: Whenever there is a ride request and a driver is available, it will receive this event and the data related to the different parameters sent by the customer to make the ride request.

2. Cancel Request Ride: In case a requested ride is no longer available because timer has expired or another driver has accepted the ride, it will trigger this event to inform the driver.

3. Cancel Ride: After a ride has been accepted and customer decided to cancel it.

4. Card Payment Result: A finished ride with card payment will execute a payment workflow in which multiple microservices will interact among them. Once the pay-
ment has been processed it will trigger this event that will inform the customer and the driver to take actions regarding the payment result.

And the following are the different Socket.io events to which the Customer app will subscribe:

1. Driver ride approval: When a driver accepts the requested ride the customer will be informed, receiving data related to the driver and starting to track the driver location updates.

2. Driver location updates: Drivers Mobile application will share their location every-time they are available online and during a ride life cycle. So the Customer Mobile application will received this location through this subscription.

3. Driver ride refused: In case all drivers refused to accept the ride, it will inform to the customer.

4. Driver starts ride: Once the request has been accepted by the driver and the customer is informed, then the driver can start the ride and this event will be sent to the customer.

5. Driver near: After a ride is accepted by the driver and he is getting closer to the destination it will inform the customer of this.

6. Driver cancel ride: In case a ride has been accepted but the driver decides to cancel it, it will inform and restore the data in the mobile application.

7. Driver finish ride: Once the driver has finished the ride, the workflow to process the payment will start, and in case it is paid by card, the customer will be informed. Later, it will inform directly the customer to rank the ride which is the last task in a ride life cycle.

8. Start card Payment: This event is triggered once the finish ride workflow is started and the payment has to be done by card, to send the payment request through the corresponding API. The data sent in this event will be the url redirecting the customer to the Payment Gateway website to perform the payment.

9. Card Payment Result: Due to the payment processing is performed asynchronously, as soon as there is a result from the Payment Gateway it will invoke a service which dispatches this event to inform the customer about the payment result and take the corresponding actions.

### 4.5.1.3 Create Connection

This endpoint stores the socket identification created by the microservice everytime a user opens a connection and the event is received by the microservice invoking this endpoint internally. In case a user has already a connection associated to its identification, then it will update it with the new socket identification. The socket connection will change everytime the connection is lost at some point, creating a new one.
4.5 Design of the Realtime Microservices

Definition:

1. **HTTP Verb**: POST
2. **Route**: /connection/createConnection
3. **Body**: User identification
4. **Response**: Status code

![Figure 17: Create Connection Endpoint](image)

### 4.5.1.4 Emit Event

The endpoint allows sending an event to the client by querying the socket identification by the user identification received as parameter with the provided data.

Definition:

1. **HTTP Verb**: POST
2. **Route**: /connection/emitEvent
3. **Body**: Event name, data and user identification
4. **Response**: Response status
4.5 Design of the Realtime Microservices

4.5.1.5 Send Notification

The endpoint permits to send a notification to the client querying the push notification token from the Notification Administration microservice and forward it to the Notification Service based on the platform.

Definition:

1. HTTP Verb: POST
2. Route: /connection/sendNotification
3. Body: Event name, data and user identification
4. Response: Response status
4.5.2 Driver Location Microservice

The Driver Location microservice manages the location tracking for the driver user, keeping up to date location coordinates everytime is received from the Mobile driver application.

4.5.2.1 Data Model

The Driver Location Data model 20 has one collection with the following fields:

1. \_id The MongoDB auto-generated primary key.
2. id The driver’s user identification.
3. name The driver’s name.
4. location The current driver’s location in GeoJson format with a substructure to store the location longitude and latitude coordinates.
5. available A driver’s flag status.
6. carType The driver’s vehicle type identification from the Administration microservices.
7. carId The driver’s vehicle identification from the Administration microservices.
8. deviceId The driver’s device identification to established uniqueness in case driver sets as available in multiple vehicles.
9. createdAt and updatedAt are Mongodb timestamps
The collection has the following indexes:

1. *location* set to 2dsphere index to support MongoDB geospatial queries (inclusion, intersection and proximity) [MongoDB, 2018].

2. *id* and *deviceId* fields define uniqueness.

3. *available, carType* and *updatedAt* to query available drivers with a certain type of car in a parameterized time boundary (2 hours by default) after *location* has been filtered.

4. *updatedAt* to clear inactive drivers.

5. *id* and *available* to query multiple vehicles from an available driver.

### 4.5.2.2 Fetch Nearby Drivers

This endpoint clears all inactive drivers that have not been updating their location in a parameterized time boundary. Then to query the right drivers a MongoDB aggregation [MongoDB, 2020a] is performed to get the maximum number of driver (at most 5 in requirements) which meet the next conditions:

1. Drivers within the parameterized radius from the customer pickup location.

2. Drivers with the required vehicle type sent from the customer on the ride request.
3. Drivers who are available to receive ride requests.
4. Drivers who have been active for the past two hours, meaning they have shared their current location.

Definition:

1. **HTTP Verb:** `GET`
2. **Route:** `/driverLocation/fetchNearbyDriver`
3. **Parameters:** Customer pickup coordinates (longitude, latitude) and vehicle type.
4. **Response:** List with all drivers with database fields.

![Diagram](image.png)

Figure 21: Fetch Nearby Drivers Sequence Diagram

The result from the previous aggregation query will be the result from the endpoint, after applying some formatting to fit the front-end.

### 4.5.2.3 Get Driver Status

This endpoint responds with the different vehicles that the driver has registered in the system.
Definition:

1. **HTTP Verb:** *GET*
2. **Route:** /driverLocation/driverStatus/:id
3. **Parameters:** Driver identification
4. **Response:** List with driver availability status and their car identification

![Diagram](getDriverStatus)

Figure 22: Get driver status Sequence Diagram

4.5.2.4 **Is Driver Active**

This endpoint just responses whether the driver is available for any vehicle.

Definition:

1. **HTTP Verb:** *GET*
2. **Route:** /driverLocation/isActive/:id
3. **Parameters:** Driver identification
4. **Response:** Driver availability status for any vehicle
4.5.2.5 Update Driver Status

The purpose of this endpoint is to update the driver status for a specific vehicle whenever his availability changes because of a starting or finished ride.

Definition:

1. **HTTP Verb:** *PUT*
2. **Route:** /driverLocation/updateStatus
3. **Body:** Driver identification, vehicle id and new availability status
4. **Response:** Driver availability status for any vehicle

Figure 23: Is Driver active Sequence Diagram
4.5.2.6 Disable Driver

Disables the driver availability for all vehicles whenever the driver goes out of service to stop receiving ride requests.

**Definition:**

1. **HTTP Verb:** *POST*
2. **Route:** /driverLocation/disable
3. **Body:** Driver identification
4. **Response:** status code
4.5.2.7 Update Driver Location

This endpoint will update driver location after the Socket.io event is received from the Driver Mobile application when they are sharing their current location. This happens as soon as the drivers become available to receive rides or during an active ride to share this location with the customer. So the event will include the location data in the body, among other identification data, which will be stored in the database.

Definition:

1. **HTTP Verb:** *PUT*
2. **Route:** /driverLocation/updateLocation
3. **Body:** Driver identification, vehicle identification, name, coordinates( latitude, longitude)
4. **Response:** status code
4.5.3 Request Ride Microservice

This microservice is in charge of receiving the Customer’s request rides, estimating its fare and managing the requested drivers to keep track of them by collaborating with the Driver Location Microservices.

4.5.3.1 Data Model

The Request Ride Data model (see Figure 27) has one collection with the following fields:

1. `_id` The MongoDB auto-generated primary key.
2. `passengerId` A unique user identification for the customer.
3. `driverId` A unique user identification for the driver.
4. `idPaymentMethod` The payment identification from the Administration microservices.
5. `idCashPaymentMethod` The cash payment identification from the Administration microservices.
6. `paymentMethod` The payment method identification from the Administration microservices.
7. `data` The ride request details submitted by the customer.
8. `passenger` The customer data.
9. `startingLocation` The request ride pickup location.
10. `destinationLocation` The request ride destination location.
11. `estimation` The fare estimation for the requested ride.
12. `startTime` Registered time to calculate the timer progress.
13. `isNotification` Flag indicating if the request ride has been reviewed (false) or just notified (true)
14. `createdAt` and `updatedAt` are Mongodb timestamps.

The collection has the following indexes:

1. `driverId` field defines uniqueness.
2. `driverId` and `passengerId` index to query by filtering driver and customer identification
4.5.3.2 Calculate Estimation

This endpoint is in charge of calculating the ride fare so that it can be invoked prior to
the ride request as an estimation and once the ride has finished to get the fare that the
customer will have to pay.

Definition:

1. **HTTP Verb**: GET
2. **Route**: /requestRide/:distance/:weight/:vehicle
3. **Parameters**: Distance, weight, vehicle type
4. **Response**: Figure with the calculation, disaggregating the figures by the extra
charges for exceeding weight, time and distance limits, the charge fee and total
amount.

```
calculateEstimation
```

Figure 28: Calculate Estimation Sequence Diagram

4.5.3.3 Create Request

This endpoint will create a ride request interacting with multiple microservices through
their endpoints. So, it can get or transform the different resources to perform the ride
request. It starts by fetching the drivers using the coordinates and vehicle type, so the
closest drivers can be tracked. Then based on the DriverLocation MS response, for each
selected driver a request database document will be created and then the request will
be sent through the Connection MS by using sockets. After all requests has been sent,
a timer will be started for 60 seconds, as the maximum time for a driver to accept the
request. In case a driver accepts the ride, it will be handled by the Ride MS. On the
other hand, in case the check ride request sent to the Ride MS reports no rides have been
created, then a socket event will be sent through the Connection MS to inform customer
that there are no drivers available.
1. **HTTP Verb:** POST

2. **Route:** /requestRide/create

3. **Body:** Customer, payment, payment type identifications, coordinates (longitude, latitude), vehicle type, request details, customer name.

4. **Response:** Number of drivers available

![Create Request Sequence Diagram](image)

Figure 29: Create Request Sequence Diagram

### 4.5.3.4 Cancel Request

In case a driver cancels the request ride this endpoint will be invoked to remove the database document for the request.
Definition:

1. **HTTP Verb:** DELETE
2. **Route:** /requestRide/cancel
3. **Body:** Customer and driver identification
4. **Response:** Status code

![Sequence Diagram](image)

Figure 30: Cancel Request Sequence Diagram

### 4.5.3.5 Request Exists

It checks whether there is a request ride in progress for the specified customer and/or driver identification.

Definition:

1. **HTTP Verb:** GET
2. **Route:** /requestRide/exists/:driverId/:customerId
3. **Parameters:** Customer and/or driver identification
4. **Response:** Request database object
4.5.3.6 Get Pending Requests

This endpoint queries if there is a pending request ride for the driver. This is useful when the driver loses his connection somehow, and it can get the request data to review the ride. In case there is a ride, before sending all the corresponding data back to the driver, it will update the IsNotification database field, which means that driver will review the data.

1. **HTTP Verb:** *GET*
2. **Route:** /requestRide/pending/:driverId
3. **Parameters:** Driver identification
4. **Response:** Request database object
4.5.4 Ride Microservice

The Ride Microservice manages the different stages from the active rides, once a Driver has accepted the Customer request. It communicates with the Administration Microservices to keep the ride up to date and process payments when it has finished.

4.5.4.1 Data Model

The Request Ride Data model (see Figure 33) has one collection with the following fields:

1. \_id The MongoDB auto-generated primary key.
2. idRide The ride identification from the Administration Ride microservice.
3. passengerId The customer identification from the Administration Customer microservice.
4. driverId The driver identification from the Administration Driver microservice.
5. startingLocation The pickup location coordinates in GeoJSON format.
6. startingPlace The formatted address for the ride’s pickup location.
7. destinationLocation The destination location coordinates in GeoJSON format.
8. destinationPlace The formatted address for the ride’s destination location.
9. currentLocation The current location coordinates in GeoJSON format.
10. estimatedDistance The estimated distance in the ride request.
11. \textit{estimatedTime} The estimated time in the ride request.

12. \textit{estimatedFare} The estimated fare in the ride request.

13. \textit{status} The status for the Ride microservice (created, started, canceled, pending for payment, pending for rating).

14. \textit{idPaymentMethod} The payment method identification chosen by the customer.

15. \textit{idCashPaymentMethod} cash payment method identification in case card payment gets rejected.

16. \textit{paymentMethod} extra information for the chosen payment method.

17. \textit{paymentRequest} card payment data from the Payment Gateway, the url and a payment reference.

18. \textit{driver} driver data to show to the customer.

19. \textit{customer} customer data to show to the Driver.

20. \textit{car} vehicle data to display to the Customer.

21. \textit{details} all the data about the ride from the request so that it is available when needed.

22. \textit{createdAt} and \textit{updatedAt} are Mongodb timestamps.
The collection has the following indexes:

1. idRide to perform queries by the ride identification and as unique criteria.
2. `driverId` and status to query the ride by the driver identification and the ride status.
3. `passengerId` as a unique key and to query the ride by the customer identification.

### 4.5.4.2 Create Ride

The driver mobile application will invoke this service to accept a requested ride by the customer. It has two conditions that must meet to create a ride: the request ride must exist, validated using an API call to the RequestRide microservice; and as second condition, a ride should not exist for the customer identification. If these conditions are met, then the ride will be created in the own microservice database and in the Administration Ride Microservice. The customer will be notified in case some requested driver has accepted the ride and also in case neither of them have, but this latter case is handled by the RequestRide microservice. In case one of the above conditions do not meet, then the ride won’t be created and the Customer and Driver will be informed.

**Definition:**

1. **HTTP Verb:** `POST`
2. **Route:** `/ride/create`
3. **Body:** Customer, vehicle identification, starting location, destination location, driver data.
4. **Response:** Ride status
4.5 Design of the Realtime Microservices

4.5.4.3 Alert Near Customer

Once the ride has started it will pass over different status, all of them are controlled by the driver. All of these state changes are updated in the microservice database and the Administration Ride microservice. The last step is to inform the customer about the ride updates. The first state change is the near alert, which driver uses to notify to the customer that he is close to the pickup location.

Definition:

1. **HTTP Verb**: \textit{PUT}
2. **Route**: /ride/alert
3. **Body**: Customer and ride identification

![Create ride Sequence Diagram](image-url)
4. **Response:** Ride status

![Diagram of alert near customer sequence](image)

**Figure 35:** Alert near customer Sequence Diagram

### 4.5.4.4 Rate Ride

This is the last step in the ride lifecycle invoked from the Customer mobile application to rank the driver service. This will be updated also in the Administration Ride microservice with the ranked value and description provided by the Customer and then it will proceed to remove the document from the Ride Database.

**Definition:**

1. **HTTP Verb:** POST
2. **Route:** /ride/rate
3. **Body:** Customer, Driver, and ride identification. Comment and rate (1-5)
4. **Response:** Response status
This endpoint is invoked by the driver after the customer has paid with a debit/credit card and the payment remains in a pending status. Since the Payment Gateway will be validated asynchronously invoking an Administration microservice endpoint, the driver will have to manually keep validating the payment with this endpoint, until the payment has been approved or rejected.

**Definition:**

1. **HTTP Verb:** *POST*
2. **Route:** /ride/verifyPayment
3. **Body:** Customer, Driver and ride identification.
4. **Response:** Payment status
4.5.4.6 Card Payment Result

In case of an online payment at the end of the ride, the customer will be redirected to the payment Gateway site to fill in the car payment form. After the payment has been processed, this endpoint will be invoked to, depending on the payment status, proceed to finish the ride or inform the Mobile applications for taking next actions (see figure 13).

Definition:

1. **HTTP Verb**: `POST`
2. **Route**: `/ride/cardPaymentResult`
3. **Body**: Ride identification and payment object.
4. **Response**: Response status
4.5.4.7 Finish Ride

Once the driver has arrived the customer’s destination, this endpoint will be invoked triggering the payment workflow in the case of debit/credit card payment or just updating and informing the customer for cash payment. The Ride Administration microservice has its own business logic when finishing a ride, so it is necessary to invoke it from this endpoint.

Definition:

1. **HTTP Verb:** POST
2. **Route:** /ride/finish
3. **Body:** Customer, Driver, ride, payment method, payment reverse identifications.
4. **Response:** Payment status
4.5.4.8 Cancel Ride by Driver

The Driver can cancel a not started ride. It will update the new status in the Ride Administration microservice, update the driver availability and then inform the customer about the cancelation.

Definition:

1. **HTTP Verb:** *POST*
2. **Route:** /ride/cancelByDriver
3. **Body:** Customer, Driver and ride identification.
4. **Response:** Response status

![Figure 40: Cancel ride by Driver Sequence Diagram](image)

### 4.5.4.9 Cancel Ride by Customer

Also the customer can cancel a not started ride. It will update the new status in the Ride Administration microservice, update the driver availability and then inform the driver about the cancelation.

**Definition:**

1. **HTTP Verb:** *POST*
2. **Route:** /ride/cancelByRider
3. **Body:** Customer, Driver and ride identification.
4. **Response:** Response status
4.5.4.10 Start Ride

Once the driver has picked up the customer in the specific location, he will invoke this endpoint to update the status in the Administration Ride microservice and in the Real-time Ride microservice database, and finally will notify the customer about the update.

Definition:

1. **HTTP Verb**: *POST*
2. **Route**: /ride/start
3. **Body**: Customer and ride identification.
4. **Response**: Response status
4.5.4.11 Resume Ride Driver

In case the connection is lost, the driver can get the latest ride data invoking this endpoint which will query the database and respond.

Definition:

1. **HTTP Verb**: *GET*
2. **Route**: /ride/resumeDriver/:driverId/:status
3. **Parameters**: Driver identification and ride status.
4. **Response**: Ride database Object
4. Development

4.5 Design of the Realtime Microservices

4.5.4.12 Resume Ride Customer

Likewise, the customer, in case the connection is lost, can get the latest ride data invoking this endpoint, which will query the database and respond.

Definition:

1. HTTP Verb: GET
2. Route: /ride/resumeCustomer/:customerId/:status
4. Response: Ride database Object

4.5.4.13 Ride Report

This endpoint will get all the active rides from the database and the available drivers with their location from the corresponding microservice, to set up a complete report of the current operation.
1. **HTTP Verb**: GET

2. **Route**: /ride/report

3. **Response**: Active rides list

![Ride Report Sequence Diagram]

**Figure 45**: Ride Report Sequence Diagram
5 Testing

5.1 Mobile Applications

For testing some of the exposed endpoints from the Connection, Driver Location and Request Ride microservices through the API Gateway, two very basic mobile applications were implemented to test connectivity and data display. The communication was tested using the fetch Web API [Webdocs, 2020] to invoke the API Gateway endpoints for the different microservices. Figure 46 shows the ride configuration. A form is used to setup the variables needed to estimate the ride for the customer mobile application.

![Image of the mobile application interface showing ride configuration]

Figure 46: Customer Mobile Application: Ride Configuration

After the pickup and destination location is setup using a Google Map and Geocoding service to translate places to coordinates, all the necessary variables are sent for the fare estimation using the `calculateEstimation` endpoint from the Request Ride Microservice and displaying it.
In case of the Driver app the first tested endpoint is the `updateDriverStatus` endpoint request. The socket connection is used to share the location every 5 seconds and subscribe to the request ride event in case a customer sends one.
The last test with the Mobile apps was to send a ride request from the Customer mobile application, which gets to the Request Ride microservice through the `createRequest` endpoint, and then the Request Ride using the `emitEvent` from the Connection Microservice, which will send the event to all the available drivers, as shown in the next graphic:
5.2 Web Application

A React web application was implemented with the purpose of testing Socket.io connection with the Realtime microservices that receive the most workload: The Connection and Driver Location Microservices. As can be seen in the figure 50, it has a form to configure the number of connections, the interval between events and the emitted event name. Each field will be explained in detail next:

1. **Number of Connections**: It is the number of concurrent Socket.io connections used for sending events with the purpose of emulating a mobile application for each connection.

2. **Interval between connections**: Time between emitted events in seconds, each connection will use this interval for sending.

3. **Emitted events**: The emitted event name that will be sent to the real-time microservice.
Once the connection form is configured and started, it will begin to establish the defined number of connections sending events for each of connection with the frequency set up in the interval field. This will emulate multiple devices connected to the Realtime microservices sharing the location in the case of figure 50.

5.2.1 Architecture and Configuration

The overall architecture can be seen in the next diagram:

![Diagram](image)

Figure 51: React Web application

There is a WebSocket connection protocol between the Real-time Frontend Tester to the API Gateway, followed by a routing to the Connection microservice. This routing is configured through Kubernetes Ambassador API Gateway with the Service resource [Kubernetes, 2020c] with the next fragment taken from the configuration of the Connection microservice:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: connection-service
  namespace: real-time
  annotations:
    getambassador.io/config: |
      ---
      apiVersion: ambassador/v1
      kind: Mapping
      name: connection-service_mapping
      prefix: /real-time/connection/
      service: connection-service.real-time:3000
      use_websocket: true
```
Therefore, from the client the WebSocket connection will be established to the domain using the route /real-time/connection/, which is where the Connection microservice will be receiving all incoming requests, including WebSockets from Socket.IO, as the use_websocket is active in the configuration. Consequently, the frontend application and the Connection Microservice will integrate the corresponding Socket.IO library [SocketIO, 2020] for the client and the server side to be able to communicate.

For the scaling the next configuration file was used so a minimum of 1 replica was provisioned and a maximum of 5 with an 80% CPU utilization threshold for deploying new instances.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: connection-autoscaling
spec:
  maxReplicas: 5
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: connection-deployment
  targetCPUUtilizationPercentage: 80
```
The cluster was set up with five nodes, using the *Standard_B2* virtual machine size for each of them the low-cost Azure VM family, provided with 2 virtual CPU cores, 4GiB of memory and 8GiB local SSD. Once the kubernetes environment has been set up and the microservices have been deployed first as images using Docker [Docker, 2020a], then, by using the kubernetes command-line tool kubectl, we executed the Ambassador API Gateway Service, the Connection and DriverLocation Microservices as Kubernetes Services considering the autoscaler configuration for each microservice. In the Kubernetes Dashboard [Kubernetes, 2020d], all resources are shown as follows:

<table>
<thead>
<tr>
<th>Node</th>
<th>Status</th>
<th>Version of Kubernetes</th>
<th>Type ofSD</th>
<th>Node of node</th>
<th>Tamanio del nodo</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALM</td>
<td>Successful</td>
<td>1.22.1</td>
<td>Linux</td>
<td>1</td>
<td>Standard 1.22</td>
</tr>
</tbody>
</table>

Figure 52: Azure Kubernetes Services
As can be seen in Figure 53, the 5 nodes were registered in the cluster to run pods by the Kubernetes Orchestator Load Balancer. In Figure 54 the different deployment resources are shown including the Driver Location and Connection Microservice, starting with one Pod for each of them.
5.2.3 Database

For the Database the MongoDB Atlas service [MongoDB, 2020d] were used to test the Database connection and operations. However, as MongoDB Atlas free tier was used for testing, only a single database was used for testing both microservices. So, the Sharding feature was not enable and the virtual machines for the MongoDB Engine were shared, providing very limited resources. Therefore, the focus of these tests was more on the behavior of the Kubernetes environment than on the MongoDB behavior.

The Database collections are shown next:

![Kubernetes Dashboard Overview](Figure 55: Kubernetes Dashboard Overview)

5.2.4 Test Plan

The test plan was designed to track system availability and scaling after receiving a controlled amount of traffic that was increased gradually, analyzing CPU, Memory and Kubernetes resource usage for each test case. This is also known as Loading Testing [Jiang and Hassan, 2015] to determine the limits of the current solution and identify some useful metrics that can be analyzed. Therefore, the test plan was split into three phases: Design, execution and analysis of the results. The next are the different test cases that were designed and executed using the architecture described in Figure 51 with a 3 second interval to send socket events:

1. 20 active connections
2. 100 active connections
3. 500 active connections
4. 1000 active connections
5. 1500 active connections
6. 2000 active connections

Each connection emulated the driver mobile application sharing its location once it becomes available to receive request rides for the customer, in the Software Requirement
Specification the interval was set to 5 seconds, but as the purpose was to stress system, it was changed to a higher frequency of 3 seconds. The 20 and 2000 connections results are shown next, followed by an overall graphic.

20 Connections: For this test case the client sent 2380 socket event for the 20 connections, as can be seen in Figure 56.

![RealTime Tester](image)

**Figure 56: 20 Active connections Front-end**

Looking at the Kubernetes Dashboard for the Connection Microservice Deployment, there is only one active pod with a 9% CPU utilization, the maximum CPU usage was 0.026 cores.

![Kubernetes Dashboard](image)

**Figure 57: 20 Active connections Connection Microservice**

For the Driver Location Microservice Deployment, the CPU was 10% for one active pod with a 0.027 CPU cores usage.
500 Connections: In this test the client sent 85500 socket event for the 500 connections, as it is shown next in Figure 59.

As can be seen in Figure 60, Connection Microservice scaled to 2 instances with a current CPU utilization of 76%. The CPU usage graphic shows that it is below 0.5, which is the threshold for CPU usage. This figure is calculated out of the 50% core limit for the
requests in the Deployment configuration and the 80% set in the Horizontal Pod, resulting in 0.4 Core usage for the threshold. In the graphic CPU usage slightly exceeds the limit of 0.4. This happens because the frequency between events sent from all connections is high and the Kubernetes needs some time to diagnose CPU usage and scale. This also will happen in the 2000 connections test case described below.

Figure 60: 500 Active connections Connection Microservice

For the Driver Location Microservice Deployment, the CPU was also 76% and it scaled also to 2 instances. The CPU usage distribution is very similar to the connection Microservice, showing the strong relation between both microservices.
2000 Connections: For the test case with the most active connections, the Front-end application was ran using two computers each with 1000 connections sending the same type of event. Both clients sent a total of 948000 events for the 2000 connections, as shown in the figure 62 and 63.
In the next Kubernetes Dashboard for the Connection Microservice, the pod deployment scaled to 3 replicas, after exceeding the 80% threshold for the CPU utilization. Looking at the CPU Usage chart, there can be seen three pikes: the first when the deployment scaled from 1 replica to two, and the second from 2 to three replicas, with a maximum of 0.7 cores usage. The memory went to a maximum of 954 MB which is understandable since system needs to keep 2000 connections alive. Although the threshold is close to 0.4 CPU core usage, as in the 500 connections test, this time also exceeded the boundary with an even higher value, almost twice the limit. This also happens because the microservice is receiving a very large workload, creating a delay for the Kubernetes Pod Autoscaler to react.

In the case of the Driver Microservice, the total number of pod replicas was 4, after exceeding the 80% threshold for the CPU utilization. The CPU chart shows not very...
defined pikes for every time it scaled up, at least the last two pikes are very clear where the deployment went from 2 replicas to 3 and from 3 to 4 in the last pike. At the end the CPU utilization with the 4 replicas was with 25% usage, that’s why it started to decreased CPU cores usage.

![Figure 65: 2000 Active connections Driver Microservice](image)

## 5.3 Results

The next are the different metrics calculated after performing the different test cases and analyzing the results:

### 5.3.1 Number of Pods per Socket Connections

In Figure 66 the different number of replicas scaled during testing for the different connections. For the Connection and Driver Location Microservices, the number of replicas by default was 1, after 500 socket connections both microservices scaled to 2 replicas, followed by 3 for 1000 and 1500 connections. The Driver Location scaled up to 4 replicas to handle 2000 connections whereas Connection Microservice remained with 3. This result suggests that the Driver Location Microservice received larger workload than the Connection Microservice.

This could happen because the Driver Location Microservice has more database intensive operations, since it needs to update the location every time the Connection Microservice sends the request. Although the Connection Microservice also writes in the Database, it happens with a lower frequency, since this is done for creating/updating Socket connections only.
5.3.2 Maximum CPU Core Usage per Socket Connections

The distribution of Figure 67 is very similar to the previous one, because the scaling configuration receives as input the CPU usage. So, the more CPU cores the microservice uses the more likely to scale up. Although the Connection Microservice seems to use more CPU cores than the Driver Location Microservice, this is just in average. So, the Driver Location microservice received pikes that slightly exceeded CPU 70% threshold, making it scale up on the 2000 connections while the Connection microservice stuck to 3 replicas.

5.3.3 Memory Usage per Socket Connections

The memory usage for the Driver Location Microservice was higher than the Connection Microservice after 100 connections, as can be seen in Figure 68. This also could happen
because Driver Location Microservice is more write intensive Database operations making it store more data in memory related to these write operations. And as some of these operations failed because the Database went unavailable for some time, it kept retrying writes.

![Maximum Memory Usage Connection MS, Memory Usage Driver Location MS vs Socket Connections](image)

Figure 68: Memory usage per Socket Connections

### 5.3.4 CPU Cores per Connections

In Figure 69 the overall CPU cores usage can be seen for all connections. It is very interesting that the CPU usage increased with a high slope from 100 to 500 connections, when it scaled up the first time from 1 to 2 pods for the Connection and Driver Location Microservice. Then it kept increasing at a lineal way. This could occur because the workload was delivered suddenly in the configuration by default and the microservices could not scale up as fast as needed to avoid to reach the 70% CPU usage.

![Maximum CPU Cores vs Socket Connections](image)

Figure 69: CPU Cores per Connections
6 Conclusion and Future Work

After applying the microservice architecture, the proposed system reflects a decomposed architecture, which is loosely coupled, providing single responsibility for each microservice. This allows each microservice to be maintained in isolation, avoiding a single point of failure as in monolithic architecture, since each microservice will be deployed independently and will have its own resources. Then each of them will be responsible for complying with the functional and not functional requirements. Although special consideration needs to be given to microservices that communicate between them, since they will have a resource dependency. So, the microservices that have this dependency will have to be maintained in group to keep performance and availability.

In the case of availability, microservice resilience is a key issue, because if the failover, replication and monitoring are managed correctly, the microservice will have a high availability. So, the deployment environment is essential to support these features along with scalability.

To get a high scalability, resource allocation is very important, being the load balancer the component in charge of this task, splitting microservice workload among system resources, and being responsible for managing infrastructure resources.

Performance is very dependent on scalability, supposing that the microservices are well implemented, since a good allocation of resources is essential for not degrading the service when traffic pikes are received.

All these quality attributes depend on how well the architectures of the microservices independently and as a whole are designed. Therefore, it is very important to deeply understand the system to prioritize microservice identifying the most essential for the core business functionality. Since those microservice will receive the most traffic and need to be resilient enough to bear the workload. There will always be microservices with less workload than others, and microservice which depend on others to provide their functionality, creating a bound that can impact their performance.

As a huge benefit a different technology stack can be applied in each microservice, as it was done for the Real-time and Administration microservices, allowing us to use the most suitable technologies for covering the system requirements. For DevOps this pattern also facilitates fully automated deployments reducing complexity and increasing maintainability, since each microservice is managed like a single project, being portable and reusable.

As any other pattern, microservices pattern also has some drawbacks such as the development is more complex than in a monolithic application, so the planning and design are a key issue for a successful architecture.

Also, the fact that multiple technologies can be adopted can force development team to work on multiple languages and frameworks, which some companies cannot afford.

Testing is also a challenging task for a microservice architecture, because monitoring each of them requires specialized tools for tracking them independently and as a whole in case of internal communication.

Achieving data consistency is more complex than in a monolithic architecture, since each
microservice has its own database, and if the traffic is high like in this project, there will be a need to implement a distributed database. Therefore, ACID transaction will require extra concerns such as adding database implementation patterns and extra complexity when data modeling.

Fortunately, Kubernetes covers the majority of the previous described issues. This system is able to automate the operational tasks using declarative configuration files. So, complex tasks related to scalability, load balancing, service discovery, API Gateway and monitoring allow us to deploy an elastic and fully automated system without too much effort. Kubernetes also gives a self-healing capacity to the pods inside a cluster, so recoverability is defined after declarative configuration files guaranteeing a number of pod instances running.

One of the most challenging aspects from the project was the testing phase. The purpose was to emulate a production environment for the microservices to analyze its behavior after stressing the system. But the proposed architecture requires a very specialized setup for Azure and MongoDB Atlas. This setup is very costly, so the testing had to be done with trials and testing subscriptions, which at the end couldn’t reflect the production environment as intended. Especially with the database limitation, which became a bottleneck during testing, adding a performance overhead in the most stressful part of the test, crashing the database and forcing the system to make retries. However, the final result was promising, because we built an elastic solution that can be easily configured based on the system requirements without extra coding effort. The proposed architecture and deployment can easily be used for a low or high workload after user demand, making scalability one of the greatest benefits for a company that is starting with the platform and in the future may grow without too much effort for adapting the current software solution.

When analyzing the microservice business logic decomposition, each microservice provided a single responsibility in the cluster showing a low coupling which was one of the main purposes for choosing the Microservice Architecture. Although we could identify a very Probably the strongest dependency is from the Ride, Request Ride and Driver Location Microservices to the Connection Microservice. The purpose was to delegate the socket.io and notification responsibility to a unique microservice in charge of handling all the communication. However for the other microservices is very frequent to send events and push notifications increasing the workload that the Connection microservice receives. Perhaps it will be better to implement a Bus where multiple microservices can connect with to send/receive events and notifications on demand without delegating this responsibility to a single Microservice. This will be implemented for future work as a proof of concept to test if the Communication Microservice responsibility can be broken down into a component in charge of coordinating communication following a Broker pattern.

For future work also the Administration and Real-time back-ends are going to be deployed on the same Kubernetes cluster. This will allow us to continue with the integration testing and carry out the Loading testing to test the scalability, performance and availability of the whole solution. It is important also to get closer to a real production database platform, because it was a limitation in the testing phase of Real-time back-end. In this way, with a better MongoDB cluster tier, we will benefit from the sharding features and more resources from the virtual machine. The main goal is to bring the closest to a production environment, so we can test all the non-functional requirements. And based on
the results applying the necessary re-configurations to comply with these requirements.
References


References


