UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

DEGREE IN COMPUTER SCIENCE

SPECIFIC TECHNOLOGY OF COMPUTATION

UNDERGRADUATE FINAL DISSERTATION

Scalable platform to support automatic number-plate recognition in urban environments

Alberto Aranda García

December, 2020

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Technologies and Information Systems Department

SPECIFIC TECHNOLOGY OF COMPUTATION

UNDERGRADUATE FINAL DISSERTATION

Scalable platform to support automatic number-plate recognition in urban environments

Author: Alberto Aranda García

Supervisor: David Vallejo Fernández

Supervisor: Carlos González Morcillo

December, 2020

Alberto Aranda García

Ciudad Real – Spain

E-mail: Alberto.Aranda3@alu.uclm.es

© 2020 Alberto Aranda García

TRIBUNAL:

Presidente:

Vocal:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE   **VOCAL**   **SECRETARIO**

Fdo.:   Fdo.:   Fdo.:

Resumen

El presente trabajo surge de la necesidad de investigadores de la Escuela de Canales, Caminos y Puertos de Ciudad Real de realizar despliegues masivos de cámaras de tráfico que, gracias a la aplicación del reconocimiento automático de matrículas (ANPR), proporcionen datos sobre los que llevar a cabo análisis de tráfico urbano. En el seno de Furious Koalas Interactive, una spin-off de la Universidad de Castilla-La Mancha, se ha diseñado e implementado una plataforma escalable que facilita la configuración y despliegue de estos dispositivos. Con una interfaz web *responsive*, la plataforma permite tanto a operadores de cámaras de tráfico como a supervisores definir experimentos de monitorización de tráfico y seguir su estado en tiempo real, permitiendo modificaciones en los parámetros de las cámaras durante el transcurso de los experimentos. Una capa de integración gestiona la orquestación del proceso ANPR y la comunicación con las cámaras de tráfico, siendo estas últimas también gestionadas por la plataforma. Por último, la arquitectura de la plataforma se ha diseñado sobre servicios gestionados de Google Cloud Platform, con especial énfasis en el servicio AppEngine, dotando al sistema de capacidad para escalar a medida que se incorporan usuarios y/o dispositivos de captura.

Abstract

This work is born from the need of researchers at the School of Civil Engineering (Ciudad Real) to massively deploy traffic cameras which, thanks to the application of automatic number plate recognition (ANPR), provide data allowing to perform urban traffic analysis. Within Furious Koalas Interactive, a University of Castilla-La Mancha spin-off, a scalable platform allowing the configuration and deployment of these devices was designed and implemented. With a responsive web interface, the platform allows both traffic camera operators and supervisors to define traffic monitoring experiments and follow their state in real-time, allowing modifications in camera parameters during the course of the experiments. An integration layer manages the ANPR process orchestration and communication with traffic cameras, which are managed by the platform as well. Finally, the platform architecture has been designed over Google Cloud Platform managed services, with an emphasis on the AppEngine service, giving the system a capacity to scale as new users and/or capture devices join the platform.

Agradecimientos

Este trabajo representa el esperado final de una etapa crucial en mi vida, de la que guardo un gran recuerdo gracias a las personas con las que he tenido la suerte de compartirla.

En primer lugar, me gustaría agradecer a mis directores toda la ayuda prestada: a Carlos, por todos sus consejos durante el desarrollo del proyecto, y a David, porque sin su paciencia y guía no habría podido sacar adelante este trabajo.

A mis compañeros de Furious Koalas, por haberme hecho sentir tan acogido en todo momento y por todo lo que hemos compartido, en especial a Carlos, con quien desarrollar este proyecto ha sido una experiencia sumamente enriquecedora y amena.

A todos los profesores que he tenido, porque cada una de las cosas que me han enseñado forman parte de quien soy, y porque sin ellos no podría haber llegado aquí.

A mis amigos, por su buen humor, por su buen corazón, y por poder contar siempre con ellos.

A Lisa, por haber sabido entenderme, aguantarme, apoyarme, y no haber cesado nunca de darme ánimos cuando más falta hacían.

Y, por supuesto, a mi hermana Virginia, a mis padres, y a toda mi familia por enseñarme a no rendirme, por haberme educado, amado y acompañado en todas mis decisiones, buenas o malas.

A todos vosotros, gracias de corazón.

<div align="right">Alberto Aranda García</div>

The woods are lovely, dark and deep,
But I have promises to keep,
And miles to go before I sleep,
And miles to go before I sleep.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

As Information Technology continues advancing as a fundamental part of modern society, we are becoming more and more used to the changes it is bringing to the ways we work, learn, discover, consume, and entertain ourselves. However, the popularization of IT does not stop there: as Computer Science advances, especially in the field of Artificial Intelligence (AI), the ways in which it can contribute to research fields of all sorts are becoming more and more apparent. Urban Traffic Analysis is one of them, with multiple techniques from fields such as Computer Vision being applied to it [BVO11].

This project was born from the collaboration between the **Escuela de Ingenieros de Caminos, Canales y Puertos de Ciudad Real** (School of Civil Engineering) and **Furious Koalas Interactive**, a University of Castilla-La Mancha spin-off, with the objective of addressing the need to obtain data about the traffic network of Ciudad Real. The previous process to obtain it consisted of manually analyzing traffic video to extract the visible vehicle license plates and annotate them. The work put into this project will turn it into an improved, scalable, automatic process, that will help researchers and can be applied in other contexts.

1.1 Motivation and problematic

The consolidation of the automobile as the main mean of transportation in developed countries is easily explained: it provides both a point-to-point, temporal flexibility and overall comfort that is very appealing to travelers. However, urban transportation problems are evident in countries regardless of their development level, with issues such as traffic congestion seeming intractable both to policymakers and researchers. The monitoring of urban traffic is a starting point to tackle this and other related issues, from the management of the daily traffic flow in a defined area to the design of urban mobility plans.

Despite today's renewed interest in traffic flows and congestion, there is evidence of policy being implemented to address these issues since ancient civilizations. Going back to the Roman Empire, one of its most notorious features is its well-developed road system. There is, however, evidence that this infrastructure has evolved over time: while older and smaller cities such as Pompeii have narrower streets, which was in many cases not suitable

for wheeled traffic, later coloniae as found in Western Europe show streets wide enough to accommodate wheeled two-way traffic [vT11]. Another evidence of the way policy against traffic congestion was implemented through infrastructure improvement is the fact that the width of city gates found in ancient archaeological sites has been found to be positively correlated not only to their estimated population, but to their estimated traffic flow [Han20].

Nowadays, traffic congestion is one of the main challenges of modern society. The time, monetary and environmental costs of this worldwide issue are notorious, but still difficult to assess. Traffic jams deteriorate the quality of urban life and lead to economic loss. Research has tried to quantify the costs of congestion: a 1994 study conducted in the United States gives an estimated figure of $640 per driver and year [AS94]. Other studies such as the Urban Mobility Report [SLE19] give a total estimated cost of $166 billion in 2017, based on fuel consumption and hours spent on traffic. These figures, however, do not take into account the costs of accidents, air pollution and its effects on health and the environment, nor the negative economic effect of the unpredictability of traffic delays.

From a policy-making perspective, the importance of understanding traffic patterns and behaviors is essential, as decision-making in infrastructure build and maintenance depends on knowledge about the needs of road users. While the analysis of these needs is complex, as they depend on socio-economic and environmental factors (see [KDDT13]) and tend to vary over time, a good understanding of the current situation through a traffic model is an excellent starting point.

Traffic monitoring is the first step towards the building of a traffic model. For decades, different techniques and tools have emerged in order to perform urban traffic data gathering, with a transition from traditional manual recording to more sophisticated, automated techniques, which are proving to have not only lowered costs, but also increased performance and accuracy. Essentially, the latter techniques consist on the placement of sensors in the traffic network. The work in [PSK+10] gives a summary of the different kinds of available sensors, classifying them in intrusive and non-intrusive sensors. Intrusive sensors need to be installed "*under the road pavement, in saw-cuts or holes on the roads*". They stand out in accuracy for vehicle detection; however, their installation and maintenance cost is higher. Examples of intrusive sensors include inductive loops, piezoelectric cables, and magnetometers. On the other hand, non-intrusive sensors are located above ground, above the road or on its side. Examples of such sensors are microwave and laser radar, infrared, ultrasonic, or sensors video image processing (VIP) methods. This project will be focused on the latter kind of methods. Overall, non-invasive sensors overcome the disadvantages of invasive sensors, but as a downside, they have a larger size and are generally more power-hungry.

In the field of video image processing methods, much effort has been put to detect vehicles in urban traffic footage. For instance, the work in [GMMP02] describes some algorithms for the detection and classification of vehicles based on images captured from traffic. As

these classification methods add information about the kind of vehicles that traverse the network, they suppose a step forward compared to sensors which only provide vehicle counts. However, a bigger degree of observability can be reached with methods based on automatic number plate recognition (ANPR), with some studies in the field of urban traffic modeling ([CMJ08], [SCJRG17]) showing how the additional license plate data provided by ANPR can be used.

While the implementation of ANPR systems for traffic surveillance is on the rise, its costs remain elevated: for example, in [E$^+$08], the hardware is estimated to cost \$20,000 per camera, while the installation and maintenance costs are estimated to cost \$4,000. This raises the need to research alternative architectures and platforms for the deployment of ANPR systems for traffic monitoring.

1.2 Document structure

The undergraduate final dissertation regulations from the Escuela Superior de Informática of the Universidad de Castilla-La Mancha proposes a structure which, adapted to the project content, is the following:

Chapter 2: Objectives

General description of the project and specific objectives or sub-objectives of the platform.

Chapter 3: Background

This chapter provides an overview of the research fields involved in any way in the system development, and makes a comparative study of the existing technologies to choose the most viable one in every case.

Chapter 4: Method and work phases

This chapter describes the work methodology to be followed, detailing each work phase. The used resources, both software and hardware, will be described, along with the project limitations.

Chapter 5: Architecture

The different modules composing the platform will be described, following a *top-down* perspective, which shows a broad view of the system, continuing with the subsystem analysis in an increasingly exhaustive way.

Chapter 6: Results

The evolution that the project has undergone will be described, detailing the iterations and experimental tests of the system.

Chapter 7 : Conclusions

This chapter describes the conclusions reached after the execution of the project, addressing possible improvements and new applications as well.

Chapter 2

Objectives

Given the approach and the ideas developed in the introduction, this chapter describes the work to be done, describing first the general objective, which is then derived into a set of detailed sub-objectives. This way, a set of milestones guiding the work towards its final objective is explained.

2.1 General objective

The project's general objective is the development of a cloud platform managing the deployment of urban traffic analysis experiments using low-cost capture devices and the ALPR technique. The platform must provide the means to deploy, control and monitor capture devices over a wide urban area, orchestrate the ALPR process and store the results.

The development has been carried by the company **Furious Koalas, S.L.**, the final client being the **Escuela de Ingenieros de Caminos, Canales y Puertos de Ciudad Real**, which plans to use the platform in real-world experiments used to conduct research in the area of urban traffic analysis.

2.2 Specific objectives

Below, the specific sub-objectives of the project are presented.

- Support for the definition of experiments: Development of a module allowing the users to define an experiment and its characteristics, which will include its starting and ending date and hour, and the number and identification of capture devices. The configuration of each capture device will be defined through this module as well. It will include the frequency of delivery of images to the server, regions where it is expected to retrieve plates, and optional camera configuration parameters (for example, recording frames per second, ISO sensitivity or exposure time).

- Retrieval of data from the capture devices: The capture devices must be able to send images to the platform, via an authenticated endpoint. Reliability must be taken into account, as in the event of the platform suffering from downtime, data from the capture devices may be lost.

- Support for the ALPR process orchestration: The platform must support the delivery of images to an ALPR service or model, and collect the result. It is necessary to preprocess the images, authenticate with the ALPR service if necessary, and send the images periodically, in order not to saturate the service or model. Besides, network errors or other kinds of failure must be taken into account, retrying whenever necessary.

- Portability between devices: The interface must be accessible and usable by both desktop and mobile devices, as it will be used both by in-field camera operators and by supervisors of the whole process, who will use it from their offices. Therefore, responsiveness criteria must be taken into account.

- Real-time monitoring and configuration changes: As one of the features of the platform is the monitoring of experiments, it is necessary that the interface is updated in real-time with the latest image sent by each capture device. As the platform will be used by several users concurrently, and in order to keep the load in the server low, a solution where the server pushes changes to each client when updates arrive is preferred to a long polling approach.

  Real-time configuration changes related to capture devices must be sent to them as well. The objective is that the user gets feedback as soon as possible on the effect of these changes.

- Detection of errors at the capture devices: The platform should warn the user when a capture device is not working as expected (for example, the device is not communicating, the image seems invalid, or no plates have been recognized), so that corrective actions may be taken.

- History of captures: The user interface should offer the possibility to query the history of captures of each capture device, showing the image, timestamp of the capture and recognized plates. The history should load more and more items on demand, whenever the user scrolls down. The objective is to allow the user to have a perspective of the evolution of the experiment for each capture device.

- Ease of use: The platform is meant to ease, as much as possible, the deployment and monitoring of ALPR experiments at a big scale; therefore, it should respect usability criteria. The nature of the experiments should be taken into account, as they may have high costs associated and there may be only one opportunity to perform them, making the users of the platform subject to stress.

- Deployment of the system to the cloud: Considering the projected use case, as well as the possibility of a future commercial version, the platform should be ready to be deployed, as easily as possible, to the chosen Cloud service for its exploitation, with a minimal amount of configuration.

Chapter 3

Background

This chapter will focus on the foundations of this project. First, a brief state-of-the-art of Automatic Number Plate Recognition will be exposed. A general description of the different stages of this process will be given, among the different algorithms available and their different implementations through an analysis of the current literature. Some fundamental concepts of Computer Vision will be reviewed, focusing on their application on ANPR and how they are used to overcome its main challenges. Next, an overview of some of the available commercial and open-source ANPR will be given, on which we will base the decision for the solution to use in the project. Second, there will be a review of the Public Cloud, with an introduction to it and its different models and their advantages, followed by a review of the main Cloud providers, exposing their service offerings, how they evolved, and the value they provide to customers.

3.1 Automatic Number Plate Recognition

ANPR (Automatic number plate recognition) or Automate license plate recognition (ALPR) consists in the extraction of the information of the license plate of a vehicle from an image or a sequence of images [DISB12]. Historically, it has been applied, for instance, in security systems, vehicle access control, and parking lots. Today, it is used as well, for example, in electronic toll services and automatic traffic fining systems, motivated by the vast integration of Information Technologies (IT) into all areas of modern life. ANPR is, therefore, a practical application of Artificial Intelligence (AI) that is widely deployed. Concrete examples of everyday scenarios that involve ANPR systems are automated border crossing of vehicles in certain countries and access control of vehicles of authorized personnel to the premises of many companies.

The design and implementation of ANPR systems is an ongoing field of research in computer vision, which relates to areas such as neural networks, machine learning, and pattern recognition. Similarly, many commercial and open-source ANPR tools and platforms are available in the market. This section will explain the different stages of the ANPR process, exposing the algorithms and mathematical principles behind each one of them. Following that, an overview of some commercial ANPR systems will be given.

3.1.1 ANPR steps

According to [PSP13], the general process of the automatic plate number recognition process may be summarized in several well-defined steps. Each step involves a different set of algorithms and/or considerations, the steps being (1) vehicle image capture, (2) number plate detection, (3) character segmentation, and (4) character recognition. In addition, a last step that consists in error detection in the recognized plates can be considered.

Vehicle image capture

The first step in the ANPR process is the capture of images from the vehicles whose plates we want to recognize. The capture step, which can seem straightforward at first, has a critical impact in the successive stages of the process, the final result being highly dependant on the quality of the captured images. The task of correctly capturing images of moving vehicles in real-time is complex and subject to many variables of the environment, such as lighting, vehicle speed, and angle of capture. Depending on the capture device, it is possible to adjust camera settings, such as shutter speed and sensitivity, in order to adapt to the environment and produce captures with better quality. One kind of equipment that is very commonly used for ANPR captures is infrared-sensitive cameras, in combination with an infrared light source. The reason behind this is that vehicle number plates of most countries are made with materials that are infrared-reflective, which makes them highlighted in images taken by IR cameras, making those captures very good candidates for ANPR.
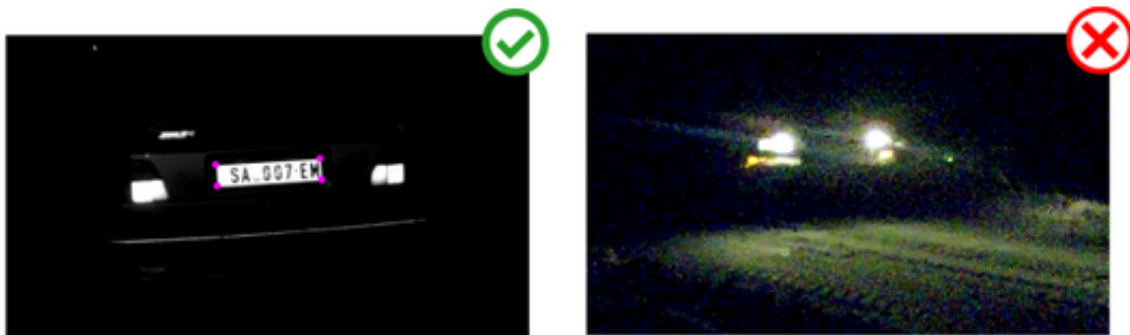
Figure 3.1: Left: Picture taken with an IR-capable camera. Right: Picture taken under poor lighting. Source: OpenANPR Documentation[2]

Number plate detection

Following the image capture, the next step in the ANPR process is the detection of the area in which a number plate is expected to be located. Images are stored in digital computers as matrices, each number representing the intensity of light of an individual pixel. An image can also be modelled as a discrete function $f(x, y)$, where x and y are spatial coordinates,

---

[2]http://doc.openalpr.com/camera_placement.html#lighting

and *f* returns the value for the intensity of light at that point. This modeling will prove to be useful in order to define transformations on function *f*.

Therefore, the challenge of number plate detection is detecting a rectangular area that corresponds to the number plate in an original image. Humans trivially solve this problem since very young ages, which remains an unsolved mystery because of both the complexities of biological vision and our limited understanding of visual perception in a dynamically and varied physical world. Thus, while humans might define a number plate in natural language as something such as *"a plastic plate attached to the front and back of a vehicle in order to identify it"*, it is necessary to find a different definition that is understandable by machines. Different techniques and algorithms give different definitions of a number plate: for example, in the case of edge detection algorithms, the definition could be that a number plate is a *"rectangular area with an increased density of vertical and horizontal edges"*. This high occurrence of edges is normally caused by the border plates, as well as the limits between the characters and the background of the plate.

Edge detection is based on a set of convolution operations; the result of this series of convolutions give us the area of the plate. For instance, a periodical convolution of the function *f* (defining an image) with a matrix **m** can be used in order to detect a specific type of edge in an image [Mar07]:

$$f'(x,y) = f(x,y) \otimes \mathbf{m}[x,y] = \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} f(x,y) * \mathbf{m}[mod_w(x-i), mod_h(y-j)]$$

Where *h* and *w* are the height and width of the image represented by *f*, and the expression **m**[x, y] represents the element in xth column and yth row of matrix **m**.

Different kinds of convolution matrices **m**, also known as kernels, will result in different kinds of detected edges. A kernel describes how each pixel of the input image is affected by its neighboring pixels during the convolution operation. Intuitively, a kernel acts as as a matrix of weights that "slides" over the original image (defined by *f*), performing an element-wise multiplication with the part of the image that it is on at each moment, and then summing the partial results into a single output pixel. Therefore, the formula to obtain each output value is the following linear combination, assuming a 3x3 kernel:

$$y = x_0 * m_0 + x_1 * m_1 + x_2 * m_2 + x_3 * m_3 + x_4 * m_4 + x_5 * m_5 + x_6 * m_6 + x_7 * x_7 + x_8 * m_8$$

One of the most widely used filters for edge detection is the **Sobel operator**. It consists of two 3x3 kernels, one for vertical edges, and the second of horizontal edges, whose goal
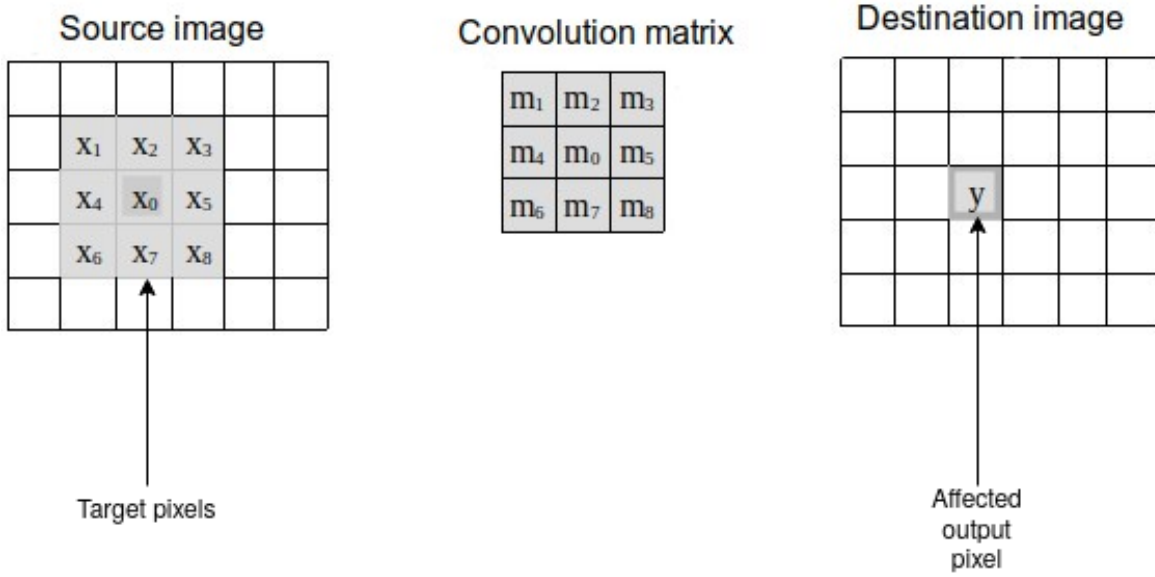
Figure 3.2: Affected output pixel being influenced by its neighbours according to the convolution matrix

is computing approximations of the gradient, taking into account that digital images are represented with discrete points. The kernels G_x and G_y are defined as:

$$G_x = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} G_y = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

With this information, the approximation for the magnitude of the gradient at each point can be computed combining the approximations to the horizontal and vertical gradients:

$$G = \sqrt{G_x^2 + G_y^2}$$

Alternatively, as a less computationally expensive option, the following approximate calculation can be performed:

$$|G| = |G_x| + |G_y|$$

Besides, the gradient's direction can be calculated as well, obtaining information about an edge's orientation:

$$\theta = atan(\frac{G_x}{G_y})$$

Figure 3.3: Example of application of the Sobel operator. From left to right: Original image, Gx, Gy. Source: OpenCV library documentation[4]

Once the Sobel operator has been applied, it is necessary to perform an analysis of the result to detect the plate area. There are many possibilities for this analysis; one of them is a statistical analysis of the projections of the image into the *x* and *y* axes. We define these vertical and horizontal projections as:

$$p_x(x) = \sum_{j=0}^{h-1} f(x, j)$$

$$p_y(y) = \sum_{i=0}^{w-1} f(i, y)$$

Where *h* and *w* are the width and the height of the image, respectively.

Therefore, the horizontal projection p_x of function *f* at point *x* is the summation of all pixels in the xth column. The same reasoning applies for the vertical projection p_y. Then, the problem of finding the plate becomes an analytical problem about finding the peaks of the projection functions. Normally, this analysis is performed in two steps: first, the vertical projection is analyzed in order to detect the vertical area of the number plate (also known as "band"). This phase is known as "band clipping". With the result of the "band clipping" process, a second peak analysis is performed on the horizontal projection in order to detect the plate boundaries on the X-axis; this is known as "plate clipping".

Because of the nature of the described analysis, it might be possible that we end up with several candidates to plate boundaries; in this case, it is possible to perform a heuristic analysis on the candidates. This analysis may include elements that are dependant on the plate regulations of the geographical region where ANPR takes place, such as the plate width or the width/height ratio. Regulation-independent heuristics may also be considered, such as the height of the peak (higher peaks mean a higher concentration of edges, which are more likely in plate regions).

---

[4]https://docs.opencv.org/3.4/d5/d0f/tutorial_py_gradients.html

It is worth mentioning that the Sobel operator is one of the most performant techniques for number plate detection [MBRP10], but it is not the only one. Other techniques discussed in literature, and widely used in commercial solutions, include other edge detection approaches such as the Canny operator [Can86], neural network approaches [ÖÖ12], and custom techniques for ANPR such as the Sliding Concentric Window (SCW) [AAP+08], among others.

Amid all the reviewed literature about number plate detection, it has been noted that most algorithms perform best in restricted environmental conditions such as a determined distance from the vehicle, angle, number plate shape, and illumination, for example.

Character segmentation

Once the plate region has been detected, it is necessary to divide it into pieces, each one containing a different character. This is, along with the plate detection phase, one of the most important steps of ANPR, as all subsequent phases depend on it. Another similarity with the plate detection process is that there is a wide range of techniques available, ranging from the analysis of the horizontal projection of a plate to more sophisticated approaches such as the use of neural networks.

Vertical and horizontal projection-based techniques are the most simple, computationally cheap, and common ones. Similarly to the analysis previously explained in the Plate Detection section, an analysis of the projection's peaks and minimum values can allow us to segment characters; minimum values, in this case, represent separations between characters.
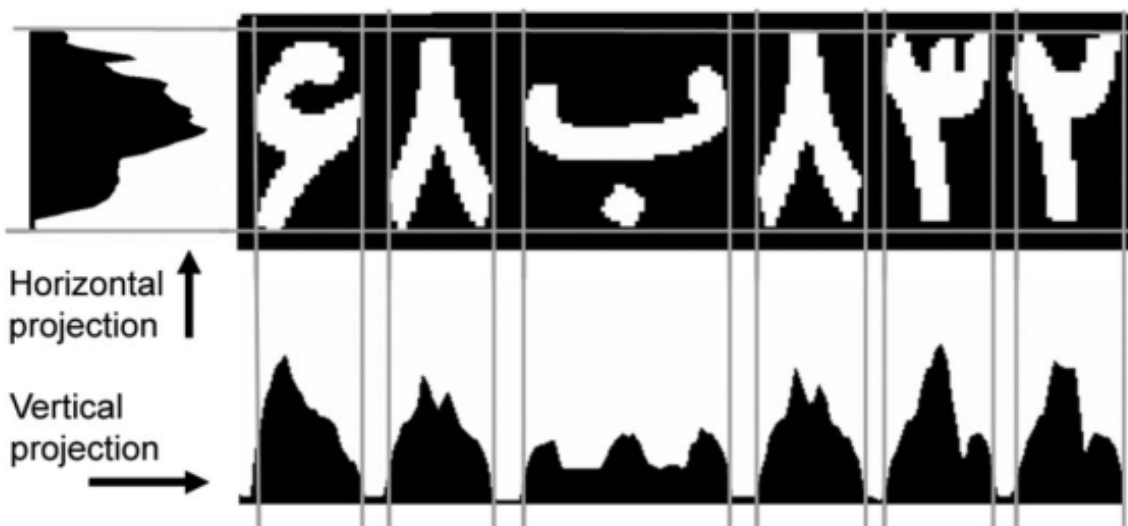
Figure 3.4: Character segmentation using the vertical and horizontal projections[AALK06]

Even though histogram analysis is one of the simplest, most common and straightforward

techniques for character segmentation, it has proven to reach very satisfactory results: for instance, [SMX04] achieves and accuracy result of 99.2% on a dataset of over 30.000 images of Chinese number plates.

Connected Component Analysis (CCA), also known as Connected Component Labelling (CCL), is a technique that can be used both for character segmentation and as a later step known as piece extraction, in which, once segmentation has been performed, several pieces of neighboring, same-color pixels are extracted; for each segment, one of those pieces represents the character.
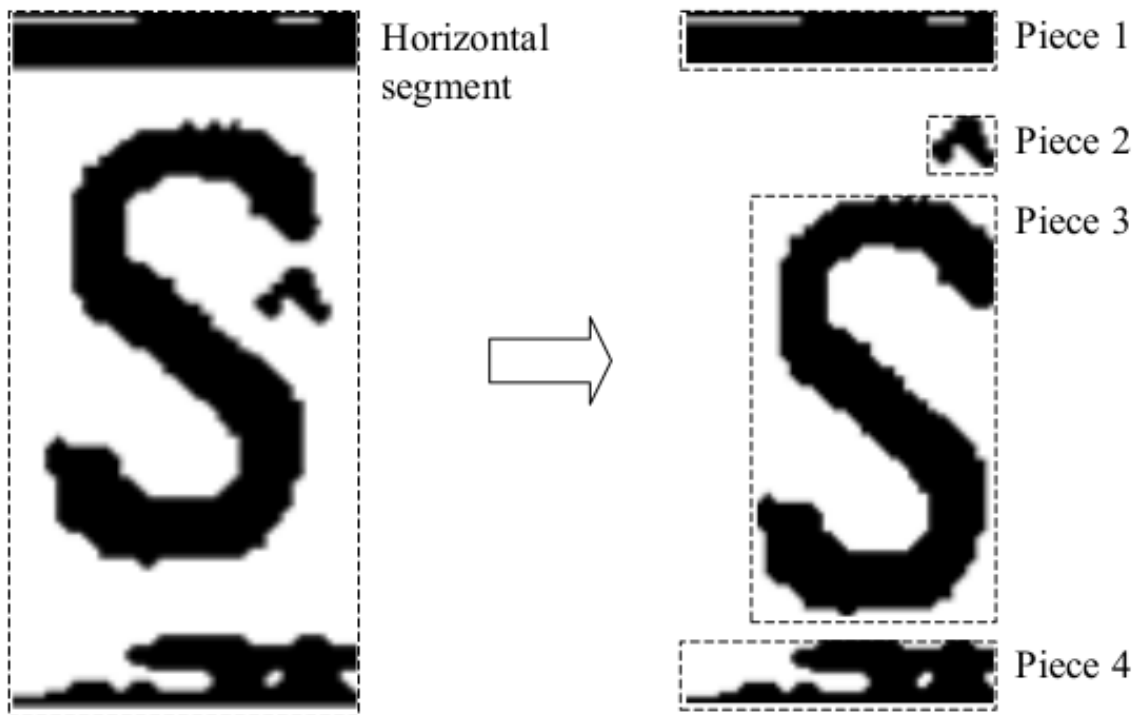
Figure 3.5: Piece extraction from a horizontal segment [Mar07]

CCA is one of the most vital techniques in computer vision, as it has a wide number of use cases. It can be classified as a clustering algorithm which labels elements based on their connectivity; for this purpose, either a 4-connectivity or, more usually, an 8-connectivity can be considered. In the case of computer vision, CCA operates on an already binarized image, labeling connected groups of same-colour pixels. There are two main implementations of CCA. The first one is based on graph theory, and it consists in performing the equivalent of a graph traversal. Briefly explained, this technique locates the first foreground pixel in the image, gives it a label, and then assigns the same label to all its neighbors with the chosen connectivity; the process is repeated for all unlabeled foreground pixels in the image. This technique requires only one pass on the image and has multiple implementations, such as [AQIS07], which optimizes previous one-scan implementations. The second kind of CCA algorithms perform two passes on the input data: the first one assigns temporary labels and

records equivalences between them, and the second one applies those recorded equivalences. The Hoshel-Kopelman algorithm, originally described in [HK76], is a widely known example of the two-pass technique.

Mathematical morphology is another remarkable technique in character segmentation; it is based on set theory, random functions, topology, and lattice theory. It defines basic operators such as dilation, erosion, closing, and opening. This technique shows very good results in degraded plate images, with works such as [NYK$^+$05] showing an 84% accuracy rate among a set of degraded plate images.

While, so far, only image-based have been analyzed, we may also mention video-based approaches, which add temporal and context information to the analysis of each frame. [CH98] is an example of such approach. In this work, character extraction is modeled as a Markov Random Field (MRF), where randomness is used to describe the uncertainty in pixel label assignment. This allows us to add previous contextual information or constraints in a quantitative way. With this modeling assumption, a genetic algorithm with a local greedy mutation operator is used in order to optimize the objective function. Better performance than single-frame based approaches has been shown, along with the possibility of parallel hardware implementations.

To conclude, it can be affirmed that character segmentation is, along with plate detection, one of the most important steps in the ANPR pipeline, as incorrectly segmented characters are very unlikely to be correctly recognized in the next step; indeed, most recognition errors in ANPR systems are caused by detection and segmentation failures, instead of recognition shortcomings. In line with the other steps in the ANPR process, the quality of the captured picture is critical, although some of the analyzed literature focuses on the segmentation of degraded plates, often by taking advantage of previous knowledge.

Character recognition

The next step in the ANPR process consists in recognizing each of the previously segmented characters. At this point, several already existing techniques for character recognition can be employed, ranging from Artificial Neural Networks (ANN) [KC11] and template matching [CLCW09], to the use of already existing OCR solutions such as the open-source TesseractOCR [5] tool [Smi07].

3.1.2 ANPR solutions overview

This section details the different ANPR solutions available on the market, both as commercial products and as open-source software. The different available services will be presented, with an emphasis on the chosen solution. For the latter, an overview of the techniques and algorithms it uses for each step of the ANPR process will be given.

---

[5]www.github.com/tesseract-ocr/tesseract

Plate Recogniser

Plate Recogniser [6] is a company offering a cloud-based number plate recognition service. It is marketed as being trained to work with non-ideal capture conditions such as "sun glare, blurry images, fast vehicles, night time, and many more". It offers an SDK to be installed in the capture devices that processes both images and video, and apart from the cloud service, the possibility of an on-promise installation is possible, although the application remains closed-source. It proposes several pricing plans depending on the volume of requested images to be processed. In terms of cost per picture, the most economic one costs $250 for 500.000 lookups over the course of a month [7], or $0.0005 per lookup.

Anyline

Anyline is a software company offering an ANPR solution [8] as one of its products, mostly focused on European license plates. Concretely, it offers an SDK compatible with Android, iOS and UWP, with additional integration with some common frontend frameworks, which suggests it is mainly designed for integration with mobile and/or web applications. The ANPR process is carried locally in the device where the SDK is installed, which Anyline markets as one of its advantages as no data is sent to third parties. However, among its disadvantages are the fact that the product is closed-source, and that there exists no publicly available pricing information about the system.

Eocortex

Eocortex is a company focused on video analytics solutions for IP (Internet Protocol) cameras [9]. One of the solutions it offers is an ANPR service [10] with a per-camera licensing mode. Among the features of the service, it is mentioned that it can recognize the plates of vehicles "moving at a speed of up to 150 kph", which can have the standards of any of the 196 countries it supports. Overall, its tight requirements about the characteristics of the image capture devices and the lack of a publicly available pricing are characteristics that discarded this option as an ALPR service to use.

OpenALPR

The chosen ALPR service is OpenALPR. OpenALPR is both an open-source library [11] written in C++ for the recognition of license plates in images and video, and a company that offers different solutions based on it. The latter offers an API, called CarCheck, that allows

---

[6] www.platerecognizer.com

[7] www.platerecognizer.com/pricing

[8] www.anyline.com/products/scan-license-plates (Accessed November 2020

[9] www.eocortex.com

[10] www.eocortex.com/products/video-management-software-vms/license-plate-recognition

[11] www.github.com/openalpr/openalpr

querying an already trained and improved model. According to a benchmark [12] performed by the company, this API identifies correctly 99.02% of license plates on the first estimate, while the open-source library correctly identified a 42.16% of license plates. As well as previously analyzed providers, the cost of the API depends on the volume of performed requests. The cheapest cost per request corresponds to a €395 plan for 125.000 requests, or €0.00316 per lookup.

The OpenALPR library documentation contains a section [13] detailing the steps taken during the recognition process and the algorithms used in them. Below, these steps will be analyzed, in order to understand how the process is implemented, and to analyze whether or not they vary with respect to the ANPR steps exposed in the section before.

The first step corresponds to the **detection** phase. In this phase, the Local Binary Patterns algorithm (generally used for face recognition) [AHP06] is used to find possible plate regions. Each of these regions is passed to the consecutive steps for its processing. As the OpenALPR documentation details, the detection phase is, generally, the one with the biggest computational cost. As a matter of fact, the library offers support for hardware acceleration in order to improve its performance.

The images go then through a **binarization** step, which is executed several times, one per each possible plate region detected in the last phase. The binarization phase creates several binary (black and white) images for each possible plate region. Multiple binary images, with different binarization thresholds, are generated. A single binarized image may lose characters if the image is either too dark or too bright. In this phase, both the Wolf-Jolion [WJ04] and Sauvola [SP00] methods are used with different parameters. Each of these binary images is processed in the consecutive steps.

In the **character analysis** phase, Connected Component Analysis (CCA) is performed in order to find blobs that are candidates to characters. Some of the found blobs can be discarded depending on whether they have approximately the specified width and height expected for a plate character. Besides, the blobs have to be aligned between them and have a similar width and height. This analysis is performed several times on each image. If an image contains valid blobs, it is passed to the next step.

The following phase is **border plate detection**. First, all the lines of the image are analyzed with the Hough transform[Hou62] algorithm. Then, this list of lines, along with the character height (computed in the last phase) is used to find the most likely borders. A number of configurable weights are used to determine which borders are the most likely, using a default border based on the height and weight of the plate.

Then, given the plate borders, a **deskew** phase assigns the plate region a standard size and

---

[12]www.openalpr.com/benchmarks
[13]www.doc.openalpr.com/opensource.html#openalpr-design

orientation. This will give us a plate image that is correctly oriented (eliminating rotation and skew).

The **character segmentation phase** is then carried. A vertical histogram is used to find gaps between the plate characters. This phase will, as well, clean up the character boxes by eliminating small, unconnected spots and filter character regions that are not high enough.

The **OCR (Optical Character Recognition) phase** is then applied. For each character image, it computes all possible characters and their confidence level. For this phase, the TesseractOCR free software tool, previously trained, is used.

Finally, given a list of all possible characters and their confidence levels, the **postprocessing** phase determines the best letter and number combinations in the potential plate.

As a conclusion of the analysis of the ANPR implementation in this library it can be observed that, although some phases and algorithms vary slightly with the ANPR process as it was explained in the section before, others did not, giving evidence of the variety of techniques that are employed in real-world ANPR systems.

3.2 Cloud Computing

3.2.1 Introduction

Cloud Computing is the delivery of computing resources and storage via the internet. It usually refers to a model where IT resources and applications are delivered on an on-demand, pay-as-you-go basis.

The term, which was coined after the symbol used in network diagrams, had its first known mention in a Compaq internal document dated November 14, 1996, and titled "Internet Solutions Divison Strategy for Cloud Computing". It was popularized by Amazon in 2006 when releasing its Elastic Compute Cloud (EC2) service, now part of Amazon Web Services (AWS), and has since then become widespread in the IT industry.

The provided resources are usually pulled from a shared pool. The facilities supporting that pool can range from a dedicated server to entire data centers, often replicated between different geographical locations for disaster recovery. Multitenancy is achieved by means of virtualization and, more recently, containerization.

Cloud typology

The ownership of the aforementioned shared pool determines a classification between Public Clouds and Private Clouds. Private Clouds (also known as 'on-premises' clouds) are internal to an organization, which manages the physical premises they are based on. Public Clouds, on the other hand, are managed by a cloud provider, which is responsible for the management, maintenance, and updating of the physical appliances. In the model known as Infrastructure as a Service (**IaaS**), the cloud provider manages the networking, storage,

servers, and virtualization, and the client is responsible for all the environment running over the Hypervisor: OS, Middleware, Runtime, Data, and Applications. In the Platform as a Service (**PaaS**) model, the cloud vendor takes the responsibility for the OS, Middleware, and Runtime, offering the client a platform to deploy its applications on.

■ User managed	■ Provider managed		
On premises	**IaaS**	**PaaS**	**SaaS**
Application	Application	Application	Application
Data	Data	Data	Data
Runtime	Runtime	Runtime	Runtime
Middleware	Middleware	Middleware	Middleware
Operating system	Operating system	Operating system	Operating system
Virtualization	Virtualization	Virtualization	Virtualization
Networking	Networking	Networking	Networking
Storage	Storage	Storage	Storage
Servers	Servers	Servers	Servers

Figure 3.6: Separation of concerns between the user and cloud provider in the On Premises, IaaS, PaaS and SaaS models, according to Alibaba Cloud. [15]

Function as a Service (**FaaS**), also known as Serverless, is a novel cloud execution model, introduced by AWS in 2014 with its "*AWS Lambda*" service, and replicated by all major cloud providers. FaaS goes one step beyond PaaS, as it allows customers to completely abstract from the provisioned compute resources that execute their code; instead, customers can submit their code in independent units of execution ("functions") and are billed by the number of executions, the total execution time, and the employed memory. This allows for a pay-as-you-go model in which customers only pay for the resources they use, in a fine-grained fashion, instead of paying for a server even when it's on standby waiting for requests. Scalability is, as well as in the PaaS model, the responsibility of the cloud provider, so developers and operators don't need to set up scaling policies. In addition to the cost and scalability advantages, productivity is an added benefit to the FaaS model, as developers and systems administrators stop being concerned about various issues such as multithreading or resource allocation optimization, simplifying the process of software development. Among the disadvantages of this model, there is the added latency when cold-starting FaaS code

---

[15]https://www.alibabacloud.com/knowledge/difference-between-iaas-paas-saas

(executing it for the first time after it has been inactive), resource limits that might make it unsuitable for certain use cases, added difficulty for debugging deployed code, privacy (resources are shared, and third party employees could have access to the deployed code), lack of open standards, and vendor lock-in.

Cloud Security and the Shared Responsibility Model

As it has been exposed, services offered by public cloud providers allow customers to abstract at will from different layers. From a security perspective, the Shared Responsibility Model, as described by Amazon Web Services [TO13] and other cloud providers, applies: as different parties (the Cloud Provider and the Consumer) are involved in the operative of cloud architecture, responsibilities must be clearly defined and understood. In general, the cloud provider will assume responsibility for the underlying physical infrastructure of the system and networking. Depending on the cloud service, the Cloud Provider assumes different levels of responsibility: for example, in PaaS-based services, the Provider takes actions such as OS-level security patches, while in IaaS-based services, this is the responsibility of the customer. This separation of responsibilities must be well described and understood by the customer, as it's critical for the security operative.

3.2.2 Advantages of the Public Cloud

Since its creation, Cloud Services have been widely used both by individual developers and organizations to sustain their needed IT infrastructure. The reasons for its remarkable success are varied; some of them are explained below.

Economies of Scale

Since compute and storage resources are pulled from a shared pool, it is possible for the Cloud Provider to reduce expenses proportionally to the number of existing customers. The role of economies of scale in cloud computing has been widely discussed [JW11], and, indeed, the use of the sum of computing resources is optimized with the Cloud model, with better efficiency in the utilization of the shared infrastructure, translating into lower prices for the end customer.

Less upfront investment

Cloud Computing allows individuals and organizations to move from a model based on Capital Expenditures (Capex) to a model based on Operational Expenses (Opex), thanks to the pay-as-you-go billing model. Therefore, the upfront investment in computing resources for new projects is substantially reduced. However, in some cloud providers, it is possible to access discounted rates in exchange for an upfront payment. This option is commonly used in production projects with a steady number of requests. It conforms to a Capex model, which can be an advantage for organizations in which the Opex model has not been introduced.

Geographical scalability

Major cloud providers have premises in many regions across the globe, which proves to be an advantage when scaling a platform or application globally. This gives the cloud customer the potential to reach end-users at a lower latency, thus, providing a better experience for them. For application developers, the environment they deploy their code on tends to be very similar among all geographical locations, making it easy to scale globally.

Managed services

Cloud services allow the customer to abstract, at will, from different layers. This relieves the customer from the burden of having to consider some aspects of IT operations, such as network security, OS updates, backups, scalability and stability, or compliance. Depending on the service, some of these aspects may be exposed to the customer to different degrees; however, managed services offer the possibility of less expenditure on infrastructure maintenance and systems operations. Smaller teams can be responsible for larger parts of the infrastructure, allowing organizations to move and innovate quicker, and dramatically reducing downtime.

Quick deployment

In a cloud computing environment, new resources are quickly available, and they are taken from a seemingly unlimited pool of computing power and storage. This capability to use new resources on demand allows organizations to reduce deployment times to minutes, while, at the same time, having to stop guessing the future capability needs.

3.2.3 Cloud provider review

The worldwide cloud services market is in continuous growth, as organizations from all industries are driving digital transformation initiatives. Expenditure in storage, compute, and other on-demand cloud services are therefore growing: a report [16] made by the Canalys market analysis firm in January 2020 estimated a total expenditure in 2019 of \$107 billion, up from \$78 billion in 2018. Competition between the leading cloud providers is intensifying, and each one of them is continuously trying to innovate with new services and offerings in order to attract customers. The same analysis by Canalys draws the picture of a cloud market dominated by four major providers, namely, Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, and Alibaba Cloud. The following table shows the worldwide expenditure on each cloud provider, along with their market share and annual growth, in the years 2018 and 2019, according to the aforementioned analysis from Canalys.

As can be observed, all major providers have experienced a positive annual growth; on the other hand, the market remains dominated by AWS, followed by Azure and GCP. Below,

---

[16]https://www.canalys.com/newsroom/canalys-worldwide-cloud-infrastructure-Q4-2019-and-full-year-2019

Cloud service provider	Full-year 2019 (US$ billion)	Full-year 2019 market share	Full-year 2018 (US$ billion)	Full-year 2018 market share	Annual growth
AWS	34.6	32.3%	25.4	32.7%	36.0%
Microsoft Azure	18.1	16.9%	11.0	14.2%	63.9%
Google Cloud	6.2	5.8%	3.3	4.2%	87.8%
Alibaba Cloud	5.2	4.9%	3.2	4.1%	63.8%
Others	43.0	40.1%	34.9	44.8%	23.3%
Total	107.1	100.0%	77.8	100.0%	37.6%

Note: percentages may not add up to 100% due to rounding

Table 3.1: Worldwide cloud infrastructure spending and annual growth (Canalys estimates, full-year 2019)

these top three major cloud providers will be analyzed, along with their fundamental service offerings and their applications.

Amazon Web Services

With presence in over 20 regions around the world in 2020 and an estimated 32.3% market share, AWS is, undoubtedly, the leader in the cloud market. The AWS platform was launched internally in July 2002, and at its origins, it consisted of a disparate collection of tools and services. In late 2003, Chris Pinkham and Benjamin Black presented an internal paper describing an efficient way to scale up Amazon's retail website, applying different abstractions to decouple applications from infrastructure, and standardizing and fully automating the latter. Near the end of the paper, they mentioned the potential to generate revenue from the standardized infrastructure, proposing to "*sell it (the infrastructure) as a service*"[17]. In 2006, AWS was officially launched with an initial offer of three products: a storage service (**S3**), a message queuing service (**SQS**), and a virtual machine lease service (**EC2**). All of these services form a central part of AWS, providing the foundations on which many other services are based on.

Today, AWS offers a catalog of more than 200 services, with new ones being announced every month; they are grouped into areas, ranging from generalist ones such as "Compute", "Storage" or "Database", to more specialized areas like "Machine Learning", "Internet of Things" or "Blockchain". Many of those services and their improvements are made based on customer's feedback, some of them being the first offering of their kind in the cloud market, and replicated later by other cloud providers. Therefore, most of the services that will be explained below are available too under other Cloud Providers, under different denominations, and sometimes, with slightly different terms and sets of features. As summarizing such a vast catalog is out of the scope of this document, this section will explore some of the main areas of the AWS platform briefly describing the most important services in each of them.

In the "Compute" area, the main component is the aforementioned "*EC2*" service, which

---

[17]https://www.networkworld.com/article/2891297/the-myth-about-how-amazon-s-web-service-started-just-won-t-die.html

allows us to rent virtual machines ("instances") on the cloud, billed on a per-second basis. This is, therefore, an IaaS service, on which the cloud provider (Amazon) offers a Hypervisor over which the customer can run the Operating System of their choice. There are several instance types available[18], fit for different use cases: general-purpose, compute optimized, memory-optimized, accelerated computing, and storage optimized. Besides, there are different pricing models meant for different use cases: the "*on-demand*" model has the highest price per second, the "*reserved instances*" model allows a discount over the "*on-demand*" model in exchange for a use commitment (typically 1 or 3 years), and the "*spot instances*" model gives a varying discount in function of the current instance demand. *Spot instances* may be terminated at any moment by the cloud provider with short notice if the instance demand from cloud users grows. The different combinations between instance types and pricing models give application developers a lot of flexibility to adapt to different use cases, and most cloud providers offer similar options. Along with "*EC2*", "*Lambda*", the first FaaS service, are the pillars of the "Compute" area offering.

Regarding the "Storage" area, the most important service is *S3*, an object storage service with eventual consistency designed for a 99,999999999% (11 nines) durability over the period of a year. *S3* organizes data within *buckets*, which contain a set of stored objects. Objects are stored as key-value pairs, the key being the *path* of the object, and the value being the binary data. Each bucket operates in a geographical region, and the storage price per GB varies per region; in order to replicate buckets between regions, the "*cross-region replication*" feature is offered. Some other features that are remarkable in S3 are automated encryption, object versioning, and access control based on different policies. Another core feature of *S3* is its tight integration with other managed services: for instance, it may be used to host static websites in conjunction with *CloudFront*, a Content Distribution Network service. With hundreds of Points of Presence (PoP) around the globe, static websites are served with low latency. *Lambda* may be used as well to execute ad-hoc code, both in conjunction with *CloudFront*, or directly in response to *S3* events, allowing, for example, to execute specific code whenever an object is updated. The *Athena* service may be used to query files in tabular format (CSV or Apache Parquet, for example) stored in *S3* using the SQL language; therefore, analytics can be performed over objects stored in *S3*. As a matter of fact, AWS encourages the creation of Data Lakes using *S3* as its primary storage platform [19], creating a central repository to store the entire set of an organization's structured and unstructured data, with virtually unlimited scalability, ensuring the decoupling within data storage and data processing, and offering many possibilities of integration with other AWS services.

Concerning the pricing scheme, *S3* offers several storage classes meant for different use

---

[18]https://aws.amazon.com/ec2/instance-types

[19]https://docs.aws.amazon.com/whitepapers/latest/building-data-lakes/amazon-s3-data-lake-storage-platform.html

cases [20]. Within a bucket, storage classes can be configured at the object level. *S3 Standard* is the default storage class, meant for frequently accessed data, and offering the minimum retrieval latency and highest throughput performance. *S3 Standard - Infrequent Access* offers the same performance with a lower storage cost in exchange for a retrieval fee, and *S3 One Zone - Infrequent Access* doesn't replicate the data between zones (the former options replicate data between three zones), offering a 20% discount over the *Infrequent Access* option. Finally, there are two storage classes meant for archival purposes: *Glacier* and *Glacier Deep Archive*. Both classes offer substantial discounts over the *Standard* class, in exchange for longer recovery times and a higher retrieval fee: there are three retrieval options, with times ranging from some minutes to several hours. The targeted use cases for these storage classes are, for instance, an alternative to magnetic tape devices, backup and disaster recovery, and long term data retention for regulatory compliance. In addition to the storage cost of all the described classes, networking transfer fees when retrieving objects apply as well.

It is worth noting that it is possible to define *lifecycle policies*, which allow us to define transition actions specifying the moment at which the storage class of an object should change: for example, it is possible to define a policy moving an object from the *S3 Standard* class to *S3 Standard - Infrequent Access* 30 days after they are created. On the other hand, for long-lived objects with unknown access patterns the *S3 Intelligent - Tiering* class automatically moves objects between frequent access and infrequent access tiers depending on their usage.

Finally, for uploading big volumes of data to *S3* in a performant way, several options exist. First, it is possible to establish a dedicated wired connection between the customer premises to AWS datacenters with the *DirectConnect* service. The dedicated connection can have a bandwidth of either 1 Gbps or 10 Gbps, providing a more consistent network performance than Internet-based solutions. Another option is the *Snowball* service, which consists in a physical device sent to the customer's premises that, once loaded with the data to be uploaded to *S3*, is sent back to AWS. The storage capabilities of the device range from 42TB to 80TB [21]. Snowball devices are fitted with Edge Computing capabilities as well, which allow us to preprocess data before storing it to the Hard Disk Drive; some of them are fitted with dedicated GPUs meant to accelerate Machine Learning processes. Finally, for massive transfers of up to 100 Petabytes, the *Snowmobile* service [22] allows to, after an initial assessment, receive a ruggedized shipping container, fit with Hard Disk Drives and pulled by a semi-trailer truck, at the customer's premises. The *Snowmobile* can then be connected to the customer's internal network with the help of AWS personnel, and start the data transfer, which will be loaded to *S3* once the *Snowmobile* is back at AWS' premises. This allows us to complete a 100PB data migration in a few weeks, which could otherwise take more than

---

[20]https://aws.amazon.com/s3/storage-classes
[21]https://aws.amazon.com/snowball/features
[22]https://aws.amazon.com/snowmobile

Figure 3.7: AWS Snowmobile presentation at RE:Invent 2016

Figure 3.8: A Snowball device

20 years under a 1Gbps *Direct Connect* line.

Moving forward to the "*Database*" area, the most remarkable services are *RDS* and *DynamoDB*. *RDS* is a managed relational database service [23], currently supporting the Relational Database Management Systems (RDMS) PostgreSQL, MySQL, MariaDB, Oracle Database and Microsoft SQL Server, as well as Aurora, which is a database engine developed by Amazon and compatible with PostgreSQL and MySQL. RDS can be considered as a PaaS service, and therefore, typically time-consuming database administration activities such as provisioning, backups, security updates, monitoring, and scaling are provided by AWS. The user only has access to the database application, and not to the operating system that the database runs on. Similarly to *EC2*, several instance types are available depending on the intended use case, and pricing schemes such as *reserved instances* are available as well. In addition, for the Aurora engine, it is possible to opt for the *Aurora Serverless* model,

---

[23]https://aws.amazon.com/rds

in which, instead of provisioning database instances, the customer specifies an amount of capacity units, which are a combination of compute and storage capabilities. The database then starts, scales up or down, and shuts down automatically in function of the demand and needs of the application, which proves to be a simple and cost-efficient solution for intermittent, unpredictable, or sporadic workloads. Finally, *DynamoDB* is a fully managed NoSQL key-value and document database that is scalable and offers single-digit millisecond response times. Like *Aurora Serverless*, it is not necessary to provision database instances; instead, customers can either pay on a per-request basis or, in case of workloads with predictable demand, pay for a determined amount of provisioned read and write capacity. As can be noted, AWS (among other cloud providers) is pushing to expand the *serverless* framework, allowing FaaS services (*Lambda*) to interact with other services where it's not necessary to provision and maintain any server. Thus, it becomes possible to create applications whose usage is billed merely on a per-request basis, scalable, and with low maintenance costs.

Azure

Azure is the platform under which Microsoft offers its cloud services. Formely known as *Windows Azure*, it was released on February 1, 2010, and renamed to Azure on March 25, 2014 [24]. As of November 2020, Azure offers a total of 270 services [25] in 43 different regions worldwide. [26]

A brief analysis of the core services provided by Azure enables to establish multiple parallels between those offered by Amazon Web Services. For example, the object storage service *Azure Data Lake Storage* can be considered as the equivalent of *S3* in AWS, *Azure Functions* would be the equivalent of *Lambda*, and database services such as *Azure SQL Database* can be compared to *RDS*. The equivalent of *EC2*, called *Azure Virtual Machines*, constitutes one of the main value propositions from Azure, as Microsoft claims that running Windows Server instance on its cloud is up to 5 times cheaper than doing so in AWS. [27]

The Hybrid Cloud is another core value addition that Azure proposes when compared to other cloud providers; as a matter of fact, support for the migration to the cloud from on-premises environments can be easier when an organization is already heavily invested in the Microsoft Technology Stack, with integration with services such as *Active Directory* being an added value in this context. This, added to some unique services such as *Azure Policy*, which verifies and enforces compliance with a defined set of policies across a cloud environment, make Azure an option to take into account by large organizations.

---

[24] www.azure.microsoft.com/en-us/blog/upcoming-name-change-for-windows-azure

[25] www.azure.microsoft.com/en-us/services/

[26] www.azure.microsoft.com/en-us/global-infrastructure/geographies

[27] www.azure.microsoft.com/en-us/overview/azure-vs-aws/

Google Cloud Computing Platform

Google launched its cloud offering, "Google Cloud Platform" (or GCP) on April 7, 2008. The infrastructure behind GCP is the same as the one Google uses for its internal and end-user products, providing customers with the ability to deploy cloud resources in 22 geographical regions (locations) divided into 67 zones, as of the beginning of 2020. Each zone is an isolated set of physical resources on which cloud resources can be deployed; zones can consist of one or more data centers. This decoupling between regions, zones, and physical data centers allows customers to design robust, highly available systems that are able to recover from physical failures by spanning multiple zones and regions.

One unique feature about the infrastructure behind Google Cloud Platform is its backbone fiber network: with thousands of kilometers of fiber optic cable around the globe, including submarine cables across the Atlantic and Pacific oceans, Google operates one of the largest and most modern computer networks. In order to provide end-users access to this network, more than 100 Points of Presence (PoPs) are available around the globe. GCP customers can benefit from this infrastructure with the "Premium Network Tier" service: once this feature is activated for a group of serving virtual machines (instances), the customer gets a single, anycast IPv4 or IPv6 Virtual IP (VIP). When end-users connect to this address, they are routed through the closest PoP to their location; from this point to the region where the serving instances are located, the connection will be routed through GCP's dedicated fiber network, reducing the number of hops the connection runs through, and, thus, improving latency, performance and reliability. This feature dramatically simplifies architecture designs, as it is possible to develop applications with a unique, global IP address; global load balancing is automatically provided, and it is possible to overflow or fail-over into other regions keeping relatively good performance.

Google App Engine

Google App Engine (or **GAE**) is a PaaS solution, integrated within *GCP*, offering hosting for web-based applications. Google announces it as a "*fully managed, serverless platform*" [29]. It was released on the 7th April, 2008, as a preview version, and got out of preview on September 2011. Since then, it has been continuously updated. GAE offers runtimes for the languages Go, PHP, Java, Python, Javascript, Ruby, and the .NET framework. It is also possible to use custom runtimes, including any additional component that developers may need, such as application servers or language interpreters. This flexibility, and the options for scaling, monitoring and debugging, makes GAE a good candidate to run a microservices architecture [LLH16].

---

[28]https://www.blog.google/products/google-cloud/expanding-our-global-infrastructure-new-regions-and-subsea-cables

[29]https://cloud.google.com/appengine/docs
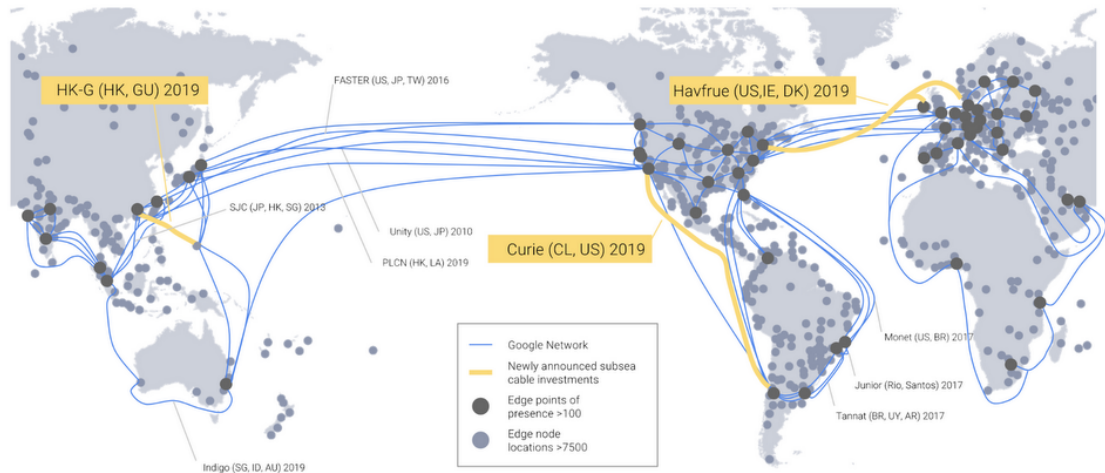
Figure 3.9: Google's global network[28]

GAE offers two kinds of environments to run applications on [30]: the *standard* environment and the *flexible* environment. Both environments are meant to be complementary, and are meant for different use cases; however, depending on the environment, different GAE features will apply. One of the main differences between them is that the *standard* environment is more restrictive in the sense that applications run in a *sandbox*, using provided runtimes with specific versions of languages and frameworks. Operating System access, both from the application and from the developer, is restricted: for example, the application cannot use the file system or custom binaries, and the developer cannot connect to the running environment with tools such as SSH. Communication with other GCP services is possible with a provided API. On the flip side, these restrictions allow a much quicker scaling of the application, as a custom-designed scaling algorithm is used for the standard environment. A typical use case for the standard environment is a stateless web application (for example, a REST API) intended to have low response times, and running no other background processes. Regarding resource allocation, standard environment applications can run on different instance classes [31], each one comprising different amounts of CPU and memory. Instance classes can be divided into *backend* instances and *frontend* instances.

The flexible environment, on the other hand, allows us to execute any custom Docker container [32] provided by the user. These containers can include any language interpreter and/or execution runtime specified by the developer, including any required dependency

---

[30]https://cloud.google.com/appengine/docs/flexible/go/flexible-for-standard-users

[31]https://cloud.google.com/appengine/docs/standard#instance_classes

[32]https://www.docker.com

and custom binary. Running multiple processes is allowed as well. Unlike the standard environment, it's not possible to call other GCP services using the build-in provided API; instead, it's necessary to install the Google Cloud client libraries as a dependency in the provided container. This allows the application to be more portable among other services. Finally, it is possible to choose different instance types for the flexible environment, each with different CPU and memory. This is unlike the standard environment, which limits the CPU and memory resources available to applications. On the other hand, when it comes to scaling, the flexible environment has more limitations: unlike the standard environment, which scales in a matter of seconds, flexible environment instances can take minutes to provision. This causes that a minimum of one flexible instance must be up at all times in order to serve traffic uninterruptedly, causing additional costs in applications where traffic is not continuous.

It is worth mentioning that, in order to ease the development and testing of applications, a "free tier" [33] is offered. In the case of GCP, this "free tier" is comprised of both an initial credit offering of $300 for new accounts, expiring after a year, and an "*always free*" program providing free access to some GCP services. *AppEngine* is included in the "*always free*" program only for the standard environment, offering 28 hours of free "frontend" instance hours per day, and 9 hours of free "backend" instance hours per day. The "*always free*" program is, therefore, another point in favor of the standard environment when developing and testing small, stateless applications.

In order to access applications hosted on *AppEngine*, a domain name, with the format *appname.appspot.com*, is provided. It is possible to use custom domain names with the *Cloud DNS* service. As mentioned, similar integrations with other Google Cloud services exist: for example, the NoSQL database *Google Cloud Datastore* [34], or *Google Cloud SQL* [35] for applications requiring a MySQL, PostgreSQL or Microsoft SQL Server database.

One of the main concerns when developing applications to be deployed to PaaS services such as *AppEngine* is the risk of vendor lock-in. Vendor lock-in is a situation where customers are dependent ("locked-in") on a Cloud Provider, because of a tight coupling between their applications and the cloud services of the provider they're deployed on. In the case of *AppEngine*, this risk becomes especially evident for *standard environment* applications because of the restrictions this environment imposes, which forces the application architecture to conform to them, and encourages the use of other Google Cloud services; in the case of the flexible environment, it is possible, in theory, to build containers that can run indistinguishably both on *AppEngine* and on other environments, but, in practice, programmers are still encouraged to use other GCP services to develop their applications.

---

[33]https://cloud.google.com/free/docs/gcp-free-tier
[34]https://cloud.google.com/datastore
[35]https://cloud.google.com/sql/docs

One of Google's main responses to these concerns is *Kubernetes* (commonly written as *k8s*)[36], an open-source portable and extensible platform allowing the deployment of containers (which can contain applications or other workloads), developed by Google and maintained by the *Cloud Native Computing Foundation*, a Linux Foundation project. *Kubernetes* abstracts away many aspects of proprietary cloud infrastructure, allowing applications to run on different cloud providers, or on-premises, with minor changes. Many features commonly found in PaaS, such as load balancing, health checks, horizontal scaling, deployments, automated rollbacks, and monitoring, are also present in Kubernetes; however, *Kubernetes* is not monolithic. It is a collection of independent, optional solutions that can be used depending on the use case and needs, giving developers the flexibility to use it to the extent they need. This can be compared to the possibility of using different features and cloud services in a PaaS environment. GCP's offering to run *Kubernetes* workloads is the *Google Kubernetes Engine* (GKE) service, and other cloud services have similar offerings for *Kubernetes* loads.

---

[36]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes

Chapter 4

Method and work phases

This chapter will discuss the methodology that will be used to tackle all the objectives of the project in the most efficient possible way, as well as the needed resources.

4.1 Work methodology

During the development of the platform, the main stakeholder is the involved staff members of the *Escuela de Ingenieros de Caminos, Canales y Puertos de Ciudad Real*, which will be the first users of the platform. Therefore, after an initial requirements gathering, several follow-up meetings were scheduled, including demo sessions so that the results of each iteration could be presented, and suggestions could be made.

Given the fact that the development would be made by a very small team, that requirements could change during the review and demo sessions, and that the chosen third-party service for ANPR would need to be tested, it was necessary to adopt a work methodology that would prevent these obstacles and uncertainty from having an impact on the project. Agile methodologies, as defined in the Agile Manifesto [1] are a widely used approach in development that tackle this kind of issues in an effective manner. As opposed to the "*waterfall*" model, agile methodologies focus on iterative and incremental development cycles, promoting shorter feedback cycles leading to more flexibility to change and earlier product validation, among many other advantages [Boe07]. Each iteration is composed of different phases covering requirements, analysis, design, development, testing and delivery; the end result is an increment consisting on a working implementation of a subset of functionality. This result is reviewed to see the degree to which it fulfills its goals; then, the conclusions, along with other inputs such as new information about the market, identified improvement points, or priority changes, are used as inputs for the planning of the next iteration.

There are numerous frameworks and methodologies within the *agile* approach. For example, Scrum [2] is a framework that defines a set of events (such as sprint reviews, retrospectives and plannings), roles (such as Product Owner and Scrum Master), artifacts, and rules and values that lead the development process [SS11]. Extreme Programming (XP) and

---

[1] www.agilemanifesto.org
[2] www.scrum.org/resources/what-is-scrum

Lean Software Development are other examples of widespread agile software development frameworks [MMSU15]. For our use case, we have decided to use the principle of iterative and incremental development in conjuction with Kanban [3], a visual workflow management framework which organizes development tasks in a three-step divided board (To Do, Doing, Done) [And10].

4.2 Resources

The following section will enumerate and briefly discuss the resources, either hardware, software, or of any other kind, which are necessary to tackle the development of the platform, as well as its deployment and operation.

4.2.1 Hardware resources

- *Asus X554LA-XX1248H laptop*: It is a mid-range laptop which will be used for the development of the platform. It is fitted with an Intel i3-5005U processor, 8GB of RAM, and 1TB of HDD storage; these features perfectly support the development of web-based microservice applications meant to be deployed on the cloud.

- *Raspberry Pi Zero W*: This model is a variant of the famous *Raspberry Pi* System on a Chip (SoC). Released in 2017, it is a reduced version of the *Raspberry Pi*, featuring a 1GHz single-core CPU and 512MB RAM. Besides, the *Zero W* version features improved connectivity over 802.11 b/g/n Wireless Lan, and Bluetooth 4.1. This, added to a camera connector, and a reduced price tag of $10, make it a very good device to test the platform from the point of view of an image capture device. For this purpose, a *Raspberry Pi Camera Module v2*, equipped with a Sony MX219 8-megapixel sensor, is connected to the camera port.

- *Redmi Note 4X*: This Android smartphone will be used to test the platform in a real world environment, allowing us to test the responsive web interface.

4.2.2 Software resources

Operating system

- *Ubuntu 16.04 LTS*: Launched for the first time in 2004, this GNU/Linux flavor is one of the most popular ones among desktop users [4]. Based on *Debian*, it provides the *apt* package manager, which allows us to download the necessary development software.

- *Raspbery Pi OS*: Formerly known as *Raspbian*, it is *Raspbery Foundation*'s officially supported operating system [5]. It is downloadable in several images with vaying degrees of preinstalled software; for our testing purposes, we will use the *minimal* one, customizing it for our needs.

---

[3] www.atlassian.com/agile/kanban
[4] www.distrowatch.com
[5] www.raspberrypi.org/downloads/raspberry-pi-os/

- *Android*: This widely known mobile Operating System will be used in the smartphone used to test the operation of the platform in a real-world environment.

Development tools

- *GNU Emacs*[6]: This highly extensible text editor, with support for multiple programming languages, will be used for the development of the platform.

- *Firefox Developer Tools*: The developer tools featuring in the Firefox browser will allow us to debug, step by step, the *Javascript* code that will be written for the functional part of the web interface, as well as to render the website in different screen resolutions.

- *Git* and *BitBucket*[7]: The version control system *Git* will be used to maintain a code repository, containing the full history of changes (*commits*), and allowing us to work on different versions of the codebase (*branches*) to isolate the development of different features.

- *Trello*[8]: The free version of this list-making web-based application will be used to organize the platform development. The different features to be developed will be separated and organized into cards, which can have different status (To Do, Doing, Done) in line with the Kanban approach.

Programming languages

- *Python*: This interpreted, general-purpose programming language will be used to write the web server of the platform, as well as to write the client-side code that the testing devices will execute.

- *HTML/CSS*: The Hypertext Markup Language will be used to format the pages used in the web interface. The Cascading Style Sheets language will be used to describe the presentation of the webpage.

- *Javascript*: This interpreted programming language, meant to be executed in web browsers, will be used for the functional part of the webpage, performing, among other tasks, asynchronous calls to the server for image retrieval and device configuration.

- *Bash*: This shell language will be used to write scripts automating the building and deployment of the application to the cloud, as well as to initialize the capturing process in the testing capture device.

---

[6]www.gnu.org/software/emacs
[7]www.bitbucket.org
[8]www.trello.com

Libraries, frameworks, and SDKs

- *Flask*[9]: An open-source, lightweight web application framework, with support for HTML templates, that will be used to write the web application server with a minimal overhead.

- *Bootstrap*[10]: A responsive CSS framework, featuring an advanced grid system, that will be used to develop the responsive web interface, showing the different capture devices and their real-time footage, as well as the recognition result.

- *JQuery*[11]: A Javascript library that simplifies the access to the HTML Document Object Model (DOM) and its manipulation; it is used for the functional part of the web interface.

- *Google Cloud SDK*: This software package provides utilities for interacting with Google Cloud resources, allowing to use them and to deploy infrastructure and applications, as well as to test *AppEngine* applications locally.

Documentation tools

- LaTeX: This text composition language, widely used in the academic community to create written documents with a high typographic quality, is used for the generation of this documentation. The online tool *Overleaf*[12] is used to simplify the work, as it provides document hosting and a in-browser editor with collaborative edition features.

- *Markdown*: This lightweight markup language, widely used to write readme files, will be used for plain-text documentation of the code repositories.

- *Draw.io*[13]: An online diagram creation tool, featuring different icon sets, that will be used for the creation of figures to illustrate this document.

4.2.3 Cloud resources

This section includes the Cloud resources, which cannot be classified as either hardware or software resources, as they consist of both the underlying physical infrastructure and a software layer to make it available as a service, adding different functionalities depending on the type of the service (for instance, IaaS or PaaS). The chosen Cloud vendor to run our platform on will be **Google Cloud Platform** (GCP), and therefore, all the resources mentioned below refer to GCP services.

- *Google App Engine*: This PaaS solution will be used to deploy the web platform, as it fulfills the criteria needed to host the application in a scalable, cost-effective way.

---

[9] www.flask.palletsprojects.com/en/1.1.x
[10] www.getbootstrap.com
[11] www.jquery.com
[12] www.overleaf.com
[13] www.draw.io

- *Cloud Datastore*: GCP's managed NoSQL database will be used to store the different entities that are present in the system, containing information such as the configuration of each capture device, the users of the system, and the captured images and their ANPR result.

- *Cloud Storage*: This managed object storage service will be used to store the images taken by the capture devices and generate references to them.

Chapter 5

Architecture

In this chapter, the architecture proposed for the development and deployment platform is exposed from a technical point of view. The architecture will be introduced following a **top-down** approach, in which, first, a general overview of the different components of the system will be given, followed by an explanation of each of the defined modules. For the design of the architecture, its components have been separated according to the functional requirements to be implemented, although systematic, non-functional requirements have been considered as well, and therefore contribute to shape the end result of the architecture.

5.1 Systematic requirements

Due to the nature of this development, and the methodology used to tackle it, requirements can change in each iteration as new features are implemented. In this context, it is necessary to take **scalability** into account for the development of the architecture; here, scalability is defined as the capability of a system to integrate new components, growing and adding functionality without negatively affecting the rest of the system. Related to the scalability requirement (in the sense of adaptation to change), there are two other similar, systematic requirements: **evolvability**, defined as the system being able to tolerate software or hardware modifications, and **integration**, defined as the capability of the platform to integrate new devices. Beyond these considerations, as the platform will be used to define experiments which are not easily replicated and that can evolve a potentially high number of devices, **availability** is necessary, defined as the platform robustness and fault-tolerance, detecting failures and correctly dealing with their consequences. Regarding **security**, the kind of data that the platform manages is of the uttermost importance; the EU General Data Protection Regulation (GDPR) defines personal data as "*any information relating to an identified or identifiable natural person*" [1]. Under this definition, number plate data could be considered as personal information, and therefore special measures have to be taken to ensure its confidentiality. On the other hand, it is necessary to deploy mechanisms that deal with potential abuses of the deployed sensor network. Finally, **manageability** has to be considered, providing the system with mechanisms designed to ease the use of the platform.

---

[1] www.gdpr.eu/eu-gdpr-personal-data/

5.2 Architecture overview

Taking into account the aforementioned requirements, as well as the functional requirements, a layered architecture has been chosen for the platform. This will allow us to easily grow the system by adding new functionality while keeping it simple to understand, keeping the system maintainable with low coupling between layers. Besides, as the platform encompasses different deployment environments (capture devices, cloud platform), the layered approach proves to be advantageous for the organization of deployments and isolation of security threats.

The architecture is composed of 3 independent layers, with different goals, deployment environments, and levels of responsibility. Figure 5.1 offers a general diagram of the architecture. The *Perceptual layer* is the one that will control image capture devices, and allow them to communicate with the rest of the platform. It does so by means of a capture client that is installed in each capture device. The *Smart Management Layer* is the one with the highest degree of responsibility. It is in charge of receiving information from the capture devices, which is encapsulated in a format known as "*control packages*"; each *control package* is persisted and sent to the ALPR service for processing. It also provides the modules necessary for the definition of experiments. Finally, the *Online Monitoring Layer* will be in charge of providing the real-time visualization of the experiment. It should also provide an interface for the experiment definition module, allowing modifications to the experiment (which can include modifications to the configuration of the capture devices) before or during its execution.

Each of these layers consists of several modules, each one of them encapsulating a specific functionality that adds value to its layer, eventually contributing to its goal. The communication between layers is made through a system of requests; concretely, most modules of the *Smart Management Layer* will provide an interface that the other layers can use according to their needs. Each layer is independent of the others, and they don't know each other's implementation details and the way they work; this contributes to the scalability of the system, as it allows functionality to be added to each layer isolating the impact it provokes on the others.

In the following sections, the inner workings of each layer and module are detailed, focusing not only on their operation and goals, but also on the used languages and frameworks, as each layer will execute in different environments. In some cases, implementation details will be given; finally, this chapter will conclude by giving an overview of some of the design patterns used for the implementation of the platform.

5.3 Perceptual layer

The Perceptual layer is in charge of directly interacting with the capture devices or sensors that will be deployed on the field; therefore, it constitutes the lowest-level layer of the
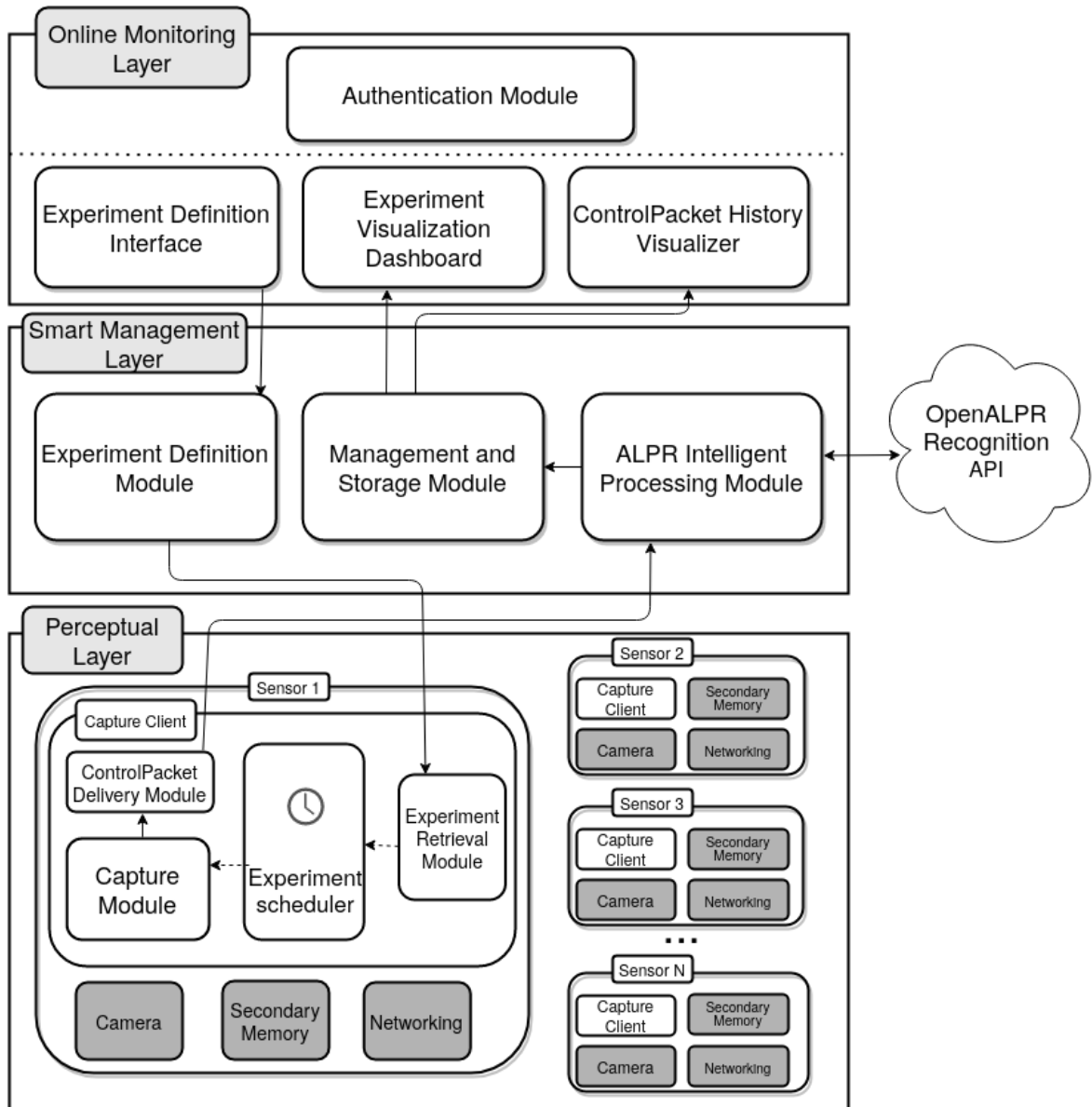
Figure 5.1: General diagram of the architecture

architecture. Concretely, it will consist of a set of capture clients; each one of them will be deployed to a capture device and will interact with its underlying hardware resources, according to the instructions (defined in *experiments*) received from the Smart Management Layer. As each capture client is deployed in a different capture device, clients are isolated from each other, and each client is unaware of the existence of other clients. Their responsibility is limited to retrieving their corresponding *experiment* from the upper layer and, according to its definition, take pictures and send them to the upper layer in the format of *Control Packages*, as well as storing the images in secondary memory. It is possible to reprocess the stored images in a stage after the experiment; this provides the system with fault-tolerance in the event of network connectivity issues.

Below, each module that the capture client consists of is analyzed, detailing their design and functionality

5.3.1 Experiment Retrieval Module

When a capture client starts its execution, it needs to retrieve a certain configuration from the upper layer, relative to both the camera parameters and the date and time when the camera must start recording. The Experiment Retrieval module is in charge of performing this task, as well as storing this configuration, scheduling the experiment so that it starts at the configured time, and periodically querying for configuration changes.

As there can be several capture clients, each one of them deployed in a different capture device, a mechanism for the identification of each client is necessary, so that the server can send back the configuration that corresponds to the client. In the case of our implementation of the client for the *Raspberry Pi Zero W*, the ID for each device is specified in a configuration file stored in a memory stick, among with the address of the server that hosts the Smart Management Layer. With this design, replacing a faulty sensor is a simple and straightforward task, as it consists only on provisioning a new capture device and capture client, and providing it with the right identification. This is related to the systematic requirement of evolvability.

Once the capture client has queried the Smart Management Layer for its configuration providing its ID, it receives an Experiment object. This object is encoded in the JSON format; each of its fields corresponds to a configuration parameter that will be used during the experiment. An example JSON Experiment object is the following:

```
1  {
2      "begTime": "2020-06-10T09:00:00",
3      "endTime": "2020-06-10T11:00:00",
4      "resolution": "1024x720",
5      "mode": "manual",
6      "exposure_time": 1000,
7      "freq_capture": 1000,
8      "iso": 320,
9      "rectangle_p1": [
10         280,
11         262
12     ],
13     "rectangle_p2": [
14         1024,
15         574
16     ]
17 }
```

Listing 5.1: JSON Experiment Definition example

Many of its parameters can be adapted to the physical location of the sensor and the environment conditions, having the flexibility of being able to define different configurations for each sensor: for example, it is possible to adapt to the position of the sensor, the weather conditions (rain, fog, etc.), and lighting level, among other factors. For instance, depending on the position of the sensor and the networking conditions it may be possible to adjust the resolution level: a capture device located with a good view to traffic, but with a slow network connection, may need a lower resolution level than a capture device located further away from traffic, but with access to an unmetered, fast Wi-Fi connection.

The *begTime* and *endTime* fields refer to the date and time when the device should start and stop recording, respectively. Their values are strings conforming to the ISO 8601 specification for the encoding of date and time-related data [Hou93]. The *resolution* field must indicate a capture resolution that is supported by the underlying sensor; in the case of the Pi-Camera used during the development of the platform, this resolution can go up to 3280x2464 pixels [2]. The *mode* field can be set to either "manual" or "auto"; when it is set to manual, it is possible to set the additional fields *exposure_time* and *iso*. The *exposure_time* indicates the fraction of a second that the light is allowed to enter the camera second in each capture; it can be adjusted in function of the current environment light. Similarly, the *iso* field indicates the sensitivity of the sensor to light; low values can be set for good luminosity levels, taking into account that bigger values can produce grain in the final images. Finally, the *rectangle_p1* and *rectangle_p2* fields allow to define, optionally, a subregion inside each captured image. This subregion will be the part of the image that is persisted to storage and sent in a *ControlPacket*, the rest of the image being discarded; this allows to save storage and bandwidth costs in scenarios where there is certainty about the area where the number plates will appear.

Once this JSON message is retrieved from the upper layer, it is the responsibility of the experiment retrieval module to parse the configuration and store it so that it can be used later. Concretely, there exists a *Config* class, that is instantiated when a configuration message is received. This class parses the message and stores the configuration; then, the upper layer is queried again periodically in order to check whether there are changes to the configuration, and in this case, the configuration object is updated.

After the configuration is parsed and stored, the experiment scheduler is called with the received *begTime* and *endTime* configuration values, and periodically updated as well whenever these values change.

Experiment Scheduler

The experiment scheduler is a submodule whose only responsibility is programming the execution of the capture, thus, initializing the "capture module" at the time indicated by the *begTime* field of the configuration; similarly, the capture must stop whenever the *endTime* is

---

[2]www.picamera.readthedocs.io/en/release-1.12/fov.html#camera-modes

reached.

As the capture devices that have been chosen for the development and testing of the platform have a UNIX-based Operating System, it was decided to use *cron* [3], a widely used daemon to execute scheduled programs. *Cron* is configured through *ctrontab* files, which contains each command that is scheduled to be executed, as well as the date and time when it should run, which can be periodic.

The "Experiment scheduler" submodule, therefore, consists on a method of the *Config* class which writes the corresponding entries in the *crontab* file: the entryoint of the "Capture module" is scheduled to be executed at *begTime*, and a *kill* signal is sent to this module at *endTime*. Besides, and additional check is performed to run the "Capture module" if the current time is between the *begTime* and the *endTime*, and to send a kill signal to any running "Capture module" instances should the current time be greater than *endTime*.

5.3.2 Capture Module

This module is in charge of interacting with the underlying camera hardware, configuring it according to the parameters found in the *Config* class, and starting the capture. It is also its responsibility to persist the captured images in secondary memory, and to provide them to the "ControlPacket Delivery module".

In the case of the *Raspbery Pi Camera Module v2* used in the capture devices where the client will be deployed, there is an open source Python interface to it, named Picamera [4]. Picamera provides an API that allows us to configure the camera module with all the parameters (ISO sensibility, exposure time, etc.) indicated in the experiment definition. Below is an example of how these parameters are set up:

```python
with picamera.PiCamera() as self.camera:
    config = Config()
    self.camera.resolution = config.resolution
    self.camera.framerate = config.freqcapture
    self.camera.exposure_mode = config.mode
    if config.mode == "manual":
        self.camera.iso = config.iso
        self.camera.shutter_speed = config.exposure_time
    else:
        self.camera.iso = 0
        self.camera.shutter_speed = 0
```

Listing 5.2: Camara parameter setup process

As can be observed, the Picamera API offers a straightforward and convenient way to configure the camera module with the needed parameters. The *iso* and *shutter_speed* parameters are set up only when the manual mode is activated; otherwise, default values are set

---

[3] www.man7.org/linux/man-pages/man8/cron.8.html
[4] www.picamera.readthedocs.io/en/release-1.13

up, as these parameters will be managed by the camera module directly. The capture module is responsible for periodically updating the camera parameters whenever the configuration changes; to achieve this, the Observer design pattern is used, which will be explained in the last section of this chapter.

In order to manage the output of the capture images, Picamera allows us to set a destination for the capture that can either be a file, or a Python stream. In our case, as the captures are stored in secondary memory and sent as ControlPackets, two outputs for the capture are indicated: a file, and a custom stream managed by the "ControlPacket Communication module". The code listing below shows the process that sets up the capture destinations, and that starts it.

```
controlpack_output = ControlPacketOutput()
disk_recording_output = "~/" + str(time.time()) + ".h264"

self.camera.start_recording(disk_recording_output, format="h264")
while(True):
    self.camera.wait_recording(config.freqcontrolpack, splitter_port = 2)
    self.camera.capture(controlpack_output, 'jpeg', quality = config.jpeg_quality_percentage)
```

Listing 5.3: Camera capture setup process

First, a *ControlPacketOutput* instance from the "ControlPacket Communication module" is initialized; then, the path of the file which will contain the capture result in secondary memory is stored in a variable. For this capture, the H264 [5] video format has been chosen, as it was considered that the final video would greatly benefit from inter-frame compression, given the fact that capture devices are generally located at fixed places. After the video stream has been initialized, an infinite loop, which manages the picture capture for the *ControlPacketOutput* is initialized. We take advantage of the *splitter_port* parameter of the *wait_recording* function, which allows us to operate on different recording streams without interference between them [6]. After the camera has waited for the amount of time specified in *config.freqcontrolpack*, a capture is performed to the *controlpacket_output* stream, indicating the JPEG format with the quality specified in *config.jpeg_quality_percentage*.

Picture compression greatly improves the memory footprint of the captured images, enabling the system to perform well even under constrained network conditions, such as the General Packet Radio Service (GPRS), which offers speeds up to 171 kbps. Compression levels of up to 65 can be used without compromising the ANPR process, being able to reduce the captured images by up to a factor of four.

---

[5] www.itu.int/rec/T-REC-H.264
[6] www.picamera.readthedocs.io/en/release-1.13/recipes2.html?highlight=splitter_port#recording-at-multiple-resolutions

Figure 5.2: Images compressed with different JPG compression levels, with filesize data. Compression levels below 60 significantly compromise the ANPR success rate. The first three digits of the license plate are blurred for data protection [ÁBSCV+20]

5.3.3 ControlPacket Communication Module

This module is in charge of receiving the images that the "Capture module" takes, creating ControlPackets, and sending them to the upper layer by calling the REST endpoint it provides. It must also provide a secondary operating mode in which, instead of processing the images as they are captured, it processes the video stored in secondary memory and creates the corresponding ControlPackets, which are sent to the server; this mode is used for capture devices with no network connectivity during the experiment. Finally, it is worth noting that, in the response to each sent ControlPacket, the upper layer may include the up-to-date Experiment definition; rather the processing it, the "ControlPacket Communication module" must pass it to the "Experiment retrieval module", so that changes in the configuration are correctly handled. This design ensures that the Experiment definition information is constantly updated during the capture, preserving the separation of concerns between modules.

In this context, a ControlPacket is defined as the basic unit of information that flows between the Perceptual layer and the Smart Management Layer. It is composed of the following fields:

- Client ID. The unique ID of the capture device where the client is running on.

- Timestamp. Time mark associated with the moment when the image was taken.

- Latency. Round-trip time (RTT) of the last ControlPacket that was sent. It is 0 for the first ControlPacket sent in a capture session.

- Image. Binary serialization of the captured image.

ControlPackets will be processed in the upper layer, being enriched with more fields that correspond to the "Smart Management" layer context, as will be explained in the following

section.

One of the main considerations that must be taken into account for the implementation of this module is the parallel and non-blocking nature of ControlPacket delivery. As images are captured continuously, they have to be sent to the upper layer immediately, regardless of the delivery status of previous images, which can mean that several images can be in process of being sent in any given amount of time. This is an important concern to take in consideration for the availability systematic requirement of the architecture, as issues in the network connectivity, or even in message receipts by the upper layer, should not negatively impact the image capture, or the delivery of future ControlPackets.

The mechanism that has been chosen to achieve this is by means of threading, using the Python module *threading* [7], from the standard library. It is worth mentioning that in CPython, which is the most widely used Python implementation, threading is limited in the sense that only one thread is executed in a given moment in time. This is due to the enforcement of a Global Interpreter Lock, or GIL. However, this limitation only applies to Python compiled code, and the execution of library methods written in the C language, as well as I/O operations managed by the kernel, are exempt from it. Therefore, the existence of such limitation should not negatively impact our use case of the "*threading*" library when using the CPython interpreter, as both the capture process and the delivery of ControlPackets by using network sockets will be able to run in parallel.

An unique, main thread, denominated "*ControlPacketThread*", is created as the image capture stream begins. Its function is to wait for the moment when there is a new image available in the picture capture buffer; for this, the *condition* functionality of the Python *threading* module is used. This mechanism allows a thread to wait until it receives a notification from another section of the program, then continue with its execution. This is often used as a lock to control access to some shared program state, when synchronization between concurrent threads is needed; however, in our use case, this functionality will be used so that the *ControlPacketThread* can get notified whenever a new image is captured.

```
class ControlPacketThread(Thread):

    # ... #

    def run(self):
        while(True):
            with self.output.condition:
                self.output.condition.wait()
                image_stream = self.output.image
                timestamp = self.output.timestamp
                Thread(target=self.send_controlPacket, args=(image_stream, timestamp,)).start()
```

Listing 5.4: ControlPacketThread definition

---

[7]www.docs.python.org/3/library/threading.html

After the notification, the *ControlPacketThread* creates a new thread whose only responsibility is delivering the captured image to the upper layer as a ControlPacket, and then continues waiting for new captures. This new thread is created with the *Thread* constructor, which allows us to specify, in the *target* parameter, a method to be run in the newly created thread, with the arguments indicated in *args*. Below is the definition of this method.

```python
def send_controlPacket(self, image_stream, timestamp):
      config = Config()
      r = requests.post(config.address_server + '/send_controlPack', headers = {
          'client_id' : config.client_id,
          'timestamp' : str(timestamp),
          'latency' : str(self.last_known_latency)
       }, files = {'image' : image_stream})

      self.last_known_latency = r.elapsed.total_seconds() * 1000

      config.update(json.loads(str(r.text)))
```

Listing 5.5: send_controlPacket method definition

The *send_controlPacket* method sends the ControlPacket as a HTTP POST request to the upper layer, indicating the *client_id*, timestamp and latency as headers, and the image data in the body of the request. It then updates the *last_known_latency* value so that following requests can use it, and calls the *update* method of the singleton *config* object with the response from the upper layer, so that the "Experiment Retrieval module" performs the corresponding updates in the experiment definition.

Finally, the ControlPacket Communication module has a batch processing mode, that processes an already stored video in H264 format, divides it in ControlPackets according to the defined capture frequency definition, and sends them to the server. It is independent of the previously explained online processing module which processes images as they are taken. Instead, its implementation is simpler, as there is no need to parallelize requests; it consists on a method that processes the video, selecting frames that are evenly distributed according to the capture frequency, converting them to the JPEG format, formatting them as ControlPackets and sending them to the upper layer.

5.4 Smart Management Layer

This layer has the largest amount of responsibility in the platform: its functionality ranges from handling the communication with the Perceptual Layer, providing each sensor with its Experiment definition and receiving ControlPackets, to processing each captured image, sending it to the OpenALPR service to receive the recognised plates. It must also be able to persist all the information related to each received ControlPacket and their ALPR result in a database, and provide them to the upper Online Monitoring Layer whenever it requests this information. Finally, it has to provide means to authenticate users in the platform.

For the deployment of this layer, the Google AppEngine PaaS (Platform-as-a-Service) was chosen. This provides this layer with the advantage of scalability, being able to auto-scale as needed depending on the dimension of the sensor network it is serving, and avoiding the complexity of having to provision and maintain our own servers to scale the system. The *flexible environment* option has been chosen in order to make this layer more portable among PaaS providers and to have better access to third-party libraries, as well as to be able to make use of multithreading, which is limited in the *standard environment*. This layer will be implemented as a web server, using the *Flask* framework, and it will provide different endpoints which will be consumed both by the lower and by the upper layers.

All the modules in this layer provide a set of HTTP endpoints, where requests can be sent to; this provides compatibility for the consumers of this API, as they can consume it as long as they are able to send standard HTTP requests. It is worth noting that, even though in the explanation of certain functionality, the specific layer that is supposed to consume it will be indicated, in reality, the API and its consumers are decoupled. This provides the architecture with extensibility, as new modules that are based on the functionality already provided by modules in this layer can be added without affecting the existing implementation.

5.4.1 Experiment Definition Module

This module is responsible for providing an interface for the definition and update of Experiment Definitions, as well as persisting this configuration and sending it to each capture device client that requests it. It should also provide validation of each received Experiment Definition, checking that they are consistent, and providing default values when necessary.

Experiment Definition objects were introduced in the "Experiment retrieval module" section. For the storage of these entities, as well as the rest of the entities of this module, it has been decided to use the NoSQL database provided by the **Google Cloud Datastore**[8] service; this will allow us to persist our data without having to provision or scale any database instance. Besides, as there is no need to perform relational queries on the stored data, the key-value nature of this database is not considered a handicap. In order to interact with this database, the *ndb*[9] Python library will be used, which gives us an interface that allows us to define entities, the keys that identify them and the properties they contain, as well as to query the needed information.

The *ndb* library provides an API to define entities that is based on extending a class called *Model*. Each entity can have one or more properties, which are named values of one of the supported datatypes (integer, float, datetime, string, or a reference to another entity, for instance). These properties are indicated, in the model definition, as class variables. Below

---

[8]www.cloud.google.com/datastore
[9]www.googleapis.dev/python/python-ndb/1.4.2/index.html

is an extract of the ControlPacket class definition:

```python
from google.appengine.ext import ndb

class ClientConfig(ndb.Model):

    # ... #

    begTime = ndb.DateTimeProperty()
    endTime = ndb.DateTimeProperty()
    freqCapture = ndb.FloatProperty()
    freqControlPack = ndb.IntegerProperty()
    mode = ndb.StringProperty()
    exposure_time = ndb.IntegerProperty()
    iso = ndb.IntegerProperty()
    rectangle_p1 = ndb.IntegerProperty(repeated=True)
    rectangle_p2 = ndb.IntegerProperty(repeated=True)
    jpeg_quality_percentage = ndb.IntegerProperty()

    def to_dict(self):
        # ... #
```

Listing 5.6: ClientConfig class definition

As can be seen, *ndb* provides a set of classes, such as *IntegerType*, that correspond to each datatype that a property can have. It is possible to define array datatypes with members of the specified type with the *repeated* optional parameter; this is done with the *rectangle_p1* and *rectangle_p2* properties. This way of defining a model is similar to a DDL (Data Definition Language) table definition in relational databases. The *to_dict* method provides a convenient way to transform a *ClientConfig* object into a dictionary. *ControlPacket* entities are then created by instantiating the *ClientConfig* class and calling the *put()* method; calling this method on an entity that was already created has the effect of updating it. Below is an example of how this feature is used when updating a previously defined client configuration:

```python
import utils
from backend.models.ClientConfig import ClientConfig

class ClientConfigManager(utils.Singleton):

    @staticmethod
    def update_config(client_id, mode=None, exposure_time=None, iso=None, rectangle_p1=None,
         rectangle_p2=None):

        clientConfig = ClientConfig.get_by_id(client_id)

        if mode is not None:
            clientConfig.mode = mode
        # ... set more parameters ... #

        clientConfig.put()
```

Listing 5.7: ClientConfigManager class definition

The *update_config* method begins by querying the database by calling the *get_by_id* method of the *Model* class, which *ClientConfig* inherits from. This method is one of the ways that the *ndb* API provides to query the database. Then, each entity attribute is overwritten with the value that is passed as input, and finally, the *put()* method of *ClientConfig* is called in order to update the entity.

More broadly, the *ClientConfigManager* class introduced above is a "manager" class in the context of the web server where this layer is being implemented. A common convention in the programming model used in the implementation of web applications is to separate the code receiving HTTP requests, parsing and routing them, from the code actually processing them. The former is encapsulated in classes and/or methods generally known as "routers" or "handlers", while the latter is encapsulated in "controllers" or "managers". Therefore, the code introduced above is part of a "manager" that gets called by a "handler" whenever it receives an HTTP request; below is an extract of the code of this "handler":

```python
from flask import Blueprint, jsonify, make_response, request

bp_config = Blueprint("bp_app_config", __name__)

@bp_config.route('/update_config', methods=['POST'])
def update_config():
    client_id = int(request.values.get('client_id'))
    mode = request.values.get('mode') if request.values.get('mode') else None
    # ... retrieval of parameters ... #

    ClientConfigManager.update_config(client_id, mode=mode,
                                      exposure_time=exposure_time,
                                      iso=iso,
                                      rectangle_p1=rectangle_p1,
                                      rectangle_p2=rectangle_p2)

    return make_response(jsonify({'status': 'updated'}), 200)
```

Listing 5.8: /update_config endpoint handler

The Flask framework greatly simplifies the low-level handling of HTTP requests, easing the development of "handler" functions. In the above code listing, after importing the needed classes and methods from Flask, be begin by creating a *Blueprint*. In Flask, Blueprints are used to ease the development of modular applications [10]; their basic concept is that they are meant to record operations to be executed when dispatching requests. and generating URLs from one endpoint to another. The Blueprint *bp_config* will be in charge of registering all the operations related to the Experiment Definition Module.

In order to register which method should be run in response to a request, function decorators are used [11]. Function decorators are syntax sugar for composing methods; concretely,

---

[10]www.flask.palletsprojects.com/en/1.1.x/blueprints/
[11]www.python.org/dev/peps/pep-0318/

before the definition of a new function, a line starting by the @ sign indicates that the new function is the result of composing the body definition with the function indicated after @. For example, the code in the following listing:

```python
@dec
def func(*args):
    pass
```

Is equivalent to:

```python
def func(*args):
    pass
func = dec(func)
```

Without the intermediate assignment to the variable *func*. It is worth noting that this pattern is possible thanks to the fact that, in Python, functions are first-class objects, which means that it is possible to assign them to variables, allowing us to pass them as arguments to other functions as well as returning them from other functions. This is a basic feature found in functional programming languages, which Python draws some inspiration from, allowing us to create new abstractions based on this concept.

Resuming the explanation about the *update_config()* handler function, it can be observed that it is decorated with the *route* method of the *bp_config* Blueprint; as arguments to this decorator function, the *'update_config'* string is passed, which represents the endpoint to be handled, and the array of methods to be handled, in this case, POST. As can be observed, the Flask framework, with the *Blueprint* class and by using function decoration, provides us with a very simple way to define routing in an HTTP web server.

As *update_config* is a handling function, its responsibility is limited to extract the values of the required parameters from the HTTP request, invoke the corresponding method of *ClientConfigManager*, and giving a HTTP response to the client that performed the request. As can be observed in its definition, those tasks are performed in order: it begins by obtaining the *client_id*, which is the only required parameter, and continues by receiving the rest of the parameters, which are optional (*mode*, *exposure_time*, *iso*, *rectangle_p1*, and *rectangle_-p2*. Then, it calls the *update_config* method, and it finishes by returning a new *Response* object by using the auxiliar *make_response* and *jsonify* functions: the response consists on a confirmation that the desired entity has been updated, with the 200 HTTP code.

Apart from defining and updating experiment definitions (also known as client configurations), the Experiment Definition module has the responsibility of providing these definitions, both when explicitly requested, and as a response whenever a capture device communicates by sending a Control Packet. In order to accomplish this, a new "handler" function is defined:

```
from flask import Blueprint, jsonify, make_response, request

bp_config = Blueprint("bp_app_config", __name__)

@bp_config.route('/get_config<int:clientId>', methods=['GET'])
def get_config(clientId):
    config = ClientConfig.get_by_id(str(clientId))
    if config:
        return jsonify(config.to_dict())
    else:
        abort(404)
```

Listing 5.9: /get_config endpoint handler

As one can see, the interface of the Experiment Definition module is expanded with a new endpoint, */get_config*. Whenever the server receives a request to this URL followed by the ID of a client, it will give a response containing the JSON object corresponding to the experiment definition of the requested client ID, or an empty response with the HTTP 404 status code. In this case, no manager code is called from the handler; instead, the *ClientConfig* model is directly called from the handler, querying for the configuration of the corresponding client. Therefore, the separation of concerns between "handler" and "manager" is broken in this particular case in favor of conciseness; however, a "manager" can be created at any time should complexity increase during the evolution of the platform.

Finally, when it comes to reporting the status of the Experiment Definition to the capture devices each time they communicate, this module provides the means to query this definition to the ALPR Intelligent Processing Module, which is in charge of the communication with the capture devices, by means of the *ClientConfig* class.

5.4.2 ALPR Intelligent Processing Module

This module is in charge of the receipt of ControlPackets from the Perceptual Layer, the request of recognized plates to the OpenALPR Cloud service, and the persistence of the images and recognition results. Similarly to the rest of the modules of the Smart Management Layer, this module provides a set of endpoints that expand the interface of the layer, allowing modules from other layers to call them as needed.

Below, the*ControlPacket* model is introduced, which, similarly to the previously introduced *ClientConfig* model, makes use of the *ndb* library for its definition. The *ControlPacket* model has all the fields which are received from the Perceptual Layer (*client_id*, *timestamp* and *latency*), plus three fields which contain information relative to the plate detection process, which are *plate*, *confidence* and *json_response*. Some of the fields are defined passing the parameter *indexed=True* to its property class; concretely, the fields *client_id* and *timestamp*. The reason for this is that these two fields are considered to be the unique identifier of each ControlPacket, and queries will be performed based on them.

```python
from google.appengine.ext import ndb


class ControlPacket(ndb.Model):
    client_id = ndb.IntegerProperty(indexed=True)
    timestamp = ndb.DateTimeProperty(indexed=True)
    latency = ndb.FloatProperty(indexed=False)
    plate = ndb.StringProperty(indexed=False, repeated=True, required=False)
    confidence = ndb.FloatProperty(
        indexed=False, repeated=True, required=False)
    json_response = ndb.TextProperty(indexed=False)

    def to_dict(self):
        # ... #

    def get_packet_id(self):
        return str(self.client_id) + "_" + '%f' % (utils.unix_time_millis(self.timestamp) / 1000)

    @staticmethod
    def query_by_packet_id(packet_id):
        (client_id, timestamp) = packet_id.split("_")
        client_id = int(client_id)
        timestamp = datetime.datetime.fromtimestamp(float(timestamp))
        return ControlPacket.query().filter(ControlPacket.client_id == client_id).filter(
            ControlPacket.timestamp == timestamp).fetch()[0]
```

Listing 5.10: Controlpacket model definition

Indeed, because of the design of Google Cloud Datastore, queries run on predefined index tables that are periodically updated, and it is, therefore, necessary to define which properties must be indexed at the moment of the creation of the model. Besides, two methods are defined for the use of our application, *get_packet_id* and *query_by_packet_id*. They are based on the concept of a "packet_id", which consist on the concatenation of a packet's *client_id* and its *timestamp*.

For the storage of the captured images, Cloud Datastore is not used, as can be observed by the fact that there is no *image* field in the *ControlPacket* model class. Instead, the object storage service **Google Cloud Storage** [12] has been used, as it provides a simple and cost-effective way to store binary files, and prevents some of the downsides that come with storing binary blobs in Cloud Datastore (such as the maximum size). For integration in Python programs, the *cloudstorage* library is used, which provides a very simple interface to the Cloud Storage service. Below, the *ControlPacketManager* class, which contains all the controller functions in this module, is introduced, along with the *save_image* function, which saves the provided image in Cloud Storage:

The *cloudstorage* library provides a very simple interface to the Cloud Storage service; indeed, in our use case, creating a new object consists on calling the *open* method with its path, writing mode and optional *content_type*. This returns a class whose *write* method is

---

[12]www.cloud.google.com/storage

called passing it the binary image as an argument, and the process is finished by a call to *close*.

```python
import utils
import cloudstorage as gcs


class ControlPacketManager(utils.Singleton):
    def save_image(self, packet_id, image):
        gcs_file = gcs.open("/{bucket_name}/{packet_id}".format(
            bucket_name=constants.BUCKET_NAME, packet_id=packet_id), "w", content_type="image/jpeg")
        gcs_file.write(image)
        gcs_file.close()
```

Listing 5.11: ControlPacketManager definiton

It is worth mentioning that, to build the path of the object in Cloud Storage a *bucket name* is concatenated with the ID of the packet. In Cloud Storage, buckets are the basic containers were data is stored: each object is contained within a bucket [13]. Bucket names are globally unique; in our case, the bucket name is stored in a configuration constant.

Requests to the OpenALPR service for license plates are performed by calling the API REST that it provides. Documentation about the structure of the requests and responses, as well as code samples, is available in the official documentation [14]. The ALPR Intelligent Processing Module processes the response from OpenALPR, extracting the candidate license plates and their respective confidence value, an integer ranging from 0 to 100; then, it stores them in the ControlPacket *plate* and *confidence* fields. The raw response from the OpenALPR service is stored in the *json_response* field in case further analysis of this data is desired.

Now that the *ControlPacket* model has been introduced, as well as the image persistence in Cloud Datastore and the number plate request to the OpenALPR service, a general idea of the processes that are executed when every ControlPacket arrives has been exposed. Concretely, a function handler is triggered when HTTP POST requests arrive at the endpoint "*send_controlPack*", and forwards all the necessary parameters, which are included in the request body, to the *add_controlPacket* method of the *ControlPacketManager* class, previously introduced. This method is the one in charge of orchestrating all the processing of a ControlPacket, querying the OpenALPR service and persisting the ControlPacket to Cloud Datastore and the image to Cloud Storage. Finally, the handler of the "*send_controlPack*" endpoint retrieves the *ClientConfiguration* that corresponds to the client ID that sent the ControlPacket, and includes it in the response so that the client has the most up-to-date Experiment Definition.

---

[13]www.cloud.google.com/storage/docs/key-terms#buckets

[14]www.doc.openalpr.com/api.html#carcheck-api

5.4.3   Management and Storage Module

The role of this module is to offer certain functionality that the upper layer will need to correctly authenticate end users of the platform and to access the ControlPacket history of the different capture devices.

ControlPacket History Retrieval Submodule

This submodule is in charge of providing the means to audit in real-time the stored ControlPackets and to perform analysis on them. It will extend the API of this layer with new endpoints meant to be consumed by the upper layer; however, no new models will be added, as all the required functionality is covered by the already existing ControlPacket entity. The handler methods of the newly created endpoints will be organized under the *bp_app_get-Data* Blueprint, and will make use of the already existing *ControlPacketManager*, extending it with new methods when needed.

The first endpoint that is added is called */get_Data*. Its functionality is to return the last ControlPacket received by each capture device that is registered in the system, and that thus has an Experiment Definition. Optionally, it is possible to pass a Client ID to this endpoint, calling it in the form of *get_Data/<int:client_id*, to get the last ControlPacket of the capture device with the specified Client ID. The handler method then calls a static method of the *ControlPacketManager* called *get_last_controlpackets*, that accepts a list of Client IDs as an argument, representing the capture devices whose last ControlPackets we want to get. The manager class then calls the model *ControlPacket*, which makes use of the *filter* and *order* methods provided by *ndb* to get the required ControlPacket; the Datastore table in turn is queried, filtered by Client ID and ordered by timestamp in a descending manner.

The next endpoint provided by this submodule is */get_image<packet_id>.jpeg*. Its objective is to provide the Online Monitoring Layer with the means to, once it has received a ControlPacket from a certain capture device, access the captured image in JPEG format. As was explained in the ALPR Intelligent Processing Module, capture images are stored in Google Cloud Storage, a different datastore than the one where ControlPackets are persisted. Thanks to the concatenation of the Client ID and the timestamp of a ControlPacket, its Packet ID is built, which is equivalent to the path of the stored image, plus the ".jpeg" extension. This new endpoint takes the path of the desired image as an input and returns the image in binary format. It gets the image thanks to the *get_image* method of the *ControlPacketManager* class, which uses the *cloudstorage* library to get the image file from Cloud Storage.

Another functionality that should be offered by this submodule to the Online Monitoring Layer is the possibility of sending regular updates about the latest ControlPackets that have been received from each capture device. This will give the user interface a convenient way to periodically update the information about each capture device, allowing the user to receive constant, updated information about the capture process. To achieve this, one of the possi-

bilities that was considered was sending update notifications from the server to the client by using the WebSocket protocol [FM11]. This protocol provides a full-duplex communication channel over a single TCP connection, and is designed to work over the HTTP 80 and 443 ports; it is, therefore, compatible with HTTP, and a HTTP connection can be changed to a WebSocket connection with the HTTP upgrade header. The fact that a server can initiate communication with a client implies a break with the classical client-server model; however, it can prove to be superior to other alternatives such as long polling in this kind of scenario where clients wait for updates from a server. Despite this, applications can face several obstacles when using this protocol, as it has not been adopted in a widespread way yet. For example, WebSocket connections may fail when traversing restrictive firewalls or proxies that don't support them; this is notoriously common in the case of connections over mobile, 2/3/4G networks. [ME12] analyses this and other issues in the adoption of WebSockets for handheld devices, particularly focusing on battery save issues. Application designers usually work around these issues by implementing mechanisms that use standard HTTP connections when the WebSocket protocol doesn't work or is unreliable. However, given the fact that a big number of end-users of the platform are likely to connect through a mobile network, it was decided that the complexity of adding support for the WebSocket protocol was unnecessary.

Instead, it has been decided to provide the update functionality in a standard HTTP endpoint, that will be queried periodically by the Online Monitoring Layer. Concretely, a Packet ID will be received, and the latest ControlPacket that has been received by the capture device that sent the original Packet, if any, will be returned. The fact that this endpoint doesn't return any ControlPacket if there are no updates allows saving network traffic in a context where frequent updates may be requested by multiple clients, to many capture devices. This functionality is exposed in the */get_update* endpoint, which takes the Packet ID from the *last_packet_id* parameter in the request body, and calls a method in the *ControlPacketManager* called *get_update* passing the Packet ID to it; if a new ControlPacket is returned from the manager function, it is returned in the response, and otherwise an empty response with the HTTP 304 (not modified) status code. Overall, this solution allows the upper layer to query for updates with a high frequency, allowing the user to receive information about the capture status in a reasonable amount of time.

Finally, for the historical analysis of the received ControlPacket, functionality allowing to get a list of the last *N* ControlPackets that were sent by a particular client is required. This will let the user visualize the evolution of an experiment, being able to see all the recent images and the detected license plates that were detected, which will prove to be useful information for any adjustment that may be required in the capturing device. This functionality is implemented with the */get_history* endpoint, that accepts as parameters the *client_id*, the desired number of ControlPackets to get (*n_elements*), and a *cursor*. This *cursor* is an op-

tional string, and is used for paginated requests: when the number of ControlPackets stored in the database is greater than *n_elements*, the *ndb* library returns a cursor that can be used in a subsequent request to receive new packages. Overall, from an implementation perspective, the way in which the *ControlPacket* model is queried is very similar to the */get_data* endpoint, as a filter by the Client ID and a sort by timestamp are applied.

User management submodule

The role of this submodule is to support the authentication of users that are authorized to use the platform. The Online Monitoring Layer will perform requests to this submodule, and, as a response, it should receive an indication of whether or not a certain combination of user a password is valid, and the user is allowed to use the platform. As part of user management, it was decided not to include any permission and/or role system that would allow to control which functionality of the platform each user is allowed to access; instead, every user is allowed to access all the functionality of the system. This decision was taken due to the fact that all the users of the platform perform similar tasks, and moreover, no destructive operation (for example, the deletion of ControlPackets or Experiment Definitions) is allowed in the platform.

For the implementation of this functionality, a new model, called *AppUser*, is created. It has two string parameters: *username*, which is indexed, and *password*, which consists on the hash of the password string that the user chose. A handler function, listening on the */user_login*, reads the *username* and *password* fields of every POST request it receives, and queries the *AppUser* to compare whether a user with the received username exists, and whether the passwords match, returning the appropriate response. Overall, this design for the User Management Submodule provides a basic login functionality that covers the needs of the application, and that is easily extensible.

5.5 Online Monitoring Layer

The role of this layer is to let the user monitor the state of a traffic analysis experiment. To ease as much as possible the use of the system, increase compatibility, and avoid the installation of software by the user, the interaction with this layer is carried through a web browser. The functionality of this layer can be divided into two blocks: first, the functionality related to the definition of experiments, which will consist of a set of forms aimed to make the selection and definition of the different parameters as convenient as possible. Second, the functionality related to the monitoring of ongoing experiments, which must include both the receipt of updates of the last ControlPackets received by the capture devices, as well as the possibility to see the history of captures of a particular device.

The systematic requirement of manageability is of the utmost importance in this layer of the architecture, as it will be the one that the end user will interact with. One of the main

challenges to be addressed is the fact that this interaction can be performed through many different devices. For example, a laptop or desktop computer may be used to set up the experiment definitions for multiple capture devices, or to see the history of captures of an experiment once it has finished. On the other hand, operators of capture devices that connect from the field will generally make use of mobile devices, which can range from smartphones to tablets. Therefore, it is necessary to design the user interface with the idea that it will need to work under many different screen resolutions and screen sizes. In web design, an interface with this characteristic is considered to be "responsive" [Gar11], and it proves to be a fundamental requisite in most modern web interfaces.

As techniques to create responsive webpages have become widespread, frameworks that simplify performing this task have emerged. For the development of the modules explained below, the *Bootstrap* [15] framework has been chosen. It is an HTML, CSS and Javascript open-source framework used for the creation for responsive design, providing simple, intuitive and clean results. It is based on dividing the element of the page into columns with customized sizes, which are adapted depending on the size of the device that renders the page. This is a very good fit for some of our use cases, as it will allow a simple way to organize the information about the different capture devices in a grid, for instance. *Bootstrap* was initially developed by Twitter in 2011, and it is hosted, maintained and under development in a GitHub repository [16] under the MIT license.

This layer will be hosted in the Google App Engine PaaS, in instances from the standard environment, using the Flask framework. Webpages served by this layer need to include dynamic content. One of the ways it is achieved through the layer is by the use of templates when generating a webpage in the server; concretely, the Jinja2 [17] template system will be used. Jinja2 allows us to include Python expressions embedded in HTML templates, which will be evaluated in the server when a page is requested. This will allow us to work with the responses from the Smart Management Layer and include them in the different HTML templates in a very convenient way. Once the web browsers receive the different dynamically generated webpages from the server, in some cases, AJAX queries will be performed to the server, for instance, to support interactive functionality, or to receive updates; this will be detailed below, in the description of each module.

5.5.1 Authentication Module

The role of this module is to provide an interface allowing the users of the platform to log in, as well as verifying that each request made to the rest of the layer has been performed by an authenticated user. Logged in users receive a session cookie, which the web browser will attach to subsequent requests to the server.

---

[15]www.getbootstrap.com/
[16]www.github.com/twbs/bootstrap
[17]www.jinja.palletsprojects.com/en/2.11.x/

In order to ease the development of this module, the *Flask-Login*[18] library was used. It provides several methods that handle the session cookie creation and validation processes while implementing standard security measures; it also supports alternative request authentication methods, such as HTTP 'Authorization' headers. It provides an API based on function decorators that is simple and concise, allowing us, for instance, to easily define the endpoints that require authentication by simply adding a decorator annotation on top of their handling function definition.

The user login functionality provided by this module can be reached through the */login* endpoint. A POST request to it must be performed in order to log a user in the system, with the *username* and *password* in its body. The handler function will hash the password with the SHA-256 function and query the appropriate route of the Smart Management Module to check whether or not the provided combination of username and password is correct. When so, a session cookie will be built using the default class provided by the library (called *UserMixin*), and the method *login_user* will be called to return a response; otherwise, a response with the 401 (Unauthorized) HTTP status code will be returned. On the other hand, when a GET request arrives, a static webpage containing a login form, called *login.html*, is returned.

5.5.2 Experiment Visualization Dashboard

This module provides a page containing a dashboard in which a logged-in user can see a summary of all the capture devices and the last ControlPacket each one of them sent. This page serves as the main page of the platform, the Authentication Module redirects to it after the user logs in, and all the rest of the functionality is accessible through it. All the information about the last sent ControlPackets is rendered on the page, including the captured image, the candidate plates, and the latency. The page is organized as a grid; in each cell, the information about a capture device is provided. *Bootstrap* automatically resizes the grid, adding columns and rows depending on the screen size of the device where the browser runs. Figure 5.3 shows a mockup of the Experiment Visualization Dashboard, as visible when rendered in a desktop web browser.e

The Experiment Visualization Dashboard is accessible through the /index endpoint, which takes an optional *client_id* parameter in case the user only wants to load the information about a single capture device. When a request arrives, the handling function performs a call to the History Retrieval Module of the Smart Management Layer, in order to get the last ControlPacket sent by the different capture devices. Once this information is received, the *index.html* template is rendered and returned to the web browser.

The Python code embedded in the *index.html* template handles the processing of the ControlPacket data returned by the Smart Management Layer, generating a set of *div* elements

---

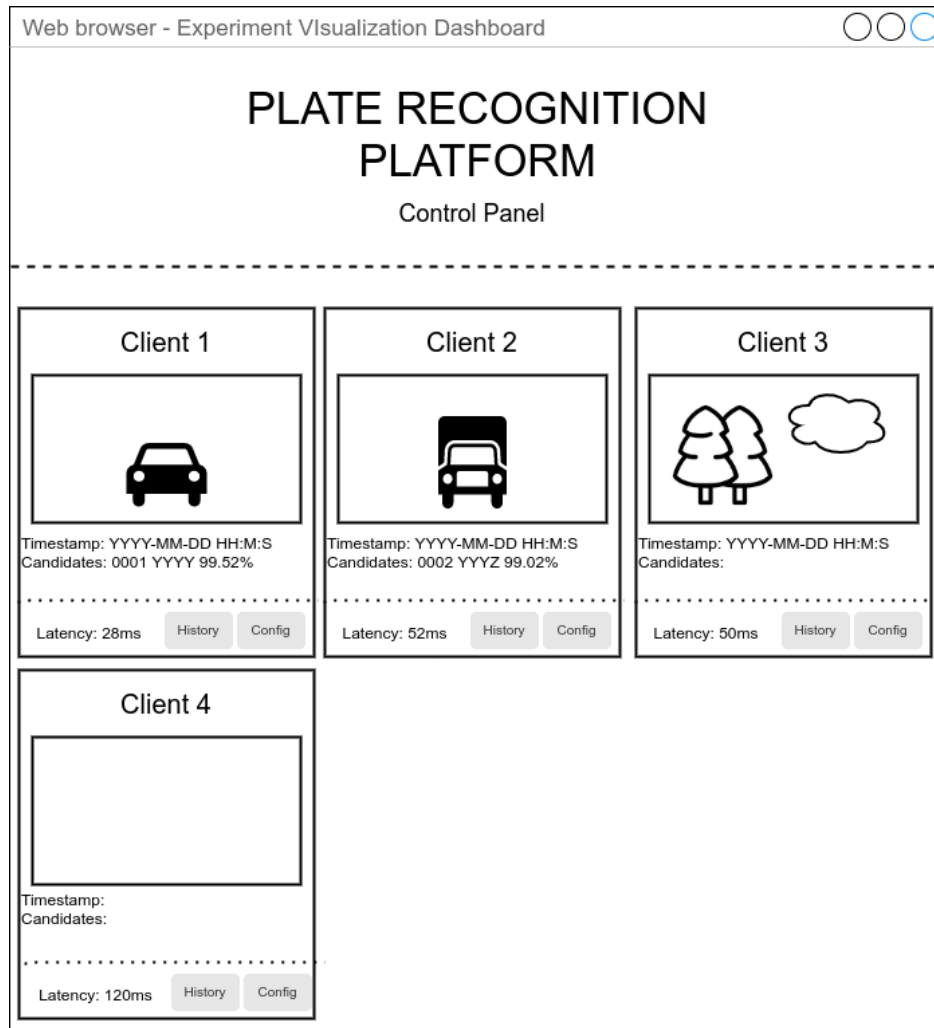[18]www.flask-login.readthedocs.io/en/latest

Figure 5.3: Mockup of the Experiment Visualization Dashboard

in order to correctly show the information on the page. The template also imports a set of Javascript functions that are used to request periodical updates; concretely, for each capture device, a request is performed to the */get_Update* endpoint of the server, which queries the Smart Management Layer for updates. The *jQuery* library is then used in the web browser in order to update the corresponding elements.

5.5.3 Experiment Definition Interface

This module is in charge of providing a form that will allow the user to visualize the current configuration, also known as experiment definition, of a capture device, and to update it. In order to ease as much as possible the filling of the different parameters, and to prevent errors, only allowed values should be allowed to be introduced.

A template with the name of *config.html* contains the form to be filled up. It is contained in an element which has the *modal* class, which is predefined by *Bootstrap*; by showing the form in a modal window, the transition between this interface and the main Experiment Visualization Dashboard is smoother. Embedded Python code controls the different *option*

59

and *input* HTML tags that must be enabled or checked depending on the current configuration state. This template is rendered on GET calls to the */clientConfig* endpoint, while POST calls to the same endpoint will submit a form; when it happens, the handling function verifies the received fields, and then performs the corresponding call to the Experiment Definition Module in the Smart Management Layer.

5.5.4 ControlPacket History Visualizer

The goal of this last module of the Online Monitoring Layer is to provide a visualization of the last ControlPackets sent by a particular capture device. This visualization will be shown when clicking on a specific link of the Experiment Visualization Dashboard, and will be opened as a modal window. A default number of ControlPackets will be shown, and the user will be able to load more on-demand when scrolling down in the visualization.

The endpoint that will return the visualization has the name of *get_Update*, and it will get the required information from the Management and Storage Module of the Smart Management Layer. Unlike other endpoints in this layer, *get_Update* does not render a template returning a set of HTML elements; instead, it returns a JSON object, that a Javascript method in the web browser side processes. This decision was taken in order to simplify the lifecycle of the HTML elements created for this visualization: as they are not recreated each time that *get_Update* is called, but rather updated, performance is increased (network traffic reduces as well) and the implementation is simplified. A listener is set up so that every time the user scrolls through all the available ControlPackets, more are requested, based on the value of the *cursor* that was returned by the last call to *get_Update*, achieving pagination.

5.6 Design patterns

Design patterns are reusable solutions to commonly found design problems that can happen in many different situations, facilitating the creation of quality designs. A design pattern is defined as a set of techniques that solve common problems in software development [Pre95]. An adequate choice of design patterns during the design of the architecture can contribute making it simpler and, therefore, improve its reusability, scalability, and maintainability. Below, different patterns that have been chosen in the development of the platform are introduced.

5.6.1 Singleton pattern

Singleton is a creational pattern that solves two different problems: ensuring that a class only has a single instance, and providing a global access point to that instance. As it solves these two different problems at the same time, some authors consider it as an anti-pattern, as it breaks the *Single Responsibility Principle*, and have found that its use is often correlated with more fault-prone code structures [Vok04]. However *Singleton* remains a useful design pattern when used under specific use cases, such as access control to a unique, shared

resource (for instance, a database, or a file).

In the architecture for the platform, the *Singleton* pattern is used, for instance, in the *Config* class that is used in the Perceptual Layer. There, a unique object is used by the three modules of the layer, for different purposes. Concretely, the Experiment Retrieval Module populates the created *Config* instance after it receives the configuration, and updates them when needed. The Capture Module, on the other hand, needs to read the *Config* object in order to set up the camera parameters for the capture. Finally, the ControlPacket Communication Module reads the experiment configuration each time it needs to send a ControlPacket.

Therefore, the use of the *Singleton* pattern is considered to be justified in this case, as it encompasses both issues that the pattern addresses: a single instance must be provided for a class, and it is accessed by multiple points in the code. The usage of the pattern consists on *Config* inheriting from a *Singleton* class, whose code is exposed below.

```python
class Singleton(object):
    """ Singleton base class to extend from """
    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls, *args, **kwargs)

        return cls._instance
```

Listing 5.12: Singleton Pattern implementation

In Python, the *__new__* method of a class is called at the time of the creation of a new instance, and returns the new instance (unlike *__init__*, which does not return anything as it is in charge of initializing an instance once it has been created). By using this method to return an existing instance of the class should it exist, or otherwise create and return it, it is ensured that only one instance of the child class exists, while providing a global unique access point to it.

5.6.2 Observer pattern

Observer is a behavioral design pattern allowing to define a subscription mechanism that allows one or several objects to be notified about an event that the object they are observing reports. The object that is being observed (also known as notifier, or publisher) is not aware of the type of the objects that observe it (also known as listeners, or subscribers). This is achieved because subscribers implement a common interface that allows them to be notified, allowing the publisher to call this notifying interface in a uniform way. This way, decoupling between the publisher and subscribers is attained.

In the proposed architecture, there are two cases where the *Observer* pattern is used, with different implementations; both of them are found in the Perceptual Layer. The first one is

related to the *Config* class; as it can be updated at any moment by the Experiment Definition Layer that is responsible for it, but the Capture Layer depends on its attributes in order to properly manage the capture process, the *Observer* pattern proves to be useful to solve this issue. In this case, the pattern is implemented by means of a class *Subject*, that *Config* will inherit; this class will provide methods (*attach* and *dettach*) allowing observer objects to sign up for updates, and a *notify* method allowing the *Subject* to alert the subscribers. The *CameraManager* will act as the subscriber, receiving updates whenever the *Config* notifies them, and modifying the capture process accordingly. It is worth noting that, as the reader may have noted, the *Config* class inherits from both the *Singleton* and the *Subject* classes: indeed, multiple class inheritance is allowed in Python, which allows to easily implement those two patterns in this case.

The second case where this pattern is used in the Perceptual Layer is in the ControlPacket Delivery Module; indeed, as was explained, the use of the *Condition* method of the *threading* library allows the *ControlPacketOutput* class, which contains a buffer where captured images are stored, to trigger a notification to the *ControlPacketThread* class, which will then create a new thread to process the image stored in the buffer.

Overall, the use of the observer pattern in these two examples allows an easy way to trigger notifications between objects that keeps simplicity, but still ensures isolation between each module of the layer.

Chapter 6

Results

This chapter comprises two parts. The first one describes the developed prototype of the project, including the capture sensors that were used and the modules and functionalities that have been implemented, including the user interface. It will also give an example of a real-world traffic analysis experiment that was carried through the platform. The second part describes the evolution of the project during its development period, specifying the iterations involved in it and the tasks performed in each one of them. The final part shows some statistics related to the work made in the code repository where the project is hosted.

6.1 Project Prototype

The end result of this project is a fully functional platform, consisting of a server and user interface that can be deployed to Google App Engine, and a capture client that can be installed on a Raspberry Pi Zero W. After the user takes the necessary steps to set up the capture devices (1), he or she can log into the platform and see the dashboard with all the capture devices and their last communication (2), define experiment definitions for each capture device (3), and see the full history of Control Packets for any given device (4).

6.1.1 Capture devices

First, the capture device model that has been used during the development and testing of the platform is shown. The Raspberry Pi Zero W is visible, with a PiCamera attached to it and a 5V external battery used to power it, in figure 6.1.

The capture device is contained in a custom-made enclosure. Added to the described components, a USB drive must be added containing the initial configuration for the capture device, which includes information about the Wi-Fi network it must connect to, the address to connect to the Smart Management Layer, and optionally, the initial configuration of the experiment definition. It is worth noting that the recorded video will be stored in this USB drive.

Two different capture devices were used for the testing of the platform, in order to prove the possibility of easily replicating the software installation for a second device, as well as the capability of the whole platform to handle multiple capture devices.

Figure 6.1: Interior of an image capture device

6.1.2 Login and dashboard

Figure 6.2 shows the login page of the platform; it consists on a simple page consisting on an *username* and a *password* field. These credentials are provided by the platform administrator, as there is no need for functionality to sign up to the system.

Figure 6.2: Login page of the platform

Once the user has logged in, he or she has access to the Experiment Visualization Dashboard, and can visualize the list of capture devices and their last sent picture. Figure 6.3 shows it for the case where there is only one client; as new clients communicate, the dashboard will be filled in a grid pattern, adapting to the screen size of the device that renders the interface. Figure 6.4 shows a zoomed-in image of how a capture device is represented in the dashboard; in that case, the captured image contains no detected number plates. It is possible to see the timestamp when the image was taken, the latency of the ControlPacket, and the

"History" and "Config" options allowing to access the Experiment Definition Interface and the ControlPacket History Visualizer, respectively. However, from this page, it is possible to draw a rectangle representing the area of an image where number plates are expected, by dragging and dropping the cursor over the desired region of the image, which will be handled by the Experiment Definition Interface module.

Figure 6.3: The Experiment Visualization Dashboard
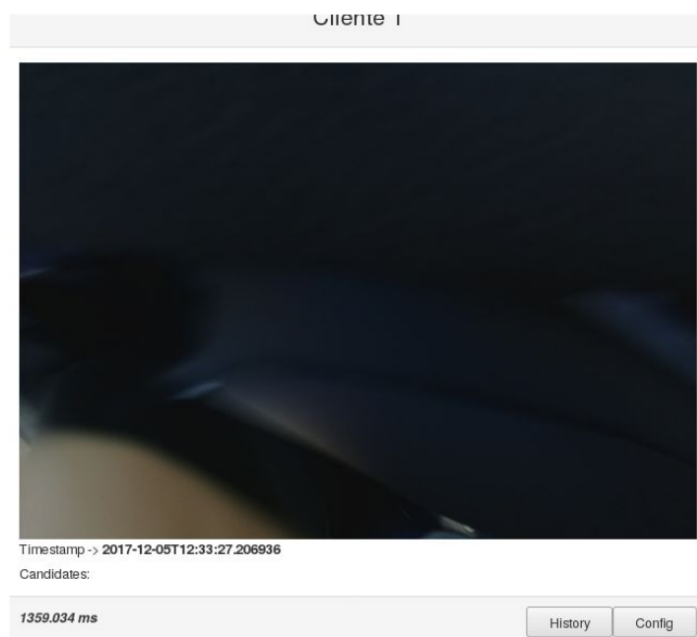
Figure 6.4: Close up view of the representation of a capture device in the dashboard

6.1.3 Experiment Definition

As it was showed in the visualization dashboard, for each capture device, it is possible to access its experiment definition by clicking its "Config" button. When done, the Experiment Definition Interface, is opened in a new modal window. In figure 6.5, the form used for the experiment definition is shown. In this case, the "auto" mode is selected in the leftmost part of the form, and the rightmost part of it shows the two automatic modes that are supported by

our capture device: "sports" and "night". Selecting the "manual" option in the leftmost part would enable selectors in the rightmost part to choose the desired ISO sensibility and shutter speed. After the "Update" button is pressed, all changes defined in this form are applied to the desired capture device once it communicates with the Smart Management Layer, affecting ongoing experiments, if any.
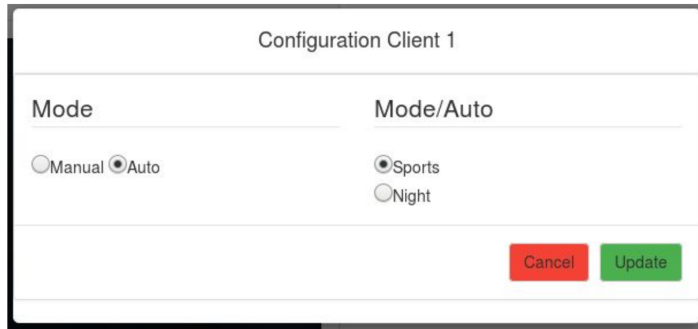
Figure 6.5: Form used in the Experiment Definition Interface

6.1.4 ControlPacket History

Finally, whenever the user clicks the "History" button for a capture device, a new modal will open, containing the last received Controlpackets, with the newest first, in a scrollable format. Figure 6.6 shows this history visualizer, including a ControlPacket for which there has been a plate detected.

Figure 6.6: The modal window opened for the ControlPacket History Visualizer

6.2 Practical Example

It is worth noting that, after the project was developed, it has been used to successfully conduct a real-world experiment with 30 nodes. The experiment was carried in the cam-

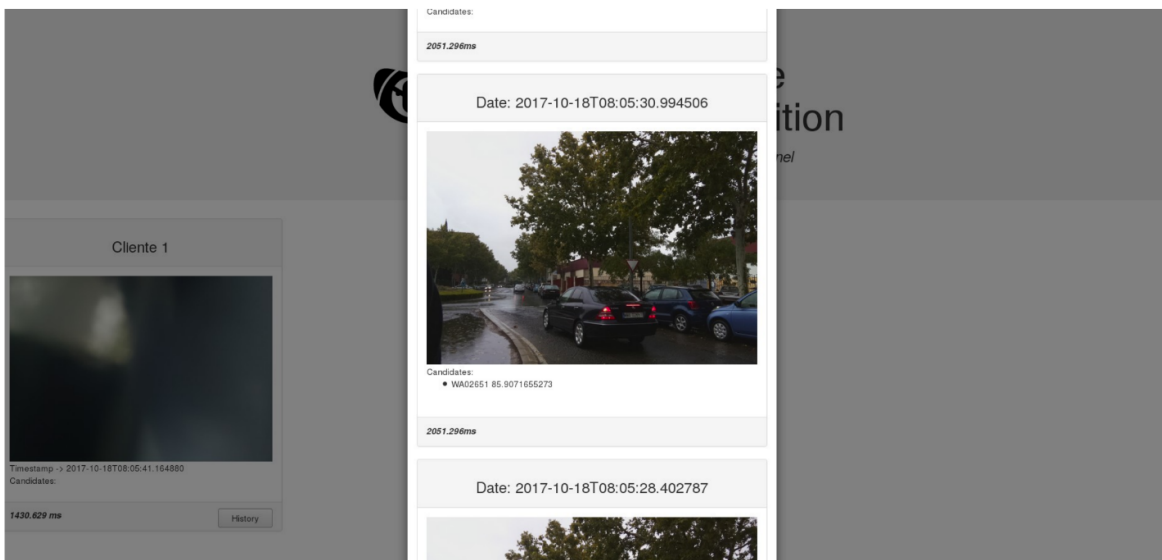pus area of Ciudad Real, and the gathered data was used by researchers of the *Escuela de Caminos, Canales y Puertos* in order to conduct traffic flow analysis and estimation. The results are published in the *Sensors* journal [ÁBSCV+20].

The fact that this experiment was successfully conducted not only proves that the architecture adapts to the scalability and integration constraints it was designed for, but also that the platform solves a real-world need, adding value to a research work and contributing to it. Figures 6.7 and 6.8 show a set of the capture devices that were deployed on the field during the experiment.
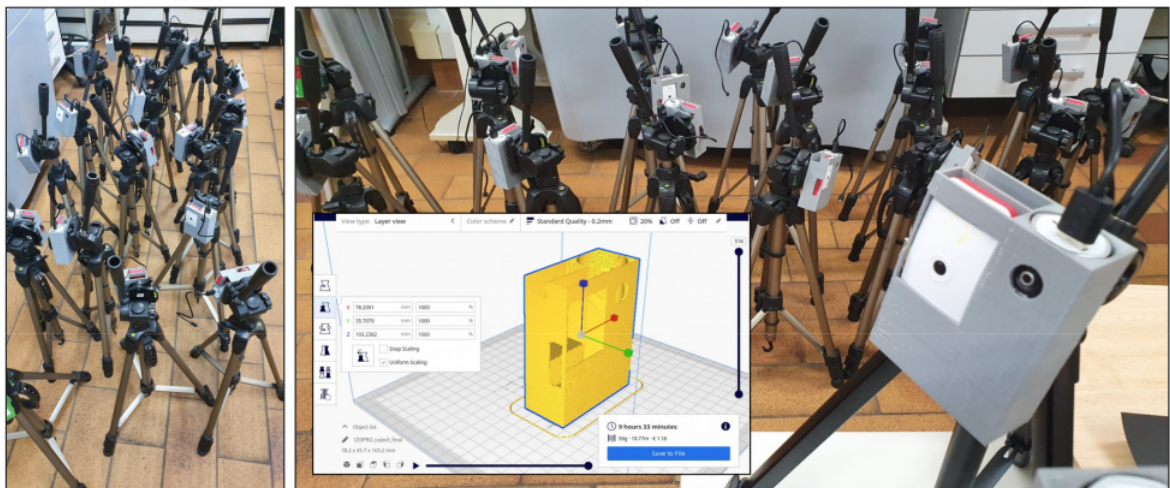
Figure 6.7: Set of capture devices to be deployed, along with the prototype of their enclosure [ÁBSCV+20]

Figure 6.8: Capture devices deployed at several locations [ÁBSCV+20]

6.3 Project Evolution

This section details the different iterations, based on the methodology explained in chapter 4, during which the project has been developed until the end result. At the end of each iteration, a meeting was scheduled with the tutor and, depending on their availability, with the stakeholders from the *Escuela de Ingenieros de Caminos, Canales y Puertos*. Depending

on the evolution of the project, demos were carried during those meetings as well; their goal was to validate the developed functionalities and to decide the steps to undertake during the next iteration. The platform was developed by two people: Carlos Cebrián, and the author of this document.

Iteration 1: State-of-the-art research			
From:	03/07/2017	**To:**	17/07/2017
Research was conducted about the current of cameras for number plate recognition. The hardware platform for the Perceptual Layer (Raspberry Pi Zero W) and how to configure it was studied. Different solutions for ANPR were researched, choosing to use the commercial cloud version of OpenANPR after a comparison of its cost versus the cost of the training and deployment of a custom algorithm			

Iteration 2: Design and initial implementation of the Perceptual Layer			
From:	17/07/2017	**To:**	31/07/2017
Initial development of the different modules of the Perceptual Layer, implementing the whole sequence of operations made by it from its automatic boot and experiment retrieval to the capture and delivery of ControlPackets to a local development server. Some isolated tests of the OpenALPR cloud service were carried with the captured images in order to validate it.			

Iteration 3: Design and initial implementation of the Smart Management and Online Monitoring Layers			
From:	31/07/2017	**To:**	21/08/2017
An initial version of the three modules of the Smart Management Layer was implemented. In the Online Monitoring Layer, an initial version of the Experiment Visualization Dashboard was implemented, aiming to finish and initial end-to-end working version of the platform. In this iteration, all the deployment and testing of the platform has been carried in a local environment.			

Iteration 4: Deployment to App Engine and integration with the Perceptual Layer			
From:	21/08/2017	**To:**	11/09/2017
Once the three layers had been developed, they were integrated, correcting the errors which couldn't be detecting at development time and adding some functional improvements. The Smart Management Layer and Online Monitoring Layer were deployed to Google App Engine, releasing the development team from the burden of maintaining a local development server, and allowing to test the behaviour of the whole platform in its final deployment environment; integration with the Perceptual Layer was addressed.			

Iteration 5: End-to-end and acceptance testing and documentation			
From:	11/09/2017	**To:**	25/09/2017

At this point, a MVP (Minimum Viable Product) of the platform was developed, and for the first time, with all the layers being integrated, it was possible to fully test it. Different tests were carried placing a testing capture device from the Perceptual Layer in different locations and under varied conditions, all of which proved to be satisfactory; on the other hand, multiple functional improvements were identified, adding them to the backlog in order to address them in future iterations.

Iteration 6: User authentication, visualization of history of ControlPackets, and video recording			
From:	25/09/2017	**To:**	08/10/2017

Some of the previously identified functional improvements were implemented. Concretely, in order to improve the security of the platform, the Authentication Module was implemented; the ControlPacket History Visualizer was implemented as well, as the necessity to visualize previous ControlPackets was identified in the last iteration. The necessary functionality to support both modules was implemented in the Smart Management Layer as well. Finnally, in the Perceptual Layer, the Capture Module was improved, allowing it to record video to secondary memory.

Iteration 7: Perceptual Layer bug fixing and UI improvements			
From:	08/10/2017	**To:**	20/10/2017

Bug fixing was performed over the functionality implemented in the last iteration, and the integration and testing of the login functionality was carried. The visualization of the history of ControlPackets was improving by adding a functionality that fetches more ControlPackets as the user scrolls down. Finally, some UI improvements were made, including the inclusion of a default template and background image for the login form, and a header in the main page.

Iteration 8: Code refactoring and documentation			
From:	20/10/2017	**To:**	10/11/2017

Due to the fact that many new features were implemented in the previous iterations, it was decided to dedicate an iteration to a reorganization of the codebase and its documentations. Improvements were mainly carried in the Online Monitoring Layer, as it was the layer where most new functionality was implemented.

Iteration 9: Image transfer optimization			
From:	10/11/2017	**To:**	20/11/2017

After analysis of the performance indicators of the platform, it was found out that image transfer times were a bottleneck in the architecture, and therefore, it was decided to improve them. The solution that was taken was modifying the architecture to abandon the *base64* image encoding that was previously used, saving processor cycles both in the client and in the server, as well as arround 30% of network traffic.

Iteration 10: Implementation of a system to define regions of interest			
From:	20/11/2017	**To:**	04/12/2017

Implementation on a functionality consisting in offering the user the possibility to define a region in the captured image where number plates are most likely to be found. The goal is that only this region is sent to the ANPR service, which would allow to save resources. For the implementation of the functionality, the Experiment Definition Module and Interfaces were modified, as well as the ANPR Processing Module.

Iteration 11: Possibility to define camera parameters			
From:	04/12/2017	**To:**	15/01/2017

Some of the real-world tests that were carried in previous iterations were unsatisfactory due to the captured images being blury, specially in low-light situations. Therefore, tests were carried of captures changing manually the camera parameters relative to exposition time and ISO sensibility. As they were satisfactory, it was decided to implement the possibility to define these parameters in the Experiment Definition, even during the course of a capture, which had implications in the whole system, specially in the Perceptual Layer.

Iteration 12: Delivery of ControlPackets for offline-taken capture			
From:	15/01/2018	**To:**	22/01/2018

The ControlPacket Communication Module of the Perceptual Layer was expanded with a new mode of functioning, consisting on processing the stored video from an previous capture, extracting its frames and generating the ControlPackets that correspond to them, and sending them to the Smart Management Layer; this will allow capture sensors which were offline at the time of the capture to deliver their results once they have network conneectivity.

Iteration 13: Instructions for the operation of the platform and other documentation			
From:	22/01/2018	**To:**	31/01/2018
Once the system was in a stable version, and ready for experiments with multiple capture sensors, a guide detailing how to operate and deploy it was elaborated, along with documentation about the entire system design			

6.4 Development Cost

The platform was developed between July 1st, 2017 and January 31st, 2018, a 7 month period which, after deducting non-working and holiday days, amounts to a total of 120 working days. With 5 hours worked per day, this is equivalent to 600 worked hours; as there were two people involved in the project, this amounts to a total of 1200 hours worked on the project. Assuming a compensation of €30/h for a junior software developer, a total cost of €36,000 is estimated.

The required tools and devices for the development of the platform were: two computers, equivalent to the *Asus X554LA-XX1248H* laptop, which has a retail price of €469 at the *PC-Componentes*[1] e-commerce, and two smartphones used to test the responsive web interface, equivalent to the *Redmi Note 4X*, with a cost of €147.90 each. The usage of Google Cloud Platform incurred in no cost, as it was covered by the *free tier*[2], which includes a 300$ free credit for new accounts. On the other hand, the *starter* plan of the OpenALPR CarCheck API [3] was used, which costs $29 (currently €24.20) per month. In order to test the platform, a 5 month-long subscription was afforded.

The resources used for the development and testing of the Perceptual Layer were; a *Raspberry Pi Zero W* (€10 retail price) and a PiCamera Module V2.1 (sold for €23.63), a 5000 mAh Powerbank used to power the Raspberry (sold for €8.29), a 16GB MicroSD card (sold for €6.39) storing the OS and capture client, and a 128MB memory stick (€1.97).

The total cost of the development of the platform, taking into account the expenses explained above, is €37,405.14. Table 6.1 summarizes the employed resources and their total cost.

6.5 Project Statistics

During the development of the system, *Git* repositories have been used, which allow simultaneous collaborations from the different members of the team, as well as version controlling. Concretely, two code repositories have been used: the first one hosts the code related to the Perceptual Layer, while the second one covers the code related to the Smart Management

---

[1] www.pccomponentes.com
[2] www.cloud.google.com/free
[3] www.openalpr.com/software/carcheck

Resources	Units	Cost/u	Cost
OpenALPR CarCheck API Basic Plan (monthly)	5u	€24.20	€121.00
Asus X554LA-XX1248H Laptop	2u	€469	€938.00
Redmi 4X Smartphone	2u	€147.9	€295.8
Raspberry Pi Zero W	1u	€10.00	€10.00
PiCamera Module V2	1u	€23.69	€23.69
15000 mAh Powerbank	1u	€8.29	€8.29
16GB MicroSD Card	1u	€6.39	€6.39
128MB Memory Stick	1u	€1.97	€1.97
Developer Salary	1200u	€30.00	€36,000.00
		TOTAL	€37,405.14

Table 6.1: Total cost calculation (including taxes)

and Online Monitoring Layers. Using tools such as *Gitstats*[4], it is possible to extract some useful statistics from the repository for its study.

The repository hosting the Perceptual Layer code sums, at the time of this writing, 754 lines of code (LOCs) in 66 commits, of which 83.42% are written in *Python*, 12.20% in *Bash*, and the remaining 4.38% correspond to configuration files and deployment guides. On the other hand, the repository hosting the Smart Management and Online Monitoring Layer sums 1845 lines in 349 commits, of which 42.93% are written in *Python*, 42.22% in *HTML*, *CSS* and *JavaScript*, and the remaining 14.85% in other languages. Below, the count of lines of code of both repositories is plotted against time.
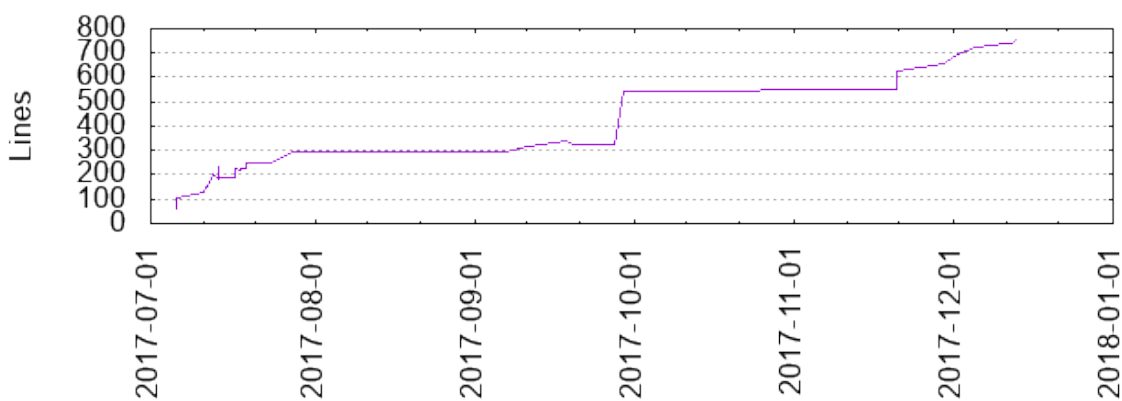

Figure 6.9: Evolution of Lines of Code (Perceptual Layer repository)
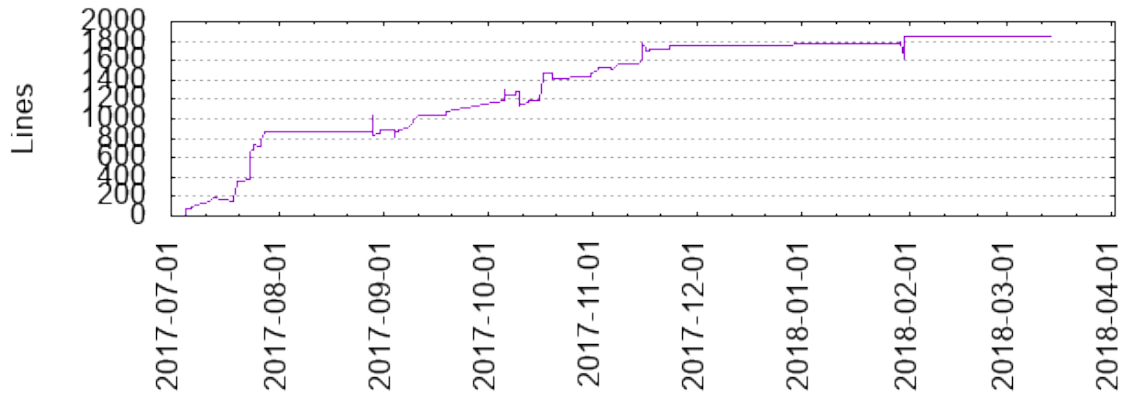
---

[4]www.github.com/dmitryn/GitStats

Figure 6.10: Evolution of Lines of Code (Smart Management and Real-Time Monitoring Layers repository)

Chapter 7

Conclusions

In this chapter, the conclusions obtained after the design, implementation, testing, and use of the platform are detailed. First, the degree of fulfillment of each of the objectives defined in chapter 2 is assessed, in order to determine whether or not the main objective has been reached. Second, possible future work lines are detailed, based on both the experience during the development of the platform, feedback from the users, and the state-of-the-art. Finally, the personal opinion of the author will be detailed, concluding this document.

7.1 Achieved Objectives

Starting from the objectives that were defined, this section details the solutions that have been developed to address each one of them.

1. *Support for the definition of experiments.* The user can define the experiment settings related to the camera parameters and areas of interest through the Experiment Definition Interface, which gets stored thanks to the Experiment Definition Module. The rest of the experiment settings can be directly set through the latter; in the Perceptual Layer, the Experiment Retrieval Module will fetch the required settings, providing them to the capture device. Additionally, the defined parameters will be applied to the capture device as soon as they are received, even during ongoing experiments.

2. *Retrieval of data from the capture devices.* The capture images are sent in the format of ControlPackets, which encapsulates all the required data, via the ControlPacket Delivery Module, and received via the ANPR Intelligent Processing Module. Each capture device is identified with its Client ID. Reliability against network issues is addressed by storing the whole capture in secondary storage, in video format, and offering the possibility to send it back to the Smart Management Layer after the experiment, whenever network connectivity is available.

3. *Support for the ALPR process.* The ANPR Intelligent Processing Module supports the request of license number plates to the chosen OpenALPR Cloud Service; however, decoupling from this service is ensured, as the requests can be sent to any other API.

4. *Portability between devices.* One of the implementation details of the Online Monitoring Layer is the fact that it generates responsive webpages thanks to the *Bootstrap*

framework; therefore, compatibility with any modern web browser, no matter the device it runs on, or its screen size, is ensured. Besides, some design choices, such as the use of the WebSocket protocol, were discarded in favor of improved portability between devices.

5. *Real-time monitoring and configuration changes.* The Online Monitoring Layer fetches new ControlPackets periodically (by default, every 5 seconds), ensuring that the users have an updated view about the state of every capture device at any given moment. Load and real-world testing with multiple concurrent users have shown that the platform scales well to this situation, allowing each user to have current information about the experiment. On the other hand, changes performed to any Experiment Definition are communicated to the capture devices as soon as they communicate with the Smart Management Layer, and the latter handles these changes as soon as they are received, updating the camera settings and other parameters accordingly.

6. *Detection of errors at the capture devices.* The Experiment Visualization Dashboard provides continuously updated information about each control device, including the latency of arriving packets, the detected number plates, and the captured image, and allows any user that is monitoring it to perform corrective configuration changes immediately.

7. *History of captures.* The ControlPacket History Visualizer provides the functionality which allows the user to see the history of captures for any given capture device, loading more items as he or she scrolls down, providing a full perspective of the evolution of the experiment.

8. *Ease of use.* Both the User Interface of the Online Monitoring Layer, the deployment process of the Smart Monitoring Layer, and the configuration process of capture devices in the Perceptual Layer, were designed with usability in mind, aiming to ease the use of the platform as much as possible, and providing written guides when necessary.

9. *Deployment of the system to the cloud.* The Smart Management and Online Monitoring layers were deployed to the App Engine service of Google Cloud, being possible to deploy more instances of the platform should the need arise.

7.2 Future Work

This section details lines of future work, identified as both improvements to the current functionality of the system, and additions to it.

1. *Release of a commercial version.* Once the system is stabilized, and, as is the case, proves to behave well in real-world situations, it is possible to offer it as a service for clients. Examples of possible clients that may find value in this platform range from

research groups to public institutions, governments, international organizations, and private companies.

2. *Security improvements.* While the confidentiality of the network traffic between all layers is ensured with the HTTPS protocol, and users need to log in in order to use the Online Monitoring Layer, more security mechanisms can be implemented and enforced. For example, it is possible to provide capture devices with a secret token that identifies them to the Smart Management Layer, ensuring that only authenticated devices can use its API. On the other hand, the use of encryption can be enforced in the secondary storage of capture devices, improving the confidentiality of recorded video.

3. *Development of a custom ALPR model.* During the development of this platform, the OpenALPR commercial service proved to be a good choice in terms of cost-benefit. However, as the system scales to more capture devices, the cost of this service can become prohibitively high. Similarly, as more labeled images are available from the result of previous experiments, the cost of training a custom model decreases, making it become a more appealing option.

4. *Increased portability between cloud environments.* During the development of this system, Google Cloud Platform and its App Engine service proved to give us the flexibility and ease of use that was needed. While it is possible to keep the platform deployed in this service with a low operation cost, for some use cases, it may be necessary to deploy it to other cloud service or on-premises (for example, for compliance reasons). A migration of the system to a vendor-agnostic platform such as Kubernetes may avoid vendor lock-in and give support for these use cases.

5. *Support for more kinds of capture devices.* It is possible to adapt the platform to other kinds of capture devices, which enables exploration of even lower-cost hardware platforms. As the Capture Module is clearly isolated from the rest of the components of the Perceptual Layer, it is possible to modify it to adapt to other capture libraries, therefore adapting the whole Layer to any device that runs a Linux flavor.

6. *Anomaly detection of capture devices.* In order to ease the monitoring of experiments where a big number of capture devices is present, it is possible to implement an anomaly detection algorithm with the objective of alerting the users of devices that may be malfunctioning. Variables such as the latency of ControlPackets or the number of detected ControlPackets can be used, as well as the captured images.

7. *Geo-location of capture devices.* Data about the geographical location of each capture device can be optionally added to the different Experiment Definitions in order to both ease the monitoring of experiments and have more data available for future analysis. A map widget could be added to the Experiment Visualization Dashboard in order to facilitate the geo-location.

7.3 Fulfilled Competences

Below, in table 7.1, the fulfilled competencies relative to the *Computation* intensification are detailed, emphasizing their relationship with the work performed in this final dissertation.

Table 7.1: Justification of the specific competences addressed in the final dissertation

Competence	Justification
Capacidad para conocer los fundamentos, paradigmas y técnicas propias de los sistemas inteligentes y analizar, diseñar y construir sistemas, servicios y aplicaciones informáticas que utilicen dichas técnicas en cualquier ámbito de aplicación.	In order to obtain the best possible results from the plate recognition phase, the methods and techniques behind ALPR were analysed, choosing the system that best suits this specific case's needs after a review of its used techniques and algorithms.
Capacidad para conocer y desarrollar técnicas de aprendizaje computacional y diseñar e implementar aplicaciones y sistemas que las utilicen, incluyendo las dedicadas a extracción automática de información y conocimiento a partir de grandes volúmenes de datos.	Machine Learning algorithms are used in ALPR solutions (particularly in number plate detection and character recognition), which the platform will use extensively. On the other hand, future lines of work (such as anomaly detection in capture devices) are based on Machine Learning techniques.
Capacidad para desarrollar y evaluar sistemas interactivos y de presentación de información compleja y su aplicación a la resolución de problemas de diseño de interacción persona computadora.	In the design of the web interface, the way in which the information is presented to the user was taken into account, focusing the design on a simple and intuitive interface which allows the user to easily take control of the experiment in an interactive way.

7.4 Personal Opinion

This work puts an end to my undergraduate studies. During these years, I had the opportunity to learn, from different perspectives, about my passion, Computer Science, which is one of the main forces transforming the world as we know it. I am glad to say that my university studies have greatly helped me through my path to become a good Computer Engineer, giving me the necessary tools to responsibly contribute to this changing world.

I could verify that most, if not all of the courses I took during my studies, helped me during the development of this project. I could also learn about the implications that a development of this magnitude faces, with multiple systems in different environments interacting between them, and the design choices that need to be made. I had the opportunity to learn, in a small and autonomous team, about modern web application development, the public cloud, and the applications of low-cost capture devices such as the Raspberry Pi. While the range of technologies to chose from was very big, I think we made the right decision choosing App

Engine, Flask, and Bootstrap. App Engine offers developers a free environment to develop and test their applications on, which can scale as needed, while Flask offers a minimalist approach to web server development, and Bootstrap fulfills the requirement for responsive pages we had. Overall, I could see how different technologies could be combined to create a new solution. On the other hand, I could verify the utility of agile methodologies in the context of developments with changing requirements and uncertainty.

This project helped me also to validate the potential that Computer Engineering has to complement other areas. Like any other development project, a greater degree of refinement could have been achieved, as there is always room for improvement; however, I feel satisfied that the solution we implemented helped improve an otherwise tedious and manual process, enabling it to be more cost-effective and scalable and, therefore, contribute to the development of the urban traffic analysis research field.

APPENDICES

Appendix A

Instructions for the setup of the platform

Below, the documentation file provided to the users of the platform that details how to operate it is transcribed. It contains instructions on how to install the capture client on the Raspberry Pi devices and set up the configuration on the Smart Management Layer. Finally, instructions on how to access the web interface are provided.

A.1 Instructions

The goal of these instructions is to specify how to deploy the ANPR platform.

A.1.1 Requirements for the deployment

- Access to the Google App Engine project console
- n Raspberry Pis fitted with Camera Modules
- n SD cards
- n pendrives in order to store the configuration and video captures

A.1.2 Setup of the clients

First, we must configure the Raspberries (clients). The recommended procedure is mounting the original SD card, performing the changes, and cloning the image to the n-1 remaining cameras.

The first step is Wi-Fi setup. For this, we edit the */etc/wpa_supplicant.conf* file and we make sure that the eduroam username and password are updated and valid. In this file, we store the SSID and authentication parameters of the hotspot that the Raspberry connects to when no eduroam connectivity is located; we must make sure as well that this information is valid and corresponds to the desired hotspot.

Once this change is performed, and after making sure that the image is functional, the memory card can be cloned.

```
$ dd bs=4M if=/path/to/image.img of=/dev/sdX
$ dd bs=4M if=/dev/sdX of=/path/to/image.img
# ...
$ e2fsck -f /dev/mmcblk0p2
```

83

```
$ resize2fs /dev/mmcblk0p2
```

Listing A.1: Camara image cloning process

The second part of the configuration is located inside the pendrive of each Raspberry. Its functionality is identifying each one of them. Inside each pendrive, there must be two files: *ADDRESS_SERVER*, containing the backend web address, and *CLIENT_ID*, containing the numeric, unique ID that is assigned to the device. Therefore, we will make sure that each pendrive contains those two files, and that each Raspberry has a different *CLIENT_ID* assigned.

A.1.3 Server setup

Inside the application database, which we can access from the Google App Engine console, there must be a *ClientConfig* entity per each Raspberry that is deployed.

Therefore, we will add each entity that is necessary, clicking on *Create Entity*. For each entity, the *Key identificator* will be a custom name that will match the *CLIENT_ID* of the corresponding Raspberry.

We will edit, as well, the rest of the compulsory fields: *begTime* and *endTime*, with the starting and ending date of the experiment (in UTC), *freqCapture* with the number of FPS that we want to record, *freqControlPack* with the seconds between deliveries of ControlPackets, and *mode*, which is advised to set to "sports".

A.1.4 Usage of the platform

Once the configuration is finished, we can start using the platform. The Raspberries, once launched, will connect to the specified WiFi network (Eduroam or, by default, the specified network), get their corresponding configuration, and start the capture once the experiment beginning time is reached, if everything works as expected.

It will be possible to follow the development of the experiment in the user panel [1], after logging in with the required credentials.

---

[1]http://www.platerecognitionplatform.appspot.com

Appendix B

Testing the platform in real-world environments

During the development of the platform, several tests of it were performed. This was partially done in order to validate the integration of the layers of the platform and the reliability of the system; however, there was also a need to test the chosen OpenALPR service with the captures that the Raspberry PI devices took. This appendix exposes two different tests: the first one validates the operation of the platform under normal environmental circumstances, while, for the second one, issues were detected, and new functionality was implemented to address them.

Overall, the system performed correctly when used under clean meteorological conditions with daytime light, and with the capture device located in a strategic point. This was checked even when the licence plates had a significant skew. In the showed images, the end of each number plate is deleted in order to protect personal information.
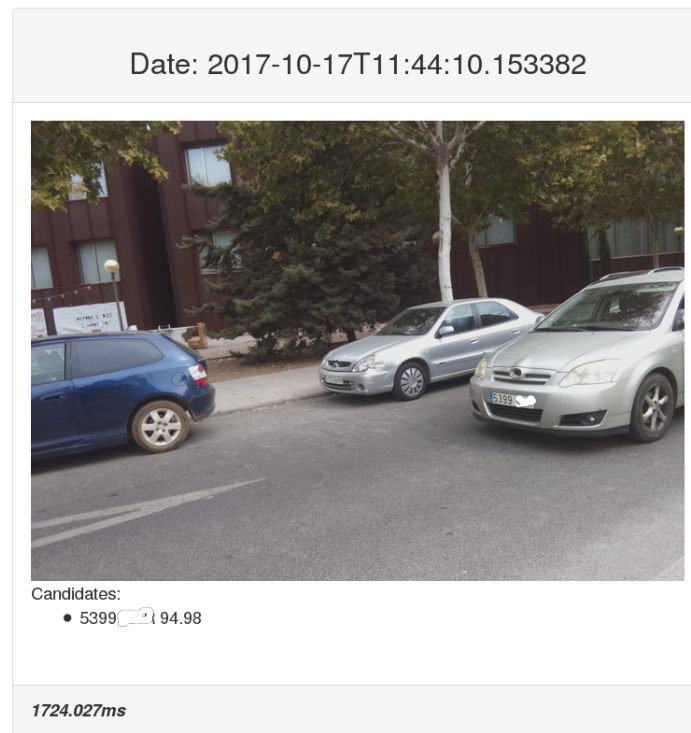

Figure B.1: Result of a plate detection in ideal conditions.

However, when light conditions worsen and motion blur appears, license plates are sometimes unreadable.

Figure B.2: Blurred car image in low-light conditions

The possibility to set camera parameters such as ISO sensibility and shutter speed was implemented after observing this behavior in the tests of the system. The possibility to define an area where license plates of interest are expected to appear was implemented as well, in order to prevent situations such as the one seen in the figure above, where license plates of parked vehicles are detected.

Bibliography

[AALK06]   Christos Nikolaos E Anagnostopoulos, Ioannis E Anagnostopoulos, Vassilis Loumos, and Eleftherios Kayafas. A license plate-recognition algorithm for intelligent transportation system applications. *IEEE Transactions on Intelligent transportation systems*, 7(3):377–392, 2006.

[AAP⁺08]   Christos-Nikolaos E Anagnostopoulos, Ioannis E Anagnostopoulos, Ioannis D Psoroulas, Vassili Loumos, and Eleftherios Kayafas. License plate recognition from still images and video sequences: A survey. *IEEE Transactions on intelligent transportation systems*, 9(3):377–391, 2008.

[ÁBSCV⁺20]   Fernando Álvarez-Bazo, Santos Sánchez-Cambronero, David Vallejo, Carlos Glez-Morcillo, Ana Rivas, and Inmaculada Gallego. A low-cost automatic vehicle identification sensor for traffic networks analysis. *Sensors*, 20(19):5589, 2020.

[AHP06]   Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns: Application to face recognition. *IEEE transactions on pattern analysis and machine intelligence*, 28(12):2037–2041, 2006.

[And10]   David J Anderson. *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[AQIS07]   Ayman AbuBaker, Rami Qahwaji, Stan Ipson, and Mohmmad Saleh. One scan connected component labeling technique. In *2007 IEEE International Conference on Signal Processing and Communications*, pages 1283–1286. IEEE, 2007.

[AS94]   Richard Arnott and Kenneth Small. The economics of traffic congestion. *American scientist*, 82(5):446–455, 1994.

[Boe07]   Barry Boehm. A survey of agile development methodologies. *Laurie Williams*, 45:119, 2007.

[BVO11]   Norbert Buch, Sergio A Velastin, and James Orwell. A review of computer vision techniques for the analysis of urban traffic. *IEEE Transactions on Intelligent Transportation Systems*, 12(3):920–939, 2011.

[Can86] John Canny. A computational approach to edge detection. *IEEE Transactions on pattern analysis and machine intelligence*, (6):679–698, 1986.

[CH98] Yuntao Cui and Qian Huang. Extracting characters of license plates from video sequences. *Machine Vision and Applications*, 10(5-6):308–320, 1998.

[CLCW09] Zhen-Xue Chen, Cheng-Yun Liu, Fa-Liang Chang, and Guo-You Wang. Automatic license-plate location and recognition based on feature salience. *IEEE transactions on vehicular technology*, 58(7):3781–3785, 2009.

[CMJ08] Enrique Castillo, José María Menéndez, and Pilar Jiménez. Trip matrix and path flow reconstruction and estimation based on plate scanning and link observations. *Transportation Research Part B: Methodological*, 42(5):455–481, 2008.

[DISB12] Shan Du, Mahmoud Ibrahim, Mohamed Shehata, and Wael Badawy. Automatic license plate recognition (alpr): A state-of-the-art review. *IEEE Transactions on circuits and systems for video technology*, 23(2):311–325, 2012.

[E⁺08] Andrew Eberline et al. Cost/benefit analysis of electronic license plates. Technical report, 2008.

[FM11] Ian Fette and Alexey Melnikov. The websocket protocol, 2011.

[Gar11] Brett S Gardner. Responsive web design: Enriching the user experience. *Sigma Journal: Inside the Digital Ecosystem*, 11(1):13–19, 2011.

[GMMP02] Surendra Gupte, Osama Masoud, Robert FK Martin, and Nikolaos P Papanikolopoulos. Detection and classification of vehicles. *IEEE Transactions on intelligent transportation systems*, 3(1):37–47, 2002.

[Han20] John W Hanson. Using city gates as a means of estimating ancient traffic flows. *PLoS one*, 15(2):e0229580, 2020.

[HK76] Joseph Hoshen and Raoul Kopelman. Percolation and cluster distribution. i. cluster multiple labeling technique and critical concentration algorithm. *Physical Review B*, 14(8):3438, 1976.

[Hou62] Paul VC Hough. Method and means for recognizing complex patterns, December 18 1962. US Patent 3,069,654.

[Hou93] Gary Houston. Iso 8601: 1988 date/time representations, 1993.

[JW11] Kevin L Jackson and Robert Williams. The economic benefit of cloud computing. *NJVC Executive*, 2011.

[KC11]   H Erdinc Kocer and K Kursat Cevik. Artificial neural networks based vehicle license plate recognition. *Procedia Computer Science*, 3:1033–1037, 2011.

[KDDT13]   Barbara Karleuša, Nevena Dragičević, and Aleksandra Deluka-Tibljaš. Review of multicriteria-analysis methods application in decision making about transport infrastructure. 2013.

[LLH16]   J. Lin, L. C. Lin, and S. Huang. Migrating web applications to clouds with microservice architectures. In *2016 International Conference on Applied System Innovation (ICASI)*, pages 1–4, 2016.

[Mar07]   Ondrej Martinsky. Algorithmic and mathematical principles of automatic number plate recognition systems. *Brno University of technology*, pages 20–23, 2007.

[MBRP10]   Zuwena Musoromy, Faycal Bensaali, Soodamani Ramalingam, and Georgios Pissanidis. Comparison of real-time dsp-based edge detection techniques for license plate detection. In *2010 Sixth International Conference on Information Assurance and Security*, pages 323–328. IEEE, 2010.

[ME12]   Giridhar D Mandyam and Navid Ehsan. Html5 connectivity methods and mobile power consumption. *Accessed January*, 16:2013, 2012.

[MMSU15]   Gurpreet Singh Matharu, Anju Mishra, Harmeet Singh, and Priyanka Upadhyay. Empirical study of agile software development methodologies: A comparative analysis. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–6, 2015.

[NYK⁺05]   Shigueo Nomura, Keiji Yamanaka, Osamu Katai, Hiroshi Kawakami, and Takayuki Shiose. A novel adaptive morphological approach for degraded character image segmentation. *Pattern Recognition*, 38(11):1961–1975, 2005.

[ÖÖ12]   Fikriye Öztürk and Figen Özen. A new license plate recognition system based on probabilistic neural networks. *Procedia Technology*, 1:124–128, 2012.

[Pre95]   Wolfgang Pree. Design patterns for object-oriented software development. 1995.

[PSK⁺10]   G Padmavathi, D Shanmugapriya, M Kalaivani, et al. A study on vehicle detection and tracking using wireless sensor networks. *Wireless Sensor Network*, 2(02):173, 2010.

[PSP13]   Chirag Patel, Dipti Shah, and Atul Patel. Automatic number plate recognition system (anpr): A survey. *International Journal of Computer Applications*, 69(9), 2013.

[SCJRG17]   Santos Sánchez-Cambronero, Pilar Jiménez, Ana Rivas, and Inmaculada Gallego. Plate scanning tools to obtain travel times in traffic networks. *Journal of Intelligent Transportation Systems*, 21(5):390–408, 2017.

[SLE19]   David Schrank, Tim Lomax, and Bill Eisele. 2019 urban mobility report. *Texas Transportation Institute,[ONLINE]. Available: http://mobility. tamu. edu/ums/report*, 2019.

[Smi07]   Ray Smith. An overview of the tesseract ocr engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, volume 2, pages 629–633. IEEE, 2007.

[SMX04]   Zhang Sanyuan, Zhang Mingli, and Ye Xiuzi. Car plate character extraction under complicated environment. In *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No. 04CH37583)*, volume 5, pages 4722–4726. IEEE, 2004.

[SP00]   Jaakko Sauvola and Matti Pietikäinen. Adaptive document image binarization. *Pattern recognition*, 33(2):225–236, 2000.

[SS11]   Ken Schwaber and Jeff Sutherland. The scrum guide. *Scrum Alliance*, 21:19, 2011.

[TO13]   Dob Todorov and Yinal Ozkan. Aws security best practices. *Amazon Web Services [Online]. Available from: http://media.amazonwebservices. com/AWS_Security_Best_Practices.pdf*, 2013.

[Vok04]   Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904–917, 2004.

[vT11]   Cornelis van Tilburg. Traffic policy and circulation in roman cities. *Acta Classica*, 54:149–171, 2011.

[WJ04]   Christian Wolf and J-M Jolion. Extraction and recognition of artificial text in multimedia documents. *Formal Pattern Analysis & Applications*, 6(4):309–326, 2004.