Rafael
Oliveira

Aplicações de IoT No Contexto de uma Cidade Inteligente

IoT Applications in a Smart City Context

Universidade de Aveiro
2021

Rafael
Oliveira

Aplicações de IoT No Contexto de uma Cidade Inteligente

IoT Applications in a Smart City Context

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica do Doutor João Paulo Silva Barraca, Professor auxiliar da Universidade de Aveiro, e do Doutor Joaquim José de Castro Ferreira, Professor adjunto da Escola Superior de Tecnologia e Gestão de Águeda.

o júri / the jury

presidente / president

Professor Doutor Arnaldo Silva Rodrigues de Oliveira
Professor Auxiliar da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Sérgio Armindo Lopes Crisóstomo
Professor Auxiliar da Faculdade de Ciências da Universidade do Porto

Prof. Doutor João Paulo da Silva Barraca
Professor Auxiliar da Universidade de Aveiro

agradecimentos / acknowledgements

Aos orientadores pela ajuda nesta difícil etapa. Aos meus colegas e amigos pelo apoio ao longo destes incriveis anos. Á minha familia que me suportou e acreditou nas minhas capacidades durante todo este tempo. Obrigado a todos que fizeram parte desta fase difícil mas ao mesmo tempo fantástica. Este trabalho é financiado pela FCT/MCTES através de fundos nacionais e quando aplicável cofinanciado por fundos comunitários no âmbito do projeto UIDB/50008/2020-UIDP/50008/2020, assim como pelo Fundo Europeu de Desenvolvimento Regional (FEDER), através do Programa Operacional Regional do Centro (CENTRO 2020) do Portugal 2020 [Projeto PASMO com o nº 000008 (CENTRO-01-0246-FEDER-000008)].

Palavras Chave cidade inteligente, iot, pasmo, plataforma, serviços publicos, fusão sensores

Resumo

Nos últimos anos, as soluções Smart City amadurecem muito rapidamente em conjunto com IoT e serviços na cloud. Estas tecnologias facilitam a criação de serviços e a incorporação de aplicações direcionados á melhoria da qualidade de vida do cidadão, oferecendo formas das empresas implementarem suas soluções. Por meio de rápidos avanços na qualidade dos sensores, novos métodos surgiram, combinando diferentes tipos de dispositivos para criar uma melhor imagem da realidade. O objetivo desta dissertação é fornecer informações úteis através de serviços públicos, que podem ser acedidos por pessoas que visitam ou residem na Costa Nova e Barra. Também fornece uma solução para o problema de classificação de tráfego que projetos baseados em dados de radar tendem a enfrentar. Estas aplicações beneficiam dos dispositivos implementados no projeto PASMO, como sensores de estacionamento, radares e câmeras de CFTV. Ao disponibilizar os serviços publicamente, as empresas têm a oportunidade de construir as suas próprias aplicações em cima destes, usando os dados dos sensores sem estar diretamente conectado ao armazenamento de dados. O exemplo desenvolvido nesta dissertação oferece uma experiência de dashboard onde os utilizadores podem navegar por gráficos que fornecem uma variedade de dados e mapas em tempo real. Também fornece uma API pública que os investigadores e empresas podem usar para desenvolver novos aplicativos no contexto do PASMO. A outra área abordada neste documento é a classificação de tráfego. Embora os dados fornecidos sejam confiáveis, um grande problema provém da precisão da classificação dos veículos fornecida pelo radar. Ainda assim, este dispositivo oferece valores precisos quando se trata de detecção, com as câmeras fazendo um bom trabalho na parte de classificação do tráfego. O objetivo é combinar estes dois dispositivos para apresentar informações corretas, usando algoritmos de detecção de objetos e métodos de fusão de sensores. No final, o sistema irá enriquecer o projeto PASMO, tornando seus dados facilmente disponíveis ao público e corrigindo problemas de precisão de alguns dispositivos.

Keywords smart city, iot, pasmo, plaftorm, public service, sensor fusion

Abstract

Over the last few years, Smart City solutions mature very rapidly alongside IoT and cloud computing. These technologies made it easier to create services and incorporate applications devoted to improving citizen's quality of life and offer ways for businesses to implement their solutions. Through rapid advances in the quality of sensors, new methods emerged, combining different types of devices to create a better picture of the environment. The purpose of this dissertation is to provide useful information thought public services, that can be accessed by people visiting or residing in the beach area of Costa Nova and Barra. It also provides a solution for the traffic classification problem that projects based on radar data tend to face. These applications take advantage of the devices implemented in the PASMO project, such as parking sensors, radars, and CCTV cameras. By making the service public, businesses have the opportunity to build applications on top of it, utilizing the sensor data without being directly connected to the data storage. The example developed in this dissertation offers a dashboard experience where users can navigate through charts that provide a variety of data and real-time maps. It also provides a public API that researchers and businesses can use to develop new applications in the context of PASMO. The other area tackled in this document is traffic classification. Although the data provided is reliable for the most part, one big issue is the accuracy of vehicle classification provided by the radar. Still, this device offers precise values when it comes to detection, with the cameras doing a good job in classifying traffic. The goal is to combine these two devices to present much precise information, using state-of-the-art object detection algorithms and sensor fusion methods. In the end, the system will enrich the PASMO project by making its data easily available to the public while correcting the accuracy problems of some devices.

Contents

List of Figures

List of Tables

Glossary

API Application Programming Interface
AP Average Precision
BoF Bag-of-Freebies
BoS Bag-of-Specials
cuDNN CUDA Deep Neural Network
CRF-Net Camera Radar Fusion Net
CPU Central processing unit
CIMI Cities in Motion Index
CCTV Closed-circuit television
CCOC Cloud City Operations Center
CSV Comma-Separated Values
COCO Common Objects in Context
CNN Convolutional Neural Network
EU European Union
XML Extensible Markup Language
FPS Frame Per Second
GraphQL Graph Query Language
GPU Graphics Processing Unit
GMT Greenwich Mean Time
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure
ILSVRC ImageNet Large-Scale Visual Recognition Challenge
IT Information Technology
ICT Information and Communications Technology
Inteli Inteligência Em Inovação, Centro De Inovação
IBM International Business Machines
IoS Internet of Services
IoT Internet of Things
IoU Intersection over Union
JSON JavaScript Object Notation
LED Light-emitting diode
M2M Machine-to-Machine
mAP mean Average Precision
MME-YOLO Multi-Sensor Multi-Level Enhanced YOLO
NEC Nippon Electric Company
OBU On Board Unit
OS Operating System
PAN Path Aggregation Network
PASMO Plataforma Aberta para o desenvolvimento e experimentação de Soluções para a MObilidade
PIH Porto Innovation Hub
RSS Really Simple Syndication
R-CNN Regions with Convolutional Neural Networks
REST Representational state transfer
RSU Road Side Unit
SOA Service Oriented Architecture
SOAP Simple Object Access Protocol
SPA Single Page Application
SSD Single-Shot Detector
SCoT Smart Cloud of Things
SDK Software Development Kit
SPP Spatial Pyramid Pooling
TensorRT Tensor Runtime
tkDNN toolkit for Deep Neural Network
TLS Transport Layer Security
TRUST Transportation and Road Monitoring System for Ubiquitous Real-time Information Services
URI Uniform Resource Identifier
URL Uniform Resource Locator
UI User Interface
VM Virtual Machine
WSGI Web Server Gateway Interface
WSDL Web Service Description Language
W3C World Wide Web Consortium
YOLO You Only Look Once
YOLO9000 You Only Look Once 9000
YOLOv3 You Only Look Once version 3
YOLOv4 You Only Look Once version 4

CHAPTER 1

Introduction

Vehicular services provide tons of information about the traffic and consequently the city. These applications help people to have a better understanding of the world around them through interactive and informative user interfaces. Not only that, but traffic data can demonstrate people's behavior, being helpful to not only the council of the city but the businesses in the area.

1.1 Context and Motivation

Throughout the last decade, the word "smart" as being used to describe devices that provide information about the environment and help users with regular daily tasks. This simple concept of the technology being "smart" was used by companies throughout the years, mainly to market their product. When searching for the interest of the word "smart" on Google trends, a spike in interest can be seen in June of 2008, precisely when Apple introduced the iPhone 3G. This event marked a change in the cellphone world since this product had the possibility of receiving emails and viewing full internet pages using a touch screen. Fast forward to today, and the word cellphone was replaced by smartphone. The consumer started investing more in these small devices since they have the appeal of being able to do tasks similar to a computer.

Following this investment by consumers comes more research, not only for smartphones but for other smart devices that can be used in separate fields. By the mid-2010s, the Internet of Things (IoT) technology started to gain interest, being used to tackle challenges that require tons of data from plenty of devices. This technology has become the central building block for smart cities owing to its potential in exploiting information and communication technologies [2]. Thus, the rise in research on the subject of Smart Cities can be related to the popularity of IoT. This improvement in research came to address critical issues, such as mobility, energy, and infrastructure. These problems started to appear with the increase of population and, most importantly, urbanization of modern cities since the beginning of the millennium [3]. Nowadays, the widespread of the internet made it easier to share data and services with the world. Moreover, it can transfer this information fast and reliably because of all the internet

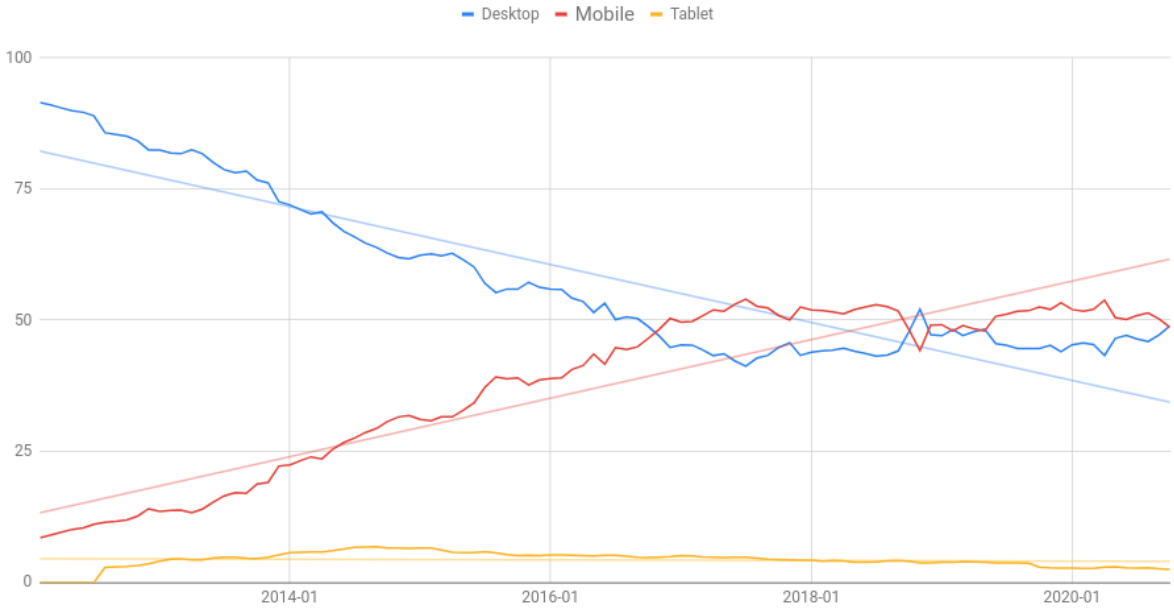


Figure 1.1: Comparation of internet traffic between desktops, mobile phones and tablets [1]. The stats show that the majority of internet traffic comes from mobile devices, with the tread revealing that these will probably continue the lead.

coverage available. Smart Cities use it to established communication between devices and publish their services to the public, thus helping users to understand the environment around them.

1.1.1 PASMO

This dissertation is part of a project called PASMO (Open Platform for the development and experimentation of Mobility Solutions). This concept was developed in Instituto de Telecomunicações - Universidade de Aveiro with the purpose to provide solutions for intelligent mobility problems and an open platform for researchers to develop their ideas. Companies can experiment with PASMO, using technologies and data that they may not have access to. This collaboration exists to test those technologies and validate equipment, applications, and services. It also strengthens the position of companies, with high innovation potential, in new markets. [4]

PASMO is composed of three subsystems, reflecting the different layers of intelligent transport systems:

- Data collection from roads, public spaces, and vehicular resources.
- Data transfer from telecommunications and data infrastructure.
- Applications.

This dissertation will focus on the applications and data collection layers, correlating them to build valuable services. The latter is composed of three resources that provide different types of information. The road resources holds an infrastructure of Road Side Units (RSU), built to provide support for wireless vehicular communications and to allow the installation of modules like radars, environmental sensors, or CCTV cameras. Having all these devices in one

unit is of interest since we can use the sensors for the function that they were designed to serve. The public spaces contribute to the well being of citizens, providing Internet access, weather stations, and smart parking sensors. The last two give the tourists and residents information about the beachside so that frustrating occurrences such as searching for a parking spot and bad weather can no longer be an inconvenience. It also provides a testing site for companies to validate their new applications. Finally, the vehicular resources that offer support to vehicular communication. To take advantage of this component, an On Board Unit(OBU) needs to be installed in the vehicle if it doesn't have it. Since a small fraction has this technology, PASMO grants access to a portable, compact and robust OBUs, able to be used in many types of vehicles. This unit will allow vehicle-to-vehicle and vehicle-to-RSU communication. This resource can provide alerts of events occurring on the road, informing the user about an accident, for example.

The equipment is installed in the municipalities of Ílhavo and Aveiro in strategic locations to capture the most valuable information about the environment around the beaches of Barra and Costa Nova. Some of it is installed on the regions (parking sensors), while others (RSUs) are on the highways and roads accessing the geographical areas.

1.1.2 Motivation

Even though PASMO's architecture is well established when it comes to the communication between devices, it lacks services in the application layer, which is essential to provide researchers and users with data from the sensors. This last layer is the most important to establish a Smart City solution because if the data is collected and not provided to the public, it just stays stored without granting any benefits to the city. This dissertation follows the central philosophy of the project, provide services so that researchers can test their technologies. Publicizing these applications built around the data from the sensors and providing easier access to it drives companies to work with these services and develop their solutions. Although the application layer has the goal to motivate researchers to use the services, it also offers regular users applications to expand their knowledge about the regions. Applications are useful to improve the quality of life of the residents and the traveling experience of the tourists.

When discussing the data collection layer, many of the devices already installed are not being utilized to their fullest potential. The CCTV cameras fall in this category, where they are sometimes used to troubleshoot radar failures but never applied in a solution. One of the problems where these devices can shiny is on traffic classification because of the inferior accuracy provided by the radars. By utilizing the cameras, the information provided by the data collection layer can become more accurate, painting a more beneficial picture of the environment.

1.2 Objectives

The goal of this dissertation can be divided into three essential services. The public services belong to the application layer, using data collection to get the necessary sensor data. Data from two resources were utilized, namely roads and public spaces, to provide traffic and

parking, respectively. The last service fits in the data collection layer and provides a road resource, offering information about traffic classification. In the end, the system will provide:

- An Application Programming Interface(API) with traffic and parking data.
- A dashboard that gets information from the above service and displays it in a user-friendly way.
- A set of algorithms that offers better traffic classification by utilizing data from the road resource.

The API can be accessed by the public and provides an easy way to use the data collected from the radars and parking sensors. Here it's crucial to develop an easy to use application with useful information about the environment. This public service will feed a web application developed with users and system admins in mind. This app has to be a highly interactive, informative, and user-friendly dashboard. When implementing these two services, it's essential to focus on the use-cases of the public and researchers since they are the target audience. Lastly, it's necessary to implement a state-of-the-art detection algorithm that classifies traffic and helps to produce accurate traffic information by using the camera's images. The objective is to provide a reliable way to implement an object detection algorithm inside the RSU so that there are no delays. By developing these three services, especially the API, PASMO will have a solid and straightforward way to provide sensor data to the public. It will give users the ability to know the state of the environment around the beach area through the dashboard. Moreover, it offers administrators an easy application to check the state of the sensors. Finally, it also improves the data collection layer by increasing the accuracy of traffic classification, together with a simple web application for diagnosing the results.

The challenge of this dissertation has a focus on managing and presenting data, as well as full-stack solutions, image processing, and sensor fusion.

1.3 Contributions

For this dissertation, two main contributions were developed and deployed publicly, the API and the dashboard.

The solution for providing users with an easy way to get and manipulate sensor data came in the form of a public API. This service provides data from a laboratory weather station to projects related to environmental monitoring. The Transportation and Road Monitoring System for Ubiquitous Real-time Information Services(TRUST) project aims to develop a meteorological monitoring system capable of identifying and alert risk conditions. This project is part of the SARWS project that is an European EUREKA initiative that focuses on the development of real-time location-aware road weather services. The API service is also used in the official website for the PASMO project [5], where is mainly used in a real-time map with traffic information.

The public dashboard offers users the ability to view the collected sensor data, as well as other information created by using it. It also gives administrators a user-friendly platform to check for possible sensor/service failures. A summary of the complete dashboard was deployed in the official PASMO website, with a hyperlink to the full service.

1.4 Dissertation Structure

The document starts with the Introduction, the current chapter, presenting the context and motivation behind this dissertation. It also clarifies the goals of the dissertation, its structure, and contributions that derive from solutions to the problems discussed. The second chapter, State of the Art, presents a review of the literature related to the topics of this dissertation. It is divided into several sections that provide history on various topics, such as Smart Cities, web services, cloud computing, and object detection. It also discussed several solutions implemented in these domains, beginning with different implementations of Smart City solutions in Europe, followed by technologies used in web services, object detection, and sensor fusion. The third chapter, Requirements and Solution Architecture, describes the scope of the PASMO project, detailing its structure, the location of the sensors, and the data they produce. It also outlines the requirements needed to develop a reliable platform before presenting the solution for the system. Chapter four, Implementation, details the development process, along with all the technologies and methods used to produce the most reliable and efficient solution. The fifth chapter, Results, presents the results of the developed work, discussing usability, efficiency, and accuracy of the whole system. In the final chapter, Conclusion, the document closes with an overall discussion of the solution implemented, the challenges that appear along the way, and future work that can further improve the system.

CHAPTER 2

State of the Art

2.1 Smart Cities

Since the 2010s, when the Smart City concept started to get more researched and, consequently, more popular, investigators from different areas argue about a definition for this concept. Some researchers focus on reaching a concrete answer by compiling several definitions given by different entities, in the case of [6] more than 20. This discussion happens because researchers have distinctive views on what makes a city smart. Furthermore, not all cities suffer from the same problems or even the same domain of issues, making the approaches used significantly different. Even though it seems that researchers can not come up with a definition that suits everyone, it is agreed that for a city to be smart, it needs to have services acting on a problem. IBM had an early entry in the Smart City area, with a talk, A Smarter Planet: The Next Leadership Agenda, dating back to 2008, offering some understanding on what a Smart City needs as a core. The basis of the concept is summarized in marketing language as the three Is: Instrumented, Interconnected, and Intelligent [7]. The first one means the ability to capture live data through the use of several types of sensors. Interconnected means that the various services communicate between themselves, integrating the data collected. Finally, Intelligent refers to the incorporation of analytics, optimization, and visualization of services to make better decisions.

At its core, the main goal of a Smart City is to collect data, using several types of sensors, and build services on top of this information to solve specific problems. The best way to achieve this objective is to provide businesses and researchers with the data collected so that they can build platforms and offer services that may help the daily life of the citizens.

2.1.1 Smart City Implementations in Europe

Even though the Smart City concept provides excellent solutions for urban problems, it's the city council that decides if the investment in these technologies is worth it or not. The answer was made apparent by the digital revolution that provides efficient solutions to urban systems by utilizing new technologies and infrastructures. The change in mindset around

the beginning of the 2010s to use technologies to attack urban problems influence various industry leaders. Business opportunities appear in multiple fields, such as ICT, environmental concerns, education, and tourism.

The European Union (EU) primary goal is to achieve highly efficient and sustainable cities while still increasing economic growth, thus addressing and solving social and environmental concerns. To speed and improve the development of new solutions, the cities in the EU focused on two critical ideas: Smart City Clusters and Smart districts serving as Living Labs. The first approach establishes the belief that with the entity's cooperation, we reach the goals quicker and develop a product with better quality. The alliance between private companies and institutions enhances business performance, resource efficiency, economies of scale, and new opportunities [8]. Using this approach ensures that each problem is being addressed by the best organization, promoting an agile and flexible development model. Nevertheless, these entities will need to implement their services in the city at some point. However, before taking this big step, Smart districts provide a research-oriented place where solutions can be designed, developed, and tested. For the most part, these are buildings that can be radically changed to understand the impact of these modifications, acting as "incubators for a Smart City." [9] Since these structures act as a development and testing center, they are regarded as living labs. The Europen Commission defines Living Labs as a "user-centered, open innovation ecosystems based on a systematic user co-creation approach integrating research and innovation processes in real-life communities and settings". [10] Here, citizens and users of the services are encouraged to take part in the development and testing process. Not only this strategy builds relationships between stakeholders but also creates user-driven services. The latter assures that user's needs are met because they are providing feedback to create a better product, almost erasing the risk of the users not liking and not using these services.

Although necessary to understand the concepts used in Smart Cities, it's crucial to study good practices when implementing these systems. Selecting the best practices relies on not only choosing the most suitable technology but also solving the most critical problems. An excellent example of using the best methods can be seen when studying Barcelone, Spain. Barcelona is in the top three smartest cities, according to a report [11] carried out by Philips and SmartCitesWorld. This report target 150 entities, from governmental departments to service providers. The enablers gave an eighteen question survey regarding challenges of implementations, services and cost savings, and beliefs around smart cities. Barcelona stands out for creating 47000 jobs through the implementation of IoT systems, saving 42.5 million euros in water and generating 36.5 million euros in a year thanks to smart car parks [12].

At the beginning of the 2010s, the city council was determined to make Barcelona the first smart city in Spain. The strategy consisted of using new technologies and infrastructure to foster economic growth and give a better quality of life for its citizens. Several businesses and universities were involved in implementing this strategy, as well as the Autonomous Government of Catalonia that supported the more significant projects. However, one crucial step to take when building a Smart City is the willingness to change, and this could not be

more characteristic of innovative chief technology officer Francesca Bria. The commitment to turn Barcelona into a Smart City and the integration of stakeholders into the city's future gave the city third place in smartest cities. This mentality was accompanied by meters that monitored and optimizeed energy consumption as well as parking apps. This effort to change provided the city with a wide range of benefits, such as:

- More efficient energy through the use of LED-based lights.
- Management of environmental variables with the help of sensors regarding humidity, temperature, or air pollution.
- Reduction in the smell of trash and noise pollution from vehicles collecting it by using smart bins.
- Attacking mobility problems by implementing a city bike system and thus reducing the number of cars circulating in the city.
- Utilizing sensor information to avoid overwatering gardens and adapting the schedule of irrigation systems.
- Improving the quality of the Smart City solutions by encouraging citizens, businesses, and developers to join forces in its development.

As a result of this visionary system and well-coordinated collaborations, Barcelona was named the European Capital of Innovation in 2014, and its Smart City implementation is viewed as a system to learn from.

Not leaving the Iberian Peninsula, Portugal is a good example where the domain of Smart tourism can thrive. Because of its geographic location, Portugal always had a high flow of visitors and tourists, especially in the summertime. Figure 2.1 shows that this number reached an all-time high in August of 2019, providing it 3020.03 million euros of revenue.

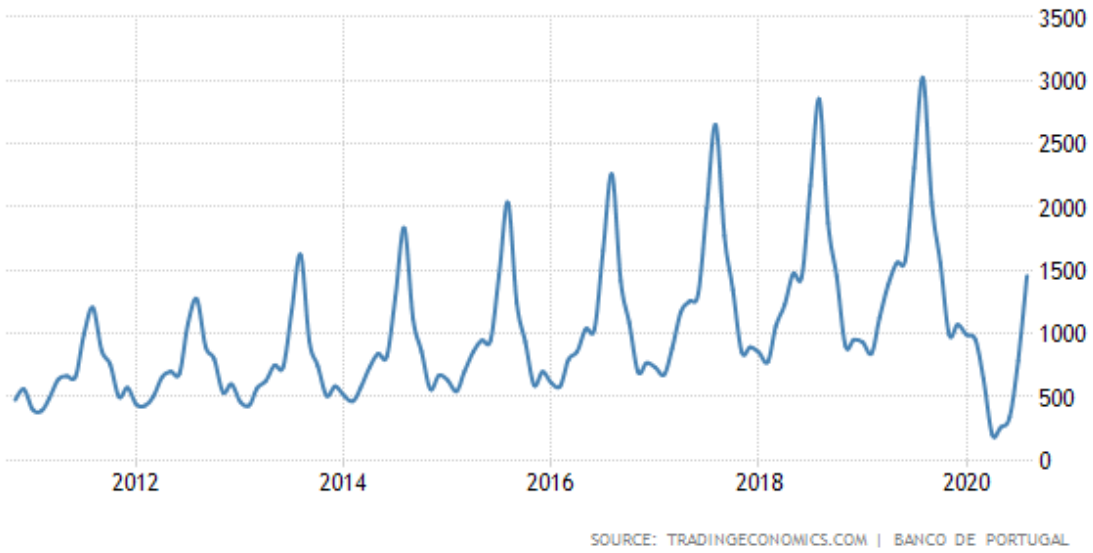


Figure 2.1: Portugal tourism revenues in millions of euros. [13]

Another domain highly invested in is renewable energy. Over the years, placed high in the rankings of greenest countries to live in, holding 9th place in the world nowadays in the

category Planet and Climate [14]. Good Country Index created these rankings, measuring multiple variables such as the country's ecological footprint and the percentage of renewable energy used. It seems that energy and tourism may be the most appropriate domains for Portugal to focus. However, its cities are also worried about other fields, such as mobility and developing new technologies.

Lisbon, the capital, is regarded as one of the smartest cities in Portugal by different parties. Even though Portugal doesn't appear in the top ten, it has several cities that rank high in the smart index. Lisbon and Porto are regarded as the smartest cities in Portugal by different parties. The "Índice de Cidades Digitais," developed by Inteli, is the main measure used to rank cities. As described in Table 2.1, Lisbon went from being the smartes1t city in Portugal in 2012 to give first place to Porto in 2016 and not appearing in the top 5. [15]

Position	2012	2016
1	Lisboa	Porto
2	Almada	Águeda
3	Cascais	Cascais
4	Aveiro	Bragança
5	Vila Nova de Gaia	Guimarães

Table 2.1: INTELI (2012, 2016)

At the end of 2016, Lisbon was not in a great place in the rankings, but that would slowly change as 2019 was approaching. In the sixth edition of the IESE Cities in Motion Index, 174 cities of 80 countries were studied. The Cities in Motion Index (CIMI) was created to help with the ranking process. The calculation of this index considered many variables that describe a Smart City [16]. Between them, urban planning, technology, mobility, and environment are the ones that characterize the intelligence of a city, with the others, such as economy and governance, being enablers to develop the project.

In this study, Lisbon has ranked 47 places above Porto in 44º place, with a CIMI of 63.52 and performance of relatively high [16]. A victory nonetheless, but Lisbon needed to attract more entities to boost the economy and creating jobs. The city created several projects to met these goals, promoting solutions for energy-efficient housing, mobility, and smart living. Some of these innovative systems can be found on a website [17] promoted by the council of the city. It offers a clear picture of what domains are addressed and what initiatives were implemented in each one.

One curious case study happened in July of 2017 when the city council awarded NEC Portugal with a contract to implement a smart city infrastructure aimed at improving the operation and coordination of multiple city services. NEC utilized its Cloud City Operations Center (CCOC) to integrate internal systems controlled by Lisbon's city government and other external systems operated by partners [18]. This platform helps to plan and manage the solutions to the problems in Lisbon regarding energy efficiency, mobility, and ICT [19]. CCOC offers ways to store large amounts of data, in this case regarding sensors in buildings

and information about e-vehicles charging points. It also provides users with a dashboard that can be useful to inspect the collected data and build other applications for traffic control.

Lisbon seems to work more on problems related to mobility and energy efficiency. This attention to these specific problems occurs due to the density of residents and tourists in the region. Porto also focuses on the same issues for the same reasons. It tried to follow a Smart District approach where the city functions as a living lab encouraging citizens and businesses to attack problems that impact Porto. By building the Porto Innovation Hub (PIH), the city tried to encourage an innovative and entrepreneurship ecosystem. This initiative incentivizes business relationships and simplifies the processes by putting citizens in the center of the method of innovation.

Still, some cities, like Aveiro, have a higher focus on creating new technologies and perfect existing ones to improve the quality of life of their residents. Even though the objective is slightly different when comparing to other cities, the strategies used are quite similar. Aveiro also possesses an initiative that brings various stakeholders together, such as people, companies, and research institutes, to build a smart and digital city. This project is called Aveiro Tech City [20] and has the goal to build a more connected city to create new jobs, better healthcare, safer roads, and innovative technologies.

Some research even goes further by studying the university campus to classify the smartness of a city. The results show that students consider the University of Aveiro to be a very high level of smartness [21] but, mixed opinions were made about its infrastructure and care for zero-emissions vehicles, like bikes and electric cars. Nevertheless, throughout the last two years, the university provided more parking slots for bicycles, open areas to study, build new infrastructures, and reduce paper waste by digitalizing canteen payments. Even though much more can be done, the establishment is on the right track. The University of Aveiro must be a place where technology can thrive since its investigation provides enormous help in developing new services. The researchers become even more relevant when knowing that Aveiro wants to be one of the first cities to implement the use of 5G technology into its startups. The project Aveiro Steam City [22] is trying to reach this achievement, and it's here that the concept of Smart Clusters comes into the picture. This initiative is a collaboration of several parties, including Instituto de Telecomunicações, located in the University Campus and some companies, such as INOVARIA and Altice Labs.

Although Aveiro is more focused on developing technology, some of the cities around it are investing in Smart Cities projects related to tourism. Ílhavo was two parishes with beach areas, Barra and Costa Nova, with a high flow of tourists to these regions. Even during seasons with a low tide of visitants, there is still a decent number of residents occupying these areas, especially Barra. It also has few roads connecting these areas, making it easy to study traffic flow. Due to these advantages, many entities like Portugal Telecom and Altice Labs rush to develop, implement, and test solutions incorporated in a Smart City context. However, PASMO was the project where the council of Ílhavo invested the most since it offered the best solutions for mobility, parking, and weather analysis.

2.1.2 Platforms used in Smart City Development

A typical Smart City architecture possesses a massive amount of sensors describing the conditions of the environment. The fact that there are thousands of devices always connected, gathering different types of information causes one of the biggest problems in this context. All these problems arise because the whole point of a smart solution is to collect, analyze, and act on data from a wide variety of sources.

The number of devices installed and connected for IoT purposes is at an all-time high, with CISCO predicting that we will reach the 1.8 billion device mark by 2022, with more than 14 billion M2M connections [23]. It is also noted that Smart Cities related topics, such as energy, connected cities, and mobility, are the fastest growing.

The solutions for these obstacles need to provide a service that is not very complex since it's highly likely that several entities will use it. Communication protocols, types of databases, and varying types of sensor data are the three main factors that needed to be taken into account when building these types of platforms. The last one is probably the hardest to solve because the platform needs to be completely abstract to the type of data received.

Because of the complexity of this problem, many entities chose to use already well-established platforms, like FIWARE. However, there are more local solutions, like the Smart Cloud of Things (SCoT), developed in Instituto de Telecomunicações located in the University of Aveiro.

2.1.2.1 FIWARE

FIWARE is an open-source platform funded by the European Commission, whose mission is to build an open ecosystem to eases the development of new applications in a smart city context. It focuses on public and implementation-driven software standards to help researchers and developers of various sectors [24].

To approach Smart City problems described in the introduction of this section, FIWARE created an OpenStack-based cloud environment. This approach makes it easier for researchers and developers to access the services that FIWARE provides. The system contains a set of open standard APIs to access context information. Context information is represented by values assigned to attributes that characterize those entities relevant to applications [25]. This data can come from several sources, such as mobile apps and sensors. However, the crucial thing is that its type doesn't change when the platform receives it from different devices. For example, if several devices have a humidity sensor and are storying this information, the platform will save the data as describing humidity values, regardless of the type of device.

The component that handles the context information is the Context Broker. It provides data independently of the source, meaning that it can create standardized API, even if the low-level devices in two separate cities are different. The platform gives developers a straightforward way of exploring the sensor data without them worrying about all the details about these devices or how the data is stored. Using this approach, that considers a set of universal standards for context data management, FIWARE can facilitate the development of solutions for various smart domains.

2.1.2.2 SCoT

One of the many platforms that follow a similar concept is the Smart Cloud of Things. SCoT is a Machine-to-Machine (M2M) platform that combines two major concepts, IoT and Internet of Services (IoS), to provide a straight forward service creation in scenarios related to IoT and M2M.

The platform is built to follow a Service Oriented Architecture (SOA), meaning that it offers several services to analyze, process, and manipulate sensor data, making it accessible for researchers to build services using SCoT. It provides several components that cover aspects related to network, device management, services, and applications [26]. In this environment, several tenants can use different sensors to deploy services quickly and effortlessly over a wide range of scenarios. The goal is to connect devices to the cloud, serving as a link between the sensors and third-party services using their data.

SCoT can be divided into four major domains: Sensor, Network, Service, and Data.

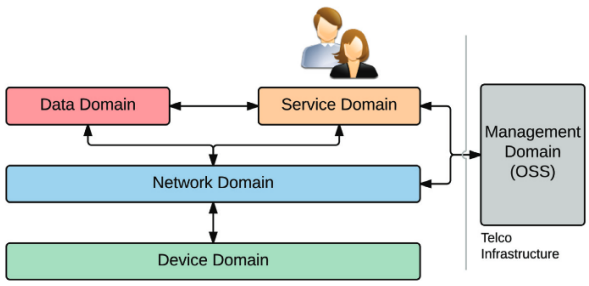


Figure 2.2: Architecture of the SCoT platform, representing the different domains.

The Sensor domain holds the sensors, actuators, and gateways. It allows any devices to communicate with the M2M network ranging from micro-controllers to cell phones. Even though SCoT data is JSON standardized, sensors can send other types of information, being the gateways responsibility to adapt to the new representation. The Network domain serves as points where the devices can connect and send data. The Service Domain provides an SOA environment that provides the developers to build services using data from the sensors. The Data domain follows a similar approach to FIWARE, where it offers a context storage solution that is agnostic to the representation scheme [26]. This approach allows users to search for complex information without the need to understand the representation of the database.

SCoT appears as an excellent solution to the problem of device management found in the context of Smart Cities. When developing a system that will gather all types of sensor data, it is not ideal to have a structured representation of the data sent by these devices. The platform should be indifferent to the sensor that is publishing information, ensuring compatibility between all devices. SCoT provides a robust platform capable of gathering data from devices effortlessly and facilitates service development. The platform is already involved in multiple projects, with scenarios ranging from smart agriculture to mobility and smart lighting.

Before the internet was easily accessible for the majority of people, companies produce software to meet user needs in local networks since machines didn't exchange data between themselves on a global scale. With the widespread of the internet and the popularization of distributed systems, companies started to provide services through this network. The software needs to mature rapidly since many of the technologies used before the internet became obsolete. This happened mainly due to the implementations of firewalls to block ports and secure networks. The only ports available were the ones used for web traffic, and thus, HTTP was chosen to transfer data remotely from one machine to another.

All the technologies utilized in this time would mature extremely fast to keep up with demand. One company gives an example of the growth of web services and clouding computing. AWS is one of the most successful cases of web services providers. After 2013 the company saw an eruption in this domain, which made its revenue increased from 3.1 to 35 billion dollars a year [27]. This growth shows popularity in cloud services and a significant shift from local software to web services. This change occurs because of the appeal of these technologies not only for developers but for users too. Most of them prefer to have all services accessible by a browser or an easy-to-install application on their phone, instead of installing complex software on their machine. For developers, it provides the advantage of not needing to buy and maintain hardware to host their software. Instead, they are all deployed on a machine in some warehouse, having only to worry about the software.

Nowadays, web services have become a popular way of offering online services and support business-to-business application integration. By using them, Smart City's solutions can offer a higher level of efficiency when delivering online services to its users.

2.2.1 API Specifications

There are several technologies developed to exchange data between the internet, but nowadays SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are the main approaches used. Traditionally, services were accessible using SOAP over HTTP (Hypertext Transfer Protocol), with REST being used more recently. Although many big enterprises, such as Google, still use SOAP, there are very few that only offer this technology, considering most of them offer REST interfaces or both.

SOAP is a standard protocol proposed by the W3C [28] that provides a XML-based message framework to perform remote calls to distributed services. Entities that consume SOAP services rely on a Web Service Description Language (WSDL) specification [29] to describe the content and availability of those services. It was one of the first protocols used to exchange messages through the internet, so companies utilized it and still provide support to it.

The main advantages of this protocol are the fact that it can use by type of transport protocol and be applied in any programming language. This makes it simple to exchange information among different servers, avoiding complications such as message formats and explicit calls. Another essential feature is SOAP's built-in error handling. The protocol

responds with standard error message codes that offer tons of help when debugging. It helps clients better understand the errors returned by the web service.

Even though SOAP is a feature-rich protocol, its completeness brings some disadvantages. Since it uses XML as its language, it makes the communications with the service more verbose, spending more resources to decode the XML message. This limitation makes it "heavier" than other protocols and conceptually more difficult for the client since it requires a higher effort by the client-side. The "heaviness" of the messages translates to a bad performance since it needs more bandwidth to exchange information [30].

Even though SOAP is quite complex, making its performance worse when comparing to other styles [31], it's still utilized in some web services where the aim is high security and a distributed setting. Some examples are financial services, payment gateways or other high-security apps, telecommunication services, and distributed environments.

After some time, an alternative architecture was introduced in the form of REST. Over the years, REST became widely used across multiple areas in IT due to its efficiency in accessing complex systems.

In the early days of web develop servers needed to remember specific details about clients accessing the system, i.e., its state. REST came as a solution to this problem relying on a client-server style, where the client initiates the request, followed by the servers processing it and sending the response to the client. This style of communication was used in combination with a concept where the client provides all the information needed so the server can perform a particular action. By using this process, the need to have client information in the server is eliminated, making the server simpler and more scalable. This simplicity makes the style light-weight and easy to develop, compared to SOAP. Unlike SOAP, REST is not restricted to the use of XML, being able to utilize other message formats such as JavaScript Object Notation (JSON) or Really Simple Syndication (RSS). Supporting smaller message types makes the transport of information faster and efficient.

Even though REST is simpler and faster, it lacks standards compared to SOAP, making it less secure. Moreover, it is not suitable for distributed systems since it uses a client-server approach. It's also tied to the HTTP protocol for transport, not being able to use others if necessary. However, one of the most common problems in REST is over and under fetching. Over fetching means that the client gets more information than is needed, while under fetching indicates that not enough data is provided for the requirements of the application. These situations happen because of fixed data structures, being hard to design a REST API around these problems.

Despite having some disadvantage that makes it impractical to be applied in some systems, REST is still practiced in many web services that follow a client-server strategy, in particular public APIs. Due to its popularity, REST has vast support when talking about tools for its development, making it a more easygoing solution to be implemented.

Recently, a stable alternative was released, called GraphQL [32], that aimed to solve the problems of under and over fetching found in REST. It was developed internally by Facebook to solve these issues and further optimize performance and flexibility. GraphQL attacked the

fetching problems by not relying on fixed data structures.

GraphQL is a query language, i.e., it retrieves data from a database by sending queries that allow the user to ask for specific types of data. In some situations, this approach reduces the number of requests to the API from several, using other specifications, to one, using GraphQL. Here objects are represented by nodes, defined using a schema, while the edges describe the relationship between these objects. So, quoting GraphQL Co-Inventor Lee Byron, using GraphQL, users need to "Think in graphs, not endpoints." Endpoints provide fixed data structures, but graphs don't.

Using this specification, developers only need to send a query to the GraphQL server that specifies the data requirements that they need. The server responds with an object containing all the information that the query asked. In other specifications, like REST, the user may have to make more than one request to fulfill its requirements since the endpoints have fixed data structures. This query request approach solves many problems in the fetching process with other solutions, but tooling support is limited when comparing to REST. This makes it hard for GraphQL to support results caching, unlike REST which, takes advantage of HTTP caching to have better performance.

Even though the three specifications provided vary a lot, complete APIs can take advantage of all of them by implementing every approach. The development of such a service will take time and cost resources, but in the end, it would cover almost all the use-cases of its users.

2.2.2 Web Services Architectures

Traditionally, monolithic architectures were adopted in many systems to build web services, meaning that the service was composed of a single unified unit. Even though this trend is going away, many businesses still adopt this style, building one self-contained and autonomous software where its components are interconnected. This architecture would found a decline in popularity over the years because of the numerous limitations and issues brought by this style. Since this approach compiles multiple services in a single code base, it makes it challenging to adopt innovative technologies, scale or implement new changes. Furthermore, if a component needs to be updated, the majority of the application had to be rewritten. These inconveniences made it challenging to develop software when multiple parties were involved, which goes against the basic principles of Smart City solutions.

Nowadays, with the increase in cloud computing popularity in the DevOps field, many companies have migrated their systems to the cloud and succeeding in building a set of independent and modular components that are easy to test, maintain, and understand. This type of organization where unconnected services work together to form a single application is called microservices architecture. This approach is used by many big enterprises, such as Netflix, Amazon, and eBay [33], that saw the benefit of having independent services that can be developed by smaller teams working in parallel. Microservices can rely on technology heterogeneity, which means that each service can use different technologies to achieve the desired goals. Furthermore, we ensure that the whole system doesn't shut down when a service crashes. Even if these services depend on each other, they should be able to operate without

this connection, affecting the system performance but providing the service nonetheless. Another great advantage when using this approach is that the process of scaling remains much more manageable than in a monolithic architecture. Effortless scalability is possible since only services that need scaling are updated, in contrast to the monolithic architecture, where the whole system needs to be changed.

Microservices are a significant shift in the DevOps domain, allowing collaboration between teams, or even companies, developing web services. Even with all these advantages, some problems came along with this strategy, manly service discovery, security, and failure handling. The latter doesn't refer to total failure, i.e., crash, but to malfunctions that occur a certain number of times or even slowness in the responses. These types of failures are much harder to solve or even detect, for that matter. Other concerns arise because of the natural structure of the web services, being easily accessible. Users regard the security of these online applications as one of the most crucial elements, especially when the service provides online payment. The survey [34] shows that abuse of functionality and spoofing are on top of the attacks performed in cloud services. Even though some of these exploits can be related to user carelessness, the developer needs to ensure that the least amount of attacks happen. All these difficulties come because microservice's architecture is composed of independent components. Nevertheless, the advantages outdo the disadvantages, plus there are a lot of methods and tools available to solve these problems [35], [36].

2.2.3 Deploying Web Services

It was already stated how cloud computing helped to improve solutions for IoT, Smart Cities and web services in general. However, it is beneficial to understand how this approach works to appreciate why it is utilized so much in IT. Besides, it is also essential to understand the use of lightweight virtualization solutions to have a more efficient and agile system. These solutions can run a virtual instance of a computer system in a layer abstracted from the hardware. This method makes it easy to run multiple independent applications on a single server, providing more effortless management of the system. Moreover, the adoption of cloud computing has been growing so much that some reports state that, by 2021, data centers will run half of all company's workloads [37].

Cloud computing provides an exceptional alternative to access and deliver services over the Internet, allowing faster innovation, low operating costs, and reliability. It enables not only users but also businesses to access resources from anywhere when needed. Moreover, this technology has been identified to have a major influence on small and medium-sized enterprises, in particular, the availability of broadband internet access and data security and privacy concerns [38]. When it comes to how the service provider implements the architecture of its cloud, the shift from a centralized system to a more distributed approach is apparent [39]. These environments are many times comprised of multiple virtual nodes, namely OSs and applications. Virtualizing resources come with the advantages of being easier to manage because of OS abstraction, which makes the service is independent of the underlying OS and hardware. Cloud computing uses this approach to implement multiple virtual servers into a

single physical machine. This method can provide several services using only one computer if the hardware is powerful enough. Furthermore, this approach can be used to guarantee availability by deploying redundancy [40], i.e., having several virtual nodes deploying the same software so that if one fails, the service is still being provided. Finally, because it is agnostic to the hardware and OS, the migration of these virtualized components is a trivial task, offering developers an easy way to move services between machines.

The virtual nodes can be created using either virtualization or containerization, with Virtual Machines(VMs) used to achieve the former, and containers used to accomplish the latter. VMs are used to replicate servers by including a complete OS with all the drivers, libraries, etc. Usually, they aren't utilized to virtualize services since VMs require the simulation of the hardware and consequently occupying a lot of space in the system. A better way to virtualize a service is by using containers. Containers allow the virtualization of different instances in a single OS, with them sharing the underlying kernel. By using this approach, only the application needs to be simulated, not the whole OS, meaning that a single OS can serve many services.

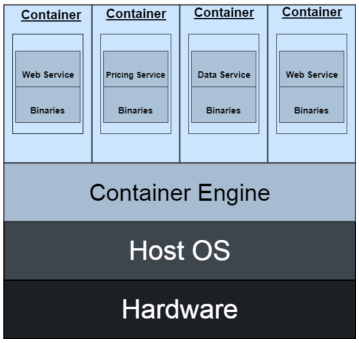


Figure 2.3: Typical container architecture.

In a typical container architecture, as represented in Figure 2.3, the container engine sits on top of the OS, supporting services, libraries, and binaries, while still able to communicate with each other. This strategy is very lightweight compared to VMs since, as stated, a container will run a process and does not require virtualization of the full OS, giving them a better performance over VMs. Although containers achieve better performance over VMs, the latter have an edge when it comes to security and isolation [41]. Even though the use-cases vary significantly, some researchers suggest a hybrid virtualization approach, where deploying containers inside VMs provide the performance of the former with the security properties of the latter [42].

2.3 Object Detection Solutions

Research in object recognition has been around since the 1960s, but only in the 21st century, the field saw its increase in popularity when an efficient working face detector was developed. More face detectors appear through the years, but it was at the beginning of the 2010s that object classification would have incredible improvements by utilizing deep learning. In 2012 an

annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) was held. During this year's competition, a team of researchers trained a Convolutional Neural Network(CNN) in ImageNet, to classify images [43]. This team's model outperformed every other competitor for ImageNet. Although this is a big step in object classification, detecting things is much harder. Detection refers to both identify the position and the class(what it is) of an object in an image. Nevertheless, CNNs helped with this challenging task, though in conjunction with other techniques, since using only the network would have a huge computationally cost. Lots of algorithms were developed, still using these networks, but approaching the problem in different ways. After some years of research, the algorithms became so good at detecting objects that the paradigm shifted to other objectives. In the mid 2010s, researchers knew that the accuracy of the detectors was great, but they were extremely slow. So, the next step was to improve the time an algorithm took to process an image. Nowadays, this technology evolved to be able to perform in real-time scenarios with high detection accuracy.

2.3.1 Non Real-Time Object Detection

The deep learning era of object detection can be divided into two genres: two-stage detection and one-stage detection. The first one frames the detection as a "coarse-to-fine" process, while the second is a method that "completes in one step." [44] The expression "coarse-to-fine" means that the two-stage detection initial segmentation gives not much detail, i.e., only proposes a set of regions of interest. Only in the second stage, when the classifier is applied, we get the objects detected, i.e., a fine segmentation. The expression that describes one-stage detection is rather clear, stating the process is completed in one step. The algorithm uses only one CNN to detect and classify the objects, being finished after passing through the network only one time.

The Regions with Convolutional Neural Networks(R-CNN) family of algorithms ruled the first genre, improving significantly with every version. The idea of R-CNN is to begin by selecting some regions of interest before feeding the image to the network. This process is called selective search, and its goal is to create a set of region proposals in an image, i.e., select areas that might contain objects. These regions are chosen by analyzing the texture, color, or intensity of an image. Once these are processed, they are feed through a CNN to compute features in that area. In the end, a support vector machine reads the feature vector and classifies the object in that region [43].

R-CNN still took extensive amounts of time to train networks, plus the selective search was fixed, meaning that it couldn't learn to generate better region proposals in the first stage of the algorithm. Furthermore, its real-time performance was not even taken into consideration since it took around 47 seconds to test each image. Thus, the efforts turned out to implement a faster algorithm while taking advantage of the accuracy of R-CNN. Some of its popular implementations, as well as the original, used selective search to find region proposals. This process is quite slow and time-consuming, affecting the algorithm's performance. Thus, the clear performance boost was to eliminate selective search and make the network learn the

region proposals.

Faster R-CNN [45] is one of the fastest algorithms based on R-CNN. It accomplishes this by replacing the selective search process with a convolutional feature map that is used to predict region proposals. Figure 2.4 shows that by using this approach Faster R-CNN outperforms other R-CNN implementations in test time while increasing accuracy.

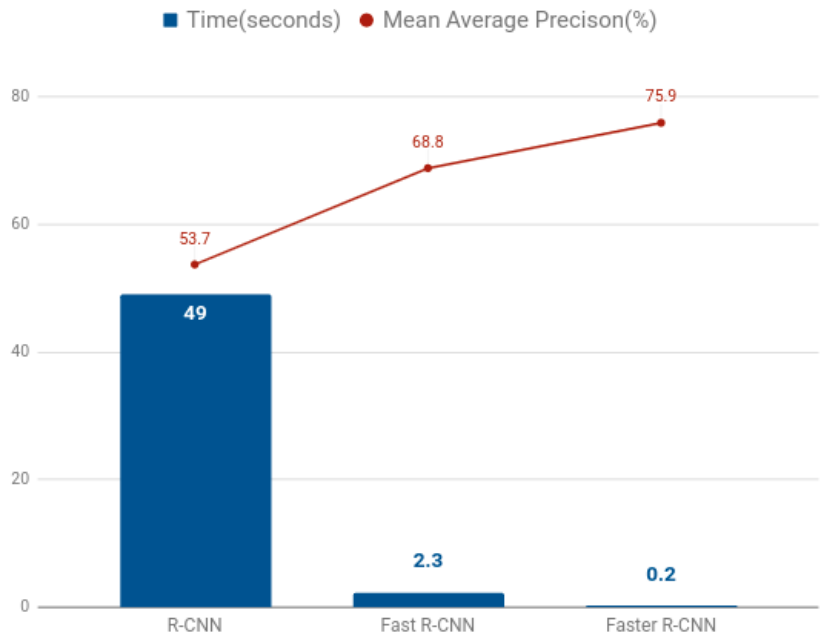


Figure 2.4: Comparison of test-time speed and mean average precision of algorithms based on R-CNN [45]–[47]

With this performance, Faster R-CNN can process a video at five frames per second using a powerfull GPU. It is a magnificent improvement of R-CNN but its a little slow to be called real-time object detection. The processing time was high due to the nature of the algorithms themselves, it being a two-stage detector. However, after the paradigm shifted to the genre of one-stage detection, the real-time performance started to appear. During this era, algorithms such as YOLO and SSD would set major milestones while constantly pushing the envelope, both in terms of speed and accuracy.

2.3.2 Real-Time Object Detection

The previously discussed object detection algorithms use regions to locate the object within an image, i.e., instead of the network looking at the complete picture, it scans parts of it where there is a higher probability of having an object. To solve the object detection problem, YOLO [48] takes a different approach that pays off since it outperforms all R-CNN versions. YOLO is a single shot detector where the goal was to look at the image only once but cleverly, hence its name You Only Look Once. YOLO divides the image into an SxS grid where each cell is responsible for predicting two things, some number of bounding boxes and the confidence values for each. These values describe the probability of a certain box containing an object. This first process only tells the algorithm where the objects are, not what they are.

So, following the previous step, the network will predict class probabilities and compute the bounding boxes weighted by their likelihood of containing an object [48].

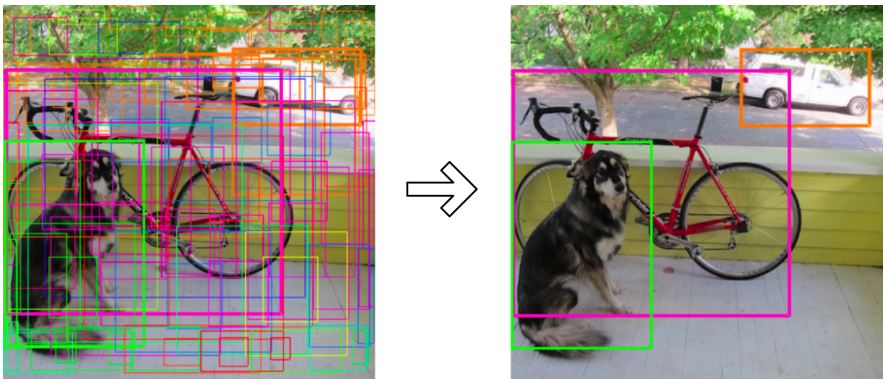


Figure 2.5: The difference in the number of bounding boxes before and after the threshold is applied

The result given by the network is a ton of bounding boxes, the majority with low confidence values. Thus, by applying a threshold to the predictions, the final detection image is processed. This value is usually above 0.6, meaning that the algorithm only displays boxes that have a 60% or more probability of having an object. YOLO is way faster than other object detection algorithms but struggles when it comes to detecting smaller objects. Even with some problems, it outperformed all R-CNN based algorithms in speed. Now with 45 FPS, YOLO was a better real-time object detector than Faster R-CNN, with little cost in accuracy.

Real-Time Detectors	mAP	FPS
Fast YOLO	52.7	155
YOLO	63.4	45

Not Real-Time Detectors	mAP	FPS
Fast R-CNN	70.0	0.5
Faster R-CNN ZF	62.1	18

Table 2.2: Comparison of the performance and speed of object detectors

Another example of a single-stage detector is the Single Shot MultiBox Detector (SSD) [49]. It works similarly to YOLO, in the sense of using a fully convolutional approach in which the network can find all objects with one pass of the image. This algorithm shares a lot of techniques with YOLO, but it has a feature that was very important to boost its performance, which is anchor boxes. While some R-CNN versions use an algorithm to help generate bounding boxes, SSD pre-computes multiple anchor boxes utilizing a network to extract feature maps [49]. This process gives it a significant advantage considering other algorithms have fixed selective search. YOLO didn't utilize this method, simply predicting the coordinates of the bounding boxes. SSD, on the other hand, predict offsets for the pre-computed boxes because it's much easier to adjust these values instead of computing the exact coordinates. Moreover, SSD uses a network to generate anchor boxes so the algorithm can learn how to create better ones.

Using this whole process, SSD outperformed the first version of YOLO, both in speed and

accuracy. However, the developers knew that YOLO had a computationally expensive way to predict bounding boxes and offered poor detection accuracy in small objects. Therefore, the second version of this algorithm, YOLO9000, used modified techniques of already existing solutions to solve these issues.

The first version of YOLO was fast but fell short in terms of accuracy compared to other object detectors. The second version YOLO9000 [50] focused on primarily improving the accuracy of detection while not compromising too much on speed. In some cases, the developers took inspiration from other algorithms, such as SSD, trying to improve their features, in this case, the use of anchor boxes. As stated previously, SSD uses pre-computed anchor boxes to predict regions where the objects may be located, making it more efficient to work with offsets than with exact coordinates. Even though the detector learns whose boxes have a better intersection with the object, the aspect ratio of anchor boxes is fixed, giving very little freedom when it comes to the shape of these boxes. YOLO9000 developers, when looking at training data, realized that these aspect ratios do not always correspond to reality. The solution was to run k-mean clustering on the training set to find proper dimensions. The result was a set of boxes, called dimension clusters, that describe how close they are to overlap an object, i.e., the best possible starting place for these boxes. This technique YOLO9000 able to surpass the accuracy of 9 anchor boxes by using only 5 clusters.

Another method used to try to fight the apparent problem that the detector had with small objects was multi-scale training. The training process of the first version used only one aspect ratio, 448x448, resizing all the images to that size. YOLO9000 resized the network every few iterations, choosing a new image dimension between 320x320 and 608x608. This procedure makes the model robust enough to run on inputs with different sizes, learning to predict well across a variety of dimensions, ensuring that all the features learned from the small images will translate well to a network operating bigger ones. Multi-Scale training also avoids overfitting without influencing performance in a significant way since the only thing changing is the image size.

Even though these changes boosted the mAP and the FPSs from 63.4% and 45, in the first version, to 77.8% and 59, YOLO9000 was still quite bad at detecting small objects, not performing well in CCTV footage of traffic, for example. Nevertheless, the third version of the algorithm addresses this issue. YOLOv3 [51] utilized the multi-scale training idea of the previous version and applied it in detection. Images are downsampled by 32, 16, and 8 to perform prediction across different scales, helping with the small object matter. Although other features were implemented, this method made version 3 stand out in performance.

The most recent YOLO version, YOLOv4 [52], makes a significant jump in speed and accuracy compared to the previous algorithms. Here the researchers try to obtain performance by not modifying the prediction algorithm itself but by changing the architecture of the object detector, composed of the backbone, neck, and head. This means that the dense prediction component, which is the one responsible for computing the bounding boxes and its confidences, uses the YOLOv3 algorithm. This component is located on the head of the architecture that connects to the neck. This part holds some layers used to collect feature maps from different

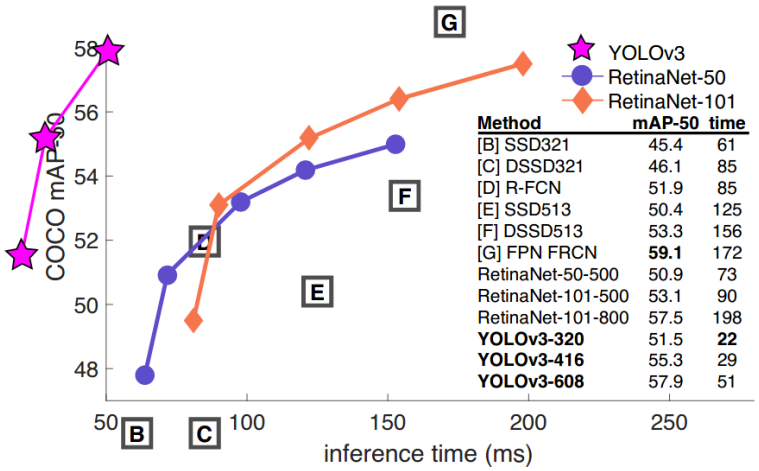


Figure 2.6: Speed and accuracy of YOLOv3 and other algorithms. This version has the best tradeoffs between speed and accuracy across all object detection algorithms.

stages. YOLOv4 applies two methods in this component, Path Aggregation Network (PAN) and Spatial Pyramid Pooling (SPP), that are used to detect objects in different scales. The former combines information from all the layers to avoid duplicated predictions, while the latter separates the most significant context features without compromising speed. Finally, CSPDarknet53 was used in the backbone since it was shown to be the most optimal model.

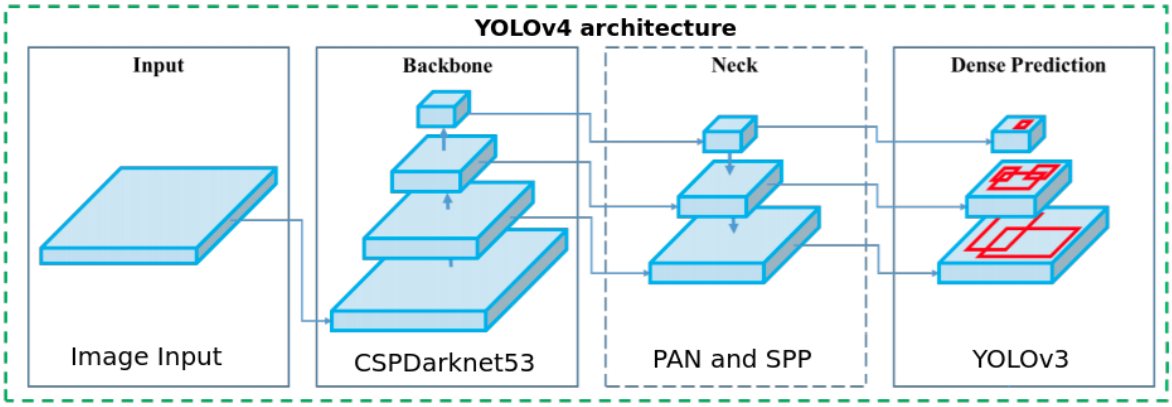


Figure 2.7: YOLOv4 architecture.

When it comes to training optimization, numerous strategies were used that can be classified by BoF (Bag-of-Freebies) or BoS (Bag-of-Specials). The first details methods that can be utilized without any cost to performance, i.e., for free. The second describes techniques that only increase the inference cost by a small amount but offer large improvements in the accuracy of the detector, i.e., getting a special offer. All these methods make YOLOv4 a state of the art object detector, tested in various GPU architecture. Figure 2.8 shows the results of the average precision for different values of IoU.

Object detection algorithms approached a stage where their performance surpassed the real-time speeds, using good GPUs. Nowadays, these detectors can achieve good performance on non-top of the line graphics cards. Moreover, with the development of smaller algorithms,

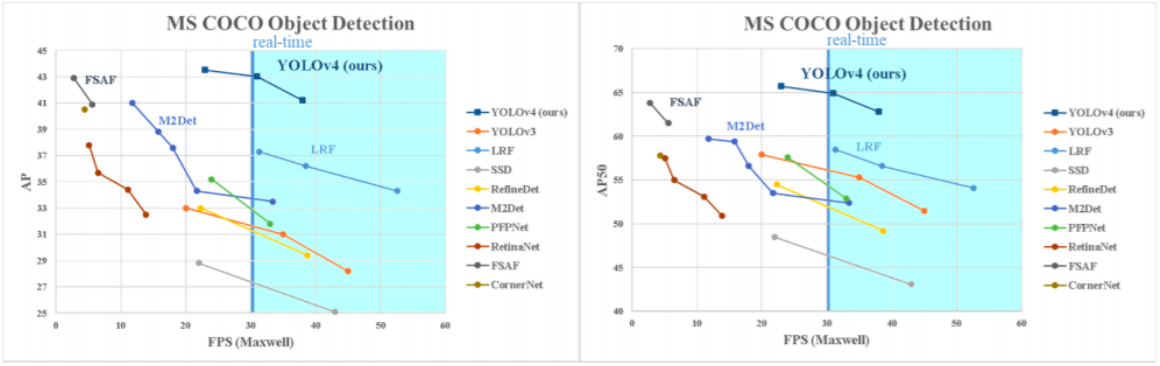


Figure 2.8: Comparison of the accuracy and speed of object detectors in the Maxwell architecture. The left chart shows the AP for IoU values varying between 0.50 to 0.95, at a step of 0.05, i.e., ten values tested. The AP value on the right chart is computed at a single IoU of 0.50.

with fewer convolution layers, based on bigger ones, less powerful GPUs can achieve real-time speeds, though with some cost inaccuracy. YOLOv4 tiny comes as an example, with some researchers achieving 270 FPS with a top of the line graphic card [53]. These numbers mean that even small GPUs, for example, the ones implemented in embedded systems, can achieve real-time speeds of 25 to 30 FPS. This provides a significant benefit for object detection units that work with CCTV footage since instead of processing the video in one big machine, a board with an embedded GPU can consume images directly from the camera. These scenarios are quite common, so these small detectors will probably end up being more utilized than the heavier ones.

2.4 Sensor Fusion

Over the years, radars and sensors technology matured fast, and became cheaper, due to the rise in popularity of IoT and Smart Cities [54]. Other components used in these fields, like cameras, are also getting better, but their image quality in certain conditions suffers a lot. The most common are bad weather and low luminosity, which sometimes influence the image quality so severely that it becomes almost impossible to extract any information from the camera. Nevertheless, these devices provide a much better classification of the environment than radars, being worst when it comes to detection. The limitation and qualities of each are well known, so the best solution to get a better picture of the environment is to fuse data from the two. This information is combined from areas where these devices shine, i.e., classification data from the cameras and detection data from the radars. Thus, sensor fusion's goal is to combine information from various devices to produce more precise data or, in this case, making object detection more accurate.

This method offers a reliable solution when it comes to traffic problems, being either in the domain of autonomous vehicles or IoT. Even though these two fields are distinct at its core, the solutions for object detection vary little. The more reliable approach is, as detailed above, to combine data from a camera and a radar. We can't escape the basics when discussing

the latter since, even though the quality of the sensors influences the final data, most of the radars measure the same types of information. Furthermore, the user doesn't need much work since the radar provides well-structured data. We can't say the same about cameras considering that it is much harder to get quantifiable information from a video. Object detection algorithms help solve this problem since they can not only predict the coordinates of an object in a video/image but also classifying this object. Moreover, with further calculations, other variables, like distance from the object or its speed, can be predicted from the values provided by the detector.

Many methods are being designed to optimize results when combining these devices. One of them uses a radar sensor that outputs a sparse 2D point cloud in conjunction with the channels red, green, and blue of a stream from a camera. All this information is collected and feed into a neural network built on RetinaNet that they called CRF-Net [55]. This approach tries to outperform image-only object detectors by using radar and camera data. It can calculate the distance from a detected object, making it perfect to be applied in autonomous driving. However, the network needs 43ms to process the fused data at an image resolution of 360x640 pixels. For higher resolutions, CRF-Net needs 103ms to process the information, which equates to 9 FPS, making it impossible to use in real-time scenarios.

Another approach based on object detection algorithms [56] combines one millimeter-wave radar with a camera to solve the problem of low visibility video in foggy weather. Here YOLO9000 is used to recognized traffic and later be fused with the radar information. The low visibility problem is solved by defogging the image to have a better view of the environment. After the fusion, regions of interest are drawn in the video based on this whole process. Even though this approach solves a real common problem(low visibility in cameras), the real-time performance was not meet, running at 15 FPS. However, it uses an obsolete version of YOLO, being very likely to meet real-time performance if a newer algorithm was utilized.

Even though these strategies provide high precision of object detection when combining data from devices, most of them lack real-time performance. This obstacle, however, can be easily solved, in some cases, by using a better and faster detector or even the right hardware. The latter is crucial to extract all the performance out of the detectors. After the increase in popularity of the Rasberry Pi products, some companies saw a great opportunity to invest in creating products that provided something that the Rasberry's lacked, a good GPU. Nowadays, UDOO, with the UDOO BOLT line of products, and NVIDIA with the Jetson kit, provide excellent solutions for compact and power-efficient modules that can be implemented in the field of artificial intelligence.

2.5 Technologies applied in a Smart City context

Sensors are the backbone of a Smart City, allowing real-time monitoring of the environment. Focusing on the context of traffic management, this section is majorly dominated by stationary radars [57]. These devices are normally spread throughout the roads of a city, being able to observe the traffic and measure its flow or speed. Even though radars are the traditional way to monitor traffic, many cities try to implement different solutions using other types of devices.

One example is Pittsburgh, which deployed a system called SURTRAC [58]. The sensor infrastructure of this project can range from cameras, radars, or even induction loops. All these devices help the platform to craft the most efficient plan to then apply to the connected intersections. This process helps with the management of traffic queues by focusing on higher flows. With the help of this system, the traffic efficiency improved by 25% to 40% while also reducing carbon emissions by over 20%.

Even though this solution didn't apply any object detection algorithms discussed previously, other smart cities took advantage of the classification power provided by these detectors. YOLO is being used by Elan Electronics as a traffic solution, being deployed in Taiwan [59]. Although there aren't many examples of fully developed solutions since these algorithms only became reliable quite recently, researchers are working on solutions as close to a smart city scenario as possible. MME-YOLO [60] uses YOLOv3 as a base, adding posterior adjustments to make traffic detection more reliable. The tests are performed with CCTV images, mimicking what a sensor infrastructure in a city might look like. One particularity of this setup is the way the cameras are installed and where they are pointing. Instead of surveilling the occlusion spots, where the density of traffic is high, like near a traffic light, it records the whole stretch of road. This approach, illustrated in Figure 2.9, even with clear disadvantages such as greater exposure to vehicle lights, provides a better picture of the movement of traffic, making it easier to detect large traffic jams.

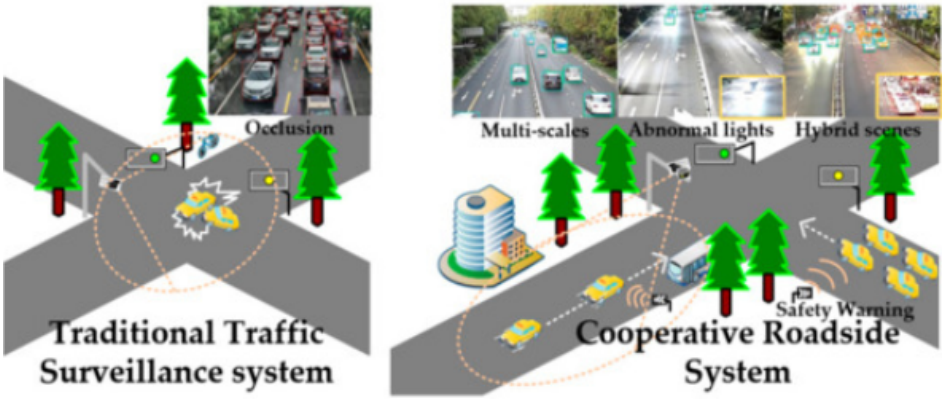


Figure 2.9: Difference between the tradicional CCTV setup and one capturing the whole streach of road.

Many more applications can be found in the same scenario, where an object detector is receiving video footage from a CCTV device to identify some type of traffic behavior. In [61] YOLOv3 was used as a base of object detection to identify illegally parked vehicles. This algorithm was complemented with a movement tracking to understand if the vehicle was parked or not. Even though this approach and MME-YOLO are not implemented in a smart city solution yet, they are inserted in a scenario related to this concept, which is smart mobility.

Nevertheless, the use of only CCTV as a form of detection can produce misleading results in some conditions, such as bad weather or poor lighting. To solve this issue, many

solutions combine two types of sensors (radars and cameras) to achieve the best detection and classification possible. The examples discussed above were applied in a self-driving car context, but other solutions solve issues more related to smart cities, like level crossing. In [62] a system was developed to detect vehicles and pedestrians in a level crossing area. This system makes use of both radars and cameras to achieve better detection and provide risk management in these areas. The algorithm relies on features instead of raw pixels, which provide better performance in terms of speed but less accuracy when comparing to detectors discussed previously. The document not only provides a robust system but also discusses a crucial idea that can be applied to almost every level crossing. In most of these areas, radars are already installed, so it would be easy to deploy a camera next to it and implement the system. It also states that this approach is less cost-efficient than deploying other devices like ultrasonic sensors. These are more expensive than radars, needing to be complemented with other devices because of their lack of efficiency in adverse weather conditions.

Even though these types of solutions help the city residents or tourists, they can be obscure to people, in the sense that they will probably not even know that the system exists. To publicize these solutions, many smart cities develop public services where users can access information provided by the sensors previously installed. Barcelona took this approach by building a dashboard to monitor data, called Smart Citizen Kit [63], measured by the citizens. This approach involves people in the measurement process, raising awareness of environmental concerns. The dashboard receives all the data collected from the low-cost sensors deployed throughout the city, indoors and outdoors, and displays it on an interactive map. Here the user can search for the active sensors in a certain area to know the light conditions, air temperature, humidity, noise, and much more.

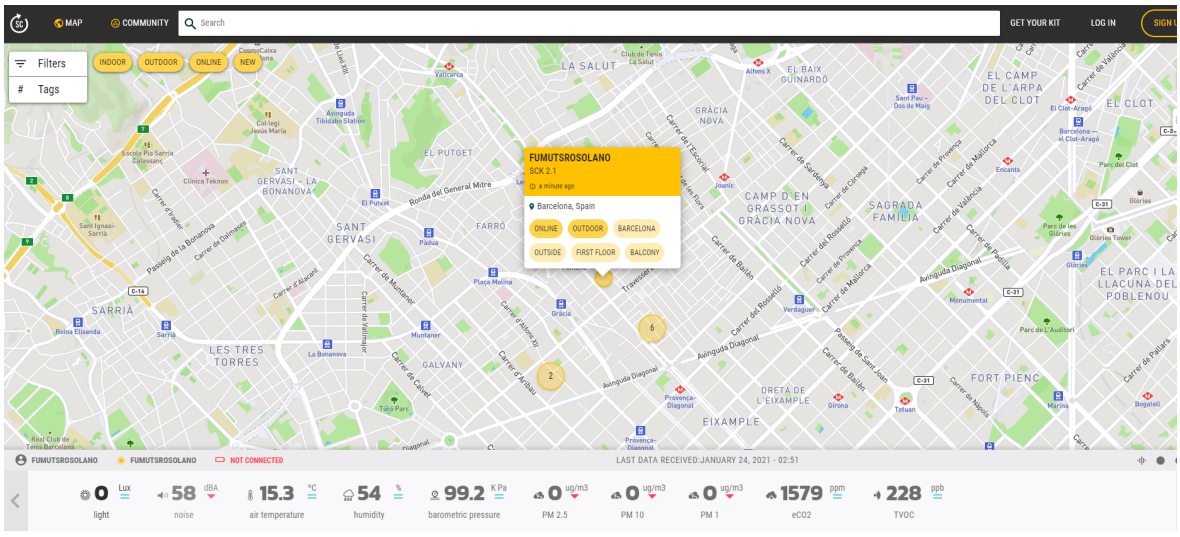


Figure 2.10: Dashboard from the Smart Citizen project, derived from [64].

By giving the residents part of the responsibility of monitoring environmental data, they can feel more involved in the project and be a helping hand in the city's progress.

Even though the systems discussed differ a lot in their approach and use of sensors,

the diversity of implementations and data is what builds an excellent smart city solution. Nevertheless, the core of these applications is the same, which is actively using sensor's data to build solutions that help the residents of the city.

CHAPTER 3

Requirements and Solution Architecture

3.1 Scope

The initial plan was to explore three types of domains (parking, traffic, and traffic classification) to solve the issues regarding this dissertation. These areas are linked to the sensors used to implement its systems. The objective of the Parking domain is to utilize the information provided by the parking sensors to build a map with icons, describing their state and location. The second domain would be more focused on a dashboard component to visualize the state of the radars and traffic, being complemented with a map that provides real-time information. The last domain will use both the cameras and the radar's data to achieve a more precise traffic classification. An object detection algorithm will be implemented in some piece of hardware, utilizing the CCTV images to reach the goal estabilshed.

Before discussing the solutions themselves, we have to understand the target user of these services. As stated before, this dissertation falls in the context of the PASMO project, where the goal is to provide a platform for companies to experiment with solutions and offer users services to better their quality of life. Still, system administrators need a simple way to visualize failures in the sensors, to act rapidly and accordingly. Therefore, we need to consider these two parties since the data displayed must reflect the user's needs.

The base idea behind the two first domains was not only to provide a complete dashboard example on how researchers can use PASMO sensor data but also help administrators to detect failures. The dashboard also has user-friendly maps that show what is happening in the region. The traffic classification domain would be complementing the platform to provide more reliable traffic information. The goal of this component is not to offer a public service. Instead, the purpose is to be implemented as a way to utilize the cameras and produce more precise data.

To understand the sensor data that describes the real world, one needs to acknowledge the location and orientation of the sensors. For the parking sensors, only location is relevant

since they are all pointing up to detect a vehicle. However, when discussing the radars, their orientation matters a lot because vehicle detection depends on this variable.

PASMO infrastructure has 17 RSUs installed, but only three are giving radar data. The RSUs portrayed as the icons with the numbers in Figure 3.1 represent the ones that publish radar information, while the ones with a letter don't produce any data.

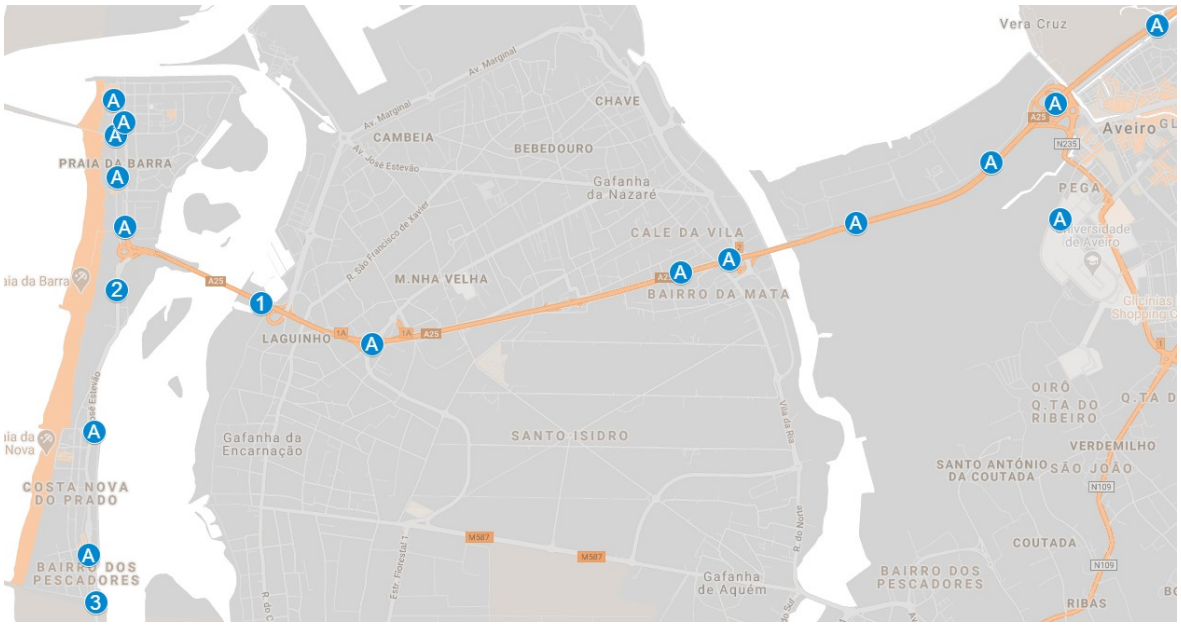


Figure 3.1: Map displaying all the RSUs installed though by the PASMO project.

These numbers 1, 2, and 3 identify the Ponte, DunaMeio, and RiaAtiva radars, respectively. The Ponte device is located at the beginning of the Barra bridge (latitude 40.628265 and longitude -8.733521), aiming east southeast. The DunaMeio radar is located at the north entrance of Costa Nova, near a restaurant with the same name (latitude 40.629108 and longitude -8.746947), and points north. Finally, RiaAtiva is located at the south entrance of Costa Nova, near a surfing school with the same name (latitude 40.607589 and longitude -8.748826), and it's pointing south. Another essential thing to understand is the limitation of the regions Barra and Costa Nova, as well as which radars work in which areas. The Ponte radar solely handles traffic in Barra while RiaAtiva only manages Costa Nova. The device DunaMeio handles traffic from the two regions since it is within the limit of both of them. Figure 3.2 illustrates what was described in above. Region borders and the location of the radars are identified as well as the orientation of the devices.

When it comes to the parking sensors, these were installed only in Barra. The devices are placed along the road near the beach and in the area near the lighthouse.

Even though both of these devices, radars and parking sensors, publish data in the SCoT platform in a JSON format, the information stored is reasonably different. It's crucial to understand the types of data that the devices give.

Beginning with the parking sensors, these publish simple information, providing their state, the timestamp of the read, and their latitude and longitude. The first variable only

Figure 3.2: Map describing regions borders as well as the location of the radars and its orientation.

has two values, 1 and 0, them meaning spot occupied and free, respectively. In contrast, the radars offer much more information about the world. These devices provide their ID, latitude, longitude, and azimuth, as well as the ID, speed, position, and length of the object detected. The speed and position are defined by two variables each, describing the value in the x and y-axis. This information, in conjunction with the latitude, longitude, and azimuth of the radar, can be used to calculate the location(latitude and longitude) of the object detected.

The third domain uses the cameras to provide a better classification of the traffic. These devices are installed alongside the radars, in the same RSUs presented above. Since the content of the cameras cannot be shared, this component will not be provided to the public. Still, it gives PASMO developers the ability to examine real-time traffic flow with precise classification.

By completing the goals presented and developing the services for each domain, PASMO will be a complete Smart City solution, providing accurate data to the public about the beachside of Gafanha da Nazaré.

As stated previously, to have this whole system working, a full-stack solution is required, together with the traffic classification component, which is a crucial element to produce reliable data.

When it comes to storing information collected by the various sensors, the SCoT platform provides a reliable and versatile way to save and access sensor data from its databases. One of the most important features it offers for the context of this dissertation is giving users the ability to choose between different types of databases. With this freedom of choosing where the data is retrieved, users can improve application performance on this factor alone. Even though data can be retrieved directly from SCoT, the front-end applications should not have direct access to it for security reasons. Furthermore, the sensor information stored will be accessed not only by the platform developed in this dissertation but by the public, so a more effortless and fast way to access data is required.

Here we arrive at another crucial component, which is the backend server that serves as an API. This element will be responsible for accessing SCoT, process its data, and provide it in a standard format for all to use. This web service was developed to follow PASMO purpose, an open platform for researchers to develop their ideas. Such a service needs to provide fast access to information and an excellent way to load balance requests so that user's applications do not get bottlenecked by it. Furthermore, it is essential to have a well-described documentation page where all the services provided are specified since we are building a public service.

The API should have requests that give raw sensor data, but other essential measurements can also be calculated by using the basic information that the devices provide. In the parking domain, one of the most relevant information to give to the user is the occupation of the parking spaces in real-time. Still, other data can be produced, such as the number of vehicles that parked and departed on a given day. Using these two values, we can also calculate the average time that a spot stays occupied to understand how much time visitors spend in the area. In contrast with the parking sensors, the radars give more information about the object they detect, and consequently, more measurements can be computed using the data from these devices. Since they provide the object's location, length, and velocity, the user can be alerted about traffic jams, average speeds in the area, and even the number of vehicles in each region. More specific information can also be known, such as traffic flow throughout time or the relation between a vehicle's class and speed. Furthermore, all of this information can be merged to offer the user a better picture of the real world.

The API will feed data to a web page where users can view an example of how PASMO data can be utilized. Furthermore, this application must make use of all the API's requests to be a more visual and straightforward way for administrators to examine the state of the system and its sensors.

Therefore, at least three pages need to be created to accomplish all the objectives discussed. The first page will include all the information related to the parking sensors, where it is

crucial to have a map with all the sensor's location and their current state. This feature will be the focus of the page since it is one of the most relevant information to the users. To be straightforward to read, the map needs icons representing the sensor's position and colors to describe the different states. The second page will cover information about the radars, where the central feature must be a chart displaying traffic flow in both the regions, accompanied by another graph showing the speeds registered in each radar. These components will mainly help the administrators to check for radar failures and act accordingly. The final page, still using the radar data, should offer an essential feature for visitors, a map with real-time traffic information. The most critical information that we can provide to both the residents and tourists is current data about congestions and waiting time, as well as an estimation of how many vehicles are in each area.

All these services need a way to be accessible to the public, needing another component to fulfill this task. A web server will be responsible for processing incoming network requests and redirect them to the API server, the documentation page, or the web application.

As stated before, the traffic classification solution will not be available publicly, staying independent from the system detailed above. The development of this component requires an analysis of both hardware and software solutions. Beginning with the latter, it is well established in the literature that using an object detection algorithm is the best way to achieve quality traffic classification when using video from cameras. Since the detectors are based on CNNs, we will need to train a model for the algorithm to learn to classify traffic. The data created by this component can then be fused with radar values to produce more precise information. The object detector model needs to be running on some type of machine, desktop, or board. The desktop can handle a large model because of its processing power, which will provide better detection. However, the board is more compact, yet less powerful, needing a smaller model and costing in accuracy. Still, this board can be applied in the RSU, capturing the video directly from the camera, while the desktop needs a live broadcast, meaning that a stream needs to be sent from the RSU to the building, costing in time and bandwidth. One thing to note is that even if the board is the better solution, the training process still needs to be executed in a powerful machine. Using a board with a weak GPU(performance-wise) is not efficient because the process takes too long to complete.

The laid down requirements should be sufficient to achieve all the goals for this dissertation but, other minor details or components may be added along with the implementation of the system if needed.

3.3 Proposed Solution

The final proposed solution expresses a lot of valuable data about the region, such as the number of vehicles flowing between areas, traffic classes, parking sensors states, number of vehicles parking and departing, and more. All these variables help tourists to avoid stressful situations like traffic congestions or looking for a parking lot. The system not only helps the tourists but the city consul too, with data that describes the critical times of the year, in terms of traffic congestion. It also presents a solution for the poor traffic classification

produced by the radars. The goal is to combine different sensor data and create a platform where the public can access useful information.

As stated previously, the API platform stores data from all the sensors in different databases, utilized according to the context of the problem. SCoT will be the solution used for data storage, being accessed by both the API and the traffic classification system. Regarding the API the objective is to create as much data as possible using the databases queries. This approach will make the process more efficient since if we retrieve data from SCoT to then be heavily modified in the API, the response time will increase, costing in performance.

The server-side implements two components, the API and the front-end, that will perform data processing in distinct ways. The former gets the sensor data from SCoT to be able to compute information for its requests. The latter accesses the API and build charts or maps from the data given by its requests. Next, it establishes communication with a client so that the page is displayed on its web browser.

Most of the frameworks developed to build APIs can't handle multiple requests by themselves, usually only serving one client at a time. The solution to this problem is to implement a component that offers load balance multiplying the resources and making the process of getting data more efficient. Another issue appears when discussing the fact that these services must be public and the clients need t to be forwarded to the correct application. The solution comes in the form of a reverse proxy server that provides features that make the communication between clients and servers much faster and smoother, such as caching common content. This component also defines the traffic allowed on certain ports and from what sources, plus the URL paths where the services are implemented.

Discussing the front-end component, it will provide three dashboard-type pages and a map with real-time traffic data. To understand how to build these types of pages, it's better to start from the end, i.e., thinking about what features the users will have access to. Use cases are used in these kinds of situations, defining the main features of a system and their interaction with the user. In the context of this system, every user performs the same role, which is consulting the dashboard for information.

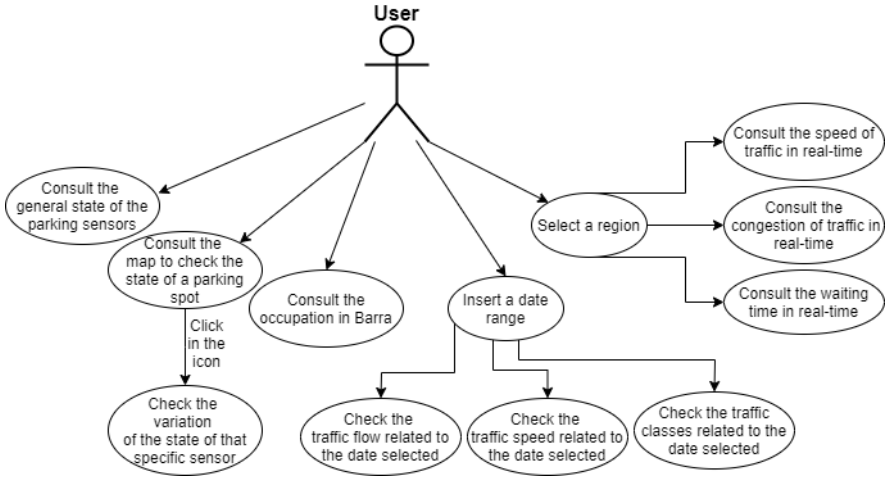


Figure 3.3: Use cases diagram.

After defining the main features, and the user's interactions, several mockups were created as user interface solutions for the four pages.

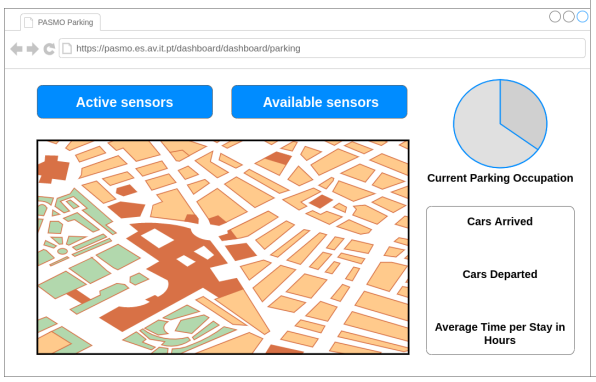


Figure 3.4: Parking Dashboard page mockup.

Figure 3.5: Radar Dashboard page mockup.

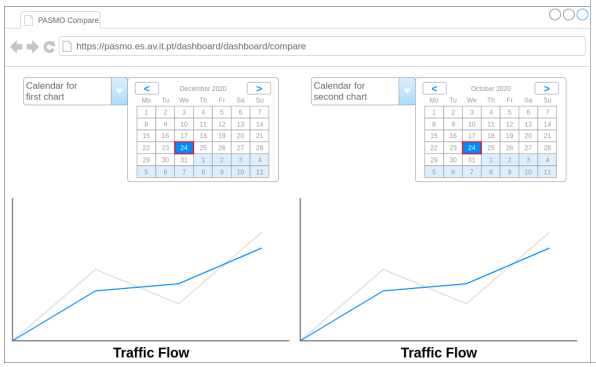


Figure 3.6: Compare Dates page mockup.

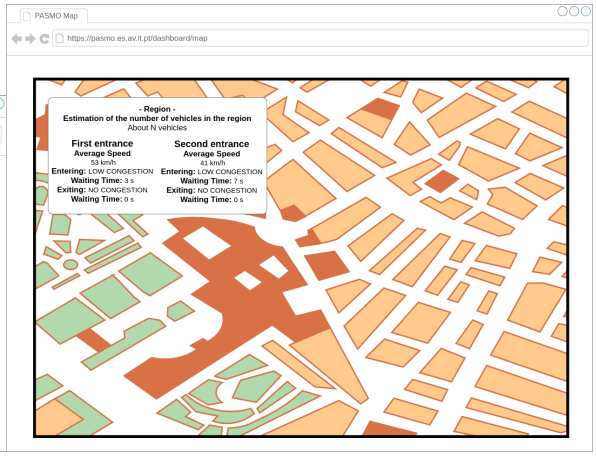


Figure 3.7: Map page mockup.

Firstly, the parking dashboard will display the sensor's location and current state, described by its color green(free), red(occupied), or blue(inactive). Above it, the number of active and available sensors will be given followed by the current occupation and details about other events. The radar dashboard's main goal is to provide information about traffic flow and vehicle speed. This page will have two line charts, one describing the variation of traffic flow and the other displaying the vehicle's speed in each radar. To the left of the last graph, a pie chart will show the percentage of vehicles of each type, pedestrian, bikes, cars, and heavy. All these elements will be influenced by a calendar found on the top of the page, where the user can filter information by selecting a particular time interval. The page "Compare Dates" provides a way for users to compare data from two dates, side by side, by replicating the traffic flow chart from the radar dashboard. The final page will hold a map, where users can view real-time traffic information. It will show a colored area that imitates both regions,

Barra and Costa Nova, which will activate a pop-up box when hovered. This element will provide information about the number of vehicles in the region hovered, plus crucial traffic data about its entrances, such as average speeds, congestion's details, and waiting time.

In the end, the system will be composed of three components, with the backend being the main element that serves the information to the public and the dashboard. This component will implement an API that serves as a link between the raw sensor data stored in the SCoT platform and the end-user. It works not only as a standardizer of data but also as a security measure since SCoT could not be linked directly to the front-end. The dashboard will utilize the requests available in the API to build charts, maps and display information in a way that a regular user can understand. This element composes the front-end part of the system, where information about the regions, its traffic, and parking state is displayed in comprehensible form.

Thus, if the user wants to check the state of traffic in real-time, for example, it will access the dashboard through its browser, communicating with the front-end. This component will then perform one or more requests to the API depending on the data needed to build the map, in this case, with all the information. Then, the back-end establishes a connection with SCoT, retrieving the necessary data to fulfill the request. After the API returns the standardized data, the front-end builds the page and communicates with the web browser so that the user can visualize the information requested. This flow is illustrated in Figure 3.8.

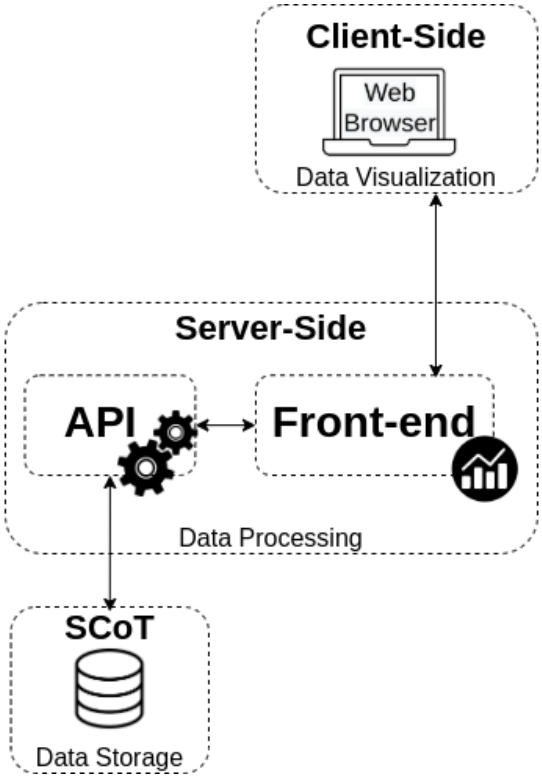


Figure 3.8: Architecture of the system that provide the web services.

The solution to the last component, which performs the task of traffic classification and independent from the system above, was to use an object detection algorithm to train a model

and incorporate it into the system. As stated before, the model can run either on a desktop or a specific board. In the context of this dissertation, a board with a good GPU is the best option since it can be later embedded into the RSU, getting the video directly to the camera. With this approach, the information transferred to the building is much smaller. If a desktop were to be used, the RSU needed to send a live video. Therefore, using the board, the video is directly fed into the device, and only the data processed is sent, saving in bandwidth. Since the board is the better choice, in this case, we have to train a smaller model of the detection algorithm. Even though the board has good performance, it can't handle large models with too many convolutional layers in real-time.

The information from the detection needs to be available to a program, which is a separate component that merges the data. We will call this piece of software the Fuser. This program needs to be connected to the object detector and SCoT so that it can receive data from both the radars and the detection algorithm. Finally, to better visualize the data that is being produced, and for debugging purposes, a web app is created. This component will communicate with the Fuser so that this can provide all the data to be displayed. This application will display a map with icons describing the position and class of the traffic alongside the real-time stream.

Therefore, the system will be composed of two algorithms that fetch and provide data to various components. The CCTV video will be feed into the object detector so that it can perform the classification of traffic. This component will provide the detection data, mainly coordinates and sizes of the bounding boxes generated, to the Fuser using a communication established at the start. The Fuser will access SCoT to retrieve radar data and merge it with the information from the detection. The final step is to send this processed information to the web application so that this component can build a page with it to be presented in a web browser. The flow described in this paragraph is illustrated in Figure 3.9.

One important thing to note is that the system lay down above is for test purposes. The objective of using a board is to embed it in the RSU, which will minimize time differences between the radar data stored and the stream considering both the radar and webcam will be connected to the board. The detector will get the video directly from the camera, and the Fuser will get the traffic data right from the radar.

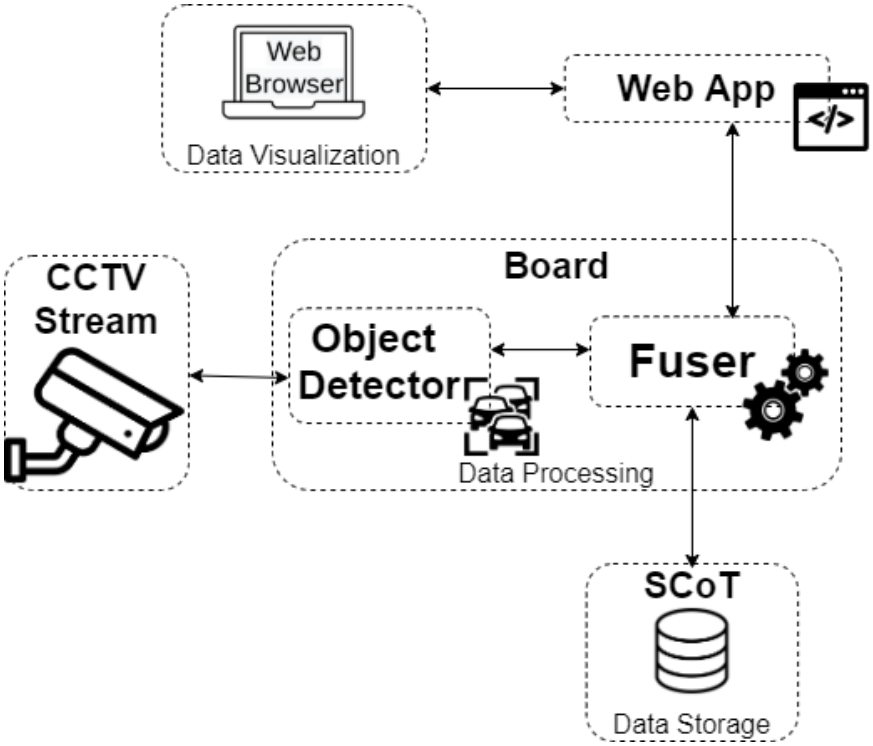


Figure 3.9: Architecture of the sensor fusion system.

CHAPTER 4

Implementation

With all requirements for the system laid down, the implementation came about. Some challenges that are very important to take into consideration when talking about public APIs are their speed, the usefulness and accurateness of the data provided, and how accessible it is. Users need to have fast, effortless, and reliable access to the service while receiving accurate and useful information. On the other hand, the difficulties of dashboard development are mainly the way the data is presented to the end-user. This data needs to be very organized and clear to make its analysis simple. Many times it's clear to the developer how the application works and what the data tells, but not to the user. Documentation can easily solve these kinds of problems, getting the user to understand what the information means or how to interact with the application. Finally, when discussing traffic recognition, it is crucial to take into consideration the error associated with classification results given by detection algorithms and how to properly fuse data from two sensors to have a better picture of the environment. Furthermore, when discussing CCTV images, it is essential to consider both the position of the cameras and their image quality. Other common problems are the synchronization between the radars and cameras and how to solve situations where one detects an object while the other doesn't. Nonetheless, these challenges are quite intriguing to solve to create a system that provides the necessary information for tourists so that they can have a better time when visiting the regions of Barra and Costa Nova.

4.1 Platform

This system provides two web services, an API to access information processed using sensor data, and a dashboard that provides a user-friendly way to view and analyze this information.

4.1.1 System Structure

Beginning from the data storage component, the SCoT platform stores data from all the sensors in different databases, utilized according to the context of the problem, them being InfluxDB and PostgreSQL.

In the server-side, the API will establish communication with SCoT, using the HTTP or TLS1.3 depending on the database. When considering the framework that would be utilized to develop this component, two popular ones written in Python 3.7 were considered. These were Django and Flask, the most complete and with better support used to develop web services. Although both are applied in web development, there is a clear difference in the context of their usage. Django is a full-stack web framework, whereas Flask is a micro and lightweight framework. Django is primarily used to develop a full web application, taking advantage of its features, such as Django's Admin page and built-in database support. Flask, on the other hand, is primarily used in an SOA context when building API services. It is also lighter and faster than Django, making it the perfect choice in this context. Nonetheless, other types of software can be utilized to help with building not only the API server but its documentation page as well. These are used to describe the components and structure of the API before developing it, focusing first on the essential feature that the service needs to provide. Swagger was chosen for this task since it offers a complete ecosystem of tools that help with the design, development, and documentation of RESTful services. Furthermore, requests can be tested on the page, giving users an easy way to experiment with API calls without the use of other applications like Postman. The documentation page generated by Swagger will be part of the front-end alongside the dashboard.

In the universe of frameworks used to build web applications, React, Angular, and Vue stand out in popularity, community support, and runtime performance. Unlike React and Vue, Angular is very feature-rich, i.e., many things that are needed to build applications are already built-in. This characteristic is an advantage for Angular considering the built-in features will always be compatible with the framework. Plus, both Angular and Vue separate the HTML and the script language used, unlike React. The structure used is more of a personal preference than an objective advantage, but it needs to be considered nonetheless. Due to the reasons stated, the framework chosen to develop the front-end application was Angular for its feature-rich nature and structure.

The last component on the server-side, that will be connected to both the API and the front-end is the reverse proxy. Nginx will fulfill this task, staying at the edge of the server, forwarding client HTTP requests to the right service. To be able to perform this action, both the API and the front-end need to be connected to it, communicating thought HTTP as well. The dashboard will also utilize Nginx to send its requests to the API so that it can get the necessary data to build the pages to send to the users.

Figure 4.1 illustrates the structure of the system, as well as the interactions described in the previous paragrahps.

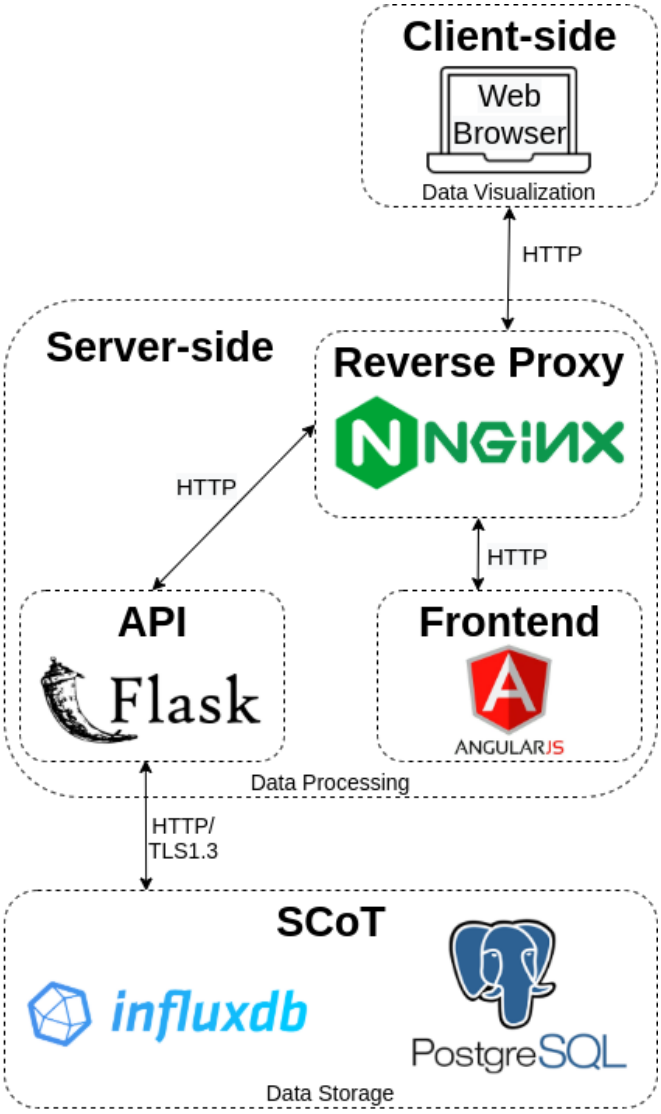


Figure 4.1: System structure of the platform that provides web services.

4.1.2 Backend

The backend is the core of the system, connecting all the components, and establishing a link between the user and the raw sensor data that is hard to read. In the context of this dissertation, the dashboard uses the processed information that the backend provides but, one must not forget that because API will be public, the data will be used by other users, most likely at the same time. To develop this service and make it accessible to the public, this component needed to have several layers of software, each one with different responsibilities. Figure 4.2 describes the structure of the backend, in which the client that is communicating with the service will go through several layers of software to reach the Flask server, where the sensor data is processed. Nginx is responsible to send the user requests to the appropriate service while the uWSGI provides load balance.

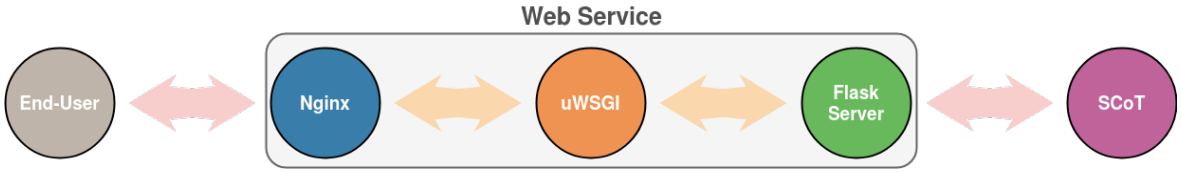


Figure 4.2: Software layers of the web service component.

4.1.2.1 Swagger

As stated before, building APIs from scratch can be a difficult challenge. Plus, when the development begins by trying to create data without a well-established use case, some information may end up never used because it is simply not useful. Thus, the process of building an API should begin with establishing what information each request will provide. Some software frameworks, like Swagger, help with this process by providing an environment where the developer can modify the API's requests, their parameters, and the data they provided. Swagger is an open-source software with a vast ecosystem of tools that help design, build, and document RESTful web services. By using the Swagger Editor tool, it is possible to generate a well-structured Flask server with a YAML file that describes the API's specifications. This file specifies all the API requests, as well as their parameters and data models. Furthermore, using another tool, called Swagger UI, it is possible to generate a documentation page from a JSON conversion of that same YAML file.

We start to set some essential values before working on the section where all the requests are defined. These are the host and basePath, specifying the URL where the server will reside, which is pasmo.es.av.it.pt, and the base path for the application, which is */api*. As disclosed before, there are two key domains when it comes to sensor data, them being parking and radars. Thus, two main paths were defined, */parking* and */radars*. As the names imply, the first describes the requests that use data from the parking sensors, while the second describes the ones that utilize data from the radars. This separation by domains helps the user identify what data belongs to what type of sensor. After this configuration is set, the requests can be designed. Usually, these hold several parameters where the user can send values to filter information and be more specific with how and what data they want. They are defined by two types, one of them being query that is attached to the end of an URL, and path, which is a part of an URL. Even though every request will be detailed individually in the following paragraphs, Table 4.1 provides a compact way to visualize all of them and their query parameters. The first two need to be provided in a datetime format, and it is here where the user defines the period from when the data will be retrieved. The other parameters, order, limit and offset, give the user more freedom to work with the sensor data. The first field describes how the data is ordered, ascendant and descendant, the second defines a limit of information given, and the third an offset. The state parameter describes the state of the parking sensors, available or active, and the last but one(day) specifies a date. Finally, groupby defines the amount of time, in seconds, that the data will be aggregated. This parameter is used in requests where there is some calculation. The user can group the data

by any amount of time, within limits, giving the possibility of knowing an average of a value per hour, for example.

	initialDate	finalDate	order	limit	offset	state	day	groupby
/parking	✓*	✓*	✓	✓	✓			
/parking/{sensorID}	✓*	✓*	✓	✓	✓			
/parking/availableSensors						✓*		
/parking/events							✓*	
/parking/latestValues								
/radars								
/radars/location								
/radars/vehicleEstimation								
/radars/{radar_id}	✓*	✓*	✓	✓	✓			
/radars/{radar_id}/{measurement}	✓*	✓*	✓	✓	✓			✓
/radars/events/{region}/{event}	✓*	✓*	✓	✓	✓			✓

Table 4.1: API query parameters per request. The checkmark tells what parameters are used while the asterisk describes the ones that are required.

Regarding the parking domain, five requests were defined. The first two, */parking* and */parking/{sensorID}*, have the same five query parameters, with two of them being required(*initialDate* and *finalDate*). The difference between these two relies on the path parameter present on the second request, where the user can provide a sensor ID. Both these requests will give the user a list of JSON objects describing the state of a specific sensor.

```
{
    "sensor_id": (integer),
    "status": (integer),
    "timestamp": (datatime)
}
```

Listing 4.1: Response from the requests */parking* and */parking/{sensorID}*

The next request, */parking/availableSensors*, gives the identification and location of the sensors, depending on the parameters state, which can only be available or active. The meaning of these values is detailed when discussing the Flask server. The result of this request is a JSON object specifying the location of all the sensors with the given state.

```
{
  "sensor_info": [
    {
      "sensor_id": (integer),
      "lat": (double),
      "long": (double)
    }
  ],
  "state": (string)
}
```

Listing 4.2: Response from the requests */parking/availableSensors*

The fourth request in the table only has one field, *day*, which is specified in the format date. It provides a list of JSON objects describing the vehicles that parked, departed, and the average time of the stay. This data relates to the day given.

```
{
    "arrived": (integer),
    "departed": (integer),
    "avgTime": (integer)
}
```

Listing 4.3: Response from the requests */parking/events*

The last request of this domain, */parking/latestValues*, doesn't require any parameters and gives the latest value posted by every sensor in a certain time window. This period will be clarified when discussing the Flask server. The request provides the same data type as the first two, */parking* and */parking/{sensorID}*.

For the radar domain, six requests were defined. The first, */radars*, returns recent information provided by the radars. No parameters need to be provided, and it gives a list of JSON objects describing all the values provided by the radar.

```
{
    "radar_id": (string),
    "radar_lat": (double),
    "radar.lon": (double),
    "radar_azm": (double),
    "xSpeed": (double),
    "ySpeed": (double),
    "xPoint": (double),
    "yPoint": (double),
    "oLength": (double),
    "object_id": (integer)
}
```

Listing 4.4: Response from the requests */radars*

The next request of this domain, */radars/location*, also doesn't need parameters and simply returns a list of JSON objects that describe the location of a radar(latitude and longitude) together with its identification.

```
{
    "radar_id": (string),
    "lat": (double),
    "long": (double)
```

```
}
```

Listing 4.5: Response from the requests */radars/location*

Another request that doesn't require parameters is */radars/vehicleEstimation*, and gives a JSON object with an estimation of the current number of vehicles in Barra and Costa Nova.

```
{
    "costa_nova": (integer),
    "barra": (integer)
}
```

Listing 4.6: Response from the requests */radars/vehicleEstimation*

The two following requests, */radars/{radar_id}* and */radars/{radar_id}/{measurement}*, have many fields in common. Firstly, radar identification needs to be provided in the path, with the only possible values being *ponte*, *dunaMeio*, *riaAtiva*, and *all*. The query parameters *initialDate*, *finalDate*, *order*, *limit*, and *offset* are present on both requests, with the first two being required. These fields perform the same functions described previously. The difference between the two is that the request */radars/{radar_id}/{measurement}* has one more path that filters data by type of measure(*speed* or *class*) as well as a query parameter(*groupby*) that can group the results by a certain length of time. Although similar in parameters, these requests differ a lot in the structure of the data provided. The first one returns a list of JSON objects that describe the relationship between vehicle class and speed.

```
{
    "object_id": (integer),
    "speed": (double),
    "class": (integer),
    "device": (string),
    "timestamp": (datetime)
}
```

Listing 4.7: Response from the requests */radars/{radar_id}*

The second one also gives a list of JSON objects but describing only one of the two measurements. The data type varies depending on the one that users select, being the left JSON the result of choosing *class*, and the right object the result of choosing *speed*.

```
{
    "timestamp": (datetime),
    "device": (string),
    "class": (integer)
}
```

Listing 4.8: Response from the requests */radars/{radar_id}/{measurement}* by choosing *class*

```
{
    "timestamp": (datetime),
    "device": (string),
    "speed_in": (integer),
    "speed_out": (integer)
}
```

Listing 4.9: Response from the requests */radars/{radar_id}/{measurement}* by choosing *speed*

The last request, */radars/events/{region}/{event}*, describes several events for the different regions and radars. These events can either be *cars_in*, describing the number of cars entering a region, *cars_out* describing the number of cars leaving, and *cars* describing the traffic flow. In the path parameter *region*, the following values are available, *barra*, *costa_nova*, *ponte*, *dunaMeio*, and *riaAtiva*. The data provided is simply a list of JSON objects detailing the number of vehicles detected.

```
{
    "timestamp": (datetime),
    "number_of_cars": (integer)
}
```

Listing 4.10: Response from the requests */radars/events/{region}/{event}*

After the configuration process, several files can be generated that will help with the development of the back-end server. Going to the "Generate Server" tab in the interface, we can see several types of frameworks. As discussed previously, Flask will be used in this context, so the "python-flask" field is selected to generate a Flask API server with all the requests configurated in the YAML file. We also need a JSON file that describes the configuration to generate the documentation page. Swagger UI offers this option when selecting "Convert and save as JSON" under the "File" tab. Using Swagger forces us to take a set back and think about the API and data structures first before implementing the code. This approach lays a clear ground on what features need to be implemented and what data to use, not wasting time on implementing requests that don't provide useful information.

4.1.2.2 API

Flask, more precisely Flask 1.1.2, has the responsibility of accessing and processing data from the different databases in SCoT, using multiple Python libraries. The platform has three databases implemented, Cassandra, InfluxDB, and PostgreSQL, that provide distinct features. Only the last two were used since they were enough to meet the requirements for this dissertation. InfluxDB is very fast when retrieving data from a single table, coming in hand when an operation needs tons of data from a single measurement. For example, if we want to know the average speed of vehicles from the last 24 hours, InfluxDB is the database to use

because it's only handling a single variable. In contrast, when a request needs to relate data from various tables, i.e., different measurements, PostgreSQL is utilized because InfluxDB does not offer this feature. For example, if we want to know the speed and class of a vehicle, PostgreSQL is used. This rule is applied to almost all requests, with some exceptions that we will be discussing later. Although this is the main followed rule when considering what database to use, other features such as simplicity of commands and speed are taken into consideration when making this choice. To access the databases described above, two Python libraries were used, *influxdb* to reach InfluxDB and *psycopg2* to access PostgreSQL.

At this point, we can start to work with the sensor data, but one detail that we tend to forget is the timestamp, more precisely in the context of timezones. Here it's necessary to decide if the API will give the timestamp of the data following the Greenwich Mean Time Zone (GMT) or the Western European Standard Time, which is the one where Portugal is located. Even though this is mainly a Portuguese project, it's better to provide the timestamp as the sensors publish it, which is in GMT, leaving the conversion to the frontend applications.

Another essential thing to discuss is the way the API handles errors and values that are not permitted. Swagger generated some of these validations, mostly the ones that check if a parameter is missing or invalid. The rest were created manually according to the context of the errors. One important thing to check is the *date* and *groupby* values. Some combinations of these two variables can result in a stall because of the amount of data being retrieved. Some requests like /parking have a specific limit on the time frame that a user can introduce. The result of this is an ERROR message telling the user that "Time interval higher than 1 day." The *groupby* hazard is solved by comparing the value introduced to the default value calculated. The default value is calculated for the circumstances where the user doesn't provide any *groupby* value. We will come to this value later, but for now, the only thing necessary to understand is that this restriction was created so that data doesn't exceed a particular size. One example of a stall happening without this regulation is when the user asks for data from a month, grouped by five seconds. The huge discrepancy in these values means that the database needs to retrieve a large amount of data, making the request slow. The validation is performed by checking if the *groupby* value introduced is less than 20% of the default. If this happens, an ERROR message is returned, informing the user that "groupby value too low!". This process ensures that the values given are always in a range where the request doesn't stall the API.

Other more request-specific validations were made, particularly in */parking/{sensorID}*, to make sure that the user doesn't introduce a sensor ID that doesn't exist. Finally, for requests where the user needs to provide a time frame, if there is no sensor data available from that period, a NOINFO message is given saying "No data found for the time interval provided."

Turning our attention to the requests, their features are laid down in Table 4.2.

Focusing on the parking domain first, almost all of the requests produce data from a single measurement(*state*), except for *availableSensors*. Even though it handles two measures(latitude and longitude), we found that it's much faster to use InfluxDB to retrieve these two then using PostgreSQL. The difference in speed happens because InfluxDB provides some useful

Request	Functionality
/parking and /parking/{sensorID}	Requests implemented to offer the users total abstract data, giving them the freedom to process it in any way intended.
/parking/availableSensors	Provides a quick view of the overall state of the parking sensors.
/parking/events	Implemented to offer some values where one can understand how the tourists/residents behave when going to Barra.
/radars	Implemented to provide the user with recent data about the state of the parking spots.
/radars	Offers real-time data about the traffic in all the entrances to both beach areas.
/radars/location	Provides the location of the radars so that admins can identify them.
/radars/vehicleEstimation	Offers an estimation on how many vehicles are currently in each region.
/radars/{radar_id}	Relates the vehicle's speed to its class, giving it identification and reporting which device detected the object.
/radars/{radar_id}/{measurement}	By user choice, provides the average speed of traffic or number of vehicles per class in a timeframe given.
/radars/events/{region}/{event}	Details various events, in a region or radar, used to describe traffic flow, i.e., vehicles entering and leaving.

Table 4.2: Functionality by request.

functions for these cases. Knowing this and going with the rules discussed above, all the data was retrieved from the InfluxDB database.

The first requests to be implemented were the ones that provide abstract data. The first retrieves all the states that the sensors publish within a specified period. One can imagine the quantity of data produced by hundreds of devices as the interval specified increases, with tests showing around 4 to 6MB stored per day. Since we cannot reduce the size of the data without affecting the easy access a RESTful service provides, the interval was limited to a time frame of one day. Even though this limitation exists, the user can still retrieve data from an entire month, for example, by making multiple requests and combining them. While this process seems painfully slow, it is faster than getting all the data at once because the API has a load balance mechanism, making it faster to process multiple small requests rather than a large-sized request.

Here, the API simply gets the values from InfluxDB, without posterior calculations. The data provided by this request gives the user information about the state of all the sensors(0 or 1, meaning free or occupied respectively), expressed by the device tag, followed by the time of the publish, represented by the timestamp tag. Regarding error handling, the API will return a warning when the interval provided is higher than one day, if the two required parameters are missing or if some value is not accepted, for example, the way to order the data.

The second request, just like the previous, was also implemented to provide abstract data, but only giving information about one sensor. The structure of the response provided is also the same as the */parking* request(state, device, and timestamp), as established previously. The values provided doesn't suffer any modification after the fetch from the database, expressing the exact data that the sensors produce.

The purpose of this request is to provide the user with a way to examine the variation of

a single parking sensor. In this context, where the user wants the information of a specific device, it is better to use this request rather than */parking* because the data given by the API is smaller, not exceeding 100KB per day. Furthermore, because the database filters the devices by their IDs, it is much more efficient than fetching all the data and search for the desired sensor after. As for error handling, the same rules are applied to this request as for the */parking* one. The only thing added is the verification if the sensor exists, informing the user if this fails.

To offer information about the overall state of the parking sensors, a request was implemented to provide their ID and location according to the parameter chosen by the user. This parameter can have the value 'active,' which gives only the active sensors, or 'available,' which returns all the sensors, active or inactive. Active sensors are described as ones that had published data in the last 24 hours, while the available sensors represent all the sensors that published data in SCoT in the last 90 days. It is essential to define this three month period since returning all the sensors that ever published data in SCoT would be an inaccurate representation of reality. It will give sensors that aren't even installed anymore because some of them were removed, leaving SCoT with data both from the removed and new sensors. With the implementation of this interval, the number of available sensors declined roughly 5%.

The data provided by this request is the location of the sensors, expressed in latitude and longitude. Going by the main rule established previously, PostgreSQL should be used, since we are trying to retrieve more than one measurement. Yet, the process of retrieving the latitude and longitude of all the sensors takes forever. This slowness happens because of the way the database is organized in the database. Values from all the sensors are inserted in one table, forcing PostgreSQL to go through all that information to find the data we are looking for. Thankfully, InfluxDB offers many useful methods that help in these types of situations. By using the function LAST while grouping the data by sensor ID, InfluxDB can return the last values published by each sensor. Since the devices always publish their location alongside their state, we can make sure that the latest published position is being retrieved. This process also helps to identify sensors that are providing the wrong latitude and longitude since we are getting its most recent value. Even though this method needs two queries to be performed, it is still faster than PostgreSQL.

Despite trying to filter off sensors that don't exist or, when specified, inactive ones, some of the locations return wrong values, in most cases, near zero for both latitude and longitude. This error occurs because of hardware malfunction since most of the sensors that give these positions are already inactive. When discussing error handling, the same rules are applied here as for the */parking* request, with the addition of the validation of the state parameter.

One important piece of information that can be computed from the sensors data is how much time the people stay parked and the ratio of vehicles that parked and departed. The states of the sensors are grouped by device ID and then fetched. Using this process, we get the variation in the state of each sensor throughout a particular day, being now possible to calculate all the three types of values mentioned. When the state changes from 0 to 1, free to occupied, it means that a vehicle just arrived at the spot and parked, while if it varies from 1

to 0, occupied to free, it means that it left the parking lot. After performing these verifications, the program will save the timestamp of each event as a "parking" or "departing" timestamp. The difference between these two values gives us the amount of time that a parking lot stayed occupied. However, other situations beyond the ones mentioned need to be considered. We need to save the timestamps when the first or last values of the block of data fetched identify a parked vehicle. If the first value is 1 we store it as a "parking" timestamp, and if the last is 1 we save it as a "departing" timestamp. This process also covers the circumstances where a vehicle is parked for more than 24 hours. With these situations covered, all possibilities will produce a pair of timestamp values, an arrival, and a departed timestamp. After this, the average is calculated, and the request's response structured, ready to be sent to the user.

A final request was implemented in this domain to provide users with recent data to apply in real-time scenarios. As stated before, some sensors are inactive, not expressing a recent picture of the parking lots, so the data needs to be filtered. Initially, the period considered was 24 hours, meaning that all the sensors that did not publish data in this time frame would not count to the calculation. The problem is that 24 hours is a long period to express real-time data. Still, this value can not be set too low, 10 minutes, for instance, because it discards vehicles that stay parked for hours. So the solution was to set the time frame as 10% more of the average time per stay, calculated in the */events* request. This process makes the data more accurate when comparing to using a low time frame by not discarding values and when comparing to using a higher time frame by not getting inactive sensors into the mix.

For the radar domain, five requests were defined, working with several measurements like vehicle class, speed, and position. Other values (latitude and longitude) were used, but regarding the radars themselves. Going by the rules discussed previously for choosing which database to use, the requests */radars* and */radars/{radar_id}* retrieve data from PostgreSQL while the others worked with InfluxDB.

The first request servers the same purpose as the first two from the parking domain, provide users with raw sensor data so that they have the freedom to create their own. It also provides recent data to be applied in real-time scenarios. However, a problem arises, which relates to the amount of data that the radars store. Data size was also an issue with the first two requests from the parking domain, but the radars produce 15 times more values than the parking sensors. The three devices combined publish almost 100MB of data per day, making it hard to send it all to the user at once. Nonetheless, most values, like speed and class of traffic, can be handled by other requests leaving only the position of the object detected for this one to manage. Since we cannot give more than half an hour of data to the user, or else it would take a long time to process, and the size of the data would be enormous, the solution was to use this request to provide the current state of the environment. This method was achieved by fetching the last 2 minutes of data published by all the radars, giving the user the ability to analyze the speed, position, and length of the traffic detected for real-time applications.

To provide the coordinates where each radar is installed, a new request was created. Even though it provides two measurements(latitude and longitude), the information was retrieved

from the InfluxDB database for the same reason as the */parking/availableSensors* request. It is faster to make two queries taking advantage of the function LAST InfluxDB provides. The final data provides the radar's ID, followed by their location(latitude and longitude).

Another crucial type of information is the one that describes traffic flow. A request was created to express this realm in both radars and regions. Here the user can select one of three events for a specific time frame. These events illustrate the number of cars entering a region/radar(*cars_in*), the number of vehicles exiting a region/radar(*cars_out*), and the amount of traffic flowing through a region/radar(*cars*), i.e., the difference between the *cars_in* and textitcars_out events. Before working with data fetch from the database, it is essential to define what does it mean to enter or exit a region or radar. Starting with the latter, the action of entering or exiting depends on the speed of the traffic detected. If the radar publishes a negative value, it means that the object is getting close to it, i.e., entering. On the other hand, if the speed is positive, it means it's moving further away from the radar, i.e., exiting. The difference between these two variables expresses the traffic flow in the device.

Examining traffic flow in the radars seems simple enough but, when it comes to the regions(Barra and Costa Nova), we need to consider the orientation of these devices since entering a radar doesn't always mean entering a region. To discuss this issue, we need to remember Figure 3.2, in Section 3.1, describing the direction of the devices as well as region limitations. There we can see that the radars "Ponte" and "Duna Meio" handle Barra's traffic, and "Duna Meio" and "Ria Ativa" manage Costa Nova.

Looking firstly at the radar "Ponte," we can see that it's facing the outside of Barra, meaning that whatever vehicles enter the device will also enter Barra. The same is true regarding vehicles exiting "Ponte," where traffic leaving the radar will, consequently, leave Barra. Turning our attention to "Duna Meio," this device operates in both regions while facing Barra. Knowing this, we can conclude that the vehicles leaving the "Duna Meio" are entering Barra while the ones approaching this device are leaving. Already we can write the equations that describe the traffic flow in Barra.

$$\begin{aligned} barra\_cars\_in &= cars\_in\_ponte + cars\_out\_dunaMeio \\ barra\_cars\_out &= cars\_out\_ponte + cars\_in\_dunaMeio) \end{aligned} \tag{4.1}$$

$$\begin{aligned} barra\_cars &= barra\_cars\_in - barra\_cars\_out \\ \Leftrightarrow barra\_cars &= (cars\_in\_ponte - cars\_out\_ponte) + \\ &\quad (cars\_out\_dunaMeio - cars\_in\_dunaMeio) \end{aligned} \tag{4.2}$$

As stated above, "Duna Meio" handles both areas, so it also plays a role in describing the traffic flow in Costa Nova. Since this radar is not facing Costa Nova, the vehicles that enter the device will also arrive in this region. The same is true for objects exiting the radar, in which case they will also leave Costa Nova. The only radar remaining is "Ria Ativa." This device is located at the bottom of Costa Nova and not facing the region, which puts it in the same scenario as "Duna Meio." This means that vehicles approaching the radar will be

entering the Costa Nova, while vehicles exiting the device will be leaving it. With all this information, we can define the equation that describes traffic flow in Costa Nova.

$$\begin{aligned} costaNova\_cars\_in &= cars\_in\_dunaMeio + cars\_in\_riaAtiva \\ costaNova\_cars\_out &= cars\_out\_dunaMeio + cars\_out\_riaAtiva \end{aligned} \tag{4.3}$$

$$\begin{aligned} costaNova\_cars &= costaNova\_cars\_in - costaNova\_cars\_out \\ \Leftrightarrow costaNova\_cars &= (cars\_in\_dunaMeio - cars\_out\_dunaMeio) + \\ &\quad (cars\_in\_riaAtiva - cars\_out\_riaAtiva) \end{aligned} \tag{4.4}$$

Figure 4.3 illustrates what was explained above to help visualize the different scenarios.

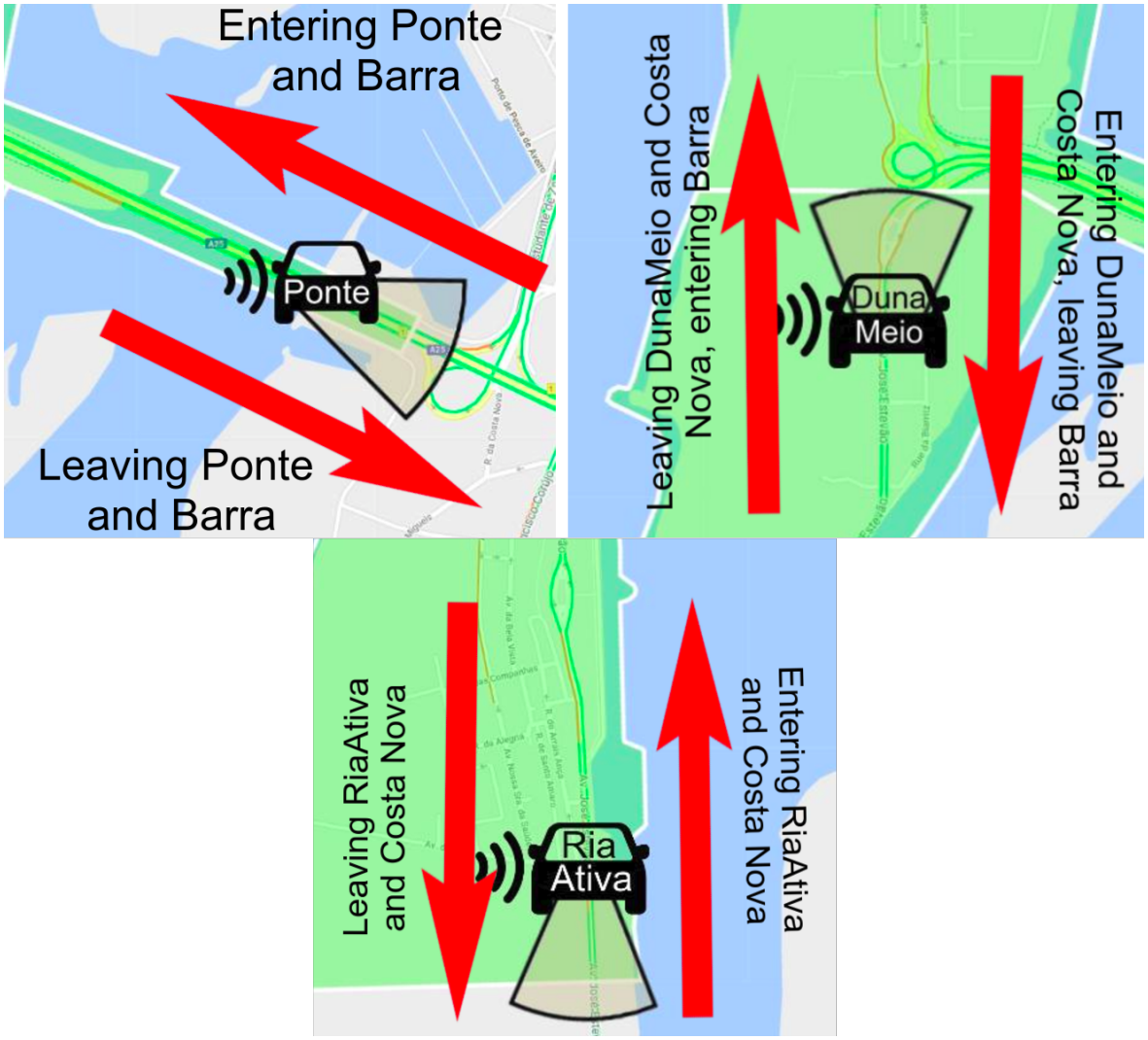


Figure 4.3: Ilustration of what was described in the previous paragraphs, using the map shown in Section 3.1.

Now that all the concepts are defined, it's time to discuss how to retrieve the information. Firstly, we need to count the number of objects with a positive and negative speed to find the vehicles entering and exiting a radar, respectively. Here, four queries were defined to perform

this calculation since the equations of the traffic flow have four variables. This process is achieved by using the function COUNT from InfluxDB, in conjunction with different filters applied in the WHERE clause. Each query is filtered by time, speed, and device to help us calculate all these values. They also take advantage of the GROUP BY clause to process information faster and returning smaller amounts of data while not compromising accuracy. In short, each query will count the vehicles detected in a specific length of time, defined by the GROUP BY. Without the use of this clause, the function would count all the values and return only one result. This information would be useless since we want variation through time, like, for example, how do the values change per hour. Thus, we need to group the data by the right length of time to accelerate the counting process. For instance, if we are fetching the data from a day while grouping the information by one hour, the database will return 24 values, making the process quite fast. However, if we group the data by one second, for the same time frame, there will be 86400 values, making the process much slower. The value of this clause is a significant problem since the API gives the user the possibility of choosing the amount of time(in seconds) to group the information. This number cannot be too low because we will arrive at the scenario where too much data is begin fetched, making the response slow. The solution to this issue is to limit the *groupby* to a value that varies depending on the time frame given. This limitation was already discussed previously, it being 20% of the default. This default variable will not only validate the user inputs but it will also be used when the *groupby* parameter is not specified. The default is calculated by running the number of seconds of the provided time frame through some equations. These were created with the goal of not exceeding the response size of 100KB.

After knowing the number of vehicles entering and exiting each radar, the equations described above were utilized to compute the traffic flow, with an InfluxDB function(CUMULATIVE_SUM) applied to each variable in the equation. This method will add each value to the previous one, giving a better picture of the variation of information through time. If this function wasn't used, each value would be independent of the rest, and the change in traffic data overtime would be hard to understand. To better visualize the influence of this function, two charts are displayed below, one showing the data where CUMULATIVE_SUM was applied(Figure 4.4), and the other where it was not(Figure 4.5).

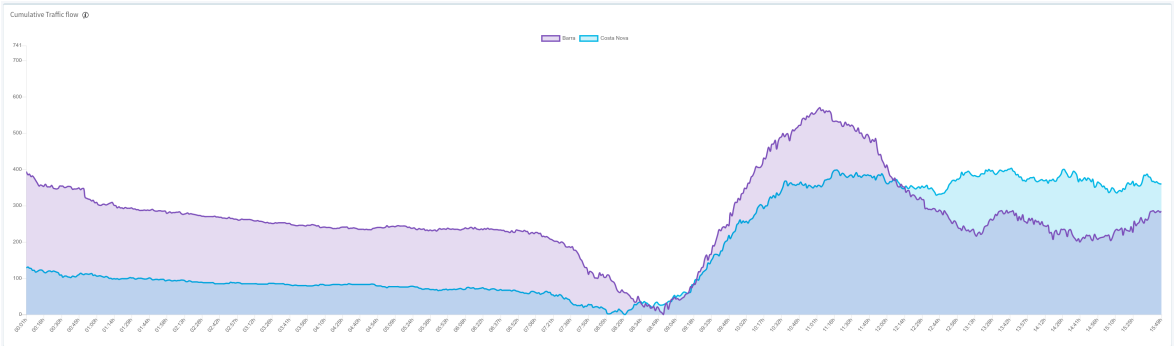


Figure 4.4: Chart that plots data where the CUMULATIVE_SUM function was applied.

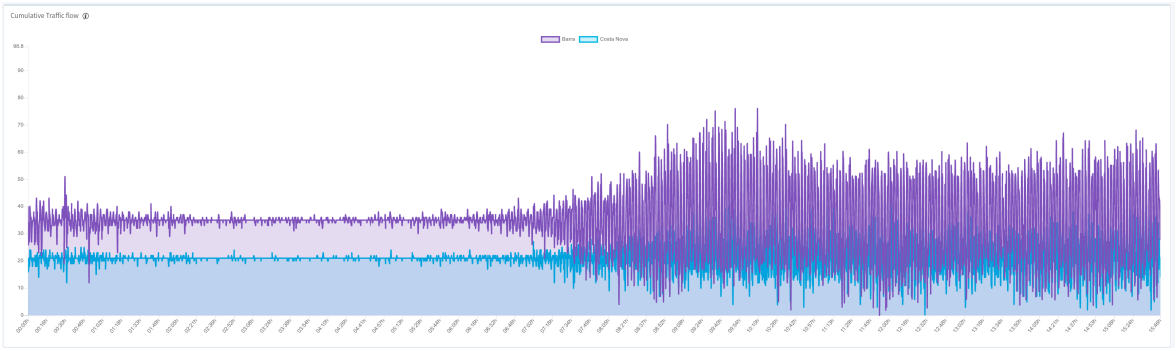


Figure 4.5: Chart that plots data where the CUMULATIVE_SUM function was not applied.

The values where the function was applied tell us much more about the traffic flow since the variation over time is apparent. Although the other chart provides a better description of the traffic flow at a precise time, the objective was to describe the traffic flow over time, making the process displayed in Figure 4.4 a better candidate.

In the end, only one more thing is changed, but only in the event *cars*, where a data shift occurs so that the minimum is zero, i.e., all the values are positive. This process does not affect the accuracy of the information since the variation of data will be the same, but better to visualize. In Section 4.1.3 when talking about the radar dashboard, this change will be easier to visualize and understand.

To understand the density of traffic in both regions, a request was created to provide an estimation of how many vehicles are present in Barra and Costa Nova at that time. It also provides the maximum number of traffic registered in these regions, so that it is easier to compare the congestion in the area. In a perfect scenario, where none of the radars fail, we can calculate the estimation by making a cumulative sum of the quantity of traffic since the time the devices were deployed. Next, after shifting all the data so that the minimum is zero, the last value would give us the number of vehicles in the region. However, this is not true in a real-life scenario because sometimes the devices fail or the platform needs a reset, leaving this period with no data. These situations are problematic when collecting traffic information because if a radar crashes when there are lots of vehicles entering a region, the calculation will not include all that traffic. Although, it can get worse when the radar restarts since all the traffic that entered when the device was down will now exit and be detected. So when analyzing the data, the user will suddenly see a ton of vehicles leaving the area without even entering. These inconveniences cause a problem in the estimation since this value will not correspond to reality.

To work around these failures, we only need to consider time frames where all the devices are up and publishing information. The first step is to find all the periods where a failure occurred. Device crashes can be found by searching for periods where there is no traffic. Yet, we have to be careful since this can also mean that there is low traffic circulation. For example, before sunrise, there are periods when there is no traffic, but that does not mean the devices are down. With these situations acknowledged, it was established that to be considered a

failure, 4 hours need to pass without the radar publishing any data. Then, a file will store the timestamps of the crashes and the ID of the radar that suffered problems. All of this is necessary to find optimal values for current and maximum vehicles in the regions. After this process, the timestamp of the last failure is saved, and we search for the max value begins. Once more, this search can only be made in intervals where the devices are not down. To find the current number of vehicles in the area, we use the *radars/events/* request to calculate the cumulative sum of traffic between the last failure and the present time. With this method, we assure that the crashes are not affecting the accuracy of the estimation. Note that the server repeats this process every day, updating the file with new device failures to provide logs of crashes and calculate the max value.

Even though the radar produces more values, two basic measurements are saved in InfluxDB, them being speed and class. Two requests were built around these, one to relate the two values and another to provide each one separated. The first request provides their relation alongside other types of data like object ID, device ID, and timestamp. Since we have a relation between two types of measurements, PostgreSQL is used because InfluxDB is not very efficient with relations between tables. The PostgreSQL database has a less straightforward way to retrieve information since the data is not identified by the radar name but by a generated ID. Moreover, all the topics save information in the same table, values, which means that data from the radars, parking sensors, or even other projects is all in one table. A clean way to solve this issue is by creating a dictionary that maps the radar names to the database IDs, helping with the code structure and making it easier to understand. When filtering for a specific radar, the WHERE clause will only have that generated ID, but when retrieving information from all the radars, we will need to filter by all three IDs. If we don't perform this filtering, all the other data from different sensors will be fetched, since there are all in one table. The usage of the function IN solves the problem, by filtering the data from the three radars like so, id IN (*id_ponte*, *id_dunaMeio*, *id_riaAtiva*). The data retrieved from the database doesn't need any change since it provides all the variables needed.

The other request that uses the two main measurements, gives the same values as the last one(class or speed), depending on what's specified by the user, independent of each other. However, this request is not limited by a time frame of one day like the other. The big difference between this request and the last one is that only one measurement is being worked on, and the information is grouped, making it faster to process. The *groupby* validation follows the same principle as the */radars/events* request, except the default value is generated using different equations but with the same goal of not exceeding the response size of 100KB.

In this request, InfluxDB is used since each call only returns a type of measure. The way we retrieve data is different for the two measurements because each one needs a different calculation. For the *speed*, the MEAN function is used to compute the average velocity of both the vehicles entering and exiting. For the *class*, we only need to separate the values 1, 2, 3, and 4, representing the different classes, and count each one. These values represent pedestrians, bikes, light, and heavy, respectively.

In the end, many more types of data could be created if several requests were to be merged.

However, most of the time, the more complex the data, the longer it will take to produce, slowing down the APIs response. Most of the requests presented were developed utilizing functions provided by the databases to create and fetch values faster. New models of data that required merging requests were not implemented because Python would need to handle these calculations making the process slower. The goal was to provide users with the measurements from the sensors and other types of data that could be created quickly using the database functions.

4.1.2.3 Load Balance and Reverse Proxy

As recommended by the Flask documentation [65], built-in server is not suitable for production and should not run by itself because it doesn't scale well, only serving one request at a time. Given that this public API is expected to handle multiple requests from various users, the solution is to use a Web Server Gateway Interface (WSGI) to perform load balance. Two of the most popular WSGI servers are uWSGI and Gunicorn. The developer's decision on which one to use seems to follow the easier-to-install route, choosing Gunicorn because it is faster to deploy applications. However, the results of the tests performed in [66] show that uWSGI has higher throughput and fewer errors when comparing to Gunicorn. The response time is also slower, making it a better choice.

Figure 4.6: Results from the tests performed in [66].

When it comes to its configuration, we have two variables that can be modified to boost load balance performance. These are the process and threads fields present in the command *uwsgi*. After performing tests with various configurations and analyzing the results, we

conclude that the configuration of 16 processes and 16 threads gives the best performance, being the fastest overall.

The uWSGI server, together with the documentation page, and the dashboard application, were implemented in independent containers, being Nginx's responsibility to provide the user with the right service. To have a sense of a service-oriented architecture, Docker was used to create, deploy, and run these services independently. Even though these applications will interact with each other, we can rest assure that if one crashes, it will not break the other ones. Docker-compose was also utilized so that we have a straightforward way to build and run all the services at the same time. Since Swagger provides a container in docker hub, in which we can build a documentation page from a JSON file, only two DockerFiles needed to be configured, one for the API and one for the frontend. This JSON file is the one generated back in the Swagger UI from the YAML configuration. For the other two services, Dockerfiles were created using images from the docker hub to establish base environments, more precisely "python:3.7-slim" to build and run the API, and "node" and "nginx:1.13.3-alpine" to build and run the frontend, respectively. The reason why the frontend needs two different images is to run the website in a JS-CSS-HTML format, making the image much smaller. This process of having two separate images, one for building porpuses and another for running the application, is called multi-stage building. This approach was also used in the API but using the same base image. However, here we run into some issues when trying to make use of the multi-stage building. This problem occurs because when using Python, we don't run any compiler that gives us an execution file. In other languages, like C, this approach is much simpler since we can build the program in an image and run the compiled file in another. However, what we can do is reduce the size of the packages used in the Flask server by taking advantage of Python virtual environments. The final goal is to have an isolated directory with all the built packages. Thus, we begin to install some dependencies on the base image, mainly for accessing the PostgreSQL database and run the uWSGI server. After this process, a virtual python environment is created, on the folder */opt/venv*, and all the necessary packages are installed using pip. In the end, we can make sure that */opt/venv* will hold all the binaries needed to run the Flask server. Thus, the only thing left to do is to copy this folder and all the code to the image that will run the API. Finally, the uWSGI server can be deployed port 8080, with all the configuration discussed previously. By using this approach, the image where the service is running becomes much smaller, making it easier to transfer between machines if necessary and not filling the system with unnecessary content.

As stated, the frontend will follow the multi-stage approach as well, but with different base images. This method will help reduce the size of the running image by a lot since *node* is quite large. Firstly, all the files are copied to the container, and all the packages specified in the file *package.json* are installed. The only thing left is to run a command(*ng build*) to compile the Angular app into a directory, generating a static page. Alongside this command, three options were introduced. The first one, *prod*, uses several techniques such as bundling methods and limited tree-shaking to join multiples files together and eliminate dead code. Another option used was *build-optimizer*, which increases the performance of the methods

used, making significant improvements, especially regarding bundle size(the single file created from multiple ones). The last one, *base-href*, defines the base URL of the application, which is */dashboard*. After the build stage is complete, we need to copy the static content generated by the building process, as well as an *nginx.conf* file, to the running image, *nginx:1.13.3-alpine*. The former goes to the folder */usr/share/nginx/html/dashboard* while the latter is moved to */etc/nginx/conf.d*. This last file will hold some configurations so that Nginx can deploy the webpage. We will discuss these configurations further ahead. Finally, by executing the command *nginx -g daemon off*, the service can start. Nginx will set the global directives according to the configuration file and run the process in the foreground. This is achieved by setting the daemon option of the command to *off*. This option prevents strange situations, like the container stopping immediately after starting, by allowing Docker to track the process properly.

Back to the configuration file, nginx.conf, we will define a server block, or also called context, that will serve our website. Firstly, we will use the *add_header* directive to set headers that will be sent to the client. The Access-Control-Allow-Origin was set to *, which authorizes requests outside the domain where the website is hosted. With the header set, it's time to work on the location block, which defines the path to the application. A location block is used to describe how Nginx should handle requests for different resources and URIs. In this case, we only need to define one location, to manage the dashboard requests, which is described by the following expression:

```
location ~ ^/dashboard
```

The "~" modifier is used to perform a case-sensitive comparison while the *^/dashboard* describes the path part of the URL. It's inside this block that the directives root, index, and *try_files* are defined. The first one describes the path for the folder that holds the static content, which is */usr/share/nginx/html*. The second defines files that will be used as the index(*index.html* or *index.htm*). The last directive checks whether the file or directory exists, redirecting to the content if it does or returning a 404 error if it doesn't.

At this stage, all the services are configured independently, missing the docker-compose, that will help us deploy all of them at the same time, and the main Nginx that is responsible for forwarding the user to the correct service. In the docker-compose file, both the documentation and the API services were configured to listen on port 8080 and the dashboard on port 80. Even though the Nginx previously configured serves the static content, it doesn't handle the other two services. For this, we need another Nginx to manage all the requests for the different applications.

However, we first need to discuss where it will be implemented so that users can access it from *pasmo.es.av.it.pt*. This domain is held in a server located in IT-UA that already serves some applications related to PASMO. Thus, it's possible to describe the three services in the Nginx file already present in the machine, not being necessary to create a new one. However, before we discuss the configuration, it is essential to understand how the file is structured.

Firstly, the directives *worker_processes* and *worker_connections* are defined by 1 and 1024, respectively. This layout means that this Nginx service can serve 1024 clients simultaneously. After this, an http block is described, to handle HTTP or HTTPS traffic, with only one server block inside, since we are only handling requests for one domain. Multiple upstream contexts are already defined inside the http block to proxy requests to the existing applications. We need to configure three more, one for each of our services, API, dashboard, and documentation. The upstream for the frontend was named *dash*, the API was defined by *restapi*, and the documentation was labeled as *swagger*. After this, three location contexts were defined to describe the paths for each service. In all of them, a *limit_except* context was configurated to accept only GET requests since we don't need other types of HTTP methods. The *proxy_pass* directive was defined according to each service, point to the appropriate upstream described above. For example, for the API, the *proxy_pass* value should be *http://restapi*.

In the end, all these applications will be under the domain pasmo.es.av.it.pt, in different paths. The API will be under */api*, the */docs* will serve the documentation, and the */dashboard* will host the frontend application.

4.1.3 Frontend

This section describes the process of building the user interface for the dashboard. The primary objective was to have a more "admin page" look while also providing relevant information for the average user. The website will be displayed as a Single Page Application(SPA), meaning that the service will rewrite the current web page with new data instead of loading an entirely new page. This approach offers users smoother navigation between the content of the website. Accordingly, the fixed structure is composed of a navigation bar, a sidebar, and a footer. The first only contains the PASMO logo and a button to collapse the sidebar. The second component has all the pages listed, grouped by "Documentation" and "Main Navigation." The former has two elements that correspond to the documentation of the website and the API while the latter is composed of "Home," "Dashboard," "Devices," and "Map." The "Dashboard" extends into three components "Parking Dashboard," "Radar Dashboard," and "Compare Dates." The footer simply shows all the parties involved in the project PASMO.

When a user goes to URL pasmo.es.av.it.pt/dashboard the home page will appear, corresponding to the option "Home" on the sidebar, showing the full name of the project as well as a button to learn more about it. When clicked, it will redirect the user to the main website of PASMO that provides all the details about the project.

The first element when extending the second option in the sidebar is "Parking Dashboard," created to display content about the parking sensors. The first row of items provides information about the total number of sensors, the active ones, and the current number of parked vehicles in those spots. This information is crucial to rapidly check if there are problems with a large number of sensors since if the number of active sensors is low compared to the available ones, it means that too many sensors aren't communicating.

When examining the right side, two elements were built to describe more quantifiable information. The one on top provides a percentage of the current occupation of the parking

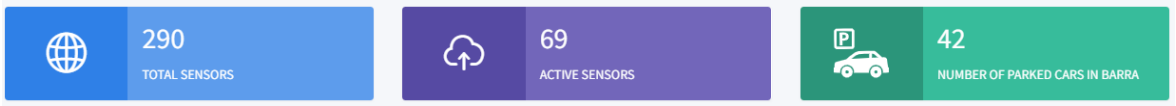


Figure 4.7: Top area of the Parking Dashboard.

lots. The one underneath displays the number of vehicles that arrived and departed plus the average parking time. This information is an aggregation from all the sensors and describes the present day. These two elements provide a straightforward way to identify the occupation in Barra as well as the average time that people spend in the region.

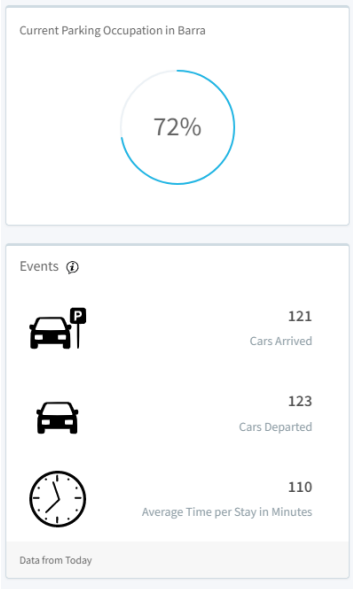


Figure 4.8: Right area of the Parking Dashboard.

The focal point of this page, being essential for both users and administrators, is the map displayed in the middle of the page. This element provides the location of all the parking sensors as well as their current state. The latter is described by the color of the icon, with red being the parking lots that are occupied, green the ones free, and blue depicting inactive sensors. The main objective of this map is to give the user a way to search for free parking when going to Barra and for an administrator to check which sensors are down. The last component presents a chart displaying the variation in the state of a selected sensor. The user can choose it by clicking an icon on the map, which will make a chart appear describing its state in relation to time, as well as the identification of the sensor in the title. If the icon selected represents a sensor that is down, the title and the content of the element alert the user that no information is available.

The second element of the second option on the sidebar, "Radar Dashboard," displays information related to the radars. The main objective of this page is to give the admin the possibility to quickly identify radar failures and provide a clear picture of the state of the traffic. Besides, it offers the council of Ílhavo a straightforward way to examine the traffic throughout the year so that traffic trends can be studied and understood. The page starts

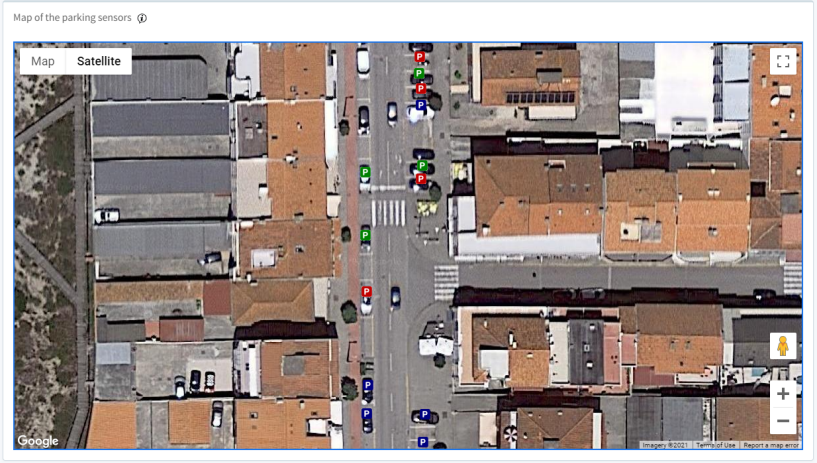


Figure 4.9: Map detailing all the sensor location and state in the Parking Dashboard.

with a header that provides two ways to filter the data presented in the charts. In the first dropdown, identified as "Select Date," users can select a date or a range of time, while the second one, named "Filter By," filters data by regions or radars. When the first filter is pressed, a calendar appears with some fast click options on the left side. These options will display the data from the recent day, the past day, and the last 7 or 30 days. By using the calendar, any range of time can be selected between the dates 25 January 2020, which is the day the radars start to collect data, and the current day. If the time frame picked extends to the present day, the charts will update every 5 minutes so that the user has the most recent information. When the period selected does not have any sensor information, an alert message will appear informing the user of that fact. This condition only occurs when all the radars stop communicating, making it impossible to collect data while they are down. The other filter in this header can supply information about the three radars or the two regions. Note that the date filter affects all the charts on the page, while the other one only changes the first two graphs. A grey line separates the charts affected by this filter from the ones that are not.

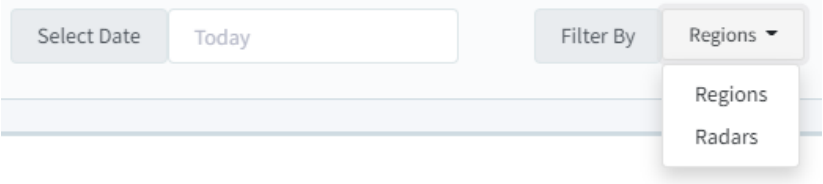


Figure 4.10: Main two filters that affect all the charts in the page.

The first chart shows the variation of traffic depending on the time selected. This variation, as explained in Section 4.1.3, is calculated from a cumulative sum of the difference of vehicles entering and exiting. Here we can better understand why it is beneficial to shift the values to make the minimum of zero. Since the significant part of this chart is the variation of the curve, it is not very relevant to have values below zero. The rising of the curve means that more vehicles are entering than exiting the region/radar, while its decline means that more are leaving. The flatter it is, the less the difference between vehicles entering and leaving, or

it could also mean that traffic is low. The values, by themselves, don't give much information. However, the comparison between them offers the user a good picture of what is happening in the regions. For example, Figure 4.11 shows three instances of time that depict three different values, with the last two expressing the maximum and the minimum of that day.

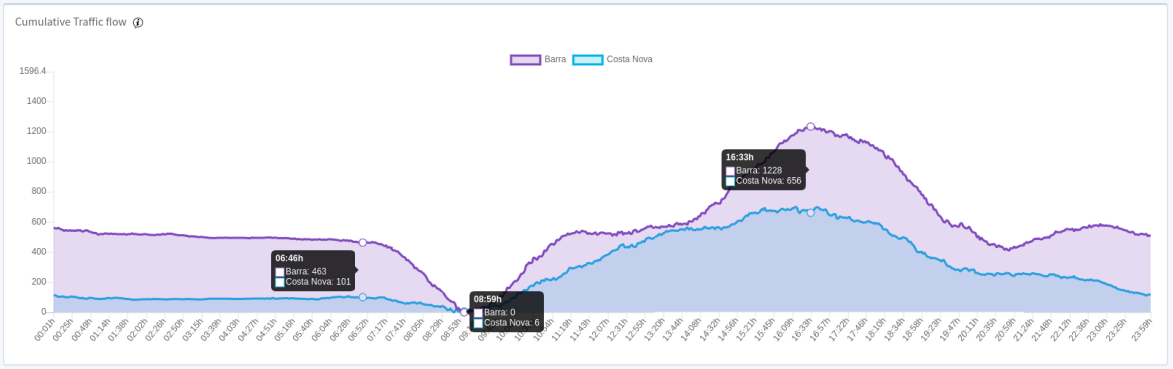


Figure 4.11: Chart that displays the cumulative traffic flow on the regions Barra and Costa Nova.

Analyzing this, we can notice that the lowest number of vehicles in Barra was in the morning while the highest was in the afternoon. Furthermore, when comparing the two first values, we can observe that the difference between vehicles entering and leaving Barra drops from 476 to 0. Considering these values are from the morning, we can conclude that this data represents people leaving to go to work. However, when comparing with Costa Nova, the discrepancy in values for the same two instances is not so large, going from 101 to 6. This difference means that Barra is a larger residential zone where Costa Nova appears to be a route to reach the highway. This fact can be better noticed when looking at the same chart describing the radars. Looking at Figure 4.12, we can observe that more vehicles are entering Ria Ativa (South Costa Nova entrance) and leaving Duna Meio (North entrance) at almost the same rate, seen by the rise and decline of the curve, respectively. Thus, concluding that many people use Costa Nova in the morning on working days as a way to reach the highway.

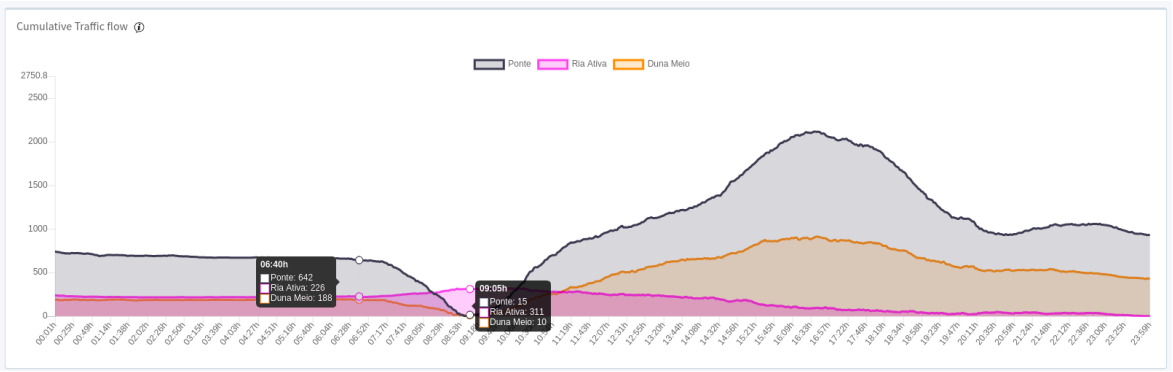


Figure 4.12: Chart that displays the cumulative traffic flow on the three different radars.

Much more information can be extracted from this chart, such as patterns of traffic, traffic variation differences between radars, changes in traffic on weekends and holidays, and of course, radar failure.

The second component on the page, a bar chart, groups the information by a certain length of time, depending on the interval selected, and displays the exact number of vehicles that enter or exit a region/radar during that time. For example, if the user selects only one day, the chart will group the information by the hour, while if two days are selected, it will group it by two hours. Both filters affect the information displayed on this graph. When viewing region data, the chart will build two bars up and two down for each group. The bars up describe vehicles entering while the bars down represent the ones leaving. The colors of the former are the ones seen in the line chart, purple for Barra, light blue for Costa Nova, black for Ponte, yellow for Duna Meio, and pink for Ria Ativa. The colors of the bars that represent the vehicles leaving have all different tones of red. While values from the first chart, by themselves, didn't show useful information, the ones from the bar graph say much more individually. Here the user can see the exact number of vehicles that circulated in a specific hour in a day, for example. The information from this graph can also be merged with data from the line chart to have a better understanding of the traffic state.

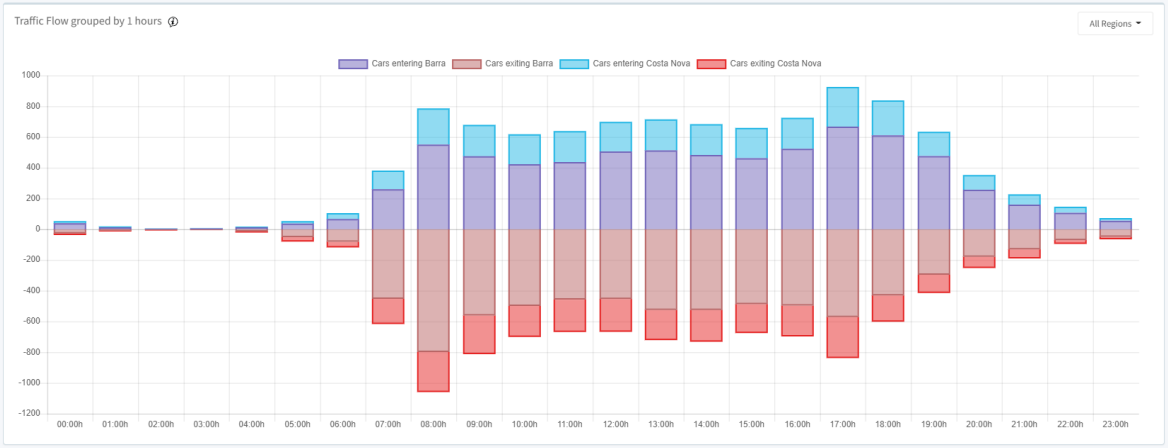


Figure 4.13: Bar chart that describes the exact number of vehicles entering and leaving a region/radar.

The next three graphs(one line chart and two pie charts) are separated from the top two by a small grey line. This separation signifies that the bottom charts are not affected by the "Filter by" button, only by the "Select Date" element. The line chart displays the average speed of traffic in the time frame selected. The default values show information about the three radars, utilizing the same colors as the other two charts. However, if users want information about the traffic speed on a specific radar, they can select the button found on the top right of the element. The dropdown displays the name of the three devices that, when selected, will change the data displayed in the chart, showing the average speed of traffic entering or exiting the chosen radar.

This component provides a way to check times when there was no traffic or relate vehicle speeds to other variables. Finally, located on the right of the speed graph, two more charts were implemented. The one on the top shows the percentage of traffic classes that were identified in the time frame selected, while the chart below shows the percentage of vehicles speeding. These pie charts work with data from all three radars.

Figure 4.14: Chart depicting the average speed of traffic. The left graph appears when "All Radars" option is selected in the filter, while the right one is built when the user selects a certain radar, in this case "Duna Meio".

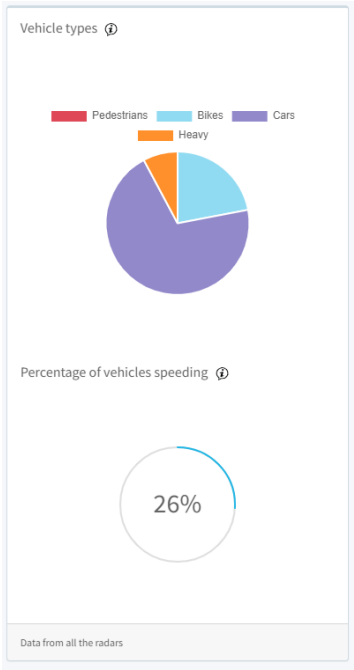


Figure 4.15: Pie charts positioned to the right of the traffic speed graph.

The final tab on the "Dashboard" category is called "Compare Dates," and it helps users contrast values from two different time frames. It is divided into zones, left and right, each one with two filters and two charts. These are the same filters and the first two charts from the previous page. The filters work the same way, but the right one only affects the right charts and vice-versa. With this format, it is possible to compare not only time frames but also different types of data for the same period, i.e., contrast the behavior of the regions to the radars. For example, the user can know which radar had more traffic when a rapid flow of vehicles entered a region.

The next element of the sidebar, "Device," shows a map with all the devices installed in the PASMO infrastructure and how the architecture is built. This page doesn't use data from the API.

The final page in the "Main Navigation" section provides the user with a map showing

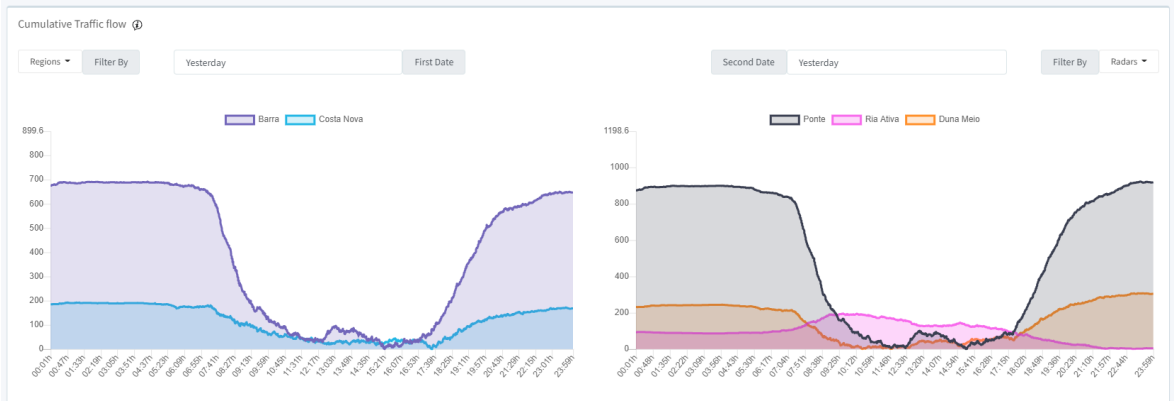


Figure 4.16: Traffic flow charts found in "Compare Dates". This Figure depicts the possibility of comparing the radars and regions values, to understand how the first influences the second.

real-time information about the regions and their entrances. When the page is loaded, two colored areas are shown on the map depicting Barra(the one on top) and Costa Nova(the one below). The color of these two areas will change depending on a percentage. This value represents the current vehicles in the region compared to the maximum registered. This percentage varies both the hue and lightness of the colors, but the former is what affects the change most significantly. As the percentage increases, the colors vary between green, yellow, orange, and red, i.e., as the percentage rises, the hue value decreases. Finally, the areas can also appear black when both radars from that region are not communicating with the platform.

Figure 4.17: Map depicting Barra and Costa Nova, with colored areas surrounding them. In this example, both areas have a green color which means that there aren't many vehicles in both regions.

A white line surrounds these areas, turning grey when the user hovers them. This process causes a box to change state on the left of the map, going from presenting an information message to displaying current data about the region hovered. This element starts by giving

the estimated number of vehicles in the area, along with the percentage discussed above. The number shown and the box where this value is located have the same color as the area selected. Below this information, data about two entrances for the region appears. Traffic can enter Barra through Costa Nova (Duna Meio radar) or the bridge (Ponte radar). Costa Nova also has two entrances, one on the north (Duna Meio radar) and the other on the south (Ria Ativa radar). The best data that is needed to tell traffic status is the average speed. This value is shown below the description of the type of information, "Average Speed," accompanied by some arrows to represent the direction of traffic. The green arrow represents the speed of the vehicles entering the region, while the red arrow describes the opposite. Below this data, some unquantifiable information is shown, describing the status of the congestion. The phrases, high congestion, medium congestion, low congestion, and no congestion, vary depending on the waiting time. The first one appears when the waiting time is higher than 18 seconds, the second when it is between 9 and 18, the third when it is higher than 0 and lower than 9, and the last when there is no waiting time. Regarding the waiting time value, it is calculated by taking into account the speed limit of the road and the detection range of the radars. For example, for the bridge entrance, the limit was set to 80 km/h, which means that every vehicle traveling at or above this speed will have a waiting time of zero seconds. For traffic traveling below this value, we need to calculate some traveling times by utilizing the speed formula. The distance considered for this calculation was 300 meters, which is the maximum detection range of the radar. First, we calculate the time it takes for a vehicle traveling at the limit speed to cover the distance. Then, the time it takes for traffic traveling at a speed below the limit to cover the same space. The difference between these values will give us the time that a vehicle traveling below the speed limit will have to wait compared to traffic traveling at or above that speed.

$$\begin{aligned} speed\_limit &= \frac{300}{time\_limit} \\ speed\_avg &= \frac{300}{time\_avg} \\ waiting\_time &= time\_limit - time\_avg \end{aligned} \tag{4.5}$$

Essentially, the slower the traffic, the more time a driver has to wait. This value accompanies the informative phrases discussed above to provide the user with quantifiable information. This page is more directed to regular visitors or residents, offering a way to check the current traffic state on every lane of every entrance for the two regions. Figure 4.18 shows the box of data that was discussed in this paragraph. The location of the radars is also present on the map, being used for faster navigation where users can click on icons or select the entrance text found in the pop up to zoom into that area.

The documentation section of the application helps the users to understand the system's services, both the API and the dashboard. The first tab, "API Docs," opens a new window with the documentation page generated by Swagger UI. Here all requests are specified, along with their parameters and types of data, offering the user the possibility of testing requests

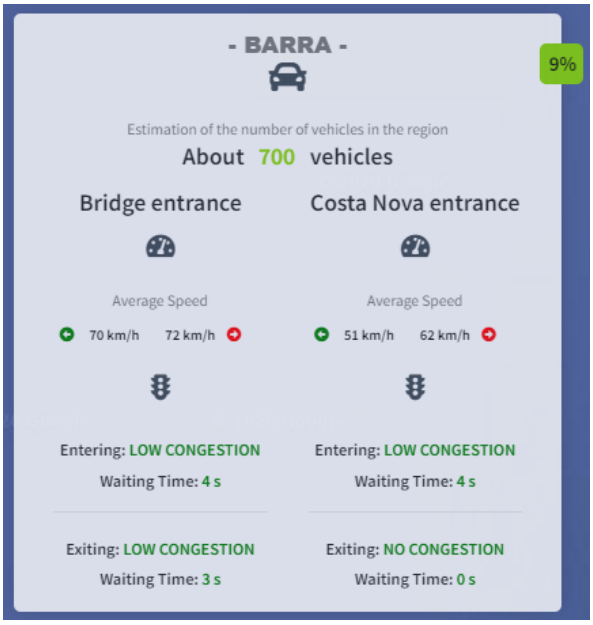


Figure 4.18: Box that pops-up when the user hovers an area. Its purpose is to provide a real-time picture of traffic in the entrances of the two regions.

directly on the page. The following tab, "App Docs," provides users with detailed information about how the frontend application works as well as what the different charts and maps show. The text descriptions are accompanied by images and graphs to provide a visual component to the explanation.

4.2 Vehicle Classification

This section details the process of training a YOLOv4 model to perform traffic classification as well as the methods used to merge this information with the radar data.

4.2.1 System Structure

The following system needs to use an object detection algorithm to classify traffic using video streams from the cameras, being made available to the detector by the use of the TLS1.3 protocol. YOLO has grown through its various updates to achieve real-time performance and have close to or even better accuracy than other algorithms. YOLOv4 is their newest version that provides better accuracy and runtime speed. Each version released offers different configurations of models that vary in accuracy and runtime speed, influenced by the change in the number of convolutional layers or their size. Earlier, we discussed the possibility of using a desktop or a board to run the object detector. Since the board was considered a better choice for the context of this dissertation, we have to train a smaller model of YOLOv4, YOLOv4 tiny. NVIDIA offers many solutions with JETSON products. The one that will be used in this dissertation is the Jetson Nano, which provides 128 Cuda-cores with Maxwell architecture-based GPU. This board does not include an operating system by default, but NVIDIA provides SDKs with some essential software already installed. These are the JetPack SDKs, which include a Linux operating system, CUDA accelerated libraries and other SDKs

that up the performance. The JetPack installed in Jetson Nano was version 4.4, which holds Ubuntu 18, CUDA 10.2, CUDNN 8.0.0, and TensorRT 7.1.3. A deep neural network library, tkDNN, will utilize the last two to extract the best performance out of the board. The results from its detection need to be available to the algorithm that will merge the data afterward, the Fuser. The best solution to establish this communication is to use sockets as a way for YOLOv4 to provide the detection values to the Fuser. This program is implemented in Python 3.7 and has the responsibility to create more reliable data merging the detection and radar values. The latter will be retrieved from SCoT by the same means and using the same transport protocols as the API. Finally, the web app will communicate with the Fuser so that this can provide all the data to be displayed in the client web browser. The framework Django was used to build this application since it is a fast and reliable way to build a website. The web application sets a connection with the Fuser, via WebSockets, to receive the data already merged. Another WebSocket is established between the app and the browser to display the information to the client. We utilize a Django feature called Channels, based in WebSockets, to build all the communications described.

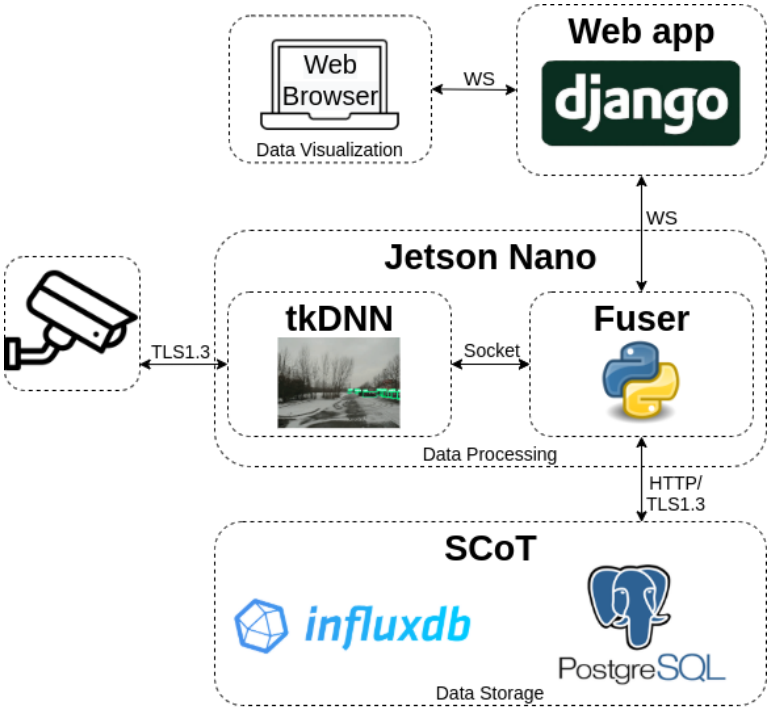


Figure 4.19: Structure of the vehicle classification system.

4.2.2 Object Detection

To achieve good accuracy on an object detection algorithm, we need to follow a particular pipeline, which begins with configuring the model and selecting proper labels for the images. This process starts by extracting frames from CCTV videos in varied weather and light conditions, to then begin the slow process of labeling the images extracted. Before training, some files need to be configured so that we end up with better detection and a smooth training process. The latter will create several weight files that must be tested to establish what is the

one with the best accuracy. The model can then be implemented on the Jetson Nano board utilizing libraries that boost the performance of detection.

4.2.2.1 Extracting and Labeling CCTV Images

YOLOv4 tiny trained in the COCO dataset already offers good accuracy as is, being able to be implemented on the board and deliver decent accuracy. However, we can obtain a significant increase in accuracy by training weight files using custom images from the cameras. This boost in precision happens because these devices are static, recording video with the same background every time. With this knowledge, the model can very easily learn to ignore the surrounding scene, focusing on the regions that matter the most.

The custom images can be extracted from the video content provided by the CCTV cameras. These are deployed in the same location as the radars and provide a stream to the IT building. It's from this live video that frames can be extracted and labeled to perform the training process. Using a Python library aimed to solve computer vision problems, OpenCV, a script was created to save images every 13 frames. These were stored in three separate folders, one for each camera, and then studied. Typically, there is a fair amount of time where no traffic is circulating in those areas. Combining this with the fact that the script is extracting around two images per second, a ton of frames did not provide much interest since no vehicles were visible. In the end, most of the images in which no traffic appears were deleted, leaving only those of importance. A couple of dozens were still used so that the network can learn where the background is in the image and ignore it.

Going by the tips on "How to improve object detection" [67] on the official repository of the framework used for training, we see a recommendation of 2000 or more images per class. Following this advice, 8000 were extracted for each camera and labeled. Several types of weather and light conditions were considered, labeling about 2000 images at night time and 1000 on a rainy day, with the rest representing normal conditions. To make the labeling process a lit bit easier, a Python application [68] was used where the user can draw a box around the object in an image and get a file with the labels well-formatted. YOLO only accepts a specific syntax of object labels, structure as follows:

```
<object-ID> <center-position-in-x> <center-position-in-y>
<width-of-the-box> <height-of-the-box>
```

All these values, except the object-ID, are not expressed in pixels but rather, saved in a percent relative to the image size, i.e., if the center position of the box in x and y is 0.5, it means that it's located in the center of the image. In the end, after the labeling process is complete, three folders, one per camera, hold all the files generated along with their corresponding images.

4.2.2.2 Configurations and Training

Over the years since the first version of YOLO was available, many libraries were developed to run the detector in different types of scenarios. However, the majority of these didn't improve the training process, with many of them ignoring it completely. Thus, the original

software created to train and run YOLO algorithms, Darknet, is the go-to when it comes to the training of models using custom images. Darknet is an open-source neural network framework written in C and CUDA, created by the same researchers that developed the YOLO algorithm.

To train a YOLOv4 model to detect custom objects, some configurations need to be made before running Darknet. Firstly, we need to move all the images and label files to the *data/obj/* folder. After this, a simple Python script is used o create a file(train.txt) in the directory *data/*, containing the path to all the images saved previously. Another two files must be created in the same folder, one describing the name of the four classes(pedestrian, bike, car, and truck), and the other holding five values defining the number of classes and paths to relevant folders. The information saved inside this last file is the following:

```
classes = 4
train  = data/train.txt
valid  = data/test.txt
names = data/obj.names
backup = backup/
```

This configuration means that we will train four classes, defined in *data/obj.names*, by using the custom images described in *data/train.txt* and the weights saved in the folder *backup/*. The valid configuration is later used for testing purposes. The final file we need to configure is the model itself. Darknet offers various cfg files with different YOLO versions, being yolov4-tiny.cfg the one we want. Here some lines need to be modified to make it right for our detection. First, changing the *max_batches* to 8000, which is the number of classes times 2000. This value defines the number of iterations that will be executed in training. The *steps* value will be 80% and 90% of *max_batches*, i.e., 6400 and 7200. These numbers represent the iterations where the learning rate will be adjusted to a lower value, increasing training time. The *classes* value, in all three YOLO layers, is set to 4, and the *filters* line is set to $(classes + 5) * 3$ which is 27, defining the number o CNN kernels there are in a convolutional layer. These kernels are matrices that move over the input images to extract features from them. The last value configured, *anchors*, represents the initial size of the bounding boxes. We can discover the best values for this variable by running the command *./darknet detector calc_anchors*, which will go through the training dataset and estimate the best initial size of the bounding boxes. It is crucial to perform this step since its easier for YOLO to learn small adjustments rather than large ones.

With all the configurations done, we simply need to download the pre-trained weights file to run Darknet. As stated before, the program will run 8000 iterations, saving the weights files every 1000. This process is essential to study the different iterations and choose the one with the best performance. More iterations don't always mean better accuracy since the model can be overfitting, meaning that it performs well on a training dataset but not on a testing one.

4.2.2.3 Running the Model

After the training process, we end up with three files, one for each camera. The next step is to configure the Jetson Nano board so that it can use these files to detect traffic in real-time.

Even though Darknet is optimized for the YOLO algorithm, Nvidia provides software that significantly boosts the performance of its hardware, like cuDNN and TensorRT. Developers can utilize these frameworks to build software on top of them, withdrawing the maximum performance possible from Nvidia GPUs. This was the concept behind tkDNN [69], build a library using cuDNN and TensorRT primitives to achieve excellent performance on NVIDIA GPUs, especially on Jetson Boards. Considering that the SD image for the Jetson Nano already provides an Ubuntu system with all the frameworks installed, we only need to build the project to be able to run it.

This library comes with weights pre-trained in the COCO dataset. However, these are of no interest since we will use the ones previously trained. Darknet comes into the picture one more time to perform a conversion of these files. This procedure needs to occur because tkDNN does not support the type of weights defined in Darknet. The command *./darknet export* is executed, creating a folder named layers, which holds a file for each YOLO layer with the corresponding weights and bias.

After setting up all the files, the library can be compiled to a folder named build, that will hold the executable files as well as the layers folder discussed above. Although, before running the software, it is crucial to define the best values for batch size and inference mode. The former can be configured with the values 4 and 1, while the latter can be set to FP32, FP16, or INT8. However, only the first two interferences can be used by Jetson Nano since it has a Maxwell GPU, which doesn't support INT8 in hardware. These values give more accuracy in favor of speed, so it is crucial to balance these two. Tests were performed in 20 minutes of footage on the dunaMeio RSU to check the best candidate for this scenario. Figure 4.20 shows the results of FPS and mAP of all the combinations of the values above.

Looking at the graph, we can conclude that interference F16 with a batch size of 4 is the right approach when it comes to balancing accuracy and performance. F32 provides better accuracy, but it's within the limit of what it considered real-time speeds. In this case, it's better to choose the interference with a higher FPS because the accuracy cost is not that significant. The library is now able to run with the best performance possible, giving us the result shown in Figure 4.21.

Even though we maximize performance in the interference and batch size category, sometimes the FPS's drop to low values, which can cause a significant problem. Since tkDNN is using OpenCV to read a stream of video, it will try to keep up with it if the software is lacking behind. This correction makes the program jump to the recent time on stream, meaning that if the software is, for example, two seconds behind the stream, it will skip these two seconds to try to keep up with it. This jump will only occur if the detector is running on a lower FPS than the stream. Initially, the stream was being sent at 30FPS, which was too high for the program to handle. Thus, we gradually reduced the stream's frames per second until ending up with a value where tkDNN could keep up with the stream. This value came

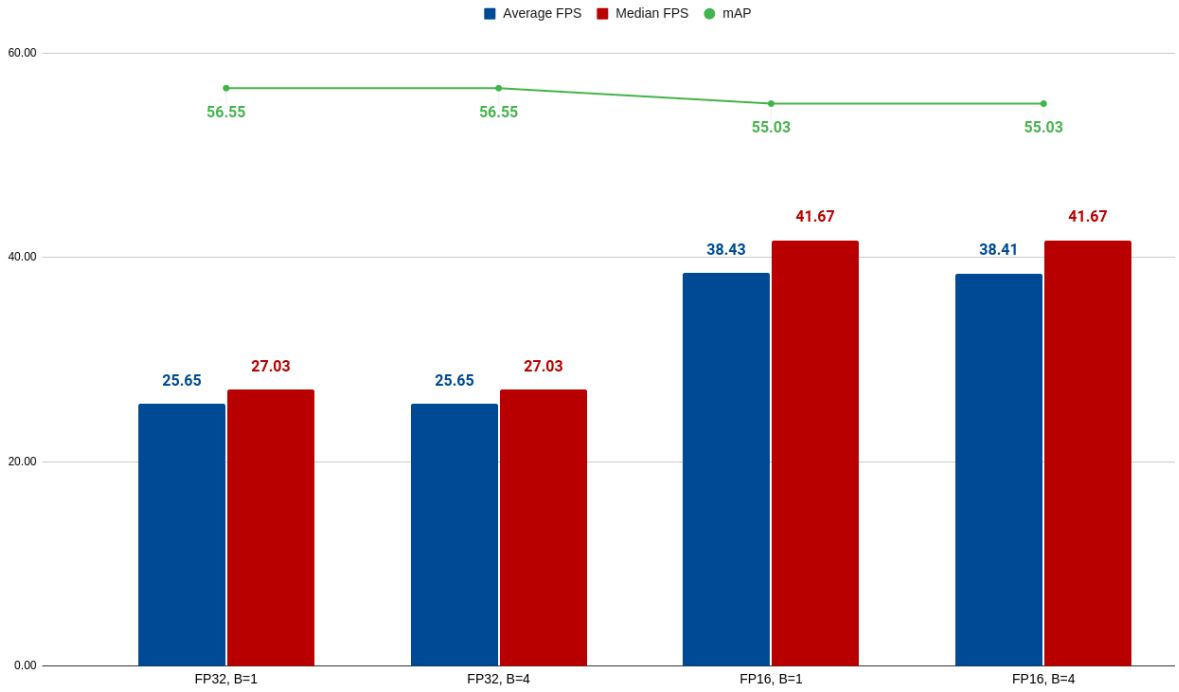


Figure 4.20: Average and median FPSs accompanied by mAP of the different possible values for batch size and inference.

Figure 4.21: tkDNN running on a stream from DunaMeio camera.

to be 20FPS, and here the detector runs at a faster pace than the stream, meaning that the program sometimes needs to wait for it. However, this is not a problem since no detection values are lost, unlike in the initial scenario.

With all these configuration, we assure that the tkDNN is at maximum performance, not skipping frames, and thus, not losing potential detection values.

4.2.3 Sensor Fusion

When running YOLO on Jetson Nano, the algorithm provides information about the bounding boxes as well as class identification and its probability. Although these values tell us a lot about the object detected in the video, it doesn't provide much information about the state of the real world. Thus, this data needs to be converted to something useful, like geolocation of the vehicle detected. Only after this process, we have the necessary information to merge the two types of data, radar and camera information.

4.2.3.1 Calculating Vehicles Location

Firstly discussing the values given by the tkDNN, the specific variables are class ID, probability of being that class, box center in x, box center in y, box width, box height. The most important values to compute the geolocation are the last four, with the rest being useful in some scenarios. We can perform the conversion using different approaches, such as trigonometry or machine learning techniques. The latter was chosen since it is a more straightforward way to achieve the required results. Thus, we need a method that can produce two outputs by using several initial values. This can be achieved using multi-output regression, i.e., problems that involve predicting two or more values given an input. The process is to check where the bounding box stands on the image and save the closest geolocation possible, as seen in Figure 4.22. The end goal is to build a CSV file with all this information. This process is made "by hand," thus it has some error associated with it. Nonetheless, it achieves the required goal.

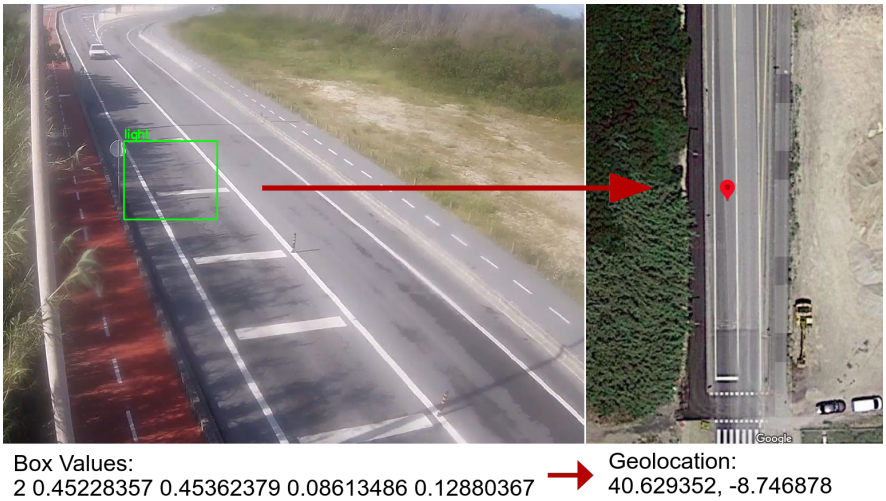


Figure 4.22: Example of translation from bounding box values to latitude and longitude.

After this, a model is trained using the values in the CSV file, utilizing scikit-learn, a machine learning library for Python. Since the vehicle's trajectories are always the same, i.e., traffic can't appear in the middle of the road, the idea was to save ten of each trajectory(roadway and sidewalk). This amount of data should be sufficient for the model to learn to compute accurate values. Of course, we need to consider trajectories with different classes of vehicles since bounding boxes for a heavy differs massively from the ones for a bike. Moreover, a

pedestrian is probably will never be detected in the middle of the street. These are just some situations where the model can rely on the class of the object to provide useful information.

By implementing this model on the Fuser, the program will be able to receive bounding box values from tkDNN through a socket and estimate the latitude and longitude of the vehicle detected.

4.2.3.2 Obtaining Radar Data

As detailed previously, SCoT will provide the radar data. The Fuser will get this information by the same means as the API, but only using the psycopg2 Python library. Since we need all the details about the object detected, PostgreSQL is used to fetch the required information. The data is retrieved in chunks of one second to make it easier to compare. A problem arises when looking at radar timestamps on the database and comparing them to the stream is that these two are not at all synchronized, with the stream being way behind in time. Thus, instead of radar information of the last second, we retrieve the data 11 seconds prior. This shift accounts for the difference between the device and the stream. Nonetheless, this process probably wouldn't be necessary if the camera and the radar directly feed data to the board.

In Section 3.1, we lay down the structure of the data that the radar produced. It was established that the device didn't provide the latitude and longitude of the object detected but instead, it gave information about the distance to the radar(in meters) on an axis(x and y). In this 2D space, the radar is positioned at the point (0,0). By using these values alongside the azimuth also given in the data produced by the device, we can calculate the geolocation of the object detected.

Diving now in the field of navigation, the azimuth is the angle measured clockwise from the north baseline. For example, if the azimuth is 45º, it means that the device is pointing north-west, and 180º indicates that it is pointing south. With this value, we can understand where the radar is oriented. Figure 4.23 provides an example using the azimuth from the bridge radar, which is 249 degrees.

To calculate the latitude and longitude using the distances and azimuth, we have to follow some formulas. The constants 110540 and 111320 are used to account for the earth's oblateness (polar and equatorial circumferences are different).

$$
\begin{aligned}
radar\_azm\_rad &= radar\_azm \times (\frac{\pi}{180}) \\
r &= \sqrt{x^2 + y^2} \\
angle &= \text{atan2}(y, x) \\
dx &= r \times \sin(radar\_azm\_rad + angle) \\
dy &= r \times \cos(radar\_azm\_rad + angle) \\
delta\_longitude &= \frac{dx}{111320 \times \cos(radar\_lat)} \\
delta\_latitude &= \frac{dy}{110540}
\end{aligned}
\tag{4.6}
$$

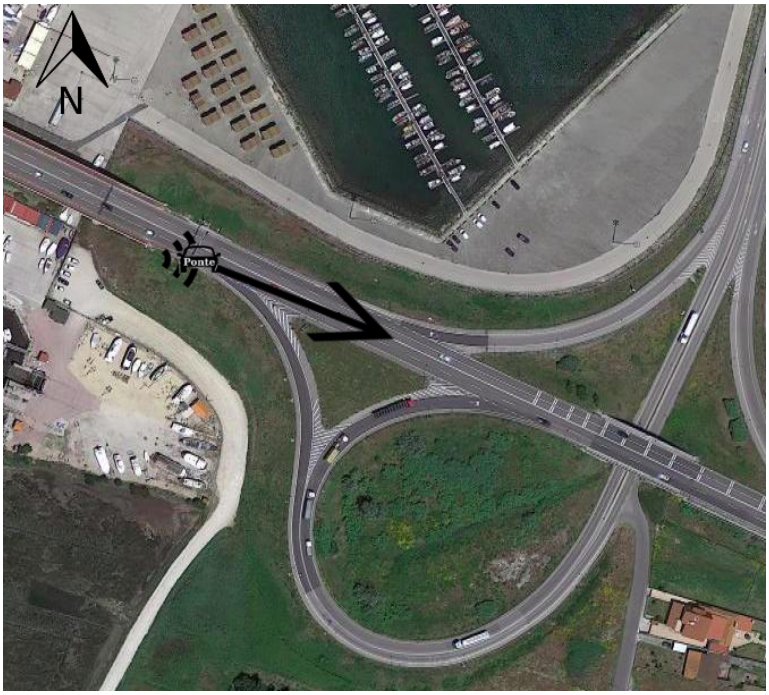
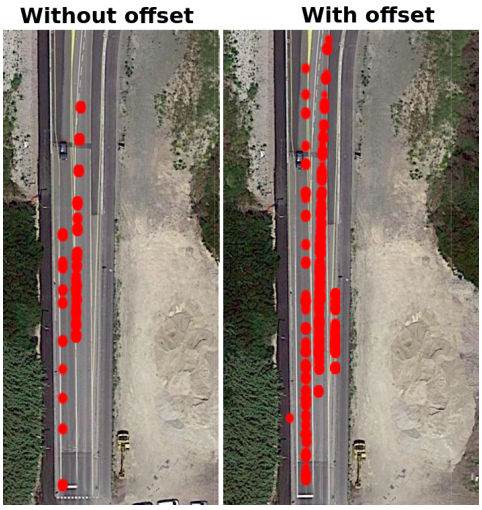


Figure 4.23: Azimuth of the radar, which indicates to where it is pointing relative to the North.



(a) Adjustments in dunaMeio radar data.

(b) Adjustments in ponte radar data.

Figure 4.24: Difference in location values when applied the offset. Red dots represent detected objects by the radar. The dots with the offset are more inline not only with the road but with the sidewalk as well.

Some example values were displayed on a map to check if the calculations were precise. The results showed that the farthest way the object was to the baseline direction that the device was pointing to, the higher error in detection. Even so, this problem can be easily solved with an offset value that shifts the geolocation by a certain amount. Some radars are easier to fix than others, mainly because of the way the road is aligned to the north baseline (RiaAtiva and DunaMeio). Figure 4.24 shows the difference before and after applying this value to the DunaMeio radar.

The corrections that help to have a more accurate location were adding 0.000018 to the longitude of dunaMeio, -0.00002 to the latitude and -0.00005 for the longitude of ponte, and 0.000011 to the longitude of riaAtiva. With these offsets applied to the latitude and longitude, we end up with precise geolocations for all the radar's reads.

4.2.3.3 Fusion

With all the things set, it was time to combine the data. One thing to remember, which improves performance, is that the GPU handles only image processing while the CPU performs the calculation and exchange of data. Another thing to note is that the detection produces a large number of values in one second while the radar gives three maximum. This amount of data can be an advantage since the average of more values is regularly more accurate than a single one. Considering that the Fuser will receive data from two different sources, it is essential to separate these two. The best solution is two create two threads, one to receive the detection values, convert them to coordinates, and save them in a list and another to retrieve radar data, calculate the geolocation, fuse it, and send it to the web application. This list holds the latitude and longitude of the vehicles detected by the camera and will be shared between threads. We will call the one that handles the radar calculations *threadRad*, and the one responsible for the detection data *threadDet*.

Regarding the first one, it's necessary to implement a way in which the detection library and the sensor fusion program can communicate. This connection can be achieved using sockets, with tkDNN posing as a server and the Fuser being its client. With the communication established, the server can send the bounding box values produced by the detection process to the client. When receiving this data, the Fuser uses the trained model to estimate latitude and longitude values of the object detected by the tkDNN. After this, the coordinates are stored in the shared list. A small condition is checked so that we can minimize the size of this list. The program will compare the last saved coordinates with the ones recently calculated. If these are close together, it means that they refer to the same vehicle so, we can store their average instead of the two. This thread will continuously be updating the list with coordinates calculated from the bounding boxes.

The second thread will retrieve the data from the radars with a delay, for reasons already stated. It also calculates the latitude and longitude of the traffic detected by the radar and applies the offset explained previously. At this point, we have both the location of the objects detected by the camera and the radar. The program will loop through all the coordinates calculated from the radar data and compare it to the coordinates found in the shared list. The difference between these two positions gives us a value that we can use to check if the data depicts the same object. If this value is in a defined threshold, it means that both devices detected the same vehicle, and the classification from tkDNN is used. If the difference doesn't fall within this limit, the data provided by the radar is used to predict the vehicle's class. The threshold value depends on the radar and will be discussed after this paragraph. Still, when both devices detect the same object, we will end up with detection from the radar and classification from the cameras, taking advantage of the best of both worlds. However, there

are times where this condition doesn't apply, leaving us with no correlation between the two devices. In these situations, only one device can be used to achieve classification and detection. The reason that we choose the radar to cover these circumstances is that the detection that it provides should be more trusted. If the cameras give a location that is not close to the one the radar detected, it could mean an error in the prediction of coordinates or a fault on the detection algorithm side. To achieve reasonable classification accuracy using the radar's values, one must understand traffic behavior in different locations. Even though the length of the vehicle given by the device is one of the most important to classify traffic, we cannot use it alone. For example, even if the length value says otherwise if the coordinates show that the object is on a sidewalk, it must be a pedestrian or a bike. We utilize the same strategy used in the translation of bounding box values to geolocations. The idea was to get data from the radar and check the stream to see to what class of vehicle it corresponded to. A model was then trained with a CSV file holding several correlations of radar values to vehicle class, in the same library as before, scikit-learn. In the end, the model was implemented into the Fuser, helping to perform a better estimation of the type of vehicle detected when the cameras can't be used. The past method of prediction only used the length in the calculation, causing errors regularly when the object was far away from the baseline that the radar was pointing to. This approach improves the previous one by using all the object values in the prediction.

In the last paragraph, it was stated that a threshold helps us identify if the data from the radars and the cameras are related. The idea was to define a rectangle where its width would correspond to a lane, and its length would correspond to two light vehicles. If the location calculated from the radar and the one computed from the detection are both in this space, it means that they are detecting the same object.

4.2.3.4 Web Application

The fusion process is complete after using the methods described above. However, to provide better visualization of the results, the Fuser can send the refined data to a web application to be displayed on a map. This exchange is achieved by making use of WebSockets. The web app, created using Django, poses as a server while the Fuser, the client, connects to it. After this, the Javascript component of the application also connects to the server so it can receive the data. It will receive the latitude, longitude, and class, being able to draw icons on the map on the location given that represent the class of the vehicle. These are deleted after one second, so the map only shows recent data.

The user can select which radar to view by using a dropdown on the top of the page. This option will affect not only the map but the stream displayed on the left of the page.

CHAPTER 5

Results

Given the context of the services and application created, various tests were conducted to examine the performance and usability of the applications. The methods used differ depending on the service and their outcomes will be detailed, explaining if they translate to good performance.

5.1 API Performance

Regarding the API, it is important to perform tests on its speed of response and understand the size of the data that is being given. When users access this service to retrieve data from it, they will expect the data to be delivered fast and organized. Even if the API has some problem or there is no data to be retrieved, users expect a warning regarding these issues. Since this service is provided to the public, it is essential to understand if the design and structure of the service meet the user's standards.

5.1.1 User Evaluation

An evaluation was conducted on users with good technical knowledge of data analysis. The participants tested all the requests available, looking at speed, data structures, and error management. Five people composed the group of evaluators, all of them studying Computing Engineering, with the majority claiming to have a good understanding of API configurations. The goal of this test was to conclude if the structure of the requests created, and their data were understood by the average user.

In the evaluation, the participants were provided one to three of each request, giving it a total of 32. All of them were retrieving data from 30 days except the ones that have a maximum limit of one day. For these, the time frames used were one day, one hour, and 10 minutes. The users could also visit the documentation page to have a better understanding of the structure of the data. After analyzing each one, a questionary was filled to provide quantifiable data on various variables. The questions were the following:

- Did you understand what type of data the request gives? The answers range from 1 to 10, with 10 being understood perfectly
- Was the data well structured? The answers range from 1 to 10, with 10 being very well structured
- Is the URL configuration user friendly? Range from 1 to 10, with 10 being very user friendly
- How fast was the request? The answers range from 1 to 10, with 10 being very fast

Lastly, the user was asked to type five requests that only returned error messages. The goal of this last process was to have feedback on the effectiveness of error handling. The final questions were regarding how the service handles errors, the documentation page, and how experience s the participant when it comes to API specifications.

The results of the questionary were the following:

	Was the data well structured?		Is the URL configuration user friendly?		How fast was the request?	
	Average	Median	Average	Median	Average	Median
/parking	9.8	10	6.4	7	5.6	5
/parking/{sensorID}	9.8	10	6.4	7	8.2	8
/parking/availableSensors	7	7	9.4	9	8.6	9
/parking/events	8.4	8	9.8	10	8.2	8
/parking/latestValues	9.8	10	9.8	10	9.4	10
/radars	4.8	5	9	9	8.8	9
/radars/location	6.6	6	9.6	10	9	9
/radars/vehicleEstimation	8.8	9	9.6	10	8.4	9
/radars/{radar_id}	6.6	7	6.4	7	6.6	7
/radars/{radar_id}/{measurement}	9.2	9	7	7	6.6	6
/radars/events/{region}/{event}	9.4	10	6.6	7	7.2	7
	Rate documentation page		Rate error handling		Understanding about API configurations	
	4.6	5	7.8	8	5.8	6

Table 5.1: Results of API usability tests.

As we could expect, the more abstract requests were reported as the slowest, with the one that provides raw radar data the most confused in terms of data structure. The participants also reported that the data could be more compact, for example, having only speed instead of speed in the X and Y-axis. Nevertheless, after explaining the purpose of the request most of them comprehended the choice. However, some insisted that variables, such as sensor location, didn't need to be separated into latitude and longitude. The error handling and the other requests had positive scores, but many stated that the documentation page should be more detailed. In the end, the users understood the data being given and its purpose, not criticizing the waiting time too much, getting a good experience in the majority of the requests.

5.1.2 Speed and data size

As stated in the previous chapter, load tests were conducted to the uWSGI server to understand how the two variables affect its performance. Even though developers advise the processes

from the server to be the same as the CPU processes, we started with half that amount. This configuration was P8T8, meaning eight processes and eight threads. To perform these tests, we run 1000 requests, with 25 of them being made at the same time. The one used was radars/events with a time frame of 7 days because we consider this to be one of the most meaningful for users. The best practice is to start with the processes equal to the CPU cores, but we will test lower values to see the influence on the time. The results of the different combinations examined are displayed in Figure 5.1.

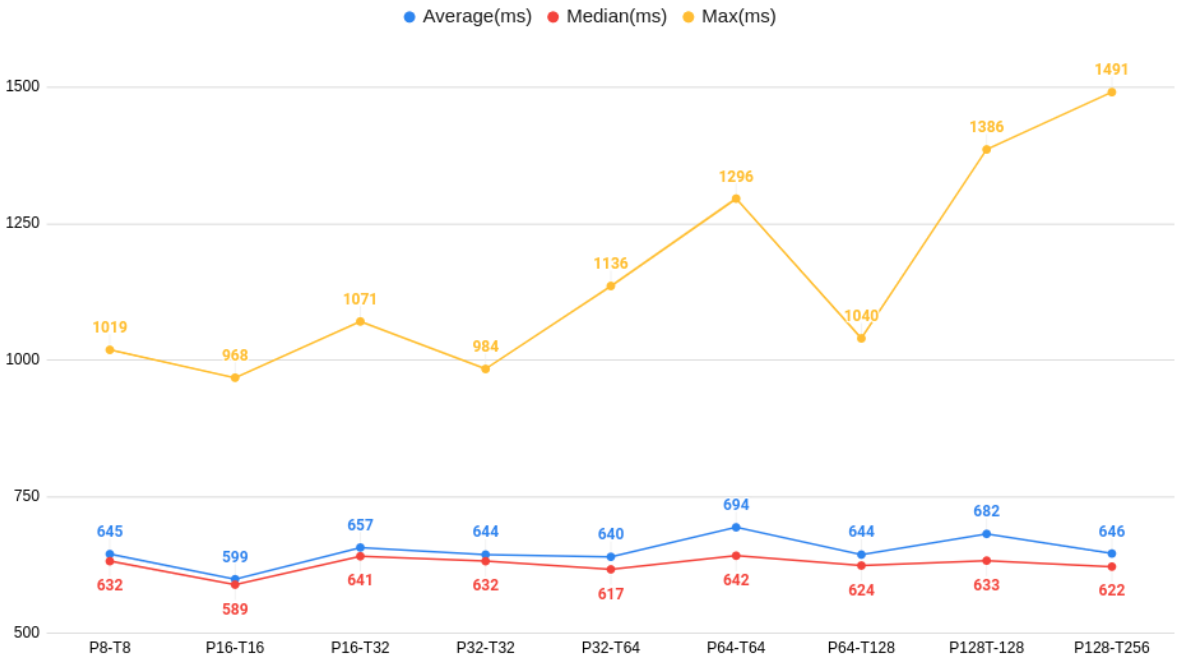


Figure 5.1: Results of the testing of several configurations test showing the average, median, and max time of 25 requests in milliseconds. These are expressed using a letter(P for processes and T for threads) followed by their value.

After analyzing the results, we can see that the configuration of 16 processes and 16 threads gives the best performance, being the fastest overall.

Even though an uWSGI server helps a lot with load balance, if the size of request response is too large, it can cost an application performance when the network has low bandwidth. Knowing this, the data size of each request is almost as important as response speed since these two go hand in hand. The goal was to have the response time below 500 milliseconds and the data size under 200KB. We can expect this for most of the requests except the larger ones, like /parking, that give the raw data of the sensors. The tests were conducted in the same network where the API is implemented, providing the best-case scenario and in a household with a 100MB/s internet speed, providing an average scenario. We run each request ten times, setting the time frame to 30 days except for the ones that have a maximum limit of one day. Table 5.2 shows the results of these tests.

As expected the response times fell on the values stated, except the largest requests, being the ones that retrieve raw sensor data or have a wide time frame. Nevertheless, the user can use the *limit*, *offset*, and *groupby* parameters to further lower response times and data

Requests	Time Lab(ms)	Time Home(ms)	Size(KB)
/parking 10 minutes	38	91	34
/parking 1 hour	139	176	201
/parking 1 day	2320	2320	4278
/parking/{sensorID} 10 minutes	23	41	3
/parking/{sensorID} 1 hour	30	40	4
/parking/{sensorID} 1 day	35	85	17
/parking/availableSensors	27	69	36
/parking/events	563	591	3
/parking/latestValues	63	64	23
/radars	66	182	137
/radars/location	96	150	3
/radars/vehicleEstimation	459	642	3
/radars/{radar_id} 10 minutes	49	62	18
/radars/{radar_id} 1 hour	119	102	125
/radars/{radar_id} 1 day	846	869	1855
/radars/{radar_id}/{measurement} 30 days	1981	2293	65
/radars/{radar_id}/{measurement} 7 days	470	523	65
/radars/events/{region}/{event} 30 days	679	825	57
/radars/events/{region}/{event} 7 days	264	335	56

Table 5.2: Results of API performance tests.

size. When discussing the larger requests, other approaches can be utilized like constantly pooling data every 10 minutes. Although these provide large amounts of values, they fulfill their purpose of giving the user the freedom of creating new types of data with raw sensor information.

To be noted that these results are performed in favorable situations, where only one request is being handled. At times where the API is under heavy load, these times will increase. Some load tests were performed, in the household scenario, to understand the effect that a large number of requests has on the API. These were conducted by making 5000 connections to the service, by 100 users to the
event's request, using a time frame of seven days.

	Min	Mean	Median	Max
Connect	457	2170	2175	2865
Processing	94	244	236	1243
Waiting	94	244	235	1243
Total	706	2414	2423	3505

Table 5.3: Results of the connect, processing, waiting, and total times of the load tests in milliseconds.

We can see the change in response time when several clients are accessing the service. During the test, 2105000 bytes of data were transferred, with the whole process lasting 122 seconds. These numbers give us a transfer rate of 17.25 Kbytes/sec, which is somewhat low, but the times presented are not astronomically large since most users won't leave the website when the response lasts less than six seconds. The times are heavily influenced by the connection, which can be handle with more paralyzation of the service.

Similar to the API evaluation, two initial questions were presented to the participants regarding their experience with user interfaces and their knowledge of the acspasmo project. This questionary was made to separate the partakers into several groups. First, the tech and non-technologic people, and second, the ones with(regular users) and without(admins) knowledge of the project. This separation has the goal to provide us with opinions from very different groups of people that will use the application. The technologic users also referred to as power users, know the standards of web design, such as icons, buttons, and calendars, while the others are not very familiar with the UI components. The other group is composed of regular users who don't fully understand how the whole system and pipeline of software, while the admins have a clear grasp on the subject. The evaluations rely heavily on all of these aspects, like being comfortable with the technology and knowing PASMO infrastructure and how the different components interact with each other.

The two methods of evaluation utilized to conduct the tests were Cognitive Walkthrough and Heuristic. These provide the evaluator with enough information to have a good understanding of how the users view the application.

5.2.1 Cognitive Walkthrough

Cognitive Walkthrough is a usability inspection method, i.e., a method where a facilitator invites a participant to perform scripted tasks in a specific user interface. Thus, it's task-specific, meaning that the performance of the users executing that task will be evaluated to understand the problems with the interface.

The tests were performed with the participants accompanied by the developer of the dashboard. Since this method is highly influenced by the user's experience, we asked them to rate their technical understanding of websites and UIs as well as their knowledge regarding the PASMO project. The average age of the participants was 29, varying between 16 and 52 years old, with the majority being in their mid 20s. This large age range includes various user's experience, making it suitable to understand how it influences the results. All the participants in the mid 20s group were in the IT field, making them more experience. Even thought the understanding of web interfaces ranges a lot between participants, one thing in common is the knowledge of PASMO. Some stated that they heard about the project, but the majority had zero familiarity with it.

Before executing the tasks, the participants had two to three minutes to explore the application so that they could be more used to it. Initially, the page where the user should go to perform the task was described in the script. However, to understand if the sidebar accurately defines the content, we let the users discover on which page the task should be performed. Still, if they got stuck, a little help directing them to the page is given. The tasks for each page are the following:

- Parking Dashboard
  - Find a free parking spot and see its ID

  - See if there is a problem with the majority of the sensors
  - Find the average time that people spend on Barra
  - See the variation of an occupied parking spot
- Radar Dashboard
  - See the traffic variation in the regions between 03 and 10 of August of 2020
  - See the number of vehicles that entered the Ponte radar in 23/08/2020 at hour 10 and
  - Find the time where the highest traffic speed occurred, in that day
  - Find which radar was inactive in the past seven days
- Radar Dashboard or Compare Dates
  - At 02/11/2020, find which radar caused the curve to drop, in the morning
- Map
  - Find the entrance, in Costa Nova, with the highest congestion or waiting time, and in that entrance which lane has the fastest traffic

Regarding the tasks on the Parking page, most of the participants clear them with ease. The hardest one came to be the second one, where the users needed to relate the active and total sensors or look at the number of blue icons to arrive at a conclusion. Only 30% choose to follow the latter approach since the former is the first information that the users see when entering the page. Here 20% needed a little help to complete this task, such as pointing to where they should search. It is relevant to state that this users belong to the less experience participants and didn't quite understand the meaning of total and active sensors at first. Another relevant result occurred in the last task of this page, where 50% of the participants got a little bit lost, but none needed help. The users that got lost stated that they ignore the parts of the page with text, focusing more on the map. Nevertheless, these results were heavily influenced by the first minutes where the users could navigate the website freely. The ones that clicked on the icons to pop up the chart already knew how to perform the task. Beyond these separate cases, the participants found the tasks relatively simple, and the page very comfortable to navigate.

The Radar dashboard had more critics and not so good results compared to the previous. This outcome can be attributed to the administration style that the page offers, i.e., presenting many filters and tons of diverse data. Overall, 44% of the participants stated that they felt a bit lost, and 16% claiming they got lost a lot. The hardest task, with an average of 6.5 points of difficulty, came to be the second one since the users needed to select at least two filters and search for the hours and radar in the bar chart. Here only 30% of the participants went directly to the right chart, while 40% needed help to be guided to the bar chart. The last but one also had a high difficulty, 5.6 point, because the users needed to understand what it meant for a radar to be inactive. Another interesting result was the fact that 20% of the participants didn't utilize the fast option, provided in the calendar, to filter by the last seven days. Instead of that percentage selected, one week "by hand" in the custom date. When asked why the majority responded roughly the same, stating that they "had done it that way

before so (they) assumed that was the right approach." The major critics of this page were the lack of alerts when the radars were down. Participants asserted that some type of information should be displayed, warning the user that the data could be deceptive because of a radar crash.

Even though the ninth task can be performed on both pages, only one participant out of the ten completed it on the Compare Dates page, because he/she remembered the page when exploring the application in the beginning. It seems like most of the users work with the tools that they recognize, only leaving them if there is no other option.

The final task is performed on the Map page, where the participants need to find a decent amount of values. Most of the users, 90%, recognize that the red arrow signified the exiting lane, while some were slow to realize what the waiting time meant. One of the less experienced participants also got a little bit lost because, as stated, they "thought the values weren't separated because there was no line in the middle" of the entrances. Nonetheless, this page received the most positive feedback of them all, with comments like "this is more important than the charts if I wanted to go to the beach."

By studying the results of these tests, it's evident that the majority of the users found the application useful and packed with good information. The more experienced and knowledgeable users advise for more alerts not only in the Radar dashboard as stated before but also on the Parking page regarding the state of the majority of the sensors. They also suggested that a Parking dashboard with similar filters as the Radar's page could be of great use. The less experienced users also advise for more alerts and real-time features, like notifications of high congestion. Although the critics, the dashboard received positive feedback across the board.

5.2.2 Heuristic Evaluation

A more quantifiable type of evaluation is the Heuristic method. The most used and highest-regarded set of heuristic laws are the 10 Laws of Usability by Jakob Nielsen. The original heuristics are the ones used for the tests, which provide the most complete general principles for interaction design, been utilized for the last 20 years[70].

- Visibility of system status: The design should always keep users informed about what is going on, through appropriate feedback within a reasonable amount of time.
- Match between system and the real world: The design should speak the user's language. Use words, phrases, and concepts familiar to the user. Follow real-world conventions, making information appear in a natural and logical order.
- User control and freedom: "Emergency exit" so users don't have to go through an extended process.
- Consistency and standards: Users should not have to wonder whether different words, or actions mean the same thing.
- Error prevention: Eliminate error-prone conditions, or check for them and present users with a confirmation option before they commit to the action.
- Recognition rather than recall: Minimize the user's memory load by making elements, actions, and options visible.

- Flexibility and efficiency of Use: The use of shortcuts speeding up the interaction for the expert user.
- Aesthetic and minimalist design: Interfaces should not contain information which is irrelevant or rarely needed.
- Help users recognize, diagnose, and recover from errors: Error messages should be expressed in plain language detailing the problem, not in error codes.
- Help and Documentation: Provide documentation to help users understand how to complete their tasks.

In order for the users to understand these heuristics and evaluate the application the right way, some other functionalities were presented. These include choosing a date where there is no radar data available so that the evaluators can visualize the error messages and surfing the documentation page. The latter is also complemented by little bits of information scattered throughout each page, offering information about what data is represented in the chart or how to use certain features.

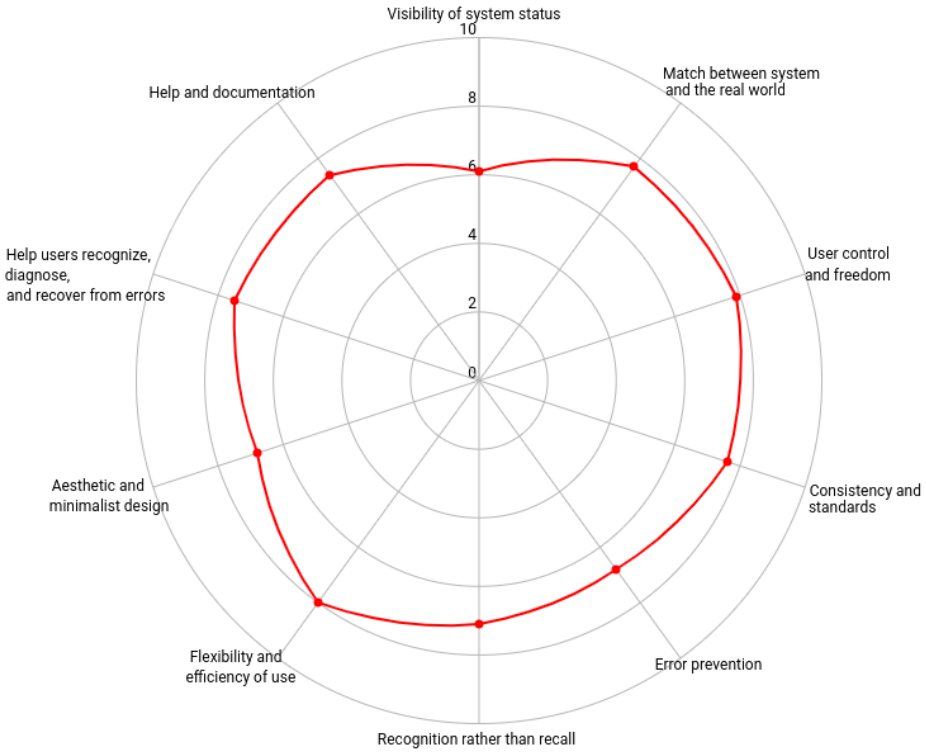


Figure 5.2: Average heuristic results for every principle.

The feedback of this evaluation is similar to the cognitive walkthrough, where the participants expressed the lack of alerts and information about the system state. Others stated that some information may not be relevant to the average user, such as the variation of a traffic sensor. The problems discussed fall in the heuristics "Visibility of system status" and "Aesthetic and minimalist design," which are the ones with the lowest scores. However, beyond these two, and the "Error Prevention" evaluation, which could be higher, the dashboard got a good overall score.

5.3 Training Results

As detailed in section 4, the training process lasted 8000 iterations, saving results every 1000. It was also stated that it's necessary to test all the weights and not only the last one because other iterations can provide better accuracy. Darknet can be used to make these kinds of tests, using labeled images to calculate the mean average precision(mAP). These images cannot be the same as the ones used in training because it could cause deceptive results. Therefore, 2000 more images were labeled to achieve accurate results on these tests. To check not only the mAP but also the AP of each class, we can run *./darknet detector map* on the 2000 images considered. Darknet receives them and calculates the mAP and other details regarding the weights introduced. Even though we know that the first iterations will not have good precision, all the weights were tested to understand the change in the mAP throughout the training process. The following figure show the results from these tests.

Figure 5.3: Changes in mAP throughout the iterations trained in different datasets, Ponte, DunaMeio, RiaAtiva, and COCO at IoU=0.50.

The results show great improvement in accuracy when compared to the weights trained in the COCO dataset. Even being the same model, we have a clear advantage when it comes to the environment recorded by the cameras. Since these devices are locked in place, the background of the stream is always the same, making it easier for the model to learn to ignore it. The images provided in the training process, where there is no traffic, also helped because it gave a picture of what the background should look like. Another thing that improved performance is the fact that we know where to expect bounding boxes, meaning that detection should only be found near the roads and sidewalks. The model already knew this fact before starting the training process because we pre-calculated boxes by looking at the dataset. So the adjustments made throughout the iterations were less drastic, providing better improvements. By computing custom anchor boxes before training instead of using the default ones, the

mAP by 1.72% in Ponte, 4.59% in DunaMeio, and 3.47% in RiaAtiva.

The tests also provide precision in each class, giving us a clear picture of what types of vehicles are limiting the mAP. By using the weights with the best accuracy, iteration 8000 for all the models, we run the same command stated in paragraphs above on the same test dataset. This process gave us the following results:

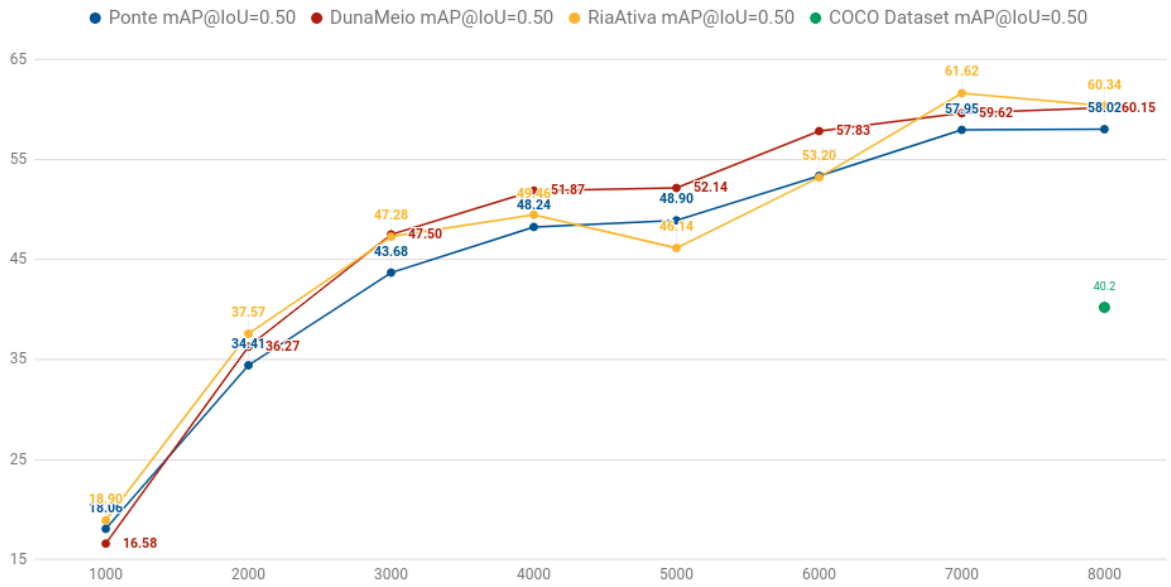


Figure 5.4: Results of precision per class in every model.

Looking at the chart, it's clear that the detection of pedestrians isn't satisfying, especially in Ponte. The bad results occur because of the positions of the sidewalks since it's the only place where pedestrians can be detected. In DunaMeio, much of the left sidewalk is covered by plants, and in Ponte, the right side barries hide the pedestrians and bikes circulating. RiaAtiva provides decent results because it has more of an open environment, letting the camera detect classes. Another situation to consider is that the farther away a vehicle is from the camera, the harder it will be to distinguish between light or heavy. Many trucks have the same front as cars, making the classification difficult from a far distance.

The last test was performed to understand how bad weather and low luminosity affected the detection of traffic, where the test dataset was composed of 500 images of each scenario. The results are displayed in Figure 5.5.

It's clear that extreme conditions affect the results, but the decrease in precision is not that significant. The majority of problems in these situations are due to light exposure. For example, when it's raining, the video becomes too grey, meaning that a light grey vehicle can disappear by "merging" with the road. In some conditions, it's even hard for a human to recognize the said vehicle, making it impossible for the algorithm to detect it. Another example of poor light exposure is the detection performed at night, where the lights from the vehicles and light poles cover almost half the video, being quite difficult to detect. This problem majorly occurs in DunaMeio and RiaAtiva since they are almost parallel to the road.

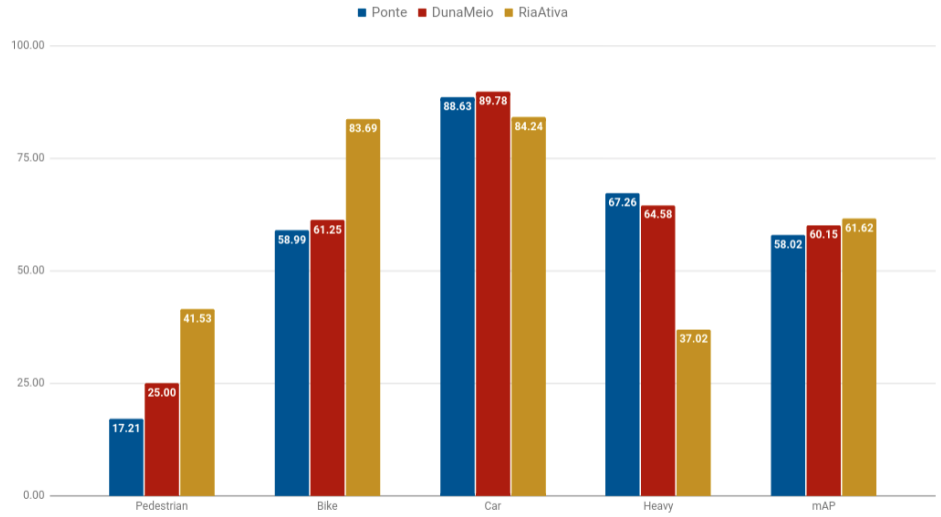


Figure 5.5: Difference in precision values in several conditions of light and weather.

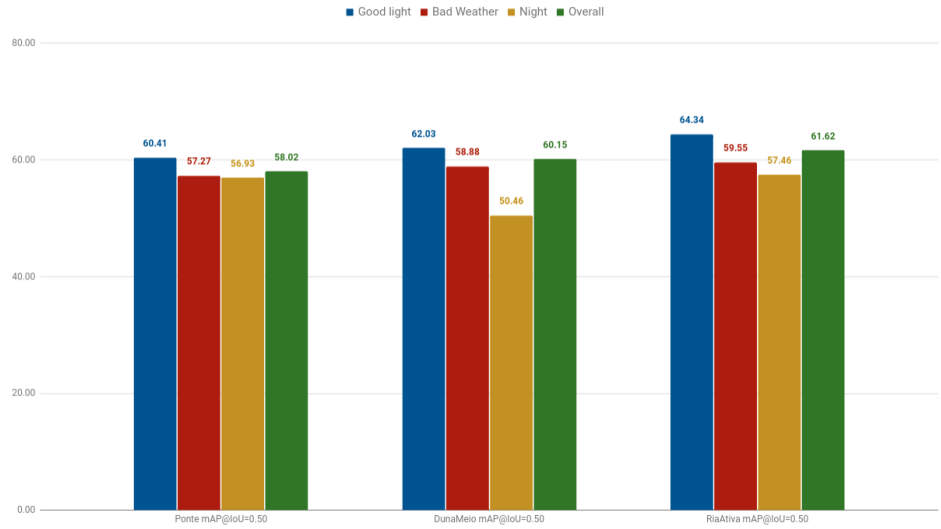


Figure 5.6: Some examples where the light and weather conditions affect the results by a lot.

5.4 Jetson Nano Performance

It was established that the algorithm would be running on the board 24/7, detecting and producing data along the way. In this context, it's crucial to understand how the performance of the hardware changes over time. Here we run the system for twelve hours straight, capturing the FPS every hour that passes. The test started at 8 a.m in the RSU DunaMeio, where the average, minimum and maximum values were extracted. We don't need to test the accuracy every hour because it's not affected by the hardware. The results were the following:

Even though we got some low values at the start and end of the tested time frame, the fluctuation of FPSs was not that significant, with the difference between the max and min values being 1.43. Nonetheless, this variation was to be expected because there are periods throughout the day where there is heavy traffic. These are usually between 7 a.m and 10 a.m and between 6 p.m and 10 p.m. Although the FPS decrease in these situations, the drop is

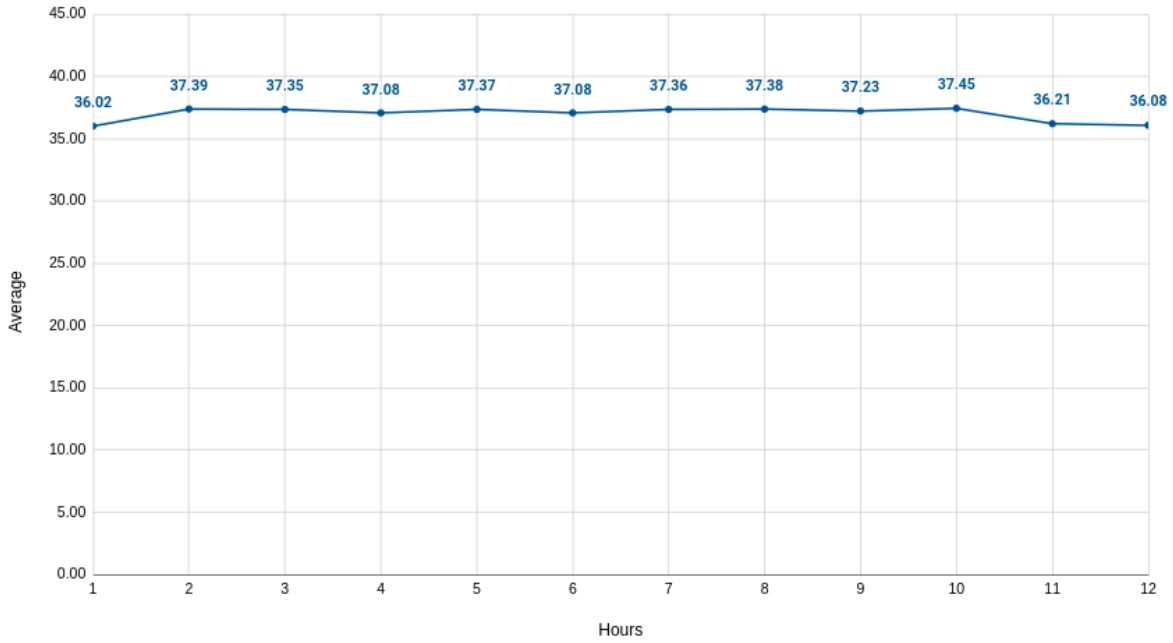


Figure 5.7: Results of the average FPS values in the 12 hour run.

not that significant. Furthermore, the performance is stable at the other times of the day.

Together with the average, we also retrieved the maximum, minimum, and median values, them being 41.90, 3.65, and 41.67, respectively. These numbers show us that, for the most part, the FPSs are higher than average since the mean is lower than the median, concluding that, even thought some drops may occur, the board can sustain real-time performance for the majority of the time.

5.5 Fuser

Finally, we arrive at the Fuser, where several tests were performed to understand how much the algorithm improves classification. To achieve this goal, two different tests were performed, one of them using only data from the DunaMeio camera and radar. Firstly the data structure was modified to give the web application the two-vehicle locations computed. The object position calculated by the radar was depicted with red points, while the location estimated using the camera is given in green. The main goal is to understand how these two differ and if they are synchronized. In Figure 5.8, we can observe that the data computed using the object detector is much more abundant, with the radar providing fewer values. This happens because of the rate that tkDNN is detecting objects in the video. This approach isn't a disadvantage since it's the average of all these values that are utilized, not only one of the many produced. Nevertheless, the locations computed by both methods seem to align quite well since we can see the radar data surrounded by the values produced through the camera.

The final test performed had the objective of understanding how much the sensor fusion improved the accuracy of the classification and which device contributed more to it. To achieve this goal, a 20-minute video, with good weather and light conditions, was used to

Figure 5.8: Values from the radar and cameras with two seconds of difference. The former is represented by the red dots while the latter by green points.

perform the evaluation. In the end, the data from the cameras were used 55% of the time in DunaMeio, 62% in RiaAtiva, and 41% in Ponte. These results are mainly influenced by problems in synchronization since the estimation of the object location is accurate in both devices, as seen above. Heavy traffic jams are also related to the synchronization issue since it's difficult to distinguish which vehicle is which if they are too close together.

Regarding the test for the accuracy of the algorithm, the method used was to check the time where classification was performed correctly and compare it to the overall time that the vehicle was visible on the stream. These results were compared to the old method of classifying traffic, using only the length of the object, to see the improvements made.

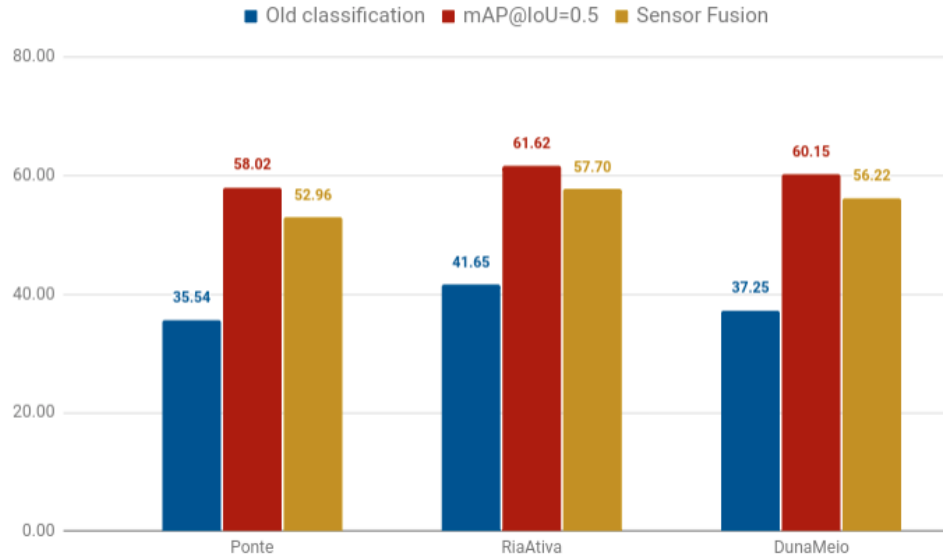


Figure 5.9: Comparation of the old classification with the results from the custom training and the sensor fusion.

The results clearly show an improvement over the old basic method of classification. Nevertheless, the accuracy from the sensor fusion is not equal to the precision that we got

when testing the custom models. This difference occurs because the classification from the object detector isn't always used, as discussed previously. Still, the values from tkDNN being used 100% of the time is an unrealistic scenario, especially during traffic jams. In these types of situations, even for the radar, it's hard to distinguish vehicles when they are so close together. In the end, the results came close to the perfect scenario, being a clear improvement for the traffic classification.

These tests were performed with the help of the web application. The page displayed an icon in the map representing the vehicle's class, accompanied by the real-time stream.

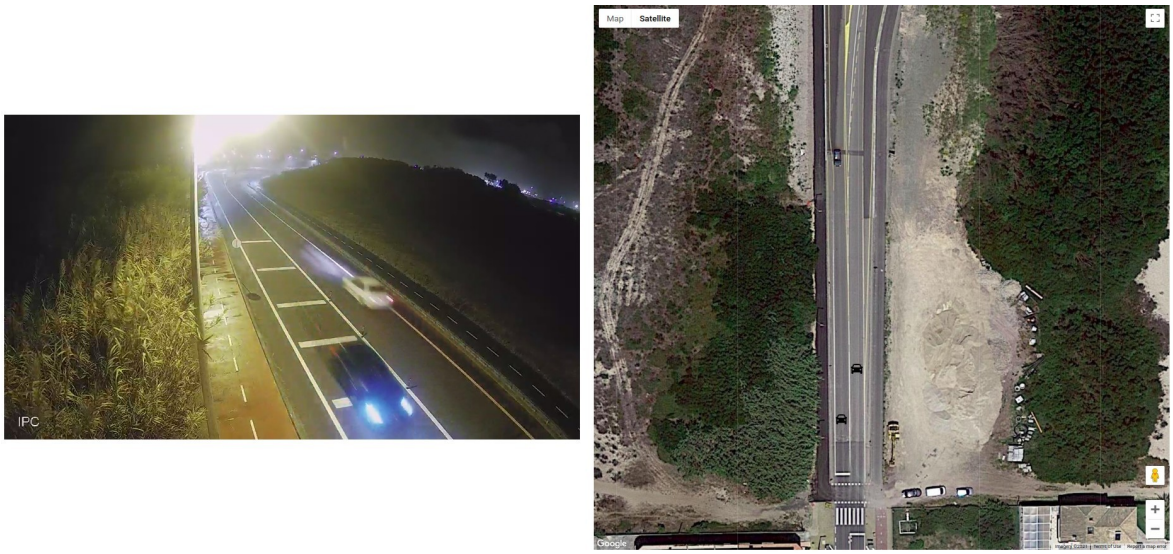


Figure 5.10: Final result of the web application, showing the result of sensor fusion on the right and the stream from the cameras on the left.

CHAPTER 6

Conclusion

The increase in popularity of Smart City solutions presented several systems to attack regular problems. Even though some of these solutions are related to traffic management, this topic not one of the most popular when it comes to research. Smart tourism also falls way behind, but investment in these fields would benefit all the stakeholders, such as local businesses, residents, city council, researchers, investors themselves, and not forgetting, the tourist. The solutions presented in this dissertation use state-of-the-art methods to solve problems encountered in the two domains stated above.

The first system implements services for a domain that the PASMO project was lacking, the application layer. A well-structured API gives data collected from the sensors installed while the dashboard offers an easy way to navigate through this data that tells the state of the environment and the sensors. This solution aims to provide a straightforward way for researchers to utilized sensor data and for users to check the state of the areas of Barra and Costa Nova. The second system let us learn how the conditions of the environment influence object classification as well as traffic problems in areas with a high flow of vehicles, offering an efficient solution to improve accuracy in traffic classification.

After analyzing the critics from the users that tested the public system, we can conclude that the services provided an overall positive change in the way they view the beachside area. The overall feedback from the participants to improve the platform is to implement alerts where users can be warned of possible radar and parking sensor failures. The more advanced evaluators recommend having some kind of interaction between charts since it would give a better understanding of the state of the traffic. Nevertheless, while missing some features, like a mobile application, 70% of the participants said they would use the application in the future, especially for finding a parking spot and search for traffic jams. The API also had a good impact on users with more programming experience, stating that it provides useful data plus the freedom of creating new information using some requests. The majority of the criticisms were related to the documentation page, where it seemed too complex at first view. The lack of more types of data, like the variation in occupation throughout a day, was also an

issue. Nonetheless, more requests can be easily created in the future if deemed necessary, plus users can create their custom data type, like the one recommended. Every service provided by the public system is available in the domain pasmo.es.av.it.pt.

Regarding the traffic classification, the results seem promising, offering a clear improvement over the past method. It also provided a great picture of how object detection is influenced by weather conditions, low luminosity, camera quality, and stream FPSs. Although this system, in particular, could suffer some improvements, it provides a great base solution on how the data from the two devices should be merged.

6.1 Future Work

Since this dissertation is related to the project PASMO, much more types of data can be added to the API with the consequent addition of pages to the dashboard. In particular, the information produced by weather stations that will be installed on the beaches of Barra and Costa Nova in the future. A laboratory version is already available in the API for testing, but its data is useless in the context of this dissertation. Nonetheless, future projects related to the environment and traffic in these areas could utilize the service to offer their sensor data to the public.

By looking at the user's criticism, it's clear that the majority would be happy with a mobile application that provides information about the parking spots and traffic, plus the state of the weather and estimation of people on the beach. This service can be developed in the future by merging data from all the domains, radars, parking, and weather.

Regarding the classification system, the main components that can be improved are:

- The synchronization of values from the two devices
- The detection of pedestrians in Ponte and DunaMeio
- The source of energy feeding the board

Since the RSUs were already installed, we weren't able to easily implement and test a solution to a scenario where the board would be inside the units. Nonetheless, nothing from the algorithm itself would change only the way data was fetched. It's hypothesized that this approach would minimize time differences between the radar data and the stream, considering both devices would be connected to the board. In this scenario, tkDNN would get the camera video like a "webcam" and the radar data through a socket.

The second topic can be solved by labeling more images where pedestrians are found, giving more training data of this class to the model. Still, this approach would work up to a certain extent because, for the most part, the pedestrians blend with the background when they are far away. Another problem to consider is the fact that a bicycle can be classified as a pedestrian as well since a person is riding it. These are both two extreme cases but must be considered nonetheless. The clear solution to this problem is to have a powerful board that can handle a larger model and a higher video resolution, but the cost to reward needed to be well analyzed.

The final point of improvement is the source of energy that feeds the board. Through the implementation and tests, it was identified that Jetson Nano is very unstable with its performance when using different energy sources.

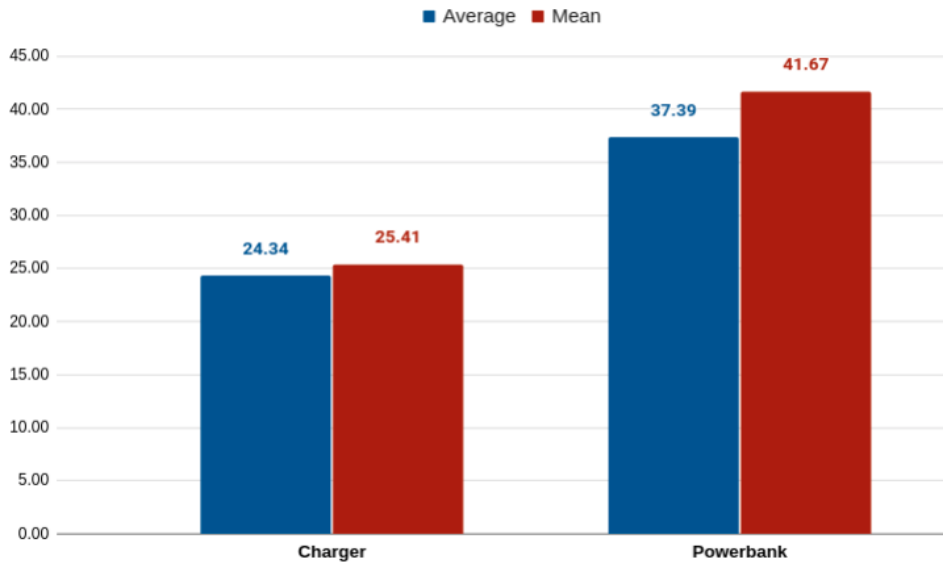


Figure 6.1: The difference in performance when using different sources of energy. The charger, in this case, produces over-current, forcing the system to throttled.

We can see that the drop in performance is significant when not using the right source, making the board throttled due to over-current. This solution is much cheaper than the one above since we would simply need to find the right charger for Jetson Nano.

Bibliography

[1] Globalstats. (). "Desktop vs mobile vs tablet vs console market share worldwide," [Online]. Available: https://gs.statcounter.com/platform-market-share#monthly-201202-202010.

[2] A. Alavi, P. Jiao, W. Buttlar, and N. Lajnef, "Internet of things-enabled smart cities: State-of-the-art and future trends," *Measurement*, vol. 129, Jul. 2018. DOI: 10.1016/j.measurement.2018.07.067.

[3] H. Ritchie, "Urbanization," *Our World in Data*, 2018, https://ourworldindata.org/urbanization.

[4] J. Ferreira, J. Fonseca, D. Gomes, J. Barraca, B. Fernandes, J. Rufino, J. Almeida, and R. Aguiar, "Pasmo: An open living lab for cooperative its and smart regions," pp. 1–6, 2017. DOI: 10.1109/ISC2.2017.8090866.

[5] I. I. de Telecomunicações. (). "Open platform for the development and experimentation of solutions for mobility," [Online]. Available: https://pasmo.pt.

[6] V. Albino, U. Berardi, and R. Dangelico, "Smart cities: Definitions, dimensions, performance, and initiatives," *Journal of Urban Technology*, vol. 22, pp. 3–21, Feb. 2015.

[7] O. Söderström, T. Paasche, and F. Klauser, "Smart cities as corporate storytelling," *City*, vol. 18, no. 3, pp. 307–320, 2014. DOI: 10.1080/13604813.2014.906716. [Online]. Available: https://doi.org/10.1080/13604813.2014.906716.

[8] D. Alaverdyan, F. Kučera, and M. Horák, "Implementation of the smart city concept in the eu: Importance of cluster initiatives and best practice cases," *International Journal of Entrepreneurial Knowledge*, vol. 6, pp. 30–51, Jun. 2018. DOI: 10.2478/ijek-2018-0003.

[9] A. Radecki, "Smart cities - state of the art in europe," *URBACT*, Nov. 2015.

[10] E. Commission. (). "Living labs for regional innovation ecosystems," [Online]. Available: https://s3platform.jrc.ec.europa.eu/living-labs.

[11] M. R. Paul Simpson, "Smart cities: Understanding the challenges and opportunities," pp. 30–51, Feb. 2018. [Online]. Available: https://doi.org/10.1080/13604813.2014.906716.

[12] Z. G. I. of Technology. (). "Smart city series: The barcelona experience," [Online]. Available: https://www.e-zigurat.com/blog/en/smart-city-barcelona-experience.

[13] B. de Portugal. (). "Tourism revenues in portugal increased to 1455.94 eur million in august from 786.61 eur million in july of 2020," [Online]. Available: https://tradingeconomics.com/portugal/tourism-revenues.

[14] S. Anholt. (). "The good country index," [Online]. Available: https://index.goodcountry.org/.

[15] M. Alves, R. Dias, and P. Seixas, "Smart cities no brasil e em portugal: O estado da arte," *urbe Revista Brasileira de Gestão Urbana*, vol. 11, pp. 1–15, Oct. 2019. DOI: 10.1590/2175-3369.011.

[16] P. J. E. R. Prof. Pascual Berrone, "Iese cities in motion index," pp. 1–15, 2019. DOI: 10.15581/018.ST-509.

[17] C. M. de Lisboa. (). "Lisboa inteligente," [Online]. Available: https://lisboainteligente.cm-lisboa.pt/.

[18] N. S. C. Solutions, "Cloud city operations center," Nov. 2017. [Online]. Available: https://es.nec.com/es_ES/solutions_services/smartcity/pdf/Brochure_CCOC_en.pdf.

[19] S. C. I. System. (). "Sharing cities site lisbon," [Online]. Available: https://smartcities-infosystem.eu/scis-projects/demo-sites/sharing-cities-site-lisbon.

[20] C. M. de Aveiro. (). "Aveiro tech city," [Online]. Available: https://www.aveirotechcity.pt/en/about-us.

[21] D. Galego, C. Giovannella, and Ó. Mealha, "Determination of the smartness of a university campus: The case study of aveiro," *Procedia - Social and Behavioral Sciences*, vol. 223, pp. 147–152, 2016. DOI: https://doi.org/10.1016/j.sbspro.2016.05.336. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877042816304165.

[22] C. M. de Aveiro. (). "Aveiro steam city," [Online]. Available: https://www.aveirotechcity.pt/en/Projects/AVEIRO-STEAM-CITY.

[23] Cisco, "Cisco visual networking index (vni) complete forecast update," Dec. 2018. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf.

[24] FIWARE. (). "What is fiware?" [Online]. Available: https://www.fiware.org/about-us.

[25] T. Zahariadis, A. Papadakis, F. Alvarez, J. Gonzalez, F. Lopez, F. Facca, and Y. Al-Hazmi, "Fiware lab: Managing resources and services in a cloud federation supporting future internet applications," pp. 792–799, 2014. DOI: 10.1109/UCC.2014.129.

[26] M. Antunes, J. Barraca, D. Gomes, and R. Aguiar, "Smart cloud of things: An evolved iot platform for telco providers," *Journal of Ambient Wireless Communications and Smart Environments*, vol. 1, no. 1, pp. 1–24, Jul. 2016, ISSN: 2246-3410. DOI: 10.13052/ambientcom2246-3410.111.

[27] Amazon, "Amazon.com annual report 2019," Amazon, 2019. [Online]. Available: https://d18rn0p25nwr6d.cloudfront.net/CIK-0001018724/4d39f579-19d8-4119-b087-ee618abf82d6.pdf.

[28] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (soap)," 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[29] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl)," 2007. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508.

[30] P. A. Castillo, J. L. Bernier, M. G. Arenas, J. J. M. Guervós, and P. García-Sánchez, "SOAP vs REST: comparing a master-slave GA implementation," *CoRR*, vol. abs/1105.4978, 2011. arXiv: 1105.4978. [Online]. Available: http://arxiv.org/abs/1105.4978.

[31] A. Bora and T. Bezboruah, "A comparative investigation on implementation of restful versus soap based web services," *International Journal of Database Theory and Application*, vol. 8, pp. 297–312, Jun. 2015. DOI: 10.14257/ijdta.2015.8.3.26.

[32] T. G. Foundation. (). "Graphql: A query language for your api," [Online]. Available: https://graphql.org/.

[33] R. Chen, S. Li, and Z. Li, "From monolith to microservices: A dataflow-driven approach," pp. 466–475, 2017. DOI: 10.1109/APSEC.2017.53.

[34] J. B. Hong, A. Nhlabatsi, D. S. Kim, A. Hussein, N. Fetais, and K. M. Khan, "Systematic identification of threats in the cloud: A survey," *Computer Networks*, vol. 150, pp. 46–69, 2019, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2018.12.009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128618308259.

[35] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83 088–83 100, 2019. DOI: 10.1109/ACCESS.2019.2924414.

[36] T. Yarygina and A. H. Bagge, "Overcoming security challenges in microservice architectures," pp. 11–20, 2018. DOI: 10.1109/SOSE.2018.00011.

[37] Cisco, "Cisco annual internet report (2018–2023)," Amazon, Mar. 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf.

[38] R. Deil and P. Brune, "Cloudy with a chance of usage? – towards a model of cloud computing adoption in german sme," Jun. 2017.

[39] A. Oludele, E. C. Ogu, K. Shade, and U. Chinecherem, "On the evolution of virtualization and cloud computing: A review," *Journal of Computer Sciences and Applications*, vol. 2, no. 3, pp. 40–43, 2014, ISSN: 2328-725X. DOI: 10.12691/jcsa-2-3-1. [Online]. Available: http://pubs.sciepub.com/jcsa/2/3/1.

[40] J. Fan, C. Guan, K. Ren, and C. Qiao, "Guaranteeing availability for network function virtualization with geographic redundancy deployment," 2015.

[41] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," pp. 1204–1210, 2016. DOI: 10.1109/CCAA.2016.7813925.

[42] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, "Containers and virtual machines at scale: A comparative study," pp. 1–13, Nov. 2016. DOI: 10.1145/2988336.2988337.

[43] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," vol. 60, no. 6, pp. 84–90, May 2017, ISSN: 0001-0782. DOI: 10.1145/3065386. [Online]. Available: https://doi.org/10.1145/3065386.

[44] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," 2019. eprint: 1905.05055.

[45] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016. eprint: 1506.01497.

[46] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014. eprint: 1311.2524.

[47] R. Girshick, "Fast r-cnn," 2015. eprint: 1504.08083.

[48] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016. eprint: 1506.02640.

[49] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," *Lecture Notes in Computer Science*, pp. 21–37, 2016. DOI: 10.1007/978-3-319-46448-0_2. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[50] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," 2016. eprint: 1612.08242.

[51] A. Farhadi and J. Redmon, "Yolov3: An incremental improvement," 2018. eprint: 1804.02767.

[52] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020. eprint: 2004.10934.

[53] Z. Jiang, L. Zhao, S. Li, and Y. Jia, "Real-time object detection method based on improved yolov4-tiny," 2020. eprint: 2011.04244.

[54] P. Shawn DuBravac Carlo Ratti, "The internet of things: Evolution or revolution?" AIG, 2015. [Online]. Available: https://www.aig.com/content/dam/aig/america-canada/us/documents/insights/aig-white-paper-iot-english-digital-brochure.pdf.

[55] F. Nobis, M. Geisslinger, M. Weber, J. Betz, and M. Lienkamp, "A deep learning-based radar and camera sensor fusion architecture for object detection," pp. 1–7, 2019. DOI: 10.1109/SDF.2019.8916629.

[56] Q. Jiang, L. Zhang, and D. Meng, "Target detection algorithm based on mmw radar and camera fusion," pp. 1–6, 2019. DOI: 10.1109/ITSC.2019.8917504.

[57] A. Allström, J. Barcelo, J. Ekström, E. Grumert, D. Gundlegård, and C. Rydergren, "Traffic management for smart cities," in. Dec. 2017, pp. 211–240, ISBN: 978-3-319-44922-7. DOI: 10.1007/978-3-319-44924-1_11.

[58] S. F. Smith, G. Barlow, X.-F. Xie, and Z. B. Rubinstein, *Surtrac: Scalable urban traffic control*, Jun. 2018. DOI: 10.1184/R1/6561035.v1. [Online]. Available: https://kilthub.cmu.edu/articles/journal_contribution/SURTRAC_Scalable_Urban_Traffic_Control/6561035/1.

[59] O. C. A. Council, "Smart cities in taiwan. business directory," 2020. [Online]. Available: https://www.tycg.gov.tw/uploaddowndoc?file=hotnews/202101111401470.pdf&filedisplay=%E8%87%BA%E7%81%A3%E6%99%BA%E6%85%A7%E5%9F%8E%E5%B8%82%E5%B7%A5%E5%95%86%E5%90%8D%E9%8C%84.pdf&flag=doc.

[60] J. Zhu, X. Li, P. Jin, Q. Xu, Z. Sun, and X. Song, "Mme-yolo: Multi-sensor multi-level enhanced yolo for robust vehicle detection in traffic surveillance," *Sensors*, vol. 21, no. 1, 2021, ISSN: 1424-8220. DOI: 10.3390/s21010027. [Online]. Available: https://www.mdpi.com/1424-8220/21/1/27.

[61] W. Chen and C. K. Yeo, "Unauthorized parking detection using deep networks at real time," in *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2019, pp. 459–463. DOI: 10.1109/SMARTCOMP.2019.00088.

[62] M. A. B. Fayyaz and C. Johnson, "Object detection at level crossing using deep learning," *Micromachines*, vol. 11, no. 12, 2020, ISSN: 2072-666X. DOI: 10.3390/mi11121055. [Online]. Available: https://www.mdpi.com/2072-666X/11/12/1055.

[63] G. Camprodon, Ó. González, V. Barberán, M. Pérez, V. Smári, M. Á. de Heras, and A. Bizzotto, "Smart citizen kit and station: An open environmental monitoring system for citizen participation and scientific experimentation," *HardwareX*, vol. 6, e00070, 2019, ISSN: 2468-0672. DOI: https://doi.org/10.1016/j.ohx.2019.e00070. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2468067219300203.

[64] (2021). "Smart citizen," [Online]. Available: https://smartcitizen.me/kits.

[65] (). "Deployment options," [Online]. Available: https://flask.palletsprojects.com/en/0.12.x/deploying.

[66] K. Griffiths. (Dec. 2012). "Uwsgi vs. gunicorn, or how to make python go faster than node," [Online]. Available: https://blog.kgriffs.com/2012/12/18/uwsgi-vs-gunicorn-vs-node-benchmarks.html.

[67] AlexeyAB. (Dec. 2020). "How to improve object detection," [Online]. Available: https://github.com/AlexeyAB/darknet#how-to-improve-object-detection.

[68] ivangrov. (Dec. 2020). "Open labelling," [Online]. Available: https://github.com/ivangrov/YOLOv3-Series/tree/master/%5Bpart%204%5DOpenLabelling.

[69] M. Verucchi, G. Brilli, D. Sapienza, M. Verasani, M. Arena, F. Gatti, A. Capotondi, R. Cavicchioli, M. Bertogna, and M. Solieri, "A systematic assessment of embedded neural networks for object detection," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, vol. 1, 2020, pp. 937–944.

[70] J. Nielsen. (Apr. 1994). "10 usability heuristics for user interface design," [Online]. Available: https://www.nngroup.com/articles/ten-usability-heuristics/.