FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of a hardware module for integration of traffic data from CCTV cameras

Daniel Matos Barros

Master in Electrical and Computer Engineering

Supervisor: António Ramos Silva

Co-Supervisor: Marta Campos Ferreira

July 1, 2022

Abstract

One of the factors that significantly contribute to the evolution of humanity, whether in the economic or even industrial sector, is transport. The rapid growth of this technology has triggered the intensification of traffic at high and worrying levels. These phenomena have led various entities to seek solutions capable of resolving the impact caused by this problem. However, the technologies developed imply huge investments for situations that do not always justify the associated costs.

Therefore, developed in the company *SOLTRÁFEGO, SA*, this dissertation presents a low-cost module capable of integrating traffic data from CCTV cameras to communicate the occurrence of traffic-related circumstances to traffic light controllers. The designed device is a Raspberry Pi 2 running a software system. On the one hand, this software is responsible for communicating through the MQTT protocol with CCTV cameras, supporting a non-relational database system that stores various local information, running a REST API, and checking logical conditions associated with processes representing traffic circumstances. On the other hand, it runs a web interface that allows the configuration of these logical conditions through the connection of nodes and enables the constant monitoring of the system at the level of the integrated traffic data, the status of the configured processes, and the setup of the cameras to which the system will connect. Besides that, it was designed the electrical schema of a board that will make compatible the voltage range at which the Raspberry GPIOs interface and the traffic light controller operate. After the testing and validation phase, it was possible to verify that all the implemented functionalities performed as expected in an environment very close to a real but controlled situation. The central aspect that stands out is the ability of the Raspberry Pi 2 to have had no problems during the most demanding load tests. Still, the module has room for improvement, and a test phase in a real, more intense environment and over a more extended period is needed.

In conclusion, the main requirements were successfully fulfilled, demonstrating this system's potential, which, with robustness and reliability improvements, can fill a previously unexploited gap in the literature and in the market. Furthermore, the industrial application of the solution presented in this report will benefit several entities without the financial capacity to acquire complex and expensive systems.

Resumo

Um dos factores que contribui em grande medida para a evolução da humanidade, seja no sector económico, ou mesmo industrial, é o transporte. O rápido crescimento desta tecnologia tem desencadeado a intensificação do tráfego a níveis elevados e preocupantes. Este fenómeno tem levado várias entidades a procurar soluções capazes de resolver o impacto causado por este problema. No entanto, as tecnologias desenvolvidas implicam enormes investimentos para situações que nem sempre justificam os custos associados.

Assim, desenvolvido na empresa *SOLTRÁFEGO, SA*, esta dissertação apresenta um módulo de baixo custo capaz de integrar os dados de tráfego das câmaras de CCTV para comunicar a ocorrência de circunstâncias relacionadas com o tráfego a controladores semafóricos. O dispositivo concebido é um Raspberry Pi 2 que executa um sistema de software. Por um lado, este software é responsável pela comunicação através do protocolo MQTT com câmaras de CCTV, pelo suporte dum sistema de base de dados não relacional que armazena várias informações locais, pela execução uma REST API, e pela verificação de condições lógicas associadas a processos que representam circunstâncias de trânsito. Por outro, executa uma interface web que para além de permitir a configuração destas condições lógicas através da ligação de nodos, possibilita a monitorização constante do sistema ao nível dos dados de tráfego integrados, o estado dos processos configurados, e o setup das câmaras às quais o sistema se irá ligar. Além disso, foi também desenvolvido o esquema eléctrico de uma placa que irá compatibilizar a gama de tensões a que a interface GPIOs Raspberry e o controlador do semafórico operam. Após a fase de teste e validação, foi possível verificar que todas as funcionalidades implementadas executam como seria expectável num ambiente muito próximo de uma situação real embora controlada. O aspecto central que se destaca é a capacidade do Raspberry Pi 2 não ter tido problemas durante os testes de carga mais exigentes. Ainda assim, o módulo tem margem para melhorias, e é necessária uma fase de testes num ambiente real, mais intenso e durante um maior período de tempo.

Concluindo, os principais requisitos foram implementados com sucesso, demonstrando o potencial deste sistema que, com melhorias na robustez e fiabilidade, pode preencher um espaço na literatura e no mercado que até então não foi explorado. Além disso, várias entidades sem capacidade financeira para adquirir sistemas complexos e dispendiosos ficarão bem servidas com a aplicação industrial da solução apresentada neste relatório.

Acknowledgements

The elaboration and development of this dissertation were only possible due to the commitment and resilience of the author, as well as the people who contributed with various forms of support. Therefore, I express my deepest gratitude to everyone involved in the process. To Professor António Ramos Silva, my supervisor, who was ready, from the beginning, to clarify any doubts and give valuable suggestions in elaborating several topics of work. To professor Marta Campos Ferreira, my co-supervisor, who was always pertinent in her suggestions and for establishing good communication between the author and the company that proposed the project to be developed. I thank them both for the availability, sympathy, and passion they have shown on a subject that I find truly interesting.

To my family, parents, and brother have always encouraged me throughout my journey with support, affection, understanding, and much patience. To my friends, who are a vital support structure and without whom I would not be where I am today. To the comrades in music, Gregório, Hugo, and Cruz, for their companionship, friendship, and brotherhood. To all of my friends from Bubaismo for all the crazy dinner parties. I would also like to take this opportunity to express my gratitude to Pikotado, one of the great friends that the university gave me, for his help in certain complicated moments of my student life. To him and all my comrades from FEUP.

Thank you!

Daniel Barros

"We are what we repeatedly do.
Excellence, then, is not an act, but a habit"

Aristotle

Contents

List of Figures

List of Tables

Abbreviations and Symbols

API	Integrating Application Software
AUC	Area Under the ROC Curve
CNN	Convolutional Neural Networks
COCO	Common Objects in Context
CPU	Central Processing Unit
CSS	Cascading Style Sheets
DADBC	Data Augmented Deep Behavioral Cloning
DMF	Duel Media Filter
FCN	Fully Convolution Networks
FPS	Frames Per Second
GB	GigaByte
GLR	Generalized Likelihood Ratio
GMM	Gaussian Mixture Model
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HMPD	Hybrid Meta-heuristic Pedestrian Detection
HO	Hybrid Observer
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICA	Independent Component Analysis
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPCA	Incremental Principal Component Analysis
LLNF	Locally Linear Neuro-Fuzzy
LoG	Laplacian of Gaussian
LOLIMOT	Local Linear Model Tree Learning
LSTM	Long Short Term Memory Networks
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
MPA	Multi-Page Applications
MQTT	Message Queuing Telemetry Transport

MSE	Mean Squared Error
NoSQL	Not only Structured Query Language
NPM	Node Package Manager
OSI	Open Systems Interconnection model
PCB	Printed Circuit Board
SBC	Single-board computer
SPA	Single Page Applications
SSA	Singular Spectrum Analysis
RAID	Redundant Array of Independent Disks
RAM	Random Access Memory
R-CNN	Region-Based Convolutional Neural Networks
RDB	Relational Database
RegEx	Regular Expression
REST	Representational State Transfer
ROC	Receiver Operating Characteristic
SD	Secure Digital
TCP	Transmission Control Protocol
TSI	Time-Spatial Image
UI	User Interface
VENV	Virtual Environment
VPN	Virtual Private Network
YOLO	You Only Look Once

Chapter 1

Introduction

This dissertation was developed by Daniel Matos Barros in the second semester of the 2021/2022 academic year, within the scope of his Master's thesis in Electrical and Computer Engineering Masters, having as supervisors Professor António Ramos Silva and Professor Marta Campos Ferreira. It was also developed in the company *SOLTRÁFEGO, SA*, the national leader in the mobility market - traffic and parking - and positioned as the preferred partner of smart cities in implementing urban mobility solutions.

1.1 Context

Since the beginning of time, humankind's evolution has been based on transport. Overall, businesses, industries, trade, and other activities depend directly on transportation and its infrastructure to move goods and services. For global economic survival, this becomes a vital aspect that has proven to be one of the primary growth factors of civilisation. The development of infrastructure supporting one of humanity's most essential means of growth did not expand as it should. This way, evolution has moved in another direction. The fact that cities have become the economic growth powerhouses of any country has transformed what would be the expansion of infrastructures into a massive aggregation of the population that makes use of the advantages of having an optimised and efficient transport system. The attraction of inhabitants to large urban areas has sometimes led to intolerable traffic congestion on city streets. The development of urbanisation and the lower cost of owning a vehicle has led to vehicular augmentation in the street [1]. As the infrastructures were not prepared for the intensity of traffic nowadays, society is confronted with circumstances that cause much waste of time and jeopardise mobility security. Several scientific works have been proposed to mitigate these problems, which have provided different solutions. The technology industry has implemented these solutions to improve the efficiency of the products on the market. However, most developed technologies require significant equipment and products or infrastructure investments, restricting this market to large urban centres that can afford those solutions. Thus, there is an opportunity to present a solution that can integrate some benefits of

those technologies. Therefore, it is possible to build a system that does not imply such a complex infrastructure and architecture but allows the use of information collected from the actual traffic state, implementing it directly in traffic controllers with the least amount of physical resources possible. This way, it can bring intelligent traffic regulation within reach of more cities.

1.2 Goals

This dissertation aims to develop a hardware solution that will collect traffic data from CCTV cameras with analytical capabilities and, based on this information, activate electrical outputs connected to traffic light controllers. This project intends to ease intelligent traffic control at a given area. Since this type of system requires significant investments. Thus a low-cost solution, allows entities with less financial capacity to access these smart systems. On the other hand, although the intended system is elementary compared to other industrial solutions, it presents considerable scalability and flexibility. Without compromising the system's operation, the solution can communicate and execute the data received by several CCTV cameras and not be limited to just one. In general, it is expected that the system will allow the traffic light controller to behave according to the data generated by the CCTV camera. Furthermore, the system should have a web interface that allows the remote control of particular configurations and the display of various information relating to traffic - traffic flow, traffic jams, vehicle detection, and so on. Hence, this system has a specific set of tasks to execute, and although it is not complex, its performance must be reliable and efficient.

1.3 Background

The developed system is composed of three blocks: CCTV camera, developed solution and the traffic controller. The first one already developed is a CCTV camera - FLEXIDOME IP starlight 8000i - 6MP [2]. The third one already implemented is a traffic light controller with multiple input pins, and by activating these same pins, it carries out intelligent traffic control. Finally, the second block will be the great purpose of this dissertation that will make the connection between the two previous blocks, increasing each component's capabilities exponentially. Additionally, this block will establish a communication environment between the different blocks to take full advantage of the characteristics of each subsystem. Therefore, the hardware module will communicate with one or more CCTV cameras that perform an intelligent analysis of the captured images and send various helpful information. On the other hand, the module will communicate with a traffic light controller by activating digital pins. As soon as a previously programmed event triggers, the corresponding output/s temporarily assumes the logic value 1.

1.3.1 CCTV Camera Features

[2] The *FLEXIDOME IP starlight 8000i - 6MP* performance line camera offers a 1/1.8" sensor, starlight performance and High Dynamic Range with a 6-megapixel resolution to provide crisp and highly detailed images. This camera has remote commissioning functionality. Using a PC or a mobile device with the Bosch Project Assistant app, it is possible to pan, tilt, roll and zoom *(PTRZ)* and point the camera to the required field of view with a single click - without having to touch the camera or lens. This device performs several functions that are extremely necessary for the operation of the whole system, of which:

- **Overall performance** - The sensor resolution is 3264 x 1840 and has a frame rate of 30 fps. Most objects of the intended application can be captured with good detail, while providing situational awareness.
- **Scene modes** - It is possible to select different scene modes for different situations such as traffic or retail environments.
- **Intelligent streaming** - The camera delivers independent, configurable streams for live viewing, recording, or remote monitoring via constrained bandwidths.
- **Intelligent Video Analytics on edge** - Allows operation in a range of applications. This include airport perimeter protection, critical infrastructure and government buildings, border protection, ship tracking, and traffic monitoring (e.g. wrong-way detection, traffic counting, roadside monitoring of parked cars).
- **Camera Trainer** - The camera allows the user to define objects of interest and generate detectors for them. In contrast to the moving objects that the Intelligent Video Analytics application detects, the Camera Trainer program detects both moving and non-moving objects classifying them.
- **Data security** - On initial setup, the camera is only accessible over specific channels. Web browser and viewing client access can be protected using HTTPS or other secure protocols. The Embedded Login Firewall, on-board guarantees protection from malicious attacks.

Analysing some of the main features of the CCTV camera that will be part of the system, it can be seen the potential that will be exploited with the development of the module. Hence, having a device with an analytical capacity and all the conditions to be included in a traffic control project, this camera's contribution can be greatly expanded.

1.3.2 Traffic Light Controller

This work ends where the traffic light controller begins. Those controllers are prepared to communicate through an interface of digital pins at a voltage between 12 and 24V. Thus, these digital pins will be activated according to the information generated and transmitted by the CCTV cameras and specific configurations established in the module.

1.4 Methodology

In developing this dissertation, an analysis, execution and testing strategy was adopted and divided into the following stages:

- **State of the art** - Research into existing technological solutions for a better overview of the system to be developed.
- **Requirements survey** - To set practical objectives of what the module should be able to perform
- **Overview of development platforms** - Analysis of the tools available and which best serve the development of the system
- **Development** - System implementation process
- **Testing & validation** - Final stage for detecting system-related faults

1.5 Dissertation Structure

In addition to the introduction, this dissertation contains six more chapters. In the chapter 2, a systematic literature review was performed to analyse how the new methods developed will be integrated into physical systems capable of executing those algorithms within the time constraints that traffic management requires. Furthermore, the efficiency of the developed techniques is also evaluated. In chapter 3, the project's requirements and considerations are explained in more detail. In addition, the available options and the most viable ones to be applied in the development of the solution are studied. The project development and implementation are documented in chapter 4 and some options that were taken during the work were also presented. In chapter 5 it were performed several tests to the system so that it is possible to validate most of the implementations and requirements. In the following chapter - 6 - the solution is analysed regarding its performance and the requirements that were implemented and fulfilled. Finally, in chapter 7 the conclusions and future works are presented.

Chapter 2

State of the Art

The subject of traffic covers a wide range of scientific areas. Its development is not only about improving transport conditions but also about how it can be optimized. Ever since the significant problems associated with high traffic flow began to persist, the technology market has been in great demand by large entities for solutions capable of overcoming these phenomena. The subject of traffic covers a wide range of scientific areas. Its development is not only about improving transport conditions but also about how it can be optimized. Ever since the significant problems associated with high traffic flow began to persist, the technology market has been in great demand by large entities for solutions capable of overcoming these phenomena.
This way, a research process was carried out to obtain the scientific papers that could provide ideas, technologies, algorithms, and other essential discoveries in developing the solution for this project. Therefore a systematic literature review was performed. Additionally, a paper based on this chapter was submitted to the Journal IEEE Latin America Transactions.

2.1 Research process

First of all, three databases were selected: *Scopus, Web of science and Inspec* to provide the desired scientific papers. The chosen query used in all databases corresponds to:

```
(((traffic AND (urban OR city) AND (cctv OR surveillance
OR camera))) WN ALL) + {ja} WN DT
```

A review of the keywords related to this dissertation topic was performed to get all articles that somehow explore them. To obtain the latest articles in the field, a 5 years was applied thus, articles that were published before the year 2017 were ignored. The total number of results was 1118 requiring further analysis. The following step consisted of analyzing the title of each article and discarding those that did not fit with the project. This led to 180 papers, but it was still necessary to make a more rigorous selection. As such, the Abstract section of each article was read and only those that were related to the dissertation topic were selected. The final result after all this process was 61 articles. Figure 2.1 illustrates the research process:

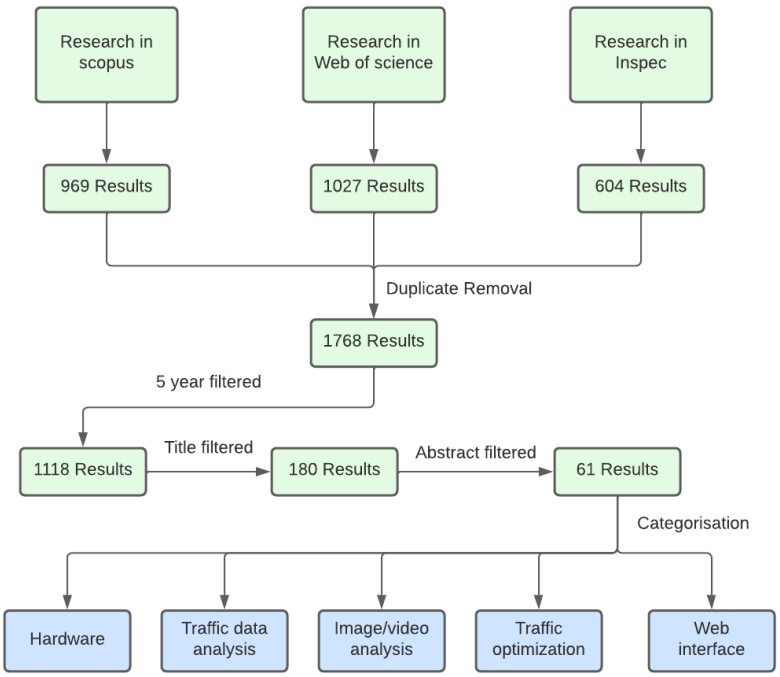


Figure 2.1: Research process scheme

The reading of the 61 articles, allow the identification of the contributions of each and, respective methods applied or developed. To facilitate the management of the information present in all the articles, the objectives in terms of functionality were established along with the respective algorithms. The distribution of the articles by year was homogeneous. Over the years it is possible to perceive an evolution in the methods developed, which were not at all innovative but were based on the previously studied ones. The articles were published in a wide variety of journals. Therefore, through this research method, it is verified that all the scientific articles collected focus essentially on the development of software algorithms for traffic analysis and management using common workstations to perform offline tests.

2.2 Current status and solutions developed

It is possible to assign different features to each science field and create a general expectation of the contribution given by those topics. After the analysis of the selected articles, essentially five major topics emerged: Image/video analysis and traffic estimation, pedestrian detection, traffic management, traffic estimation, forecasting, and web interface. To organize all the information from the articles, the *excel* tool from Microsoft Office was used and, two tables were built. In the first, the articles were associated with the respective categories in which they fell. The second table shows the articles associated with the respective algorithms and objectives. These tables are shown in figures, A.1 and A.2, in appendix A.

Thus, in the following sections, it will be analyzed not only the scientific research and its contributions but also a critical analysis will be made evaluating the contributions of the various scientific papers explored. Overall, for data and image analysis algorithms, machine learning and deep learning based methods are the most relevant and most used. For traffic management, some authors opted for simple algorithms of current traffic analysis and others for prediction algorithms.

2.2.1 Image/video analysis and traffic estimation

Although the focus of the software is not the development of a traffic estimation algorithm and all that this implies, it is essential to understand the state of the art of this type of technology. Furthermore, it is expected that in a future perspective, these new detection systems will be applied in traffic control structures. Here it is possible to assign different features to each science field and create a general expectation of the contribution given by those topics. One of the most important technologies is transforming video into computer data. Several pieces of research were already successfully done, achieving good results with different methods. However, the most recent ones allowed the development of more efficient and robust methods. Traffic analysis is crucial in generating a series of helpful information that allows the system to make the appropriate decisions. The articles unanimously estimate traffic intensity by detecting the number of vehicles on the road being observed by video surveillance cameras. Thus, different methods applied and developed for this purpose are presented below.

The detection of vehicles is based on the characteristics of those vehicles. Thus, in [3] the *Man-Woo Park et al.'s* method is developed, complemented with morphological computational processes. Additionally, the *Background subtraction* technique is often applied as a complement to several algorithms. Moreover, in [4] the *Haar-Cascade* method is used to detect vehicle shapes, however, in [5] the chosen algorithm is based on *Gaussian Mixture model*. On the other hand, in [6] *Background subtraction* is applied, in which the background image to subtract is obtained without any vehicle in the observation area, and morphological processes are simultaneously applied. The *Gaussian Mixture Model* is also applied in [7] but, although there is also support based on morphological processes, this algorithm is used for daytime and nighttime detection which is shown in some papers that detection techniques require different approaches. In most papers, the detection of night vehicles is ignored, nonetheless, in [8] the *Chunming Tang et al.'s* algorithm based on car headlight detection supplemented with morphological processes is achieved. Resorting to more complex methods, in [9] it is used the algorithm *Faster Region-Based Convolutional Neural Networks* (R-CNN) with Domain Adaptation - achieved from labeled daytime images (Source Domain) - to improve the vehicle detection at nighttime. In [10] is created a system where the main methods used in the previous article are the same, however, a different approach is conceived. The proposed method incorporates entropy estimation and a removal framework into the *Gaussian mixture model* to improve the performance of *background subtraction*. This structure is implemented in a *neural network* where learning is applied throughout the execution of the

algorithm.

Similar to the previous article, with *Caffe framework* in [11], a *CNN* is implemented with a deep learning feature capable of identifying and counting vehicles. In [12], is implemented a similar algorithm, however, the neural network toolbox is trained with positive and negative input. Furthermore, since this network is built with *matlab* neural network toolbox, a *blob* analysis is performed to improve vehicle detection. Still, morphological processes are essential, although, implemented according to the needs of the algorithm. A different approach is suggested in [13], even though a CNN is implemented. The main difference corresponds to the training stage, where there is a pre-trained model developed with *MobileNet* tool. This process runs in parallel with the testing sequence. On the other hand, some scientific papers present a complex system with interactions between external servers with different roles, still, the developed algorithm is worth reviewing. Therefore, in [14], there is a combination between several morphological processes and an application trained with an *Neural Network* running in an external server although with real-time communications. Even using a CNN-based algorithm, in [15] the possibility of improving this algorithm relative to articles using the same technology is presented. Starting from an R-CNN model it is added the *Aditional Mask Branch*, *Anchors shape optimization*, *focal loss* and *adaptive future pooling* methods, that along with a *SORT tracker* represent a modern solution for traffic analysis. Finally, a *deep CNN* can support the *You only look once* (YOLO) technology used to detect and classify vehicles, as shown in [16].

YOLO is a derivation of *DarkNet* technology - an open-source neural networks framework - and is useful for this type of applications. Therefore, it is amenable to different ways of implementation. For example, in [17], *YOLOv3* runs along with *RetinaNet* that integrates some useful features in the algorithm. On the other side, in [18] is proposed a complex framework able to detect and classify vehicles and detect their routes. As a result, the computer-vision section is based on *YOLOv3* trained with *Microsoft common objects in context* (COCO). Another promising algorithm is described in [19] considering not only a newly implemented model but also the estimation of the next "Green Time" due to the type of vehicles. Therefore, it is built a *CNN* that supports a *Single Shot Detectors technology* trained with *COCO* dataset.

Other valid solutions are also presented in [20] which proposes a *distributed mean-field-type filtering* (DMF) framework. In [21], is developed a vehicle count learning system based on *Incremental Principal Component Analysis* (IPCA) combined with morphological processes. Since, an affordable solution is essential, in [22] it is developed a low-cost image processing system using *time-spatial image* (TSI) processing. Finally, in [23], there is a concern with weather conditions, where the system will dynamically choose the respective algorithm based on identified nature of the weather. Yet, a full vehicle detection system is proposed based on *Laplacian of Gaussian edge detector* (LoG) complemented, again, with several morphological processes.

2.2.2 Critical Analysis

After analyzing the technologies implemented by scientific works, a comparative analysis between methods was performed and is presented below. This critical analysis considers several aspects beyond their performance in terms of accuracy. Therefore, three essential characteristics were taken into account:

- Real-time performance -considering that the system will execute in real-time, the performance will have to meet certain temporal requirements;
- Computational power of execution required - since the low-cost characteristic is essential, the system needed to run the algorithm will have to be affordable;
- Level of accuracy - it is necessary to obtain a minimum level of accuracy so that the efficiency of the project is not compromised.

Given these criteria, the comparison between the articles may not be entirely correct. Many of the simulations performed contain different data-sets with different weather circumstances, traffic conditions, image resolution, frame rate, and simulation environment. Regarding this last point, most of the authors perform only offline tests and simulations focusing on the performance of the algorithm at the accuracy level. In general, the machines are common workstations that typically are more powerful than SBCs. Moreover, the parameters used to determine performance differ from one article to another. Therefore, some aspects related to each technology used will be analyzed subjectively and using tables.

The table 2.1 presents some considerations related to the papers whose methods developed are based on GMM:

Table 2.1: GMM methods analysis

Articles	Accuracy	Highlights
[10]	75%	Reaches real-time
[7]	nighttime - 94.4% daytime - 96.6%	Detects vehicle components (plate, headlights, etc..)
[5]	89%	Reaches real-time

In general, *GMM* based algorithms can reach a decent performance level, with high accuracy and the capacity of real-time execution. According to the those papers, the methods were implemented with *openCV* libraries. For the simulation environment common desktops were used. So, considering that the frame rate used belongs to the range of [30-50] fps and the resolution was up to 1920x1080 there is some margin to reduce the execution time, decreasing these parameters to adequate values.

The table 2.2 presents some considerations related to the papers whose methods developed are based on YOLO:

Table 2.2: YOLO methods analysis

Articles	Accuracy	Highlights
[17]	-	Boosts traffic flow in 203% compared with static light controller
[18]	$\approx 90\%$	Powerfull simulation environment with low resolution data
[16]	$\geq 96.6\%$	Tested different *YOLO* versions - the best one: YOLOv3-spp

YOLO based algorithms seem to obtain high-level precision results. Data-sets correspond to real traffic videos with typical parameters: fps>20, resolution ranges between [640x480; 1920x1080]. Regarding [17], as a system is proposed from traffic estimation to traffic management, the simulations phase corresponds to the testing of the entire algorithm. An important point is the fact that *YOLO* technology is built on top of *DarkNet* - an open-source C library - thus facilitating developments. In [18], as proposed, it tested the ability to predict and detect the routes of moving vehicles. Although there is no term of comparison with other articles, this feature can be useful so its evaluation is important. The detection error was not beyond 6%.

The table 2.3 presents some considerations related to the papers whose methods developed are based on Neural Networks:

Table 2.3: Neural Networks methods analysis

Articles	Accuracy	Highlights
[11]	AD* = 9; RD* = 0.17	Training step on an external server
[12]	Daytime: car - $\approx 85\%$; motorcycle - $\approx 80\%$; Nighttime: car - $\approx 25\%$; motorcycle - $\approx 11\%$	Average vehicle detection time - 150 ms
[14]	-	Focus on efficiency of data exchange between servers
[13]	$\approx 97.4\%$	Reached real-time
[15]	MAPE $\leq 10\%$	Tested with heavy traffic videos
[9]	nighttime - $\approx 83\%$ daytime - $\approx 95\%$	Best overall comparatively to other methods
[19]	$\approx 72\%$	Large data problems
[24]	Congestion: MAE* ≈ 0.102; MSE* ≈ 0.108	Time consuming training network; Several datasets used on testing stage

- AD - diffusivity parallel to the maximum diffusion direction.
- RD - diffusivity perpendicular to the maximum diffusion direction
- MAE - Mean absolute error
- MAPE - Mean absolute percentage error
- MSE - Mean squared error

First of all, some articles use different evaluation parameters than most. This makes it difficult

to compare performance with other articles. Again, the simulation environments are performed on common workstations with resolution and frame rate similar to the simulations of the previous algorithms. However, the networks were previously trained with approximately 20% of the data-set content using external dedicated servers. Therefore, the accuracy is relatively high, only in [19] lower results were obtained. An interesting point is the importance of an algorithm dedicated to the detection of vehicles at night time if the solution is required to perform at night. For instance, in [12] - which is only committed to having a good performance during the day - the performance at night is extremely low. Thus, the contribution of [9] is essential.

Concerning real-time performance, there do not seem to be major problems of algorithm execution delays. On the other hand, these machines are more powerful than the hardware platform that will be used. Therefore, it is possible that some compromises must be done and parameters be changed. For the scientific papers that were not based on accuracy, given what these parameters represent, the results obtained are quite promising. Still, there is no comparison with the other papers. In an overview, these developed systems prove to be a good alternative to be implemented in the project development. Table 2.4 presents some considerations related to the papers whose methods developed are based on Morphological & other methods analysis:

Table 2.4: Morphological & other methods analysis

Articles	Accuracy	Highlights
[4]	$\geq 99.6\%$	Overal vehicle counting error $\approx 5\%$
[8]	$\geq 90\%$	Tested only at nighttime
[6]	Enconding accuracy [1]	Focus on Video Codec
[20]	$\approx 86\%$	Linear bound for the global error. On matlab implementation performance = 0.6fps.
[21]	$\approx 96\%$	Reaches real-time
[22]	Vehicle classification $\approx 93\%$	The computational time is less than 0.5 s
[23]	Detection = 95% Classification = 87%	Better results for nighttime traffic surveillance and in different weather conditions

Although the target is different from the other papers, in [6] the goal is to build a library-based video coding scheme. This contribution may be essential given the large amount of data that is handled on this type of system. Therefore, there was a clear improvement over the traditional *HEVC* method. Again, some algorithms propose to deal with a nightly execution getting results on par with the other articles. Thus, [23] provides an algorithm capable of achieving good results in both situations.

[1] Algorithm achieved as high as 47.0%, 37.1% and 34.8% BD-rate (Bjontegaard rate difference) reduction than HEVC - traditional method - under RA (random-access) , LDB (low-delay B), and LDP (d low-delay P) configurations, respectively.

Similar to the simulations analyzed before, the real-time is achieved although in [20] there had to be some adjustments. Still, as workstations are used the real-time performance for the same conditions is not assured. A general aspect observed is the fact that the occlusion of vehicles is quite expressive in the errors associated with the simulations. Overall, the contributions of these articles are valid and important for the development of the project.

2.2.3 Pedestrian detection

Intelligent video analytics for pedestrian detection plays a vital role in an effective surveillance system. Also, enhanced pedestrian detection plays a vital role in the field of security and surveillance. Various classification models were in existence for detecting the pedestrians who suffer from a variety of challenges like illumination, pedestrian outfits, gestures, occlusion, lighting, etc., that affect the accuracy of detection. Therefore, many traffic control zones contain pedestrian crossings, thus require the system to manage not only the traffic density but also the number of pedestrians that want to cross the pedestrian crossing of the zone that is under control.

Therefore, in [25], is proposed an algorithm that combines *fully convolution neural networks* (FCN) with *long short term memory networks* (LSTM) to count objects with the highest possible reliability. These objects can be either pedestrians or vehicles, thus it is useful in the same way for the Traffic analysis and estimation section. With the same capacity to identify vehicles and pedestrians, [24] it is designed a robust *hierarchical deep learning* for the task, where a robust metric learning is employed to effectively train the network. Contrary to the previous articles, article [26] focuses only on pedestrian detection based on a *hybrid meta-heuristic pedestrian detection* (HMPD) to enhance the accuracy of the classifier. The algorithm is trained using a set of human and non-human images.

2.2.4 Critical Analysis

Similar to what was done in section 2.2.2, the analysis should essentially involve comparing the efficiency of the algorithms. However, there is the same problem related to the variation of the evaluation parameters and the difference in the simulation environments. Still, the main results are present in table 2.5. Table 2.5 presents some considerations related to the papers that presented pedestrian detection methods:

Table 2.5: Pedestrian detection methods analysis

Articles	Accuracy	Highlights
[25]	Pedestrian and vehicle detection: MAE* $\approx$ 1.54; MSE* $\approx$ 3.02	The window size of the unrolled sequential frames is restricted by the available memory capacity.
[24]	Congestion: MAE* $\approx$ 0.102; MSE* $\approx$ 0.108	Time consuming training network; Several datasets used on testing stage
[26]	$\approx$ 96%	The longer the dataset time, the lower the performance

- MAE - Mean absolute error
- MSE - Mean squared error

In this way, the methods used are compared with traditional methods that were surpassed in all situations. The results are therefore very positive, but there is still room for improvement as they all have limitations. The K. R. S. Preethaa and A. Sabar article [26], present an interesting approach with high accuracy standing out from the others articles, contributing a vaguely applied solution but with good potential.

2.2.5 Traffic management

To create a practical application for the processed data the best way is to build an algorithm capable of optimizing and manage traffic according to given circumstances. Overall, most algorithms are based on the number of vehicles and the number of pedestrians and give priority to the most overloaded traffic zones over the emptiest ones, thus creating an appropriate balance. After the analyses done, a series of decisions are triggered based on the information obtained. Thus, the effectiveness of the decisions taken will be a function of the data provided. Although the circumstances of a traffic control zone may vary, the algorithms developed are not at all divergent. Thus, in [14], M. M. Iqbal et al.'s method is presented, which is based on comparing the number of vehicles in each lane and also using the *round robin* technique. Similarly, in [17] A.Goyal et al.'s algorithm compares the number of vehicles detected in each lane but takes into account the maximum "green time". in [27] the presence of an emergency vehicle is also considered, for which top priority is given. S. Zhou et al.'s algorithm [18], differs in that is more complex which through certain data performs a simulation to understand which is the best way to optimize the traffic given the circumstances.

2.2.6 Traffic data analysis & forecasting

A different perspective that will be most likely to be applied in the project given that the system will receive traffic data in advance will be traffic forecasting. Instead of just estimating the current state of traffic and making decisions based on that alone, the information provided by CCTV cameras can be used to extrapolate and predict in which direction the traffic converges. Therefore, unlike the systems studied above, when faced with a system that previously extracts and provides a series of data related to the state of the traffic, another approach is required. Several techniques have been implemented where algorithms are developed that predict the evolution of traffic. Furthermore, traffic congestion may have several origins such as the high number of vehicles, blocked roads, accidents, etc. Thus, this task is complex not only due to the high density of data that is provided but also due to the difficulty associated with interpreting it.

To analyze the real-time monitoring parameters in the resource constraint edge devices with higher accuracy, in [28], it was designed a logistic-driven public traffic management system using the *EIXGB* technique at the edge networks. This technique was developed to obtain real-time results based on the monitoring parameters of the public traffic management system with higher accuracy and minimum error. This algorithm is a specific implementation based on *machine learning* techniques - a technology widely used in applications of this scope. With the same purpose of traffic flow prediction, in [29] are proposed 2 different methods. The first, consisted of applying a *locally linear neuro-fuzzy* (LLNF) algorithm using *local linear model tree learning* (LOLIMOT) algorithm for nonlinear identification of traffic flow. Then, the second method was an upgrade of the previous one: a combination of singular spectrum analysis (SSA) and (LLNF) was performed with the ultimate goal of generating patterns that, when combined, build the overall prediction. As an interesting technology that combines two deep learning algorithms - *fuzzy logic and Neural Networks* - is illustrated in figure 2.2 showing how both methods interact:

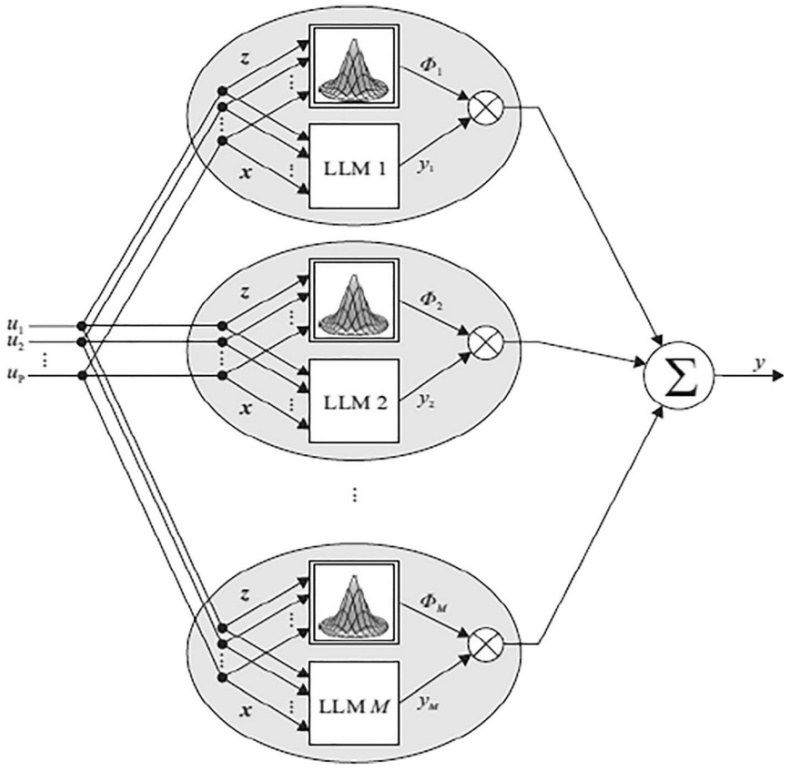


Figure 2.2: *Fuzzy logic* integrated in a *Neural Network* [30]

In this sequence comes an implementation that includes traffic optimization utilizing several input parameters - traffic speed and density, historical data, and Spatio-temporal correlation data. Thus, in [31] is developed the method of S. Chavhan and P. Venkataram, consisting of an emergent intelligence technique at the static agent. This uses analyzed, historical and Spatio-temporal data for monitoring and predicting the expected patterns of traffic density (commuters and vehicles) and travel times in each zone and region. The static agent optimizes predicted and analyzed data for choosing optimal routes to divert the traffic. Thus ensures a smooth traffic flow and reduce the frequency of occurrence of traffic congestion, reduce traffic density and travel time. In the end, the algorithm not only collects data but also does several simulations to optimize the traffic.

To have a reliable system capable of collecting traffic data information provided by multi-touch cameras to determine traffic capacity of the intersection is developed an algorithm based on fuzzy logic technology in [32]. This contribution can be complemented by an interesting system developed in [33] - *data augmented deep behavioral cloning* (DADBC) - whose objective is to imitate the problem-solving skills of traffic engineers. For this, it is built a parallel learning framework, that incorporates machine learning techniques for solving decision-making problems in complex systems. Sometimes knowing the number of vehicles on the road or the number of pedestrians is not enough to make the best decision, as you may be facing an accident for example. Thus, an important approach is made in [34] in order to estimate and detect traffic incidents based on *independent component analysis* (ICA) and *hybrid observer* (HO) - *generalized likelihood ratio*

(GLR) techniques. Here, it is developed a traffic time series to obtain insight into the traffic flow and to detect traffic incidents. Then, a time series analysis is used to construct complex networks. Next, the ICA technique is proposed to monitor traffic flow. This it is implemented a piece-wise switched linear model based observer to estimate the possible occurrence of traffic incidents.

2.2.6.1 Critical Analysis

To understand the value that each scientific work can add to the project's solution it is necessary to take a look into the performance of the algorithms. A direct comparison between the different methods would not be correct as many methods can co-exist, complementing each other and others can add techniques that include certain aspects that were once not considered by some articles.That being so, in [28] the simulation results prove that the proposed EIXGB technique achieves 87-97% accuracy over the real-time traffic management data-set. The performance was based on classification error observation which is accurate. Another point that shows the added value of the developed algorithm is the comparison to standard methods. The final results are presented at table 2.6.

Algorithm	Simulation 1	Simulation 2	Simulation 3
Random Forest	0.7	0.71	0.82
Decision tree	0.67	0.67	0.84
K-nearest Neighbour	0.54	0.54	0.8
Ensemble Learning	0.81	0.82	0.91
EIXGB	**0.87**	**0.88**	**0.96**

Table 2.6: Precision of algorithms in [28]

In [29], proposes and tests two methods. As expected, the second method, which was an improvement of the first, performed not only better but was also very accurate. By analyzing the graph at figure 2.3, it can be seen the difference between the two algorithms and the respective performance relative to the real situation:

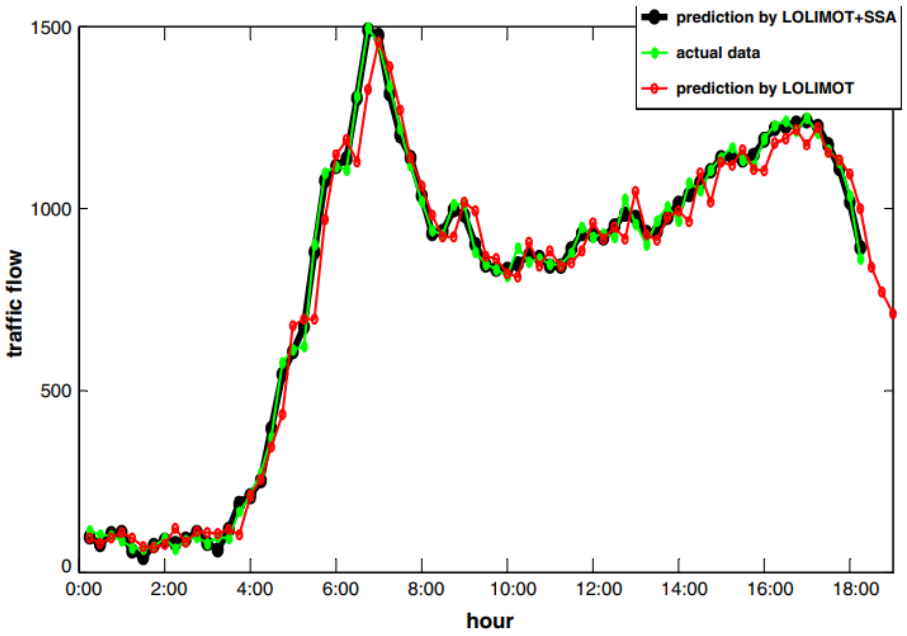


Figure 2.3: Prediction of traffic flow of *LOLIMOT* and *LOLIMOT + SSA* [29]

The system proposed in [31] is a complete algorithm, not only because of the implementation of traffic data analysis techniques but also the decision-making that the system does to optimize traffic flow. First, it is determined the observed, optimized, and predicted traffic patterns for different levels of traffic density as shown at figure 2.4. Then, for the same conditions it was determined the travel time as shown at figure 2.5.

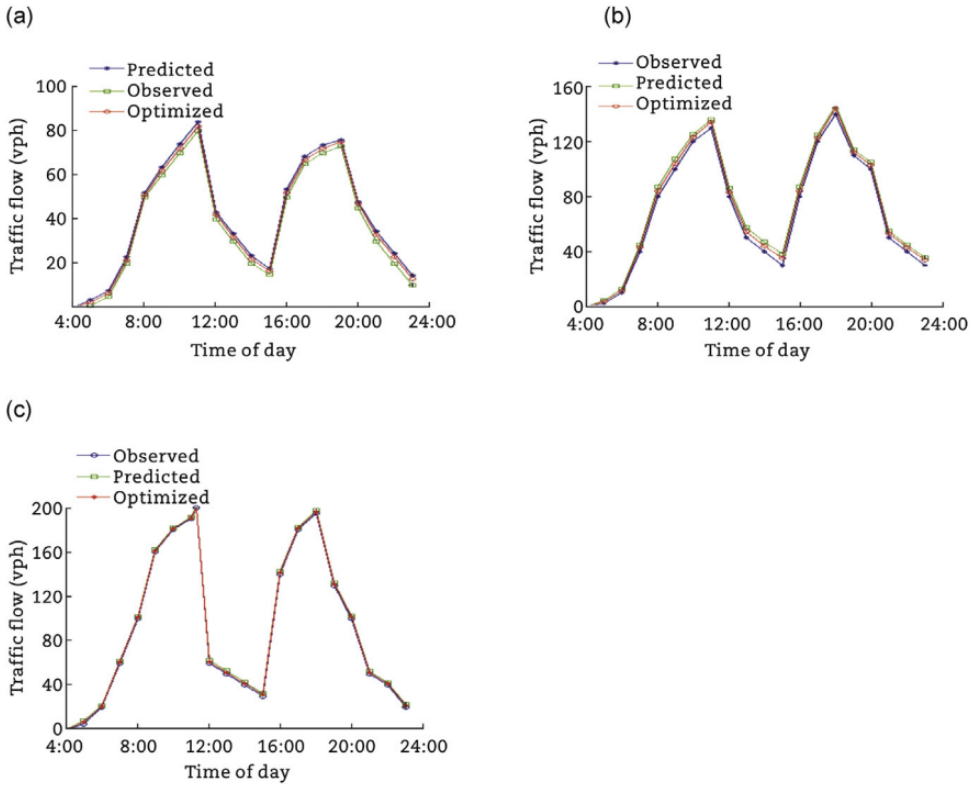


Figure 2.4: Traffic prediction on roads in zones and regions. (a) Low traffic density. (b) Medium traffic density. (c) High traffic density [31]

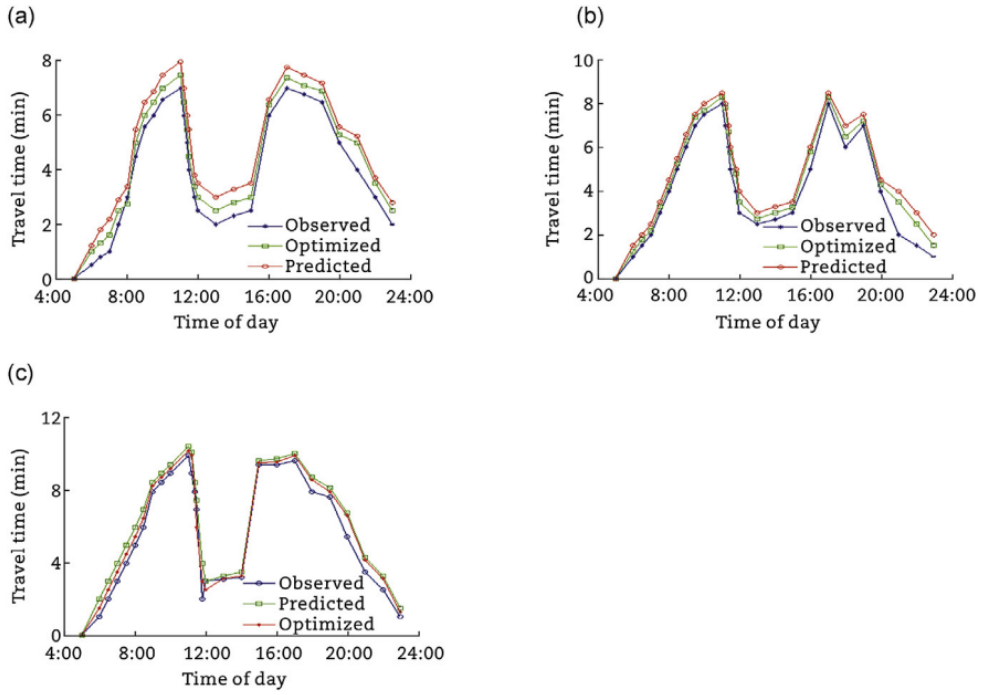


Figure 2.5: Travel time prediction on roads in zones and regions. (a) Low traffic density. (b) Medium traffic density. (c) High traffic density [31]

Overall, the proposed optimization and prediction models of traffic density and travel time are more efficient during medium and high traffic density, and results witness the highest points during peak hours and lowest points during non-peak hours and night periods. This article has given a valuable contribution to the state of the art. Regarding [32], it is possible to identify the characteristics of the intersections that most influenced their traffic capacity, namely:

- Duration of the resolving signal of a traffic light has the greatest effect on the dependent variable;
- Sampling for the maximum possible number of vehicles driving without pedestrians;
- The curvature of the carriageway when turning right;
- Time of leaving the 1st vehicle of the pedestrian crossing, taking into account the distance of 1m to the pedestrian crossing and its release;
- Number of vehicles driven in the 2^{nd} window.

The error of the results is in the range of 5–10%. Therefore, the article's contributions can have a significant impact in reducing the waiting time for a traffic light signal for pedestrians. In [33], the proposed method is highly consistent with human experts. Such a consequence can effectively reduce the workload of human experts and shorten the response time of the traffic system when a congestion event occurs. That said, the main goal of mimicking traffic engineers when it comes to making decisions has been accomplished and proves to be a good alternative to implement. S. Sheikh and A. Regan developed an interesting algorithm [34]. This can help to understand some traffic incidents that are not easy to detect at first glance. The combination of HO-GLR produces reliable detection of traffic incidents can be used to reduce traffic congestion caused by traffic incidents. As result, the proposed method obtained a detection value of 97.58% and 97.80%, a false alarm value of 2.30% and 0.23%, a classification value of 94.83% and 97.18%, mean time to detection value of 2.20 min and 3.75 min, and an AUC [2] value of 94.82% and 95.62% on the AYE and the I-880 data-sets. In all the parameters this method performed better than other standard methods implemented in traditional systems.

In summary, the contributions of each paper are valid and may have a significant impact on the future design solution. All papers proposed algorithms that performed better than traditional methods. On the other hand, some are dedicated to developing systems dedicated to certain parameters that are often neglected but contribute to the effectiveness of a traffic control system.

[2] AUC* - Area Under the ROC (Receiver Operating Characteristic) Curve

2.3 Project Assumptions

After analyzing the emerging methods and technologies, it is possible to generalize the tendency to determine the performance of algorithms by their efficiency. On the other hand, given the circumstances, the parameter of real-time execution is one of the most critical that must be well achieved by this type of system. The type of technologies on which each software is based can also be highlighted. Whether the goal is vehicle detection or traffic optimization, there is a strong tendency to develop algorithms through neural networks. It is expected that in the future, most scientific work in this field will be based on deep learning systems. According to the results, reasonable solutions bringing together a set of essential characteristics to its application in an industrial product have been revealed. As such, the approach to this dissertation project should consider some essential aspects.

It is evident that the chosen module will need to have enough processing power to allow the execution of the algorithms proposed in the methods analyzed. This performance should follow certain time constraints to ensure the viability of this project and future projects. On the other hand, the chosen hardware system does not necessarily have to match the algorithms' requirements. In this case, the solution implemented at the software level should allow its implementation in other more powerful and complex systems, if necessary. Thus, the main contributions of the state-of-the-art analysis focus on which technologies can be implemented in the future. Besides, it helped to understand how the development of this solution can be adapted if these new algorithms start to be implemented at an industrial scale. Furthermore, most papers presented incomplete systems that only relied on image/video analysis. For a traffic management system, it is necessary to go much further, especially in issues related to its optimization and constant monitoring. Thus, this dissertation project acquires a particular relevance by facilitating the addition of "intelligence" to traffic management. Therefore, through the production of traffic data, the next step would be using such data as a control measure, i.e., this will be the module's primary function.

Chapter 3

Initial Approaches

3.1 Problem Description

The development of the module begins where the operation of the previously installed CCTV camera ends. This device allows a deep configuration of several parameters related to image capture angle, zoom, contrast, saturation, brightness, sharpness, and many more. These parameters must be static and well configured, allowing the best performance possible. On the other hand, this camera can perform an intelligent video analysis, i.e., many traffic circumstances are decoded in data. The traffic images captured by CCTV cameras of traffic jams, the number of pedestrians, vehicle number and type, speed detection, object stay time, etc., will be converted into data and sent to the module. The camera is configured using software developed by *BOSCH*®, who also created the camera itself. An essential point to consider is that the camera does not generate traffic data automatically and randomly. The type of data desirable to obtain is generated by assigning specific tasks.The communication with the CCTV camera is native through a standard messaging protocol - *MQTT* following a *publish-subscribe* as a messaging pattern. Since this device operates on a private TCP/IP network, access to its configuration and communication is protected through a VPN. Thus, it is with the *OpenVPN* software that it is possible to interact with it. With this communication system previously established, there is no need for security issues to be considered in the project's development since it will be the company *SOLTRÁFEGO ,SA* that will ensure data security through the implemented system. Considering how this first block of the system operates, it is clear that the module to be developed, regarding the communication process, should: i) be able to communicate through the MQTT protocol ii) have internet access and iii) run the *OpenVPN* software.

The real focus of the system to be implemented is processing the data received at various levels. First, the main goal is to configure a set of processes. As soon as the associated conditions trigger, that event is communicated to the traffic controller by activating digital pins. Those processes represent a traffic circumstance that is desirable to know when it happens, such as traffic jams or even when there are more than ten trucks on that street. This is how the traffic status is signaled

to the traffic light controller so that it can make decisions based on the received information. Therefore, the point of interaction between the system and a supervisor will be an embedded web interface that allows the configuration of processes to be triggered. Furthermore, a data storage system is required to store much local information associated with the configured processes and the data received by the CCTV camera. In addition, beyond all functions that the module must perform in data processing and user interaction, the module must reflect its physical structure in a compact device that can be install in existing infrastructures.

3.2 *Hardware* Approach

The module can be divided into *Hardware* and *Software*. Therefore, the *Hardware* system must support all the communications physically with external services and the *Software's* execution. Those communications include the internet connection so that the module can communicate with the CCTV camera and a digital pins interface - to communicate with the traffic light controller. It should also include some peripherals, such as embedded local memory. Beyond the physical support, enough computing power will be necessary to hold data processing. According with the *Software* goals, it is foreseeable that the tasks it will have to execute do not require performance beyond what an SBC is capable of. Those goals include conducting communications over the MQTT protocol, supporting a WEB interface, and a series of algorithms responsible for data storage, internal data exchange, logical data processing, and digital pin activation.

Analyzing the essential hardware requirements that the module must-have, these are not, from an innovative point of view, something that has been developed and available in the market. The device must be able to run a solid operative system, for example, an embedded Linux OS, since it is expected that not only software will be demanding, but the high flow of data requires an efficient hardware device. In addition, a communication interface such as *Ethernet* or even *RS232* is needed to communicate with external systems. Finally, it is indispensable to have a digital output interface to communicate with decisions reached by the software component. Looking at the basic requirements, three options seem to be available:

- **Industrial computer** - This device provides what is needed and more. Given its capacity, the solution becomes very expensive. At this point, considering that it must be affordable, this option is not suitable at all;
- **Hardware development with controller and electronics from scratch** - In contrast to the previous point, this option can be a low-cost solution; however, developing a device from scratch will impose severe challenges in terms of components compatibility and communication, firmware development, hardware design and even manufacturing itself. Not to mention all the time that would have to be spent on the whole process;
- **Microprocessor development board** - All of the drawbacks pointed out in the previous options are solved by this one. Also, development boards can usually integrate other peripheral devices, which allows them to select an incomplete hardware board rather than being forced

to choose one with all the native tools. Nevertheless, it is essential to know that the more complex a platform is, the more expensive it will be.

3.2.1 Possible Solutions

There is a wide offer in the technological market to meet the required requirements. Thus, through an analysis of the various platforms available, it is possible to understand what type of module suits the project's needs. Therefore, it is necessary to make a previous selection of the various products present in the market. This process of choice is, above all, flexible and volatile. Many boards meet all the mentioned requirements. Thus, the decision process will have to go through other criteria such as price, current availability in the market, and even the usage level of the product, which is reflected in the support by the community that helps to overcome specific problems that may occur during the development of the project. The most common development boards that meet those requirements are the *Single-board computer* (SBC). Therefore it is essential to review the options on the market and figure out which is the best option [35]. Table 3.1, is a comparison between several options available in the market that fit the primary requirements. The parameters present in the table 3.1 result from analyzing the manufacturers' websites and some sales sites such as Amazon [36]. By analyzing several online forums and other websites such as *Stack Overflow*, [37], it was possible to assign a concise utilization level to each board. The *high* usage corresponds to a clear dominance of the board in the discussions among the community. *Medium* usage means that although the SBC is not much discussed, it still has some relevance. Regarding the *low* usage, the board does not have relevant importance in the community.

Table 3.1: SBCs and respective features

Products	CPU Cores	RAM	Usage	Standard price
Raspberry Pi 4 [38]	4	4GB	High	40€
Nvidia Jetson Xavier NX [39]	6	8 GB DDR4x	High	180€
Odroid N2+ [40]	4	4GB DDR4	Medium	75€
Odroid XU4[41]	8	2GB DDR3	Medium	50€
Asus TinkerBoard [42]	4	2GB DDR3	Low	150€
LattePanda Delta 432 [43]	4	4G DDR4	Low	330€
Rock64 Media Board [44]	4	1GB DDR3	Low	20€
AML-S905X-CC [45]	4	2 GB	Low	50€
Banana Pi M5 [46]	8	4GB DDR3	Low	90€
VIM2 SBC [47]	8	3GB DDR4	Low	75€

3.2.2 Solutions Critical Analysis

Apart from a few cases, most SBCs are similar, and even the lowest-performing board will be able to handle the system execution - a priori. Thus, the following criterion would be their price, which, although it can be quite variable, there is a standard that restricts some choices. This way, even if a given board presents a good performance, its price is relatively high, then not the best choice. Considering standard pricing, overpricing is present on some boards: *Nvidia Jetson Xavier NX*, *Asus TinkerBoar* and *LattePanda Delta 432*. With more than one option left, the following criterion should fall on the utilization level of the board. Therefore, popularity will be helpful during the project's development, so it should be considered. The development board that has undoubtedly been the most used in various applications corresponds to the *Raspberry Pi*. Version 4, is one of the most popular ones. An associated disadvantage is the possibility of the product being out of stock. Other boards follow, and although they do not have such a significant impact on the market, they still have their relevance. Boards such as *Odroid N2+* and *Odroid XU4* reveal a good alternative. Overall, there is a wide offer that meets the basic requirements of this project. Thus, it is necessary to carry out an analysis to show the best option. After comparing a few features associated with each board, there were still a few options left. Thus, the current market status must be analyzed to choose the best option when acquiring the module. In this case, given the product's price and size in the community, the *Raspberry Pi* platform is the best solution.

3.3 *Software* Approach

This system requires a simple and robust architecture. As a traffic control technology, it is essential to be reliable, secure, and accurate. In addition, it will operate in a real-time environment, so the development of the algorithms must take into account their runtime. The software system will consist of blocks and sub-blocks with specific functions. First of all, the system should interact with a user and, perform data processing without any external interference in parallel. This operation mode makes clear the need to implement two separated layers from the *Open Systems Interconnection model* - OSI:

- **Presentation layer** - also known as the *front-end* layer, its main component is a *graphical user interface* (GUI) that allows users to interact with the application, accessing its functions and services. Therefore, the web interface that will interact with a user to enable the configuration of processes, data queries, etc., will be implemented under the *front-end* concept;
- **Data access layer** - also known as the *back-end* layer, refers to the development of the "server-side" system. It contains behind-the-scene activities that occur when performing any action on a website (front-end), such as databases and scripting. Thus, the system will perform all processing and activities related to traffic data integration, database storage and management, and web interface management under the back-end concept.

The module's structure is divided into two large blocks. One will be responsible for the interaction with the user - the front-end - and the other will be responsible for the processing and storing of internal data without any access by an external entity - the back-end. It is also possible to view the system interaction structure of this interaction as the client-server model, with the back-end behaving as a server and the front-end as a client.

3.3.1 Front-end approach

A web interface will be the single point of interaction between the system and a user. It will need to perform some functions related to process configuration. Furthermore, this interface will provide a dashboard where several pieces of information related to the data received by the camera will appear, such as traffic flow, number of vehicles, number of pedestrians, and number of queues, among others. As a whole, the goal is to provide a settings page where it will be possible to perform functions related to system administration, namely password change, and changing the MQTT broker IP address that serves as an intermediary in the MQTT communication between the module and the camera.

3.3.1.1 Solutions

When developing a Web interface, it is necessary to understand its purpose. There are some available options for web interface development with advantages and disadvantages for each:

- Development of the code from scratch;
- Usage of a front-end framework.

Thus, some frameworks are explicitly designed to support the development of a web interface - Javascript frameworks. This way, there are several open-source options [48] whose wide use for the most varied purposes demonstrates their added value. Some of the Javascript frameworks are:

- Angular
- Vue.js
- React
- Svelte
- Ember
- Lit
- Alpine.js
- Preact
- Solid
- Stimulus

The technologies presented are probably the most common and complete available. Furthermore, they are all capable of integrating development tools that help with other aspects such as page styling - CSS & HTML frameworks. If the code is developed from scratch, achieving a high level of customization is advantageous. However, it has the disadvantage of being a much more complex and time-consuming process. On the other hand, the use of frameworks is much more appropriate since the purpose of the web interface is in no way atypical, and therefore customization is not essential. Moreover, the main advantage lies in making the development process more straightforward and, at the same time, more functional, wasting less time on details that are already

implemented. Further on, some points will be discussed that will be important when choosing the technology that will support the web interface.

3.3.1.2 Solutions Critical Analysis

The critical analysis of the solutions for the development of the web interface should rely not only on some parameters that assess the efficiency of the technology for this type of application but mainly on the developer's personal experience. Similar to the hardware module selection criteria, the level of use can be a helpful parameter. The interaction between the community is much greater relative to others with less popularisation providing support that could be very useful and convenient when faced with problems associated with the use of the tool. According to [48], figure 3.1 illustrates a chart that shows the most significant usage ratio of some JavaScript frameworks based on the answers of developers when asked about the tools they use:

Figure 3.1: Chart with level of usage of JavaScript frameworks [48]

The usage ratio is defined as follows:

$$\frac{would\ use\ again + would\ not\ use\ again}{total}$$

As such, by developer experience and popularity criteria, three options are chosen: *Angular*, *Vue.js* and *React*. Another interesting parameter is the level of developer satisfaction with the tools they use. Therefore, figure 3.2 shows the satisfaction level of the users of each framework:

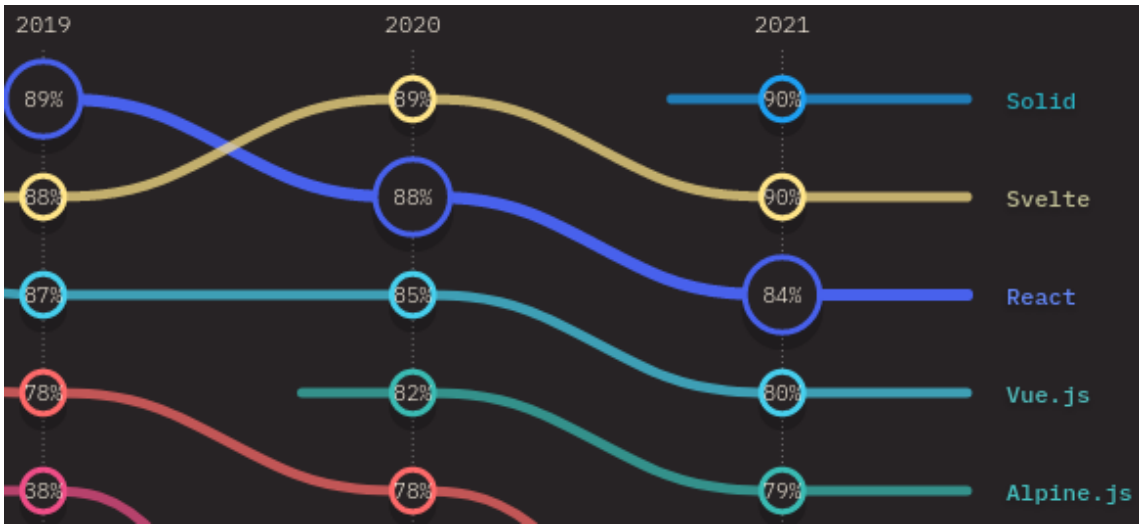


Figure 3.2: Chart with level of satisfaction of JavaScript frameworks [48]

The satisfaction ratio is defined as follows:

$$\frac{would\ use\ again}{would\ not\ use\ again + would\ use\ again}$$

Finally, Y. K. Xing compares three frameworks in various aspects, including data binding, language-based, technical support, volume, and performance [49]. It was concluded that *Angular* contains the most comprehensive functions and features suitable for enormous commercial projects, especially in e-Business. *React* and *Vue.js* are suitable to live stream, communication, blog, and small or medium-scale applications. On the other hand, in [30] it was demonstrated that the *Vue.js* framework is more suitable for the development of Multi-Page (MPA) and Single Page (SPA) applications. React also has similar results in both MPA and SPA categories but of lower values, showing that it can also be used in developing MPA and SPA web applications. Angular, according to the results of the analysis, is not suitable for the development of MPAs. Since Angular has the best result in the SPA category, Angular is currently the best framework for creating SPA applications.

Given the contribution of each of the technologies, since it is a relatively simple web interface, the choice should remain mainly on the developer's knowledge and experience in using these tools. Therefore, considering the community, its intrinsic capabilities, and the developer's experience, the javascript framework was chosen to develop the web interface is *Vue.js V2*.

3.3.1.3 Tools and Frameworks

Now, it is possible to choose some tools that will help develop the web interface. These tools are entirely associated with the Javascript framework. Thus only after choosing the framework is it possible to decide which tools to use. For the development, a *User Interface* component library

was chosen that provides building blocks for productive user interface development with the *Vue.js* framework. There are several options available [50]:

- BootstrapVue
- Quasar
- Vuetify
- Buefy
- CoreUI Vue
- Vue Material
- Vuesax
- PrimeVue

For this project, it makes sense to look into the UI component library, which has a variety of components and is highly reliable. Although *GitHub* stars are not an absolute measure of reliability, they do show a clear indication. So in this case, complete and popular projects such as *BootstrapVue*, *Vuetify*, and *Buefy* would be the best choices. With only three options left, the best choice should fall on the developer's experience, which in this case, *Vuetify* corresponds to the best one. Therefore, *Vuetify* is a Vue UI component framework based on Material Design, a popular design language developed by Google. It consists of UI guidelines for cards, shapes, interactions, depth effects such as lights and shadows, and more. It helps to build professional-looking websites and applications without advanced design skills required. Using its pre-made components, it quickly builds web pages matching the design quality of Google products such as Google Analytics and Gmail. On the other hand, *Vuetify* is not very customizable, presenting a prevalent and little differentiated style and appearance. This point is reflected in a web interface that does not stand out for its style, given the tool's low level of customization. Still, it was discussed earlier that this drawback was not a problem for the system.

3.3.2 Back-end approach

While the front-end system is based only on the development of one block, the back-end will have to take a targeted approach to implement multiple blocks and the communications between them. This system will essentially have to perform the following functions:

- **Integrate traffic data sent by the CCTV camera** - Since the communication between the camera and the module is by MQTT protocol, it should subscribe to MQTT topics to receive the messages;
- **Processing of traffic data** - The provided information will not be compacted enough to be stored and analyzed; therefore, it is necessary to perform filtering and selection of the collected data;
- **Execute a database system** - A data storage system is required to store the integrated traffic data locally;
- **Store the traffic data in database** - The data collected and processed must be properly stored;
- **Analyse configured processes** - In order to determine whether the conditions involved in the process are met, an algorithm needs to be constantly checked by accessing saved traffic data and configured processes;

- **Control digital pins according to processes analysis** - If the process triggers, the pins associated with it must be activated;
- **Communicate with front-end system** - There is a critical need for back-end and front-end systems to communicate, so a proper communication system will have to be implemented;
- **Handle security systems** - The access to the web interface is not open to everyone. Thus, there must be a group of functions dedicated to session management, authentication, and security in verifying access credentials - username and password.

With a well-defined vision of the back-end requirements, it is necessary to define, before choosing the tools to use later on, in which language this system will be developed. There are available several options:

- JavaScript
- Python
- PHP
- Java
- C#
- Ruby
- Go
- Kotlin
- Perl
- Scala

The functions to be performed are simple and entirely achievable by each of the languages above. Thus, the criteria for choice will have to focus on other factors. The first will be its size among the community. It serves significantly to help solve specific problems associated with the development that will eventually arise. The table 3.2 shows the most popular programming languages among the community according to [51], the *Popularity of Programming Language Index*. It was created by analyzing how often language tutorials are searched on Google:

Table 3.2: Popularity of Programming Language Index

Rank	Language	Share	Trend
1	Python	27.85%	-2.5%
2	Java	17.86%	-0.1%
3	JavaScript	9.17%	+0.4%
4	C#	7.62%	+0.7%
5	C/C++	7.0%	+0.4%
6	PHP	5.36%	-1.0%
7	R	4.34%	+0.5%
8	TypeScript	2.39%	+0.7%
9	Objective-C	2.25%	+0.0%
10	Swift	2.05%	+0.3%

Python is an excellent choice for its compatibility with advanced technologies like Machine Learning, Deep Learning, IoT, data science, etc. Meanwhile, it is widely used for web development too. One of the most vital points is its vast collection of standard libraries and intuitive syntax, which helps make the overall coding process simple. On the other hand, *Python* has some disadvantages, such as being slow at run-time, not great for mobile application development, and

high memory consumption. For the project size, these aspects do not present a significant disadvantage. However, if the system is scaled up to enormous proportions, the hardware may have to be changed to support the required computing power. The development of a mobile application would not be at all simple to be done in this language, so another one would have to be chosen. Still, an important factor is the experience of the developer, possibly avoiding a steep learning curve to gain time during the project's development. Considering these two aspects, the language chosen to implement the back-end system is *Python*.

3.3.2.1 Back-end and Front-end interaction

The modus operandi of back-end and front-end interaction will be one of the first aspects to be established. It will affect how the other blocks will operate and, consequently, the tools used for their development. When the web interface is executing in the browser, in most cases, the communication happens through *HTTP* request/response payloads. This interaction is always initiated by the front-end that sends simple *HTTP* requests without a body, form data, and *JSON*-formatted data. The back-end responds with particular contents of the HTTP body:

- HTML-formatted responses;
- Other static files (CSS, JS, images, etc.);
- *JSON*-formatted data.

Given the type of data that will be transmitted and the high volume of traffic that may be transmitted between both systems, the *JSON* tool is ideal for this function. *JSON* - JavaScript Object Notation - is a lightweight data-interchange format. It is easy for humans to read and write and for machines to parse and generate. The JSON messages will be based on a collection of name/value pairs, which can be realized as an object, record, struct, dictionary, hash table, keyed list, or associative array. The type of data to be sent will determine precisely the structure of the message.As mentioned earlier, the back-end system must assume functions related to security and restricting access to the front-end. Those functions are reflected in session management, validation of access credentials, and authentication management. These types of security systems are somehow complex and are not the goal of this project. Therefore, there is a solution capable of simplifying this process and ensuring that the system works securely - *REST API*. This is an application programming interface that conforms to the constraints of *REST* - representational state transfer - architectural style and allows for interaction with RESTful web services. On the other hand, an *API* is a set of definitions and protocols for building and integrating application software. It is sometimes referred to as a contract between an information provider and an information user-establishing the content required from the consumer (the call) and the content required by the producer (the response) [52].

Even though using a REST API is essential in this project, it becomes evident that using a framework that supports its implementation is beneficial. Developing code from scratch will undoubtedly bring problems and a too-long development period, not revealing any associated advantage. Thus, given the programming language being used, it is necessary to choose the framework for web development that best suits the system. As is well known, Python is one of the languages with the most significant number of libraries and tools to aid its development. For web development, the available frameworks are immense, namely [53]:

- **Full Stack Framework** - Full-stack is one of the best Python web application frameworks, a one-stop solution for all app-building requirements. This approach has many databases and components commonly included in the full-stack framework, such as form validation, form generators, and template layouts;
- **Micro-framework** - These web application frameworks are known as lightweight frameworks since they do not offer different patterns and functionalities compared to a full-stack framework, such as a multi-threaded database abstraction layer, form validation, specific tools, and libraries;
- **Asynchronous Framework** - Asynchronous web frameworks allow a user to handle large sets of concurrent connections and are mainly built for Python, which uses the *asyncio* networking library of programming languages.

Given the size of the project, a micro-framework can perform the required tasks. Its low computational weight is an advantage over a full-stack framework. On the other hand, it does not provide some functions, such as form validation, that would previously have to be implemented manually. However, in this case, the front-end will perform these tasks. Analyzing the available micro-frameworks, *Flask* is undoubtedly the most recommended [54]–[56], given its simplicity, flexibility, supporting documentation, and amount of associated plugins. Nevertheless, *Flask* is not async-friendly, is HTML-oriented, and not API-oriented. Setting up a large project requires some previous knowledge of the framework. The only threat this may have on the project implementation is to hinder the development of asynchronous functions, given that it is not optimized for such. Overall, other micro-frameworks such as FastAPI, Bottle, Pyramid, CherryPy, Falcon, Tornado, etc., would serve the purpose and be equally valid. However, they all have associated disadvantages, and the decision became a personal choice by the developer's previous contact with *Flask*.

3.3.2.2 Storage System

The implementation of a database management system will be necessary for:

- Storing traffic data integrated from the CCTV camera;
- Store the configured processes from the web interface;
- Store system-related data such as access credentials, MQTT broker IP, number of associated sensors, etc.

Thus, the structure of this system could be based on two concepts. The first and most common is a *RDB* which stands for relational database, corresponding to a collective set of multiple data sets organized by tables, records, and columns. *RDB*s establish a well-defined relationship between database tables that communicate and share information, facilitating data search, organization, and reporting. *No SQL* - not only SQL - stores information in *JSON* documents instead of columns and rows used by relational databases. The decision that best suits the needs of the database system is based on the analysis of how the sent traffic data is structured. This information is presented in CCTV camera configuration software documentation - [57]. Figure 3.3 is an example of sent data through the MQTT protocol and how an MQTT client receives it:

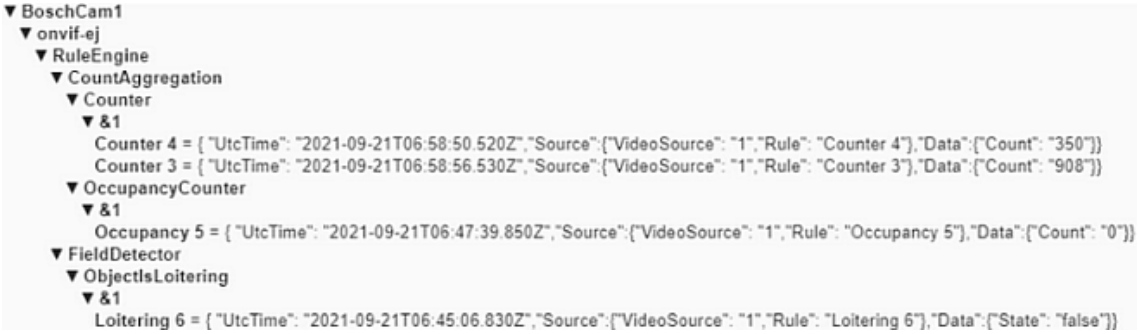


Figure 3.3: Example of MQTT data sent by CCTV camera and received using the client MQTT Explorer [57]

As can be seen, in each message, several parameters can be considered as *keys* with associated *values*. This is precisely how data is structured in *JSON* format. On the other hand, depending on the message, the type of objects can change, and so can the associated value. For example, if it is desirable to receive a *Data* object of type *Count*, the associated value is an *Integer*, but if the *Data* object is of type *State*, the associated value is a *Boolean*. This specific situation makes implementing a *RDB* more complex. In addition, all the manipulation and exchange of data between the back-end and front-end is done in JSON format. Thus, by structuring the storage system in JSON, this process becomes more accessible from the developer's point of view and more optimized for the system. Overall, *NoSQL* tends to be a better option for modern applications with more complex, constantly changing data sets, requiring a flexible data model that does not need to be immediately defined [58]. In addition *NoSQL* is optimized for a relatively fast runtime. The choice of software to implement the database will have to be made with some care. Ideally, the

software should be simple and have many data manipulation tools, such as search, insert, delete, ID lookup, and so on. Furthermore, many of these systems require dedicated servers, which leads to increased consumption of hardware resources such as RAM and CPU power. Looking at the available options, the most popular among all is *MongoDB*, with a very expressive dimension in the community and a complete set of tools. In this segment, there are other options that prove to be quite complete and well-rated in terms of developer opinion such as *AmazonDB*, *CouchBase*, *Redis*, *ArangoDB*, among others. All these options would undoubtedly be more than enough for the needs of the storage system. However, all these database systems are quite complex and, in most cases, require dedicated hardware for their execution. So it is necessary to look for a "lighter" solution. After searching the Github website for NoSQL databases, some alternatives were found [59]. The filtering process was initially based on selecting the document/object-oriented, highest-rated, and lightweight databases. The options that fit the best were *TinyDB* [60] and *LiteDB* [61]. After looking at the support documentation for each, *LiteDB* revealed some obstacles associated with account permissions and the allowed number of files, which would be a major constrain to the project. However, according to the documentation, its entire system proved robust, simple, and functional for the required functions. *TinyDB*, on the other hand, does not have any obstacles associated and has proven to be a straightforward and practical system to implement. Furthermore, its dimension in the community is not very expressive, but it exists and has been increasing. However, there are some threats that can be associated with this software. The system works in an environment that constantly generates data over several months and even years will generate a considerable amount of data. Thus, when data manipulation is performed in the database, for example, analyzing the status of configured processes, the system may no longer be able to do it in real-time. Therefore, there is a concern about the scalability of the storage system that will affect the integration of several CCTV cameras or even the amount of information received and stored by only one camera. So, to solve these problems, these were the considered solutions:

- **Use a hardware system with more resources** - This would make the project more expensive;
- **Implement a more optimized and complex database system** - This new implementation would make the developer's job more difficult and would probably require dedicated hardware - as in the previous point;
- **Implement a relational database system optimized to handle large amounts of data** - This solution would become critical, as it would require changing many system concepts, both in data communication and data manipulation and storage.

Even so, in the validation phase of the project, a load test should be carried out to simulate huge amounts of data and verify the behaviour of the system, namely in terms of execution time. If the result is below what the real-time performance should be, the alternatives described above should be considered to solve the problem.

3.3.2.3 Data Integration

The sending of data by the camera will be constant. Therefore, it is necessary to have a script dedicated to the integration of this data. It is mandatory to communicate through the MQTT protocol, so its structure is established by the MQTT clients and the MQTT broker. It was suggested by the company *SOLTRÁFEGO, SA* to run the broker inside the module and use the software *Eclipse Mosquitto* as the MQTT broker. This software will run parallel to the rest of the system. Therefore, the clients will be the CCTV camera external to the module and the module itself, with a dedicated script. The communication pattern follows the *publish-subscribe* concept. In this case, the camera will publish its traffic data to the broker, and the script will subscribe to specific topics since too much information does not interest the system. The development of this algorithm will use the *Eclipse Paho* library to facilitate the process of subscription and data integration. This library enables applications to connect to an MQTT broker to publish messages, subscribe to topics, and receive published messages. It also provides some helper functions to make one-off publishing messages to an MQTT server very straightforward. Overall it will be a handy tool in developing this subscription algorithm. In addition to the data integration process, it is necessary to have some functions that filter the obtained data and insert them into the database. It makes sense that the data goes through this manipulation and storage process as soon as the data is collected. So similarly using the Python language, any data received will be collected, filtered, and stored in the database.

3.3.2.4 Process Trigger

The back-end system should also analyze the configured processes in the web interface. These processes are stored in the database. It is necessary that the conditions involved in each process are analyzed and that it is determined if the process must trigger an output. If this is the case, the associated digital pins must be physically mapped and placed with the logical value of 1. Here the challenge lies in creating a system that analyzes the rules associated with the configured processes. As suggested by the company *SOLTRÁFEGO, SA*, it should be possible to create logical conditions ($\leq, \geq, AND, OR, =, \neq$), compound variables ($(a+b) \geq K$), generating an event that will be mapped to the module outputs, such as:

- IF (Flow A $\geq$ 30) AND (FLow B $\geq$ 50) $\rightarrow$ activates output 4
- IF Jam in Zone A $\rightarrow$ activates output 3
- IF Stay-time of vehicle $\geq$ 30s in Zone A $\rightarrow$ activates output 2

Like all algorithms to be implemented, it is possible to do it from scratch or use some dedicated tool to facilitate the development process. Initially, the development of an algorithm establishes how the data of the configured process is stored in the database system. Secondly, what will be the strategy of analysis of the logical conditions established. This one became the best of all the previously mentioned advantages of storing and exchanging data in JSON format. This, allows for software to decide how data is stored and how the defined rules and associated conditions are

evaluated. This software is *JsonLogic* and was developed to solve challenges like the pointed ones. This JSON extension is a way to share logic between front-end and back-end code and store it in a database. In addition, this software can be parsed in Python and has a relatively simple source code. Furthermore, accordingly to its documentation, it is possible to add new operators, which can be helpful in the developing process. Although this tool fits perfectly with the needs of this specific project process, JsonLogic does not have much relevance within the community, and its last update was on March 31, 2017. Thus, the implementation of such software creates yet another barrier prone to failures and associated errors. To prevent these threats, it is necessary to test the system at various levels - for long periods, test all the possibilities - to detect associated bugs, which would not have been possible before. Still, the *JsonLogic* is open source, and its code is simple so that errors can be fixed when discovered.

3.4 Preliminary Architecture

Compiling all the ideas developed previously, the result is reflected in an architecture represented by blocks and sub-blocks with particular assigned tasks. This architecture is shown at figure 3.4. This way, a well-structured, robust and efficient software system is required to perform the necessary tasks. For the system to interact with a user through a web interface and perform all data processing, it was realized that the best structure would be based on the *Backend-Frontend* concept. On the front-end, to implement a web interface that allows the configuration of processes, it was decided to use the *Vue.js* framework. In addition, using a UI kit, the design implementation becomes faster with a simple and standard design. *Vuetify* proved to be a solid, well-known, and well-reputed solution. *Vue.js* framework provides the *Fetch* tool that will be used to exchange data, through HTTP requests, with the back-end. Communication between these two blocks is supported on the back-end side by a *REST API - Flask*. The use of this solution essentially serves to perform functions related not only to subsystems communication but also to security. Thus, as the functions to be performed in this system are perfectly within reach of a micro-framework, *Flask* proved to be a suitable solution. In addition, to communicating with the front-end, this block will store data in the database system, using methods from that system. This way, the storage system will be implemented under the concept of Non-Relational Database with the *TinyDB* software. After analyzing the traffic data sent by the CCTV camera and the way it is structured (JSON format), it is considered that it is not necessary to change the way it is stored. The choice of *TinyDB* to implement the database was supported by the tool's simplicity, for not requiring a dedicated server and for having a series of methods and functions at its disposal that, at first analysis, prove to be sufficient for the data manipulation needs. This data comes from the web interface and mainly from the CCTV camera, which transmits it via the MQTT protocol. For this, it is necessary to have an algorithm that behaves as a *subscriber* to constantly subscribe to the topics associated with the camera and integrate the data generated. The *Paho* library, available to be integrated with the Python language, will facilitate the development of this script as far as subscriber functionalities are concerned. As an MQTT communication requires an intermediary - MQTT Broker - it

was suggested by *SOLTRÁFEGO, SA* to use the *Eclipse Mosquitto* software and include it in the module. Therefore, this software will run in parallel with the back-end and front-end. To complete the desired functioning of the system, an algorithm that constantly analyses the processes configured on the web interface remains to be implemented. This algorithm is responsible for verifying if the conditions involved in each process are met for it to trigger. Furthermore, since the goal is to communicate the trigger to the traffic light controller, the script will have to physically map the digital pins associated with the process and assign them the logical value of 1 or 0. These processes will have rules made up of logical conditions. To facilitate the logical analysis of the rules, the software *JsonLogic* will be used, which allows not only the creation of a dynamic and scalable structure to store any logical condition but also provides a method that checks the state of that condition. This way, it will not be necessary to develop an algorithm capable of handling logical conditions from scratch.

To conclude the system architecture analysis, the diagram also shows the interactions between the different blocks. The block responsible for most of the communications is the REST API. With the front-end, this script performs an exchange of JSON messages after receiving HTTP requests by Vue.js. If it receives data, it will store it in the database through methods provided by TinyDB. On the other hand, knowing when a process triggers it is a useful feature. That signal will be communicated to the REST API by the process trigger that is transmitted to the web interface. Finally, the subscriber receives data from the MQTT Broker and will insert it into the database.

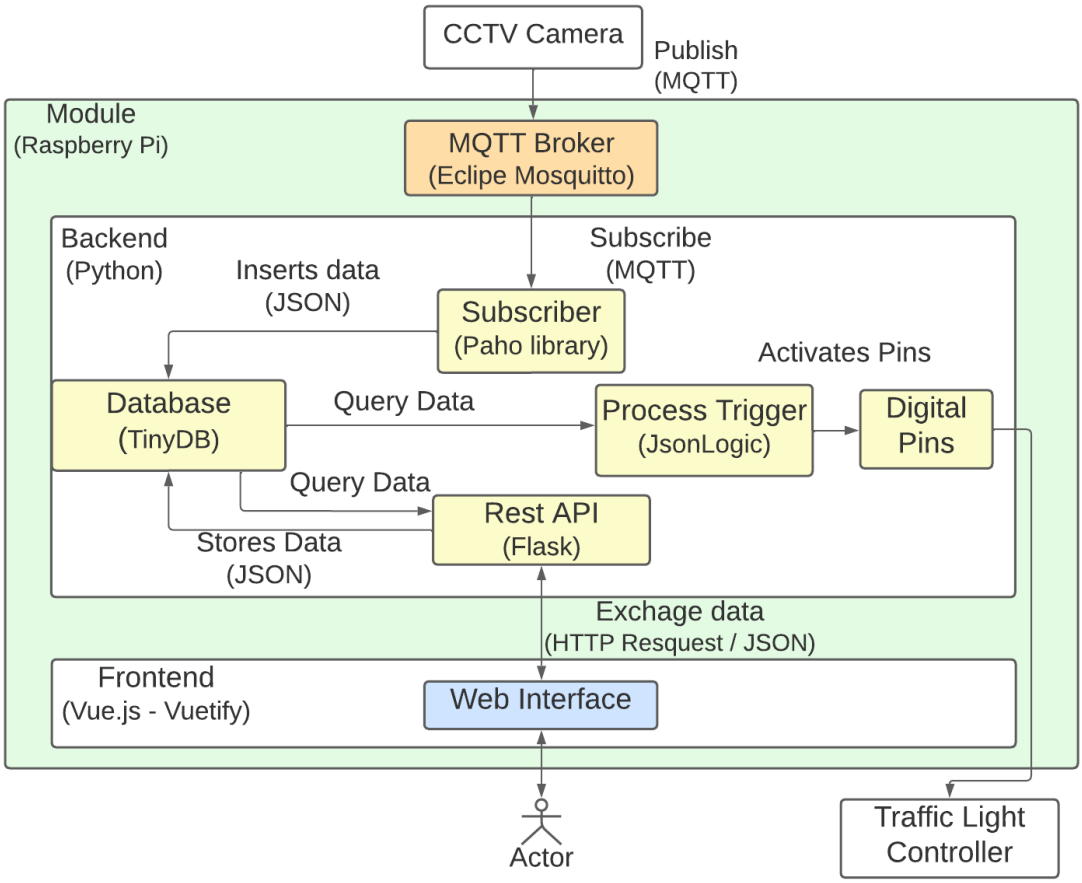


Figure 3.4: System architecture

Chapter 4

Development

In the previous chapter, 3, it were presented the goals and requirements of the project and the solutions that best suit its implementation. This chapter will explain the development of each system associated with the project and justify certain decisions that had to be taken during its implementation. This chapter presents the development of the Hardware and then the Software. Regarding the software section, it is pretty extensive since this was the focus of the project. So, there is a division in implementing the Front-end system because it was the first to be developed. Then it is completed with the approach to the back-end development. Finally, it is demonstrated how the Software was implemented in the Hardware system to obtain a functional module. Additionally, some reflections are made on the advantages and disadvantages of the product and its associated cost.

4.1 Hardware Development

In 3.2, it was concluded that the best approach to the hardware system for this work was an SBC, and from the various options available on the market, the best product would be a Raspberry Pi 4. This system is already complete enough for the project's needs, and in an initial approach, there was no need to explore further how the hardware system would be constituted beyond an SBC. However, the module's output will be connected via a digital pin interface to the traffic light controllers. Thus, the controllers operating in SOLTRÁFEGO's infrastructure work with voltage values in the [12-24]V range. The Raspberry Pi's digital pins work in a much lower voltage region - [0-3.3]V. This makes the direct connection between these two systems incompatible. Thus, this section will describe the developed solution to solve this mismatch problem. The goal is to build a board that can connect to raspberry Pi GPIOs and converts their signal from 3.3V to 12/24V. A simple way to develop this board is to use optocouplers that will act as a digital switch. Hence, when the digital pin from Raspberry is HIGH, the optocoupler will close another circuit that sends a higher voltage to the traffic light controller. The electrical scheme was designed in *Kicad* software. Thus, the approach described is illustrated at figure 4.1. The usage of optocouplers also ads a level of protection to the digital pins of the Raspberry.

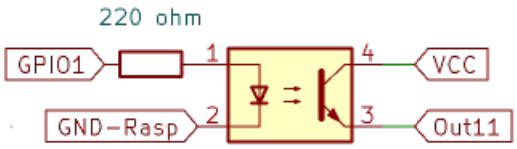


Figure 4.1: Optocoupler base circuit

The optocoupler chosen was Lite-On LTV-846S. It was certified that the current needed to activate the internal switch does not exceed the maximum Raspberry current per digital output pin. In LTV846 datasheet [62] it is shown that the current on the diode around 10 mA is within the operating region of the integrated. On the other hand, Raspberry Pi can provide 16mA per GPIO, which means that the needed current is enough to activate the optocoupler. On that side of the circuit, a resistor is necessary. The calculated value was around 220Ω - a E24 resistor series adapted. According to the manufacturer, at a current of 10mA, the voltage drop across the diode will be approximately 1.15V. Therefore, the resistance can be calculated using equation 4.1.

$$R = \frac{3.3 - 1.15}{10^{-3}} = 215\Omega \tag{4.1}$$

Looking at the available E24 series resistor values, the closest value is 220Ω, which is still perfectly compatible. It was suggested by *SOLTRÁFEGO, SA*to implement two different versions. The goal was to have a board that would invert the logic of all outputs. Thus, the traffic light

controller would be able to receive a logic value of HIGH if the pins were at LOW value. Hence, the first version operates under positive logic and follows the implementation used in figure 4.1. On the other board, to be able to reverse the logic, a small change had to be made. Instead of connecting the VCC voltage to the collector of the transistor, the digital pin is connected. Then, connecting the ground to the transistor's emitter is necessary. Both the VCC and GND terminals are obtained from the traffic light controller. The optocoupler chosen is also manufactured in a block of four. This makes it much more organized to use these blocks when building the PCB. Figure 4.2 illustrates the optocoupler connections scheme, of the positive logic board and, figure 4.3 of the negative logic board.

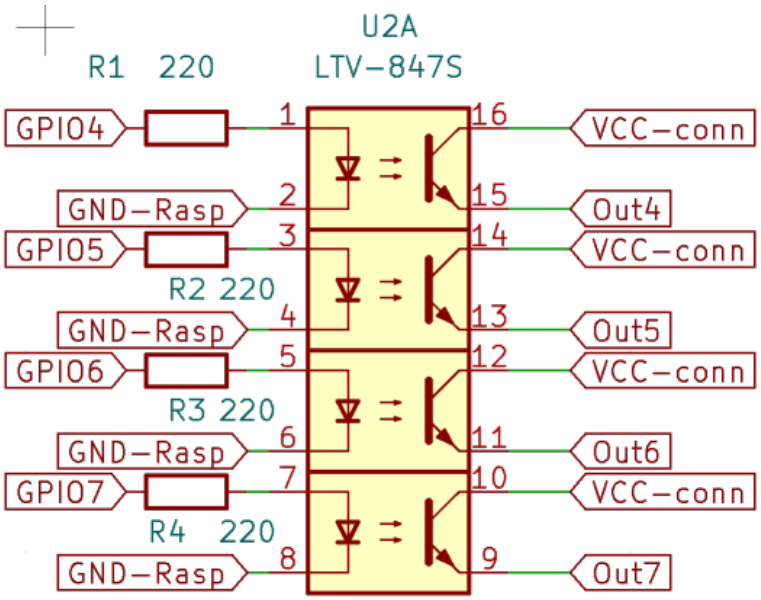


Figure 4.2: Optocoupler connections circuit on positive logic board

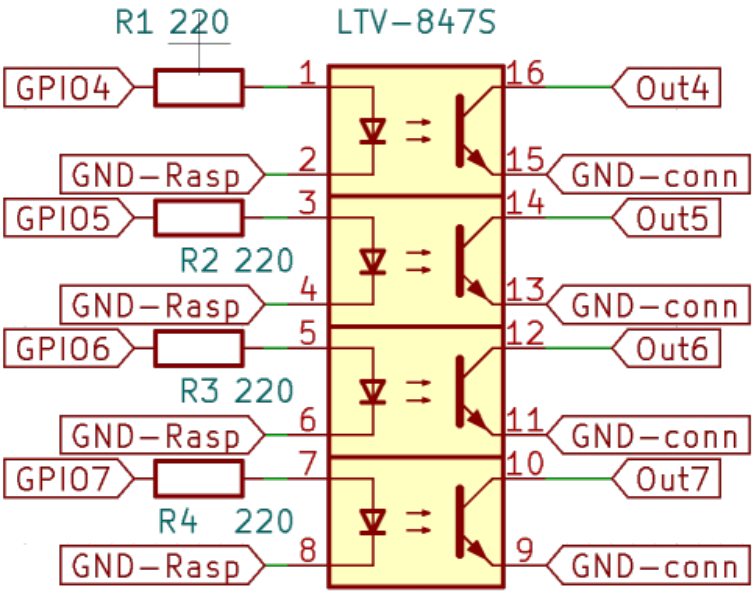


Figure 4.3: Optocoupler connections circuit on negative logic board

The Raspberry connects to the board through a flat-cable. Therefore, a 2x40 pins connector component was placed on the board so that all Raspberry Pi connections would be through this board. Thus, 24 GPIOs are available to be associated with the software's operation and communicate with the traffic controller. An important aspect to explain is that it is not necessary to include a resistor on the transistor side. The traffic light controller has an associated load that limits the current. Therefore, both wiring diagrams have been validated as "suitable for fabrication" on a PCB board by *SOLTRÁFEGO, SA*. Thus, the electrical schematics were converted to PCB layout and thus ready to be manufactured. The complete schemes are presented in appendix B, in the figures B.1 and B.2. The Gerber format files, originated after the PCB layout design, are shown in figures B.3 and B.4.

4.2 Software Development

The software development followed an organization based on the *Scrum* framework but effortlessly. The initial goal was to develop a system with the minimum requirements and perform a series of tests and validations. From that point forward, development focused on increasing the system's scalability until it reached the desired operating level. Thus, both the back-end and front-end systems were developed in parallel. However, initially, there was a greater focus on developing the web interface, mainly at the design level. Starting with the front-end implementation helped to understand how the back-end should work, whether in the structure of the data to be sent or in the timing of the communications and computational processing demands. The whole project was developed in *Arch Linux* environment in a workstation with *Visual Studio Code* software. Further on, it will be demonstrated how the software was implemented on the Raspberry Pi platform.

4.2.1 Front-end

This project was supported and structured using *Node.js* software which allows JavaScript code to be executed outside of a web browser. This platform already includes the *Node Package Manager* tool, which also needed. *npm* is a package manager for the JavaScript programming language maintained by npm, Inc. It is the default package manager for the JavaScript runtime environment *Node.js*. It consists of a command-line client called *npm*, and an online database of public and paid-for private packages called the *npm* registry. The registry is accessed via the client, and the available packages can be browsed and searched via the *npm* website. The package manager and the registry are managed by *npm* Inc [63]. The choice of these tools was mainly due to them being standard software in web development and because they are already associated with the *Vue.js* framework. Next, the default conditions of Vue.js version 2 are chosen. This version was selected and not version 3 because there are no UI kits and other necessary tools compatible with the new version of *Vue.js* yet. The general structure of the code is represented at figure 4.4. Regarding organisation, the interface pages are developed in separate files within the *views* folder.

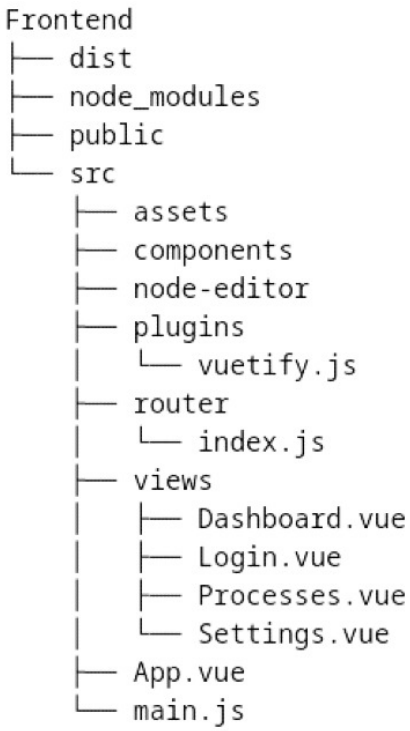


Figure 4.4: Front-end code structure

4.2.1.1 Web Interface Design and Development

The first page that was developed, not only for its simplicity but also because of its importance in security, was the *login* page. Its final look is represented at figure 4.5. The background was developed in *css* and will be the same on all pages. Therefore, it was implemented in the *App.vue* file. This is the main file where the execution starts. The routing and the rendered page are managed by the *router* tool, provided by Vue.js. To develop the page, it was used the UI *card* component provided by *Vuetify*. Then two *form-type* fields and a login button were integrated to submit the entered credentials.

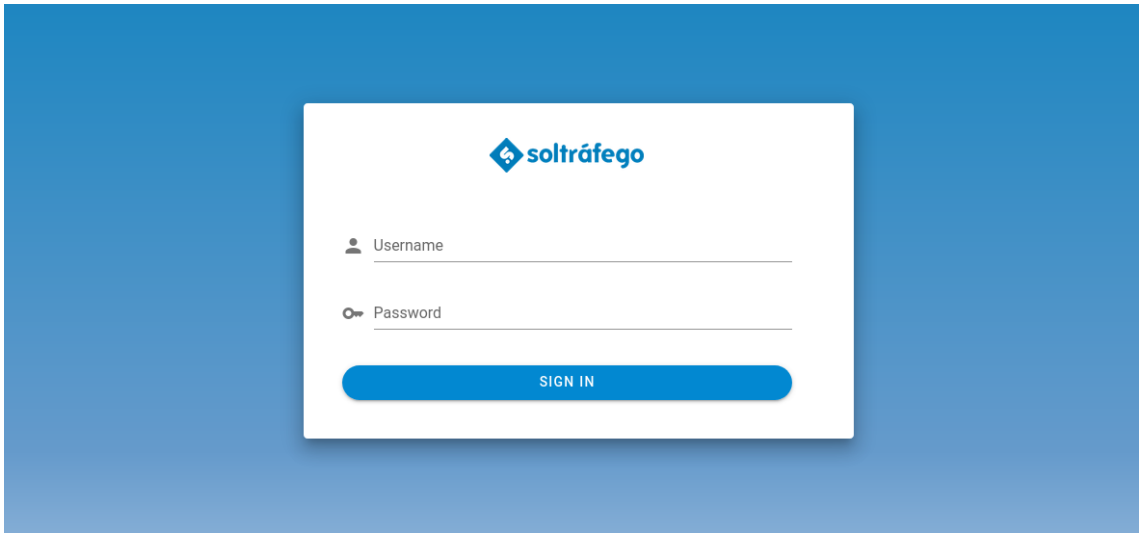


Figure 4.5: Login Page

The advantage of using a JavaScript framework like *Vue.js* reveals itself not only in developing the aesthetics of a web interface but mainly in the data processing that needs to be done. If any fields are left blank, there is a warning, and the data submission cannot be made. This is verified by `validate()` function. Figure 4.6, shows the look of the page when the fields are left empty.

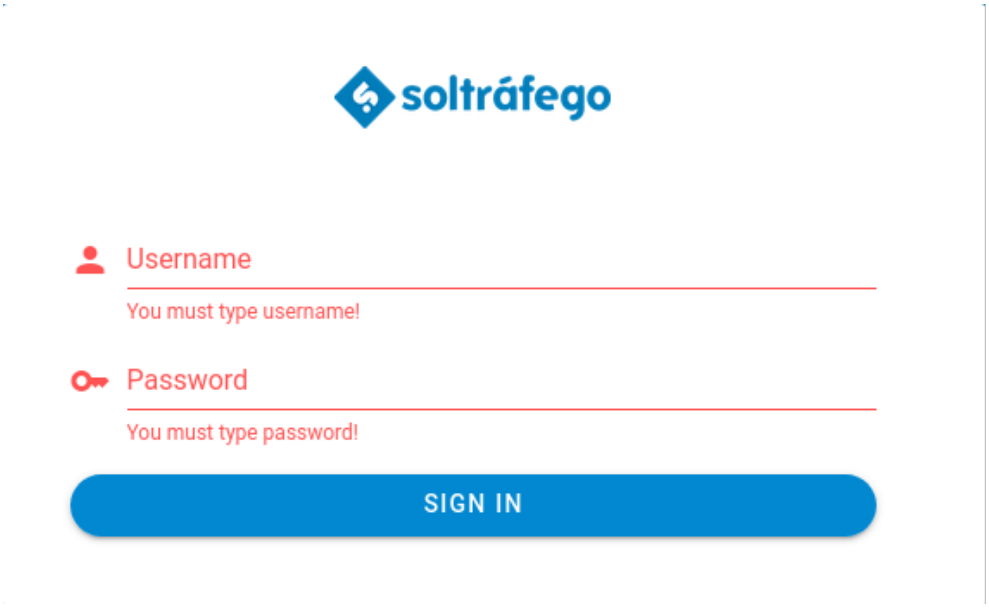


Figure 4.6: Warnings on Login Page

Once the credentials are sent, the code uses a function that can be integrated with this framework. This function is `fetch()`, and it allows to make an HTTP request of whatever method is most appropriate, in this case, a *POST*. As described in the chapter 3, the communication is always initiated by the front-end through an HTTP request. There are several HTTP methods - *GET*, *POST*, *PUT*, *HEAD*, *DELETE*, *PATCH*, *OPTIONS*, *CONNECT* and *TRACE* - each one with a specific task. The most common methods are *GET* and *POST*. However, other ones are going to be used further on. The POST method is the most suitable for sending *Username* and *Password* since POST is used to send data to a server to create/update a resource. The data sent to the server is stored in the HTTP request body. Furthermore, POST requests are never cached, do not remain in the browser history, cannot be bookmarked, and have no restrictions on data length. These features increase the security of the web interface access. Thus, the same function that sends the data awaits an answer from the back-end to confirm if the Username and Password are entirely correct or not. If something is wrong, the access is not allowed, and an error message appears in the browser console, indicating that the credentials are wrong.

The back-end system will check the credentials, returning a token if the credentials are correct. This token will certify the front-end that the user has permission to access the platform. This process will be described further on regarding the implementation of the back-end system. Thus, it is necessary to implement an authentication system in the front-end to manage the tokens received and control access to the web interface. This authentication system starts when the SIGN IN button is pressed. This activates the function that will send the HTTP request to the back-end to validate the entered credentials. This HTTP request is made through the *Axios* tool. So, if it receives a token and the corresponding username, it will store those two variables in a local storage, opening a new session for the user.

To access other pages (shown below), there is the *router* tool responsible for making this connection. This way, whenever access to any page is requested, the *router* will first check if the authenticated user matches the stored token and will send an HTTP request to the back-end to check the token validity. If something wrong happens with this procedure, the router will direct the user to the login page. Finally, it is necessary to delete the token and user variables as soon as the user logs out. This manner, these variables are removed from the local storage and the *router* redirects to the login page. After a successful login, the dashboard page will show the user graphically and interactively the environment captured by the camera. The general appearance of the page is shown at figure 4.7.

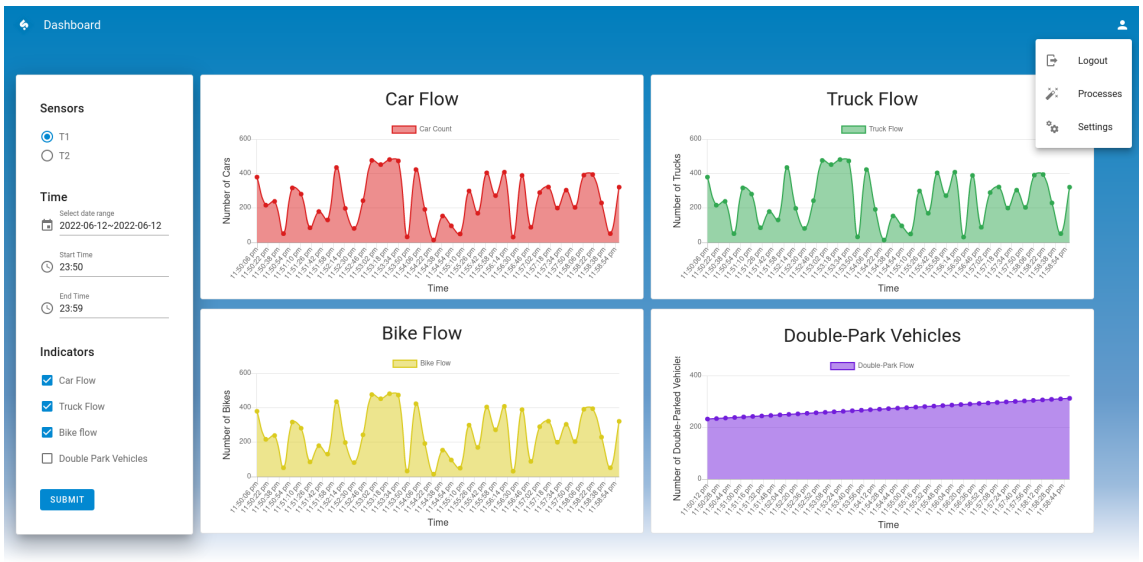
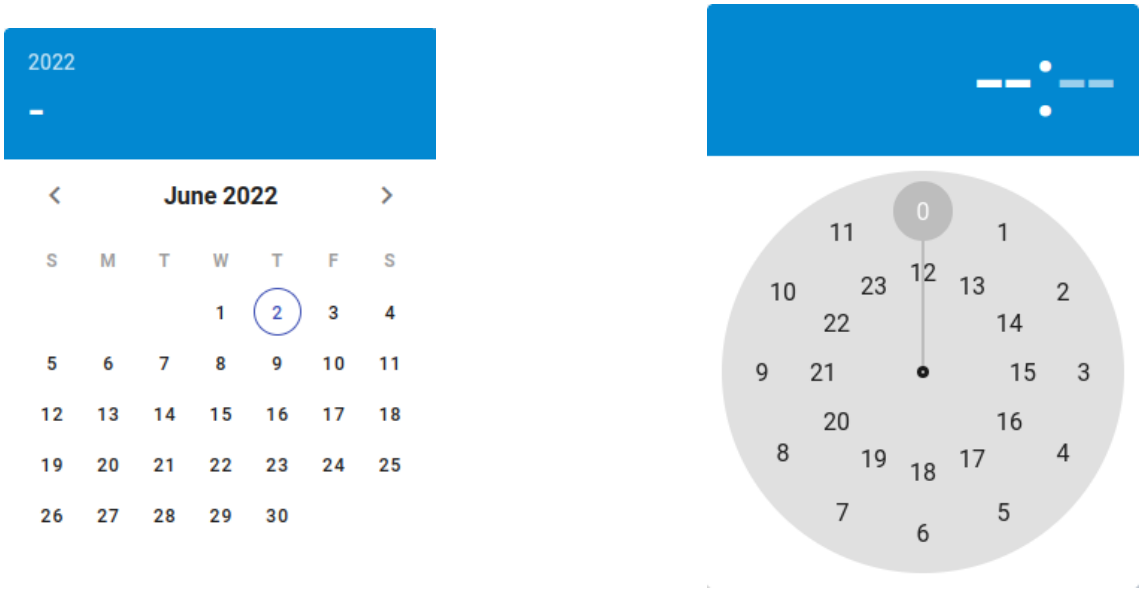


Figure 4.7: Dashboard Page

The page contains some indicators that demonstrate a series of information related to the analysis made by the CCTV camera. This implementation allows a more detailed analysis of the traffic behavior in the area that is being observed. By analyzing the charts, the study of the data characterizing the traffic becomes much more objective, allowing the actions of traffic light controllers in that area to be modified in the future in order to optimize traffic flow. To make this page more interactive and more flexible, on the left side, is the filter panel that allows selecting the data that is desirable to observe. This panel consists of 3 fields. The first allows selecting the area desirable to observe. It was used the *radio* component of *Vuetify* to choose the camera. As the module may be connected to one or more cameras, an HTTP request is sent to the back-end so the system knows how many cameras are connected. This allows this page to show only the available cameras dynamically. Thus, two options will appear if the system is connected to two cameras. If only one camera is connected, only one option will appear, and so on.

The following field allows the user to define the time interval and only observe the data that is part of that interval. To implement the time field, the Date-picker and Time-picker components

were used. These components are pretty complex and efficient. Figure 4.8 shows the look of these components. This system deals with date and time. If the configurations on the front-end are made in a time zone different from the one the camera operates in, there will be a problem associated with the selected time. The JavaScript DateTime system cannot completely deal with that issue. This way, it was used the *Day.js* library that is prepared to transform and make compatible all the systems that use times in their operation. Thus, the simplicity of *Day.js* allows it to deal with this problem efficiently.



(a) Date-picker component

(b) Time-picker component

Figure 4.8: Time field components

Finally, the function of the last field is to select the indicators to be changed. Thus, by selecting the *checkboxes* - component used - and the button submit, the selected charts are going to be updated given the camera that was selected and the time interval. This feature gives the freedom to assign different filters to each graph. Thus, it is possible to observe different time intervals and sensors for each graph at the same time. When this submission is performed, the data is appropriately transformed into a JSON message and an HTTP request with the *GET* method is sent to the back-end. The charts were implemented with *Chart.js* framework. This tool, makes the construction of a chart very simple and efficient. Thus, area-chart type graphs were used, where the main challenge would be just to receive the data from the back-end and build the data structure to be imported by the framework. This data must have the following structure:

```
dataChart: [
{
  name: "Name",
  data: { object1 : key1, object2 : key2, object3 : key3, ... },
},
```

The data received from back-end was already structured in such a way that it only had to be inserted into the *data* object. This way, each time new data arrived it was necessary to update the chart through the function `chart.updateData(dataChart)`, so that the new values were shown. An important feature that has been implemented is a notification indicating the communication with the MQTT broker is down. This feature has been implemented on all pages equally, with the exception of the login page. Thus, every three seconds a request is made to the back-end to check the communication status. If the response associated with the HTTP request is *ok*, it means that everything is normal, otherwise, a notification shown in figure 4.9 appears.

Figure 4.9: MQTT communication down notification

To navigate through the web interface, a simple and standard mechanism has been implemented. In the upper right corner of the figure, there is a *menu* - the component used - which presents a *dropdown* that gives access to the other available pages. Thus, when selecting the desirable page to be accessed, the *router* will internally point to the selected one and render it. This way, there are three options, go to the Processes page, Settings page, or simply Logout. If this one is selected, the session is closed, a message *router* will be directed back to the Login page, as explained before. Then, selecting the Settings page, it is possible to make some changes. The general appearance of the page is represented at figure 4.10.

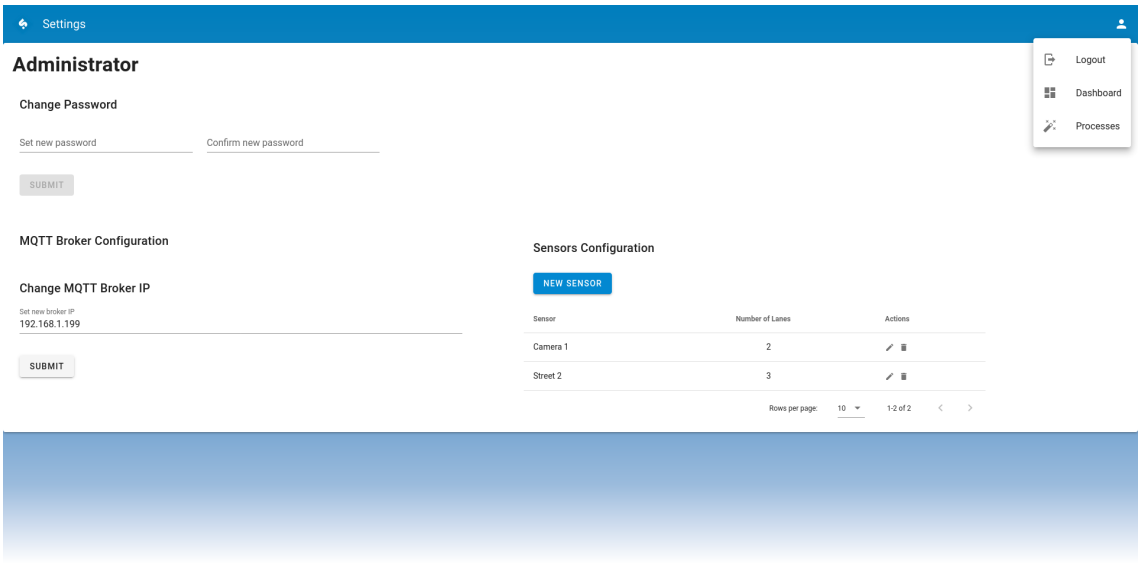


Figure 4.10: Settings Page

There are three fields for changing essential parameters. The first one allows changing the password to access the web interface. Thus, when entering a new password, it will be necessary to confirm it, as it is sensitive data, and the submission will send, like the actions previously described, an HTTP request with the *PATCH* method. Again, this data is being updated in the database, so it makes sense to use this method. Graphically it is used in the *form* component, where rules provided by Vuetify are used to verify if both text-fields are filled and if the content match. Therefore, the validation system is described by rules that check if the content is not null and if both words are the same. In the following field, it is possible to change the IP of the MQTT broker that the module connects to. The IP of the current Broker is displayed, and to change, it is necessary to enter the new IP and submit it. The Vuetify rules system allows integration with JavaScript RegEx - (Regular Expression) object that describes a sequence of characters used for defining a search pattern. This is used to verify if the Broker IP has an IPV4 structure to prevent invalid IP from being entered. The RegEx associated with the verification rule is as follows:

```
/\b((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(\.|$)){4}\b/
```

The error messages generated by the validation system are illustrated at figure 4.11.

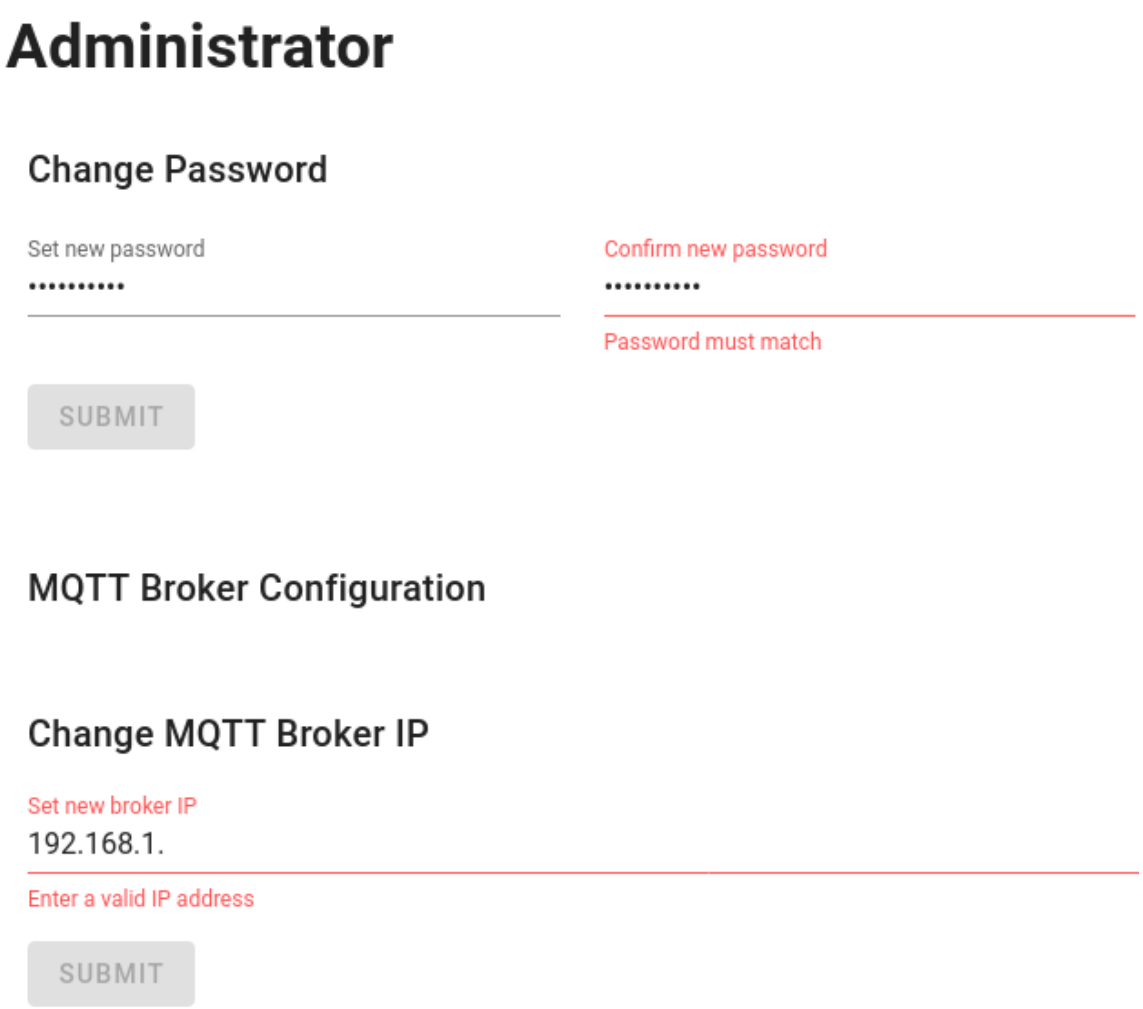


Figure 4.11: Settings Page error messages

For both fields, the changes are only made as soon as the back-end does not find any related problem. In case of an IP change, the back-end will try to connect to the new MQTT Broker. If, it cannot communicate, it is possible to assign a new MQTT Broker IP since the front-end can always communicate with the back-end. Finally, the most critical configuration field on the settings page is the configuration of the sensors. This functionality enables the system to integrate any camera and associate the desirable name and the number of lanes for the road to be monitored. The general appearance of this field is illustrated at figure 4.12.

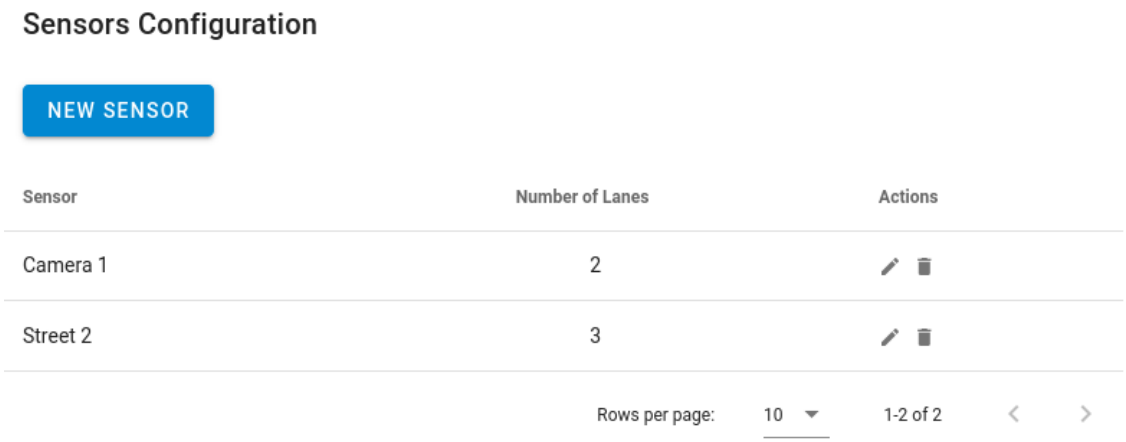


Figure 4.12: Sensors configuration field

This field is created with the Vuetify component *Table*. First, it is essential to analyze all the functionalities implemented in this table. The *NEW SENSOR* button adds a new camera to the system. Thus, selecting this button, a *dialog* is presented to choose the sensor name and the number of lanes associated with the street recorded by the camera. This *dialog* is another Vuetify component and has the appearance displayed at figure 4.13.

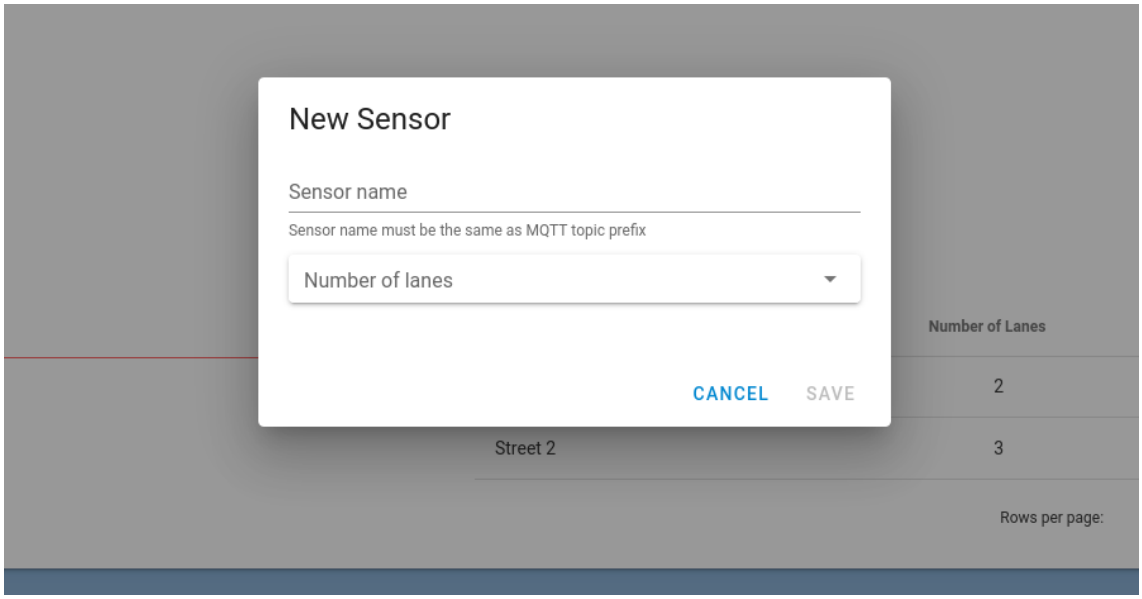


Figure 4.13: New sensor configuration

Under the sensor name, *text-field* is an implemented hint because the name associated with the sensor must be precisely the same as the name associated with the camera topic in its configuration software. Otherwise, those names will not match, and the module will ignore traffic data from that

camera. Therefore, this name filters the MQTT topics subscribed by the module. This situation will be discussed in more detail further on. Then, it is necessary to assign the number of lanes available in that camera. This information is critical since most variables require a zone, which includes the sensor name and the respective lane for the process configuration. To choose the number of lanes available, a *select* component allows selecting until the amount of 20 lanes. Figure 4.14 illustrates not only the select component but also the validation of an empty field. After filling in the fields as required, the SAVE button is enabled, allowing the new sensor to be added to the system. After saving the sensor, an HTTP request is called to insert the new sensor into the database.

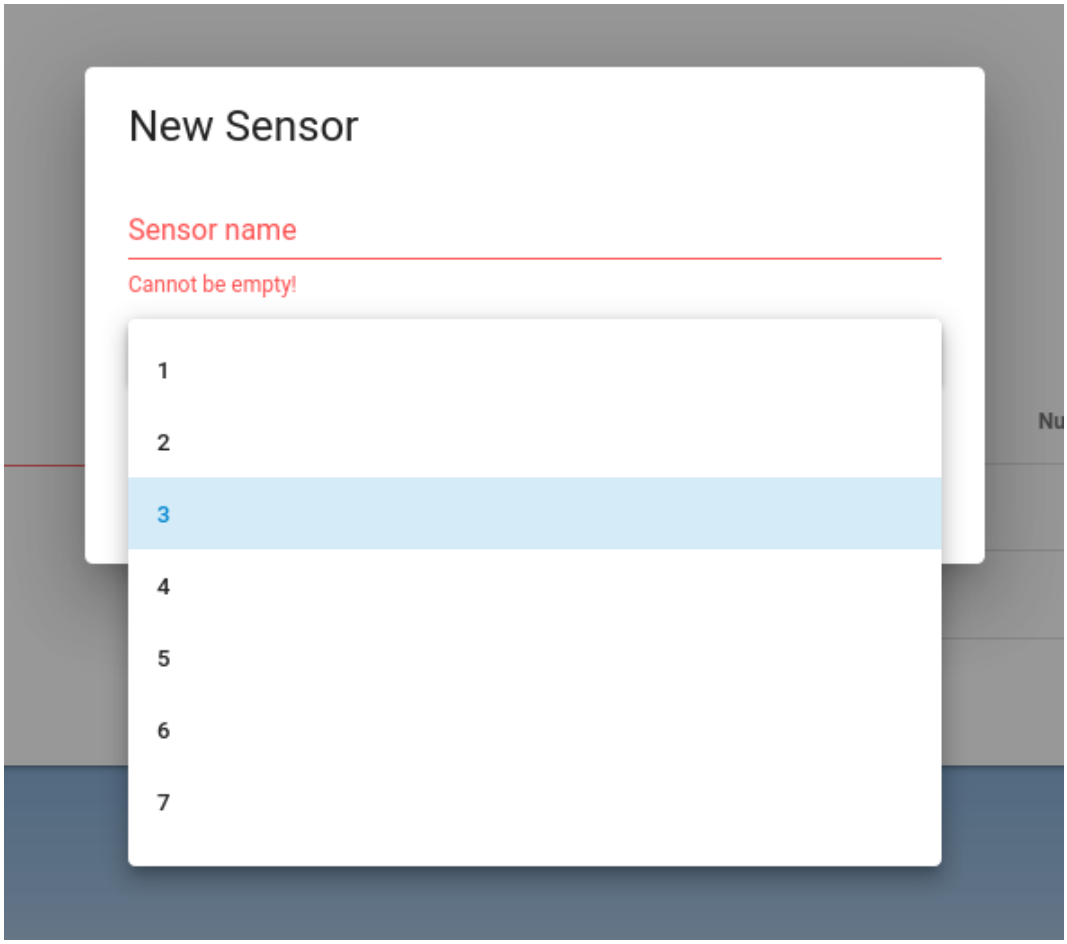


Figure 4.14: Number of lanes selection and empty field validation

Another interesting functionality is the *Edit Item* option. Selecting the pencil icon displays the same dialog when creating a new sensor, but this time with the characteristics associated with the sensor being edited. After saving the edited sensor, it is called an HTTP request to update the database since the sensor already exists. Therefore, it is used the PATCH method. Next to the edit icon is the remove sensors icon. When selecting this icon, a new dialog will appear to confirm the deletion of the sensor from the system. Consequently, the system will no longer integrate the data from the camera represented by the removed sensor. After the sensor deletion is confirmed,

an HTTP request is sent to the back-end with the DELETE method to remove the sensor from the database. The dialogue that asks to confirm the deletion of the sensor is shown in figure 4.15.

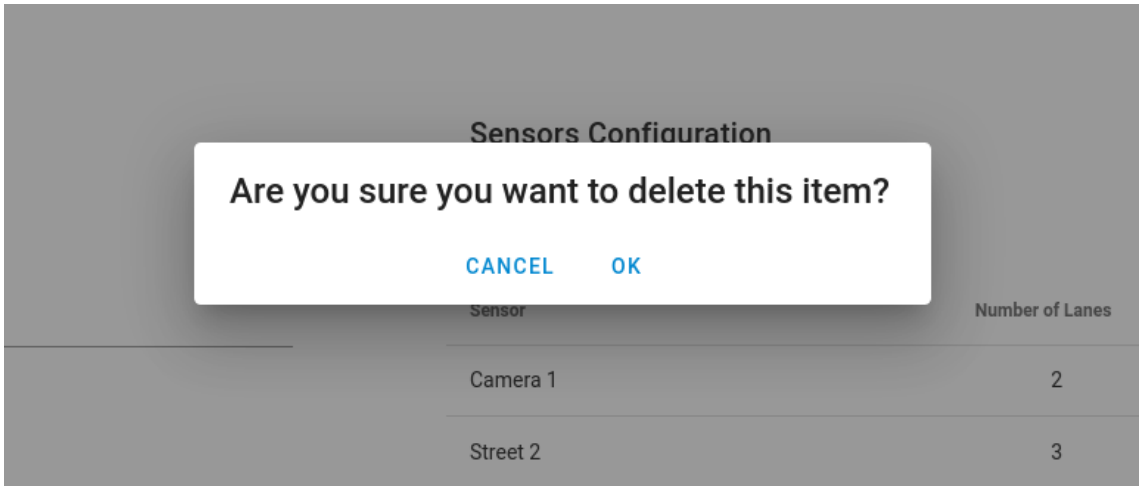
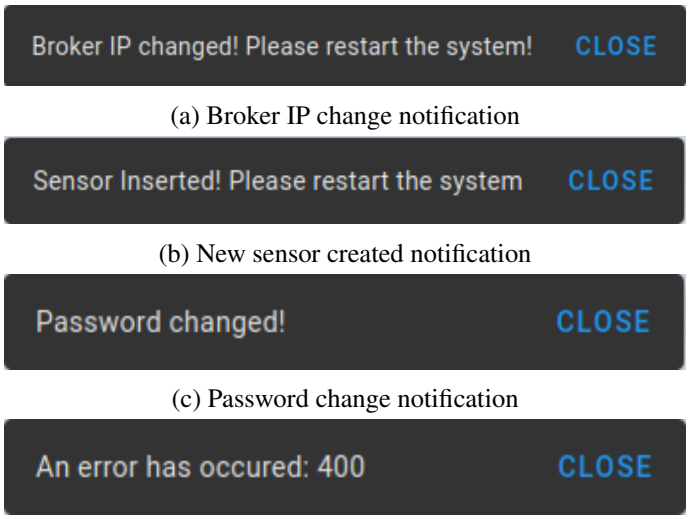


Figure 4.15: Confirmation of sensor deletion dialog

At the end of the table, there is the possibility to choose how many rows are shown per page and navigate through the pages. This is an automatic implementation in the table component. After adding, editing, or removing sensors in the web interface, it is required to restart the system. In the MQTT back-end system, the sensors stored in the database are read only once. The sensors management is rarely changed. This way, it does not make sense to read the sensors in the database in a loop. This is an optimization measure that does not have a relevant negative impact on the system's performance. Furthermore, restarting the module prevents internal errors from happening in the execution of the algorithms.

The last feature implemented on this page is a notification system. If there is any change related to password, MQTT Broker IP or sensors, a notification with appropriate information will appear at the bottom of the page. Additionally, if anything is wrong with backend-frontend communication, the notifications are presented with an error message, including the HTTP status code, and the changes are not applied. This way, the user can understand where does the error come from. This notification was implemented with *snackbar* component, and four examples are shown at figure 4.16.



(a) Broker IP change notification

(b) New sensor created notification

(c) Password change notification

(d) Error notification

Figure 4.16: System change notifications

Finally, there is the page that will allow the configuration of Processes. This functionality is one of the most important in this project. Through the configuration of those processes, traffic-related information associated with each process will be communicated to the traffic controller. Figure 4.17 represents the general aspect of the Processes page. Although not visible in the image, the menu to navigate the web interface is also available since the appearance is dynamic. It is also important to mention that whenever the page is updated, a function is called to receive the stored processes in the database through the GET method. Within the same function, the received data is converted, mainly at the date-time level. This is where the *Day.js* library comes in, which will convert the received date, initially in ISO 8601 format, in YYYY-MM-DD format, and the hour in HH:mm format. This function is executed in `mounted()` stage, i.e., called after the component has been mounted. More specifically, when Vue.js is building a web page, it has a life-cycle with several stages. In this case, it is appropriate to receive the information from the database when the components are ready to show them.

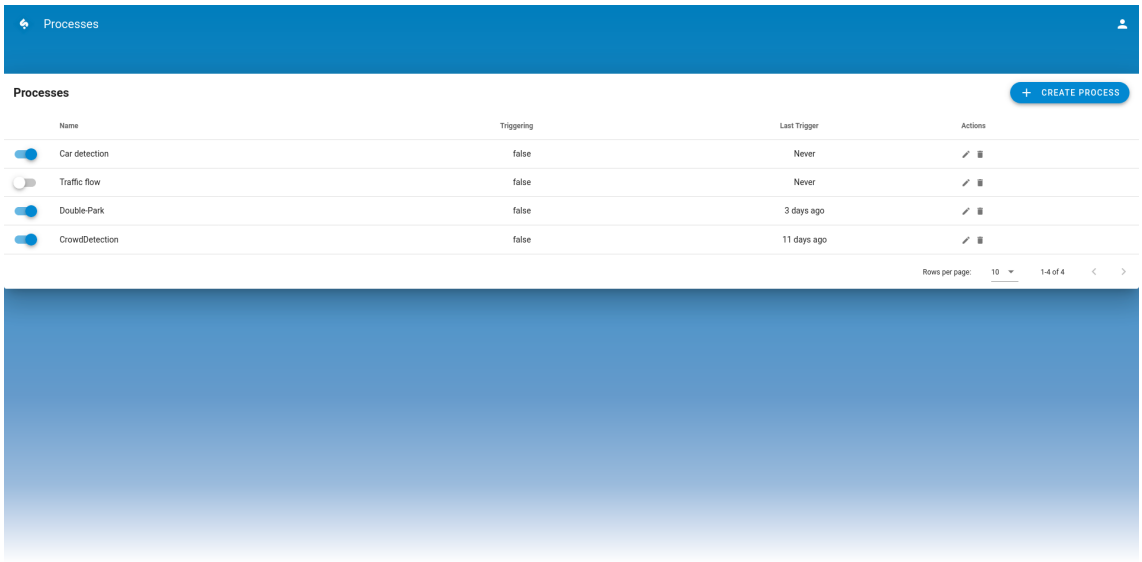


Figure 4.17: Processes Page

The configured processes are presented in a table in figure 4.17. Here is associated with each process its name, status - enabled or not - the triggering state, the last time the process triggered, and a few options to edit or delete the configured processes. Once again, the Vuetify resources helped in the development of this page. First, the Table component was used, which has features such as selecting the number of processes to show or even sorting their items. Then, the *switch* button that indicates the status of the respective process is also a component already developed that makes sense to be implemented on this page. The triggering column constantly indicates whether a process is triggering, i.e., whether its logical condition is True. So, this information is contained in a key that indicates the process status - *true* or *false* and whenever the front-end receives the JSON message with the process-related information. This will parse that same key and put the information in the table. Likewise, the Last trigger column shows how long ago the process fired for the last time. The Day.js library is used again since the value associated with the key that contains this information is in ISO 8601 format. This way, it would not be easy for the user to get information from a date-time in the format: 2022-04-31T00:00:00.000Z, for example. Thus, through the function `dayjs(time).fromNow()`, this value is converted into the time difference between the current time and date and the date-time associated with the last time the process fired. The result is presented in the table as figure 4.18 illustrates. If the process is never initiated, it makes no sense to execute the Day.js library function, and so the information in the table is replaced with *Never*.

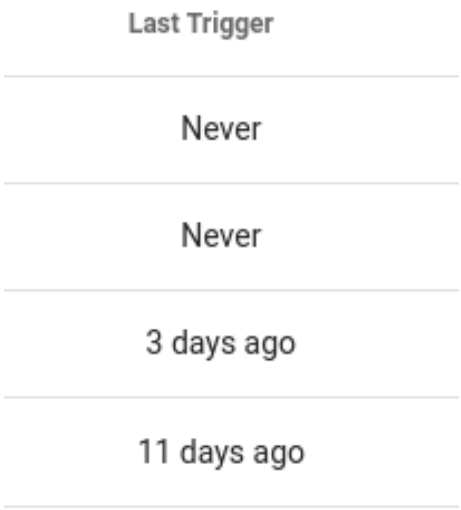


Figure 4.18: Last Trigger column information

Similar to the interaction with the sensors made in the settings page, there are some dynamics to assist in the creation, editing, and removal of processes in this table. Thus, when a process is deleted, a confirmation message will appear in the middle of the page, as shown at figure 4.19. This confirmation is made through the *dialog* component with the *Cancel* button and the *Ok* button. When confirmed, an HTTP request will be made with the DELETE method to remove the process from the database adding the process id to the request, so the back-end knows which process to remove.

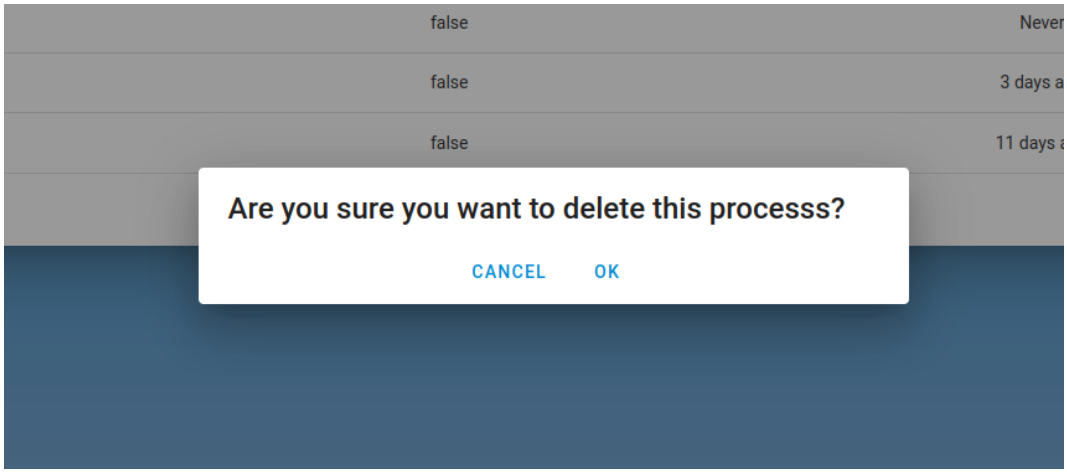
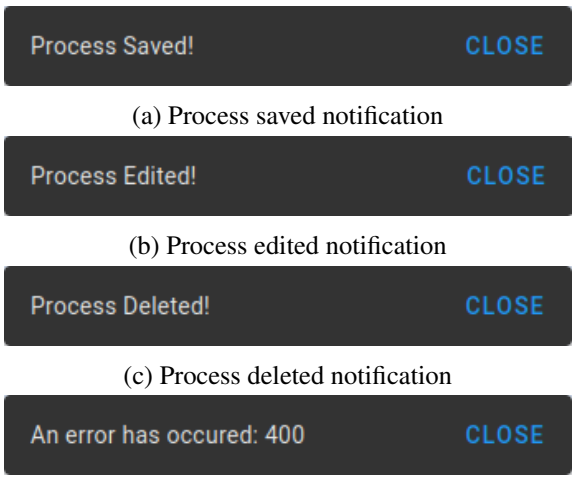


Figure 4.19: Delete process confirmation message

Since it is possible to configure processes with logical conditions that are constantly being checked, a notification should appear indicating that this process has triggered as soon as a process fires. For that, a loop was implemented where every three seconds makes an HTTP request with the GET method to obtain all the processes stored in the database. This way, each message has an associated object indicating whether or not the notification should be activated. The implemented algorithm would be more efficient if the back-end could signal directly that the process fired. However, the front-end can only initiate any communication between the web interface and the data system. This way, it is necessary to have a function that constantly checks if any process has been fired. For the back-end to be able to initiate communication, an event-driven bidirectional socket for real-time web applications would have to be implemented. However, if a process fires, the maximum time it would take for the notification to appear would be approximately three seconds - worst case. Nevertheless, that trigger would still be instantly communicated to the traffic controller - notifications do not interfere with communication between the module and external systems. It is considered that it does not make sense to implement a dedicated server that would support the execution of a bidirectional socket when there are no time constraints between a process trigger and the user becoming aware of that event. This page also has a notifications system at two different levels. The first one is similar to the settings page, where the event of creating, editing, and removing processes is notified with the appropriate information displayed. Also, an error message appears if there is an error related to the communication with the back-end. A few implemented examples are shown in figure 4.20.



(a) Process saved notification

(b) Process edited notification

(c) Process deleted notification

(d) Error notification

Figure 4.20: System change notifications

The other one is dedicated only to the process trigger. During the loop, several processes can trigger. Therefore, when the system receives all the processes, it will have to generate one notification per trigger, each dedicated to one process. To overcome this challenge, it is used *Vuex*. This is a state management pattern + library for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated

predictably. This way, an array of notifications with an associated state was created. This algorithm is interesting since this array is empty, and as notifications are activated, this array will store temporarily. Thus, it is possible to have many notifications at the same time. Otherwise, only one could be displayed at a time. The style associated with these notifications is different from the previous ones so that these can stand out. In figure 4.21, those notifications are illustrated.

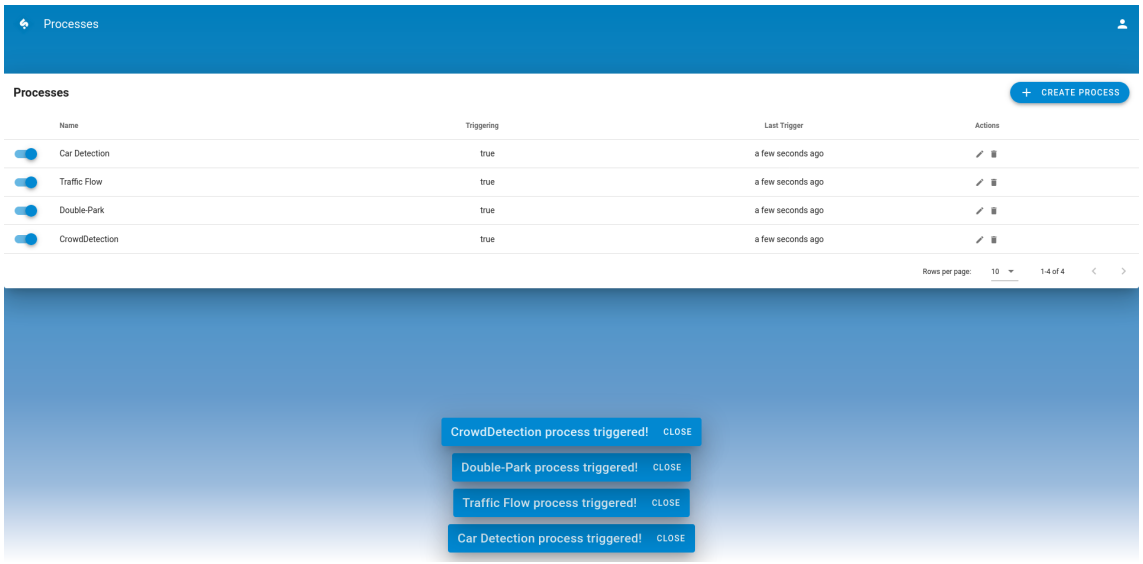


Figure 4.21: Processes trigger notifications

The action of editing and creating a new process via the button in the top right corner of the Process page will be explained in the next section. The system implemented to create/edit processes is relatively complex. Therefore, it is located at a section to expose how this was developed.

4.2.1.2 Web Interface Process Configuration

The functionalities for creating or editing a process are interlinked. The difference is whether the process already exists or not. This way, both will point to the same interface. Again, through the *dialog* component, this interface is rendered, whose goal is to configure a process. So when creating a new process, an interface will appear as shown in figure 4.22.

Figure 4.22: Process configuration interface

Two fields must be edited to create the process. The first one is a column that allows associating several essential pieces of information with the process. The other one is to create the process rule. In that column is necessary to assign the name of the process. The component used for this field is the *text-field* associated with the card component used to support the entire column. As on the dashboard page, it is possible to set the interval at which this process will be active in the *Time* field. The tools used are the same, and as such, when defining the *data range*, the time set in the *start time* corresponds to the first day of the interval, and the time set in the *end time* corresponds to the last one. Since the *date-time* information is stored in ISO 8601 format, it is necessary to transform these times into this same format. Therefore, the Day.js library is used to provide functions to perform this conversion. In the enable field, a top layer is added that allows defining whether the process is enabled or not. Thus, even if the time set for its activation is within the current time, it is possible to deactivate the process if the enable button is off. The state associated with the switch type button is stored and shared with the button in the table associated with each process. Since they both represent the state of the process they are associated with, they share the activation state. To complete the process setup, it is needed to define the most crucial feature - the logical condition. In chapter 3, it was explained that the system's response to deal with process logic conditions would be implemented with *JsonLogic*. To work with this type of rule are necessary two features:

- **Store the logical condition with an efficient syntax** - This syntax must deal with different rules because they can be very complex and have many different operations. Therefore it is necessary to avoid redundant and ambiguous condition characterization.

- **Verify the condition logical value** - It is necessary to execute the rule, read the syntax, and verify whether the condition is true or false.

Therefore, it is necessary to guarantee that the front-end will export the configured rule to JsonLogic syntax. The process of configuring the logic condition can be done in different ways. Given that the goal is to create traffic circumstances events with several possibilities associated, a graphic configuration seems an appropriate and practical option. Besides, this interface has become more user-friendly, improving his experience. Therefore, it was decided that the process rule configuration would be based on blocks and connections between them. Implementing a system like this one can be truly difficult and time-consuming. Thus, after searching for node graph editor frameworks, three valid options were found:

- *Litegraph.js* [64]
- *NodeGraphProcessor* [65]
- *Rete.js* [66]

All of those frameworks seemed to correspond to the needs. NodeGraphProcessor and Litegraph presented some additional features to customize the appearance of the nodes and their connections. However, these frameworks are not entirely prepared to be integrated with Vue.js. Therefore after successive attempts, it was impossible to compatible LiteGraph.js or NodeGraphProcessor with this web interface, which was the first and second option, respectively. On the other hand, the integration of Rete.js with Vue.js was successfully achieved. In general, a node editor framework is based on several blocks and their connections. In some frameworks, it is possible to assign different functions to each block, for example, an arithmetic operation. At the end of the linked chain, the software can execute those functions and produce a result. Although rete.js has this functionality built-in, only its aesthetic component matters to create the rule in this case. Initially, rete.js presents some standard nodes, allowing to make links between them already. Figure 4.23 shows not only these nodes but also the details about the components of each node.

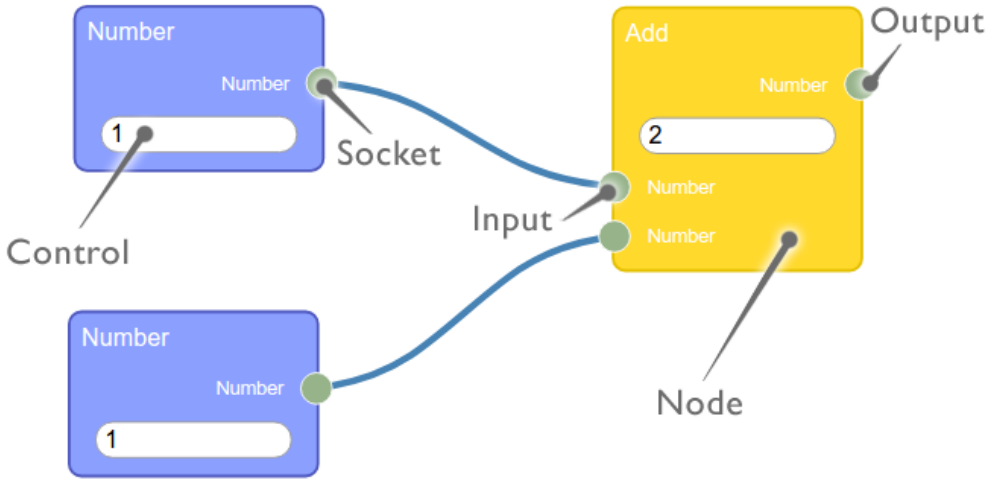


Figure 4.23: Rete.js basic nodes and components details

The development of the nodes will have to be worked on by creating the necessary blocks to give the user the freedom to configure the process rule. The implementation of nodes should consider what type of rules the user wants to create and the associated variables. It was suggested by *SOLTRÁFEGO, SA* that it should be possible to configure rules associated with the variables that are related with received data - *Jam Detection*, *Traffic Flow*, *Crowd Detection*, *Double Park Detection*, *Vehicle Stay Time*, *Vehicle Detection*, and *Vehicle Number*. In addition, selecting the zone and/or the vehicle type associated with the variable should be possible. To create logical conditions, it is still necessary to integrate arithmetic and logical operators. All of the implemented nodes are presented at table 4.1.

Table 4.1: Implemented nodes and the respective group

Variables	**Arithmetic Operators**	**Logical Operators**
Constant	$+$	*AND*
Duration	$-$	*OR*
Crowd Detection	$\times$	*NOT*
Vehicle Detection	$\div$	$=$
Vehicle Stay Time	*Max*	$\neq$
Traffic Flow	*Min*	$>$
Jam Detection		$<$
Double Park Detection		$\geq$
Vehicle Number		$\leq$
GPIO		

Regarding the group of variables, two generic modules have been implemented. The *constant* allows introducing a number that can represent the number of vehicles to be compared in a logical condition and the *duration* represents a time interval in seconds. The other nodes are directly related to the traffic data that the camera will generate. Therefore, inside each node, it is possible to select the zone and the vehicle type associated with the variable. In some nodes, it does not make

sense to define those features, so functionality will not be available. To have a better perspective on the blocks developed, figure 4.24 shows all the nodes that correspond to the group of variables.

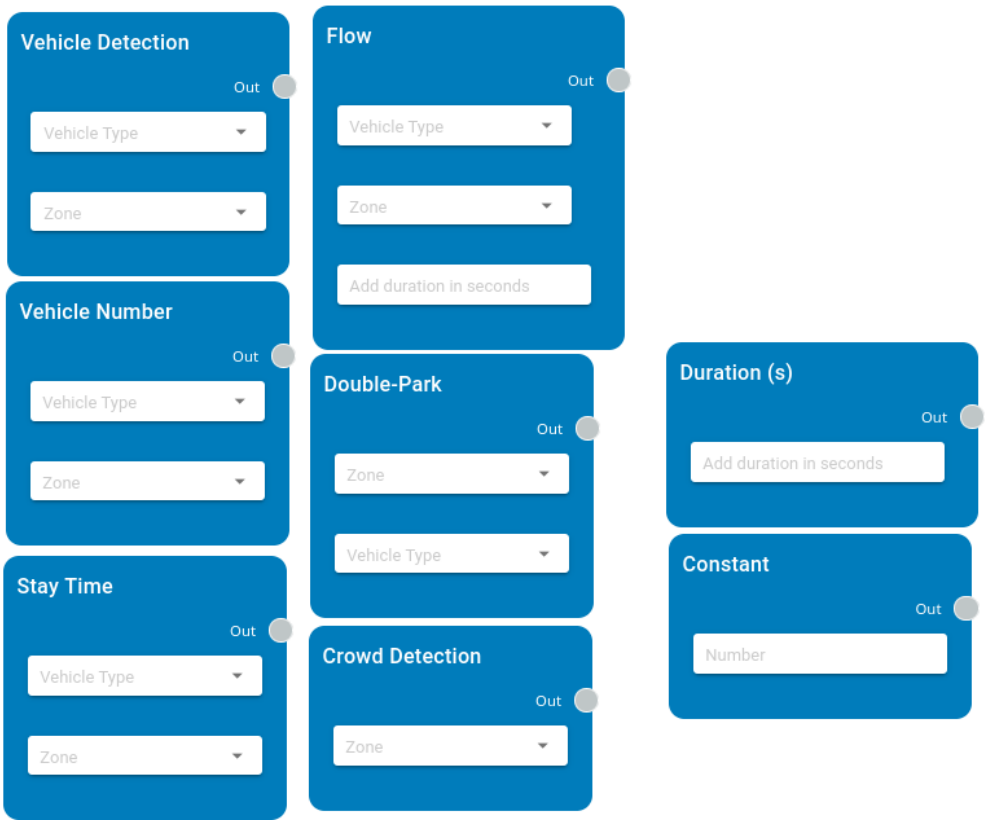
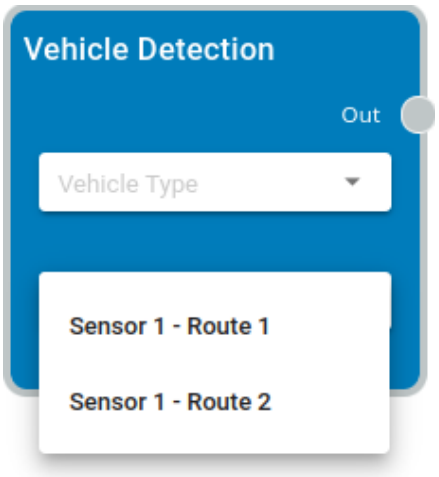


Figure 4.24: Rete.js nodes of variables

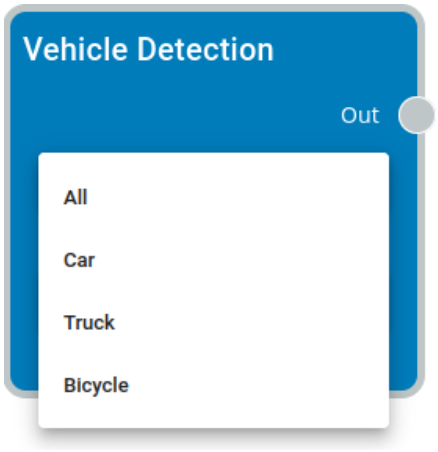
Although the GPIO node is part of the group of variables, this block is different. First, it will be from it that the pins associated with the process will be chosen. Then, it allows for deciding whether a pin that has been selected will have inverted logic associated with it or not. This inverted logic assigns the pin the opposite state of the logical condition. Thus, if the rule associated with the process is True, the logic value of the pin with inverted logic will be 0; otherwise, it will be 1. On the other hand, this node will close the blockchain. So, after creating the logic condition, all that remains is to connect to one or more GPIO blocks if it is desirable to associate more than one pin. Figure 4.25 illustrates the appearance of this block.

Figure 4.25: Rete.js GPIO node

Through the *control* component, the functionality of choosing the type of vehicle and the zone was implemented. So inside that same component was integrated another component - *select* - provided by *vuetify*. This allows the user to indicate without fail which zone or type of vehicle is being referred to. In addition, several problems related to redundancy will be avoided. The options available when choosing the zone will be dynamic as this depends on the number of cameras connected to the module. On the other hand, it is unnecessary to make the type of vehicles dynamic since the camera can only detect cars, trucks, and bicycles. Figure 4.26 shows the options for defining the zone and the type of vehicle through control components associated with the variable.



(a) Zone options



(b) Vehicle-type options

Figure 4.26: Control components

To facilitate the interaction, the user can remove the selected option by clicking in the cross icon, as shown in figure 4.27.

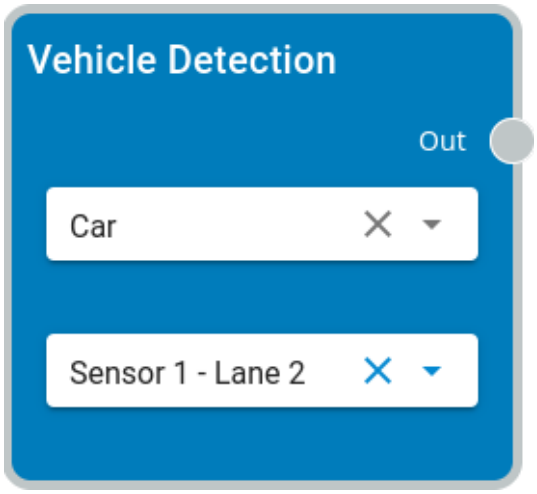


Figure 4.27: Rete.js nodes of variables

Similar to the previous nodes, the GPIO node has two Vuetify components implemented - *select* and *switch*. The *select*, component allows to select the pin, and the *switch*, allows to choose the logic associated with that pin. Figure 4.28 illustrates the appearance of these functionalities.

(a) Rete.js GPIO node details

(b) Rete.js GPIO node selection

Figure 4.28: Rete.js GPIO node options

The other controls that were implemented are much simpler. Thus, the *text-field* component of *vuetify* was integrated since only an integer needs to be entered. Their implementation was not so complex regarding the operator groups, both the logical and the arithmetic ones. Firstly because they have no associated characteristics, each node represents an operation that will have one or more inputs depending on the operator it represents. Thus, its output can be just a number in the case of arithmetic operators or a boolean in the case of logical operators. The implemented operator blocks are shown in figure 4.29.

(a) Logical operator Nodes (b) Arithmetic operator Nodes

Figure 4.29: Operator nodes

As mentioned earlier, some of the nodes only accept two inputs. This happens in those where the *inputs* are identified as A and B. This option is mandatory because, for the operations where this was implemented, it does not make sense that, for example, the operator $=$ receives three inputs. After all, the goal is not to test if $A = B = C...$, but only if $A = B$. On the other hand, for operators such as *AND* or $+$, it does make sense to test if A *AND* B *AND* C is True or calculate $A+B+C...$. These operators have their input named as *Inputs* to facilitate interaction between the

user and the process configuration. To ensure that no illogical rules are created, a simple system has been developed at the socket level of each node. Four socket types were created: number, boolean, constant, and vehicle. This way, the inputs and outputs of each node are associated with a socket type, allowing the data that it receives and results from the operation to be compatible with its socket. This only works because it is possible to assign different socket types to the inputs and outputs of the same node. To demonstrate the example of the configuration of a minimally complex rule, take the following example:

- If *Number of cars* in *Sensor 1 - Lane 1* $>$ 30 *AND* there is a *Jam* in *Sensor 1 - Lane 2* -> activate *GPIO 4*

Therefore, the node's configuration that corresponds to that rule is demonstrated in figure 4.30.

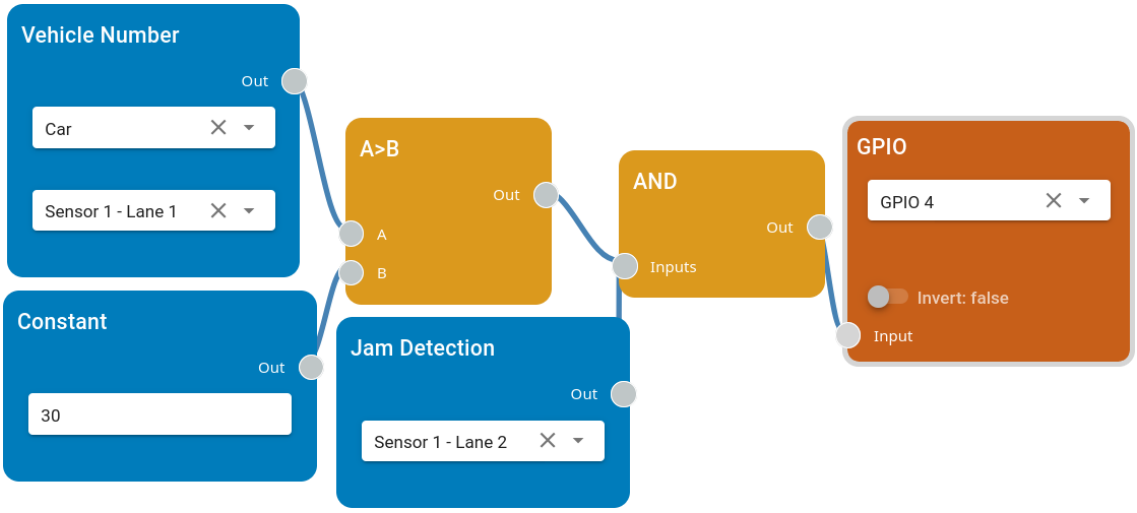


Figure 4.30: Nodes configuration of process rule

After finishing the process configuration, it is necessary to validate it. The system will only analyze on the column side if all the fields are completed. The Vuetify rules system performs this analysis. This way, in all fields except 'Select date range', the Vuetify rules system checks if the fields are empty or not. In the 'Select date range' field, the verification goes through a *RegEx*, checking if the field has two dates. This is necessary because the Vuetify date picker allows to select only one date, and in this case, it is critical to have two. The RegEx used was:

```
/^\d{4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])~
\d{4}-(0?[1-9]|1[012])-(0?[1-9]|[12][0-9]|3[01])$/
```

If the verification is unsuccessful, the page appearance is illustrated in figure 4.31.

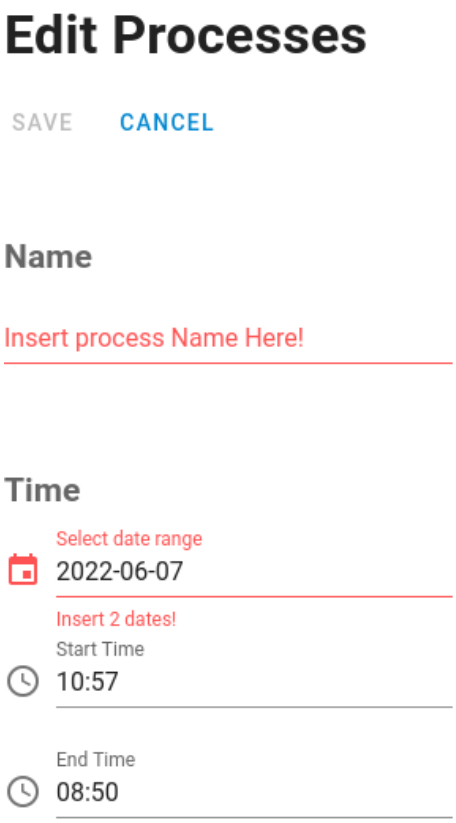


Figure 4.31: Invalid column fields validation

The *VALIDATE RULE* button is available to click only if this analysis succeeds. Otherwise, it will be impossible to check the right side's logic condition and save it. Like the button for validating the rule, the SAVE button is only available once the logical condition is validated. On the side of the logical condition, the system will verify if at least one GPIO block is present in the interface and if all the blocks have their sockets connected to other blocks. This ensures that the node chain is closed with the essential blocks. Additionally, by limiting the connections between sockets of different types, it is ensured that it will not be possible to configure rules that jeopardize the system execution. However, there is one more way to bypass these verification rules: it is possible to create two individual blockchains. To detect this situation, a function has been implemented that checks whether the GPIO nodes are connected to the same node. Thus, it will not be possible to implement two independent logic conditions in the configuration of the same process. As soon as this rule system finds no errors, the *SAVE* button becomes enabled, and the process can be saved. Then, it is possible to save the configuration. From that moment on, a series of complex processes related to data binding will be triggered. The following section describes how this system was implemented.

4.2.1.3 Web Interface Data Binding

After saving the configuration, it will be necessary to send the data to the back-end for storing it in the database. It is necessary to build a message in JSON format to be sent with all the details related to the process. The keys present in this message are:

- *name* - Name of the process
- *startTime* - Day and hour that process will start being active
- *endTime* - Day and hour that process will stop being active
- *enable* - Top layer to activate process
- *gpios* - All gpios associated to the automation and respective Invert variable state
- *rules* - Logic condition under JsonLogic syntax
- *blueprint* - Raw export of the configuration of rete.js nodes

Furthermore, three keys also belong to the message:

- *triggering* - Indicates if process is currently triggering
- *lastTimeTrigger* - Indicates the last time the process triggered
- *notification* - Indicates if it a notification of a triggered process must be shown

The front-end never changes those keys. They indicate information so the user can be aware of the process's current status.

Both the *name* and the *enable* are trivial to select, and it is only necessary to fetch the data associated with the variables directly. Again, the Day.js library is used as all data related to the date and time is stored in ISO 8601 format. In this case, it is necessary to transform the set dates and times into a Start Time and End Time, each with an associated day and time. For the *gpios*, the information will be obtained through a function implemented in each GPIO node that returns the selected pin and the respective state of the *Invert* variable, as demonstrated in the following example:

```
[{ "gpio":1, "invert":true } , { "gpio":2, "invert":false },...]
```

This way, an array will have to be built in which each object contains two keys - the Gpio pin and the respective logic. For the *blueprint*, this represents the exported code that describes the nodes' configuration, characteristics, and connections. This detail is vital to be stored because there is the option to edit the process. Thus, if the user wants to edit a process after configuring it, the graphical interface with the nodes, which represents the logical condition of the process, will have to appear

as it was previously defined. Finally, the rules key will describe the logical condition associated with the process. This description will have to be defined under the JsonLogic syntax, and this message will be stored directly in the database. This way, when the back-end verifies the logical condition, it will only be necessary to collect the value of the rules key without transforming the data again. In this way, one of the most considerable challenges in web interface development was precisely to obtain, through the graphical configuration of the nodes, their representation under the JsonLogic syntax. Hence, it is necessary to understand how this syntax works.

First, there are two different properties - variables and operators. Each of these properties has arguments. If the rule only implies operators and the variables are already defined, for example - *IF ((3 > 1) AND (1 < 3))* - the representation of this condition with JsonLogic syntax would be as follows:

```
rules = {"and" : [
  { ">" : [3,1] },
  { "<" : [1,3] }
]}
```

Otherwise, if the rule implies variables that are not defined yet, for example - *IF ((StayTime > 110) AND (Jam))* - the representation of this condition with JsonLogic syntax would be as follows:

```
rules = {"and" : [
    {">" : [ { "var" : "StayTime" }, 110 ]},
    {"==" : [ { "var" : "Jam" }, True ] }
]}
```

In this case, it is necessary to supply the function that will check the state of the condition with the value of the variables in a *data* message also in JSON format. This would have to be done in the back-end system when this verification happens. In short, a variable and an operator will have associated arguments that can be values, variables, or other functions. This reveals that this syntax works recursively. After analyzing how JsonLogic works, it is possible to implement an algorithm that allows the configuration of the logical condition to be exported in the proper format. Then, a function was developed for each node that allows that same block to export every bit of the rule. That is, for blocks with inputs, their respective links are checked. Thus the following code is exported:

```
    {"operator" : [ "input1", "input2", ... ]}
```

For nodes that only contain two fixed inputs, the analysis is simpler, as it is enough to check what is on each input. For those that may have multiple inputs, a loop is created that, as it analyses each input inserts it into the exported message of the node. On the other hand, the nodes *constant* and *duration* work as variables with a defined value. So they only need to export the respective

value entered in the control field.

Only the variable-type nodes are missing. Although at first glance, it makes sense to consider them as variables in JsonLogic syntax, it was decided that these nodes will be operators just like the rest of the nodes. This interesting implementation will impact the back-end development in the logical condition analysis phase. As such, that process will be explained later. For now, the configuration of these nodes will be exported as the same as for the other nodes. So instead of checking what data is linked to the inputs, which in this case do not exist, those data that can correspond to *zone*, *vehicle type* or *duration* are present in the controls integrated within each block - as well as in the *constant* and *duration* blocks. Finally, it is necessary to bundle each message block exported by each node in the same way JsonLogic requires it. It can be seen that this message can be constructed this way if this process starts from the last block. Thus, the arguments of the previous function will always be the functions that follow. By inverting the sequence, it is possible to build the JSON message that contains the logical condition associated with the process. This way, the GPIO block will analyze which operators are connected to its input and start the process of grouping the exported blocks from each node. Since it is possible to add several GPIO blocks, the message construction will be initialized by only one node. Once the message is complete and validated, the web interface will make an HTTP request where it can use two different methods. If the process has been created before, it should be updated in the database. Thus, the most appropriate method is PATCH. The POST method will be used if the process is set up for the first time. These two methods will tell the back-end whether to update the database (PATCH) or introduce a new process (POST).

4.2.2 Back-end

The back-end system will be implemented in the *python* language. This high-level language allows the system to be implemented from scripts running in parallel. In addition, some of its libraries allow this to be done automatically, thus avoiding the need to implement threads. Therefore, the code was developed in four major blocks:

- REST API - Implemented with *Flask*, runs on its own loop. It is responsible for front-end communication and storing information generated on the web interface - the configured processes. It also provides the data stored in the database as the front-end requests it.

- Database - JSON files that have stored data in JSON format. It follows the non-relational database structure and is implemented with *TinyDB*.

- MQTT system - A library with several functions for deserializing JSON messages was implemented, as well as an MQTT client with a subscriber function to integrate the data sent by the CCTV camera.

- Process trigger - It analyzes the logical conditions associated with the configured processes and, depending on the result, and pin configuration physically maps the digital pins of the module and assigns them the logical value of 1 or 0.

Thus, the software will only listen to two cycles in parallel. One is dedicated to the execution of the REST API, and the other is responsible for data integration through the MQTT system and the analysis of the configured processes. Thus the structure of the implemented code is shown in figure 4.32.

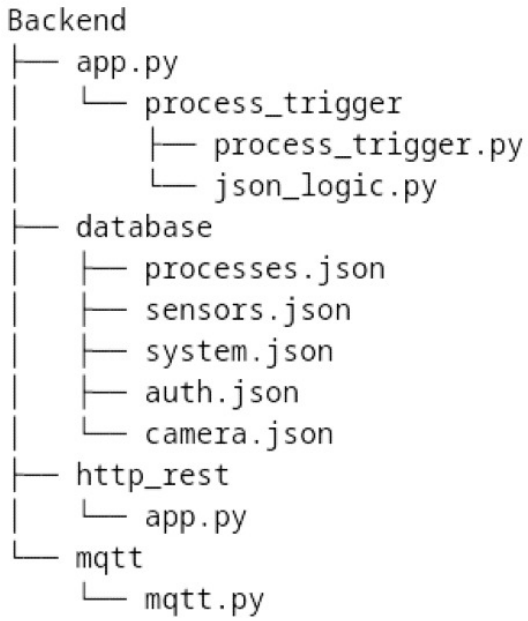


Figure 4.32: Back-end code structure

A feature that had to be implemented for the whole back-end to work correctly was a Virtual Environment (venv). Some software and tools are being used on this back-end and their versions and dependencies often conflict. Since the installation system - *pacman* - is limited to installing the most current version of each software, the back-end system ends up not even running. This way, a virtual environment is the best solution to manage the dependencies of this solution, both in development and production. The implementation of *venv* is quite simple since python allows to create it directly with the following command line:

```
[user@user ~]$ python3 -m venv venv
```

Then it is necessary to activate it:

```
[user@user ~]$ . venv/bin/activate
```

Finally, it has an associated list that includes all tools and software and the corresponding version prepared to avoid any errors related to dependencies or incompatibilities. That list contains the following software:

```
datetime==3.10.5
Flask==2.1.1
Flask-Cors==3.0.10
Flask-JWT-Extended==4.4.1
filelock==3.7.1
more-itertools==8.13.0
paho-mqtt==1.6.1
python-dateutil=2.8.2
six==1.16.0
tinydb==4.7.0
Werkzeug==2.1.1
```

Each implemented file is inside the folder whose name refers to the function performed. This makes it easier to understand the code. Finally, the files that will have to be executed are: `\http_rest\app.py` - REST API - and `app.py` - data integration and process trigger. Thus, the following sections will describe how each subsystem works and how each one was implemented.

4.2.2.1 Database System

The database system is quite simple, being implemented with the TinyDB software. This software creates a .json file to create a database where it stores messages in JSON format. It is necessary to store much different information, such as the data generated by the CCTV cameras, the processes configured in the web interface, and other internal system information. As such, five different files were created, each dedicated to storing different types of data:

- *auth.json* - It will store the access credentials for the web interface. Since there is only the administrator role, each stored document - so called by TinyDB - will have the username and password as objects.
- *camera.json* - All kinds of data sent by the CCTV cameras will be stored. Each document will have enough information to distinguish it from the others. This way, it is only necessary to enter them directly because TinyDB assigns each one an id. This will undoubtedly be the file with the most data entered.
- *processes.json* - All processes configured in the web interface will be stored. Just as in camera.json, it can store the documents directly in the database, and TinyDB is responsible for assigning an id to each one. Logically, the number of data inserted will not exceed hundreds or even tens of documents. However, this system allows a much larger volume of data.

- *sensors.json* - All sensors configured in the web interface and their respective number of lanes will be stored. It is possible to store the documents directly in the database, and TinyDB is responsible for assigning an id to each one. A maximum number of sensors to be stored in the system was not defined.
- *system.json* - Here, some useful information will be stored while running the back-end and front-end systems. The IP of the broker operating in the MQTT communication between the cameras and the module and a few temporary pieces of information will be stored.

It is expected that the software will not have failures associated with the high volume of stored information. Delays could arise during the file search process if the files reach vast amounts of documents - the name TinyDB assigns to each inserted JSON message. During implementation, some simple tests were made to ensure that the module could work over a long period. This period varies depending on the flow of sent and stored data. Therefore, only a proper load test can determine what the maximum limit of the system can support. All the algorithms that will execute within the back-end will use various methods implemented by TinyDB as shown in [67]. These methods allow for the database's manipulation in various ways. Some of the methods that will be most used are:

- `db.search(query)`
- `db.search(where(key)==value)`
- `db.insert(json_message)`
- `db.update(json_message)`
- `db.upsert(Document(json_message)`
- `db.get(query)`
- `db.contains(query)`
- `db.all`
- `doc.doc_id`
- `db.remove(doc_ids=[i, ...])`

Thus, a local storage system that supports the module's needs is implemented, working simply and effectively, providing a series of tools for its use.

4.2.2.2 REST API

The entire REST API implementation was done through the *Flask* software, both in terms of structure and use of the functions provided. This script has two primary functions. The first one is a token generator. This token is provided to front-end after a successful login by the *JSON Web Tokens* technology. This way, the authenticated user is allowed to enter on web interface while the token is available. The `login()` function gets the credentials sent from front-end - *username* and *password*. Then, it searches in database a document with that same username. After that, it will compare the respective password. However, this is not done directly, i.e. there is an encryption system. In this way, the stored password is encrypted. For this purpose the library werkzeug.security was imported, which allows not only encrypting a password, but also comparing two encrypted passwords. Thus, the verification is done through the function

`check_password_hash(system_password, login_password)`. This system was implemented for security reasons, so it is never possible to get the password since it it hashed. It was mentioned in 4.2.1.1 that whenever access to a page was requested, the front-end first made sure that the saved token was valid and matched the user. Thus, the following functions were implemented:

```
@app.route('/verify-token', methods=['POST'])
@jwt_required()
def verify_token():
    return jsonify({'success': True}), 200


@app.route('/protected', methods=['GET'])
@jwt_required()
def protected():
    current_user = get_jwt_identity()
    return jsonify(logged_in_as=current_user), 200
```

In this way, the back-end ensures that the registered token corresponds to the authenticated user. Furthermore, all the functions that will be presented below have the `@jwt_required()` function included. This function does not allow the functions implemented in the back-end system to execute whenever they are called. Verifying that the token associated with the user is valid is necessary. Therefore, the back-end system is fully protected. The other is to serve the web interface in various tasks. For each task, a function is implemented. The way that *Flask* works to correspond to HTTP requests is based in building a function inside the `route()` decorator to bind the function to a URL, for example: Most HTTP requests from the front-end are related to handling the configured processes. Thus, four functions have been implemented:

```
@app.post('/process')
@jwt_required()
new_process()

@app.patch('/process')
@jwt_required()
update_process()

@app.get('/process')
@jwt_required()
get_process()

@app.delete('/process')
```

```
@jwt_required()
delete_process()
```

Note that instead of `.route`, it was replaced with the HTTP method best suited to the task. So whenever the front-end creates a new process, on the back-end side, upon receiving the HTTP request with the POST method and data, the `new_process()` function is executed. If editing an already configured process is desirable, the `update_process()` function is called using the PATCH method. The following function is called in a three-second loop by the web interface or whenever it is updated. That function is `get_process()`. Besides sending all the information related to the processes, it will then iteratively go through the database, and in each process, it will set the *notification* key to false. This is necessary because the web interface should display the notification indicating that a process has triggered only in a rising edge of that event. Otherwise, the notification will constantly pop up whenever the process is triggering, which does not make sense. Finally, the DELETE method is used in any process deleted in the web interface, thus calling the `delete_process()` function. The use of the tools provided by TinyDB begins at this stage. These functions will have to interact with the database to insert, search, update or remove data.

As mentioned in 4.2.1.1 section, on the settings page, it is possible to change the password, change MQTT Broker IP and also add or remove sensors and the respective lanes. On the back-end side, it was developed the following functions:

```
@app.patch('/settings')
@jwt_required()
update_password()

@app.get('/settings-broker')
@jwt_required()
get_broker()

@app.patch('/settings-broker')
@jwt_required()
update_broker()

@app.get('/settings-sensors')
@jwt_required()
get_sensors()

@app.patch('/settings-sensors')
@jwt_required()
update_sensors()
```

The `update_password()` function receives the new password and it updates the database with `db.update(query)` function of TinyDB. Before the update, the password must be hashed. This way, using the `generate_password_hash()` function the password is hashed and then stored. The `update_broker()` and `update_sensors()` functions are also responsible for updating broker IP and the sensors associated with system when front-end sets a new the Broker IP and adds or removes a sensor. The `.get` functions are called whenever the settings page is updated. This is necessary because the information of Broker IP and sensors is constantly shown in the web interface. Therefore, this process is simple. When those functions are called, a search in the database is performed to get the desired information. Then, it is sent to the front-end. In addition, it was mentioned that when creating or editing a process, the available digital pins are required to be used. Therefore, the following function has been implemented:

```
@app.get('/pins')
get_pins()
```

This function will go through all the processes stored in the database and analyze each process's Key *gpios*. Then, it will return a list of all unused pins. When the user is assigning a zone to a variable during a logic condition configuration, it is necessary to consult all the available sensors and respective lanes to choose one. Therefore, an HTTP request is performed that calls the following implemented function that searches for the object *Zones* in the system.json database file and returns it to the front-end:

```
@app.get('/settings-zones')
@jwt_required()
get_zones()
```

To know the communication status of the module with the broker, the following function was implemented that searches the database for the communication status:

```
@app.get('/broker-state')
@jwt_required()
get_broker_communication()
```

Finally, it was implemented a function to return the desirable data for the charts:

```
@app.get('/chart')
@jwt_required()
getChartData()
```

This function receives four parameters:

- Sensor - Indicates which camera the information to be sent is associated with
- StartTime - Indicates start of time interval when data was generated

- EndTime - Indicates end of time interval when data was generated
- Indicator - Indicates which of the graphs, the data to send is related to - Count Cars, Count Trucks, Count Bikes, Double-Park Vehicles

Then, the function will return the documents that meet the parameters since they already have the desired format - *object: key*.

4.2.2.3 Data Integration

The data integration system is divided into two blocks. The first block is responsible for behaving as a subscriber and connecting to the MQTT broker to receive the camera-generated data. The other block will be the next stage through which the data will pass. Once the system receives and validates the data, a series of procedures will focus on filtering and manipulating the integrated data. In addition, after this process, the data needs to be saved in the database. To better understand the tasks associated with each block, figure 4.33 shows a diagram with the tasks of each block of the data integration system.

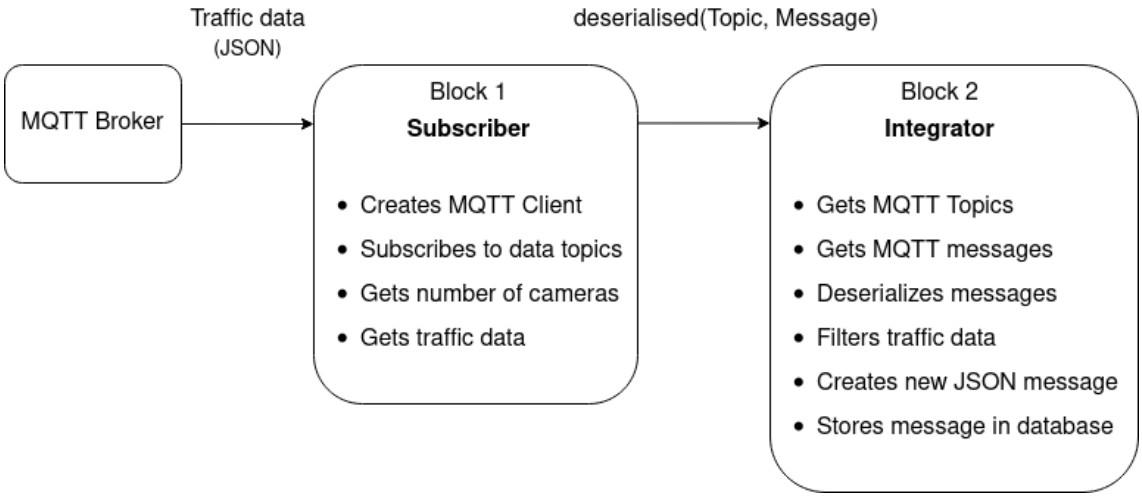


Figure 4.33: Block diagram of data integration system

The camera's data comes from the configuration of tasks that represent certain traffic circumstances. Thus, the data received results from tasks previously defined in the CCTV camera configuration software. It is not part of the project to ensure these tasks are well configured. However, there must be no problems associated with sending data not to compromise the system's testing phase. The intelligent analysis system implemented in the camera allows the configuration of many tasks. However, *SOLTRÁFEGO, SA* suggested several tasks that the module should be able to integrate. The list of tasks consists of:

- Count data and vehicle classifications by lane
- Idle vehicle detection by vehicle type and lane

- Detection of wrongly parked vehicles - Double-Parking - by vehicle-type and lane
- Jam detection by lane
- Pedestrian detection at crosswalks

The camera sends the data because each message has a topic. This topic contains much information, although some not so much. An example of a topic for the task counting cars in lane 1 is:

```
CameraName/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Car 1
```

The *CameraName* is the name assigned to the CCTV camera when the camera is configured in the configuration manager software to connect to an MQTT Broker and start sending data. This name must be the same as the one assigned to the sensor in the web interface. This correspondence is the connection point between the camera and the characteristics that the module will save and also associate with the configured processes. As seen previously, a series of variables must have an associated zone. That zone represents the camera and one of the lanes belonging to the observed road. This identification system allows it to associate any camera with any name, only necessary that the names introduced are precisely the same. Thus, this makes the identification of zones much more unique and easy to identify. Beyond that, the system does not know how many topics it should subscribe to. Hence, the first subtopic represents the camera; if there is more than one, there will be a new branching of topics. At the beginning of the execution, the subscriber reads the database where the sensors are stored and thus knows how many topics and which ones to subscribe to. The other topic component that can be configured is the last one. This represents the name of the task configured on the CCTV camera. Thus, for each task implemented, the name will have to follow a set of rules to integrate this data into the system. Thus, for each task implemented, the name will have to follow a set of rules to integrate this data into the system. That set of rules consists of:

- If the task takes as argument the vehicle type and the zone, the name structure becomes: "*Task Vehicle Lane*". For example, *Counter Car 1*, *DoublePark Truck 2*
- If the task takes as argument the zone, the name structure becomes: "*Task Lane*". For example, Jam 1
- If the task takes no arguments, the name structure becomes: *Task*. For example, CrowdDetection
- All the names and arguments must start with uppercase letter
- If the task has more than one name, all the names must be collapsed into one word and each name must start with uppercase letter

The remaining names associated with the topic are generated internally by the system. This way, with the tasks that have been suggested for implementation, the list of topics that the system could receive is:

```
CameraName/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Truck 1
CameraName/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Car 1
CameraName/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Bike 1
CameraName/onvif-ej/IVA/IdleObject/DoublePark Car 2/&1
CameraName/onvif-ej/IVA/IdleObject/DoublePark Truck 3/&1
CameraName/onvif-ej/IVA/IdleObject/DoublePark Bike 2/&1
CameraName/onvif-ej/IVA/IdleObject/Idle Car 2/&1
CameraName/onvif-ej/IVA/IdleObject/Idle Truck 1/&1
CameraName/onvif-ej/IVA/IdleObject/Idle Bike 2/&1
CameraName/onvif-ej/IVA/ObjectInField/Jam 1/&1
CameraName/onvif-ej/IVA/CrowdDetection/CrowdDetection/&1
```

The only variation allowed by the module in these received topics is the lane number defined in the rule. If any of these topics are incorrectly configured, the system ignores the message and does not make any insertion in the database. To receive all this data, the system must connect to the MQTT Broker. Thus, within the *subscriber* block, the *Paho* library was used, which provides some tools to turn the system, in this case, into an MQTT client. Initially, the client is created in the file. Then the `on_connect(client, userdata, flags, rc)` function is used to execute the `client.subscribe(topics, int)` function. Here a loop has been implemented that executes the function `client.subscribe(topic_camera, int)` for each camera topic. After consulting the database to find out which sensors were associated, the *topics* argument is an array created at the beginning of execution. In this way, the system is guaranteed to receive all data from all connected cameras. Then the function `on_message(client, userdata, msg)` is implemented. Here the topic is deserialized and transformed into an array. At this point, the first data filtering is done, i.e., if the name associated with index 2 is 'RuleEngine' or 'IVA', the data will move to the next level of filtering. Otherwise, the message associated with that topic is discarded. It is explained in [57] that those names are the only ones that carry information related to the tasks configured in the camera. All of the other names correspond to internal camera system information, which is not important to be integrated into the module. This is where the functions of the subscriber block end. Thus, it will provide the next block with all the cameras' topics that correspond to the configured tasks.

In order to understand the complete structure of the topics and their messages, in figure 4.34 is represented several topics that the module may receive and their respective messages.

Figure 4.34: MQTT Topics and associated messages sent by CCTV camera T1

This capture was made through the *MQTT Explorer* software, where an MQTT client was configured that subscribes to the same MQTT broker acting as an intermediary between the camera and the module. According to the analysis of the figure, it can be seen that the data between the messages belonging to the *RuleEngine* subtopic are different from those belonging to the *IVA* subtopic. This difference exists at the level of the type of data associated with the *Data* key. In *RuleEngine*, this object has an Int variable associated - Count. In *IVA*, Data is associated with a Boolean variable - State. As such, the filtering process of the tasks will have to be different. Then, the implemented function parses the topics, and messages will have a division for each subtopic. To store the data regarding the tasks of counting vehicles, the JSON message to build to be stored is:

```
{
    "Cam": CameraName,
    "UtcTime": datetime in 8601 iso format,
    "Task": "Counter",
    "Vehicle": vehcile type,
    "Zone": CameraName-lane,
    "Count": integer
}
```

Only the *UtcTime* and *Count* are obtained from the MQTT message, and the other keys are presented in the MQTT topic. The same happens with the other tasks. Any topics or messages that do not correspond as expected are discarded for the system's security. Otherwise, there could be serious flaws in the module's operation that would compromise the safety of an entire traffic system within a real traffic control system. Thus, if the task name does not comply with the

established rules, the system ignores and discards it. This way, the name is placed in an array where all the indices are compared to the only accepted possibilities. In other words, the vehicle name will have to match one of the three available - Car, Truck, or Bike and the vector size must be exactly 3. It was decided that the best way to represent a zone in the database is the CameraName and the associated lane. After obtaining all the necessary data, the message is constructed and serialized in JSON, and TinyDB's `database_insert()` method is used to insert the message into the database. For the other tasks, the data selection and filtering processes are similar. The JSON messages that will be stored with data from Idle and DoublePark tasks have the following structure:

```
{
    "Cam": CameraName,
    "UtcTime": datetime in 8601 iso format,
    "Task": "Idle Object" or "Double Park",
    "Vehicle": vehcile type,
    "Zone": CameraName-lane,
    "State": "true" or "false"
}
```

Again, it is only the *UtcTime* and the *State* key that comes from the message. The remaining keys result from the information present in the topic. The task name validation system, described for the *Counter* task, is applied in the same way. For the CrowdDetection and Jam task, the checking and filtering process is slightly different. However, before getting the task's name, there is another sub-topic following IVA. So assigning the data filtering process for each task as they are different becomes simple. The system needs to know if the name of the sub-topic is IdleObject, CrowdDetection or ObjectInField. Therefore, for Jam, the size of the vector must be precisely 2 (Task name and lane number), and the name of the task cannot be different than "Jam 1" or Jam 2". The JSON messages that will be stored with data from the Jam task have the following structure:

```
{
    "Cam": CameraName,
    "UtcTime": datetime in 8601 iso format,
    "Task": "Jam Detection",
    "Zone": CameraName-lane,
    "State": "true" or "false"
}
```

Since, in a traffic jam, it does not make sense to associate the vehicle type, this information cannot be part of the message. Finally, the CrowdDetection task name must contain the names Crowd and Detection somewhere. This decision allows some flexibility in configuring the task's name while not compromising the system's security in any way. As there is no more information

to take from the task name, there is no problem in making the filtering system act in this way. The JSON messages that will be stored with data from the Jam task have the following structure:

```
{
    "Cam": CameraName,
    "UtcTime": datetime in 8601 iso format,
    "Task": "Crowd Detection",
    "Zone": CameraName-lane,
    "State": "true" or "false"
}
```

The implementation of the system has been optimized in some aspects. The first is related to the assignment of the filtering method depending on the task at hand. As soon as the system succeeds in assigning a method, it will not continue testing the others; that is, the algorithm moves on to building the message. Then, a loop is implemented that will execute the data integration system. During this loop, the connection to the MQTT broker is constantly checked. If there is any associated problem, it is logged in the database that the connection is down. This information is sent to the back-end to report the module's connection status to the MQTT broker. In this way, the user is immediately aware that there are problems in the system. This implementation serves more the purpose of security, since if the data generated by the camera are not integrated by the system, it will be operating in vain. Next, this data integration loop is executed along the process trigger algorithm. This algorithm will use the database system to analyze the rules associated with the configured processes. Thus, it can verify if the conditions are True or False and trigger the process. The algorithm responsible for this functionality will be explained in the next section.

4.2.2.4 Process Trigger

This algorithm will have the objective of executing the rules associated with the configured processes. Furthermore, suppose the logical condition is True. In that case, it will have to map the digital pins configured in the process and set them to the logical value of 1 or 0 according to the inversion logic assigned to each pin. The following example shows how the message that stores a process is structured:

```
{
"name": "Truck Detection",
"startTime": "2022-06-07T11:00:00.000Z",
"endTime": "2022-06-23T17:30:00.000Z",
"enable": true,
"gpios": [ {"gpio": 7, "invert": true}, {"gpio": 10, "invert": false} ],
"triggering": false,
"lastTimeTrigger": "2022-06-07T11:10:00.000Z",
"notification": false,
"rules": { ">": [{ "vehicleDetection": [ "T1-1", "TRUCK" ]}, "1" ]},
"blueprint": { rete.js_export() }
}
```

The algorithm of this script is presented in figure 4.35. It will start by iteratively going through the file where the processes are stored. Each message will check the state of the enable key and if the defined time interval belongs to the current time. This determines whether the time interval associated with the process execution is within the current time and if the process is enabled. The associated logical condition can be verified when their processes successfully pass this selection phase. The next step will be performed after checking the conditions of each process. Thus, after determining the logical value of the rule, the algorithm will search in the database if this event is a rising edge - process condition was False and now is True - or a falling edge - process condition was True and now is False - through the *triggering* Key value. In a rising edge it is necessary to turn *triggering* and *notification* key to True and update the *lastTimeTrigger* with the current time. In a falling edge, it is necessary to turn *triggering* to False since the process is no longer active. Regarding *notification* key, it will be updated to False by REST API as soon as it delivers processes to the front-end, as explained before. The *lastTimeTrigger* key should stay with the same value since it indicates when was the last time the process logic condition started to be True. Finally, after getting these possibilities, it is time to assign the logic value to process associated GPIOs depending on their logic. In a rising edge, the pins without inverted logic will be physically mapped to the Raspberry Pi's outputs and assigned a logic value of 1. The pins with inverted logic will also be mapped and assigned a logic value of 0. In a falling edge, repeat the process in reverse. That is, the pins without inverted logic will be assigned a logic value of 0, and the remaining pins will be assigned a logic value of 1 since that is the purpose of the inverted logic functionality.

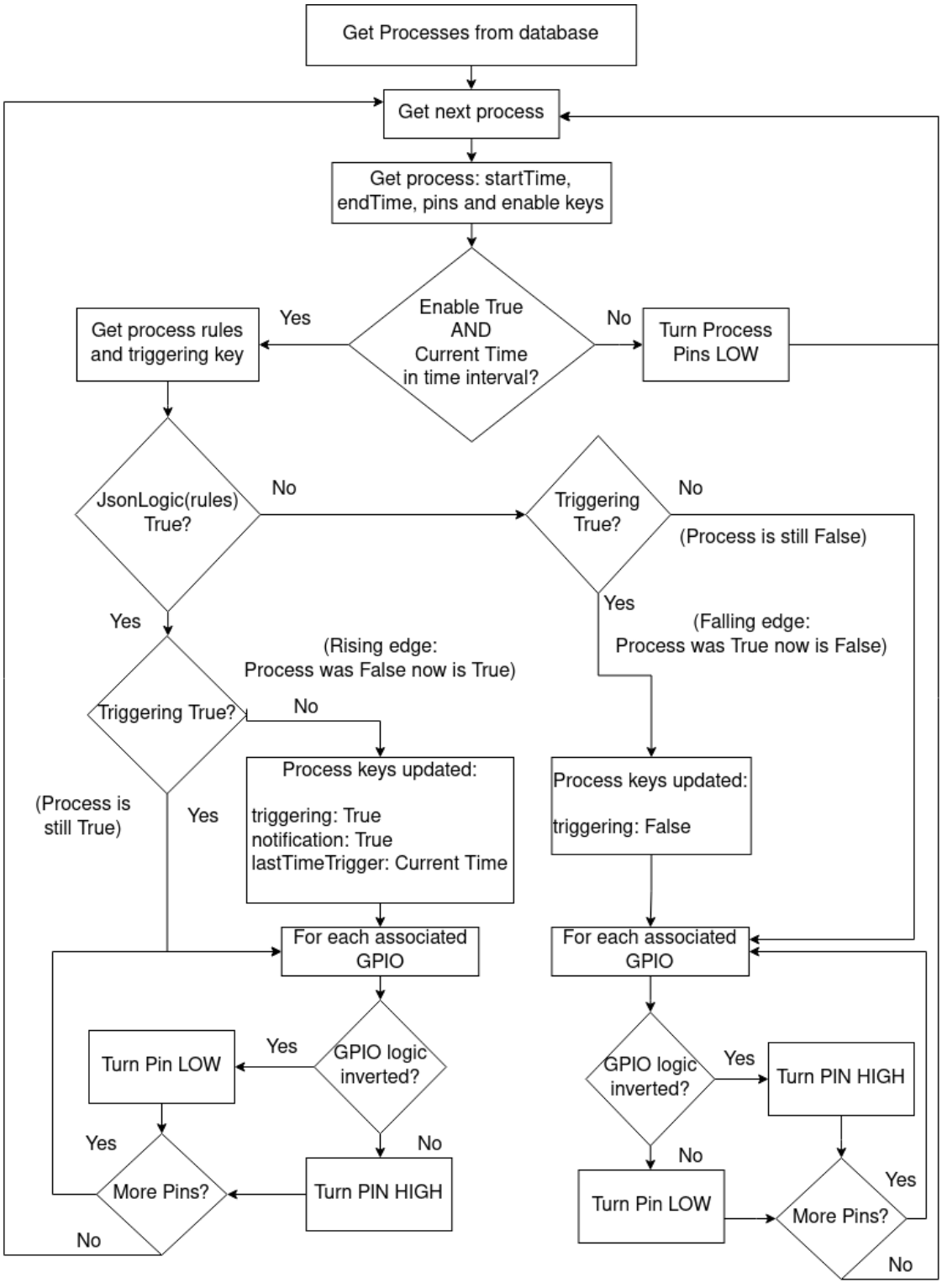


Figure 4.35: Process Trigger algorithm flowchart

As a safety precaution, a function was implemented that assigns the logic value of 0 to all process pins whose activation time interval does not contain the current time or process enable state is False. This function is essential, particularly for processes that have pins with inverted logic.

If the rule associated with the process is false, the pins with inverted logic will be at 1. Suppose the rule state does not change and the time interval of that process is exceeded, or the user changes the enable state. In that case, those pins will continue with the logic value of 1, thus deceiving the traffic light controller connected to those same pins. This function will be executed when the algorithm selects the active processes by analyzing the time interval of each one, performing on those that are discarded. Of the algorithm described, the most challenging implementation was at the level of checking the state of the logical condition of each process. One of the significant advantages of JsonLogic is that it has a function that logically determines the value of the condition. This function is jsonLogic.apply(rules, data), and its arguments correspond to the logical conditions - rules - and the value of the variables if their value had not been previously set - data. It was mentioned in 4.2.1.3 that there were nodes representing variables, such as vehicle detection, jam detection, flow detection, etc. However, instead of implementing these variables as variables of the logical condition, it was decided that they represent functions with the arguments that characterize the variable represented by the node. For example, if in the logical condition the node vehicle detection is present and if it were considered a variable within the JsonLogic syntax, a data of type string with the name of the variable and its arguments - zone and vehicleType - would have to be exported. Alternatively, it would have to be exported as a variable in object format with associated arguments. These two ways would make the system somewhat insecure on the back-end side since it would need an algorithm that parses and compares strings. Typically, these methods are avoided if possible since there can be flaws associated with string parsing. Also, internally, string comparison is a cumbersome process that makes the system less optimized. The solution to deal with this drawback is making these variables as if they were functions. The goal is to implement these functions so that the process trigger script gets the traffic data corresponding to the rule defined in the process. The way these functions are implemented could be another problem. However, JsonLogic allows the implementation of new operators in its open-source code. This demonstrates once again the benefits associated with this software. The JsonLogic source code has been integrated into this project in the Python version, and the following functions have been added, each directly related to variable nodes:

- `vehicle_number(zone, vehicleType)`
- `vehicle_detection(zone, vehicleType)`
- `flow(zone, vehicleType, duration)`
- `stay_time(zone, vehicleType)`
- `jam_detection(zone)`
- `crowd_detection(zone)`
- `double_park(zone, vehicleType)`

Each function should return either a boolean or an integer value depending on the variable. In `vehicle_number(zone, vehicleType)` function, the algorithm will search the database for the number of vehicles of type *vehcileType* in the zone *zone* when *startTime* of the message is bigger than process *startTime*. It will then fetch the number of vehicles from the current date and return the difference between these values. The function must return this difference because the record of the number of vehicles in the database counts from when the camera started sending this data. So, in a rule where it is defined, for example:

- IF *vehicle_number(Car, Camera 1 - Sensor 1)* > 20 -> activates output 4

The function `vehicle_number(zone, vehicleType)` will be called, and if it returns a value of 30, JsonLogic will check that the condition is True. On the other hand, `vehicle_detection(zone, vehicleType)` is just an algorithm that detects whether a vehicle has passed. So, each time a vehicle passes a particular zone, the condition is True and then False. This does not happen with the function `vehicle_number(zone, vehicleType)`. Thus, in order to be able to generate a pulse related to the quick passing of vehicles or to generate a continuous signal, these two functions were distinguished. So the function that only detects vehicles and generates impulses was one of the most complexes to implement. There is in the database the last count of each vehicle type. So that will be the point of comparison for when the function searches the database for how many vehicles have been detected. This is because the camera returns the number of vehicles detected and not an associated *boolean*. In addition, care was taken to update these values at the end of checking all logical conditions, as this function may be present in more than one condition. This way, if the number in the camera data database file is greater than the number of registered vehicles, it means that new vehicles have been detected. Therefore, the condition is momentarily True.

As for how the other functions implemented as operators in JsonLogic work, each one will behave differently. Starting with the function `flow(zone, vehicleType, duration)`, the flow analysis will consist of the past. First, the output of this function will be the number of vehicles because its setting represents the number of vehicles that have passed that zone during that time interval - duration. So, the flow means how many vehicles passed in that zone over time. This way, the function will fetch from the database the record whose *UtcTime* is greater than the current time minus the *duration*. This search is done by considering the results after searching by vehicle type and zone. Similar to the function `vehicle_detection(zone, vehicleType)`, the algorithm will calculate the difference between the number of vehicles present in the current record and the record that corresponds to the duration time. Thus, the function obtains and returns the number of vehicles detected in that period. The messages fetched by these two functions are associated with the Counter task. On the other hand, the following functions will search for messages whose tasks correspond to Idle Object. This is the case with the function `stay_time(zone, vehicleType)`, which will return the time that a vehicle of type *vehicleType* has been idled in the *zone* lane. Initially, this function does a filtered search by vehicle type, zone, and start time.

From the returned document list, the status of the last document is searched. If it is True, it means that this vehicle is still idle. This way, the start time associated with that event will be subtracted from the current time, determining how long the vehicle is idle. This amount of time that has been calculated is the variable returned by the function.

The implementation of the following functions is slightly more straightforward, as these will only have to return a boolean. Thus, these functions will search in the database for the task whose startTime Key is greater than the startTime of the process, and the associated zone and type of vehicle, if applicable. Once they have the list of documents, they will analyze the state of the last document and return it directly. Once the list of documents is obtained, the Key state of the last document will be returned. Concerning each of the functions, this means that:

- jam_detection(zone) - The last document has the last state of a jam detected. So if the state is True, the traffic jam still exists. Otherwise, the state is False, meaning there is no traffic jam.
- crowd_detection(zone) - The last document has the last state of a detected Crowd. So if the state is True, it means the Crowd still exists. Otherwise, the state is False, which means there is no Crowd.
- double_park(zone, vehicleType) - The last document has the last state of a detected double-parked vehicle. So if the state is True, the vehicle is still there. Otherwise, the state is False, which means the vehicle is gone.

This method of implementing new functions allows the system to be compatible with any operation, even if this represents a variable, as is the case. Thus, whenever it is desirable to add a new node that represents a new variable, a function in the JsonLogic source code should be implemented to return the desired variable.

4.3 Software Implementation on Hardware

The project's ultimate goal is to obtain a module that reaches the desired operating point. Thus, it is necessary to run all the software developed so far through the Raspberry Pi to complete the development and carry out a series of tests to validate the final implementation. The whole process developed was performed on a Raspberry Pi 2. However, the process will be the same as using other more modern platforms, such as Raspberry Pi 3 or Raspberry Pi 4.

The operating system installed on the SD card will serve as storage is the Raspberry Pi OS 32-bit. Thus, it is necessary to make the installation of the various libraries used throughout the code. This process is quite simple, given that all the software with the respective version is registered in the requirements.txt file. So, after including the project with all the code in Raspberry, just run the command:

```
[pi@raspberrypi/directory ~]$ pip install -r requirements.txt
```

This way, the back-end system is ready to run. Now, regarding back-end execution, two different scripts must be executed. The first one supports REST API execution; however, it is executed in production mode this time. This means that the application will no longer execute in developer mode. The other one is the main script *app.py* that includes the MQTT subscriber client to communicate with the MQTT broker and the process trigger algorithm that constantly verifies the logical conditions associated with the configured processes. It is important to remember that when the module starts to execute in a real situation for the first time, some files in the database must be wiped out. Those files are camera.json and sensors.json since they have stored data from the cameras and the configured zones to which the system should subscribe. This way, this data will undoubtedly interfere with software behavior. The entire project has been compiled into the folder on the front-end side. Furthermore, when running a compiled project, there needs to be a server hosting the *Vue* application. Thus, the HTTPS server that works effectively - *Caddy* - was installed. This software is an extensible, cross-platform, and open-source web server. This way, access to the interface could be done outside the local network. Inside the *Caddy* configuration file, it is necessary to define the path of the folder with the compiled web interface files and start the server according to the following code:

```
:80 {
        root * /home/user/Desktop/Jamadgy/src/frontend/dist
        file_server
        try_files {path} /
}
```

To complete the module setup, it is necessary to install the MQTT broker on the Raspberry so that no dedicated service outside the module is needed to support the MQTT communications

between the device and the CCTV cameras. This way, the chosen MQTT broker software was the same as in the development stage - Eclipse Mosquitto. It would only be necessary to change the IP, which can be done through the web interface, to the Raspberry Pi's IP. Finally, the system is ready to work in a real environment. The blocks corresponding to the completed system and their interactions are shown in figure 4.36.

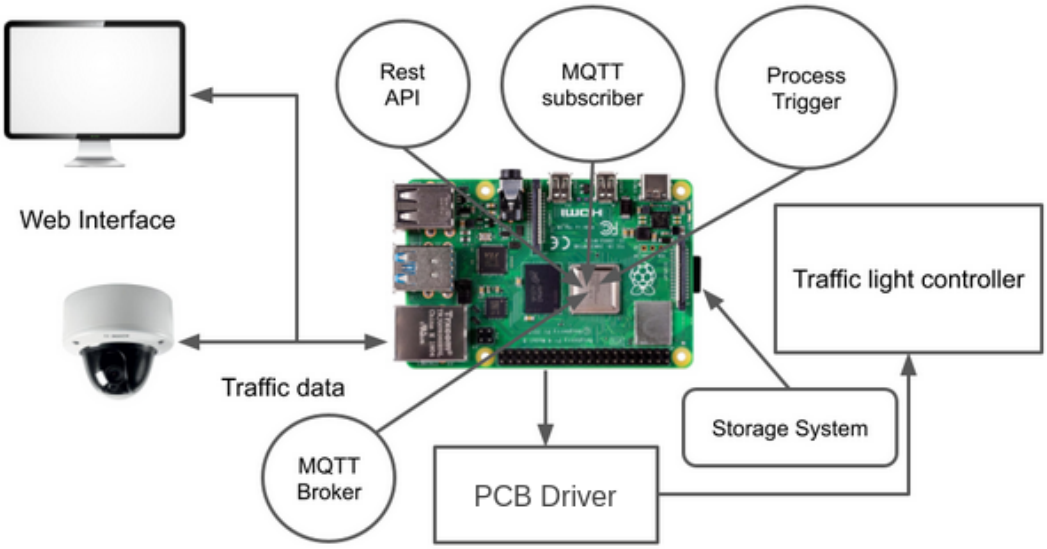


Figure 4.36: Final system blocks and components

To complete the developed software, it is necessary to add a new function that turns Raspberry Pi GPIOs into outputs and assigns them the desired logic value. Thus, the following lines were added to the process_trigger script:

```
import RPi.GPIO as GPIO   //imports library

GPIO.setmode(GPIO.BCM)  //referring to the pins by the
                          "Broadcom SOC channel"

gpio_list = [4 ,5 ,6 ,7 ,8 ,9 ,10 ,11, 12, 13, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27]         // setting the available pins

for p in gpio_list:
    GPIO.setup(p, GPIO.OUT)             //assign each pin as an OUTPUT
```

Then, the functions used to set the pins HIGH or LOW are the following:

```
GPIO.output(GPIO, 1)        //sets GPIO HIGH
GPIO.output(GPIO, 0)        //sets GPIO LOW
```

Finally, to make the whole process automatic, a Shell script was developed that allows whoever will configure the software on the Raspberry Pi not to have to execute specific and not very intuitive commands. This script provides several options:

- 1) Resume System - If the execution of the software was interrupted for any reason, this option resumes its execution
- 2) Initialize System - A series of settings must be made if the software has never been run. These include installing the libraries used by Python, creating the database system, setting up the MQTT broker IP, and setting up the Web Interface password. In addition, in the end, it runs the software
- 3) Reset System - This option allows deleting all the data from the database. Then it is necessary to re-configure the MQTT Broker IP and the web interface password, which are also included in this option
- 4) Change Web Interface Password - If the user only wants to change the password, he can do it through this option
- 5) Change MQTT Broker IP - If the user only wants to change the MQTT broker IP, he can do it through this option
- 6) Quit - To stop script execution.

4.3.1 Project Considerations

After obtaining the finished product, i.e., the Raspberry running the developed software, it is possible to reflect on the contributions that this project may have associated. The solutions on the market are very little documented as their development is private. Even so, the existing alternatives are very complex systems that need adequate infrastructures that imply huge investments. Thus, this project is a simple and much cheaper solution to operate in regions that do not justify installing these centralized and high-cost systems. Furthermore, the environment where the module was designed to operate is standard, i.e., communication with the CCTV cameras is through the MQTT protocol, and communication with the traffic light controllers happens through digital pins. This allows this module to be implemented on small structures that can provide traffic data and have traffic light controllers previously installed. On the other hand, the module's state of development does not give it the conditions to be implemented directly in an industrial environment. So, while the products available on the market are entirely suitable for installation and operation, this module does not meet robustness and mechanical strength requirements. This means that no safety systems in software failure were developed, as it would be operating in an environment where any failure is critical. It is essential to highlight that the more functionalities the module presents and the more complete it is, the higher the associated cost will be. Therefore, it is necessary to analyze whether the additional cost of the system is justified by the new technologies implemented. However,

considering the stage of development, the production cost is only based on the value of the SBC used, in this case, the Raspberry Pi 2 and its storage system - an 8GB SD card. In addition, the PCB board will cost about 30€ according to *SOLTRÁFEGO, SA*. The system is budgeted at €30 for the Raspberry Pi 2 and a further €10 associated with the storage system, giving a value of €40. If the PCB board is incorporated into the system, which is a crucial aspect, the total cost is €70. Finally, the module has limits that start when the objective is the control of large traffic structures, but it was not developed for that. This way, this project acquires an enormous potential to fill a space in the market that, until then, remains empty.

Chapter 5

Results and Validation

The validation process of the different technologies associated with the system will be divided by each subsystem. Thus, evaluation of the system performance will be focused on each block. Still, all data generated and tests are performed while all module subsystems execute in parallel, as happens in a real situation. To perform a validation that focuses on the level of software errors and associated bugs, the system was tested on the same platform it was developed. Once all the tests were performed, the system was executed in the module. In this case, it is possible to see if the activation of digital pins has the desired behavior and if the hardware computing power is sufficient to perform all the necessary tasks. Here we will focus on load tests to understand what the limitations of the module are.

5.1 Laboratory Testing

The environment where the tests are carried out is as close to a real situation. Although first-hand testing will be done in a development environment, the goal at this stage is to run the entire system on the Raspberry Pi 2 platform with an 8GB micro SD card. So, in the first phase, the testing environment consists of a workstation with an AMD Ryzen 5 3600 3.6GHz processor, an ASUS Dual series Radeon RX 580 OC graphics card, and 16GB of RAM. Kingston A2000 500GB NVMe supports storage. The operating system is Arch Linux with Linux kernel version 5.18.2-arch1-1.

To ensure a connection to the cameras, a CCTV camera provided by *SOLTRÁFEGO, SA* was configured with the Configuration Manager software. Cameras such as this will generate data and send it to the module. As such, tasks were created to generate data related to the traffic situations associated with the task. Note that the amount and type of data generated depend directly on the traffic in the area where this camera is installed. This way, using the MQTT explorer software, an MQTT client was created that subscribes to the same broker as the camera and the module. Thus, the data can be accessed by any client that connects to the broker. Currently, the broker is running on a separate server. However, a dedicated service will be created inside the Raspberry Pi platform to support the broker's execution. Thus, some tasks have been configured that the module will need to identify and register in the database. Figure 5.1 shows some data that the CCTV camera published in the broker, whose all topics were subscribed by the MQTT client.

Figure 5.1: CCTV camera published data

As said before, the generated data depends on what happens in traffic. Therefore, for this test stage, it does not make sense to wait for the camera since the traffic flow on the street where the camera is installed is too low. Furthermore, specific traffic circumstances are unlikely to happen on that street, such as a double-parked vehicle or a traffic jam. Therefore, a script was created that simulates the camera. The same data that the camera could generate will be produced faster by this script. Additionally, it is possible to receive data corresponding to specific and rare traffic

circumstances. Hence, this script generates one MQTT message per second, and each message represents a traffic situation. It was defined messages for all the different situations the module should be handling. This way, it is possible to generate all types of data much faster. Furthermore, it is possible to simulate several cameras to verify if the system can handle data from all of them, simultaneously. Before executing this script, the system was turned on while the real camera was sending data. It was possible to verify that everything worked as expected on the MQTT client-side, i.e., data was correctly integrated, filtered, and inserted into the database. It is essential to know that the name of the camera associated with the MQTT topic was previously associated with the module by the web interface. However, this test will be carried out further on. From this moment on, the tests triggered during this phase will have as data generator and publisher the script described above that allows not only simulating the real camera but also other cameras with all kinds of data. The list of topics created in the camera simulator script is illustrated in chapter C. There, two cameras were simulated - T1 and T2 - and each was assigned different tasks. First of all, it is necessary to add a new sensor on the settings page of the web interface, as it is shown at figure 5.2.

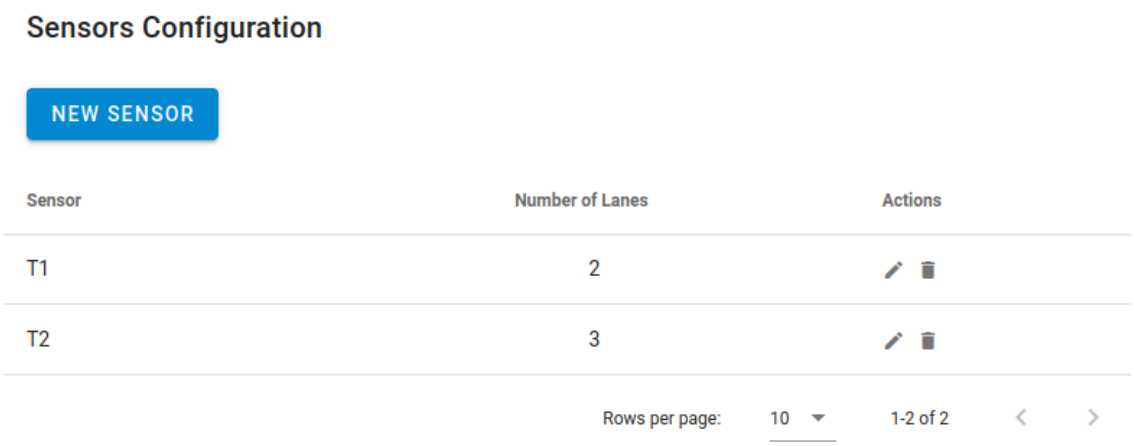


Figure 5.2: Sensors configuration

Everything went as expected, from the sensor configuration to the notification indicating that the sensor was inserted into the database. Checking the database, the file had the following information:

```
{
"_default": {
    "1": {
        "name": "T1",
        "lanes": 2
    },
    "2": {
        "name": "T2",
        "lanes": 3
         }
    }
}
```

This test started with the camera.json file empty. With the module ready to accept a new sensor, it is time to start the script so the data can be generated and published. After the data publishing, the database file camera.json had stored the information ilustrated at figure 5.3.

{"_default": {
"1": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Idle Object", "Vehicle": "Car", "Zone": "T2-1", "State": "true"},
"2": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Idle Object", "Vehicle": "Car", "Zone": "T2-1", "State": "false"},
"3": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Idle Object", "Vehicle": "Truck", "Zone": "T2-2", "State": "false"},
"4": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Idle Object", "Vehicle": "Car", "Zone": "T2-2", "State": "true"},
"5": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Double Park", "Vehicle": "Bike", "Zone": "T2-1", "State": "true"},
"6": {"Cam": "T2", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Double Park", "Vehicle": "Bike", "Zone": "T2-2", "State": "true"},
"7": {"Cam": "T1", "UtcTime": "2022-06-12T20:09:57.935426Z", "Task": "Crowd Detection", "Zone": "T1", "State": "true"},
"8": {"Cam": "T1", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Crowd Detection", "Zone": "T1", "State": "false"},
"9": {"Cam": "T1", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Counter", "Vehicle": "Truck", "Zone": "T1-1", "Count": 138},
"10": {"Cam": "T1", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Counter", "Vehicle": "Car", "Zone": "T1-1", "Count": 138},
"11": {"Cam": "T1", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Counter", "Vehicle": "Bike", "Zone": "T1-2", "Count": 138},
"12": {"Cam": "T2", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Jam Detection", "Zone": "T2-1", "State": "true"},
"13": {"Cam": "T1", "UtcTime": "2022-06-12T20:10:04.942367Z", "Task": "Jam Detection", "Zone": "T1-2", "State": "false"},
"14": {"Cam": "T2", "UtcTime": "2022-06-12T20:10:10.949030Z", "Task": "Idle Object", "Vehicle": "Car", "Zone": "T2-1", "State": "true"},
"15": {"Cam": "T2", "UtcTime": "2022-06-12T20:10:10.949030Z", "Task": "Idle Object", "Vehicle": "Car", "Zone": "T2-1", "State": "false"},
"16": {"Cam": "T2", "UtcTime": "2022-06-12T20:10:10.949030Z", "Task": "Idle Object", "Vehicle": "Truck", "Zone": "T2-2", "State": "false"}
}
}

Figure 5.3: camera.json file after script execution

It is possible to check that all data was correctly filtered, stored and each object had the correct information. Furthermore, it is verified that the software is prepared to handle different cameras, and the sensors management system allows for adding new sensors. During this operation, all the subsystems were running in parallel, and all the processes were being verified, although they enable state was false. Therefore, the output off the process trigger algorithm was to turn all pins to LOW, as expected.

As the system keeps working and the camera simulator script, the next test focus on the interaction between the user and the interface. Consequently, the communication between the front-end and the back-end through the REST API. To better organize the shown Log files, the interaction will be done by each web page. So, during this interaction, a login is performed, and on the settings page, the system password, MQTT Broker IP, is changed, created, edited, and deleted sensors. In chapter D section D.1 are represented the lines that belong to the REST API logs indicating all the HTTP requests and the performed actions. A simple analysis of the log messages indicates

that all of the actions taken in the web interface were well communicated to the back-end system and appropriately executed. So everything went as expected. An important aspect is the fact that the password is hashed. This happens for security reasons,i.e., the password is constantly stored after it is hashed. This way, it cannot be decrypted. When it is printed [GET SENSORS] or [GET BROKER], the page has been updated, so it is necessary to return all the appropriate data to the front-end system. There were no general errors, and the system behaved as expected. It is also essential to interact with the processes page. The number of options in this page is extremely large. The `get_processes()` function is being called by the front-end every three seconds. This way, system logs would be too much extensive to be shown. A simple solution is to increase the timeout value to ten seconds temporarily. In chapter D section D.2 are represent the lines that belong to the REST API logs indicating all the HTTP requests and the performed actions. It is shown that all the functions are correctly called since no HTTP request presents associated errors.

All the possibilities of interaction with processes were performed, such as creating, editing, deleting, enabling, and updating. An important aspect is that the function `get_sensors()` is being called when a node is placed in the rule configuration interface. This is expected since, in those nodes, it is necessary to choose a zone. The sensor and the respective lane define that zone. Therefore, the front-end calls this function to know which zones do exist. On the web interface side, there are no limitations when configuring processes. All the described functionalities work as expected, and therefore, in this test phase, both the front-end and back-end systems had no associated errors and behaved as expected. To test the process triggering system and the communications with the front-end regarding their configuration, four processes were created:

- Car detection - IF more than 5000 cars are detected in T1 - lane 1 zone -> GPIO 3 (non-inverted) is activated
- Traffic flow - IF more than 300 cars are detected in T2 - lane 1 zone in 2 seconds -> GPIO 2 (inverted) is activated
- Double-Park - IF a bike is double-parked in T2 - lane 1 zone -> GPIO 5 (non-inverted) and GPIO 25 (inverted) are activated
- CrowdDetection - IF a Crowd is detected in T1 zone -> GPIO 6 (non-inverted) is activated

None of these processes have been triggered before. Their enable state and the defined time range interval include the actual time when this test was performed. The first two processes are not supposed to trigger; thus, their enable state was *false*. The goal is to manipulate the database to create a situation so Double-Park and CrowdDetection processes can be triggered. Their enable state was set as *True* and the defined time range interval included the actual time when this test was performed. This way, if the system triggers those processes and assigns the correct values to the associated GPIOS, this validation is a success. Given the number of possibilities of rules configuration, assigned pins, and all the circumstances that must happen in a real traffic situation, it is impossible to cover all types of tests. The *processTrigger* script logs are illustrated in chapter E. Those logs show that the first two processes were *OFF*. Therefore it was necessary to shut down

all of the associated pins. Regarding the other processes, the first time they were verified, a Rising Edge happened - their condition was False, and then it was True. This is where the notification must be shown in the web interface. In the next loop, that rising edge did not happen since both processes were already True before - displayed in lines 26, 29, 38, 41, 50, and 53. Regarding the pins management, Double-Park process has two associated pins - GPIO 5 (inverted) and GPIO 25 (non-inverted). At the rising edge, the assigned logic value to each one was correct - displayed in lanes 10 and 11. At the rising edge of the CrowdDetection process, since it has only the GPIO 6 (non-inverted) associated, it is necessary to change its logic to HIGH. Although this was a test in a controlled environment, it was successful and met all expectations.

On the web interface side, it was expected that notifications would be shown, indicating that these processes had been triggered. In addition, the table should show the current state of each process and the last time it was triggered. In figure 5.4, it is illustrated the appearance of the web interface right after the processes were triggered. It is possible to see that both notifications were well displayed, the current state of each process, and the last time they were triggered. Overall, all subsystems played their part, and there were no associated errors, even though the test was not performed in a real situation.

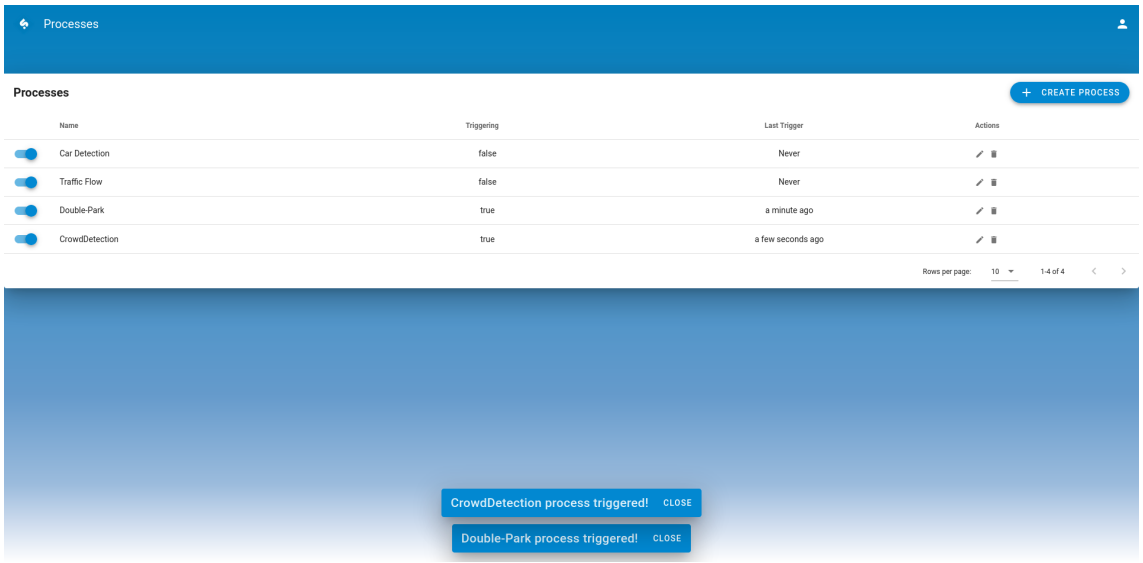


Figure 5.4: Processes page after process triggering

A raspberry Pi 2 with an 8GB SD card was used for the tests and validation of the module. After all the setup necessary to get the device working done. The testing phase started, where the general behavior of the Raspberry Pi was observed. Then immediately moved to test the triggering of configured processes and mainly the activation of GPIOs. Initially, this phase focused on understanding if there was any essential operation with associated errors or bugs that had not been detected in the previous phase. The results analyzed during Raspberry Pi execution through the console logs showed no problem associated with the system's operation. Although in this phase, the analysis of the processes was done in a controlled environment, it was verified that the analysis of the logical conditions worked correctly and that there were no performance problems or lack of computing power to support the execution of the software. This way, it has been shown that the software works correctly and that all the subsystems running simultaneously do not compromise the module's performance. The next step was to understand, through a load test, whether the device can continue to support the system. The same script described previously was used to create a series of traffic data that simulates the CCTV camera. This test started when the database file storing the data from the CCTV cameras contained around 100 000 stored messages. It is essential to mention that no subsystems were switched off to make the environment as realistic as possible. After working for a few hours, there was no crash associated or any event that could jeopardize the system. Still, it was necessary to check if the performance has decreased. On the back-end side, all of the HTTP requests were being corresponded to REST API. The other script was responsible for integrating traffic data from the CCTV camera and checking if all configured processes were able to deal with the flow of data. In this case, measuring the frequency with which the processes are checked is not accurate. This feature depends on the amount of traffic data generated and published by the CCTV camera. Furthermore, it was defined that each process is verified after each message is stored in the database. During this test, the CCTV camera simulator script generated one message per second. All of the processes were also checked one time per second. This frequency is considered reasonable for a real environment. Therefore, it is certified that, under those conditions, the module can work within the necessary time constraints.

It is necessary to analyze the module's time performance on the front-end side. It was purposely set up the conditions for the processes that triggered previously to trigger again in order to check whether the associated delay is significant or not. Here, there were no relevant delays; however, some measurements of response time from the back-end were taken through the network monitor tool provided by the Firefox browser. Figure 5.5 shows the time values associated with the back-end response for each HTTP request made.

Figure 5.5: Response time for HTTP request in Raspberry

It is possible to observe in the register that there is a response approximately every three seconds - called by the implemented loop in the front-end system. The first two values of each cycle are related to the sending of the processes. The third value corresponds to the current communication state between the system and the MQTT broker. Then, the values obtained oscillate between 8 and 57 ms. These values are already quite reasonable. Even so, the interaction with the interface was performed with a computer in the same local network as Raspberry. Still, there is always some delay associated with the ping. The Raspberry was pinged by the same machine that accessed the web interface to verify the value associated with the ping. Over ten observations the ping periods ranged between 0.978 and 1.31 ms. These values are shallow, so their impact on the back-end response time is meager. The values obtained were also compared with associated values with the execution of the software on the workstation where it was developed, i.e., a machine with much higher processing power. Thus, for a similar execution environment, the values obtained were 1 or 2 ms. This time the values are much lower, as one would expect. However, the delay associated with the Raspberry operation on the front-end side does not compromise the system performance.

The verification of the behaviour of the digital pins was an important part during this process, since it was one of the points that had not been possible to test. This way, the pins behaviour

was as expected and there was no relevant delay associated. However, during the testing process a situation occurred that could jeopardise the safety of the environment where the module operates. When the execution of the software was interrupted on purpose, the digital pins with a logic value of 1 were not switched off. This means that if the program crashes or simply stops working, the traffic light controller is not warned and communication through the GPIOs will represent an incorrect situation. This drawback was discovered at a very advanced stage of project development. Therefore, it was not possible to implement a solution to solve it. However, this solution was designed and will be described in the next chapter. This phase of testing the system, both running on the workstation and the module - Raspberry Pi - was quite preliminary. However, most of the requirements have been shown to work as expected, and no changes to the system have been necessary so far. Still, this topic will be explored in the next section.

Chapter 6

Discussion

After a whole approach to the development of the dissertation project, it is necessary to reflect on the points dealt with so far. Thus, several aspects related to the approach used, the module performance, and the objectives achieved will be discussed in this section. Here, specific issues that go beyond the development and implementation of the system will be considered. This consists on a conscious and critical analysis of the work.

6.1 Software and System Architecture

The whole system should be reflected in a simple and functional module. This point made it clear that the choice of hardware system should focus on an already developed board with several associated functionalities, mainly at the communications level. The fact that the choice was a Raspberry Pi contributed to the ease of doing the complete setup without significant difficulties, thanks to the enormous community around this product. Besides, the mass production of these products and their use over the years gives them a certain level of reliability that is necessary when it is working with critical systems such as traffic management.

The approach to the software system proved to be more complex, given the amount of functionality that this system should integrate. The usage of a methodological approach came in a natural way, starting by analyzing all the requirements. Thus, blocks were formed with particular functions assigned, and it was well defined how each one communicated and its role within the system. The structure was defined as soon as each subsystem was established. Most of the blocks were emerging due to one or more objectives being met. The choice of tools that would support the development of the subsystems left more room for analysis and comparison. The choice focused mainly on the developer's experience and the capabilities of the tool that would be chosen. Thus, it was decided that Vue.js would be a suitable choice to support front-end development, just as python provided several libraries for implementing a solid and robust back-end system. The choice of the remaining tools was narrower; however, the preliminary decision fell on the structure in which the database would be implemented, a non-relational database. One of the

most exciting decisions in this solution was using JsonLogic software. Given the complexity of analyzing logical expressions with compound variables, several logical and arithmetic operators, and the inclusion of traffic circumstances. This software simplified what would be a complex and time-consuming implementation into a simple and flexible system. At the end, the integration of the different blocks and their functions has culminated in a solid structure that is scalable and also modular, giving scope for future additions/changes to be made without compromising the implementation of the system.

6.2 Development considerations

With a well-defined structure and the software to be used, the development focused on obtaining a simple and functional system that would meet the minimum requirements. From there, the goal became to scale the system to the desired operating point. In hardware, the system itself was practically complete. However, it was verified that an intermediary would be necessary to make the communication between the Raspberry Pi and the traffic light controller compatible at the voltage level. Even though the production and construction of the PCB board were not ready, the electrical schematic was validated by *SOLTRÁFEGO, SA*, thus becoming a viable solution to the problem of compatibility in the communication of the digital pins. On the other hand, software development was the biggest challenge of this project. On the front-end, the UI kit used - Vuetify - proved to be an essential tool, mainly in terms of the design and aesthetics of the web interface. There were no limitations from both frameworks - Vue and Veutify - that jeopardized the front-end implementation. The Rete.js framework had some problems, mainly in the integration with Vue. However, its graphical tools allowed a much more pleasant, intuitive, and professional user interaction with the interface. One of the biggest challenges of the front-end development was exporting the logical conditions defined through Rete.js block binding to JsonLogic syntax. On the other hand, the rest of the rule parsing process would be significantly optimized on the back-end side. So the front-end had its setbacks, but it turned out to be the project's most solid and complete subsystem.

The back-end system was more complex since it comprises several subsystems. The software used to implement the database allowed a very optimized organization of the project because TinyDB stores the data in JSON files. Thus, an associated file was created for each data type, resulting in 5 different files - auth.json, camera.json, processes.json, sensors.json, and system.json. The biggest challenge was the implementation of the REST API because besides communicating with the front-end, it managed the communication between the other systems according to the web interface requests. This allowed the data integration to be independent and only aimed at integrating the traffic data and filtering it to be entered correctly into the database. The rules defined in the Subscriber script guarantee a flawless and secure communication system between the camera and the module. Finally, the JsonLogic software algorithm analyzes the logical conditions. Thus, the variable nodes were considered functions. This decision allows that whenever a new block needs to be added, it is only necessary to implement the function corresponding to that block to return

the data related to the logical condition checking function. The rest of the algorithm was intuitive, where its greatest complexity lay in adapting the activation of the pins to the reversing logic functionality of each. At the end, integrating the software with Raspberry required no additional implementations other than the activation functions of the digital pins, as expected. In short, the development of the project presented its challenges. Still, many of the problems were simplified by the tools used. With this, a system was achieved that meets expectations and can perform the assigned tasks.

6.3 Module Validation

An essential step in validating any system is the testing phase. Even though the final product has not reached an industrial production stage, it was possible to do a series of tests and verify the project's performance. In the development environment, all the blocks were tested individually and successfully. When moving to the Raspberry, the goal was to see if there were any undetected bugs or associated incompatibilities. Besides that, along with the development, some questions arose regarding the performance of specific software, mainly at the database level, that not being sufficiently optimized could jeopardize the module's performance. A Raspberry Pi 2 was used on the load test, and its performance remained within the time constraints, although a slight difference was noted for the tests on the workstation. Even though the system has successfully passed this testing phase, it is essential to emphasize that the module is not ready for an industrial application. For this, it is necessary to conduct more demanding tests over a longer periods and in a real environment where the system's behavior would be constantly monitored, given the traffic circumstances that are communicated. However, it demonstrates the project's potential and that the main objective was fulfilled.

As mentioned in the previous chapter, during the validation phase of the Raspberry Pi, it was noticed that if the software is interrupted, the digital pins with a logic value of 1 are not changed. This problem is reflected in the constant uncertainty of the traffic light controller knowing if the module is operating correctly. A possible solution was designed to avoid this potentially critical problem. According to the developed wiring diagram, the Raspberry Pi has four digital pins available. So, the goal would be to use only one digital pin that, during the execution of the software, would switch with a frequency of 100Hz, for example. This way, a simple micro-controller would be implemented on the board in the electrical schema, whose purpose was constantly monitoring that digital pin. Thus, if this GPIO stopped having this behavior, it would mean something was wrong with the module. When the micro-controller detected this failure, it communicated with the traffic light controller through a digital pin that would be HIGH when the module was operational and LOW when something was wrong.

To give an overview of the requirements that the project fulfills, chapter F represents the table F.1 - a simplified version of the requirements matrix developed during the preliminary phase of

the dissertation work. Analyzing the table, only the requirement *ELE_2006* was not fulfilled since there was no materialization of the developed electrical scheme in a functional PCB board. All the remaining requirements were fulfilled. A part of the objectives was not mandatory; however, for this module to reach a satisfactory market level, it should integrate at least all these functionalities.

Chapter 7

Conclusion

The need for transport, coupled with the enormous economic growth it provides, has led to the creation of various traffic networks to facilitate and even boost the transport system during human evolution. The same forces that drive residents to congregate in large urban areas can sometimes lead to unbearable traffic jams on city streets. The enormous impact caused by energy and economic costs, loss of time, and lack of security led humanity to find solutions to mitigate these effects. Since then, many solutions have been developed, most of which appear on the market through complex systems with a vast associated cost. However, several entities do not acquire these systems, either because they cannot afford them or because their needs do not justify the cost associated with massive infrastructures. This creates an opportunity in the traffic management industry to develop a low-cost, simple and reliable module that integrates the main features of a complex and high-cost industrial product.

Given the functions assigned to the module, its development focused primarily on implementing two systems that depend on each other - back-end and front-end. To this goal, some tools were used that aided the development of the system and had a very positive impact. Care was always taken to make the software simple, scalable, and optimized so there would be no problems when it was implemented in hardware. The choice of the Raspberry Pi as a hardware platform allowed not only to reduce the project cost but also to use an already complete system since it was not necessary to add peripherals to the board. However, to make the system closer to an industrial version it needed a full working PCB board that would make compatible the voltages levels at which the module and the traffic light controller operate. Therefore, the respective electrical schematic was drawn, giving space for the production of the PCB version. Nevertheless, this does not invalidate testing the system and checking its performance. The tests to which the system was submitted allowed most of the requirements to be validated, whether critical or worthwhile. On the other hand, it is essential to point out that the implemented architecture allowed some changes to be made without compromising previous development. Mainly, the implementation of logical conditions construction through the Rete.js framework and the web interface settings page that allows the definition of the sensors that the module should communicate with. To validate all

the implemented functionalities, it would be necessary to conduct a rigorous and demanding test protocol to check the system's upper limits and associated flaws and bugs. The system should be worked on to reach the industrial level suitable for its commercialization. However, analyzing the requirements matrix, almost all functionalities, especially the most critical ones, were successfully implemented.

In conclusion, the implemented system met the established goals, becoming a promise to intervene positively at the traffic management level, especially for areas that do not justify the investment in complex and expensive solutions. Thus, this dissertation eases the access to technologies that enhance the economic and social activity of a region, minimizing costs and integrating the main benefits of those solutions.

7.1 Future Work

Besides checking the positive points of the work, it is necessary to identify the points where it can be improved. Doing this opens up room to improve the system without associated weak points and with the best possible performance. The first point that must be clarified is that the module is a system that has always presented certain limitations. Its main objective has never been to replace a centralized and complex system capable of managing traffic in a large region. This project presents a system capable of filling a gap in the traffic industry. Thus, entities responsible for local traffic management that, do not have the financial capacity to acquire infrastructures that imply significant investments, can acquire this product that will serve their purpose. Then, although the testing and validation stage found that most of the system's functionalities work without a problem, the system is not entirely validated and ready for industrialization. The environment where it will be put into operation is very demanding and critical, i.e., there cannot be any faults associated with the module's operation. For this, it would be necessary to develop an entire testing protocol that would allow constant monitoring of the system's behavior while operating in a real and uncontrolled environment. Thus, in addition to more restricted and demanding testing, there are some details where the system can improve.

One of them is the module's storage system. The raspberry platform is prepared to have a storage system supported by an SD card. These kind of technologies are not very stable and do not last long. This harms the module's stability and reliability, making it necessary to do more frequent maintenance or even use another more stable storage technology. This brings another feature that could be implemented to ensure the module's reliability, which would be a RAID system. A Redundant Array of Independent Disks allows connecting two or more drives in the system to act as a fast, high-volume drive or configure them as a system drive used to automatically and instantly replicate the system data for real-time backup. This functionality would slightly affect the software's structure and especially the product's price. Thus, a central server should be implemented where all modules would store their data and constantly interact with it. This is justified if several

modules were operating. Otherwise, the cost is not justified. Another exciting implementation would be a satellite communication system to operate only in case of any internal and communications problems in the module's functioning. This way, if a problem occurs with the system, the responsible entity would be quickly notified and could quickly solve the situation. Otherwise, it would be necessary to check the system's status constantly. Another improvement that should be implemented is a module monitoring system is the solution described in 6.3. This aspect is essential for the security of the system and its reliability. Finally, improvements can also be made to the physical structure of the module. The Raspberry Pi platform has proven to be a very reliable system. However, this does not imply that it is robust and resilient to specific hostile environments. This allows to add some features related to the structure that will encompass the Raspberry Pi and the PCB board. Furthermore, there is no surge protection or systems against short circuits in the power supply circuit or even against the boards overheating. Then, the connection terminals at the GPIO pin level are not robust and reliable enough to industrialize the product. Regarding software, there are a couple issues that, although not critical, can be improved. The software developed at the level of communication between the front-end and back-end only allows this to be initiated by the web interface from HTTP requests. Thus, in certain situations such as process triggering, graph updating, and even alerting of communication failures with the MQTT broker depend on constant confirmation by the front-end. Hence, to allow the back-end to take the initiative in communications, a WebSocket could be implemented in the future. This way, besides the front-end not needing to constantly check if there are new data, the information can be transmitted in real-time, not dependent on the loops that execute the HTTP requests. On the other hand, the database system implemented has not proved to be a robust and effective solution, although it has not failed in any test. However, its use allowed to obtain a stable and functional system. Thus, software such as MongoDB could be used to ensure a more flexible and robust storage system. Once again, it should be considered that these more complex systems need dedicated hardware, increasing the project cost, and are more challenging to implement.

To conclude, there is still room to improve the developed product however, this would have to considered the cost associated with the new technologies to be implemented. Still, it is expected that the module will operate correctly in a real environment. However, given the criticality of the system where it will be inserted, more demanding and restricted tests should be conducted to validate the product at the industrial level. Even so, the development of this solution has created a new concept that could positively impact the advancement of the intelligent traffic industry.

Appendix A

Articles from the State of the Art review (excel tables)

Year	Title	Categories
2017	City-Wide Traffic Flow Estimation From a Limited Number of Low-Quality Cameras	Image analysis, AI, vehicle counting, unsupervised learning
2017	Vision-based surveillance system for monitoring traffic conditions	Image analysis, vehicle counting and tracking, Speed measurement
2017	FCN-rLSTM: Deep Spatio-Temporal Neural Networks for Vehicle Counting in City Cameras [arXiv]	Image analysis, Neural networks, Vehicle counting, crowd counting
2017	Vehicle counting in crowded scenes with multi-channel and multi-task convolutional neural networks	Image analysis, CNN, vehicle counting, static images
2017	Multi-field depth vehicle headlight detection by model construction and long trajectory extraction in nighttime city traffic	Nighttime vehicle counting, image analysis
2018	Measuring the road traffic intensity using neural network with computer vision	Computer vision, NN classification, traffic intensity
2018	An enhanced framework for multimedia data: Green transmission and portrayal for smart traffic system	Framework, ANN, Traffic management, Image analysis
2018	Vehicle Detection Using Spatial Relationship GMM for Complex Urban Surveillance in Daytime and Nighttime	Vehicle detection, nighttime and daytime, Gaussian mixture model based
2018	A Video based Vehicle Detection, Counting and Classification System	Vehicle detection, counting, classification, Gaussian mixture model based, openCV
2018	High variation removal for background subtraction in traffic surveillance systems	Background sub, NN, GMM, image analysis
2018	Vehicle detection by fusing part model learning and semantic scene information for complex urban surveillance	vehicle detection, framework, Image analysis
2019	Turning video into traffic data - an application to urban intersection analysis using transfer learning	CNN, vehicle detection, video analysis
2019	Traffic flow estimation with data from a video surveillance camera	R-CNN, image analysis, traffic flow estimation
2019	Simulation of traffic optimization to reduce congestion	Traffic management, YOLO, deep learning, vehicle detection
2019	Adaptive real time traffic prediction using deep neural networks	Vehicle counting and classification, Green-time predict, CNN, SSD
2019	Distributed Mean-Field-Type Filters for Traffic Networks	Image filtering, image analysis, vehicle detection, mean-field-type filter
2019	Traffic surveillance video coding with libraries of vehicles and background	Library, image analysis, background subtraction
2019	Vehicle Counting in Video Sequences: An Incremental Subspace Learning Approach	Vehicle counting, PCA, image analysis
2019	Road traffic monitoring system	Traffic management, vehicle detection, Image analysis
2019	Low-Cost Road Traffic State Estimation System Using Time-Spatial Image Processing	Temporal-space Image processing, Traffic estimation
2019	Robust Hierarchical Deep Learning for Vehicular Management	Deep learning, traffic surveillance, regression, congestion detection, crowd counting, ensemble learning, metric learning
2020	Intelligent video analysis for enhanced pedestrian detection by hybrid metaheuristic approach	Pedestrian detection, image analysis, HMPD, HMA, SVM
2020	Intelligent traffic analysis system for Indian road conditions	Morfological processes, Image analysis, weather state adapt, LoG, vehicle detection and count
2021	Domain adaptation from daytime to nighttime: A situation-sensitive vehicle detection and traffic flow parameter estimation framework	Vehicle detection, nighttime, daytime, faster R-CNN
2021	Vehicle Classification and Counting System Using YOLO Object Detection Technology	YOLO, CNN, vehicle detection and count, image analysis
2021	Integrating computer vision and traffic modeling for near-real-time signal timing optimization of multiple intersections	Framework, Traffic management, Image analysis, YOLO v3, COCO
2022	[1] A complex network analysis approach for estimation and detection of traffic incidents based on independent component analysis.	Traffic data analysis and forecasting
2022	[2] Artificial Intelligence-Empowered Logistic Traffic Management System Using Empirical Intelligent XGBoost Technique in Vehicular	Traffic data analysis and forecasting
2021	[3]Data Augmented Deep Behavioral Cloning for Urban Traffic Control Operations Under a Parallel Learning Framework	Traffic data analysis and forecasting
2020	[5] The Capacity of the Road Network: Data Collection and Statistical Analysis of Traffic Characteristics	Traffic data analysis and forecasting
2020	[6] Neuro-Fuzzy Modeling of Data Singular Spectrum Decomposition and Traffic Flow Prediction	Traffic data analysis and forecasting
2020	[7] Traffic Prediction Using Multifaceted Techniques: A Survey	Traffic data analysis and forecasting
2020	[8] Prediction based traffic management in a metropolitan area	Traffic data analysis and forecasting

Figure A.1: Table with articles and respective categories

Algorithms	Goal
Algoritmo próprio baseado em processos morfologicos [2]	Features extraction e vehicle counting
Algorithm based on the inverse Markov chain [2]	Traffic estimation
Method VWL [3]	Vison field calibration
Background subtraction e Haar-Cascade (for detecting vehicle shapes) [3]	vehicle counting
Method de Ross et al.'s [3]	vehicle Tracking
FCN-rLSTM [4]	Crowd and vehicle counting
CNN [5]	vehicle counting and traffic estimation
Own algorithm with morphological process[6]	Headlight detection
CNN with BLOB analysis [7]	vehicle counting and traffic estimation
Framework with ANN trained and morphological process [8]	vehicle counting
Traffic management [8]	Traffic management
GMM algorithm for daytime and nighttime [9]	Vehicle detection night and daytime
background subtraction with GMM and morphological process [10]	vehicle counting and classification
BS and entropy estimation with GMM and learning aplication in NN [11]	vehicle counting and classification
Probabilistic framework. [13]	vehicle counting
CNNfor learning e MobileNet para net training [14]	Traffic estimation
Improvement de R-CNN com focal loss, adaptive feature pooling, additional mask branch, and anchors optimization [16]	vehicle counting and classification
YOLO v3 e RetinaNet [17]	vehicle counting and classification
Traffic manangement with green time calculation[17]	Traffic management
SSD NN object detection trained with MS-COCO. Green time estimation [19]	vehicle counting and classification
Mean distributed "média-filtro" (DMF). Improvement de mean-field-type filter [20]	vehicle counting and classification
Vehicle and background segmentation com algoritmo SuBSENSE e comparação com biblioteca offline [21]	vehicle counting and classification
incremental PCA with morphological process to improve performance[23]	Vehicle movement detection
ROI selection [23]	vehicle counting
Morphological processing with BS (background without cars) [24]	vehicle counting and detection
Comparison of the number of vehicles on the roads to determine the signal to apply [24]	Traffic management
TSI [25]	Traffic estimation
Deep Network for hierarchical semantic feature extraction. Combination of several networks for better performance [26]	Crowd counting
LoG (image segmentation), morphological filtering and minimum centroid distance classifier [35]	vehicle counting and classification
YOLO with deep CNN to train network [48]	vehicle counting and classification
YOLO v3 with dataset trained with COCO [52]	vehicle counting and classification, route prediciton
Own algorithm based on the number of vehicles and their route [52]	Traffic management
Faster R-CNN - daytime - nighttime [42]	Vehicle detection night and daytime
Traffic flow theory [42]	Traffic flow estimation
HMPD [34]	detection and pedestrian counting

Figure A.2: Table with articles and respective algorithms and goals

Appendix B

Hardware Schemes

B.1 Electric Schematic

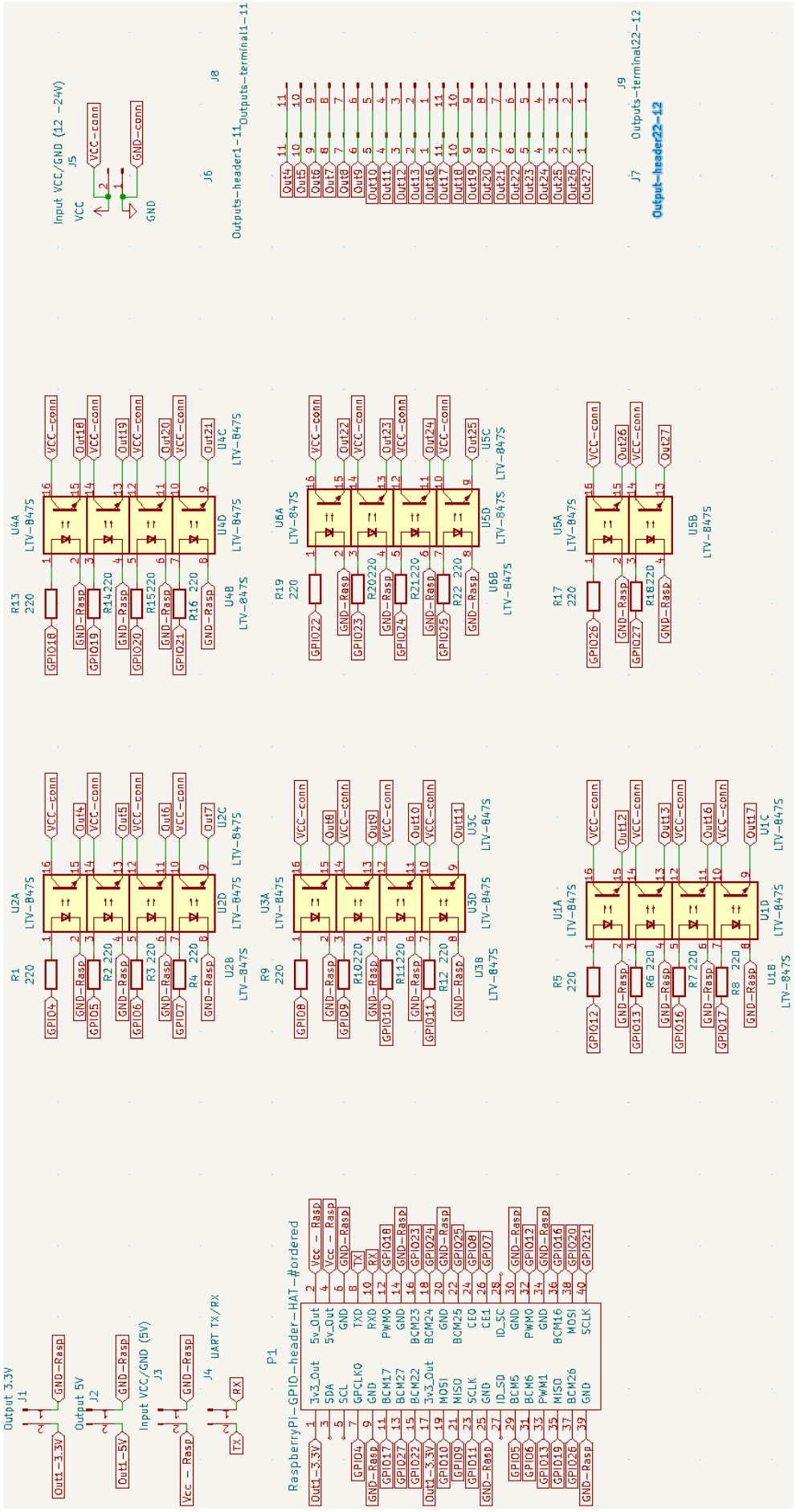


Figure B.1: Electrical scheme of positive logic board

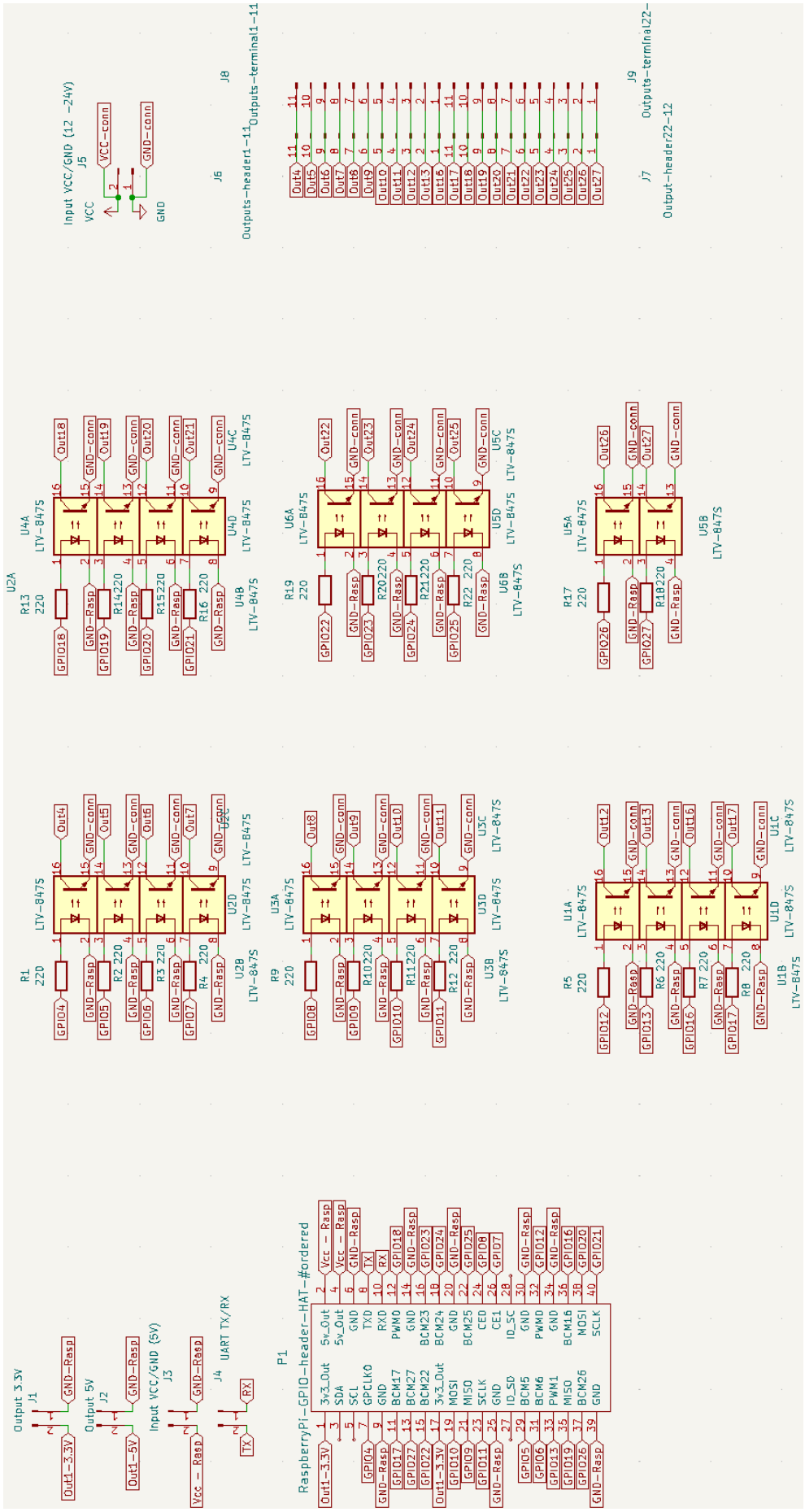


Figure B.2: Electrical scheme of negative logic board

B.2 PCB Layout

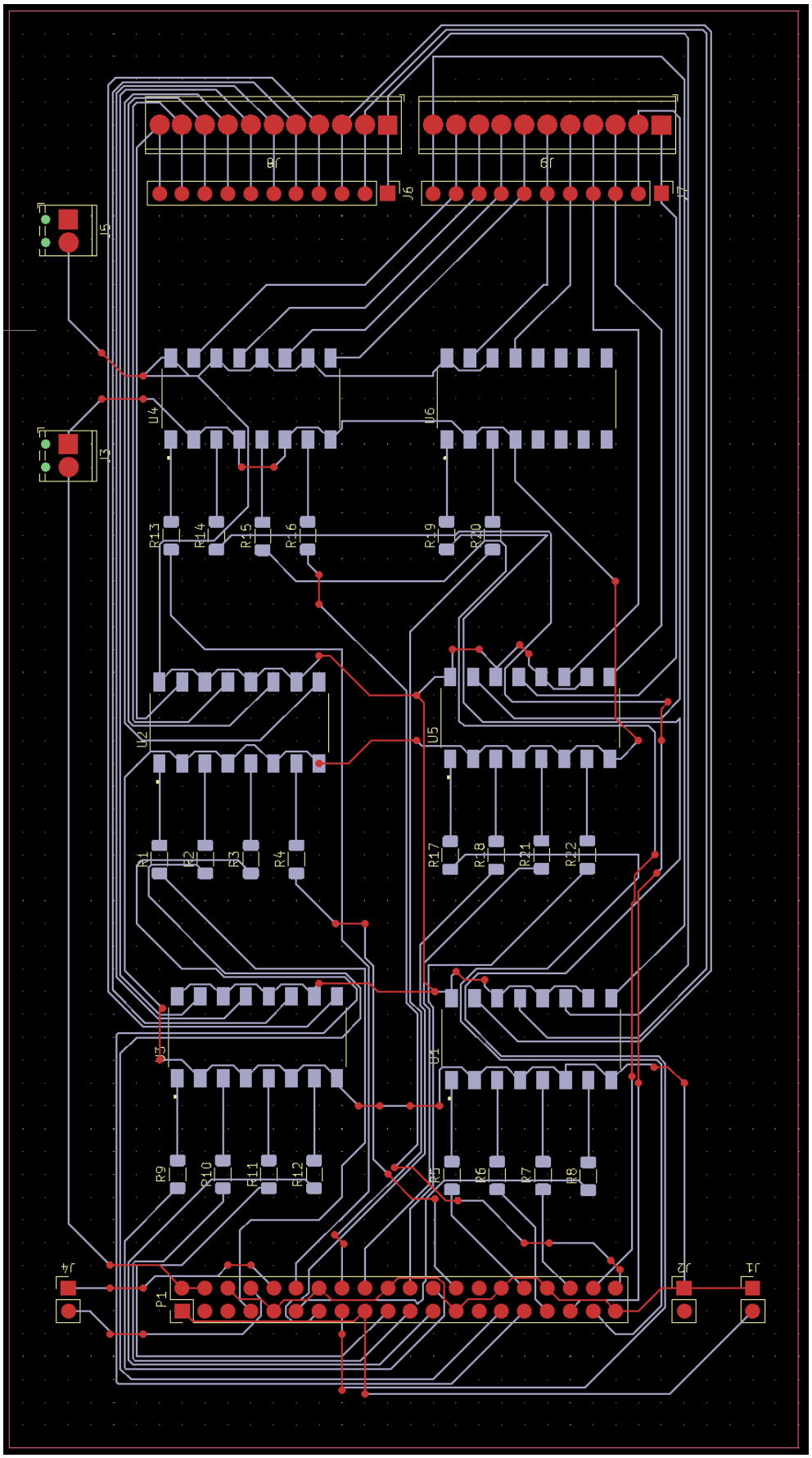


Figure B.3: PCB layout in Gerber format of positive logic board

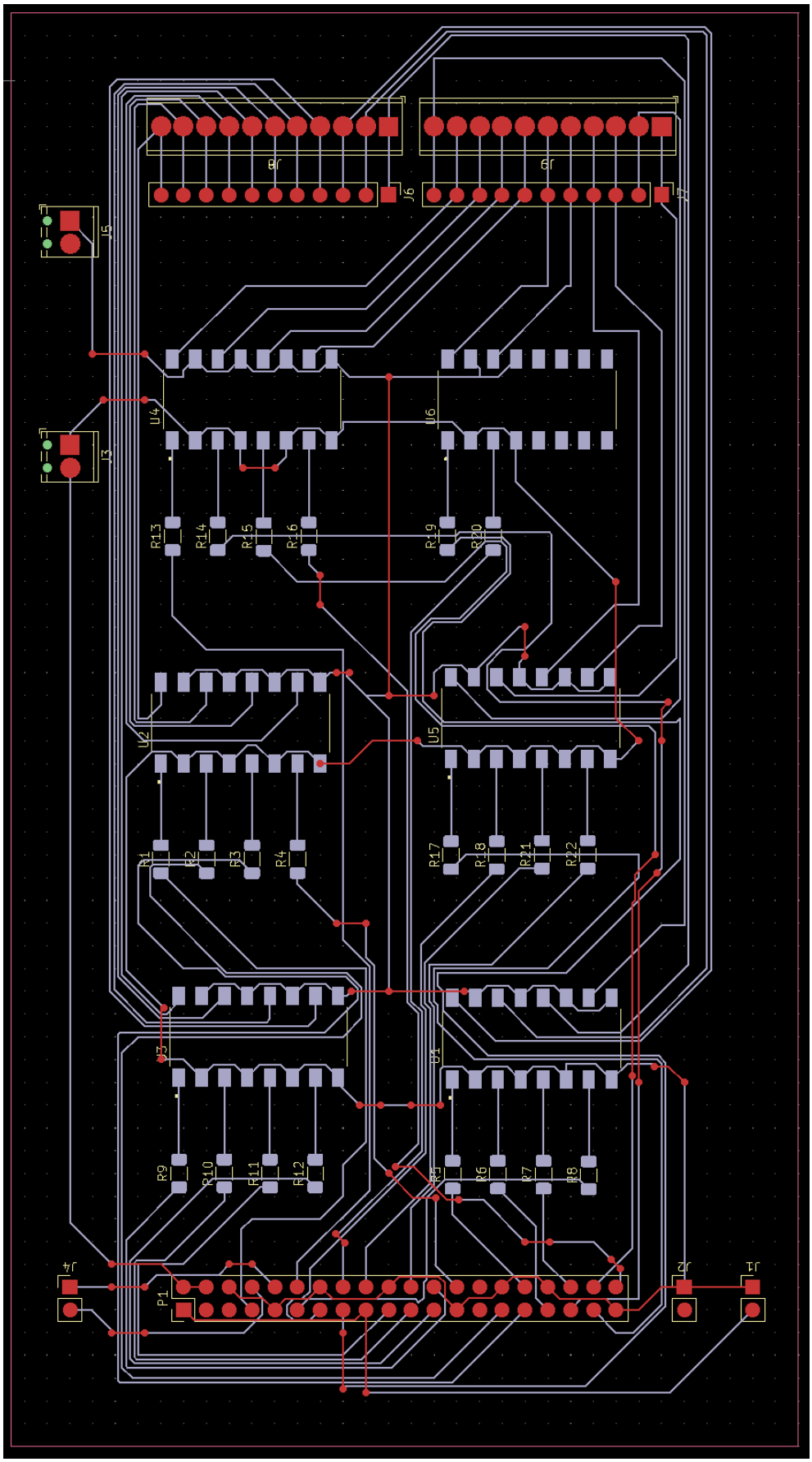


Figure B.4: PCB layout in Gerber format of negative logic board

Appendix C

MQTT topics

```
T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Truck 1
T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Truck 1

T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Car 1
T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Car 2

T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Bike 1
T1/onvif-ej/RuleEngine/CountAgregation/Counter/&1/Counter Bike 2


T2/onvif-ej/IVA/IdleObject/&1/DoublePark Car 1
T2/onvif-ej/IVA/IdleObject/&1/DoublePark Car 2

T2/onvif-ej/IVA/IdleObject/&1/DoublePark Truck 1
T2/onvif-ej/IVA/IdleObject/&1/DoublePark Truck 2

T2/onvif-ej/IVA/IdleObject/&1/DoublePark Bike 1
T2/onvif-ej/IVA/IdleObject/&1/DoublePark Bike 2


T2/onvif-ej/IVA/IdleObject/&1/Idle Car 1
T2/onvif-ej/IVA/IdleObject/&1/Idle Car 2

T2/onvif-ej/IVA/IdleObject/&1/Idle Truck 1
T2/onvif-ej/IVA/IdleObject/&1/Idle Truck 2

T2/onvif-ej/IVA/IdleObject/&1/Idle Bike 1
T1/onvif-ej/IVA/IdleObject/&1/Idle Bike 2
```

```
T2/onvif-ej/IVA/ObjectInField/&1/Jam 1
T1/onvif-ej/IVA/ObjectInField/&1/Jam 2


T1/onvif-ej/IVA/CrowdDetection/&1/Crowd Detection
```

Appendix D

REST API Logs

D.1 Interaction with Settings Page

```
127.0.0.1 - - [12/Jun/2022 22:03:35] "OPTIONS /login HTTP/1.1" 200 -
{'username': 'daniel', 'password':'\$2y\$10\$wwqxw0oo7xnzGruQROqZ/5aNkH9ai1Lg'}
logged in!
Start session!
127.0.0.1 - - [12/Jun/2022 22:03:35] "POST /login HTTP/1.1" 302 -
127.0.0.1 - - [12/Jun/2022 22:03:35] "GET / HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 3, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:03:45] "GET /sensors HTTP/1.1" 200 -
[GET BROKER]: 192.168.1.199
127.0.0.1 - - [12/Jun/2022 22:03:45] "GET /settings-broker HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:04:03] "OPTIONS /settings HTTP/1.1" 200 -
Password updated!
127.0.0.1 - - [12/Jun/2022 22:04:03] "PATCH /settings HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:04:05] "OPTIONS /settings-broker HTTP/1.1" 200 -
Broker updated: 192.168.1.199
127.0.0.1 - - [12/Jun/2022 22:04:05] "PATCH /settings-broker HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:04:14] "OPTIONS /sensors HTTP/1.1" 200 -
{'name': 'T3', 'lanes': 4}
Sensor inserted! Sensor {'name': 'T3', 'lanes': 4}
127.0.0.1 - - [12/Jun/2022 22:04:14] "POST /sensors HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:04:28] "OPTIONS /sensors?id=2 HTTP/1.1" 200 -
2
Sensor updated! Sensor: {'name': 'T2', 'lanes': 2}
127.0.0.1 - - [12/Jun/2022 22:04:28] "PATCH /sensors?id=2 HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2},
                {'name': 'T3', 'lanes': 4, 'id': 3}]
127.0.0.1 - - [12/Jun/2022 22:04:30] "GET /sensors HTTP/1.1" 200 -
[GET BROKER]: 192.168.1.199
```

```
127.0.0.1 - - [12/Jun/2022 22:04:30] "GET /settings-broker HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:04:34] "OPTIONS /sensors?id=3 HTTP/1.1" 200 -
Sensor deleted! id: 3
127.0.0.1 - - [12/Jun/2022 22:04:34] "DELETE /sensors?id=3 HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:04:35] "GET /sensors HTTP/1.1" 200 -
[GET BROKER]: 192.168.1.199
127.0.0.1 - - [12/Jun/2022 22:04:35] "GET /settings-broker HTTP/1.1" 200 -
```

D.2 Interaction with Processes Page

```
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:41:55] "GET /process HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:41:55] "GET /pins HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:41:57] "GET /pins HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:41:58] "GET /pins HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:41:58] "GET /pins HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:41:58] "GET /sensors HTTP/1.1" 200 -
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:42:05] "GET /process HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:42:05] "GET /sensors HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:42:10] "GET /sensors HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:12] "GET /pins HTTP/1.1" 200 -
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:42:15] "GET /process HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:19] "OPTIONS /process HTTP/1.1" 200 -
Process inserted!
127.0.0.1 - - [12/Jun/2022 22:42:19] "POST /process HTTP/1.1" 200 -
```

```
127.0.0.1 - - [12/Jun/2022 22:42:21] "OPTIONS /process?id=0 HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:21] "PATCH /process?id=0 HTTP/1.1" 200 -
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:42:24] "GET /process HTTP/1.1" 200 -
[GET PINS]: [1, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:24] "GET /pins HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:26] "OPTIONS /process?id=7 HTTP/1.1" 200 -
Process deleted!7
127.0.0.1 - - [12/Jun/2022 22:42:26] "DELETE /process?id=7 HTTP/1.1" 200 -
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:42:27] "GET /process HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:27] "GET /pins HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:29] "OPTIONS /process?id=3 HTTP/1.1" 200 -
Process updated! id: 3
127.0.0.1 - - [12/Jun/2022 22:42:29] "PATCH /process?id=3 HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:30] "GET /pins HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:30] "GET /pins HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:42:30] "GET /sensors HTTP/1.1" 200 -
[GET SENSORS]: [{'name': 'T1', 'lanes': 2, 'id': 1},
                {'name': 'T2', 'lanes': 2, 'id': 2}]
127.0.0.1 - - [12/Jun/2022 22:42:30] "GET /sensors HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:30] "GET /pins HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:32] "OPTIONS /process?id=6 HTTP/1.1" 200 -
Process updated! id: 6
127.0.0.1 - - [12/Jun/2022 22:42:32] "PATCH /process?id=6 HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2022 22:42:36] "OPTIONS /process?id=2 HTTP/1.1" 200 -
Process updated! id: 2
127.0.0.1 - - [12/Jun/2022 22:42:36] "PATCH /process?id=2 HTTP/1.1" 200 -
[GET PROCESSES]: Process returned!
127.0.0.1 - - [12/Jun/2022 22:42:37] "GET /process HTTP/1.1" 200 -
[GET PINS]: [1, 4, 7, 8, 9, 10, 11, 12, 13, 14,
15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26] Pins returned!
127.0.0.1 - - [12/Jun/2022 22:42:37] "GET /pins HTTP/1.1" 200 -
```

Appendix E

Process Trigger Script Logs

```
PROCESS Car detection is OFF
TURN OFF Car detection PINS: [3]

PROCESS Traffic flow is OFF
TURN OFF Traffic flow PINS: [2]

Verifying Double-Park process...
RISING EDGE
PROCESS START TRIGGERING: Double-Park
 TURN ON NOT INVERTED PINS [5]
 TURN OFF INVERTED PINS [25]

 Verifying CrowdDetection  process...
 RISING EDGE
 PROCESS START TRIGGERING: CrowdDetection
 TURN ON NOT INVERTED PINS [6]
 TURN OFF INVERTED PINS []

 PROCESS Car detection is OFF
 TURN OFF Car detection PINS: [3]

 PROCESS Traffic flow is OFF
 TURN OFF Traffic flow PINS: [2]

 Verifying Double-Park process...
 PROCESS Double-Park IS STILL TRUE

 Verifying CrowdDetection  process...
 PROCESS CrowdDetection IS STILL TRUE

 PROCESS Car detection is OFF
 TURN OFF Car detection PINS: [3]
```

```
33
34 PROCESS Traffic flow is OFF
35 TURN OFF Traffic flow PINS: [2]
36
37 Verifying Double-Park process...
38 PROCESS Double-Park IS STILL TRUE
39
40 Verifying CrowdDetection  process...
41 PROCESS CrowdDetection IS STILL TRUE
42
43 PROCESS Car detection is OFF
44 TURN OFF Car detection PINS: [3]
45
46 PROCESS Traffic flow is OFF
47 TURN OFF Traffic flow PINS: [2]
48
49 Verifying Double-Park process...
50 PROCESS Double-Park IS STILL TRUE
51
52 Verifying CrowdDetection  process...
53 PROCESS CrowdDetection IS STILL TRUE
54
```

Appendix F

Requirements Matrix

Requirement	Description	Importance	fulfilled
SYS_1001	Project cost must be low-cost	Critical	Yes
SYS_1002	System execution must be in real time	Critical	Yes
ELE_2001	Final product must be reflected in a hardware module	Critical	Yes
ELE_2002	Module must integrate a digital interface - Ethernet/RS232	Critical	Yes
ELE_2003	Module must integrate I/O interface - GPIOS	Critical	Yes
ELE_2004	Module being able to run embedded OS	Critical	Yes
ELE_2005	Module must provide integrated memory for data recording	Critical	Yes
ELE_2006	There should be a board that converts the signal from the GPIO pins to the voltage range of the semaphore controllers	Worthwhile	No
SW_3001	Module must integrate date from a CCTV camera	Critical	Yes
SW_3002	System should be able to integrate data from CCTV cameras at the same time	Worthwhile	Yes
SW_3003	It must be possible to configure the name of the cameras as sensors that will communicate with the module through the web interface	Worthwhile	Yes
SW_3004	System should be able to communicate process triggering through the GPIOs interface	Critical	Yes
SW_3005	The system should incorporate an embedded web interface.	Critical	Yes
SW_3006	It should be possible to change the IP of the MQTT broker through the web interface	Worthwhile	Yes
SW_3007	It should be possible to change the admin password through the web interface	Worthwhile	Yes
SW_3008	It must be possible to configure processes with name, enable state, GPIOs and logical conditions associated	Critical	Yes

SW_3009	It should be possible to associate a time interval to the execution of the processes	Worthwhile	Yes
SW_3010	It must be possible to create logical conditions with logical and arithmetic operators	Critical	Yes
SW_3011	It should be possible to create logical conditions with compound variables	Worthwhile	Yes
SW_3012	It must be possible to associate the following traffic situations to the conditions: vehicle count and classifications per lane, vehicle stay time per zone and traffic flow per zone and vehicle type	Critical	Yes
SW_3013	It should be possible to associate the following traffic situations to the logic conditions: detection of queues per lane, detection of double-parked vehicles, detection of pedestrians at crossings.	Worthwhile	Yes
SW_3014	It should be possible to associate more than one digital pin to each process.	Worthwhile	Yes
SW_3015	It should be possible to individually invert the logic of the digital pins associated to the process	Worthwhile	Yes
SW_3016	Information about the configured processes should be constantly shown, such as the last time it triggered and the current status.	Worthwhile	Yes
SW_3017	There should be charts on the web interface that show information related to the integrated data	Worthwhile	Yes
SW_3018	The system must store information locally	Critical	Yes
SW_3019	There must be a security system for accessing the web interface	Worthwhile	Yes
SW_3020	The system must run an MQTT broker	Critical	Yes

Table F.1: Representation the simplified project requirements matrix

Bibliography

[1] M. Hamdan, O. O. Khalifah, and T. S. Gunawan, "Measuring the road traffic intensity using neural network with computer vision," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 10, no. 1, pp. 184–190, 2018. DOI: 10.11591/ijeecs.v10.i1.pp184-190.

[2] Bosch. "Flexidome ip starlight 8000i - 6mp." (), [Online]. Available: https://resources-boschsecurity-cdn.azureedge.net/public/documents/FLEXIDOME_IP_starlig_Data_sheet_enUS_68669610251.pdf. (accessed: 16.04.2022).

[3] T. Ide, T. Katsuki, T. Morimura, and R. Morris, "City-wide traffic flow estimation from a limited number of low-quality cameras," *Ieee Transactions on Intelligent Transportation Systems*, vol. 18, no. 4, pp. 950–959, 2017, ISSN: 1524-9050. DOI: 10.1109/tits.2016.2597160.

[4] M. W. Park, J. I. Kim, Y. J. Lee, J. Park, and W. Suh, "Vision-based surveillance system for monitoring traffic conditions," *Multimedia Tools and Applications*, vol. 76, no. 23, pp. 25 343–25 367, 2017, ISSN: 1380-7501. DOI: 10.1007/s11042-017-4521-4.

[5] S. Memon, S. Bhatti, L. A. Thebo, M. M. B. Talpur, and M. A. Memon, "A video based vehicle detection, counting and classification system," *International Journal of Image, Graphics and Signal Processing*, vol. 10, no. 9, pp. 34–41, 2018, ISSN: 2074-9074. DOI: 10.5815/ijigsp.2018.09.05.

[6] C. Y. Ma, D. Liu, X. L. Peng, L. Li, and F. Wu, "Traffic surveillance video coding with libraries of vehicles and background," *Journal of Visual Communication and Image Representation*, vol. 60, pp. 426–440, 2019, ISSN: 1047-3203. DOI: 10.1016/j.jvcir.2019.03.009.

[7] S. Jun-fang, "Vehicle detection using spatial relationship gmm for complex urban surveillance in daytime and nighttime," *International Journal of Parallel Programming*, vol. 46, no. 5, pp. 859–72, 2018, ISSN: 0885-7458. DOI: 10.1007/s10766-017-0543-9.

[8] C. Tang, Y. Dong, X. Lin, and W. Xiao, "Multi-field depth vehicle headlight detection by model construction and long trajectory extraction in nighttime city traffic," in *Smart Innovation, Systems and Technologies*, vol. 62, 2017, pp. 234–246. DOI: 10.1007/978-981-10-3575-3_24.

[9] J. L. Li, Z. G. Xu, L. Fu, X. S. Zhou, and H. K. Yu, "Domain adaptation from daytime to nighttime: A situation-sensitive vehicle detection and traffic flow parameter estimation framework," *Transportation Research Part C-Emerging Technologies*, vol. 124, 2021, Li, Jinlong Xu, Zhigang Fu, Lan Zhou, Xuesong Yu, Hongkai 1879-2359, ISSN: 0968-090X. DOI: 10.1016/j.trc.2020.102946.

[10] H. Synh Viet-Uyen, T. Duong Nguyen-Ngoc, N. Tien Phuoc, and D. Son Vu-Truong, "High variation removal for background subtraction in traffic surveillance systems," *IET Computer Vision*, vol. 12, no. 8, pp. 1163–70, 2018, ISSN: 1751-9632. DOI: 10.1049/iet-cvi.2018.5033.

[11] M. J. Sun, Y. Wang, T. Li, J. Lv, and J. Wu, "Vehicle counting in crowded scenes with multi-channel and multi-task convolutional neural networks," *Journal of Visual Communication and Image Representation*, vol. 49, pp. 412–419, 2017, ISSN: 1047-3203. DOI: 10.1016/j.jvcir.2017.10.002.

[12] M. Hamdan, O. O. Khalifah, and T. S. Gunawan, "Measuring the road traffic intensity using neural network with computer vision," *Indonesian Journal of Electrical Engineering and Computer Science*, vol. 10, no. 1, pp. 184–190, 2018. DOI: 10.11591/ijeecs.v10.i1.pp184-190.

[13] B. Dey and M. K. Kundu, "Turning video into traffic data - an application to urban intersection analysis using transfer learning," *Iet Image Processing*, vol. 13, no. 4, pp. 673–679, 2019, ISSN: 1751-9659. DOI: 10.1049/iet-ipr.2018.5985.

[14] M. M. Iqbal, M. T. Mehmood, S. Jabbar, S. Khalid, A. Ahmad, and G. Jeon, "An enhanced framework for multimedia data: Green transmission and portrayal for smart traffic system," *Computers & Electrical Engineering*, vol. 67, pp. 291–308, 2018, ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2018.03.021.

[15] A. Fedorov, K. Nikolskaia, S. Ivanov, V. Shepelev, and A. Minbaleev, "Traffic flow estimation with data from a video surveillance camera," *Journal of Big Data*, vol. 6, no. 1, 2019. DOI: 10.1186/s40537-019-0234-z.

[16] J. D. Wu, B. Y. Chen, W. J. Shyr, and F. Y. Shih, "Vehicle classification and counting system using yolo object detection technology," *Traitement Du Signal*, vol. 38, no. 4, pp. 1087–1093, 2021, ISSN: 0765-0019. DOI: 10.18280/ts.380419.

[17] A. Goyal, M. Singh, and A. Aeron, "Simulation of traffic optimization to reduce congestion," *International Journal of Innovative Technology and Exploring Engineering*, vol. 8, no. 11, pp. 3780–3783, DOI: 10.35940/ijitee.K2122.0981119.

[18] S. H. Zhou, S. T. Ng, Y. F. Yang, and J. F. Xu, "Integrating computer vision and traffic modeling for near-real-time signal timing optimization of multiple intersections," *Sustainable Cities and Society*, vol. 68, 2021, ISSN: 2210-6707. DOI: 10.1016/j.scs.2021.102775.

[19] P. R. Iyer, S. R. Iyer, R. Ramesh, M. R. Anala, and K. N. Subramanya, "Adaptive real time traffic prediction using deep neural networks," *IAES International Journal of Artificial Intelligence*, vol. 8, no. 2, pp. 107–119, 2019. DOI: 10.11591/ijai.v8.i2.pp107-119.

[20] G. Jian and H. Tembine, "Distributed mean-field-type filters for traffic networks," *IEEE Transactions on Intelligent Transportation Systems*, vol. 20, no. 2, pp. 507–21, 2019, ISSN: 1524-9050. DOI: 10.1109/TITS.2018.2816811.

[21] L. Rosas-Arias, J. Portillo-Portillo, A. Hernandez-Suarez, *et al.*, "Vehicle counting in video sequences: An incremental subspace learning approach," *Sensors*, vol. 19, no. 13, 2848 (16 pp.) 2019, ISSN: 1424-8220. DOI: 10.3390/s19132848.

[22] E. Ua-areemitr, A. Sumalee, and W. H. K. Lam, "Low-cost road traffic state estimation system using time-spatial image processing," *Ieee Intelligent Transportation Systems Magazine*, vol. 11, no. 3, pp. 69–79, 2019, ISSN: 1939-1390. DOI: 10.1109/mits.2019.2919634.

[23] B. G. Rajagopal, "Intelligent traffic analysis system for indian road conditions," *International Journal of Information Technology (Singapore)*, 2020, Export Date: 20 December 2021. DOI: 10.1007/s41870-020-00447-3.

[24] Q. Wang, J. Wan, and X. L. Li, "Robust hierarchical deep learning for vehicular management," *Ieee Transactions on Vehicular Technology*, vol. 68, no. 5, pp. 4148–4156, 2019, ISSN: 0018-9545. DOI: 10.1109/tvt.2018.2883046.

[25] Z. Shanghang, W. Guanhang, J. P. Costeira, and J. M. F. Moura, "Fcn-rlstm: Deep spatio-temporal neural networks for vehicle counting in city cameras [arxiv]," *arXiv*, 10 pp. 2017.

[26] K. R. S. Preethaa and A. Sabari, "Intelligent video analysis for enhanced pedestrian detection by hybrid metaheuristic approach," *Soft Computing*, vol. 24, no. 16, pp. 12 303–11, 2020, ISSN: 1432-7643. DOI: 10.1007/s00500-020-04674-5.

[27] G. Shobana and M. Suguna, "Road traffic monitoring system," *International Journal of Engineering and Advanced Technology*, vol. 8, no. 6 Special Issue 3, pp. 1249–1251, 2019, Export Date: 20 December 2021. DOI: 10.35940/ijeat.F1215.0986S319.

[28] M. H. Alkinani, A. A. Almazroi, M. Adhikari, and V. G. Menon, "Artificial intelligence-empowered logistic traffic management system using empirical intelligent xgboost technique in vehicular edge networks," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, 2022, ISSN: 1524-9050 1558-0016. DOI: 10.1109/TITS.2022.3145403.

[29] J. Sharifi and N. Saeednia, "Neuro-fuzzy modeling of data singular spectrum decomposition and traffic flow prediction," *IRANIAN JOURNAL OF SCIENCE AND TECHNOLOGY-TRANSACTIONS OF ELECTRICAL ENGINEERING*, vol. 44, no. 1, pp. 519–535, 2020, ISSN: 2228-6179 2364-1827. DOI: 10.1007/s40998-019-00227-1.

[30] M. Kaluza, K. Troskot, and B. Vukelic, "Comparison of front-end frameworks for web applications development," *ZBORNIK VELEUCILISTA U RIJECI-JOURNAL OF THE POLYTECHNICS OF RIJEKA*, vol. 6, no. 1, pp. 261–282, 2018, ISSN: 1848-1299 1849-1723. DOI: 10.31784/zvr.6.1.19.

[31] S. Chavhan and P. Venkataram, "Prediction based traffic management in a metropolitan area," *JOURNAL OF TRAFFIC AND TRANSPORTATION ENGINEERING-ENGLISH EDITION*, vol. 7, no. 4, pp. 447–466, 2020, ISSN: 2095-7564. DOI: 10.1016/j.jtte.2018.05.003.

[32] V. Shepelev, S. Aliukov, K. Nikolskaya, and S. Shabiev, "The capacity of the road network: Data collection and statistical analysis of traffic characteristics," *ENERGIES*, vol. 13, no. 7, 2020, ISSN: 1996-1073. DOI: 10.3390/en13071765.

[33] X. S. Li, P. J. Ye, J. C. Jin, F. H. Zhu, and F. Y. Wang, "Data augmented deep behavioral cloning for urban traffic control operations under a parallel learning framework," *IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS*, 2021, ISSN: 1524-9050 1558-0016. DOI: 10.1109/TITS.2020.3048151.

[34] M. S. Sheikh and A. Regan, "A complex network analysis approach for estimation and detection of traffic incidents based on independent component analysis," *PHYSICA A-STATISTICAL MECHANICS AND ITS APPLICATIONS*, vol. 586, 2022, ISSN: 0378-4371 1873-2119. DOI: 10.1016/j.physa.2021.126504.

[35] S. Frey. "Best raspberry pi alternatives of 2021." (), [Online]. Available: https://all3dp.com/1/best-raspberry-pi-alternatives/. (accessed: 18.02.2022).

[36] Amazon. "Amazon." (), [Online]. Available: https://www.amazon.com/Odroid-Xu4/s?k=Odroid+Xu4. (accessed: 20.02.2022).

[37] S. overflow. "Questions tagged [raspberry-pi]." (), [Online]. Available: https://stackoverflow.com/questions/tagged/raspberry-pi. (accessed: 18.02.2022).

[38] R. Pi. "Raspberry pi 4." (), [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. (accessed: 18.02.2022).

[39] NVIDIA. "Jetson xavier nx series." (), [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/. (accessed: 19.02.2022).

[40] Hardkernel. "Odroid-n2+ with 4gbyte ram." (), [Online]. Available: https://www.hardkernel.com/shop/odroid-n2-with-4gbyte-ram-2/. (accessed: 19.02.2022).

[41] ——, "Odroid-xu4 special price." (), [Online]. Available: https://www.hardkernel.com/shop/odroid-xu4-special-price/. (accessed: 19.02.2022).

[42] ASUS. "Tinker board." (), [Online]. Available: https://www.asus.com/pt/Networking-IoT-Servers/AIoT-Industrial-Solutions/All-series/Tinker-Board/. (accessed: 19.02.2022).

[43] LattePanda. "Lattepanda delta 432." (), [Online]. Available: https://www.lattepanda.com/products/lattepanda-delta-432.html. (accessed: 19.02.2022).

[44] PINE64. "Rock64." (), [Online]. Available: https://www.pine64.org/devices/single-board-computers/rock64/. (accessed: 19.02.2022).

[45] L. Computer. "Aml-s905x-cc (le potato)." (), [Online]. Available: https://libre.computer/products/s905x/. (accessed: 19.02.2022).

[46] B. P. Wiki. "Banana pi bpi-m5." (), [Online]. Available: https://wiki.banana-pi.org/Banana_Pi_BPI-M5. (accessed: 19.02.2022).

[47] KHADAS. "Vim 2." (), [Online]. Available: https://www.khadas.com/vim2. (accessed: 19.02.2022).

[48] BlakeTheDev. "Front-end frameworks." (), [Online]. Available: https://2021.stateofjs.com/en-US/libraries/front-end-frameworks/. (accessed: 18.02.2022).

[49] Y. K. Xing, J. Huang, Y. Y. Lai, and Acm, *Research and analysis of the front-end frameworks and libraries in e-business development*, Conference Paper, 2019. DOI: 10.1145/3313991.3314021.

[50] A. Pattakos. "14 best vue ui component libraries 2022." (), [Online]. Available: https://athemes.com/collections/vue-ui-component-libraries/. (accessed: 16.03.2022).

[51] Github. "Pypl popularity of programming language." (), [Online]. Available: https://pypl.github.io/PYPL.html. (accessed: 17.04.2022).

[52] R. Hat. "What is a rest api?" (), [Online]. Available: https://www.redhat.com/en/topics/api/what-is-a-rest-api. (accessed: 03.05.2022).

[53] J. Patel. "7 top python frameworks you should consider for your web project." (), [Online]. Available: https://www.monocubed.com/blog/top-python-frameworks/. (accessed: 07.04.2022).

[54] Slant. "What are the best python microframeworks?" (), [Online]. Available: https://www.slant.co/topics/532/~best-python-microframeworks. (accessed: 02.05.2022).

[55] S. Emms. "13 best free and open source python microframeworks." (), [Online]. Available: https://www.linuxlinks.com/best-free-python-microframeworks/. (accessed: 02.05.2022).

[56] I. Vosloo. "Popular non full-stack frameworks." (), [Online]. Available: https://wiki.python.org/moin/WebFrameworks. (accessed: 02.05.2022).

[57] Bosch. "How to set-up the camera's mqtt client in configuration manager - bosch onvif profile m?" (), [Online]. Available: https://community.boschsecurity.com/t5/Security-Video/How-to-set-up-the-camera-s-MQTT-client-in-Configuration-Manager/ta-p/49015. (accessed: 12.04.2022).

[58] M. DB. “Nosql vs relational databases.” (), [Online]. Available: https://www.mongodb.com/scale/nosql-vs-relational-databases. (accessed: 18.04.2022).

[59] Github. “Nosql.” (), [Online]. Available: https://github.com/topics/nosql. (accessed: 21.04.2022).

[60] M. Siemens. “Tinydb.” (), [Online]. Available: https://github.com/msiemens/tinydb. (accessed: 22.04.2022).

[61] M. David. “Litedb.” (), [Online]. Available: https://github.com/mbdavid/LiteDB. (accessed: 22.04.2022).

[62] Lite-on. “Photocoupler product data sheet ltv-816/ 826/ 846 (m, s, s-ta, s-ta1, s-tp, s-tp1) series.” (), [Online]. Available: https://br.mouser.com/datasheet/2/239/LTV_8X6_series-2887056.pdf. (accessed: 10.05.2022).

[63] Wikipedia. “Npm (software).” (), [Online]. Available: https://en.wikipedia.org/wiki/Npm_(software). (accessed: 9.04.2022).

[64] J. Agenjo. “Litegraph.js.” (), [Online]. Available: https://github.com/jagenjo/litegraph.js. (accessed: 27.04.2022).

[65] A. Lelievre. “Nodegraphprocessor.” (), [Online]. Available: https://github.com/alelievr/NodeGraphProcessor. (accessed: 27.04.2022).

[66] Rete. “The javascript framework for visual programming.” (), [Online]. Available: https://rete.js.org/#/#lang=en&tosearch=The\%20JavaScript\%20framework\%20for\%20visual\%20programming. (accessed: 15.04.2022).

[67] TinyDB. “Tinydb - advanced usage.” (), [Online]. Available: https://tinydb.readthedocs.io/en/latest/usage.html. (accessed: 22.04.2022).