Università degli Studi della Calabria

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica

---

Magistralis Laurea Course in Computer Engineering for the Internet of Things

Magistralis Laurea Thesis

Creating Digital Twins with General Purpose Sensing: Architecture and Initial Experiments

Supervisor

Prof. Giancarlo Fortino
Dott. Roberto Minerva
Dott. Claudio Savaglio

Student

Vincenzo Barbuto
Matr. 224489

Academic Year 2021-2022

Abstract

Being in the middle of an industrial revolution caused by interconnected cyber physical systems, the emerging concept of Digital Twin (DT) represents a driver for "Industry 4.0". Taking advantage of the disrupting digital technologies such as Artificial Intelligence (AI), Machine Learning (ML), Big Data Analysis, and Internet of Things (IoT), the DT is a key concept for developing new applications and solutions in several domains. Indeed, by collecting simple data from the environment (*general-purpose sensing*) and determining complex aggregations of events using AI techniques and algorithms (*synthetic sensing*), it is possible to build and work with a digital version of physical objects. Although DT became very popular in the so called Industrial IoT (IIoT) domain, Smart Cities systems started to take advantage of this technology. As a result, nowadays the definition of Digital Twin is a prerequisite of many systems.

In this thesis the design, the implementation and the analysis of a prototype digital-twin-based Traffic Monitoring System (TMS) are presented, in which edge computing, transfer learning, and tiny-AI merge with the new DT technology. Overall, the aim of this thesis is to identify, classify physical objects in the context of a traffic monitoring system and use this information to automatically build DTs representing both real roads and vehicles. By virtualizing a large environment, its status data can be accessed as the normal information available on the web, and therefore it could be used to improve both the traffic congestion and viability as well as the monitoring of pollutants emitted by vehicles. Hence, this work presents the challenges faced and the full description of both hardware and software components that constitutes the whole system structure.

The deployment of ML models at the Edge of the network, allows eventually to *(i)* improve inference performances, by exploiting the Google Edge Tensor Processing Unit (TPU); *(ii)* reduce bandwidth and power consumption, by reducing the amount of data to be transmitted; *(iii)* reduce latencies, by cutting back round-trips to a server during models' execution.

Contents

Chapter 1

Introduction

The evolution of Internet of Things (IoT) technologies makes it possible to transform "physical spaces" into smart environments. Smartness means the ability to "measure" important activities, phenomena, and actions occurring in the environment and to provide insights on how to optimize the territory's resources for the best interest of the community.

Nowadays, the dynamics associated to urban activities - e.g., the concentration of economic activities and the aggregation of population in metropolitan areas - and the worrying climate crisis, is pushing toward a digitization process to create innovative and sustainable solutions able to improve territory activities such as *(i)* the waste management; *(ii)* the energy and water provisioning; *(iii)* the traffic congestion; *(iv)* the transportation management; *(v)* the power generation; *(vi)* its governance; and *(vii)* citizens lifestyle. This trend is emerging for large and densely populated cities as well as for suburban or rural areas. Even if these areas may have different and sometimes conflicting requirements, they all need sensing and actuation capabilities to measure the "usage" of the area and to orchestrate the available resources, in order to optimize their consumption in favor of the quality of life of people and enterprises of the territory.

Such a trend is recognizable since the beginning of the 21st century, in fact cities and territories have become increasingly smarter [1]. The great advancement in exploiting pervasive information technologies - such as IoT - along with the development of increasingly smart devices, gave cities the possibility to gather a huge amount of data from a wide variety of activities spread in the whole metropolitan area [2–6]. Data represents the means through which inanimate objects can have knowledge about the environment that surrounds them so as to monitor activities and evolve with the environment itself.

In large environments such as smart cities, a single-purpose sensing approach, i.e. the deployment of a single sensor network for monitoring and controlling a single aspect of interest (e.g., pollution, noise, water level), is usually adopted and implemented. Although this technique tends to be robust, especially when only a few aspects of the environment are monitored, it loses effectiveness when the features to monitor increase, thus requiring the deployment of a larger set of sensing infrastructure. Moreover, as a further issue, in large environments different networking capabilities are needed in order

to function and collect data, which is also typically siloed, i.e. information is collected and maintained in isolation. Due to all these factors, the integration and the creation of relationships between different types of data taken in different instants and location (e.g., traffic and pollution intensity) are difficult to correlate as well as the deployment and maintenance of such a huge number of sensors and networks, which comes at a great cost.

Ideally, a single network with flexible and adaptable sensors (i.e., sensor capable of measuring basic phenomena and, by using Artificial Intelligence or mathematical models, deriving precise and specific measures) would be a major step towards the large-scale "softwarization" of large environments. The basic idea is to collect general measures from the terrain and by means of software techniques to derive specific measures and values. This approach means to move the sensing from specific hardware to specialized software operating in conjunction with "general-purpose" sensing [7]. Using just a single sensor board to detect many aspects (one-to-many, see Figure 1.1), may prove to be more flexible, scalable, and suited for large contexts with thousands of facets to be measured. The so-called general-purpose sensing, aims at reducing costs and enhances sensing and computing capabilities by making objects smarter than ever.

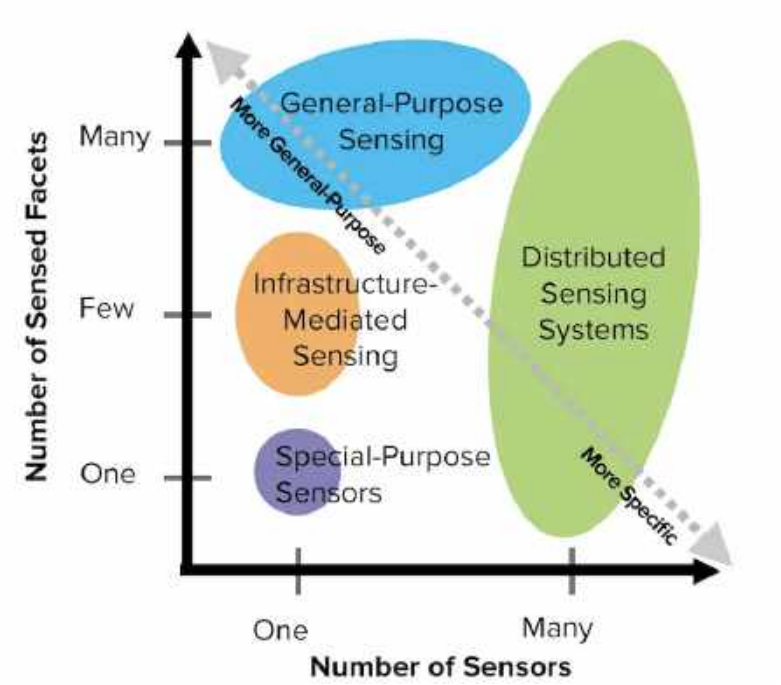


Figure 1.1: *Canonical approaches in environmental sensing, source [7]).*

In this context (*general-purpose sensing* applied to large environments) the concept of Digital Twin (DT) found a fertile ground. A DT represents the virtual (digital) companion of a physical object (PO) [8], i.e., a software counterpart of a thing built by means of data collections which describe status and properties of a physical object

using sensors in the field. DTs offer a way to move from the physical space to the virtualized one, in which we are able either to monitor the POs for their entire life cycle, by keeping permanent connections between them and their logic objects (LOs), or to simulate their behavior under several conditions, by replicating LOs concurrently in several environment. Moreover, DTs can help optimize physical asset performance through efficient predictive and preventive maintenance operations, reducing overall operational costs and downtime, by offering access to real-time data as well as historical information. Therefore, DTs represent the point of connection between the physical and digital world [9].

1.1 Objective

This thesis is written as a result of the Double Degree program attended and of the collaboration between professor Noel Crespi's laboratory (DICE Lab, Telecom SudParis, France) and professor Giancarlo Fortino's laboratory (SPEME Lab, Unical, Italy) and aims to explore the usage of *general-purpose sensing* in association with the creation and exploitation of DTs, by laying the groundwork of a digital-twin-based Traffic Monitoring System (TMS). *General-purpose sensing* is, in principle, a technique that could improve this digitization process, offering a smart way of gathering huge quantities of data with minimal effort. These data can be intelligently analyzed and understood in order to represent actors operating in the environment and to represent them as Digital Twin. This approach to the construction of Digital Twins is complementary to the mainstream one, in which instead DTs are manually built starting from data gathered from the environment. In this thesis, the objective is to automatically build DTs by collecting data from the environment and incrementally refine and improve the logical representation of the physical objects.

In order to assess the approach (it could be summarized as the combination of "general purpose sensing", "synthetic sensing" and "reverse Digital Twin creation") a hardware and software system is needed. The proposed system therefore consists of the following building blocks:

- an edge engine, i.e., a single device comprising several hardware modules, whose main task is to collect data for digitizing the specific road (local, highway, city street, pedestrian street, etc.) on which it is placed on;
- communication system, it is used for forwarding the information collected to all the involved software components;
- a controller and forwarder (NodeRED), it is used in order to dispatch information to different components or activate components;
- databases for storing the data related to the different DTs "captured";
- a DT framework (based on Eclipse Ditto) for organizing and querying the different DTs.

The edge engine is the major focus of the thesis. It captures and provides relevant information useful both for checking the road viability remotely - such as traffic congestion, road condition, etc. - and for monitoring environmental parameters (temperature, humidity, pollutant etc.). Moreover, it is able to detect both the number and the specific type of vehicles which, in a period of time, pass through that street. All the data acquired by devices on the field are both stored and visualized on a specific dashboard and used by other the software parts of the system to create and maintain one DT that represents the specific road in which the hardware device is placed (R-DT); and one DT for each specific type of vehicle detected (V-DTs).

The final objective is to acquire enough data, exploiting the new *general purpose sensing approach*, to create a signature of each vehicle, i.e. something that can identify the class of a given product (specific type, brand and model of a vehicle). This signature can be achieved by capturing images, engines' sounds and pollution data of a given vehicle, and can be stored inside the DT. The latter could become a kind of signatures' repository that deep learning algorithms could use to identify specific vehicles and, based on that, perform related actions such as monitor single vehicles' environmental impact, vehicles' failures recognition, emergency detection, etc.

The rest of the Thesis is organized as follows. Chapter 2 presents the theoretical concept that leads the research (general purpose sensing and DTs) and how they can be merged to create innovative solutions in large environments such as Smart Cities. Then, Chapter 3 introduces the hardware components and the main software tools used for building the proposed system. Chapter 4 offers an in-depth description of the digital-twin based TMS: *(i)* goal and purposes; *(ii)* hardware and software modules placement; *(iii)* implementation details. Chapter 5, instead, presents a full analysis about benefits achieved exploiting Edge AI in terms of inference latency, accuracy and bandwidth consumption. Moreover, a comparison between performances on different hardware configurations (Edge TPU, Embedded CPU and Desktop CPU) is shown. Finally, Chapter 6 includes the concluding statements concerning the findings in this research to fulfill the above mentioned thesis's objectives, as well as the future works needed to improve the TMS.

Chapter 2

State of The Art

2.1 General Purpose Sensing and AI

In order to exploit a full-flexible utilization of sensing devices, an emergent trend is to populate hardware boards with several elementary sensors. This trend originated from the capabilities exposed by smartphones where a set of well defined sensors can generate a wealth of data (by means of edge AI software) exploitable by many different applications [10]. There seems to be the possibility to apply similar mechanisms to other implementations of Internet of Things. For instance, by populating hardware devices with several elementary sensors, the resulting boards are said to be "general purpose" since they are able to collect a wide range of features. Noise, temperature, humidity, illuminance and many other aspects can be measured and collected by just one single board. Under a simple perspective, these features can approximate/resemble the human sensing capabilities. From an analysis of these features some characterizing data (e.g. movements, opening of doors, etc.) can be derived by means of the application of AI algorithms [7]. Instead of using one sensor for one specific measure (single purpose approach) there is the possibility of capturing new measurements by means of algorithmic solutions. In principle, these boards can be placed everywhere in the environment and still sense many aspects from it.

Hence, the approach that follows from this new way of thinking is that, ideally, it is possible to digitize a given environment by collecting many of its facets using just one board equipped with simple sensors. This method is proposed in literature as *general-purpose sensing* [7]. The benefits of using it are many: *(i)* reduces deployment and maintenance costs; *(ii)* reduces the hardware components cost; *(iii)* reduces the aesthetic and social costs [11]; *(iv)* it moves the complexity from hardware to software.

The huge amount of data gathered from the environment can then be used by modern Artificial Intelligence (AI) techniques to learn patterns about the external world. In fact, the fuel of Machine Learning (ML) algorithms is data, which allows to train increasingly powerful models. Therefore, the current trend is not only to provide a new sensing method to "listen" to the environment, but also to power up this feature by moving complexity to the software layer [12]. General purpose sensing along with AI could

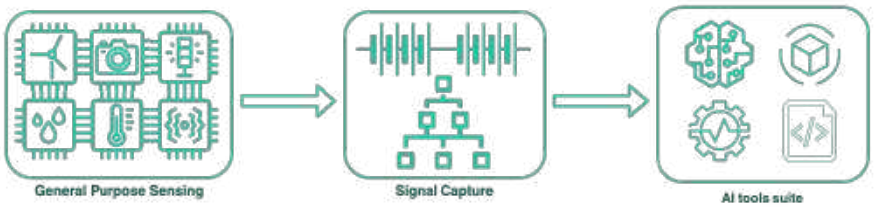


Figure 2.1: *General Purpose Sensing and AI*

bring to a new generation of systems in which they are able both *(i)* to understand stimulus coming from the external world and *(ii)* and act based on them (Figure 2.1).

Although this new approach seems to work fine in small contexts (such as smart home [7]), it needs to be tested in large and complex environments such as industrial plants or smart cities. Moreover, in practice the general purpose sensing technique didn't reach a definitive way to work, since it still has to find the right set of sensing capabilities that may be employed to serve the wider range of application contexts.

Computer vision, thanks to improvement in video-based sensing, is close to achieving the objectives set by *general-purpose sensing*. Cameras allow the possibility to gather a huge amount of information, which can be further analyzed with the newest powerful Deep Learning (DL) algorithms. Thanks to techniques such as object detection and image classification, boards equipped with cameras could use video streams along with data acquired by sensors to predict complex events that fully describe the environment of interest. The issues with cameras, however, is to preserve the privacy of people, in fact video analysis should not be invasive and should not be used to derive and acquire personal information. The right trade-off between information capture and privacy assurance is a major issue.

While providing several pieces of information, cameras also acquire sensitive data which could lead to privacy issues. In fact, people do not like the fact that their data is stored in an unknown server. Several attempts have been performed to overcome these issues such as filtering the video stream to mask out sensitive information [13], but the most innovative one is to perform computer vision sensing at the edge of the network, i.e. as close as possible to end users. Exploiting EdgeAI, data doesn't need to be stored since they are processed and analyzed in real-time [14].

2.2 Digital Twin

According to Kritzinger, et al. [15] there are plenty of definitions of DT and there is not a common one, since it depends on the research's focus area. The term "Digital Twin" - as a digital equivalent to a physical object - appears for the first time in a Product Lifecycle Management (PLM) cours a the University of Michigan by Michael Grieves [16] in 2003. In his paper, Grieves, presents a three-dimensional concept model of a DT, where the main actors are the physical object (PO) (in the real space), its virtual counterpart -

i.e. the logical object (LO) - (in the virtual space) and the data/information connection between them. Hence the Digital Twin itself, is nothing but the digital side (LO) of this model, which continuously interacts and receives information from sensors placed on the real object. Moreover, it can respond to environmental stimuli using actuators.

Later on, the rise of Industry 4.0 and the further developments in the IoT field - such as remote sensing, data ubiquity, automation and actuation capabilities - led companies and institutions to focus on manufacturing and smart objects. On the face of it, LNS research defines a DT as "an executable virtual model of a physical thing or system" [17]. They highlight that each physical thing is defined in the real space by its own attributes and characteristics. Both of them can be captured and portrayed into the virtual space, creating the object's logical counterpart.

As it is pointed out in [8], DTs present several properties including *(i)* representativeness, *(ii)* contextualization, *(iii)* modeling, *(iv)* reflection and *(v)* entanglement. Each of them contributes to describing the fundamental features of the DTs.

This property allows logical objects to reflect exactly the real ones in a digital domain, paving the way toward new applications. In fact, both traditional computer-aided design (CAD) and advanced sensor-enabled IoT systems cannot compare with the real DTs' power, since *(i)* CAD lives in the simulation worlds, without interacting with the real objects; and *(ii)* advanced IoT systems - although they acquire real-time data from sensors on the field - don't favor interaction between the components and the full process life cycles.

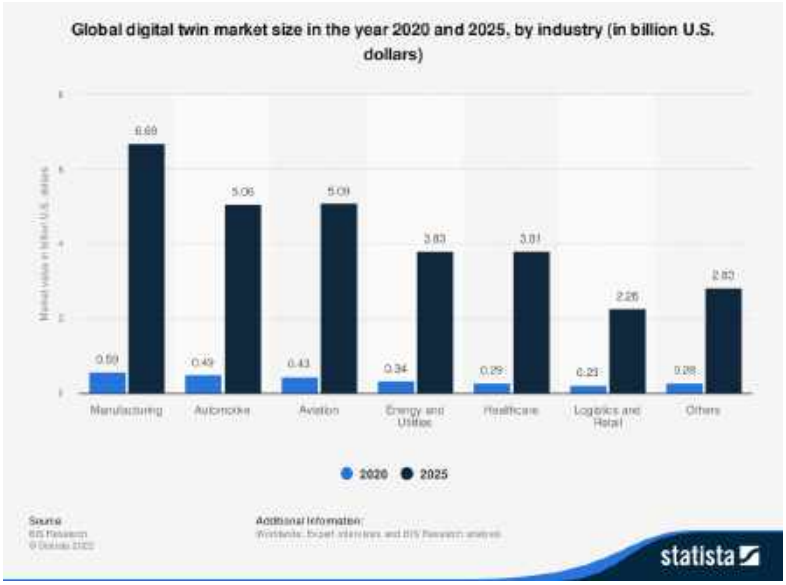


Figure 2.2: *Global digital twin market size in the year 2020 and 2025, by industry (in billion U.S. dollars) [18].*

Increasingly cheaper constrained devices, with high computing capabilities, are pushing several industries to invest in DT and its technology. As mentioned by Perrot, et

al. [19], *(i)* automotive and aircraft sectors can not do without these tools, since they are the key actors for pursuing innovation; *(ii)* energy sector takes advantage of DTs to improve and monitor both their plants and their production; *(iii)* health-care is using these technologies to develop IoT solutions able to heal chronic diseases; *(iv)* smart cities exploit the digital world to improve several activities, from traffic management to pollution monitoring; and *(v)* manufacturing adopted digital twins to boost productivity and efficiency. By 2025, the global DT market size is about to grow exponentially, reaching over six billion U.S dollars in the manufacturing sector [18] (Figure 2.2).

2.3 Digital Twins and Signatures

LOs are able to faithfull represent POs in the virtualized space because they can capture the main POs' characteristics, i.e. those features that describe real objects in the application context. These aspects are also known as *signatures*, since they uniquely represent that specific thing. For instance, each vehicle has its own engine sound (even electric cars for safety reasons [20]), emits a certain number of pollutants (CO_x and NO_x among others [21]) and has its own geometry (captured by computer vision). The combination of the three of them, could be used as a signature able to identify the specific kind of vehicle (and even the model). Moreover, by collecting the sound of engines that need maintenance, it is also possible to predict malfunctioning of vehicles avoiding dangerous situations.

Therefore, by acquiring signatures from POs, DT became a repository which stores and then provides APIs, allowing third party applications to access this information. AI and ML models could be trained using signatures coming from LOs, improving their inferences. In fact, detecting and classifying a specific vehicle's model in a large environment such as smart cities, turns out to be very challenging. There are plenty of vehicles, some of them similar to each other, even for the most accurate object detection model it is hard to provide precise predictions. Using unique vehicle signatures could overcome this issue. Figure 2.3 shows how data are gathered from the POs and stored into a repository which can be used to manage LOs. In this way each LO is able to reflect the status data of the corresponding PO, i.e. keeps active the remote virtual link with the real object (entanglement).

2.4 Digital Twins for large environment: Smart Cities

The concepts of Digital Twin and Smart Cities go hand in hand. Cities are nothing but complex living systems, or rather a system of systems [22] in the real world. Managing them has become really hard due to the population growth [23]. Digitalization (by improving software and hardware capabilities) is the only key to solve the increasingly high demand which big cities as well as suburban or rural areas have to face. DTs represent an effective means to achieve this level of innovation as fast as possible. In fact, DTs are basically virtual evolving models of the physical world. The improving of

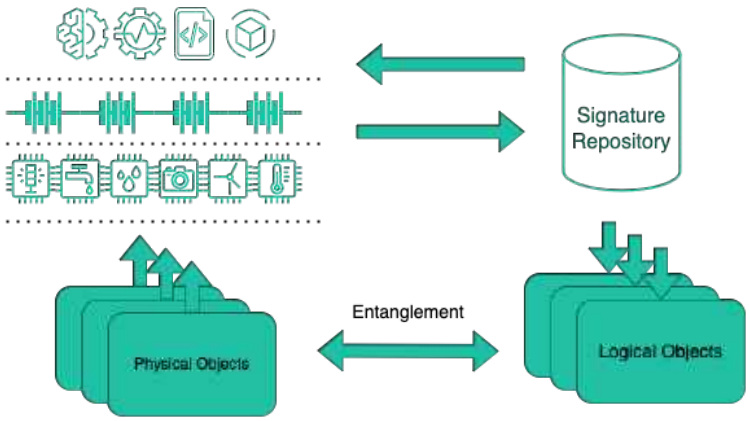


Figure 2.3: *Digital Twins and signatures overview*

sensing capabilities - due to the rise of IoT - along with the new abilities to manage and program a virtualized space, can favor this transition toward digitalization.

Although the initial focus by researchers about DT and its applicability was in the smart manufacturing sector, currently they are trying to exploit DTs representations for targeting many different applications domains. However, Nochta, Wan et al. [24], point out that when dealing with complex and heterogeneous systems, DT applications have to take into account several information, such as the urban and socio-political context, to produce value for citizens. The authors highlight the need for guidelines, or rather a framework able to direct Digital-Twin-based solutions toward workable systems and useful results.

Back in the early 2000s, the BIM (Building Information Modeling) technology [25] was introduced by the architecture, engineering, and construction (AEC) industry to bring innovation by reducing architectural projects costs and delivery time, enhancing digitalization and softwarization of urban areas. Using BIM, each single building is designed virtually, using 3D computer-aided design (CAD), and tested in order to achieve the projects' specification. Then, it is built physically in the real world [26]. Projects continue to "live" virtually for further analysis with the aim of improving and maintaining the physical building. The availability of new models or techniques can be readily used and the old buildings can be tested with the new solutions. By exploiting remote sensing in the physical building and by favoring the communication with its digital representation, researchers came up with the idea of Building Twin [27]. Key enablers in this approach are data, both static and dynamic, coming from several sources. Data manipulation and checking have the goal of gathering as much information as possible from the physical environment and to use it for flushing out potential issues or design flaws in the project. Data represents the bridge between the physical building and its digital counterpart (Figure 2.4). The latter along with modeling, bring knowledge to the softwerized design and implementation beyond the single building.. The so-called

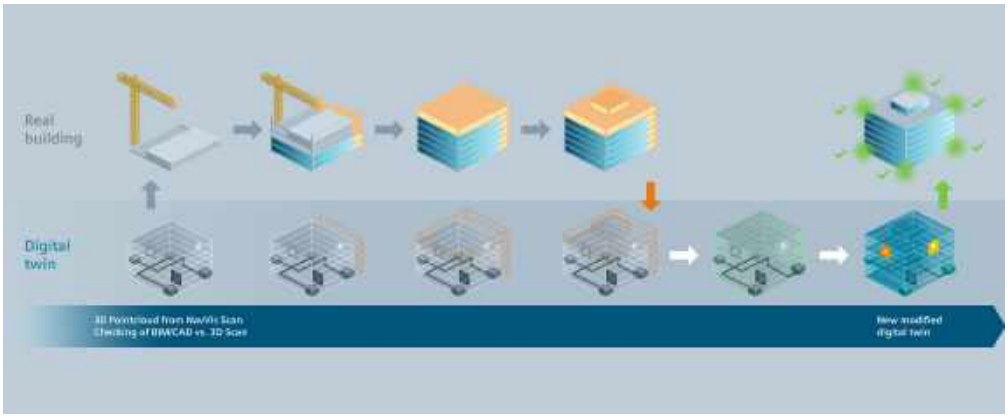


Figure 2.4: *Real building and its Digital (Building) Twin [28].*

Building Twin can now completely understand its surroundings, interact with either third-party applications or citizens and use the information sensed to adapt themselves to human needs.

BIM technology and DTs can also be applied to other cities' backbone, such as the transportation infrastructures [29]. The idea is the same as the one used for the Building Twins, i.e. to develop a 3D CAD model using BIM; to collect as much data as possible from the physical infrastructure (bridge, local road, highway, railway...) through pervasive sensing; and to guarantee the communications between the digital and the real object. Dealing with the DT of a specific infrastructure led to several benefits, such as:

- monitoring continuously and remotely the infrastructure status and its issues;
- performing some simulations to avoid traffic congestion and to improve viability [30, 31];
- performing further forecasting to avoid structural errors that could threaten citizens life;
- monitoring the life cycles of infrastructures to program maintenance intervention [32].

Kumar, S.A.P., Madhumathi, R., Chelliah, P.R. et al. [31], presents in their paper a "*digital twin-centric approach for driver intention prediction and traffic congestion avoidance*". Their aim was to offer a futuristic view of developing viable, self-learning, and automated traffic management systems. The solution proposed consists in *(i)* gathering real-time data from the current road and vehicle, forwarding them to fog and edge devices for data analytics; *(ii)* predict driver behavior and intention using accurate machine learning and deep learning algorithms; *(iii)* using data lake in the cloud for storing historical information. Several sensors and cameras are used to capture situational details from the environment, building a DT for the road and a DT for each vehicle.

Hence, through the proposed Virtual Vehicle model they perform the "driver intention" prediction.

Research is moving toward building a huge system of systems, able to manage each single subsystem - Building Twins, Intelligent Transport Systems... - and to promote both the integration and the interaction between them. However, this potential has not been fully developed thus far. Up until now, the most advanced implementation of DTs in smart cities comprends the geospatial mapping of cities' infrastructures to 3D CAD models - using BIM technology - along with real-time data coming from either existing cities' data lake or the environment. For example, the city of Zurich developed a 3D spatial, digital model of the city itself, providing one of the first city-scale Digital Twins of the entire urban area [33]. The project - named "Smart City Zurich" - consists of a common framework for all the numerous IoT projects that have already been implemented throughout the past years, with the aim of creating a digital city's counterpart to simulate urban planning in climate change. Like Zurich, also Munich [34], Helsinki [35], Rotterdam [36] and Rennes [37] are moving toward the city-scale DT. On the other hand, other projects such as Virtual Singapore [38] are intended to provide a 3D digital platform for citizens and researchers that works as an enabler for users to develop sophisticated applications which can solve the emerging and complex challenges that megacities are forced to deal with. The city-nation of Singapore was the first to point out the difference between mapping and twinning: the digital twin of the city is constantly up to date with real time data.

Chapter 3

Background

3.1 Hardware Components

Nowadays, there are plenty of powerful hardware devices available on the market. Choosing the right component could truly affect the project's success. Therefore, it is fundamental to have an overview of all the available devices, studying their specifications along with their advantages and drawbacks, and how they could fit into the project. However, this technological area rapidly changes. The proposed system tries to use the best solution currently available, taking into account that there is no assurance that, in a short time, better devices will appear. In this chapter I will present the hardware components chosen for this thesis.

3.1.1 Sensor Boards

According to the system's requirement, environmental and pollution data need to be gathered from the environment. In order to do that two different sensor boards have been selected.

The first one is the Gravity MEMS Gas Sensor Board, developed by DFRobot, which supports the detection of various gas concentrations like CO, C_2H_5OH (Alcohol), H_2, NO_2, NH_3 thanks to the built-in MiCS-4514 sensor, and integrates various gas concentration conversion formulas in the code to facilitate the testing and usage for end-users. Thanks to its I2C digital outputs and its 3.3 5.5V wide voltage inputs, the module is compatible with several boards, such as Arduino, ESP32, Raspberry Pi and so on. It is able to work between -30 and 85 °C and it features the following measuring range:

- 1 – 1000ppm(Carbon monoxide CO)
- 0.05 – 10ppm(Nitrogen dioxide NO_2)
- 10 – 500ppm(Ethanol C_2H_5OH)
- 1 – 1000ppm(Hydrogen H_2)
- 1 – 500ppm(Ammonia NH_3)

(a) Google Environmental Sensor Board *(b) Gravity MEMS Gas Sensor Board*

Figure 3.1: *Sensor boards*

- $>$ 1000ppm(Methane CH_4)

Thanks to the *DFRobot MICS Library* it is pretty easy to extract and process raw data coming from the sensor using either I2C protocol or analog values.

On the other hand, to gather environmental parameters, the Google Environmental Sensor Board has been chosen. The latter is a general purpose sensor board able to extend sensing capabilities of single-board computers such as Google Coral Boards family and Raspberry Pi. In order to do that, it can be attached to the controller's pins thanks to the 40 equipped female connectors. It acquires atmospheric data from the environment such as *(i)* temperature; *(ii)* humidity; *(iii)* barometric pressure; *(iv)* and light level. Moreover, the Environmental Sensor Board supports UART and I2C communications, as well as Pulse Width Modulation (PWM) technique. In addition, the board allows connecting other sensors through four Grove connectors. To program the board one need to install and therefore use the apposite python library: *coral.enviro.board*.

3.1.2 Cameras

In order to perform computer vision, i.e. the vehicle detection task, the system needs a camera module able to capture real-time video streams. The choice fell on Google Coral Camera. The latter is a 5 megapixel camera module which can be connected to Coral Boards family, and other boards as well, through MIPI-CSI interface. With a dimension of 25mm x 25mm, It provide several features, such as:

- auto exposure control;
- auto white balance;
- auto band filter;
- auto 50/60Hz lumination;
- auto black level calibration.

It is important to point out that this product has been chosen for continuity reasons, since a high percentage of devices involved in the project are developed by Google Coral.

Of course, other camera modules can be used, even those with USB connectors such as the ones used with personal computers.

Figure 3.2: *Google Coral Camera module*

3.1.3 ESP32 WROOM 32D Board

The ESP32 family belongs to Microcontroller units (MCU), which are computer board platforms with CPU, memory, busses and built-in peripherals indispensable for reading connected sensors or driving actuators. In particular, the ESP32 WROOM 32D module, integrates Wifi, Bluetooth and Bluetooth Low Energy (LE), by targeting a wide range of applications, from dealing with wireless sensor networks till managing complex tasks such as voice encoding, music streaming and MP3 decoding.

This board MCU can be programmed on Arduino IDE, using a C/C++ library called *Wiring* which makes it much easier to work with other input/output hardware modules (sensors, actuators, etc.) via software. Moreover, by using proper libraries on the IDE, it is possible to connect the board to a router, i.e., to the world wide web, enabling low-latency communications for processing raw data coming from the environment and forwarding them to proper network nodes.

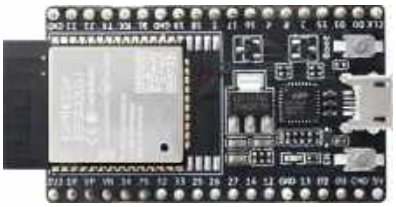


Figure 3.3: *ESP32 WROOM 32D Board*

3.1.4 Google Coral Dev Board

The Coral Development Board, is a new platform developed by Google able to bring the power of AI into constraint devices. Deeper, the Dev Board, is a single-board computer (just like Raspberry Pi) equipped with an Edge TPU (Tensor Processing Unit),

which allows these devices to achieve significantly higher performance than standard CPUs. This is due to the equipped Google System on Module (SoM) which gives to the board enough power for running fast ML inference. Moreover, Google Coral Dev Board has both an embedded CPU and an embedded GPU, is equipped with 8 GB of flash memory and 1/2/4 GB of RAM (depending on the version chosen). Moreover, the development board supports wireless connectivity, both Wi-Fi and Bluetooth 4.2. In fact, it is equipped with a Wi-Fi 2x2 MIMO module able to support several standards: 802.11b/g/n/ac for either 2.4 or 5GHz communications.

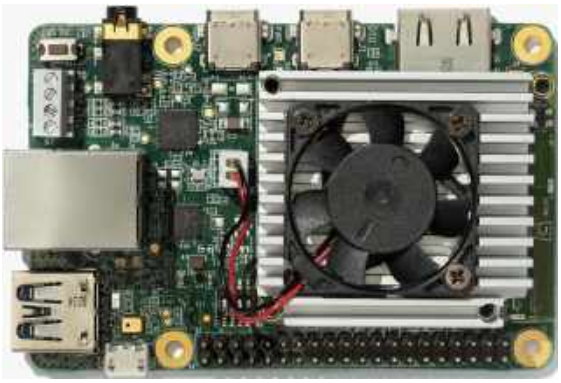


Figure 3.4: *Google Coral Dev Board*

The core component of the while board is the Edge TPU, which is placed at the heart of the device. According to Google's benchmarks [39], it is able to perform 4 trillion operations (tera-operations) per second (TOPS), dissipating just 2 TOPS per watt. To deal with all this computational power at the edge, Google developed TensorFlow Lite, a toolkit derivative of the well known TensorFlow library, that performs complex deep learning model, such as object detection, sound recognition, image classification, on Coral constraint hardware, taking advantage as much as possible of the equipped TPU.

Coral Dev Board runs Mendel Linux, a lightweight operative system branch of Debian Linux. The latter has been optimized to make it the perfect OS for embedded devices. Although one can connect a keyboard (thanks to the USB connector) and a monitor (using the HDMI 2.0a interface), there is no user interface, just a debian linux shell which relies on a debian file system. Mendel Linux comes with all the necessary libraries and software packages a developer needs to work with the board, i.e both python and C++ libraries, Edge TPU API and runtime. In order to connect safely to the board using SSH/mDNS, Google developed Mendel Development Tool (MDT). The latter a the command-line tool that allows the communication with all the devices running Mendel Linux, including this board. Using MDT, it is possible either to transmit or download information directly from/to the board (as long as it is wired to the computer which is transmitting/downloading).

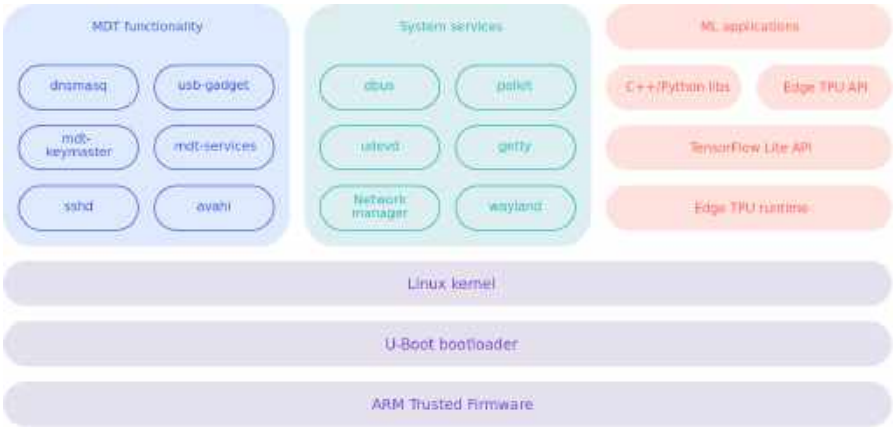


Figure 3.5: *Mendel Linux block diagram [39]*

3.2 IoT Technologies and Tools

In this section the main technologies used for the implementation of the proposed system are introduced and analyzed. For each of them, its principal features and characteristics are described by motivating its adoption for the final implementation. Table 3.1 sums up all the technologies employed.

Purpose	Technology
Communication	MQ Telemetry Transport (MQTT) (3.2.1)
Dispatching	Node-RED (3.2.2)
Telemetries Data Storage	InfluxDB (3.2.3)
Data Displaying	Grafana (3.2.4)
Digital Twin	Eclipse Ditto (3.2.5)
Edge AI	TensorFlow Lite (3.2.6)

Table 3.1: *List of all the technologies used divided per field of application*

3.2.1 MQ Telemetry Transport Protocol

MQ Telemetry Transport (MQTT) [40] is a low bandwidth communication protocol widely used in the IoT field. It was released for the first time by Andy Stanford-Clark from IBM and Arlen Nipper in 1999. Thanks to its ability to enhance Machine to

Machine (M2M) communications and the wide spreading of Wireless Sensor Networks (WSNs) , MQTT protocol became very popular during the 2000s.

MQTT implements the publish/subscribe architectural pattern (see Figure 3.6), that is an asynchronous communication mechanism in which two or more clients, defined publishers or subscribers based on their role within the system, communicate through a central server that routes messages. The key actors in MQTT communications are the following:

- **Topic** - is an endpoint that connects clients during the communications. Is an UTF-8 string that the publisher should know to send messages to subscribers.
- **Publisher** - is a client which sends (i.e. publish) messages using a specific topic. It has to know the address of the broker for sending messages in the proper way.
- **Subscriber** - is a client which receives messages sent by publishers. It subscribes on a specific topic to the broker. Each time a publisher sends a message on that topic, the subscriber receives that message.
- **Broker** - is the central server which manages topics and communications between publishers and subscribers. It keeps track of all the clients subscribed to specific topics. As soon as it receives a message along with the topic from the publisher, it checks which clients are subscribed to that topic. Finally, it forwards messages to the proper subscribers.

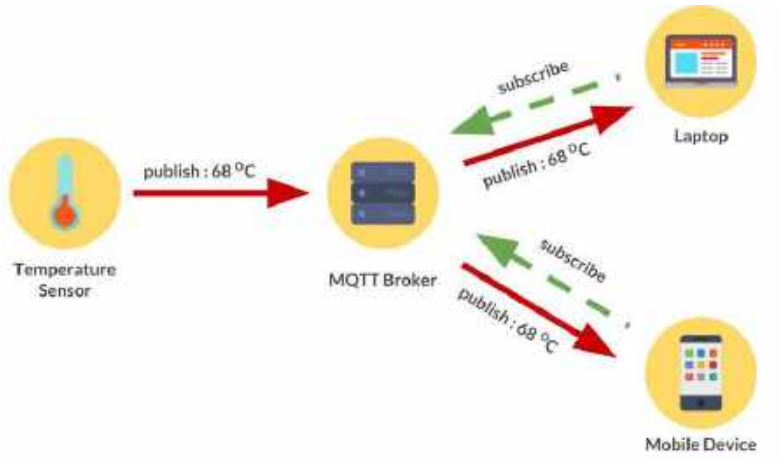


Figure 3.6: *Publish/Subscribe model overview [41]*

MQTT supports three different Quality of Service (QoS) levels both for publisher to subscriber and broker to subscriber communication. One can choose among the following:

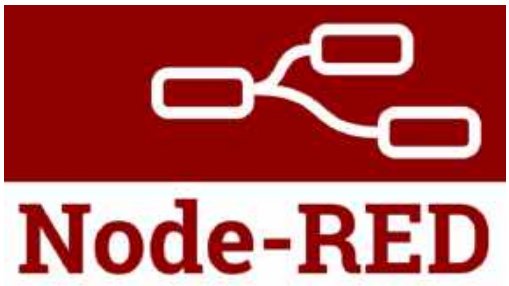
- QoS 0 (at most once). The publisher sends a message to the broker without receiving the ACK.

- QoS 1 (at least once). The publisher sends a message to the broker, then stores it temporarily waiting for an ACK from the broker. If it doesn't receive the ACK within 10 seconds, the publisher sends the packet again.
- QoS 2 (exactly once). Implements a 2-step ACK process. The sender, whether it is the publisher or the broker, sends a packet in the direction of the receiver, broker or subscriber respectively, and subsequently keeps temporarily the message pending the PUBREC. If it doesn't receive it, the message is sent again, with the DUP (duplicate) flag enabled in the header. Once the PUBREC is received, the sender sends a new packet, of type PUBREL, containing the initial PUBLISH packet id. This packet indicates to the receiver that the sender is deleting the initial message. The receiver replies with a PUBCOMP and thus closes the communication, waiting for a new packet.

QoS 2 requires more time than the other levels. Either way, the publisher has no guarantee that the subscribers actually receive the message. For that purpose, one can use the retain features which ensure that messages are sent and received.

Nowadays, several lightweight protocols are available on the market. Nevertheless, in this work of thesis MQTT has been chosen among other protocols for several reasons: *(i)* energy efficiency, suited for low power device; *(ii)* simplicity and flexibility; *(iii)* huge number of libraries and tools which implement the protocol; *(iv)* supports low bandwidth and fast communications.

3.2.2 Node-RED



Node-RED [42] is a powerful development general purpose tool for building Web-of-Things applications. It exploits the flow-based programming (FBP) [43] paradigm for wiring hardware devices along with API and web services. Node-RED was written in JavaScript as a web-based tool so as to be cross-platform.

The main concepts on which Node-RED is based on are: *(i)* **node** and *(ii)* **flow**. A node is an atomic element, equipped with one or more input/output ports. Most of the nodes in the Node-RED's palette implements a black box model: the code implemented inside the node is most of all unknown, instead the input/output specifics are available. Node-RED provides a wide range of nodes in the palette, each of them adds features to the developed flow. Moreover, the tool provides an integrated market from which one can download and add nodes to its flow.

Current legacy of nodes supported by Node-RED provide several functions, such as:

- debugging using the debug nodes;
- signal triggering using the trigger nodes;
- communication services, which endors the most known communication protocols (HTTP, MQTT, serial, radio, AMQP, etc.);
- condition operation, i.e. filtering, switching, etc.;
- data persistence.

Nodes are connected one to another using **wires**, i.e. port channels from the output port of one node to the input port of the other node. To transmit information, wires bring within them messages from the output to the input port of the nodes. Messages are nothing but JavaScript object which have two main fields:

- *topic* - represents the message's title;
- *payload* - contains data, which could be text or encrypted information.

Nevertheless, it is possible to add a new field to the **msg** object, depending on the programmer's needs. A set of nodes, wired together, builds a **flow**. Several flows can communicate with each other using *link nodes*. Another important feature of Node-RED is the *variables management*. A variable can be instantiated in the *function nodes* and they can be of three types, depending on their visibility:

- *context* - it is the default level, limits the variable's scope to the node it has been created;
- *flow* - allow the variable instantiated to increase its scope to all nodes within the flow;
- *global* - allow the variable to be visible also in other flows, not only in the one instantiated.

Variables can be used as messages as well to connect nodes and flows together.

For this work of thesis the choice fell on Node-RED since it is a powerful tool, easy to use and suited for building IFTTT (If This Then That) solutions. Its ability to wire together web services (such as the other one chosen for this thesis) and hardware devices (Sensor Boards, Google Coral Dev Board, etc.) guarantees to develop and build a complex, scalable and available system, which makes this tool perfect for the aim of this thesis.

3.2.3 InfluxDB

InfluxDB [44] is a highly performing open source Time Series Database based on a SQL-like query language called InfluxQL. It has built-in time-centric functions for querying a data structure composed of measurements, series, and points. Data in InfluxDB is

organized by "time series", which contains a measured value, like "cpu_load" or "temperature". Time series have zero to many points, one for each discrete sample of the metric. Each point consists of several key-value pairs called the fieldset and a timestamp. When grouped together by a set of key-value pairs called the tagset, these define a series. The difference between tags and fields is that tags are indexed, which means that queries on tags are faster compared to queries on simple fields, which are not indexed. Finally, series are grouped together by a string identifier to form a measurement. Conceptually you can think of a measurement as an SQL table, where the primary index is always time.

InfluxDB offers so-called *retention policies*. A retention policy describes how long InfluxDB stores data (DURATION) and how many copies of that data are stored in the cluster. The default retention policy with infinite duration and no replication is called "autogen", which has an infinite duration and a replication factor of 1.

In this thesis, to use a time series database allows to keep trace of telementries during the time. By storing the value along with a timestamp allow to perform temporal query in an easy way, which can be used for further analysis and visualization.

3.2.4 Grafana

Grafana [45] is an open-source web application for interactive data visualization and analysis. It allows inspection of info-graphics and alarms, unifying various data sources. Since it features a native integration with the most known time series databases (InfluxDB, Prometheus, etc.) and monitoring platforms (Sensu, Icinga, etc.), Grafana is one of the most important tools for monitoring tasks and therefore for data scientists and researchers.

Grafana was developed by Grafana Labs and it splitted in frontend and backend, written in Typescript and Go respectively. Go is the Google brand new programming language, which improves performance in accessing huge datasets and processing tons of data. Hence, the latter is a good feature which makes Grafana an attractive choice.

Data is visualized on Dashboards. Each dashboard contains one or more panels. Once the data source is connected to Grafana service, the creation of these panels can be done through interactive query builders with the minimum programming effort. Every panel could be connected to different data sources, offering a way to unify data in just a single user-friendly dashboard. All the information about dashboards (i.e metadata,

properties, designing elements, etc.) are stored into JSON objects. Those objects can be accessed through the dashboard's settings.

Data and telemetries can be displayed in panels using several visualization options, so users can choose the solution which fits better for their visualization task. Table 3.2 shows a list of visualization sections and types available on Grafana [46]. Each panel can then be modified according to users' will, for instance one can completely personalize its graph by choosing the color of the panel, the shape of the values' line, the text on the both axes, etc.

Options	Types
Graphs & Charts	Time series, State timeline, Bar chart, Histogram, Heatmap, etc.
Stats & Numbers	Stats, Gauge, Bar Gauge
Misc	Tables, Logs, Traces
Widgets	Text panel, Alert, Dashboard list

Table 3.2: *Visualization panels type in Grafana [46]*

For all the features above mentioned, Grafana reveals to be the perfect choice to solve the monitoring tasks of this work of thesis. It offers, indeed, a native integration with InfluxDB, hence all the environmental and pollutant data can be stored and then visualized on proper Grafana Dashboards with the minimum programming effort. Moreover, Grafana is also good looking and user friendly, which is an important feature that improves the user experience in programming complex monitoring panels.

3.2.5 Eclipse Ditto

Eclipse Ditto is a framework to interact with IoT devices by providing the Digital Twin pattern. It presents itself as an IoT middleware able to mirror physical devices as digital representations in the cloud.

Nowadays there are plenty of different communication protocols and device types available on the market. Eclipse Ditto provides a way to access them by a set of resource-based API which allows to abstract the complexity of dealing with different devices and connection types. A single device is structured into its main aspects and functionalities and is equipped with a public API that helps to access each single object's features. Ditto also provides a set of policy rules which can be modified by developers and define

how objects interact with each other. For instance, is it possible to point out which are the objects that can access a specific device's feature and with which operational mode (read-only, admin privileges, etc.). In this way, Ditto guarantees that devices are accessed only by authorized objects or users. By exploiting this approach, a single device is turned into a standard web service which can be queried and managed like any of the other services within a third party application.

A digital (virtualized) object in Eclipse Ditto is nothing but a JSON object made of devices' attributes (metadata) and features. Objects metadata are information about the device manufacturer, the geographic location, the software version, etc. which can be useful to query devices in the Digital Twin repository. Features instead, contains values received by the physical device such as the temperature, humidity, etc. Devices can be linked to a model, so as to provide a standard way to describe similar objects.

In order to connect a scalable number of devices to the backend, i.e. the Ditto server, a connectivity layer is needed. Software such as Eclipse Hono or MQTT brokers such as Eclipse Mosquitto are naturally integrated by the framework, allowing a huge number of devices to continuously send and update their real-time status.

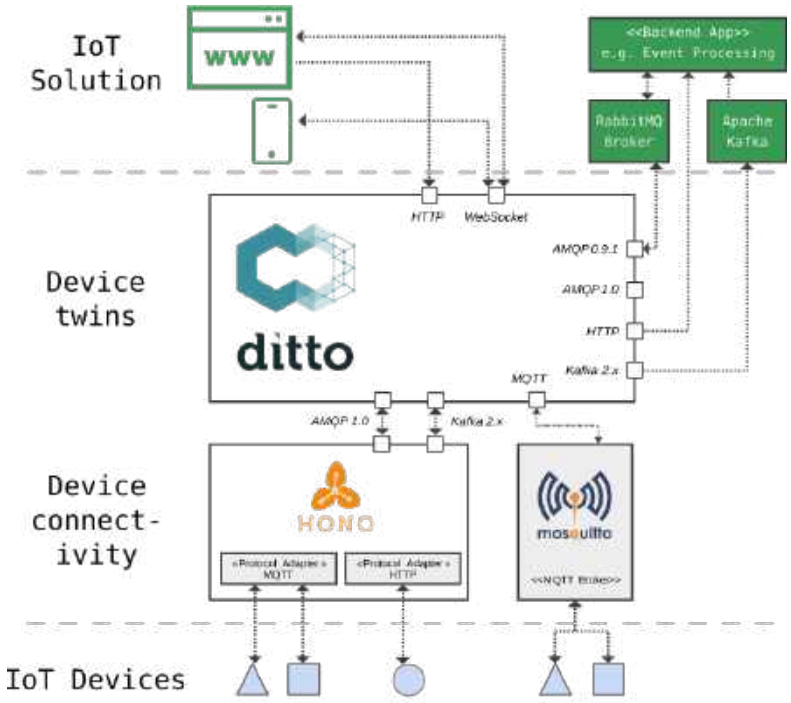


Figure 3.7: *Eclipse Ditto's high-level overview*

In this thesis, Eclipse Ditto is revealed to be the perfect framework for developing the DT concept. It is an open source software which provides native integration with several communication protocols and software as well. Moreover, being an Eclipse project, supports other projects such as *(i)* Eclipse Hono and Eclipse Mosquitto for connecting a huge number of devices; *(ii)* Eclipse Vorto for describing objects models and struc-

tures, etc. Hence, such a framework exploits the DT paradigm by adding reliability and availability to the overall system and by enhancing at the same time accessibility and connectivity.

3.2.6 TensorFlow Lite

TensorFlow (TF) Lite is a ToolKit provided by TensorFlow 2.0 which enables on-device ML on embedded, mobile and edge devices. Such kinds of devices usually have some constraints in terms of memory, power consumption, computational power, etc. which make it hard to deploy on them either ML or DL models. Hence, TensorFlow Lite brings optimized ML models for on-device machine learning by addressing 5 principal constraints:

- latency, since there's no round-trip to a server and back;
- privacy, since data doesn't leave the device;
- connectivity, which is not required;
- size, reduced model and binary size for memory space optimization,
- power consumption, to guarantee efficient inference and a lack of network connections.

In order to reduce models' size, a TF Lite model is exported in an efficient portable format, named FlatBuffers [47], which is identified with the extension *.tflite*. The latter presents several advantages with respect to the standard TF's protocol buffer, such as *(i)* reduced size, due to the small code footprint and *(ii)* faster inference, since input data are directly fed into without performing other packing/unpacking steps. Optionally, a TF Lite model could also include additional information, i.e. metadata. The latter are further human-readable models' descriptions and machine-readable data for enhancing on-device inference. A TF Lite model can be achieved in the following ways:

- use an existing TF Lite models;
- create your own TF Lite model, by exploiting the TF Lite Model Maker library to train a model with a custom dataset;
- convert a TF model into a TF Lite model, by using TF Lite Converter.

For its feature of bringing AI in edge devices and its native integration with Google Coral Dev Board, TF Lite appears to be the right tool to achieve the objectives of this thesis. Its ability to optimize on-device ML and DL tasks, by reducing memory usage as well as the bandwidth and power consumption, is a key point that could enhance the overall system performance.

Chapter 4

Digital-Twin-based Traffic Monitoring System

4.1 Goals and Purposes

In order to promote the virtualization of a large physical environment, and therefore to model the DTs promoting *general-purpose sensing* approach, the outlined system should expose the following features:

- gather data from the environment, using several elementary sensors placed on the environment itself;
- process all the information acquired using low-latency and low-cost communications;
- store relative information in the database to ensure persistence of data;
- apply early on AI algorithms in order to identify and classify physical objects.

Hence, both system's hardware and software architecture must be aligned in order to collaborate with each other to pursue the desired goal. Moreover, the TMS will be able to detect the specific type of vehicle passing by the observed road, and send the information acquired to the backend logic for the DT creation and management. To add this feature, the system should implement also some deep learning algorithms able to *(i)* detect the vehicles; and *(ii)* count them. Since the system design is using video capture, it is necessary to apply real-time inference on a video stream coming from a camera.

In order to reduce the latency and improve system's performances by moving computation instead of data, this TMS exploits the concept of Edge-AI [48], i.e. brings the AI algorithms at the edge. To deploy complex models in constraint devices and to perform real-time inference as fast as possible, it is needed to use lite ML models. In fact, both the memory and computing power of edge devices are often constrained. Models can be optimized in a variety of ways to allow them to run within these restrictions. Furthermore, several enhancements make it possible to leverage specialized hardware for rapid

inference. Figure 4.1 represents the differences between AI at the edge and classical AI solutions in the cloud.

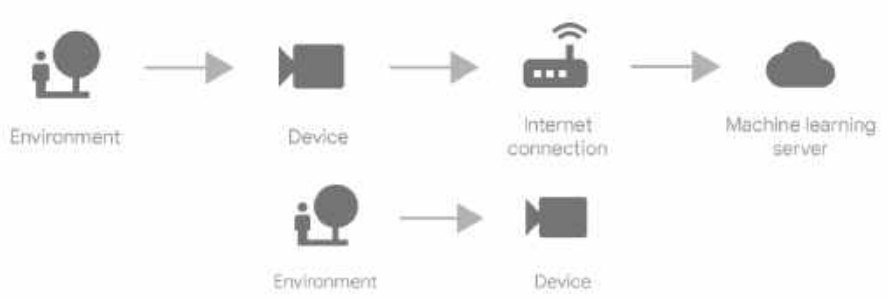


Figure 4.1: *Cloud based approach vs Edge AI [49]*

Of course, shrinking the model size leads to some accuracy drop, but it allows the model to occupy less storage space and use less RAM when it runs, reducing the inference latency during its execution. The latter could also impact the power consumption, which for some devices is a critical restriction. Hence, by exploiting the concept of Edge-AI, the TMS has several benefits such as:

- low-latency communications, by avoiding the round-trip delay of sending data to a remote server since the model is at the edge;
- low-latency inference, since it uses lite ML/DL models;
- high availability and reliability, even in case of no internet connection (all the computation is at the edge);
- no privacy issues, since data are just processed locally instead on an unknown server

4.2 Hardware Overview

The proposed system consists of several hardware devices that work together to achieve the desired goal, such as:

- two sensor boards - Gravity MEMS Gas Sensor Board for gathering pollutants data and Google Environmental Sensor Board for gathering environmental telemetries (Section 3.1.1);
- one camera module - it could be either Google Coral Camera (Section 3.1.2) or a normal USB Camera for gathering real time video streams;
- one microcontroller unit (MCU) - ESP32 WROOM 32D Board (Section 3.1.3) for interfacing with the Gas Sensor Board;
- and one development board - Google Coral Dev Board (Section 3.1.4) for applying complex ML algorithms at the edge of the network.

The sensor boards are responsible for gathering data concerning both vehicles' pollutants emitted and environmental parameters in the target space. Since all the GPIO pins of the Coral Dev Board are used by the Environmental Sensor Board, it was necessary to employ another MCU in order to gather values from the Gas Sensor Board. Once processed, raw data is forwarded toward an MQTT Broker to the controller service (NodeRED) in the form of structured data, i.e. where all IFTTT (If This Then That) rules are executed. Figure 4.2 shows the proposed hardware configuration.

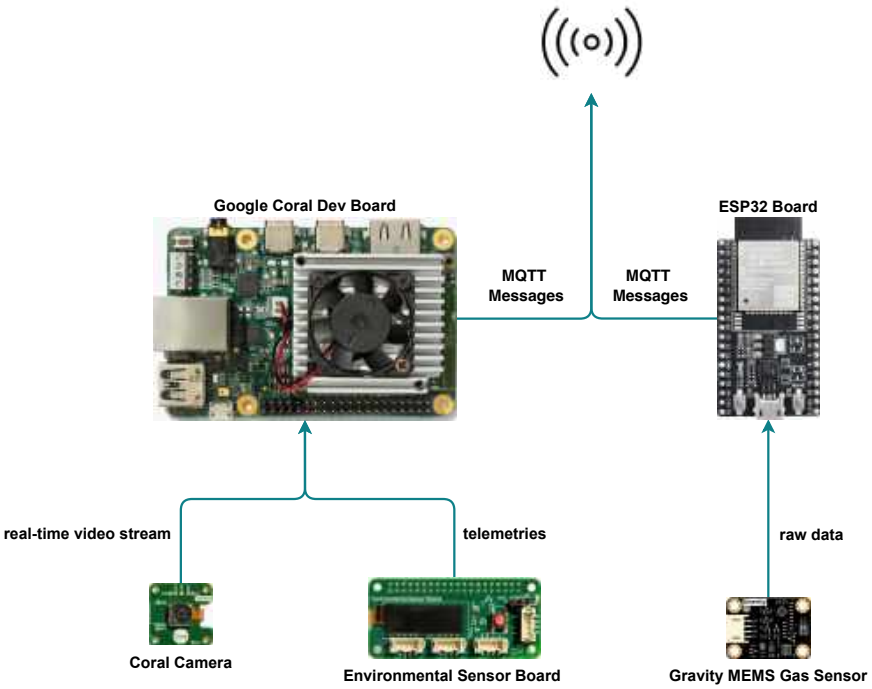


Figure 4.2: *TMS hardware overview*

4.3 Software modules

The intelligence of the Traffic Monitoring System consists of several software modules spread all over the network architecture. The Figure 4.3, shows a high level overview of all the technology used.

Since the goal is to exploit Edge AI, the scripts which perform Vehicle Detection and Counting, must be deployed as close as possible to the physical environment, i.e.on the Google Coral Dev Board. The Coral Camera, the Environmental Sensor Board as well as the Gas Sensor board and the ESP32, must also be deployed at the edge of the network, so as to gather real-time telemetries needed directly from the field.

As regards other modules, such as the backend logic (NodeRED) along with data storage (InfluxDB, MongoDB ...) and communication services (MQTT Broker) can be

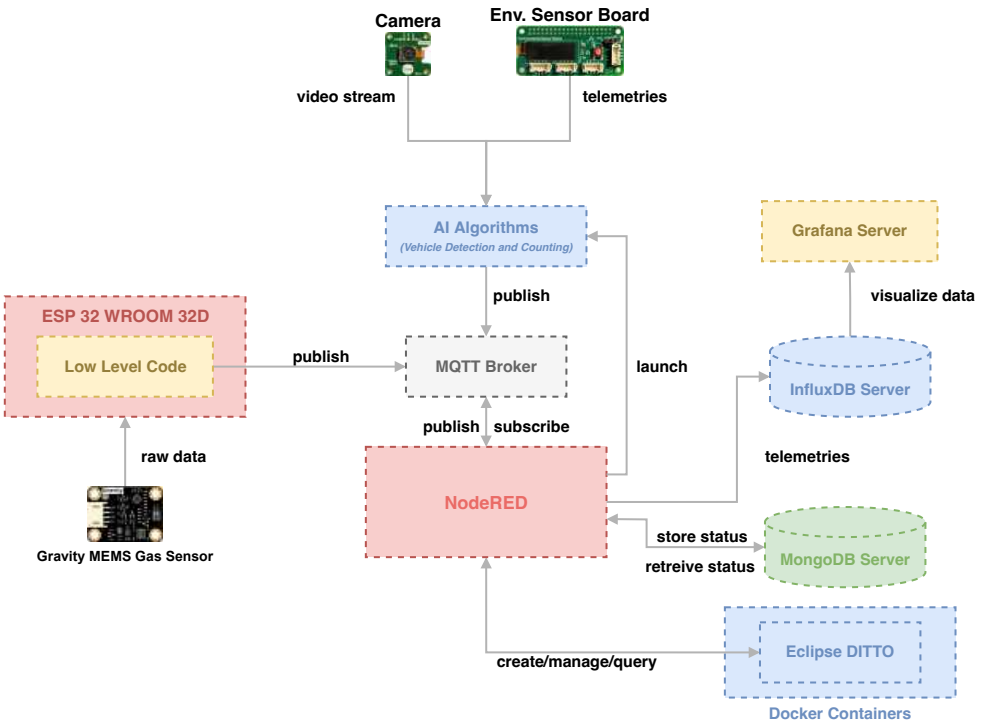


Figure 4.3: *TMS high level view*

deployed following three main techniques:

- cloud-only approach, based on the traditional cloud-centric implementation of applications where all modules run in data centers;
- edge-only approach, in which all modules run in the edge devices;
- hybrid-fog approach, in which some of the modules of an application run over different fog nodes.

Regardless the technique used, all the information acquired from the physical environment are then used either to create or manage DTs: (i) the one who represents the specific road in which the hardware devices are placed (R-DT); and one DT for each specific type of vehicle detected (V-DTs).

4.3.1 Communication

In order to either forward data and information or coordinate the work among the components, all software modules involved in the system operation exchange messages with each other continuously. The latters are sent/received using two different communication protocols:

- HTTP, used for request/response communications between the system's controller (NodeRED) and Eclipse Ditto.
- MQTT, used to send telemetries and other informations (such us vehicles detected, their current count, control messages, configuration messages) to a MQTT broker, which forward them to all the subscribers;

More in general, for sending telemetries it would be better to use MQTT over HTTP. In fact, it has been proved by professors Tetsuya Yokotani and Yuya Sasaki in their paper [50] that MQTT performs better in the IoT field than HTTP. On the other hand, to create, delete, manage and fetch multiple entries, the classical client-server approach goes the most.

One of the most important modules in the whole system's architecture is the MQTT Broker service. Since MQTT follows the publisher/subscriber paradigm (see Section 3.2.1), the broker is in charge of keeping information about all the messages' topics used by the system and the mapping between them and all the subscribers. As soon as the publisher sends a message on that specific topic, the message broker forwards it to all subscribers (which could be also just one node). In this TMS the MQTT broker is implemented using *Eclipse Mosquitto* [51], a lightweight, open source message broker, suitable for any device, from large and powerful servers to constrained boards with a limited computational power. The MQTT clients instead, are implemented as follow:

- on the ESP32 board through the *PubSubClient* library [52];
- on Google Coral Dev Board through Eclipse Paho library [53];
- in NodeRED through MQTT nodes.

As regards HTTP communications, Eclipse DITTO provides a set of REST API, by promoting secure RESTful communications between the server and multiple clients. In this way, the NodeRED service can easily either fetch or update DTs status information by building HTTP requests and processing the responses received. All these actions are possible thanks to HTTP nodes available on NodeRED's palette.

4.3.2 Data Persistence

In order to guarantee data persistence, this TMS uses two different NoSQL databases: InfluxDB and MongoDB. Each of the two has its own role inside the architecture.

Even if the TMS should be reliable and always available, to increase the robustness of the system is better to save the status of the DTs in the case of an error which leads the system to stop. In fact, when the DITTO server is shutted down, all the info about DTs is lost. To overcome this issue, each single attribute and feature of DTs can be stored in the form of JSON objects inside a NoSQL database such as MongoDB [54]. Basically, during each system start-up, the backend logic firstly checks if a R-DT with a given *things_id* exists. If yes, then a R-DT is created from its last status update, otherwise a new default R-DT is created and stored both in Eclipse Ditto and MongoDB server. In

the proposed system the status is stored in a collection named *road_dt* which belongs to a database instance called *TMS_DB*.

To store real-time environmental data, an InfluxDB (see Section 3.2.3) service is used. Being a *time-series* NoSQL database, allows to track telemetries over the time. One of the most important features of InfluxDB is that it is possible to set up a personalized *retention policy*. A retention policy describes how long InfluxDB stores data (DURATION) and how many copies of that data are stored in the cluster. The default retention policy with infinite duration and no replication is called "autogen", which has an infinite duration and a replication factor of 1.

The proposed system relies on a database called *MONITORING_STATION*. The latter has six measurements, such as: temperature, humidity, barometric, illuminance, CO and NO2. Each point has a fieldset composed by two key-value pairs:

- location (string) - which define the room in which the board is placed;
- value (float) - which is the value coming from the sensors for that measurement.

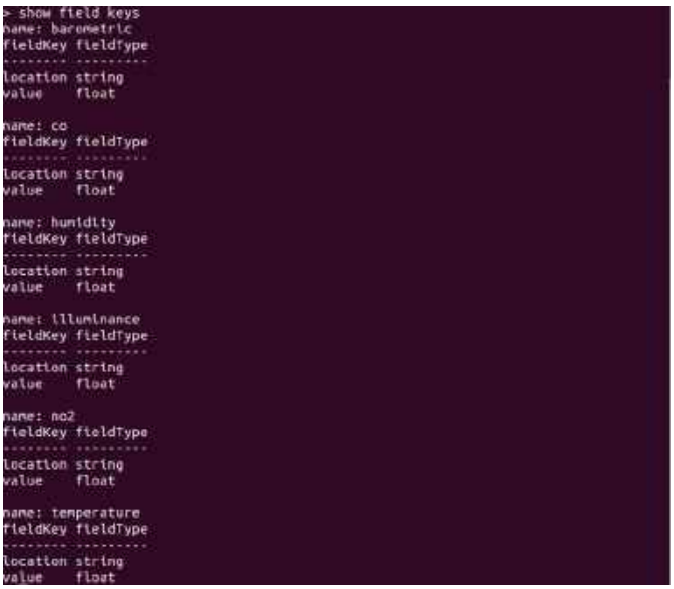


Figure 4.4: *Measurements in the database MONITORING_STATION*

Moreover, the *MONITORING_STATION* has a retention policy called *a_year* with a duration of one year (52 weeks), so as to store data for one year. Since records are not so heavy, it is important to keep data for a large amount of time to ensure Big Data frameworks to analyze them and to use them as input for their deep learning algorithms.

4.3.3 Data Visualization

System's hardware devices, along with their software modules, continuously acquire and store telemetries coming from the environment. While machines are pretty comfortable

with dealing with a huge amount of raw data, human beings are not. In order to make sense to all the information gathered and consequently transform numbers into tangible concepts, data needs to be visualized. To achieve this goal, this TMS makes use of Grafana service (see Section 3.2.4). The latter displays data in different panels, built with its interactive query-builder, inside one or several user-friendly dashboards. Thanks to the native integration for InfluxDB, it is possible to link precise measurement to configurables panels. These panels receive and display real-time data as soon as they are stored into the proper database instance.

For this TMS, by running the Grafana service, two monitoring dashboards are available:

- Environmental parameters dashboard, which contains panels related to temperature, humidity, illuminance and barometric data over the time.
- Pollutants Monitoring dashboards, which contain panels that display CO and NO_2 values over the time.

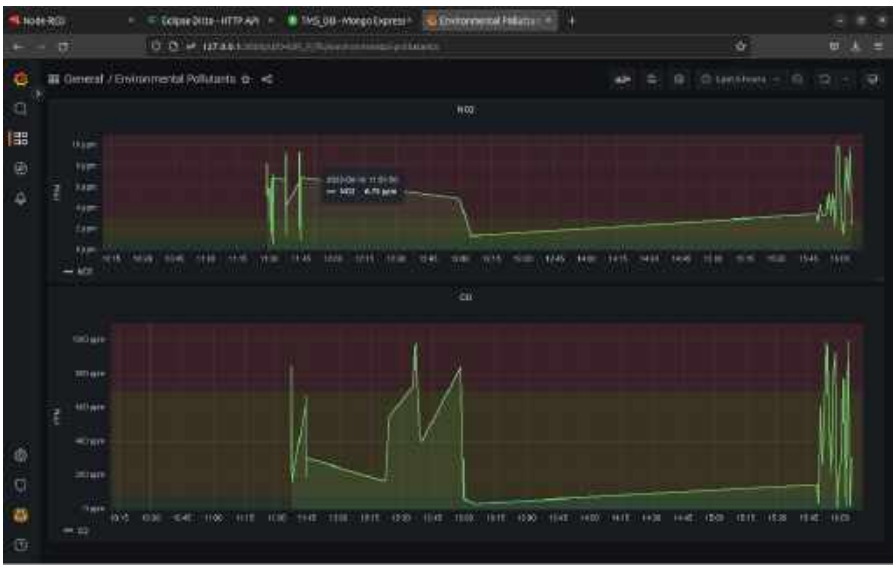


Figure 4.5: *Environmental pollutants Dashboard (example data)*

All the temporal queries are automatically taken by Grafana, so the user can access important information without any line of code. For instance, if some user wants to know either what was the amount of CO over the last hour or what was the average temperature for all days of the previous year, it has only to use a simple drop-down located inside the dashboard.

4.3.4 Controller and Dispatcher

The whole backend logic of the system has been implemented using NodeRED (see Section 3.2.2). The latter acts as controller and information dispatcher inside the system

architecture. In fact it allows the wiring between hardware devices (i.e. ESP32 and Coral Dev board) and software modules, as well as, the managing of the IFTTT (If This Then That) rules, thresholds and conditions. As it mentioned in its own section, Node-RED provides a browser-based flow editor that makes it easy to connect together flows using the wide range of nodes in the palette.

In this TMS several nodes, both internal and external, have been used, such as:

- communication nodes, for *(i)* MQTT,and *(ii)* REST messages;
- data persistency nodes, for storing data both in InfluxDB and MongoDB;
- function nodes, for implementing the system's logic;
- other nodes, for helping guarantee the right flows' operation.

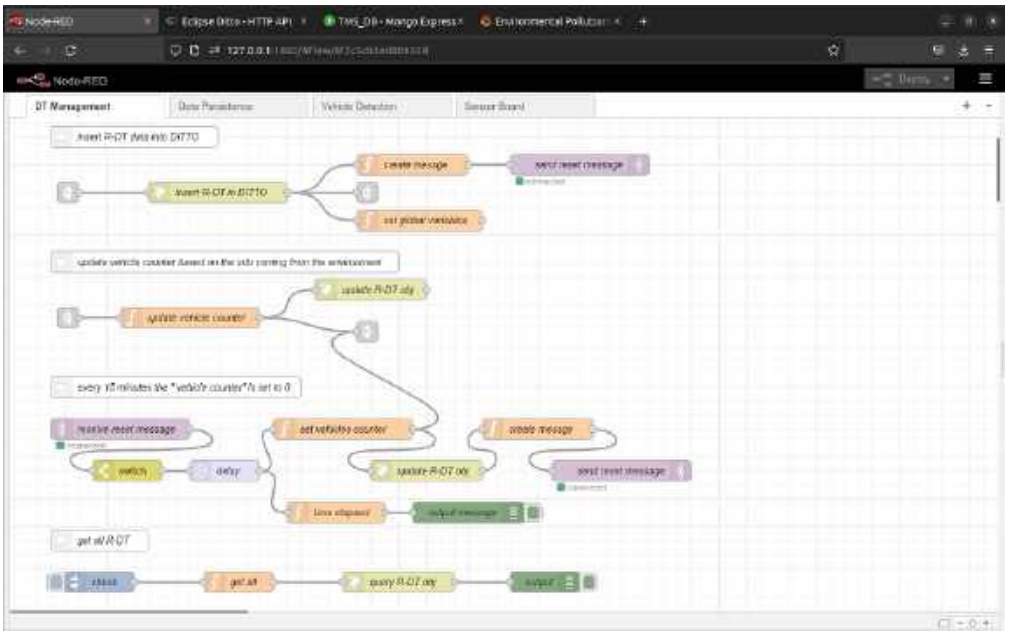


Figure 4.6: *Partial overview of "DT Management" flow*

The whole logic is implemented in four flows, each flow has a precise task:

- *DT Management flow* - deals with the creation, updating and therefore the management of all DTs, communicating with Ditto server through REST API.
- *Data Persistence flow* - deals with storing and updating the R-DTs records, communicating directly with TMS_DB (i.e. the MongoDB database).
- *Vehicle Detection flow* - performs the lunch of the scripts and the reception of messages concerning vehicles detection and counting.
- *Sensor Board* - receives telemetries coming both from the ESP32 board and the Google Coral Dev board (through MQTT broker), stores them in the InfluxDB database, and forwards values to other flows.

All the flows communicate with each other using the *link* in and *link out* nodes, which creates virtual wires to enable cross-flow-communications. Figure 4.6 shows a partial overview of the DT Management Flow.

4.4 Models' Training

There are several ways of generating TFLite models. One can either use a pre-trained model from the TensorFlow Hub [55] or train its own model using TensorFlow Lite Model Maker [56], a tool which helps to train a tensorflow model with a custom dataset exploiting transfer learning. In order to provide an accurate analysis, in this thesis both pre-trained and re-trained models have been used.

4.4.1 Pretrained Models

Instead of building neural networks from scratch, Google Coral provides a set of pre-trained models for object detection, i.e. architecture which have been already trained on a huge dataset, able to solve the object detection tasks. Most of the pre-trained models have been trained using the COCO (Common Object in Context) dataset [57], which contains 90 classes of objects. Using such models allows programmers to avoid models' training, and therefore to save time and GPU power. As a matter of fact, most of the object detection tasks can be solved by using pre-trained models, since they are able to detect the most common objects. The most suited architecture for either mobile or edge devices is *MobileNet*. The latter uses depth wise separable convolutions instead of classic used in standard networks, featuring also a good trade off between latency and accuracy. Coral suggests two models based on two variants of *MobileNet*, i.e. *MobileNetV1* and *MobileNetV2*. Both of the two models allow to perform very fast inferences and are suited for edge devices. The two models have been trained on COCO dataset and are able to run also on one or more TPUs, enhancing performances.

Another model strongly suggested by Google Coral is *SSDLite MobileDet*, which is a model trained on COCO datasets but with a different architecture, i.e. *MobileDet* [58]. The latter allows better performance in terms of accuracy rather than *MobileNet* architecture, but not in terms of latency. In fact, models trained using a *MobileDet* architecture, are on average slower than the ones trained on a MobileNet architecture.

The drawback of using pre-trained models is that they are able to detect a finite set of objects, if one needs to extend the model, using them is not the right solution. As future work (see Chapter 6), the idea is to improve the vehicle detection model by adding several classes which represent the exact brand of the vehicle. For these purposes, the pre-trained models are not suited for the proposed system but they can be used both as a start point to compare models' performances and as a base for transfer learning to retrain custom models with a custom imageset.

4.4.2 Vehicles Datasets

To train models, a huge dataset containing images about vehicles was needed. It was challenging to find one which guaranteed to detect vehicles even when the camera video is far from the road. In fact, by taking into account only zoomed images of vehicles, the issue could be that if the camera is placed far from the roam (such as the normal traffic cameras), the model fails in detecting the right class, even if it has a good accuracy.

Hence, for the purpose described, two datasets have been used to train the model:

- mini traffic detection (MTD) dataset [59], initially able to detect 8 classes of object (Bicycle, Bus, Car, Motorcycle, Person, Traffic light, Truck, Stop sign), then reduced to 5 (Bicycle, Bus, Car, Motorcycle, Truck) since the focus was on vehicles classes only;
- traffic image (TI) dataset [60], able to detect 4 classes of vehicles (Bus, Car, Motorcycle, Truck).

The second one contains labeled images about vehicles taken from traffic cameras, i.e. suited for training a model able to perform correct detections even if the camera is not too close from the road.

4.4.3 Retrained Object Detection Models

Thanks to the TFLite Model Maker library, several models have been trained using the two custom datasets described in the previous subsection. The technique used for simplifying the building model process is Transfer Learning, so as to reduce both the amount of training data and the training time required. The object detection architecture chosen is the *EfficientDet-Lite[0-4]* family, which directly comes from the *EfficientDet* architecture [61]. Figure 4.7 shows the performances in terms of latency and mAP of the aforementioned architectures.

The train process was organized in several steps:

1. Prepare the custom dataset - by retrieving data and split them in train, validation and test set.
2. Choose the object detection architecture - one of the *EfficientDet-Lite[0-4]* family.
3. Define hyperparameters for training the model - such as epochs and batch size.
4. Train the TensorFlow model - using train data, defining hyperparameters.
5. Evaluate the output model using test data
6. Export the TFLite mode - i.e. the post.trained post-training quantized model (*.tflite* format).
7. Evaluate the TFLite model.
8. Test *.tfilite* output.

During the training, hyperparameters have been modified in order to fine tune the object detection models. See the section Usage and Results to see the models performances achieved.

Model architecture	Desktop CPU*	Desktop CPU* + USB Accelerator (USB 3.0) with Edge TPU	Embedded CPU **	Dev Board † with Edge TPU
MobileNet v1	47 ms	2.2 ms	179 ms	2.2 ms
MobileNet v2	45 ms	2.3 ms	150 ms	2.5 ms
Inception v1	92 ms	3.6 ms	406 ms	3.9 ms
Inception v4	792 ms	100 ms	3,463 ms	100 ms

* Desktop CPU: 64-bit Intel(R) Xeon(R) E5-1650 v4 @ 3.60GHz
** Embedded CPU: Quad-core Cortex-A53 @ 1.5GHz
† Dev Board: Quad-core Cortex-A53 @ 1.5GHz + Edge TPU

Figure 4.7: *Comparison between the different EfficientDet-Lite architectures, source [62]*

4.4.4 Compiling for Edge TPU

Once trained and converted models to a TensorFlow Lite, i.e. after achieving quantized *EfficentDet Lite* models, they need to be compiled for the Edge TPU to be deployed on it. Basically, when a model is compiled for Edge TPU, it means that all the sets of operations necessary to execute it are translated in order to run on the TPU. Eventually, all the supported operations will run on the TPU, while the unsupported ones will still run on the embedded CPU. Hence, by not compiling the *.tflite* for the TPU, all the operations will be executed on the CPU and therefore there will be no gain in terms of latency-per-inference. This operation is executed by the Edge TPU Compiler, a command line tool that can be installed directly on Coral's. It receives a TensorflowLite model as input and produces a model compiled for EdgeTPU as output. Figure 4.8 shows the process above mentioned in action.

Since Coral's Edge TPU has a SRAM (Static RAM) of 8MB, a model which uses more than this amount of memory will badly affect the overall execution's throughput. In order to avoid that, the model can be splitted in several segments and executed as one thanks to the pipelining technique provided by Google Coral. To do that, the Coral's documentation strongly recommended using more than one Edge TPU.

4.5 Implementation

A set of the topics addressed by the research have been implemented in this Thesis. The current implementation consists in a system which creates and manages the DT of the road in which the hardware devices are placed (R-DT), using detections coming

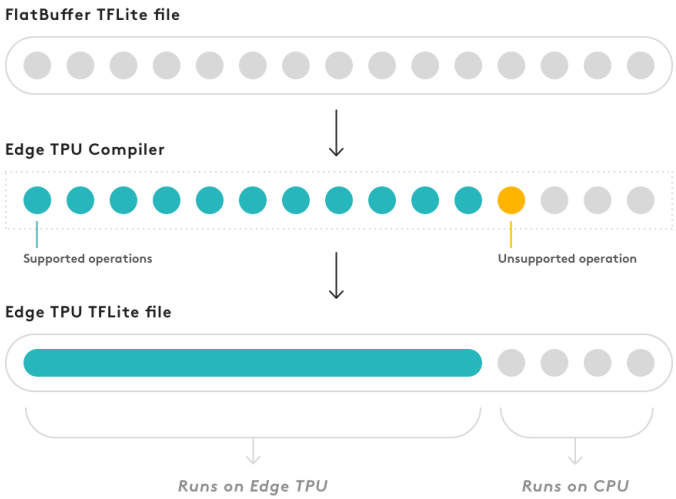


Figure 4.8: *The Edge TPU compiler in action [63]*

from the video stream and telemetries coming from both the Gas Sensor Board and the Environmental Sensor Board. The AI algorithms developed are able to both detect vehicles' classes (Bus, Car, Motorcycle and Truck) and count them as soon as they are detected. Information concerning real time inference and the current counter are then forwarded to the fog node which runs NodeRED. The latter collects the information and uses it to maintain the R-DT (or even multiple R-DTs) stored on Eclipse DITTO.

The architecture chosen for deploying services is hybrid-fog, in which modules are placed on several nodes spread all over the network:

- on the edge layer are placed all the edge devices which features data gathering and real-time inference;
- on the fog layer are placed the MQTT Broker and Node-RED service;
- on the cloud layer are placed all servers for Digital Twin managing, data persistence and data visualization.

The classic cloud-centric approach lends itself poorly to low throughput high reliability connections. Using a fog architecture allows to move computation all over the network, *(i)* reducing communication latency and bandwidth consumption; *(ii)* improving the overall speed of the system. The proposed system will analyze in the future a huge amount of data (telemetries and video stream), hence to provide an architecture able to optimize data processing and displaying is crucial for the final system deployment.

Figure 4.9 shows an overview of the current system's implementation using a hybrid-fog architecture.

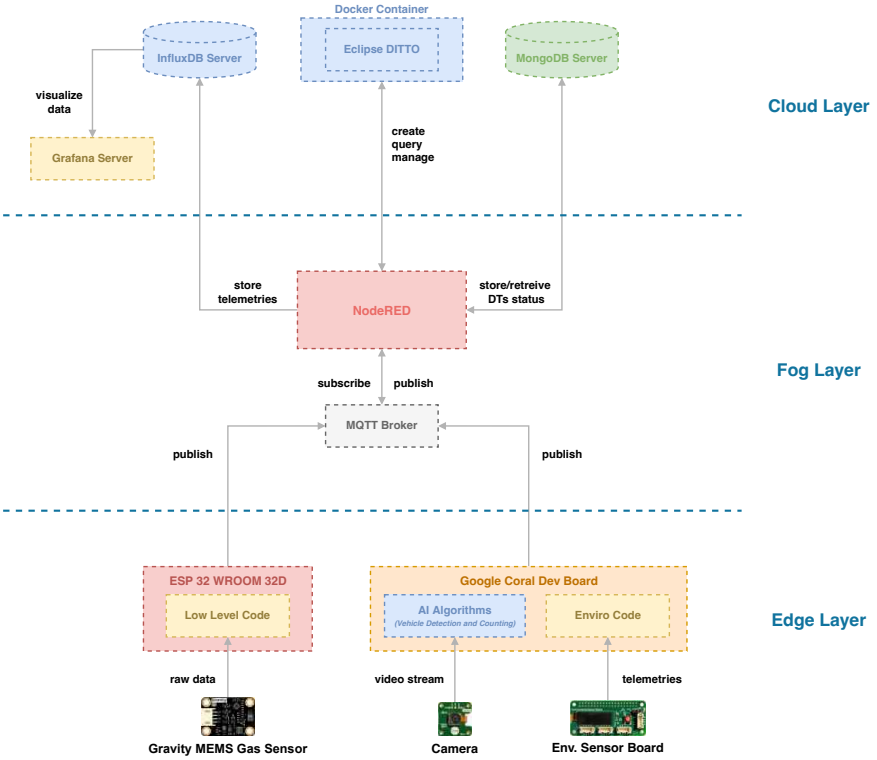


Figure 4.9: *Software overview of the proposed system's implementation*

4.5.1 MQTT Topic Definition

As mentioned in the previous sections, the MQTT resource addressing takes place using topics. In this system, each topic (concerning telemetries) is defined using OMA-LwM2M [64] semantics. Lightweight Machine to Machine is a protocol developed by the Open Mobile Alliance for remote device management in the Internet of Things and other Machine-to-Machine applications. The standard defines that a client has one or more Object Instances, where an Object is a collection of Resources. A Resource is an atomic piece of information that can be read, written or executed. Object and Resources can have multiple instances and are identified by a 16-bit integer while Instances are identified by an 8-bit integer. Objects and Resources are accessed with simple unique URIs: */objId/instId/resId/*. The unique identifiers of the resources are defined through the use of a semantic standard reported in the OMA Registry.

In this implementation it is possible to distinguish:

- **topic**, resource identifier (e.g. /deviceID/objId/instId/resId/) shown in the Table 4.1;
- **full topic**, consisting of prefix and topic.

FullTopics use, in addition to the topic, a prefix. Prefixes are used to avoid the possibility

of creating any loops between MQTT topics. Within this implementation, three distinct prefixes will be used:

- **cmd**, prefix for either issuing commands or ask the state;
- **stat**, prefix to report either the status or the configuration message;
- **val**, prefix to report telemetry values at specified ranges

Device (ID)	Object (ID)	Inst. (ID)	Resource (ID)	Topic
ESP32 (B01)	CO (3300)	0 (0)	Value (5700)	/B01/3300/0/5700
ESP32 (B01)	NO_2 (3300)	1 (1)	Value (5700)	/B01/3300/1/5700
ESP32 (B01)	Temp.(3303)	0 (0)	Value (5700)	/B01/3303/0/5700
ESP32 (B01)	Hum. (3304)	0 (0)	Value (5700)	/B01/3304/0/5700
ESP32 (B01)	Illum. (3301)	0 (0)	Value (5700)	/B01/3301/0/5700
ESP32 (B01)	Press. (3315)	0 (0)	Value (5700)	/B01/3315/0/5700

Table 4.1: *MQTT telemetries' topic definition*

In the Table 4.2, are show the topics used to send control and configuration messages, as well as messages concerning vehicles detected.

Topic	Description
tms/script/vehicles	With this topic, messages related to the vehicles type and their current count are published
tms/nodered/counter/reset	With this topic, the message to reset the counter is published

Table 4.2: *MQTT topic definition for control and configuration messages*

4.5.2 R-DT Definition

In order to define the structure of the R-DT, the official guidelines provided by Eclipse Ditto have been followed [65]. Each DT must have *(i)* a **thingsId**, which identifies uniquely the thing; *(ii)* **policyId**, which links to the policy file containing thing's authorization; and *(iii)* a **definition**, which links the thing to a specific model (i.e. a already defined structure of features and capabilities). Then, each DT has *(iv)* **attributes**, which provide more detail about the thing, and *(v)* **features**, used to manage all data about that single thing. To follow, the description of all R-DT's attributes and features:

- Attributes:
  - **covered area**, GeoJSON Object [66] characterizing the area covered by the device;
  - **road typology**, e.g. local, highway, city street, pedestrian street, etc.;

  - **number of lanes**, number of lanes divided per running direction;
  - **obstacles**, presence of traffic islands, roundabouts, etc.;

- Features:
  - **road**:
    - **viability status**, info about traffic congestion and viability status for drivers;
    - **road condition**, info about the road condition (good status, work in progress, etc.);
    - **warnings**, info about road warnings, e.g. road accidents, road closed, marketplace, etc.;
    - , **vehicles counter**, number of vehicles detected in that road per time unit, e.g. in the last 15 minutes;
    - **current vehicles**, list of vehicles (ID) currently present in the environment per time unit, e.g. in the last 15 minutes.
  - **environment**:
    - **temperature**, value coming from the DT temperature sensor;
    - **humidity**, value coming from the DT humidity sensor;
    - **barometric**, value coming from the DT barometric sensor;
    - **illuminance**, value coming from the DT illuminance sensor;
    - **CO_2**, value coming from the DT $C0_2$ sensor (global);
    - **CO**, value coming from the DT CO sensor (global → pollutant produced by vehicles);
    - **NO_2**, value coming from the DT NO_2 sensor (global → pollutant produced by vehicles);

Finally, in Ditto, the described structure is represented using a JSON object for each thing, the Listing 4.1 shows an example of R-DT.

```
{
    "thingId": "my.iot.suite:my-road",
    "policyId": "my.iot.suite:my-road-policy",
    "definition": "my.iot.suite:roadDevice:1.0.0",
    "attributes": {
        "coveredArea": { ... },
        "roadTypology" : "",
        "lanes": "",
        "obstacles": ""
    },
    "features": {
        "road":{
            "propetries": {
                "viabilityStatus" : "",
                "roadCondition" : "",
                "warnings": "",
                "vehiclesCounter": 50,
```

```
                "currentVehicles": [
                ]
            }
        },
        "environment": {
            "properties": {
                "temperature": 0,
                "humidity": 0,
                "barometricPressure": 0,
                "illuminace": 0,
                "co2": 0,
                "co": 0,
                "no2": 0
            }
        }
    }
}
```

Listing 4.1: *Example of R-DT*

4.5.3 Create and Restore DTs

As soon as the system is started up, it checks the last status of the R-DT. In fact, if Eclipse Ditto service stops working, all the info concerning DTs will get lost. To overcome this issue, the system stores the R-DT status (by updating single fields) every time a DT's change is detected. Records are stored into the *TMS_DB*, precisely in the collection *road_dt*, and are identified by a unique ID (default by mongo) and the *things_id* which identifies DTs.

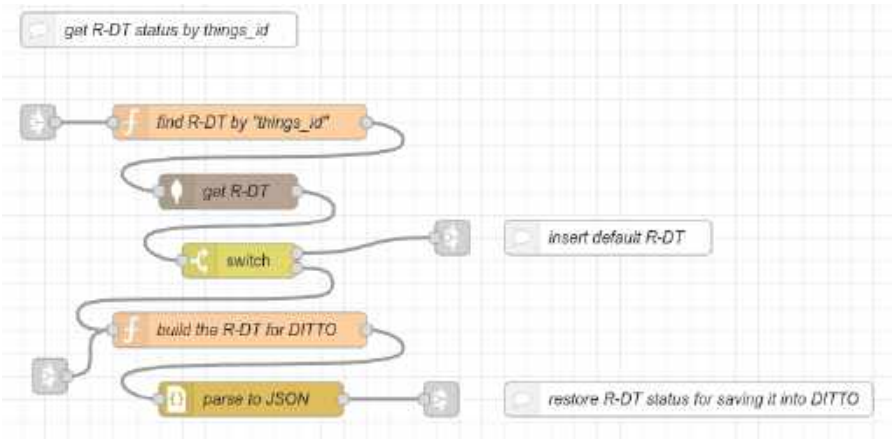


Figure 4.10: *Create/Restore R-DT.*

When the system starts, it checks if a record exists by the specified *things_id*. If yes, the DT object is restored, built and inserted into the Ditto server for applying the digital twin pattern. Otherwise, the default R-DT is created and inserted into Ditto with that *things_id*. Figure 4.11 shows the sequence diagram of the process described.

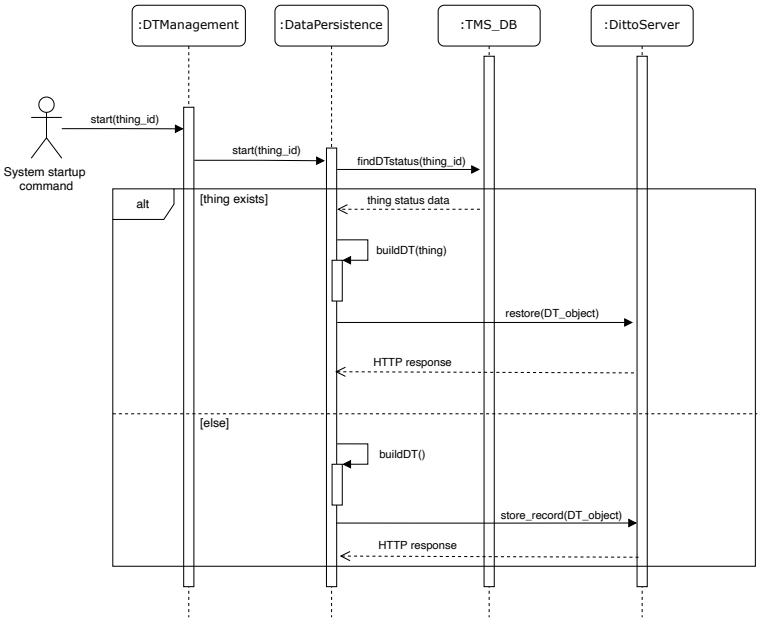


Figure 4.11: *Sequence diagram about system's startup*

4.5.4 Vehicles Detection and Counting

The python algorithms containing the AI scripts can be launched independently from the other modules through a command from the Google Coral Dev board. *VideoDetector* is the python object in charge of performing the object detection and counting task. It loads the model, captures the video, produces results and sends them to NodeRED by publishing messages on the MQTT broker.

The model's inputs for performing vehicle detection are images coming from either real-time or recorded video frames. In fact, *VideoDetector* is also able to manage video streams thanks to the opencv python library. Once captured the stream, on each frame an inference is performed. The names of the classes detected by the model, along with their probabilities values, are drowned on the frames and shown for displaying the inference's result.

One of the issues faced was how to keep track of vehicles passing by the camera. To do that, *VideoDetector* object uses *EucleudianDistanceTraker*. The operation of the latter is based on the euclidean-distance concept to keep track of objects in video frames. Iteratively, It determines whether there is a difference between two center points of an object in the current frame and the previous frame. If the result is less than a certain threshold, then the two objects are the same but in different frames. In this way the tracker is able to maintain a dictionary with the positions of each detected object. Finally, that dictionary is used by the *VideoDetector* to count vehicles. Basically, a virtual line with static x and y values is drawn on each frame. For each object in the tracker dictionary the center is calculated. If the center point crosses the virtual line,

then the counter of that specific class of vehicles is incremented. Info concerning class and current count of vehicles detected are sent to the backend logic through the MQTT broker. Figure 4.12 shows the activity diagram of the described code.

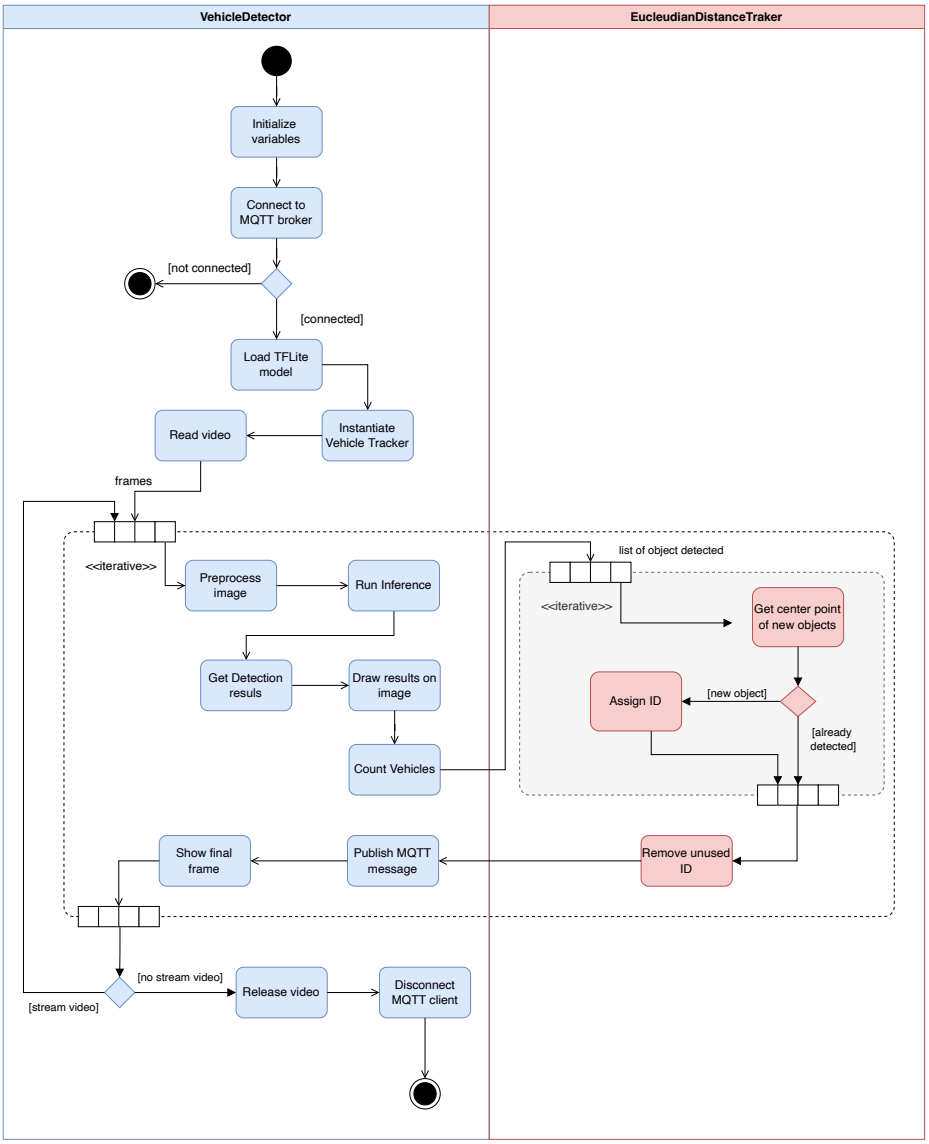


Figure 4.12: *Activity diagram of the vehicle detection and counting code*

4.5.5 Pollution Data Gathering and Displaying

While vehicles are detected and counted, the MEMS Gas sensor gathers data concerning CO and NO_2. After three minutes of necessary calibration time, the sensor sends raw

data to the ESP32. The latter is connected to the internet and, after processing telemetries, publishes values on the specific topics (which are different for both pollutants, see Table 4.1). The MQTT broker forwards data to the subscribers (NodeRED nodes). Figure 18 shows the sequence diagram of the described interaction.

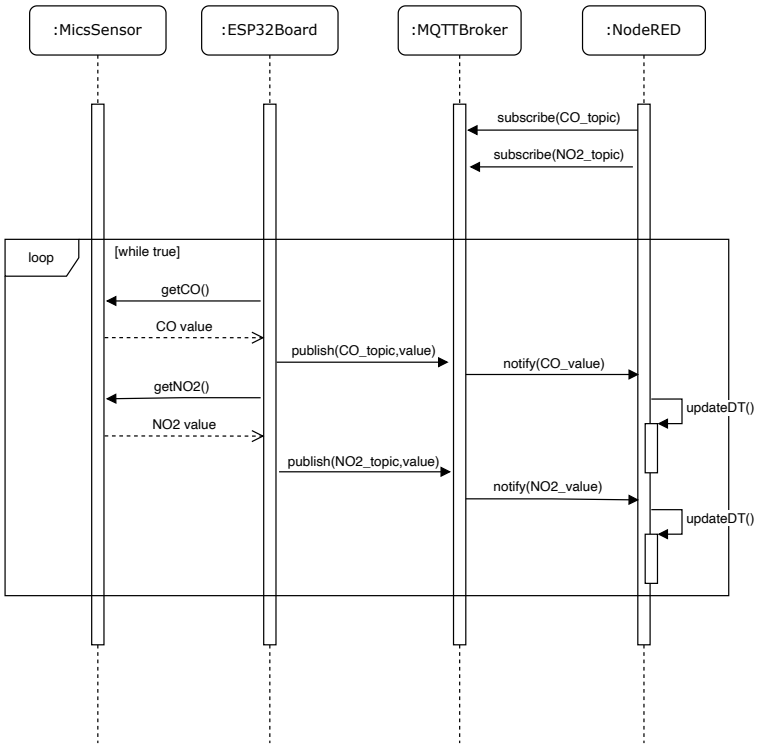


Figure 4.13: *Sequence diagram of the interaction between Node-RED, ESP32 and the Gas sensor board*

Once reached the backend logic, telemetries are directly stored into the respective InfluxDB measurements. Here, data is withdrawn by Grafana service and displayed on *Pollutants Monitoring* dashboard for further analysis. Figure 4.5 shows an example of how the data view appears.

4.5.6 Environmental Data Gathering

As regards environmental data, Coral Development Board runs a script which periodically gathers telemetries and forward them to NodeRED using MQTT messages. Just like the Gas sensor board, data is acquired and sent on specific topics to the MQTT broker. The Listing 4.2 shows the main of the python code executed by the board:

```
1 def main():
2     # Create instances of EnviroKit and Cloud IoT.
3     enviro = EnviroBoard()
```

```
    sensors = {}
    # Connect to the broker
    setup_connection()
    # Topic definition
    topic_temp = '/B01/3303/0/5700'
    topic_hum = '/B01/3304/0/5700'
    topic_light = '/B01/3301/0/5700'
    topic_press = '/B01/3315/0/5700'
    # Start the never ending loop
    while True:
        # First display temperature and RH.
        sensors['temperature'] = enviro.temperature
        sensors['humidity'] = enviro.humidity
        msg = '%.2f' % _none_to_nan(sensors['temperature'])
        send_message(msg,topic_temp)
        msg += '%.2f' % _none_to_nan(sensors['humidity'])
        send_message(msg,topic_hum)
        sleep(5)
        # After 5 seconds, switch to light and pressure.
        sensors['ambient_light'] = enviro.ambient_light
        sensors['pressure'] = enviro.pressure
        msg = '%.2f' % _none_to_nan(sensors['ambient_light'])
        send_message(msg,topic_light)
        msg = '%.2f' % _none_to_nan(sensors['pressure'])
        send_message(msg,topic_press)
        sleep(5)
```

Listing 4.2: *Python code executed by the Environmental Sensor Board*

First of all the *EnviroBoard()* is instantiated, so as to initialize the Google Environmental Board for gathering telemetries. After that, the connection with the MQTT broker is started followed by the topic definition (one for each telemetry, see Table 4.1). Then, the never-ending loop starts. Inside the while loop, telemetries are captured two by two (as suggested by the documentation) every 10 seconds: *(i)* firstly Temperature and Humidity; *(ii)* then Illuminance and Pressure. As soon as data have been gathered from the field, are sent to the MQTT broker and therefore to NodeRED which will dispatch them to proper InfluxDB measurements.

4.5.7 R-DT Status Update

When the global vehicle counter is incremented or new telemetries come from the sensor, the R-DT needs to be updated. The R-DT on DITTO server, is continuously updated with the new values through RESTful messages, while an update query is performed on the MongoDB instance for storing its current status. Moreover, as regards telemetries, as soon as the MQTT messages reach the subscribers, the new value is stored on the right InfluxDB measurement and shown on the Grafana dashboard.

A special case which requires the update of both the DT and its status, is the global counter reset. The R-DT stores the number of vehicles which pass by the road of reference in a given amount of time, i. e. 15 minutes. Every quarter of an hour the R-DT global vehicles counter is resetted, i.e. it is reduced to zero. As shown in Figure

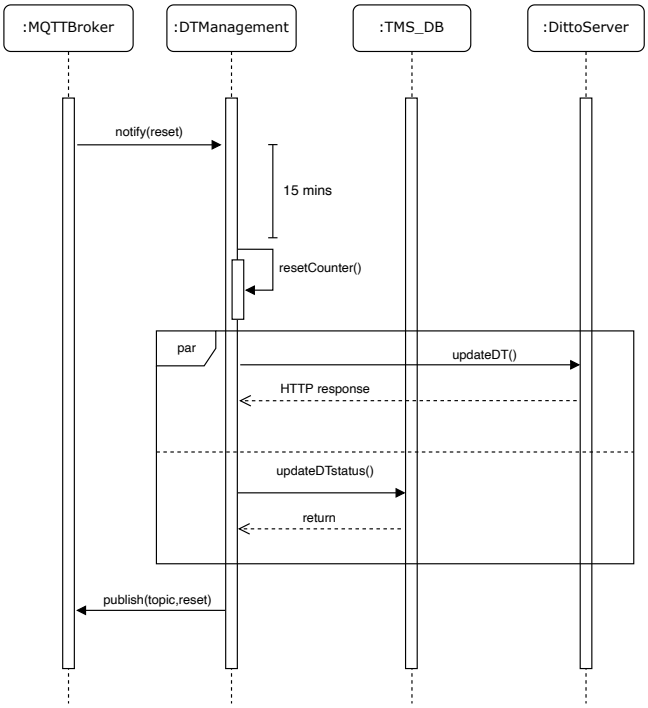


Figure 4.14: *Object interaction for resetting the global vehicle counter in R-DT*

4.14, the DTManagement flow publishes on a specific topic the message “reset”. As soon as the message is received, a 15 minute counter is started. When it expires, both the MongoDB record and the Ditto instance are updated.

Chapter 5

Simulation and Results

5.1 Edge AI: performances and benchmarks

One of the most important system requirements is to perform fast complex AI algorithms at the edge of the network, i.e. as close as possible to the physical object (road). According to the Google Coral Documentation [67], an individual Edge TPU is able to perform 4 trillion operations (tera-operations) per second (TOPS). Of course, each neural network has its own architecture, which can influence the overall performances. In order to point out the added value produced by the Edge TPU, several models with different architectures have been tested using the hardware accelerator (TPU) along with the CPU (on which are executed the TPU's unsupported operation).

Moreover, exploiting Edge AI allows it to process data locally, i.e. without sending them to the cloud. This approach has brought several benefits in terms of latency and bandwidth consumption as well as in terms of energy dissipation. All these aspects are crucial in constrained devices such as the Coral Dev Board. One of the goals of this research is also to show how those benefits could positively affect an IoT system.

The results of all the simulations taken are shown in the following sections along with an analysis about the gain that is possible to have in terms of latency, bandwidth and energy consumption by performing AI models at the edge of the system's architecture.

5.1.1 Accuracy and Latency

In order to provide an accurate analysis, each model has been tested on the script presented in the previous section 4.5.4 and therefore with the same video stream. In the next paragraphs two terms are used to measure models' performances: *mean average precision* (*mAP*) and latency. The *mAP* is the primary metric according to COCO evaluation metrics [57] and is based on *(i)* Confusion Matrix; *(ii)* Intersection over Union (IoU); *(iii)* Recall; and *(iv)* Precision. Each model is assigned its own *mAP* which represents the accuracy score averaged over all detection's categories. The term latency instead, refers to the time required for just one inference: the higher this value, the slower is the overall inference applied by the model.

The testing session pointed out that models' accuracy and latency are inversely proportional: if the accuracy increases, the latency decreases and vice-versa. Instead, accuracy is directly proportional to the model size, i.e. the amount of memory used by the quantized model. By increasing the accuracy, the memory size increases as well and the more is the model size, the lower would be the latency.

The pre-trained models perform very fast both with real-time and recorded videos. In fact, none of the three exceed 30ms per frame as inference time. The fastest one, according to the tests taken, is *SSD MobileNet V1*, which has a latency of 12.6ms. On the other hand, whatever they earn in latency, they lose it in terms of accuracy. In fact, all the three models didn't perform very well in detecting classes, sometimes they weren't able to detect the object even if it was on the frame. Among the three models, the one that represents the better trade-off between accuracy and latency is *SSDLite MobileDet*. Moreover, these models have been already trained on a dataset (COCO) and they are only able to detect 90 classes. An object which doesn't belong to this set of classes will not be detected. As future work (Chapter 6), the idea is to improve the vehicle detection model by adding several classes which represent the exact brand of the vehicle. For these purposes, the pre-trained models are not suited for this system but they can be used as a base for transfer learning to retrain custom models with a given imageset.

With respect to the previous models, the retrained ones present higher precisions. In fact they are able to detect the right vehicle class (on the average). As it has been pointed out in the introduction of this section, high accuracy means latency loss. As a matter of fact, both the retrained models have almost the same latency (around 70 ms per frame), which more than the double of the lowest pretrained model, i.e. *SSDLite MobileDet*. The neural network with the highest *mAP* between the two of them is the *MTD Model* (85,5%). The latter has been trained using zoomed images of vehicles (*MTD dataset*, Section 4.4.2), therefore is more suited for situations in which the camera is not too far from vehicles. In fact, a bad placement of the camera could lead to detection errors. The other model - *TI Model* - also performs very well. Although the *mAP* is pretty low (67,2%), the network has been trained using traffic images, i.e. images coming from traffic cameras (using the *IT dataset*, Section 4.4.2). This makes the *TI Model* very suited in those situations in which the camera is very far from the road (e..g the actual traffic camera placement). Figure 5.1 sums up, plotted on a graph, the benchmarks of both pre-trained and re-trained models in term of latency and accuracy

5.1.2 Edge TPU Performances

The same models tested on Google Coral Dev Board using the Edge TPU, have been executed also using different hardware configurations, such as the embedded Dev Board CPU, i.e. NXP i.MX 8M SoC (quad Cortex-A53, Cortex-M4F). Moreover, thanks to preliminary simulations, it was possible to execute all AI models on a very performing Desktop CPU (Intel® Core™ i7-6820HQ, 2.70GHz x 8) and eventually compare them with the outputs of the configurations mentioned above.

The purpose of this experiment was to understand the great added value of using a

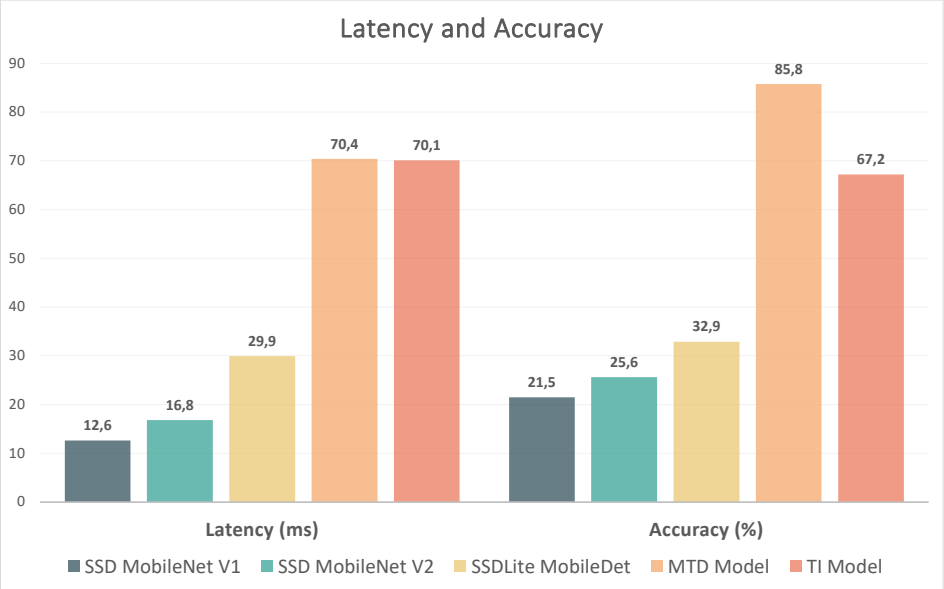


Figure 5.1: *Benchmark in terms of Latency and Accuracy of both pre-trained and re-trained models*

Tensor Processing Unit at the edge. Even if we lose some accuracy, saving computation time is an important feature for constrained devices. In fact, latency is directly proportional to the power consumption: the slower the model the more is the power dissipated by the device. To execute complex ML models requires a certain amount of energy which is not always available. Note that, for the sake of comparison, all models running on CPUs (Desktop and Embedded) and Edge TPU are the TensorFlow Lite versions and all of them have been tested using the same video stream. Figure 5.2 below shows the comparison between the inference time per frame achieved by using the different hardware configurations.

The results achieved are amazing. The execution of the pretrained models using the Edge TPU is way faster than executing the neural networks on the Embedded CPU: from being almost 17 times faster in the case of the *SSDLite MobileDet* model till being 30 times faster for the *SSD MobileNet V1* model. The differences decrease when comparing the latencies of the Edge TPU with the ones obtained using the Desktop CPU, but still the ML accelerator performs better: from 3 to 6 times faster, depending on the model used. As regards the retrained models, using the Edge TPU is still the better solution in terms of performances: it is 6 times faster than the Embedded CPU and 1,6 times faster than the Desktop CPU. Hence, the use of the Edge TPU as a ML accelerator could seriously improve the whole system performances in terms of latencies (i.e. inference per frame) but also of power consumption and memory saving (by reducing the model size).

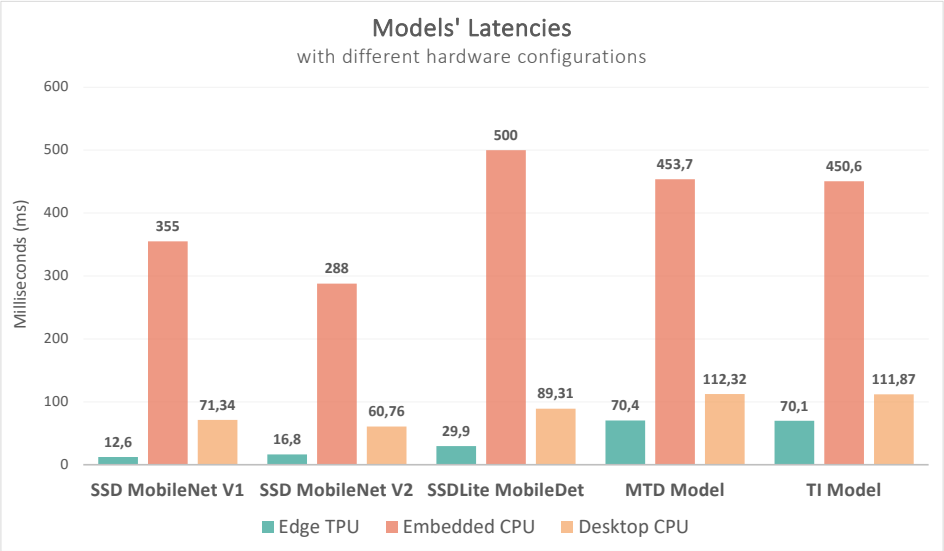


Figure 5.2: *Time per inference among different hardware configurations*

5.1.3 Low Latency and Bandwidth Consumption

Using a cloud-centric approach for computer vision tasks (such as the vehicle detection one), the camera module placed at the edge needs to capture frames from the real-time video stream, and then send them one by one to the cloud. Here, the AI modules take them as input and perform inference. This process lends to two important drawbacks:

- latency in field to cloud (and back) is not acceptable;
- real-time streaming requires high bandwidth.

Exploiting Edge AI allows to perform all these operations locally, avoiding high latencies and saving communication bandwidth since data don't need to be transmitted.

In order to show practically this concept, it is possible to calculate the amount of bandwidth that we need to send each frame to the cloud. The total upload time is the output of the following formula:

$$Total\ Upload\ Time = \frac{File\ Size\ (Mb)}{Upload\ Speed\ (Mbps)} \tag{5.1}$$

The *Upload Speed* is nothing but the bandwidth, which expresses the amount of bits that can be transmitted per second. Usually it is calculated in *Mbps* (Megabit per second). The *File Size* is the size of the frame to send and it is expressed in *Megabits* ($Mb = Megabytes\ (MB)\ *\ 8\ bits$). By setting the latency that one wants to achieve in data transmission, i.e. (*Total Upload Time*), and by using the inverse formula of 5.1,

one can calculate the bandwidth needed, i.e. the *Upload Speed*:

$$Upload\ Speed\ (Mbps) = \frac{File\ Size\ (Mb)}{Total\ Upload\ Time\ (s)} \tag{5.2}$$

The video stream used for this simulation is the same used for the inference's latency and accuracy analysis of the previous subsection (see Subsection 5.1.1). Each frame captured by the camera module, and therefore by the python algorithm, has an approximate file size of 242 KB (Kilobytes), i.e. 0,242 MB and therefore 1,936 Mb. To keep the latency below 100 ms per frame, we need approximately a bandwidth of 19,36 Mbps. Note that this value refers to just one transmission.

Using the frame rate of the real-time video stream, it is possible to calculate the total amount of frames the edge module sends to the cloud. Let's take into account just one-day interaction between edge device and cloud server. For instance, using a 20fps (frame per second) bitrate, the camera module is able to capture 72.000 frames in just one hour. By multiplying this number for 24 hours (one day interaction), the edge module has to send 1.729.000 frames eventually. Since the transmission of just one frame in less than 100ms requires a bandwidth of 19,36 Mbps, the total bandwidth used for field to cloud communication would be 33.454.080 Mbps. This value is huge taking into account that it significantly affects the power dissipation of the edge module. Moreover, if one wants to achieve lower transmission latencies, it has to increase the bandwidth as well since they are inversely proportional, i.e. achieving increasingly higher values. Figure 5.3 shows how the bandwidth required for transmitting just one frame increases as soon as the communication latency decreases.

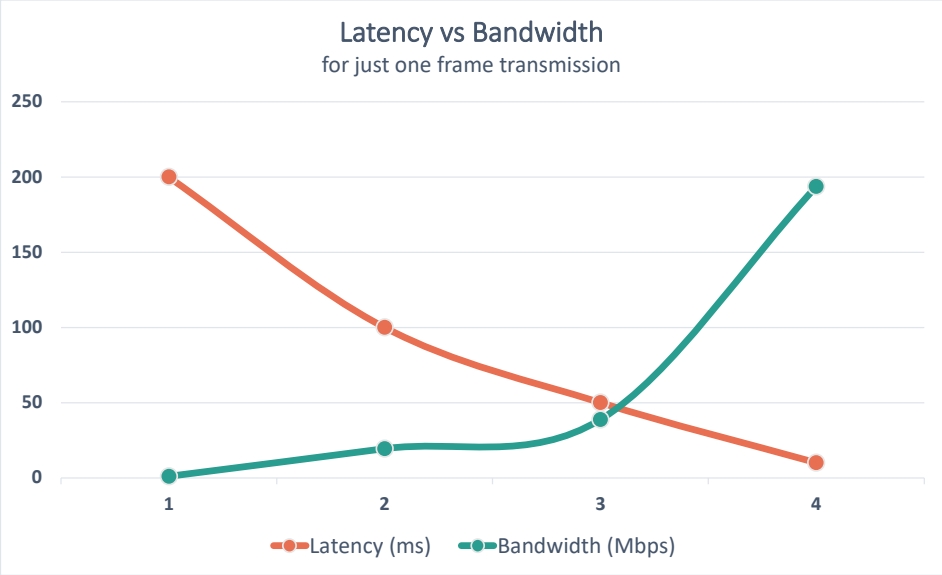


Figure 5.3: *Line graph that shows how latency changes reflects to bandwidth*

By exploiting AI at the edge, frames are processed locally, only results are sent to

either fog nodes or cloud servers. In this system, when a vehicle is detected for the first time, an MQTT message is published by the Coral Dev Board on a specific topic. The length of the payload could vary, depending on the vehicles' class detected (the string "car" has only 3 characters, i.e. 3 bytes, while the string "motorcycle" has 10 of them, i.e. 10 bytes). Moreover, an MQTT packet is made of the following parts:

1. a control header of 1 byte (fixed size);
2. a packet length field which could be between 1 byte and 4 byte, depending on the payload length;
3. a variable length header which contains the topic name;
4. the message payload;
5. 2 fixed bytes at the end of the packet as packet id.

In the worst case, the payload could be 10 bytes (motorcycle) plus other 34 bytes which are part of the string message (e.g. {*"type": "motorcycle", "current_count": "4"*}), the total payload size would be 44 bytes. The topic used is *tms/script/vehicles*, hence the variable length header (number 3 of the list) is 19 bytes. By adding to this amount the control header byte, the packet length field (1 byte) and the final 2 bytes for the packet id, the final MQTT packet size in the worst case is 67 bytes, i.e. approximately 0.000063 MB. Note the huge difference in terms of size that there is between one frame to send (0,242 MB) and one packet to send. Let's say that the mean amount of messages sent per frame is 5 MQTT packets, i.e. 335 bytes, using the Equation 5.2 it is possible to calculate the bandwidth by keeping the latency of 100ms. The value achieved is around 0,026 Mbps, compared to the 19,36 Mbps needed for sending a frame to the cloud in less than 100ms. Table 5.1 highlights the huge difference and gain in terms of bandwidth consumption that is possible to achieve by moving computation at the edge of the infrastructure.

	Edge	**Cloud**
Bandwidth	0,026 Mbps	19,36 Mbps

Table 5.1: *Bandwidth consumption: edge vs cloud*

5.2 Querying R-DT

To build the digital twin of the road means to abstract the physical object into its virtual counterpart by providing synchronous and asynchronous APIs. The latters, can be used to work with the real environment, enabling the *Object as a Service* paradigm. In fact, third party applications can communicate with the road as they do with normal web services, i.e. through request/response messages. In fact, using a set of REST API is possible to gather relevant information about environmental parameters, pollutants, road conditions, traffic viability and so on.

Once the DTs are created, the Eclipse Ditto service provides a group of well documented APIs which can be visualized using OpenAPI Swagger tool (Figure 5.4) and perform the following tasks:

- things (DTs) creation and management;
- things' features structuring;
- things' control access management;
- things' search and filtering;
- things' communications.

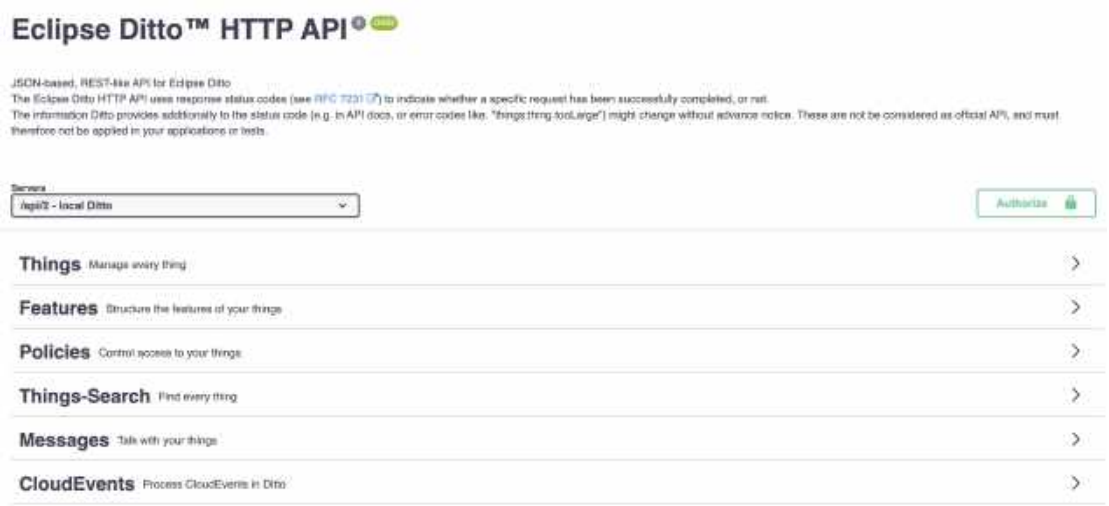


Figure 5.4: *Eclipse Ditto's OpenAPI Swagger tool*

While NodeRED creates, updates and manages the R-DT, it is possible to gather real time data by querying road's features and attributes to the Ditto server. For each information there is a specific path that needs to be composed, along with the HTTP method to perform the already known operations: GET, PUT, POST, DELETE. For instance, by executing a GET to the path *http://ip.of.the.host:8080/api/2/things* one can retrieve a list of all the visible things managed by Ditto server. By using the POST method on the same path a new thing (i.e. a new digital twin) can be created. By using the proper path for the features, it is also possible to get the real time status of that specific thing. Figure 5.5 shows an example of HTTP GET query which returns the list of all the features of the R-DT, i.e. both the road and the environmental real time information.

Of course not all APIs are allowed to be performed by third party applications. For instance, sometimes the read-only mode is preferred in monitoring applications such as this Traffic Monitoring System. To deal with this, Ditto offers API for things' policy management, i.e. help developers to control access to their digital twin. It is possible

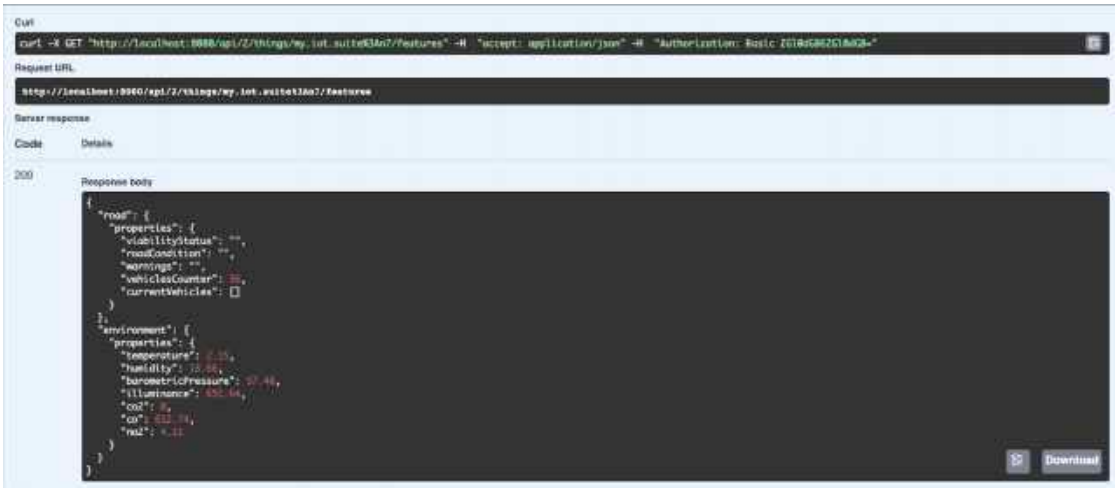


Figure 5.5: *HTTP request for getting the features list of the R-DT*

to specify which API call can be performed on which digital twins by modifying and managing the things' access policies, ensuring security and privacy.

Chapter 6

Conclusion and Future Works

According to the United Nations "The World's Cities in 2018" booklet [23], by 2030 almost 28% of the worldwide population will be concentrated in cities with at least 1 million inhabitants and nearly the 8.8% in the so called megacities, i. e. cities with at least 10 millions citizens. The lower is the urban density, the more is the demand of resources such as energy, electricity and primary goods with a consequent increase of waste that could lead to both social and economic problems. Furthermore, the current climate crisis is pushing researchers to find sustainable solutions and innovative ways to solve both noise and air pollution along with ever-increasing urban traffic congestion [68].

In this context, this work of thesis promoted an innovative approach for showing how general purpose sensing and DTs can synergically describe and digitize large physical environments. The technique chosen for supporting general purpose sensing capabilities enabled the usage of elementary sensors powered with AI algorithms: the virtualization of such sensing devices is the first step toward the development of a DT and hence of a "a device as a service", i.e. to keep a digital version of the physical environment which can respond to queries, reflect the real physical changes, store object's signatures and provide availability along with reliability (just like a normal web service). The potential of the combination of two technologies together is huge and can be applied both in small and large environments. To assess this novel approach, all these theoretical concepts have been implemented in a Traffic Monitoring System. The latter is able to monitor pollutants and environmental parameters as well as to detect and count vehicles in order to provide useful information for traffic congestion and viability.

The results of this Thesis proved that moving AI algorithms at the edge of the network is possible and high performing. During all the tests taken, Google's Edge TPU (equipped on Coral Dev Board) achieved very good performance even by running complex object detection models, with the Edge TPU resulting 10,3x faster than the Embedded CPU and 2,2x faster than the Desktop CPU as well as more efficient in term of energy and bandwidth consumption, since the cloud-centric solution consumes almost 745x times the amount of bandwidth consumed by the edge device. This behavior is due to the local execution of the optimized model size which allows to perform increasingly

faster inferences by still not dissipating a big amount of energy in remote computations.

Towards the final deployment of the presented prototype system, still additional improvement needs to be performed. First of all, the AI algorithms have to be tested on the Edge TPU, i.e. on Google Coral Dev Board. The idea of bringing AI at the edge of the network could improve performances and overcome common issues (such as privacy and delays). In fact, according to the documentation, the times that the Dev Board needs to derive an inference are significantly lower than the ones achieved in the system's test. This could power up the computer vision task, by detecting vehicles' classes in just a few milliseconds.

Another potential enhancement is the improvement of the vehicles detection capabilities, by increasing the number of car classes that the model can detect. Currently the AI model can only detect 4 classes of vehicles (bus, car, motorcycle, truck). The final goal is not only to identify the type but also the specific model of a vehicle (e. g. Citroen C4, Tesla Model X). To add this feature is not trivial since, to provide a good service, models need to be trained with a proper dataset. Moreover, another challenge will be to minimize incorrect inference results. The presence of these errors could be very common since the shape of cars is very similar nowadays.

According to the objectives described in this thesis, one of the next steps will be also to capture and analyze engines' sound coming from vehicles. Since computer vision alone could be not enough for detecting the true vehicle type and model, engines' noise can be used to build the signature that DTs can store. Therefore these information could become inputs for AI models, helping them to uniquely identify objects of interest.

Finally, from an environmental perspective, it is interesting to analyze pollutants emitted when vehicles pass by. The environmental warning is a serious issue and providing granular data about vehicle emission could seriously help to reduce pollution in metropolitan areas. Cities' governance could be aware of the whole environmental situation and take action based on it, by reducing the global pollutants' emission. By focusing on the impurities produced by a single vehicle, it is possible to add them to signatures (already equipped with images and sounds), and therefore to significantly enhance vehicle detection. In this way the system could be able not only to work in favor of the environment, but also to improve itself.

Bibliography

[1] Vito Albino, Umberto Berardi, and Rosa Maria Dangelico. "Smart cities: Definitions, dimensions, performance, and initiatives". In: *Journal of urban technology* 22.1 (2015), pp. 3–21.

[2] Hamid Menouar et al. "UAV-enabled intelligent transportation systems for the smart city: Applications and challenges". In: *IEEE Communications Magazine* 55.3 (2017), pp. 22–28.

[3] Vincent Oldenbroek, Leendert A Verhoef, and Ad JM Van Wijk. "Fuel cell electric vehicle as a power plant: Fully renewable integrated transport and energy system design and analysis for smart city areas". In: *International Journal of Hydrogen Energy* 42.12 (2017), pp. 8166–8196.

[4] Luis Sánchez et al. "Integration of utilities infrastructures in a future internet enabled smart city framework". In: *Sensors* 13.11 (2013), pp. 14438–14465.

[5] Lorena Parra et al. "Development of a conductivity sensor for monitoring groundwater resources to optimize water management in smart city environments". In: *Sensors* 15.9 (2015), pp. 20990–21015.

[6] Alexey Medvedev et al. "Waste management as an IoT-enabled service in smart cities". In: *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*. Springer, 2015, pp. 104–115.

[7] Gierad Laput, Yang Zhang, and Chris Harrison. "Synthetic sensors: Towards general-purpose sensing". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 2017, pp. 3986–3999.

[8] Roberto Minerva, Gyu Myoung Lee, and Noel Crespi. "Digital twin in the IoT context: a survey on technical features, scenarios, and architectural models". In: *Proceedings of the IEEE* 108.10 (2020), pp. 1785–1824.

[9] Christos Koulamas and Athanasios Kalogeras. "Cyber-physical systems and digital twins in the industrial internet of things [cyber-physical systems]". In: *Computer* 51.11 (2018), pp. 95–98.

[10] Alina Trifan, Maryse Oliveira, José Luís Oliveira, et al. "Passive sensing of health outcomes through smartphones: systematic review of current solutions and possible limitations". In: *JMIR mHealth and uHealth* 7.8 (2019), e12649.

[11] Chris Beckmann, Sunny Consolvo, and Anthony LaMarca. "Some assembly required: Supporting end-user sensor installation in domestic ubiquitous computing environments". In: *International Conference on Ubiquitous Computing*. Springer. 2004, pp. 107–124.

[12] Roberto Minerva, Faraz Malik Awan, and Noel Crespi. "Exploiting digital twin as enablers for synthetic sensing". In: *IEEE Internet Computing* (2021).

[13] Michael Boyle, Christopher Edwards, and Saul Greenberg. "The effects of filtered video on awareness and privacy". In: *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 2000, pp. 1–10.

[14] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. "Techology trend of edge AI". In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE. 2018, pp. 1–2.

[15] Werner Kritzinger et al. "Digital Twin in manufacturing: A categorical literature review and classification". In: *IFAC-PapersOnLine* 51.11 (2018), pp. 1016–1022.

[16] Michael Grieves. "Digital twin: manufacturing excellence through virtual factory replication". In: *White paper* 1.2014 (2014), pp. 1–7.

[17] A Hughes. "Forging the Digital Twin in Discrete Manufacturing, A Vision for Unity in the Virtual and Real Worlds". In: *LNS Research e-book* (2018).

[18] BIS Research. *Digital twins: Bridging the physical and digital*. Jan. 1, 2020. URL: https://www.statista.com/statistics/1296187/global-digital-twin-market-by-industry/.

[19] Aaron Parrott, Brian Umbenhauer, and Lane Warshaw. *Digital twins: Bridging the physical and digital*. Jan. 15, 2020. URL: https://www2.deloitte.com/global/en/insights/focus/tech-trends/2020/digital-twin-applications-bridging-the-physical-and-digital.html.

[20] Dubravko Miljković. "Sample based synthesis of car engine noise". In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2020, pp. 1012–1017.

[21] Union of Concerned Scientist. *Cars, Trucks, Buses and Air Pollution*. July 19, 2018. URL: https://www.ucsusa.org/resources/cars-trucks-buses-and-air-pollution.

[22] Susanne Dirks and Mary Keeling. "A vision of smarter cities: How cities can lead the way into a prosperous and sustainable future". In: *IBM Institute for business Value* 8 (2009).

[23] United Nations. *The World's Cities in 2018*. Jan. 2020. URL: https://www.un.org/development/desa/pd/sites/www.un.org.development.desa.pd/files/files/documents/2020/Jan/un_2018_worldcities_databooklet.pdf.

[24] Timea Nochta et al. "A socio-technical perspective on urban analytics: The case of city-scale digital twins". In: *Journal of Urban Technology* 28.1-2 (2021), pp. 263–287.

[25] Hannu Penttilä, Marko Rajala, and Simo Freese. "Building information modelling of modern historic buildings". In: (2007).

[26] Siemens Global. *Building information modeling*. URL: https://new.siemens.com/global/en/products/buildings/digital-building-lifecycle/bim.html.

[27] Siemens Global. *Building Twin: The backbone of a smart building*. Apr. 6, 2020. URL: https://new.siemens.com/global/en/company/stories/infrastructure/2020/building-twin-backbone-smart-building.html.

[28] Siemens Global. *Building Twin — Digital building lifecycle*. URL: https://new.siemens.com/global/en/products/buildings/digital-building-lifecycle/building-twin.html.

[29] Aaron Costin et al. "Building Information Modeling (BIM) for transportation infrastructure–Literature review, applications, challenges, and recommendations". In: *Automation in construction* 94 (2018), pp. 257–281.

[30] Andrey Rudskoy, Igor Ilin, and Andrey Prokhorov. "Digital twins in the intelligent transport systems". In: *Transportation Research Procedia* 54 (2021), pp. 927–935.

[31] Sathish AP Kumar et al. "A novel digital twin-centric approach for driver intention prediction and traffic congestion avoidance". In: *Journal of Reliable Intelligent Environments* 4.4 (2018), pp. 199–209.

[32] Hakdaoui Sofia, Emran Anas, and Oumghar Faïz. "Mobile mapping, machine learning and digital twin for road infrastructure monitoring and maintenance: Case study of mohammed VI bridge in Morocco". In: *2020 IEEE International conference of Moroccan Geomatics (Morgeo)*. IEEE. 2020, pp. 1–6.

[33] Gerhard Schrotter and Christian Hürzeler. "The digital twin of the city of Zurich for urban planning". In: *PFG–Journal of Photogrammetry, Remote Sensing and Geoinformation Science* 88.1 (2020), pp. 99–112.

[34] Kristina Frank. *Digitaler Zwilling: Bessere Luft durch intelligente Mobilität*. Oct. 11, 2018. URL: https://ru.muenchen.de/2018/194/Digitaler-Zwilling-Bessere-Luft-durch-intelligente-Mobilitaet-80933.

[35] Kira-digi. *The Kalasatama Digital Twins Project*. May 2, 2019. URL: https://www.hel.fi/static/liitteet-2019/Kaupunginkanslia/Helsinki3D_Kalasatama_Digital_Twins.pdf.

[36] Business Geomatics. *Für IoT- und Smart City-Anwendungen: virtualcitySYSTEMS erstellt Digitalen Zwilling von Rotterdam*. Sept. 9, 2019. URL: https://www.business-geomatics.com/2019/09/09/fuer-iot-und-smart-city-anwendungen-virtualcitysystems-erstellt-digitalen-zwilling-von-rotterdam/.

[37] *3DEXPERIENCity Virtual Rennes*. 2019. URL: http://www.ecocites.logement.gouv.fr/IMG/pdf/ecocite_fiches_top_10_2019_rennes_metropole_pia_bat_en.pdf.

[38] Singapore Government. *Virtual Singapore*. URL: https://www.nrf.gov.sg/programmes/virtual-singapore.

[39] Google Coral. *Coral technology - Advanced neural network processing for low-power devices*. URL: `https://coral.ai/technology/#model-compatibility`.

[40] Dipa Soni and Ashwin Makwana. "A survey on mqtt: a protocol of internet of things (iot)". In: *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*. Vol. 20. 2017, pp. 173–177.

[41] Aldwin Akbar Hermanudin, Fransiskus Astha Ekadiyanto, and Riri Fitri Sari. "Performance Evaluation of CoAP Broker and Access Gateway Implementation on Wireless Sensor Network". In: *2018 IEEE Region Ten Symposium (Tensymp)*. IEEE. 2018, pp. 74–79.

[42] Partha Pratim Ray. "A survey on visual programming languages in internet of things". In: *Scientific Programming* 2017 (2017).

[43] J Paul Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010.

[44] Syeda Noor Zehra Naqvi, Sofia Yfantidou, and Esteban Zimányi. "Time series databases and influxdb". In: *Studienarbeit, Université Libre de Bruxelles* 12 (2017).

[45] Mainak Chakraborty and Ajit Pratap Kundan. "Grafana". In: *Monitoring Cloud-Native Applications*. Springer, 2021, pp. 187–240.

[46] Grafana. *Visualization panels*. URL: `https://grafana.com/docs/grafana/latest/visualizations/`.

[47] Google. *FlatBuffers*. URL: `https://google.github.io/flatbuffers/`.

[48] Yen-Lin Lee, Pei-Kuei Tsung, and Max Wu. "Techology trend of edge AI". In: *2018 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*. IEEE. 2018, pp. 1–2.

[49] *Build AI that works offline with Coral Dev Board, Edge TPU, and TensorFlow Lite*. Mar. 14, 2019. URL: `https://blog.tensorflow.org/2019/03/build-ai-that-works-offline-with-coral.html`.

[50] Tetsuya Yokotani and Yuya Sasaki. "Comparison with HTTP and MQTT on required network resources for IoT". In: *2016 international conference on control, electronics, renewable energy and communications (ICCEREC)*. IEEE. 2016, pp. 1–6.

[51] The Eclipse Foundation. *Eclipse Mosquitto™: An open source MQTT broker*. URL: `https://mosquitto.org/`.

[52] ArduinoHUB. *PubSubClient*. URL: `https://www.arduino.cc/reference/en/libraries/pubsubclient/`.

[53] The Eclipse Foundation. *Eclipse Paho*. URL: `https://www.eclipse.org/paho/`.

[54] MongoDB. *MongoDB — Build Faster. Build Smarter*. URL: `https://www.mongodb.com/`.

[55] *TFLite — TensorFlow Hub*. URL: `https://tfhub.dev/s?deployment-format=lite&module-type=image-object-detection`.

[56] *TensorFlow Lite Model Maker*. URL: https://www.tensorflow.org/lite/models/modify/model_maker.

[57] Tsung-Yi Lin et al. "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[58] Yunyang Xiong et al. "Mobiledets: Searching for object detection architectures for mobile accelerators". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3825–3834.

[59] *Mini Traffic Detection Dataset*. URL: https://www.kaggle.com/datasets/zoltanszekely/mini-traffic-detection-dataset.

[60] *Label Dataset*. URL: https://universe.roboflow.com/traffic/label-yeebg.

[61] Mingxing Tan, Ruoming Pang, and Quoc V Le. "Efficientdet: Scalable and efficient object detection". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2020, pp. 10781–10790.

[62] TensorFlow. *Object Detection with TensorFlow Lite Model Maker*. URL: https://www.tensorflow.org/lite/models/modify/model_maker/object_detection.

[63] Google Coral. *TensorFlow models on the Edge TPU*. URL: https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview.

[64] Open Mobile Alliance. *OMA LightWeight M2M (LWM2M) Object and Resource Registry*. 2014.

[65] Eclipse Ditto. *Thing*. URL: https://www.eclipse.org/ditto/basic-thing.html.

[66] Howard Butler et al. *The geojson format*. Tech. rep. 2016.

[67] Google Coral. *Edge TPU performance benchmarks*. URL: https://coral.ai/docs/edgetpu/benchmarks/.

[68] *11 Sustainable Cities and Communities*. URL: https://www.globalgoals.org/goals/11-sustainable-cities-and-communities/.